THE EXPERT'S VOICE® IN ORACLE

IOUG
independent oracle users group
share your experience

# Troubleshooting Oracle Performance

SECOND EDITION

METHODICALLY IDENTIFY AND
SOLVE PERFORMANCE PROBLEMS
INVOLVING THE ORACLE DATABASE ENGINE

Christian Antognini

Apress®

# Contents at a Glance

# Introduction

Oracle Database has become a huge piece of software. This not only means that a single human can no longer be proficient in using all the features provided in recent versions, it also means that some features will rarely be used. Actually, in most situations, it's enough to know and take advantage of a limited number of core features in order to use Oracle Database efficiently and successfully. This is precisely why this book covers only the features that, based on my experience, are necessary to troubleshoot most of the database-related performance problems you will encounter.

## The Structure of This Book

This book is divided into four parts:

> Part 1 covers some basics that are required to read the rest of the book. Chapter 1, "Performance Problems," explains not only why it's essential to approach performance problems at the right moment and in a methodological way, but also why understanding business needs and problems is essential. It also describes the most common database-related design problems that lead to suboptimal performance. Chapter 2, "Key Concepts," describes the operations carried out by the database engine when parsing and executing SQL statements and how to instrument application code and database calls. It also introduces some important terms that are frequently used in the book.
>
> Part 2 explains how to approach performance problems in an environment that uses Oracle Database. Chapter 3, "Analysis of Reproducible Problems," describes how to identify performance problems with the help of SQL trace and PL/SQL profilers. Chapter 4, "Real-time Analysis of Irreproducible Problems," describes how to take advantage of information provided by dynamic performance views. Several tools and techniques that can be used with them are also introduced. Chapter 5, "Postmortem Analysis of Irreproducible Problems," describes how to analyze performance problems that happened in the past with the help of Automatic Workload Repository and Statspack.
>
> Part 3 describes the component that is responsible for turning SQL statements into execution plans: the query optimizer. Chapter 6, "Introducing the Query Optimizer," provides an overview of what the query optimizer does and how it does it. Chapters 7 and 8, "System Statistics" and "Object Statistics," describe what system statistics and object statistics are, how to gather them, and why they are important for the query optimizer. Chapter 9, "Configuring the Query Optimizer," covers a configuration road map that you can use to find a good configuration for the query optimizer. Chapter 10, "Execution Plans," describes in detail how to obtain, interpret, and judge the efficiency of execution plans.

Part 4 shows which features are provided by Oracle Database to execute SQL statements efficiently. Chapter 11, "SQL Optimization Techniques," describes the techniques provided by Oracle Database to influence the execution plans that are generated by the query optimizer. Chapter 12, "Parsing," describes how to identify, solve, and work around performance problems caused by parsing. Chapter 13, "Optimizing Data Access," describes the methods available to access data and how to choose between them. Chapter 14, "Optimizing Joins," discusses how to join several sets of data together efficiently. Chapter 15, "Beyond Data Access and Join Optimization," describes advanced optimization techniques such as parallel processing, materialized views, and result caching. And Chapter 16, "Optimizing the Physical Design," explains why it's important to optimize the physical design of a database.

## Intended Audience

This book is intended for performance analysts, application developers, and database administrators who are involved in troubleshooting performance problems in applications using Oracle Database.

No specific knowledge in optimization is required. However, readers are expected to have a working knowledge of Oracle Database and to be proficient with SQL. Some sections of the book cover features that are specific to programming languages such as PL/SQL, Java, C#, PHP, and C. These features are covered only to provide a wide range of application developers with specific information about the programming language they're using. You can pick out the ones you're using or interested in and skip the others.

## Which Versions Are Covered?

The most important concepts covered in this book are independent of the version of Oracle Database you're using. It's inevitable, however, that when details about the implementation are discussed, some information is version-specific. This book explicitly discusses the versions currently available from Oracle Database 10g Release 2 to Oracle Database 12c Release 1. They are as follows:

- Oracle Database 10g Release 2, up to and including version 10.2.0.5.0
- Oracle Database 11g Release 1, up to and including version 11.1.0.7.0
- Oracle Database 11g Release 2, up to and including version 11.2.0.4.0
- Oracle Database 12c Release 1, version 12.1.0.1.0

Be aware that the granularity is the patch set level, and therefore, changes introduced by security and bundle patches aren't discussed. If the text doesn't explicitly mention that a feature is available for a specific version only, it's available for all those versions.

## Online Resources

You can download the files referenced through the book from `http://top.antognini.ch`. At the same URL, you will also find addenda and errata as soon as they are available. You can also send any type of feedback or questions about the book to `top@antognini.ch`.

## Differences between the First and the Second Editions

The main goals set for the revision of the book were the following:

- Add content about Oracle Database 11g Release 2 and Oracle Database 12c Release 1.
- Remove content about Oracle9i and Oracle Database 10g Release 1.
- Add content that was missing in the first edition (for example, features like hierarchical profiler, active session history, AWR, and Statspack).
- Add information about PHP in the parts that cover features that are specific to programming languages.
- Reorganize part of the material for better readability. For example, splitting the chapter about system and object statistics in two.
- Fix errata and generally enhance the text.

PART I

# Foundations

*Chi non fa e' fondamenti prima, gli potrebbe con una grande virtú farli poi, ancora che si faccino con disagio dello architettore e periculo dello edifizio.*

*He who has not first laid his foundations may be able with great ability to lay them afterwards, but they will be laid with trouble to the architect and danger to the building.*[1]

—Niccoló Machiavelli, *Il principe*. 1532.

[1]Translated by W. K. Marriott. Available at http://www.gutenberg.org/files/1232/1232-h/1232-h.htm.

CHAPTER 1

# Performance Problems

Too often, optimization begins when an application's development is already finished. This is unfortunate because it implies that performance is not as important as other crucial requirements of the application. Performance is not merely optional, though; it is a key property of an application. Not only does poor performance jeopardize the acceptance of an application, it usually leads to a lower return on investment because of lower productivity of those using it. In fact, as shown in several IBM studies from the early 1980s, there is a close relationship between performance and user productivity. The studies showed a one-to-one decrease in user think time and error rates as system transaction rates increased. This was attributed to a user's loss of attention because of longer wait times. In addition, poorly performing applications lead to higher costs for software, hardware, and maintenance. For these reasons, this chapter discusses why it is important to plan performance, which are the most common design mistakes that lead to sub-optimal performance, and how to know when an application is experiencing performance problems. Then, the chapter covers how to approach performance problems when they occur.

## Do You Need to Plan Performance?

In software engineering, different models are used to manage development projects. Whether the model used is a sequential life cycle like a waterfall model or an iterative life cycle like the one used with agile methodologies, an application goes through a number of common phases (see Figure 1-1). These phases may occur once (in the waterfall model) or several times (in the iterative model) in development projects.

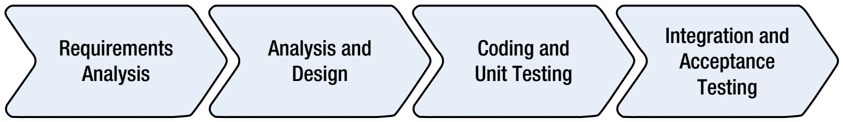


***Figure 1-1.*** *Essential phases in application development*

If you think carefully about the tasks to carry out for each of these phases, you may notice that performance is inherent to each of them. In spite of this, real development teams quite often forget about performance, at least until performance problems arise. At that point, it may be too late. Therefore, the following sections cover what you should not forget, from a performance point of view, the next time you are developing an application.

## Requirements Analysis

Simply put, a *requirements analysis* defines the aim of an application and therefore what it is expected to achieve. To do a requirements analysis, it is quite common to interview several stakeholders. This is necessary because it is unlikely that only one person can define all the business and technical requirements. Because requirements come from several sources, they must be carefully analyzed, especially to find out whether they potentially conflict. It is crucial when performing a requirements analysis to not only focus on the functionalities the application has to provide but also to carefully define their utilization. For each specific function, it is essential to know how many users[1] are expected to interact with it, how often they are expected to use it, and what the expected response time is for one usage. In other words, you must define the expected performance figures.

### RESPONSE TIME

The time interval between the moment a request enters a system or functional unit and the moment it leaves is called *response time.* The response time can be further broken down into the time needed by the system to process the request, which is called *service time*, and the time the request is waiting to be processed, which is called *wait time* (or *queueing delay* in queueing theory).

response time = service time + wait time

If you consider that a request enters a system when a user performs an action, such as clicking a button, and goes out of the system when the user receives an answer in response to the action, you can call that interval *user response time.* In other words, the user response time is the time required to process a request from the user's perspective.

In some situations, like for web applications, considering user response time is uncommon because it is usually not possible to track the requests before they hit the first component of the application (typically a web server). In addition, most of the time guaranteeing a user response time is not possible because the provider of the application is not responsible for the network between the user's application, typically a browser, and the first component of the application. In such situations, it is more sensible to measure and guarantee the interval between the entry of requests into the first component of the system and when they exit. This elapsed time is called *system response time.*

Table 1-1 shows an example of the expected performance figures for the actions provided by JPetStore.[2] For each action, the guaranteed system response times for 90% and 99.99% of the requests entering the system are given. Most of the time, guaranteeing performance for all requests (in other words, 100%) is either not possible or too expensive. It is quite common, therefore, to define that a small number of requests may not achieve the requested response time. Because the workload on the system changes during the day, two values are specified for the maximum arrival rate. In this specific case, the highest transaction rate is expected during the day, but in other situations—for example, when batch jobs are scheduled for nights—it could be different.

[1]Note that a user is not always a human being. For example, if you are defining requirements for a web service, it is likely that only other applications will use it.

[2]JPetStore is a sample application provided, among others, by the Spring Framework. See www.springframework.org to download it or to simply get additional information.

***Table 1-1.*** *Performance Figures for Typical Actions Provided by a Web Shop*

| | **Max. Response Time (s)** | | **Max. Arrival Rate (trx/min)** | |
|---|---|---|---|---|
| **Action** | **90%** | **99.99%** | **0-7** | **8-23** |
| Register/change profile | 2 | 5 | 1 | 2 |
| Sign in/sign out | 0.5 | 1 | 5 | 20 |
| Search products | 1 | 2 | 60 | 240 |
| Display product overview | 1 | 2 | 30 | 120 |
| Display product details | 1.5 | 3 | 10 | 36 |
| Add/update/remove product in/from cart | 1 | 2 | 4 | 12 |
| Show cart | 1 | 3 | 8 | 32 |
| Submit/confirm order | 1 | 2 | 2 | 8 |
| Show orders | 2 | 5 | 4 | 16 |

These performance requirements are not only essential throughout the next phases of application development (as you will see in the following sections), but later you can also use them as the basis for defining service level agreements and for capacity-planning purposes.

## SERVICE LEVEL AGREEMENTS

A *service level agreement* (SLA) is a contract defining a clear relationship between a service provider and a service consumer. It describes, among others things, the provided service, its level of availability regarding uptime and downtime, the response time, the level of customer support, and what happens if the provider is not able to fulfill the agreement.

Defining service level agreements with regard to response time makes sense only if it is possible to verify their fulfillment. They require the definition of clear and measurable performance figures and their associated targets. These performance figures are commonly called *key performance indicators* (KPI). Ideally a monitoring tool is used to gather, store, and evaluate them. In fact, the idea is not only to flag when a target is not fulfilled but also to keep a log for reporting and capacity-planning purposes. To gather these performance figures, you can use two main techniques. The first takes advantage of the output of instrumentation code (see Chapter 2 for more information). The second one is to use a response-time monitoring tool (see the section "Response-Time Monitoring" later in this chapter).

## Analysis and Design

Based on the requirements, the architects are able to design a solution. At the beginning, for the purpose of defining the architecture, considering all requirements is essential. In fact, an application that has to handle a high workload must be designed from the beginning to achieve this requirement. This is especially the case if techniques such as parallelization, distributed computing, or reutilization of results are implemented. For example, designing a client/server application aimed at supporting a few users performing a dozen transactions per minute is quite different from designing a distributed application aimed at supporting thousands of users performing hundreds of transactions per second.

Sometimes requirements also impact the architecture by imposing limits on the utilization of a specific resource. For example, the architecture of an application to be used by mobile devices connected to the server through a slow network must absolutely be conceived to support a long latency and a low throughput. As a general rule, the architects have to foresee not only where the bottlenecks of a solution might be, but also whether these bottlenecks might jeopardize the fulfillment of the requirements. If the architects do not possess enough information to perform such a critical estimation a priori, one or even several prototypes should be developed. In this respect, without the performance figures gathered in the previous phase, making sensible decisions is difficult. By sensible decisions, I mean those leading to an architecture/design that supports the expected workload with a minimal investment—simple solutions for simple problems, elegant solutions for complex problems.

## Coding and Unit Testing

A professional developer should write code that has the following characteristics:

**Robustness:** The ability to cope with unexpected situations is a characteristic any software should have. To achieve the expected quality, performing unit testing on a regular basis is essential. This is even more important if you choose an iterative life cycle. In fact, the ability to quickly refactor existing code is essential in such models. For example, when a routine is called with a parameter value that is not part of a specific domain, it must nevertheless be able to handle it without crashing. If necessary, a meaningful error message should be generated as well.

**Maintainability:** Long-term, well-structured, readable, and documented code is much simpler (and cheaper) to maintain than code that is poorly written and not documented. For example, a developer who packs several operations in a single line of cryptic code has chosen the wrong way to demonstrate his intelligence.

**Speed:** Code should be optimized to run as fast as possible, especially if a high workload is expected. It should be scalable, and therefore able to leverage additional hardware resources to support an increasing number of users or transactions. For example, unnecessary operations, serialization points, as well as inefficient or unsuitable algorithms, should be avoided. It is essential, however, to not fall into the *premature optimization* trap.

**Shrewd resource utilization:** The code should make the best possible use of the available resources. Note that this does not always mean using the fewest resources. For example, an application using parallelization requires many more resources than one where all operations are serialized, but in some situations parallelization may be the only way to handle demanding workloads.

**Security:** The ability to ensure data confidentiality and integrity, as well as user authentication and authorization, is undisputed. Sometimes non-repudiation is also an issue. For example, digital signatures might be required to prevent end-users from successfully challenging the validity of a communication or contract.

**Instrumented:** The aim of instrumentation is twofold. First, it allows for the easier analysis of both functional and performance problems when they arise—and they will arise to be sure, even for the most carefully designed system. Second, instrumentation is the right place to add strategic code that will provide information about an application's performance. For example, it is usually quite simple to add code that provides information about the time taken to perform a specific operation. This is a simple yet effective way to verify whether the application is capable of fulfilling the necessary performance requirements.

Not only do some of these characteristics conflict with each other, but budgets are usually limited (and sometimes are *very* limited). It seems reasonable then that more often than not it is necessary to prioritize these characteristics and find a good balance of achieving the desired requirements within the available budget.

**PREMATURE OPTIMIZATION**

Premature optimization, (probably) because of Donald Knuth's famous line "premature optimization is the root of all evil," is, at the very least, a controversial topic. The misconception based on that particular quote is that a programmer, while writing code, should ignore optimization altogether. In my opinion this is wrong. To put the quote in context, let's have a look at the text that precedes and follows it:

*"There is no doubt that the grail of efficiency leads to abuse. Programmers waste enormous amounts of time thinking about, or worrying about, the speed of noncritical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered. We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3%. A good programmer will not be lulled into complacency by such reasoning, he will be wise to look carefully at the critical code; but only after that code has been identified. It is often a mistake to make a priori judgments about what parts of a program are really critical, since the universal experience of programmers who have been using measurement tools has been that their intuitive guesses fail."*

My take on Knuth's paper is that programmers, when writing code, should not care about micro optimization that has local impact only. Instead, they should care about optimizations that have global impact, like the design of a system, the algorithms used to implement the required functionality, or in which layer (SQL, PL/SQL, application language) and with which features a specific processing should be performed. Local optimizations are deferred till a measurement tool points out that a specific part of the code is spending too much time executing. And because the optimization is local, there is no impact on the overall design of the system.

## Integration and Acceptance Testing

The purpose of integration and acceptance testing is to verify functional and performance requirements as well as the stability of an application. It can never be stressed enough that performance tests have the same importance as function tests. For all intents and purposes, an application experiencing poor performance is no worse than an application failing to fulfill its functional requirements. In both situations, the application is useless. Still, it is possible to verify the performance requirements only once they have been clearly defined.

The lack of formal performance requirements leads to two major problems. First, the chances are quite high that no serious and methodical stress tests will be performed during integration and acceptance testing. The application will then go to production without knowing whether it will support the expected workload. Second, it will not always be obvious to determine what is acceptable and what is not in terms of performance. Usually only the extreme cases (in other words, when the performance is very good or very poor) are judged in the same way by different people. And if an agreement is not found, long, bothersome, and unproductive meetings and interpersonal conflicts follow.

In practice, designing, implementing, and performing good integration and acceptance testing to validate the performance of an application are not trivial tasks. You have to deal with three major challenges to be successful:

- Stress tests should be designed to generate a representative workload. To do so, two main approaches exist. The first is to get real users to do real work. The second is to use a tool that simulates the users. Both approaches have pros and cons, and their use should be evaluated on a case-by-case basis. In some situations, both can be used to stress different parts of the application or in a complementary way.

- To generate a representative workload, representative test data is needed. Not only should the number of rows and the size of the rows match the expected quantity, but also the data distribution and the content should match real data. For example, if an attribute should contain the name of a city, it is much better to use real city names than to use character strings like *Aaaacccc* or *Abcdefghij*. This is important because in both the application and the database there are certainly many situations where different data could lead to different behavior (for example, with indexes or when a hash function is applied to data).
- The test infrastructure should be as close as possible to, and ideally the same as, the production infrastructure. This is especially difficult for both highly distributed systems and systems that cooperate with a large number of other systems.

In a sequential life cycle model, the integration and acceptance testing phase occurs close to the end of the project, which might be a problem if a major flaw in the architecture leading to performance problems is detected too late. To avoid such a problem, stress tests should be performed during the coding and unit testing phases as well. Note that an iterative life cycle model does not have this problem. In fact, by the very definition of "iterative life cycle model," a stress test should be performed for every iteration.

## Designing for Performance

Given that applications should be designed for performance, it would be useful to cover an approach to doing that in great detail. However, the focus of this book is on troubleshooting. For this reason, I limit myself to briefly describing the top ten most common database-related design problems that frequently lead to suboptimal performance.

### Lack of Logical Database Design

Once upon a time, it was considered obvious that one should have a data architect involved in every development project. Often this person was not only responsible for the data and the database design, but was also part of the team in charge of the whole architecture and design of the application. Such a person often had extensive experience with databases. He knew exactly how to design them to guarantee data integrity as well as performance.

Today, unfortunately, it is not always so. Too often I see projects in which no formal database design is done. The application developers do the client and/or middle-tier design. Then, suddenly, the database design is generated by a tool such as a persistence framework. In such projects, the database is seen as a dumb device that stores data. Such a viewpoint of the database is a mistake.

### Implementing Generic Tables

Every CIO dreams of applications that are easily able to cope with new or changed requirements. The keyword is *flexibility*. Such dreams sometimes materialize in the form of applications that use generic database designs. Adding new data is just a matter of changing the configuration without changing the database objects themselves.

Two main database designs are used to achieve such flexibility:

**Entity-attribute-value (EAV) models:** As their name implies, to describe every piece of information, at least three columns are used: entity, attribute, and value. Each combination defines the value of a specific attribute associated to a specific entity.

**XML-based designs:** Each table has just a few columns. Two columns are always present: an identifier and an XML column to store almost everything else. Sometimes a few other columns are also available for storing metadata information (for example, who did the last modification and when).

The problem with such designs is that they are (to say the least) suboptimal from a performance point of view. In fact, flexibility is tied to performance. When one is at its maximum, the other is at its minimum. In some situations suboptimal performance might be good enough. But in other situations it might be catastrophic. Hence, you should use a flexible design only when the required performance can be achieved with it.

## Not Using Constraints to Enforce Data Integrity

Constraints (primary keys, unique keys, foreign keys, `NOT NULL` constraints, and check constraints) are not only fundamental to guarantee data integrity, but they are also extensively used by the query optimizer during the generation of execution plans. Without constraints, the query optimizer is not able to take advantage of a number of optimizations techniques. In addition, checking the constraints at the application level leads to more code being written and tested as well as to potential problems with data integrity, because data can always be manually modified at the database level. Also, checking constraints at application level usually requires greater consumption of resources, and leads to less scalable locking schemes (such as locking an entire table instead of letting the database lock only a subset of rows). Therefore, I strongly advise application developers to define all known constraints at the database level.

## Lack of Physical Database Design

It is not uncommon to see projects where the logical design is directly mapped to the physical design without taking advantage of all the features provided by Oracle Database. The most common and obvious example is that every relation is directly mapped to a heap table. From a performance point of view, such an approach is suboptimal. In more than a few situations, index-organized tables (IOT), indexed clusters, or hash clusters might lead to better performance.

Oracle Database provides much more than just the usual b-tree and bitmap indexes. Depending on the situation, compressed indexes, reverse-key indexes, function-based indexes, linguistic indexes, or text indexes might be very valuable to improving performance.

For large databases, the implementation of the partitioning option is critical. Most DBAs recognize the option and its usefulness. A common problem in this area is that developers think that partitioning tables has no impact on the physical database design. Sometimes this is true, but sometimes this is not the case. As a result I strongly recommend planning the use of partitioning from the beginning of a project.

Another common issue to deal with during the development of a new application is the definition, and implementation, of a sound data-archiving concept. Postponing it is usually not an option, because it might impact the physical database design (if not the logical database design).

## Not Choosing the Right Data Type

In recent years, I have witnessed a disturbing trend in physical database design. This trend may be called *wrong datatype selection* (such as storing dates in `VARCHAR2` instead of using `DATE` or `TIMESTAMP`). At first glance, choosing the datatype seems to be a very straightforward decision to make. Nevertheless, do not underestimate the number of systems that are now running and suffering because wrong datatypes were selected.

There are four main problems related to wrong datatype selection:

**Wrong or lacking validation of data:** The database engine must be able to validate data that is stored in the database. For example, you should avoid storing numeric values in character datatypes. Doing so calls for an external validation, which leads to problems similar to those described in the section "Not Using Constraints to Enforce Data Integrity."

**Loss of information:** During the conversion of the original (correct) datatype to the (wrong) database datatype, information gets lost. For example, let's imagine what happens when the date and time of an event is stored with a `DATE` datatype instead of with a `TIMESTAMP WITH TIME ZONE` datatype. Fractional seconds and time zone information get lost.

**Things do not work as expected:** Operations and features for which the order of data is important might result in unexpected results, because of the specific comparison semantics associated to every datatype. Typical examples are issues related to range partitioned tables and `ORDER BY` clauses.

**Query optimizer anomalies:** The query optimizer might generate wrong estimates and, consequently, might choose suboptimal execution plans because of wrong datatype selection. This is not the fault of the query optimizer. The problem is that the query optimizer cannot do its job because information is hidden from it.

In summary, there are plenty of good reasons for selecting datatypes correctly. Doing so will likely help you to avoid many problems.

## Not Using Bind Variables Correctly

From a performance point of view, bind variables introduce both an advantage and a disadvantage. The advantage of bind variables is that they allow the sharing of cursors in the library cache, and in doing so they avoid hard parses and the associated overhead. The disadvantage of using bind variables in `WHERE` clauses, and only in `WHERE` clauses, is that crucial information is sometimes hidden from the query optimizer. For the query optimizer, to generate an optimal execution plan for every SQL statement, having literals instead of bind variables is in fact much better. Chapter 2 discusses this topic in detail.

From a security point of view, bind variables prevent the risks associated with SQL injection. In fact, it is not possible to change the syntax of a SQL statement by passing a value through a bind variable.

## Not Using Advanced Database Features

Oracle Database is a high-end database engine that provides many advanced features that can drastically reduce development costs, not to mention debugging and bug-fixing costs, while boosting performance. Leverage your investment by taking advantage of those features as much as possible. Especially avoid rewriting already available features (for example, do not create your own queuing system, because one is provided for you). That said, special care should be taken the first time a specific feature is used, especially if that feature was introduced in the very same database version you are running. You should not only carefully test such a feature to know whether it fulfills the requirements, but also verify its stability.

The most common argument against advanced database features is that applications using them are closely coupled to your current database brand and cannot be easily ported to another. This is true. However, most companies will rarely change the database engine under a specific application anyway. Companies are more likely to change the whole application before changing just the engine.

I recommend database-independent application design only when there are very good reasons for doing it. And if for some reason doing database-independent design is necessary, go back and reread the discussion about flexibility versus performance in the section "Implementing Generic Tables." That discussion applies in this case as well.

### Not Using PL/SQL for Data-Centric Processing

A special case, from the point raised in the previous section, is the use of PL/SQL for implementing batches that process lots of data. The most common example is an extract-transform-load (ETL) process. When, in such a process, the extract and load phases are executed against the very same database, from a performance point of view it is almost insane to not process the transform phase by taking advantage of the SQL and PL/SQL engines provided by the database engine that manages the source and target data. Unfortunately, the architecture of several mainstream ETL tools leads exactly to such insane behavior. In other words, data is extracted from the database (and frequently also moved to another server), the transform phase is executed, and then the resulting data is loaded back into the very same database from which it came. For this reason, vendors like Oracle started offering tools that perform transformations inside the database. Such tools, to differentiate them from the ETL tools, are commonly called ELT. For best performance, I advise performing data-centric processing as closely as possible to the data.

### Performing Unnecessary Commits

Commits are operations that call for serialization (the reason is simple: there is a single process (LGWR) that is responsible for writing data to redolog files). It goes without saying that every operation that leads to serialization inhibits scalability. And as a result, serialization is unwanted and should be minimized as much as possible. One approach is to put several unrelated transactions together. The typical example is a batch job that loads many rows. Instead of committing after every insert, it is much better to commit the inserted data in batches.

### Steadily Opening and Closing Database Connections

Opening a database connection that in turn starts an associated dedicated process on the database server, is not a lightweight operation. Do not underestimate the amount of time and resources required. A worst-case scenario that I sometimes observe is a web application that opens and closes a database connection for every request that involves a database access. Such an approach is highly suboptimal. Using a pool of connections in such a situation is of paramount importance. By using a connection pool, you avoid the constant starting and stopping of dedicated services processes, thus avoiding all the overhead involved.

## Do You Have Performance Problems?

There is probably a good chance that sooner or later the performance of an application will be questioned. If, as described in the previous sections, you have carefully defined the performance requirements, it should be quite simple to determine whether the application in question is in fact experiencing performance problems. If you have not carefully defined them, the response will largely depend on who answers the question.

Interestingly enough, in practice the most common scenarios leading to questions regarding the performance of an application fall into very few categories. They are short-listed here:

- Users are unsatisfied with the current performance of the application.
- A system-monitoring tool alerts you that a component of the infrastructure is experiencing timeouts or an unusual load.
- A response-time monitoring tool informs you that a service level agreement is not being fulfilled.

The difference between the second point and the third point is particularly important. For this reason, in the next two sections I briefly describe these monitoring solutions. After that, I present some situations where optimization appears to be necessary but in fact is not necessary at all.

## System Monitoring

System-monitoring tools perform health checks based on general system statistics. Their purpose is to recognize irregular load patterns that pop up as well as failures. Even though these tools can monitor the whole infrastructure at once, it is important to emphasize that they monitor only individual components (for example, hosts, application servers, databases, or storage subsystems) without considering the interplay between them. As a result, it is difficult, and for complex infrastructures virtually impossible, to determine the impact on the system response time when a single component of the infrastructure supporting it experiences an anomaly. An example of this is the high usage of a particular resource. In other words, an alert coming from a system-monitoring tool is just a warning that something could be wrong with the application or the infrastructure, but the users may not experience any performance problems at all (called a *false positive*). In contrast, there may be situations where users are experiencing performance problems, but the system-monitoring tool does not recognize them (called a *false negative*). The most common and simplest cases of false positive and false negative are seen while monitoring the CPU load of SMP systems with a lot of CPUs. Let's say you have a system with four quad-core CPUs. Whenever you see a utilization of about 75%, you may think that it is too high; the system is CPU-bounded. However, this load could be very healthy if the number of running tasks is much greater than the number of cores. This is a false positive. Conversely, whenever you see a utilization of about 8% of the CPU, you may think that everything is fine. But if the system is running a single task that is not parallelized, it is possible that the bottleneck for this task is the CPU. In fact, 1/16th of 100% is only 6.25%, and therefore, a single task cannot burn more than 6.25% of the available CPU. This is a false negative.

## Response-Time Monitoring

Response-time monitoring tools (also known as *application-monitoring tools*) perform health checks based on either synthetic transactions that are processed by *robots*, or on real transactions that are processed by end-users. The tools measure the time taken by an application to process key transactions, and if the time exceeds an expected threshold value, they raise an alert. In other words, they exploit the infrastructure as users do, and they complain about poor performance as users do. Because they probe the application from a user perspective, they are able to not only check single components but, more importantly, check the whole application's infrastructure as well. For this reason, they are devoted to monitoring service level agreements.

## Compulsive Tuning Disorder

Once upon a time, most database administrators suffered from a disease called *compulsive tuning disorder*.[3] The signs of this illness were the excessive checking of many performance-related statistics, most of them ratio-based, and the inability to focus on what was really important. They simply thought that by applying some "simple" rules, it was possible to tune their databases. History teaches us that results were not always as good as expected. Why was this the case? Well, all the rules used to check whether a given ratio (or value) was acceptable were defined independently of the user experience. In other words, false negatives or positives were the rule and not the exception. Even worse, an enormous amount of time was spent on these tasks.

For example, from time to time a database administrator will ask me a question like "On one of our databases I noticed that we have a large amount of waits on latch X. What can I do to reduce or, even better, get rid of such waits?" My typical answer is "Do your users complain because they are waiting on this specific latch? Of course not. So, do not worry about it. Instead, ask them what problems they are facing with the application. Then, by analyzing those problems, you will find out whether the waits on latch X are related to them or not." I elaborate on this in the next section.

Even though I have never worked as a database administrator, I must admit I suffered from compulsive tuning disorder as well. Today, I have, like most other people, gotten over this disease. Unfortunately, as with any bad illness, it takes a very long time to completely vanish. Some people are simply not aware of being infected. Others are aware, but after many years of addiction, it is always difficult to recognize such a big mistake and break the habit.

[3]This wonderful term was first coined by Gaya Krishna Vaidyanatha. You can find a discussion about it in the book *Oracle Insights: Tales of the Oak Table* (Apress, 2004).

## How Do You Approach Performance Problems?

Simply put, the aim of an application is to provide a benefit to the business using it. Consequently, the reason for optimizing the performance of an application is to maximize that benefit. This does not mean maximizing the performance, but rather finding the best balance between costs and performance. In fact, the effort involved in an optimization task should always be compensated by the benefit you can expect from it. This means that from a business perspective, performance optimization may not always make sense.

### Business Perspective vs. System Perspective

You optimize the performance of an application to provide a benefit to a business, so when approaching performance problems, you have to understand the business problems and requirements before diving into the details of the application. Figure 1-2 illustrates the typical difference between a person with a *business perspective* (that is, a user) and a person with a *system perspective* (that is, an engineer).

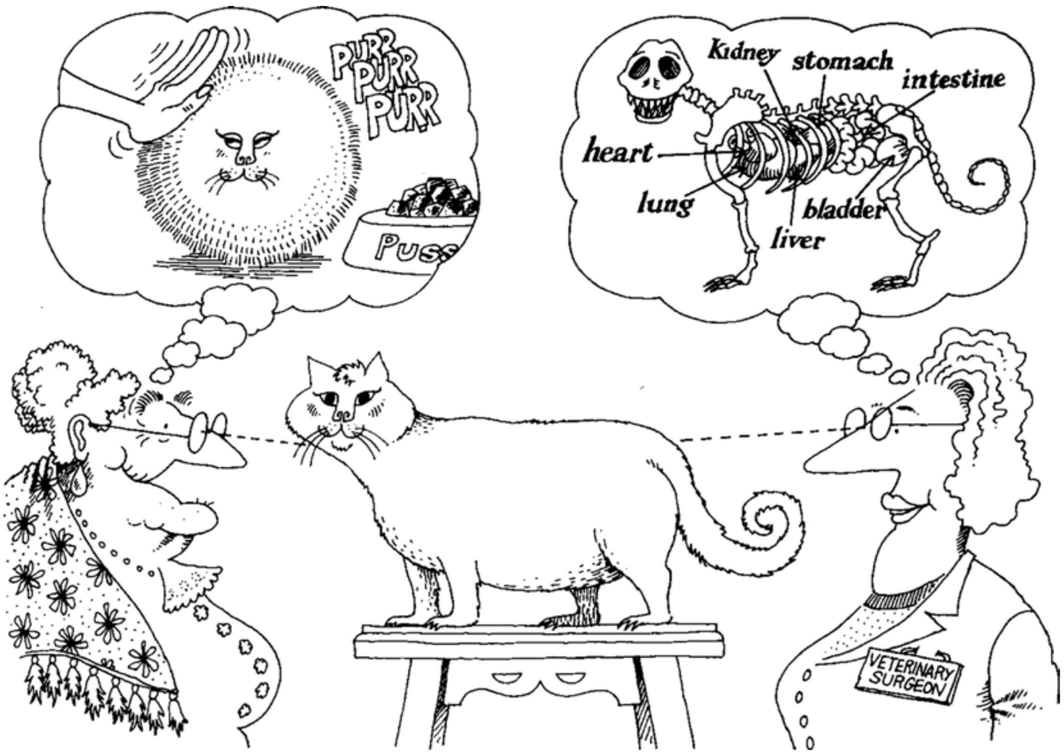


***Figure 1-2.*** *Different observers may have completely different perspectives*[4]

It is important to recognize that there is a cause-effect relationship between these two perspectives. Although the effects must be recognized from the business perspective, the causes must be identified from the system perspective. So if you do not want to troubleshoot nonexistent or irrelevant problems (compulsive tuning disorder), it is essential to understand what the problems are from a business perspective—even if subtler work is required.

[4]Booch, Grady, *Object-Oriented Analysis and Design with Applications*, page 42 (Addison Wesley Longman, Inc., 1994). Reproduced with the permission of Grady Booch. All rights reserved.

## Cataloging the Problems

The first steps to take when dealing with performance problems are to identify them from a business perspective and to set a priority and a target for each of them, as illustrated in Figure 1-3.

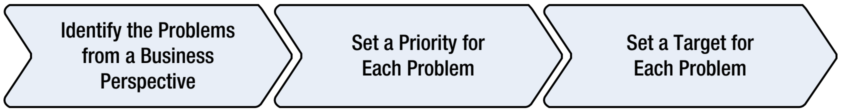

Identify the Problems from a Business Perspective
Set a Priority for Each Problem
Set a Target for Each Problem

***Figure 1-3.*** *Tasks to carry out while cataloging performance problems*

Business problems cannot be found by looking at system statistics. They have to be identified from a business perspective. If a monitoring of service level agreements is in place, the performance problems are obviously identified by looking at the operations not fulfilling expectations. Otherwise, there is no other possibility but to speak with the users or those who are responsible for the application. Such discussions can lead to a list of operations, such as registering a new user, running a report, or loading a bunch of data that is considered slow.

---

■ **Caution** It is not always necessary to identify problems from a business perspective. Sometimes they are already known. For example, the identification is required when somebody tells you something like “The end-users are constantly complaining about the performance; find out what the causes are.” But additional identification is not needed when your customer tells you that “Running the report XY takes way too long.” In the latter case, you already know what part of the application you have to look at. In the former, you have no clue; any part of the application might be involved.

---

Once you have identified the problematic operations, it is time to give them a priority. For that, ask questions like “If we can work on only five problems, which should be handled?” Of course, the idea is to solve them all, but sometimes the time or budget is limited. In addition, it is not possible to leave out cases where the measures needed to fix different problems conflict with each other. It is important to stress that to set priorities, the current performance could be irrelevant. For example, if you are dealing with a set of reports, it is not always the slowest one that has the highest priority. Possibly the fastest one is also the one that is executed more frequently, or the one the business (or simply the CEO) cares about most. It might therefore have the highest priority and should be optimized first. Once more, business requirements are driving you.

For each problem, you should set a quantifiable target for the optimization, such as “When the Create User button is clicked, the processing lasts at most two seconds.” If the performance requirements or even service level agreements are available, it is possible that the targets are already known. Otherwise, once again, you must consider the business requirements to determine the targets. Note that without targets you do not know when it is time to stop investigating for a better solution. In other words, the optimization could be endless. Remember, the effort should always be balanced by the benefit.

## Working the Problems

Troubleshooting several problems at the same time is much more complex than troubleshooting a single problem. Therefore, whenever possible, you should work one problem at a time. Simply take the list of problems and go through them according to their priority level.

For each problem, the three questions shown in Figure 1-4 must be answered:

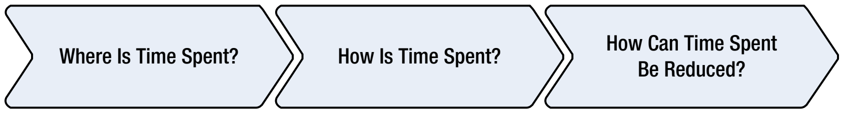


***Figure 1-4.** To troubleshoot a performance problem, you need to answer these three questions*

> **Where is time spent?** First, you have to identify where time goes. For example, if a specific operation takes ten seconds, you have to find out which module or component most of these ten seconds are used up in.
>
> **How is time spent?** Once you know where the time goes, you have to find out how that time is spent. For example, you may find out that the component spends 4.2 seconds on CPU, 0.4 seconds doing disk I/O operations, and 5.1 seconds waiting for dequeuing a message coming from another component.
>
> **How can time spent be reduced?** Finally, it is time to find out how the operation can be made faster. To do so, it is essential to focus on the most time-consuming part of the processing. For example, if disk I/O operations take 4% of the overall processing time, it makes no sense to start optimizing them, even if they are very slow.

To find out where and how the time is spent, the analysis should start by collecting end-to-end performance data about the execution of the operation you are concerned with. This is essential because multitier architectures are currently the *de facto* standard in software development for applications needing a database like Oracle. In the simplest cases, at least two tiers (a.k.a. client/server) are implemented. Most of the time, there are three: presentation, logic, and data. Figure 1-5 shows a typical infrastructure used to deploy a web application. Frequently, for security or workload-management purposes, components are spread over multiple machines as well.

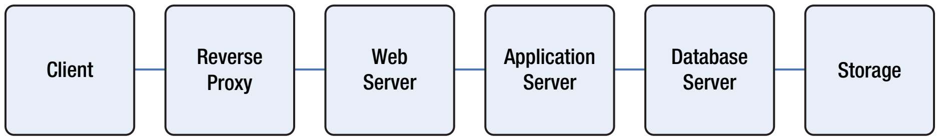


***Figure 1-5.** A typical web application consists of several components deployed on multiple systems*

To be processed with a multitier infrastructure, requests may go through several components. However, not in all situations are all components involved in the processing of a specific request. For example, if caching at the web server level has been activated, a request may be served from the web server without being forwarded to the application server. Of course, the same also applies to an application server or a database server.

Ideally, to fully analyze a performance problem, you should collect detailed information about all components involved in the processing. In some situations, especially when many components are involved, it may be necessary to collect huge amounts of data, which may require significant amounts of time for analysis. For this reason, a *divide-and-conquer* approach is usually the only efficient[5] way to approach a problem. The idea is to start the analysis by breaking up the end-to-end response time into its major components (see Figure 1-6 for an example) and then to gather detailed information only when it makes sense. In other words, you should collect the minimum amount of data necessary to identify the performance problem.

[5]While working on a performance problem, not only do you have to optimize the application you are analyzing, but you have to optimize your actions as well. In other words, you should identify and fix the problems as quickly as possible.

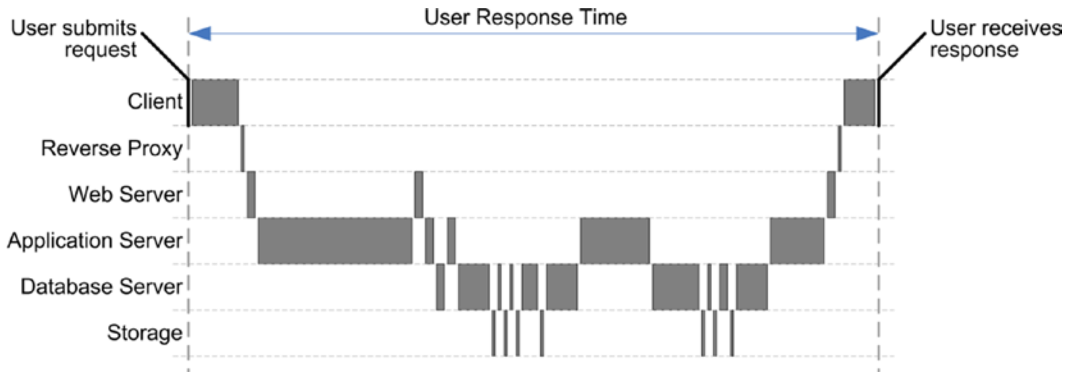


***Figure 1-6.*** *The response time of a request broken up into all major components. Communication delays between components are omitted*

Once you know which components are involved and how much time is spent by each of them, you can further analyze the problem by selectively gathering additional information only for the most time-consuming components. For example, according to Figure 1-6, you should worry only about the application server and the database server. Fully analyzing components that are responsible for only a very small portion of the response time is pointless.

Depending on what tools or techniques you use to gather performance data, in many situations you will not be able to fully break up the response time for every component as shown in Figure 1-6. In addition, this is usually not necessary. In fact, even a partial analysis, as shown in Figure 1-7, is useful in order to identify which components may, or may not, be mainly responsible for the response time.

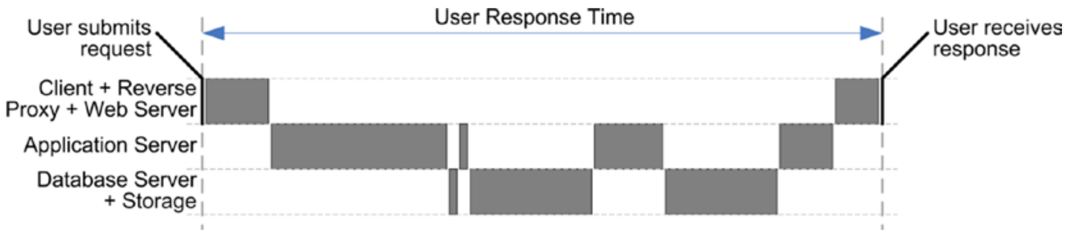


***Figure 1-7.*** *The response time of a request partially broken up by components*

To gather performance data about problems, basically only the following two methods are available:

> **Instrumentation:** When an application has been properly developed, it is instrumented to provide, among other things, performance figures. In normal circumstances, the instrumentation code is deactivated, or its output is kept to a minimum to spare resources. At runtime, however, it should be possible to activate or increase the amount of information it provides. An example of good instrumentation is Oracle's SQL trace (more about that in Chapter 3). By default it is deactivated, but when activated, it delivers trace files containing detailed information about the execution of SQL statements.

**Profiling analysis:** A profiler is a performance-analysis tool that, for a running application, records the executed operations, the time it takes to perform them, and the utilization of system resources (for example, CPU and memory). Some profilers gather data at the call level, others at the line level. The performance data is gathered either by sampling the application state at specified intervals or by automatically instrumenting the code or the executable. Although the overhead associated with the former is much smaller, the data gathered with the latter is much more accurate.

Generally speaking, both methods are needed to investigate performance problems. However, if good instrumentation is available, profiling analysis is less frequently used. Table 1-2 summarizes the pros and cons of these two techniques.

***Table 1-2.*** *Pros and Cons of Instrumentation and Profiling Analysis*

| Technique | Pros | Cons |
|---|---|---|
| Instrumentation | Possible to add timing information to key business operations. When available, can be dynamically activated without deploying new code. Context information (for example, about the user or the session) can be made available. | Must be manually implemented. Covers single components only; no end-to-end view of response time. Usually, the format of the output depends on the developer who wrote the instrumentation code. |
| Profiling analysis | Always-available coverage of the whole application. Multitier profilers provide end-to-end view of the response time. | May be expensive, especially for multitier profilers. Cannot always be (quickly) deployed in production. Overhead associated with profilers working at the line level may be very high. |

It goes without saying that you can take advantage of instrumentation only when it is available. Unfortunately, in some situations and all too often in practice, profiling analysis is often the only option available.

When you take steps to solve a particular problem, it is important to note that thanks to beneficial side effects, other problems might sometimes also be fixed (for example, reducing CPU use might benefit other CPU-intensive operations, and make them perform acceptably). Of course, the opposite can happen as well. Measures taken may introduce new problems. It is essential therefore to carefully consider all the possible side effects that a specific fix may have. Also, the inherent risks of introducing a fix should be cautiously evaluated. Clearly, all changes have to be carefully tested before implementing them in production.

Note that problems are not necessarly solved in production according to their priority. Some measures might take much longer to be implemented. For example, the change for a high-priority problem could require downtime or an application modification. As a result, although some measures might be implemented straight away, others might take weeks if not months or longer to be implemented.

## On to Chapter 2

This chapter describes key issues of dealing with performance problems: why it is essential to approach performance problems at the right moment and in a methodological way, why understanding business needs and problems is absolutely important, and why it is necessary to agree on what *good performance* means.

Before describing how to answer the three questions in Figure 1-4, I need to introduce some key concepts that I reference in the rest of the book. For that purpose, Chapter 2 describes the processing performed by the database engine to execute SQL statements. In addition, I provide some information on instrumentation and define several frequently used terms.

CHAPTER 2

# Key Concepts

The aim of this chapter is threefold. First, to avoid unnecessary confusion, I introduce some terms that are used repeatedly throughout this book. The most important include *selectivity* and *cardinality*, *cursor*, *soft* and *hard parses*, *bind variable peeking* , and *adaptive cursor sharing*. Second, I describe the life cycle of SQL statements. In other words, I describe the operations carried out in order to execute SQL statements. During this discussion, special attention is given to parsing. And third, I describe how to instrument application code and database calls.

## Selectivity and Cardinality

The *selectivity* is a value between 0 and 1 representing the fraction of rows filtered by an operation. For example, if an access operation reads 120 rows from a table and, after applying a filter, returns 18 of them, the selectivity is 0.15 (18/120). The selectivity can also be expressed as a percentage, so 0.15 can also be expressed as 15 percent. When the selectivity is close to 0, it's said to be *strong*. When it's close to 1, it's said to be *weak*.

**Caution** I used to use the terms *low/high* or *good/bad* instead of *strong/weak*. I stopped using *low/high* because they don't make it clear whether they refer to the degree of selectivity or to its numerical value. In fact, various and conflicting definitions exist. I stopped using *good/bad* because it isn't sensible to associate a positive or negative quality to selectivity.

The number of rows returned by an operation is the *cardinality*. Formula 2-1 shows the relationship between selectivity and cardinality. In this formula, the *num_rows* value is the number of input rows.

$$cardinality = selectivity \cdot num\_rows$$

**Formula 2-1.** *Relationship between selectivity and cardinality*

**Caution** In the relational model, the term *cardinality* refers to the number of tuples in a relation. Because a relation never contains duplicates, when the relation is unary, the number of tuples corresponds to the number of distinct values it represents. Probably for this reason, in some publications, the term *cardinality* refers to the number of distinct values stored in a particular column. Because SQL allows tables containing duplicates (that is, SQL doesn't adhere to the relational model in this regard), I never use the term *cardinality* to refer to the number of distinct values in a column. In addition, Oracle itself isn't consistent in the definition of the term. Sometimes, in the documentation, Oracle uses it for the number of distinct values, and sometimes for the number of rows returned by an operation.

Let's take a look at a couple of examples based on the `selectivity.sql` script. In the following query, the selectivity of the operation accessing the table is 1. That's because no `WHERE` clause is applied, and therefore, the query returns all rows stored in the table. The cardinality, which is equal to the number of rows stored in the table, is 10,000:

```
SQL> SELECT * FROM t;

...

10000 rows selected.
```

In the following query, the cardinality of the operation accessing the table is 2,601, and therefore the selectivity is 0.2601 (2,601 rows returned out of 10,000):

```
SQL> SELECT * FROM t WHERE n1 BETWEEN 6000 AND 7000;

...

2601 rows selected.
```

In the following query, the cardinality of the operation accessing the table is 0, and therefore the selectivity is also 0 (0 rows returned out of 10,000):

```
SQL> SELECT * FROM t WHERE n1 = 19;

no rows selected.
```

In the previous three examples, the selectivity related to the operation accessing the table is computed by dividing the cardinality of the query by the number of rows stored in the table. This is possible because the three queries don't contain joins or operations leading to aggregations. As soon as a query contains a `GROUP BY` clause or aggregate functions in the `SELECT` clause, the execution plan contains at least one aggregate operation. The following query illustrates this (note the presence of the `sum` aggregate function):

```
SQL> SELECT sum(n2) FROM t WHERE n1 BETWEEN 6000 AND 7000;

   SUM(N2)
----------
     70846

1 row selected.
```

In this type of situation, it's not possible to compute the selectivity of the access operation based on the cardinality of the query (in this case, 1). Instead, a query like the following should be executed to find out how many rows are returned by the access operation and passed as input to the aggregate operation. Here, the cardinality of the access operation accessing the table is 2,601, and therefore the selectivity is 0.2601 (2,601/10,000):

```
SQL> SELECT count(*) FROM t WHERE n1 BETWEEN 6000 AND 7000;

  COUNT(*)
----------
      2601

1 row selected.
```

As you'll see later, especially in Chapter 13, knowing the selectivity of an operation helps you determine what the most efficient access path is.

## What Is a Cursor?

A cursor is a handle (that is, a memory structure that enables a program to access a resource) that references a private SQL area with an associated shared SQL area. As shown in Figure 2-1, although the handle is a client-side memory structure, it references a memory structure allocated by a server process that, in turn, references a memory structure stored in the SGA, and more precisely in the library cache.

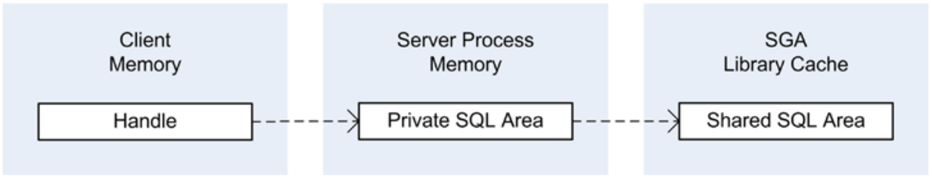


***Figure 2-1.*** *A cursor is a handle to a private SQL area with an associated shared SQL area*

A private SQL area stores data such as bind variable values and query execution state information. As its name suggests, a private SQL area belongs to a specific session. The session memory used to store private SQL areas is called user global area (UGA).

A shared SQL area consists of two separate structures: the so-called *parent cursor* and *child cursor*. The key information stored in a parent cursor is the text of the SQL statement associated with the cursor. Simply put, the SQL statement specifies the processing to be performed. The key elements stored in a child cursor are the execution environment and the execution plan. These elements specify how the processing is carried out. A shared SQL area can be used by several sessions, and therefore it's stored in the library cache.

---

■ **Note** In practice, the terms *cursor* and *private/shared SQL area* are used interchangeably.

---

## Life Cycle of a Cursor

Having a good understanding of the life cycle of cursors is required knowledge for optimizing applications that execute SQL statements. The following are the steps carried out during the processing of a cursor:

1. *Open cursor:* A private SQL area is allocated in the UGA of the session used to open the cursor. A client-side handle referencing the private SQL area is also allocated. Note that no SQL statement is associated with the cursor yet.

2. *Parse cursor:* A shared SQL area containing the parsed representation of the SQL statement associated to it and its execution plan (which describes how the SQL engine will execute the SQL statement) is generated and loaded in the SGA, specifically into the library cache. The private SQL area is updated to store a reference to the shared SQL area. (The next section describes parsing in more detail.)

3. *Define output variables:* If the SQL statement returns data, the variables receiving it must be defined. This is necessary not only for queries but also for `DELETE`, `INSERT`, and `UPDATE` statements that use the `RETURNING` clause.

4. *Bind input variables:* If the SQL statement uses bind variables, their values must be provided. No check is performed during the binding. If invalid data is passed, a runtime error will be raised during the execution.

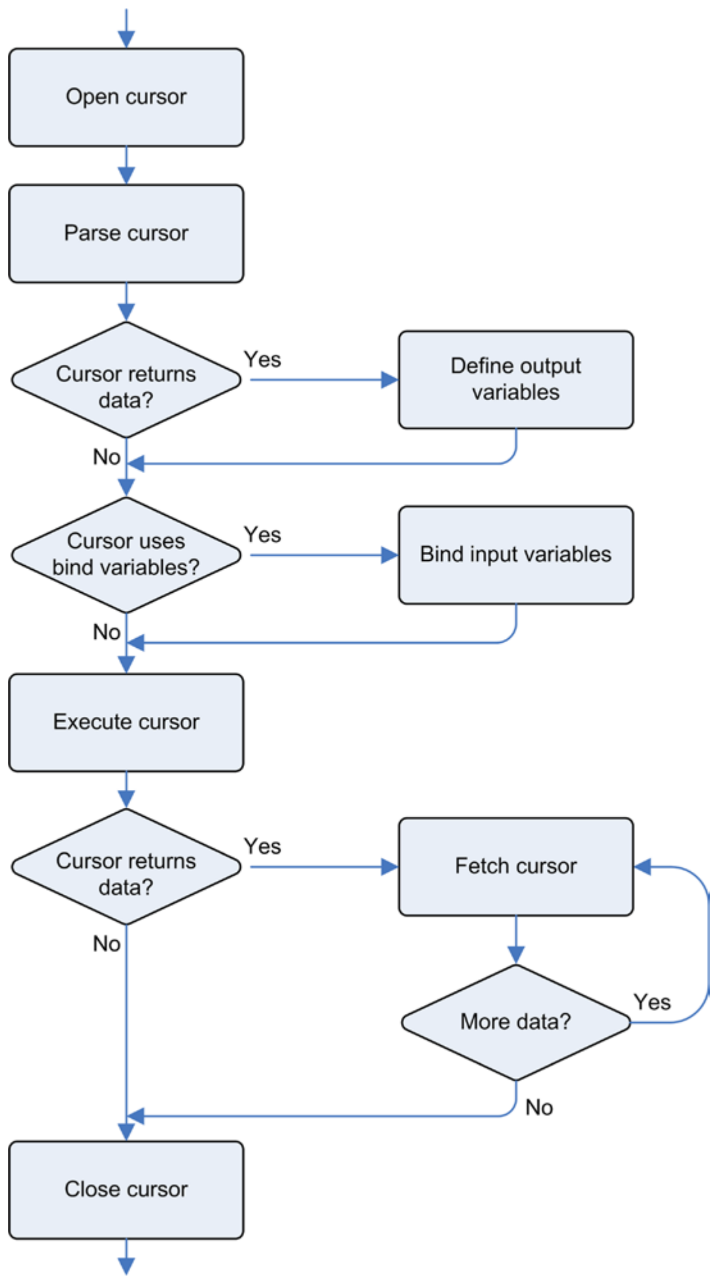


***Figure 2-2.*** *Life cycle of a cursor*

5. *Execute cursor:* The SQL statement is executed. But be careful—the database engine doesn't always do anything significant during this phase. In fact, for many types of queries, the real processing is usually delayed to the fetch phase.

6. *Fetch cursor:* If the SQL statement returns data, this step retrieves it. Especially for queries, this step is where most of the processing is performed. In the case of queries, the result set might be partially fetched. In other words, the cursor might be closed before fetching all the rows.

7. *Close cursor:* The resources associated with the handle and the private SQL area are freed and consequently made available for other cursors. The shared SQL area in the library cache isn't removed. It remains there in the hope of being reused in the future.

To better understand this process, it's best to think about each step being executed separately in the order shown by Figure 2-2. In practice, though, different optimization techniques are applied to speed up processing. For example, bind variable peeking requires that the generation of the execution plan is delayed until the value of the bind variables is known.

Depending on the programming environment or techniques you're using, the different steps depicted in Figure 2-2 may be implicitly or explicitly executed. To make the difference clear, take a look at the following two PL/SQL blocks that are available in the `lifecycle.sql` script. Both have the same purpose (reading one row from the `emp` table), but they're coded in a very different way.

The first is a PL/SQL block using the `dbms_sql` package to explicitly code every step shown in Figure 2-2:

```
DECLARE
  l_ename emp.ename%TYPE := 'SCOTT';
  l_empno emp.empno%TYPE;
  l_cursor INTEGER;
  l_retval INTEGER;
BEGIN
  l_cursor := dbms_sql.open_cursor;
  dbms_sql.parse(l_cursor, 'SELECT empno FROM emp WHERE ename = :ename', 1);
  dbms_sql.define_column(l_cursor, 1, l_empno);
  dbms_sql.bind_variable(l_cursor, ':ename', l_ename);
  l_retval := dbms_sql.execute(l_cursor);
  IF dbms_sql.fetch_rows(l_cursor) > 0
  THEN
    dbms_sql.column_value(l_cursor, 1, l_empno);
    dbms_output.put_line(l_empno);
  END IF;
  dbms_sql.close_cursor(l_cursor);
END;
```

The second is a PL/SQL block taking advantage of an implicit cursor; basically, the PL/SQL block delegates the control over the cursor to the PL/SQL compiler:

```
DECLARE
  l_ename emp.ename%TYPE := 'SCOTT';
  l_empno emp.empno%TYPE;
BEGIN
  SELECT empno INTO l_empno
  FROM emp
  WHERE ename = l_ename;
  dbms_output.put_line(l_empno);
END;
```

Most of the time, what the compiler does is fine. In fact, internally, the compiler generates code that is similar to that shown in the first PL/SQL block. However, sometimes you need more control over the different steps performed during processing. Thus, you can't always use an implicit cursor. For example, between the two PL/SQL blocks, there's a slight but important difference. Independently of how many rows the query returns, the first block doesn't generate an exception. Instead, the second block generates an exception if zero or several rows are returned.

## How Parsing Works

The previous section describes the life cycle of cursors, and this section focuses on the parse phase. The steps carried out during this phase, as shown in Figure 2-3, are the following:

1. *Include VPD predicates*: If Virtual Private Database (VPD, formerly known as row-level security) is in use and active for one of the tables referenced in the parsed SQL statement, the predicates generated by the security policies are included in its `WHERE` clause.
2. *Check syntax, semantics, and access rights*: This step makes sure not only that the SQL statement is correctly written but also that all objects referenced by the SQL statement exist and the user parsing it has the necessary privileges to access them.
3. *Store parent cursor in a shared SQL area*: Whenever a shareable parent cursor isn't yet available, some memory is allocated from the library cache, and a new parent cursor is stored inside it.
4. *Generate execution plan*: During this phase, the query optimizer produces an execution plan for the parsed SQL statement. (This topic is fully described in Chapter 6.)
5. *Store child cursor in a shared SQL area*: Some memory is allocated, and the shareable child cursor is stored inside it and associated with its parent cursor.

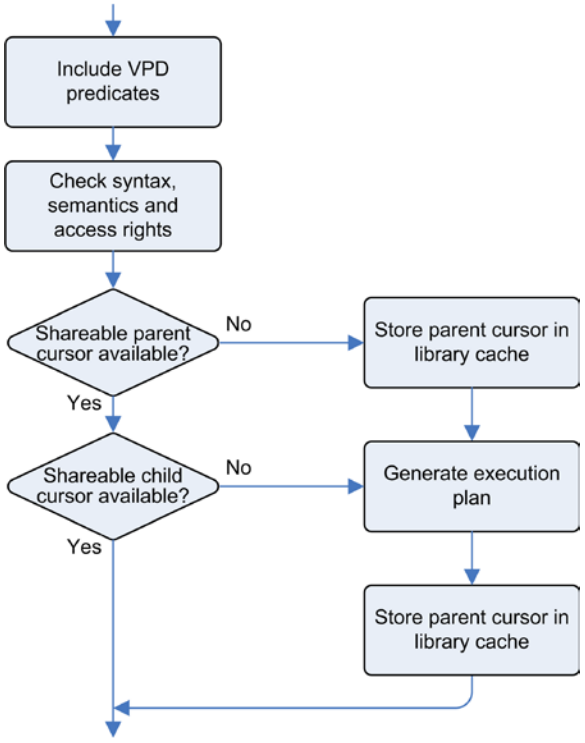

Include VPD predicates
Check syntax, semantics and access rights
Shareable parent cursor available?
No
Yes
Store parent cursor in library cache
Shareable child cursor available?
No
Yes
Generate execution plan
Store parent cursor in library cache

***Figure 2-3.*** *Steps carried out during the parse phase*

Once stored in the library cache, parent and child cursors are externalized through the `v$sqlarea` and `v$sql` views, respectively. Strictly speaking, the identifier of a cursor is its memory address, both for the parent and the child. But in most situations, cursors are identified with two columns: `sql_id` and `child_number`. The `sql_id` column identifies parent cursors. Both values together identify child cursors. There are cases, though, where the two values together aren't sufficient to identify a cursor. In fact, depending on the version[1], parent cursors with many children are obsoleted and replaced by new ones. As a result, the `address` column is also required to identify a cursor.

When shareable parent and child cursors are available and, consequently, only the first two operations are carried out, the parse is called a *soft parse*. When all operations are carried out, it's called a *hard parse*.

From a performance point of view, you should avoid hard parses as much as possible. This is precisely why the database engine stores shareable cursors in the library cache. In this way, every process belonging to the instance might be able to reuse them. There are two reasons why hard parses should be avoided. The first is that the generation of an execution plan is a very CPU-intensive operation. The second is that memory in the shared pool is needed for storing the parent and child cursors in the library cache. Because the shared pool is shared over all sessions, memory allocations in the shared pool are serialized. For that purpose, one of the latches protecting the shared pool (known as shared pool latches) must be obtained to allocate the memory needed for both the parent and child cursors. Because of this serialization, an application causing a lot of hard parses is likely to experience contention for shared pool latches. Even if the impact of soft parses is much lower than that of hard parses, avoiding soft parses is desirable as well because they're also subject to some serialization. In fact, the database engine must guarantee that the memory structures it accesses aren't modified while it's searching for a shareable cursor. The actual implementation depends on the version: through version 10.2.0.1 one of the library cache latches must be obtained, but from version 10.2.0.2 onward Oracle started replacing the library cache latches with mutexes, and as of version 11.1 only mutexes are used for that purpose. In summary, you should avoid soft and hard parses as much as possible because they inhibit the scalability of applications. (Chapter 12 covers this topic in detail.)

## Shareable Cursors

The result of a parse operation is a parent cursor and a child cursor stored in a shared SQL area inside the library cache. Obviously, the aim of storing them in a shared memory area is to allow their reutilization and thereby avoid hard parses. Therefore, it's necessary to discuss in what situations it's possible to reuse a parent or child cursor. To illustrate how sharing parent and child cursors works, this section covers three examples.

The purpose of the first example, based on the `sharable_parent_cursors.sql` script, is to show a case where the parent cursor can't be shared. The key information related to a parent cursor is the text of a SQL statement. Therefore, in general, several SQL statements share the same parent cursor if their text is exactly the same. This is the most essential requirement. There's, however, an exception to this when *cursor sharing* is enabled. In fact, when cursor sharing is enabled, the database engine can automatically replace the literals used in SQL statements with bind variables. Hence, the text of the SQL statements received by the database engine is modified before being stored in parent cursors. (Chapter 12 covers cursor sharing in more detail.) In the first example, four SQL statements are executed. Two have the same text. Two others differ only because of lowercase and uppercase letters or blanks:

```
SQL> SELECT * FROM t WHERE n = 1234;

SQL> select * from t where n = 1234;

SQL> SELECT   *   FROM   t   WHERE   n=1234;

SQL> SELECT * FROM t WHERE n = 1234;
```

[1]The maximum number of child cursors per parent cursor has been changed serveral times: up to and including 11.1.0.6 it's 1,026; from 11.1.0.7 to 11.2.0.1 it's 32,768; in 11.2.0.2 it's 65,536; as of 11.2.0.3 it's 100.

Through the `v$sqlarea` view, it's possible to confirm that three distinct parent cursors were created. Also notice the number of executions for each cursor:

```
SQL> SELECT sql_id, sql_text, executions
  2  FROM v$sqlarea
  3  WHERE sql_text LIKE '%1234';

SQL_ID        SQL_TEXT                                 EXECUTIONS
------------- ---------------------------------------- ----------
2254m1487jg50 select * from t where n = 1234                    1
g9y3jtp6ru4cb SELECT * FROM t WHERE n = 1234                    2
7n8p5s2udfdsn SELECT   *   FROM   t   WHERE   n=1234            1
```

The aim of the second example, based on the `sharable_child_cursors.sql` script, is to show a case where the parent cursor, but not the child cursor, can be shared. The key information related to a child cursor is an execution plan and the execution environment related to it. As a result, several SQL statements are able to share the same child cursor only if they share the same parent cursor and their execution environments are compatible. To illustrate, the same SQL statement is executed with two different values of the `optimizer_mode` initialization parameter:

```
SQL> ALTER SESSION SET optimizer_mode = all_rows;

SQL> SELECT count(*) FROM t;

COUNT(*)
--------
    1000

SQL> ALTER SESSION SET optimizer_mode = first_rows_1;

SQL> SELECT count(*) FROM t;

COUNT(*)
--------
    1000
```

The result is that a single parent cursor (5tjqf7sx5dzmj) and two child cursors (0 and 1) are created. It's also essential to note that both child cursors have the same execution plan (the `plan_hash_value` column has the same value). This shows very well that a new child cursor was created because of a new individual execution environment and *not* because another execution plan was generated:

```
SQL> SELECT sql_id, child_number, optimizer_mode, plan_hash_value
  2  FROM v$sql
  3  WHERE sql_text = 'SELECT count(*) FROM t';

SQL_ID        CHILD_NUMBER OPTIMIZER_MODE PLAN_HASH_VALUE
------------- ------------ -------------- ---------------
5tjqf7sx5dzmj            0 ALL_ROWS            2966233522
5tjqf7sx5dzmj            1 FIRST_ROWS          2966233522
```

---

■ **Caution** As the previous example shows, the `optimizer_mode` column doesn't show the right value for child number 1. In fact, the column shows `FIRST_ROWS` instead of `FIRST_ROWS_1`. The same behavior can be observed with `FIRST_ROWS_10`, `FIRST_ROWS_100`, and `FIRST_ROWS_1000` as well. This fact leads to the potential problem that even though the execution environment is different, the SQL engine doesn't distinguish that difference. As a result, a child cursor might be incorrectly shared.

---

To know which mismatch led to several child cursors, you can query the `v$sql_shared_cursor` view. In it you might find, for each child cursor (except the first one, 0), why it wasn't possible to share a previously created child cursor. For several types of incompatibility (64 of them in version 12.1), there's a column that is set to either N (no mismatch) or Y (mismatch). With the following query, it's possible to confirm that in the previous example, the mismatch for the second child cursor was because of a different optimizer mode:

```
SQL> SELECT optimizer_mode_mismatch
  2  FROM v$sql_shared_cursor
  3  WHERE sql_id = '5tjqf7sx5dzmj'
  4  AND child_number = 1;

OPTIMIZER_MODE_MISMATCH
-----------------------
Y
```

As of version 11.2.0.2, the `v$sql_shared_cursor` view provides a column named `reason`. Its aim is to show not only a textual description of the mismatch leading to a new child cursor, but also additional information about the mismatch. Because the information the `reason` column contains is highly dependent on the type of the mismatch, its datatype is `CLOB`, and the data is an XML fragment. For example, in the following case, three XML elements contain the key information. The reason ("Optimizer mismatch") is stored in the `reason` element, the optimizer mode (which is `1`, meaning `ALL_ROWS`) of the cursor already stored in the library cache is stored in the `optimizer_mode_cursor` element, and the optimizer mode required by the session parsing the statement (`2`, meaning `FIRST_ROWS`) is stored in the `optimizer_mode_current` element:

```
SQL> SELECT reason
  2  FROM v$sql_shared_cursor
  3  WHERE sql_id = '5tjqf7sx5dzmj'
  4  AND child_number = 0;

REASON
--------------------------------------------------------------------------------------------
<ChildNode><ChildNumber>0</ChildNumber><ID>3</ID><reason>Optimizer mismatch(10)</reason><siz
e>3x4</size><optimizer_mode_hinted_cursor>0</optimizer_mode_hinted_cursor><optimizer_mode_cu
rsor>1</optimizer_mode_cursor><optimizer_mode_current>2</optimizer_mode_current></ChildNode>

SQL> SELECT x.reason,
  2         decode(x.optimizer_mode_cursor,
  3                1, 'ALL_ROWS',
  4                2, 'FIRST_ROWS',
  5                3, 'RULE',
  6                4, 'CHOOSE', x.optimizer_mode_cursor) AS optimizer_mode_cursor,
       decode(x.optimizer_mode_current,
              1, 'ALL_ROWS',
              2, 'FIRST_ROWS',
              3, 'RULE',
              4, 'CHOOSE', x.optimizer_mode_current) AS optimizer_mode_current
FROM v$sql_shared_cursor s,
     XMLTable('/ChildNode'
              PASSING XMLType(reason)
              COLUMNS
                reason VARCHAR2(100)          PATH '/ChildNode/reason',
                optimizer_mode_cursor NUMBER  PATH '/ChildNode/optimizer_mode_cursor',
                optimizer_mode_current NUMBER PATH '/ChildNode/optimizer_mode_current'
                      ) x
WHERE s.sql_id = '5tjqf7sx5dzmj'
AND s.child_number = 0;

REASON                 OPTIMIZER_MODE_CURSOR OPTIMIZER_MODE_CURRENT
---------------------- --------------------- ----------------------
Optimizer mismatch(10) ALL_ROWS              FIRST_ROWS
```

The aim of the third example, also based on the `sharable_child_cursors.sql` script, is to show you that the execution environment can not only influence the execution plan, but also that the result of SQL statements might be different. This is another reason why the execution environment must be compatible for sharing a child cursor. For example, the output of the following SQL statements illustrates the impact of the `nls_sort` initialization parameter:

```
SQL> ALTER SESSION SET nls_sort = binary;

SQL> SELECT * FROM t ORDER BY pad;

  N PAD
--- ---
  1 1
  2 =
  3 Z
  4 z

SQL> ALTER SESSION SET nls_sort = xgerman;

SQL> SELECT * FROM t ORDER BY pad;

  N PAD
--- ---
  2 =
  4 z
  3 Z
  1 1
```

Because the execution environment is different, two child cursors with the same parent cursor are used. Notice that in this case, the mismatch is visible through the `v$sql_shared_cursor` view, specifically in the `language_mismatch` column:

```
SQL> SELECT sql_id, child_number, plan_hash_value, executions
  2  FROM v$sql
  3  WHERE sql_text = 'SELECT * FROM t ORDER BY pad';

SQL_ID        CHILD_NUMBER PLAN_HASH_VALUE EXECUTIONS
------------- ------------ --------------- ----------
1f7qg6nu40shd            0       961378228          1
1f7qg6nu40shd            1       961378228          1

SQL> SELECT child_number, language_mismatch
  2  FROM v$sql_shared_cursor
  3  WHERE sql_id = '1f7qg6nu40shd'
  4  AND child_number > 0;

CHILD_NUMBER LANGUAGE_MISMATCH
------------ -----------------
           1 Y
```

In practice, it's quite common to see more hard parses caused by nonshared parent cursors than nonshared child cursors. In fact, more often than not, there are few child cursors for each parent cursor. If the parent cursors can't be shared, it almost always means that the text of SQL statements changes constantly. This happens if either the SQL statements are dynamically generated by the application or literals are used instead of bind variables. In general, dynamically generated SQL statements can't be avoided. On the other hand, it's usually possible to use bind variables. Unfortunately, it isn't always good to use them. The following discussion of the pros and cons of bind variables will help you understand when it's good and not so good to use them.

## Bind Variables

Bind variables impact applications in three ways. First, from a development point of view, they make programming either easier or more difficult (or more precisely, more or less code must be written). In this case, the effect depends on the application programming interface used to execute the SQL statements. For example, if you're programming PL/SQL code, it's easier to execute them with bind variables. On the other hand, if you're programming in Java with JDBC, it's easier to execute SQL statements without bind variables. Second, from a security point of view, bind variables mitigate the risk of SQL injection. Third, from a performance point of view, bind variables introduce both an advantage and a disadvantage.

---

■ **Note** In the following sections, you'll see some execution plans. Chapter 10 explains how to obtain and interpret execution plans. You might consider returning to this chapter after reading Chapter 10 if something isn't clear.

---

## Advantage

The advantage of bind variables for performance is that they allow the sharing of parent cursors in the library cache and that way avoid hard parses and the overhead associated with them. The following example, which is an excerpt of the output generated by the `bind_variables_graduation.sql` script, shows three `INSERT` statements that, thanks to bind variables, share the same cursor in the library cache:

```
SQL> VARIABLE n NUMBER

SQL> VARIABLE v VARCHAR2(32)

SQL> EXECUTE :n := 1; :v := 'Helicon';

SQL> INSERT INTO t (n, v) VALUES (:n, :v);

SQL> EXECUTE :n := 2; :v := 'Trantor';

SQL> INSERT INTO t (n, v) VALUES (:n, :v);

SQL> EXECUTE :n := 3; :v := 'Kalgan';

SQL> INSERT INTO t (n, v) VALUES (:n, :v);

SQL> SELECT sql_id, child_number, executions
  2  FROM v$sql
  3  WHERE sql_text = 'INSERT INTO t (n, v) VALUES (:n, :v)';

SQL_ID        CHILD_NUMBER EXECUTIONS
------------- ------------ ----------
6cvmu7dwnvxwj            0          3
```

There are, however, situations where several child cursors are created even with bind variables. The following example shows such a case. Notice that the `INSERT` statement is the same as in the previous example. Only the maximum size of the `VARCHAR2` variable has changed (from 32 to 33):

```
SQL> VARIABLE v VARCHAR2(33)

SQL> EXECUTE :n := 4; :v := 'Terminus';

SQL> INSERT INTO t (n, v) VALUES (:n, :v);

SQL> SELECT sql_id, child_number, executions
  2  FROM v$sql
  3  WHERE sql_text = 'INSERT INTO t (n, v) VALUES (:n, :v)';

SQL_ID        CHILD_NUMBER EXECUTIONS
------------- ------------ ----------
6cvmu7dwnvxwj            0          3
6cvmu7dwnvxwj            1          1
```

The new child cursor (1) is created because the execution environment between the first three INSERT statements and the fourth has changed. The mismatch, as shown in the following example, can be confirmed by querying the v$sql_shared_cursor view. Note that the bind_length_upgradeable column exists as of version 11.2 only. In previous releases, this information is provided by the bind_mismatch column:

```
SQL> SELECT child_number, bind_length_upgradeable
  2  FROM v$sql_shared_cursor
  3  WHERE sql_id = '6cvmu7dwnvxwj';

CHILD_NUMBER BIND_LENGTH_UPGRADEABLE
------------ -----------------------
           0 N
           1 Y
```

What happens is that the database engine uses a feature called *bind variable graduation*. The aim of this feature is to minimize the number of child cursors by graduating bind variables (which vary in size) into four groups depending on their size. The first group contains the bind variables with up to and including 32 bytes, the second contains the bind variables between 33 and 128 bytes, the third contains the bind variables between 129 and 2,000 bytes, and the last contains the bind variables of more than 2,000 bytes. Bind variables of NUMBER datatype are graduated to their maximum length, which is 22 bytes. As the following example shows, the v$sql_bind_metadata view displays the maximum size of a group. Notice how the value 128 is used, even if the variable of child cursor 1 was defined as 33:

```
SQL> SELECT s.child_number, m.position, m.max_length,
  2         decode(m.datatype,1,'VARCHAR2',2,'NUMBER',m.datatype) AS datatype
  3  FROM v$sql s, v$sql_bind_metadata m
  4  WHERE s.sql_id = '&sql_id'
  5  AND s.child_address = m.address
  6  ORDER BY 1, 2;

CHILD_NUMBER   POSITION MAX_LENGTH DATATYPE
------------ ---------- ---------- --------
           0          1         22 NUMBER
           0          2         32 VARCHAR2
           1          1         22 NUMBER
           1          2        128 VARCHAR2
```

---

■ **Note** The example shows that there's a bind mismatch when bind variables of different groups are used. This occurs only when a bind variable is associated to a new group having a greater maximum size than the original. In fact, if you carefully review the example, the size of the bind variables always increases. If they decreased, all executions could share a single child cursor. In fact, child cursors created with VARCHAR2 bind variables of the maximum size support any VARCHAR2 bind variables having a smaller size.

---

It goes without saying that each time a new child cursor is created, an execution plan is generated. Whether this new execution plan is equal to the one used by another child cursor also depends on the value of the bind variables. This is described in the next section.

## Disadvantage

The disadvantage of using bind variables in WHERE clauses for performance is that, in some situations, crucial information is hidden from the query optimizer. In fact, for the query optimizer, it's better to have literals instead of bind variables. With literals, the query optimizer is always able to make the best possible estimations. This is especially true for range comparison predicates (for example, predicates based on BETWEEN, greater-than, or less-than comparison conditions), for checking whether a value is outside the range of available values (that is, lower than the minimum value or higher than the maximum value stored in the column), and when histograms are used. To illustrate, let's take a table with 1,000 rows that stores, in the id column, all the integer values between 1 (the minimum value) and 1,000 (the maximum value):

```
SQL> SELECT count(id), count(DISTINCT id), min(id), max(id) FROM t;

 COUNT(ID) COUNT(DISTINCTID)    MIN(ID)    MAX(ID)
---------- ----------------- ---------- ----------
      1000              1000          1       1000
```

When a user selects all rows that have an id of less than 990, the query optimizer knows (thanks to object statistics) that about 99% of the table is selected. Therefore, it chooses an execution plan with a full table scan. Also notice how the estimated cardinality (Rows column in the execution plan) corresponds almost exactly to the number of rows returned by the query:

```
SQL> SELECT count(pad) FROM t WHERE id < 990;

COUNT(PAD)
----------
       989

-------------------------------------------
| Id  | Operation          | Name | Rows  |
-------------------------------------------
|   0 | SELECT STATEMENT   |      |       |
|   1 |  SORT AGGREGATE    |      |     1 |
|   2 |   TABLE ACCESS FULL| T    |   990 |
-------------------------------------------
```

When another user selects all rows that have an id of less than 10, the query optimizer knows that only about 1% of the table is selected. Therefore, it chooses an execution plan with an index scan. Also in this case, notice the very good estimation:

```
SQL> SELECT count(pad) FROM t WHERE id < 10;

COUNT(PAD)
----------
         9
```

```
-----------------------------------------------------
| Id  | Operation                    | Name | Rows  |
-----------------------------------------------------
|   0 | SELECT STATEMENT             |      |       |
|   1 |  SORT AGGREGATE              |      |     1 |
|   2 |   TABLE ACCESS BY INDEX ROWID| T    |     9 |
|   3 |    INDEX RANGE SCAN          | T_PK |     9 |
-----------------------------------------------------
```

Whenever dealing with bind variables, the query optimizer used to ignore their values. Thus, good estimations like in the previous examples weren't possible. To address this problem, a feature called *bind variable peeking* was introduced in Oracle9*i*. The concept of bind variable peeking is simple. Before generating an execution plan, the query optimizer peeks at the values of bind variables and uses them as literals. The problem with this approach is that the generated execution plan depends on the values provided by the first execution. The following example, which is based on the `bind_variables_peeking.sql` script, illustrates this behavior. Note that the first optimization is performed with the value 990. Consequently, the query optimizer chooses a full table scan. It's this choice, since the cursor is shared, that impacts the second query that uses 10 for the selection:

```
SQL> VARIABLE id NUMBER

SQL> EXECUTE :id := 990;

SQL> SELECT count(pad) FROM t WHERE id < :id;

COUNT(PAD)
----------
       989

-------------------------------------------
| Id  | Operation          | Name | Rows  |
-------------------------------------------
|   0 | SELECT STATEMENT   |      |       |
|   1 |  SORT AGGREGATE    |      |     1 |
|   2 |   TABLE ACCESS FULL| T    |   990 |
-------------------------------------------

SQL> EXECUTE :id := 10;

SQL> SELECT count(pad) FROM t WHERE id < :id;

COUNT(PAD)
----------
         9

-------------------------------------------
| Id  | Operation          | Name | Rows  |
-------------------------------------------
|   0 | SELECT STATEMENT   |      |       |
|   1 |  SORT AGGREGATE    |      |     1 |
|   2 |   TABLE ACCESS FULL| T    |   990 |
-------------------------------------------
```

Of course, as shown in the following example, if the first execution takes place with the value 10, the query optimizer chooses an execution plan with an index scan—and that, once more, occurs for both queries. Note that to avoid sharing the cursor used for the previous example, the queries were written in lowercase letters.

```
SQL> EXECUTE :id := 10;

SQL> select count(pad) from t where id < :id;

COUNT(PAD)
----------
         9

-----------------------------------------------------
| Id  | Operation                    | Name | Rows  |
-----------------------------------------------------
|   0 | SELECT STATEMENT             |      |       |
|   1 |  SORT AGGREGATE              |      |     1 |
|   2 |   TABLE ACCESS BY INDEX ROWID| T    |     9 |
|   3 |    INDEX RANGE SCAN          | T_PK |     9 |
-----------------------------------------------------

SQL> EXECUTE :id := 990;

SQL> select count(pad) from t where id < :id;

COUNT(PAD)
----------
       989

-----------------------------------------------------
| Id  | Operation                    | Name | Rows  |
-----------------------------------------------------
|   0 | SELECT STATEMENT             |      |       |
|   1 |  SORT AGGREGATE              |      |     1 |
|   2 |   TABLE ACCESS BY INDEX ROWID| T    |     9 |
|   3 |    INDEX RANGE SCAN          | T_PK |     9 |
-----------------------------------------------------
```

It's essential to understand that as long as the cursor remains in the library cache and can be shared, it will be reused. This occurs regardless of the efficiency of the execution plan related to it.

To solve this problem, as of version 11.1, the database engine uses a new feature called *adaptive cursor sharing* (also known as *bind-aware cursor sharing*). Its purpose is to automatically recognize when the reutilization of an already available cursor leads to inefficient executions. To understand how this feature works, let's start by looking at some information provided by the `v$sql` view about it. The following new columns are available as of version 11.1:

`is_bind_sensitive` indicates not only whether bind variable peeking was used to generate the execution plan but also whether adaptive cursor sharing might be considered. If this is the case, the column is set to Y; otherwise, it's set to N.

`is_bind_aware` indicates whether the cursor is using adaptive cursor sharing. If yes, the column is set to Y; if not, it's set to N.

`is_shareable` indicates whether the cursor can be shared. If it can, the column is set to Y; otherwise, it's set to N. If set to N, the cursor will no longer be reused.

In the following example, based on the `adaptive_cursor_sharing.sql` script, the cursor is shareable and sensitive to bind variables, but it isn't using adaptive cursor sharing:

```
SQL> EXECUTE :id := 10;

SQL> SELECT count(pad) FROM t WHERE id < :id;

COUNT(PAD)
----------
         9

SQL> SELECT sql_id
  2  FROM v$sqlarea
  3  WHERE sql_text = 'SELECT count(pad) FROM t WHERE id < :id';

SQL_ID
-------------
asth1mx10aygn

SQL> SELECT child_number, is_bind_sensitive, is_bind_aware, is_shareable, plan_hash_value
  2  FROM v$sql
  3  WHERE sql_id = 'asth1mx10aygn';

CHILD_NUMBER IS_BIND_SENSITIVE IS_BIND_AWARE IS_SHAREABLE PLAN_HASH_VALUE
------------ ----------------- ------------- ------------ ---------------
           0 Y                 N             Y                 4270555908
```

Something interesting happens when the cursor is executed several times with different values for the bind variable. Notice in the following that child number 0 is no longer shareable and that two new child cursors have replaced it, both using adaptive cursor sharing:

```
SQL> EXECUTE :id := 990;

SQL> SELECT count(pad) FROM t WHERE id < :id;

COUNT(PAD)
----------
       989

SQL> EXECUTE :id := 10;

SQL> SELECT count(pad) FROM t WHERE id < :id;

COUNT(PAD)
----------
         9

SQL> SELECT child_number, is_bind_sensitive, is_bind_aware, is_shareable, plan_hash_value
  2  FROM v$sql
  3  WHERE sql_id = 'asth1mx10aygn'
  4  ORDER BY child_number;
```

```
CHILD_NUMBER IS_BIND_SENSITIVE IS_BIND_AWARE IS_SHAREABLE PLAN_HASH_VALUE
------------ ----------------- ------------- ------------ ---------------
           0 Y                 N             N                 4270555908
           1 Y                 Y             Y                 2966233522
           2 Y                 Y             Y                 4270555908
```

Looking at the execution plans related to the cursor, as you might expect, you see that one of the new children has an execution plan based on a full table scan, whereas the other is based on an index scan:

```
Plan hash value: 4270555908

---------------------------------------------
| Id  | Operation                    | Name |
---------------------------------------------
|   0 | SELECT STATEMENT             |      |
|   1 |  SORT AGGREGATE              |      |
|   2 |   TABLE ACCESS BY INDEX ROWID| T    |
|   3 |    INDEX RANGE SCAN          | T_PK |
---------------------------------------------

Plan hash value: 2966233522

-----------------------------------
| Id  | Operation          | Name |
-----------------------------------
|   0 | SELECT STATEMENT   |      |
|   1 |  SORT AGGREGATE    |      |
|   2 |   TABLE ACCESS FULL| T    |
-----------------------------------
```

To further analyze the reason for the generation of the two child cursors, several dynamic performance views are available: v$sql_cs_statistics, v$sql_cs_selectivity, and v$sql_cs_histogram. The first shows whether peeking was used and the related execution statistics for each child cursor. In the following output, it's possible to confirm that for one execution, the number of rows processed by child cursor 1 is higher than for child cursor 2. This is basically the reason why, in one case, the query optimizer chose a full table scan and in the other an index scan:

```
SQL> SELECT child_number, peeked, executions, rows_processed, buffer_gets
  2  FROM v$sql_cs_statistics
  3  WHERE sql_id = 'asth1mx10aygn'
  4  ORDER BY child_number;

CHILD_NUMBER PEEKED EXECUTIONS ROWS_PROCESSED BUFFER_GETS
------------ ------ ---------- -------------- -----------
           0 Y               1             19           3
           1 Y               1            990          18
           2 Y               1             19           3
```

The v$sql_cs_selectivity view shows the selectivity range related to each predicate of each child cursor. In fact, the database engine doesn't create a new child cursor for each bind variable value. Instead, it groups values together that have about the same selectivity and, consequently, should lead to the same execution plan:

```
SQL> SELECT child_number, trim(predicate) AS predicate, low, high
  2  FROM v$sql_cs_selectivity
  3  WHERE sql_id = 'asth1mx10aygn'
  4  ORDER BY child_number;

CHILD_NUMBER PREDICATE LOW        HIGH
------------ --------- ---------- ----------
           1 <ID       0.890991   1.088989
           2 <ID       0.008108   0.009910
```

The information in the v$sql_cs_selectivity view isn't only used to show you the selectivity range of each child cursor; it's also used by the database engine to select the child cursor to be used. In fact, when a cursor is bind aware, bind variable peeking takes place every time a parse is executed, and the selectivity of the predicates the cursor is based on is estimated. Based on that estimation, the right child cursor is used. Or, if no cursor for that selectivity range exists, a new child cursor is created.

---

■ **Caution** Bind aware cursors necessitate, for every parse, that the query optimizer performs an estimation of the selectivity of their predicates. Because of that, adaptive cursor sharing is sometimes not enabled by the database engine. There are two common cases to consider. The first is for SQL statements containing more than 14 bind variables. The second is when the query optimizer is unable to correctly estimate the selectivity. For example, selectivity can't be estimated for variables requiring an implicit datatype conversion (this is another good reason for using the right data types), or when the referenced objects don't have object statistics.

---

The content of the v$sql_cs_histogram view is used by the SQL engine to decide when a cursor is made bind aware, and therefore, when it should use adaptive cursor sharing. For each child cursor, the view shows three buckets. The first one (bucket_id equal 0) is associated with the executions that are considered efficient, the second one (bucket_id equal 1) with the executions that are considered inefficient, and the third one (bucket_id equal 2) with the very inefficient executions. The idea is that after an execution, the SQL engine compares the estimated cardinalities with the actual cardinalities. Then, depending on how close the two cardinalities are, the execution is associated (that is, the count column is incremented) to one of the three buckets. Later on, while executing the next parse operation involving the same cursor, and depending on how the executions are distributed among the three buckets, a cursor might become bind aware or not. For example, when the number of inefficient executions is as high as the number of efficient ones, the cursor is made bind aware. The following example illustrates this (notice that, for the child number 0, the number of efficient executions is equal to the number of inefficient executions):

```
SQL> SELECT child_number, bucket_id, count
  2  FROM v$sql_cs_histogram
  3  WHERE sql_id = 'asth1mx10aygn'
  4  ORDER BY child_number, bucket_id;
```

```
CHILD_NUMBER  BUCKET_ID      COUNT
------------ ---------- ----------
           0          0          1
           0          1          1
           0          2          0
           1          0          1
           1          1          0
           1          2          0
           2          0          1
           2          1          0
           2          2          0
```

To better understand how the content of the `v$sql_cs_histogram` view is used, I suggest you experiment with several scenarios using the `adaptive_cursor_sharing_histogram.sql` script.

There are two main limitations related to adaptive cursor sharing. First, by default, cursors are created non-bind aware. Second, the bind awareness of a given cursor isn't persisted. As a result, at least one execution, and in some situations many executions (when there were a high number of efficient executions), has to be inefficient before a cursor can take advantage of adaptive cursor sharing. To avoid these limitations, it's possible as of version 11.1.0.7 to specify the `bind_aware` hint. Notice how, in the following example, both child cursors are bind aware and use an efficient execution plan:

```
SQL> EXECUTE :id := 10;

SQL> SELECT /*+ bind_aware */ count(pad) FROM t WHERE id < :id;

COUNT(PAD)
----------
         9

Plan hash value: 4270555908

---------------------------------------------
| Id  | Operation                    | Name |
---------------------------------------------
|   0 | SELECT STATEMENT             |      |
|   1 |  SORT AGGREGATE              |      |
|   2 |   TABLE ACCESS BY INDEX ROWID| T    |
|   3 |    INDEX RANGE SCAN          | T_PK |
---------------------------------------------

SQL> EXECUTE :id := 990;

SQL> SELECT /*+ bind_aware */ count(pad) FROM t WHERE id < :id;

COUNT(PAD)
----------
       989
```

```
Plan hash value: 2966233522

-----------------------------------
| Id  | Operation          | Name |
-----------------------------------
|   0 | SELECT STATEMENT   |      |
|   1 |  SORT AGGREGATE    |      |
|   2 |   TABLE ACCESS FULL| T    |
-----------------------------------

SQL> SELECT child_number, is_bind_sensitive, is_bind_aware, is_shareable, plan_hash_value
  2  FROM v$sql
  3  WHERE sql_id = 'f364ymn1bbr4q'
  4  ORDER BY child_number;

CHILD_NUMBER IS_BIND_SENSITIVE IS_BIND_AWARE IS_SHAREABLE PLAN_HASH_VALUE
------------ ----------------- ------------- ------------ ---------------
           0 Y                 Y             Y                 4270555908
           1 Y                 Y             Y                 2966233522
```

In summary, to increase the likelihood that the query optimizer will generate efficient execution plans, you shouldn't use bind variables. Bind variable peeking might help. Unfortunately, it's sometimes a matter of luck whether an efficient execution plan is generated. The only exception is when, as of version 11.1, the new adaptive cursor sharing automatically recognizes the problem.

## Best Practices

Any feature should be used only if the advantages related to its utilization outweigh the disadvantages. In some situations, it's easy to decide. For example, there's no reason for not using bind variables with SQL statements without a `WHERE` clause (for example, plain `INSERT` statements). On the other hand, bind variables should be avoided at all costs whenever there's a high risk of being stung by bind variable peeking. This is especially true when the following three conditions are met:

- When the query optimizer has to check whether a value is outside the range of available values (that is, lower than the minimum value or higher than the maximum value stored in the column)
- When a predicate in the `WHERE` clause is based on a range condition (for example, `HIREDATE >'2009-12-31'`)
- When the query optimizer makes use of histograms

As a result, for cursors that can be shared, you should *not* use bind variables if one of the preceding three conditions is met. In all other cases, the situation is even less clear. Nevertheless, it's possible to consider two main cases:

*SQL statements processing little data*: Whenever little data is processed, the hard parse time might be close to or even higher than the execution time. In that kind of situation, using bind variables and therefore avoiding a hard parse is usually a must. This is especially true for SQL statements that are expected to be executed frequently. Typically, such SQL statements are used in data entry systems (commonly referred to as OLTP systems).

*SQL statements processing a lot of data*: Whenever a lot of data is processed, the hard parse time is usually several orders of magnitude less than the execution time. In that kind of situation, using bind variables isn't only irrelevant for the whole response time, it also increases the risk that the query optimizer will generate very inefficient execution plans. Therefore, bind variables shouldn't be used. Typically, such SQL statements are used for batch jobs, for reporting purposes, or, in data warehousing environments, by OLAP and BI tools.

## Reading and Writing Blocks

To read and write blocks belonging to data files, the database engine takes advantage of several types of disk I/O operations (see Figure 2-4):

*Logical reads*: A server process performs a logical read when it accesses a block that is either in the buffer cache or in the private memory of the process. Note that logical reads are used for both reading and writing data to a block.

*Buffer cache reads*: A server process performs a buffer cache read when it needs a block that isn't in the buffer cache yet. Consequently, it opens the data file, reads the block, and stores it in the buffer cache.

*DBWR writes*: In general, server processes don't write data into a data file, they modify only the blocks that are stored in the buffer cache. Then, the database writer process (which is a background process) is responsible for storing the modified blocks (also called *dirty blocks*) in the data files.

*Direct reads*: In some particular situations (described in Chapter 13 and 15), a server process is able to directly read blocks from a data file. When it uses this method, the blocks, instead of being loaded in the buffer cache, are directly transferred to the private memory of the process.

*Direct writes*: In some particular situations (described in Chapter 15), a server process is able to directly write blocks into a data file.

In case it isn't important to distinguish between disk I/O operations involving the buffer cache and those that don't, the following terms are used:

*Physical reads* include buffer cache reads and direct reads.

*Physical writes* include direct writes and DBWR writes.

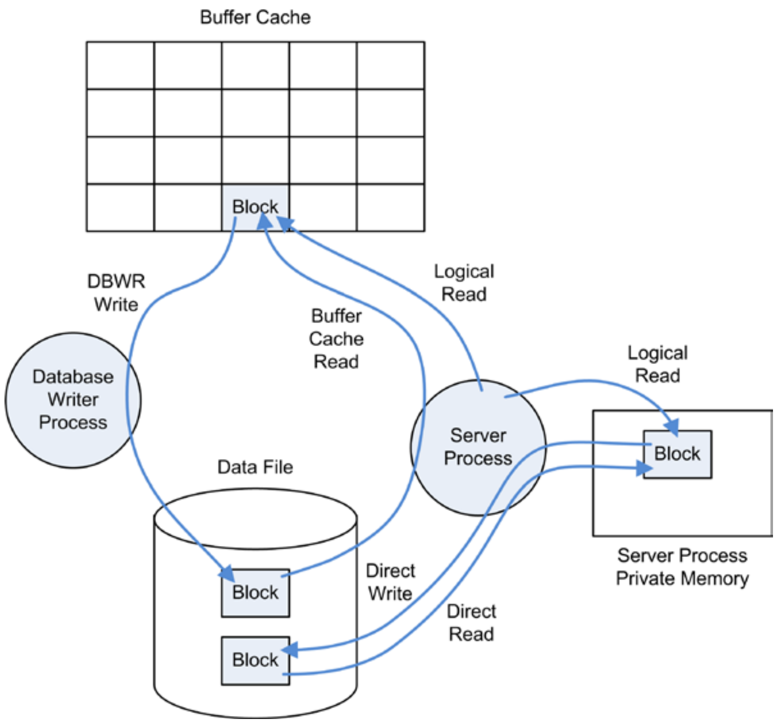


***Figure 2-4.*** *The database engine takes advantage of several types of I/O operations*

When a data file is stored on an Exadata storage server, a database engine can also take advantage of a fifth type of disk I/O operation: *smart scans*. Simply put, on an Exadata system, a database engine can use smart scans instead of direct reads. From a database engine point of view, the main difference between the two disk I/O operations is that direct reads return regular blocks, and smart scans return different data structures. The aim of smart scans is to offload part of the processing that would otherwise be done by a database engine to the storage tier. With that in mind, there are three main goals of using smart scans:

- To avoid moving irrelevant data between Exadata storage servers and database instances
- To allow the Exadata storage servers to avoid reading from disk data that isn't required
- To offload CPU intensive operations to Exadata storage servers and thereby reduce the CPU utilized by database engines

## WHAT'S ORACLE EXADATA?

*Exadata* is a database machine engineered by Oracle that is composed of database servers, Exadata storage servers, an InfiniBand fabric for storage networking, and all the other components required to host Oracle Database. However, Exadata isn't only a hardware solution. When Oracle Database runs on Exadata hardware, specific software features that are designed to result in better performance are enabled. One of the key design decisions was to offload part of the processing that would otherwise be done by database servers to the storage servers.

## Instrumentation

As mentioned in Chapter 1, every application should be instrumented. In other words, the question isn't whether you should do it but how you should do it. This is an important decision that architects should make at the beginning of the development of a new application. Even though instrumentation code is usually implemented to externalize the behavior of an application in case of abnormal conditions, it can also be useful for investigating performance issues. To identify performance problems, we're particularly interested in knowing which operations are performed, in which order, how much data is being processed, how many times operations are performed, and how much time they take. In some cases (for example, large jobs), it's also useful to know how many resources are used. Because information at the call or line level is already provided by code profilers, with instrumentation you should focus particularly on business-relevant operations and on interactions between components (tiers). In addition, if a request needs complex processing inside the same component, it could be wise to provide the response time of the major steps carried out during the processing. In other words, to effectively take advantage of instrumentation code, you should add it to strategic positions in the code. Let me stress that without information about the response time, the instrumentation is useless for investigating performance problems.

Let's look at an example. In the application JPetStore (briefly introduced in Chapter 1), there's an action called *Sign-on* for which Figure 2-5 shows the sequence diagram. Based on this diagram, the instrumentation should provide at least the following information:

- The system response time of the request from the perspective of the servlet[2] responding to requests (`FrameworkServlet`). This is the business-relevant operation.
- The SQL statement and response time of the interactions between the data access object (`AccountDao`) and the database. This is the interaction between middle tier and database tier.
- Timestamp of beginning and ending of both the request and the interaction with the database.

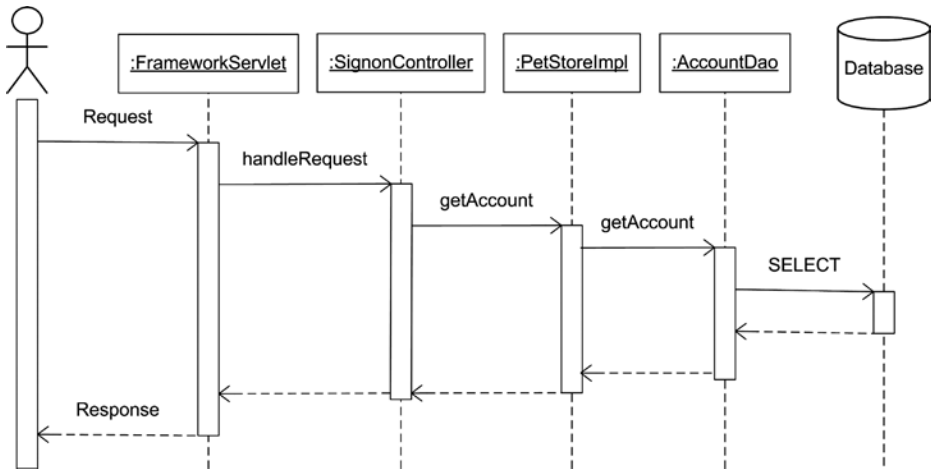


***Figure 2-5.*** *Sequence diagram for the Sign-on action of JPetStore*

[2]A *servlet* is a Java program that responds to requests coming from web clients. It runs in a J2EE application server.

With these values and the user response time—which you can easily measure with a watch if you have access to the application—you can break up the response time in a way similar to Figure 1-7.

In practice, you can't easily add the instrumentation code wherever you want. In the case of Figure 2-5, you have two problems. The first is that the servlet (`FrameworkServlet`) is a class provided by the Spring framework. Therefore, you don't want to modify it. The second is that the data access object (`AccountDao`) is just an interface used by the persistence framework (iBatis in this case). Therefore, you can't add code to it either. For the first problem, you could create your own servlet that inherits from `FrameworkServlet` by simply adding the instrumentation code. For the second one, you could decide to instrument the call to the persistence framework for the sake of simplicity. This shouldn't be a problem because the database itself is already instrumented, and so, if necessary, you're able to determine the overhead of the persistence framework itself.

Now that you've seen how to decide where the instrumentation code should be added, we can take a look at a concrete example of how to implement it in the application code. Afterward, we'll examine an Oracle-specific instrumentation of database calls.

## Application Code

In general, the instrumentation code is implemented by taking advantage of an already available logging framework. The reason is simple: it isn't easy to write a fast and flexible logging framework. Using an available framework therefore saves quite a lot of development time. In fact, the major drawback of logging is that it can slow down the application if not properly implemented. To avoid this, developers shouldn't only be careful to limit the verbosity of logging but should also implement it with an efficient logging framework.

The Apache Logging Services Project[3] provides very good examples of logging frameworks. The core of the project is log4j, a logging framework written for Java. Because of its success, having been ported to other programming languages like C++, .NET, Perl, and PHP. I'll provide an example here based on log4j and Java. For example, if you want to instrument the response time of the `handleRequest` method in the `SignonController` servlet described in Figure 2-5, you could write the following code:

```
public ModelAndView handleRequest(HttpServletRequest request,
                                  HttpServletResponse response) throws Exception
{
  if (logger == null)
  {
    logger = Log4jLoggingHelper.getLog4jServerLogger();
  }

  if (logger.isInfoEnabled())
  {
    long beginTimeMillis = System.currentTimeMillis();
  }

  ModelAndView ret = null;
  String username = request.getParameter("username");

  // here the code handling the request...
```

[3]See `http://logging.apache.org` for additional information.

```
  if (logger.isInfoEnabled())
  {
    long endTimeMillis = System.currentTimeMillis();
    logger.info("Signon(" + username + ") response time " +
                (endTimeMillis-beginTimeMillis) + " ms");
  }

  return ret;
}
```

Simply put, the instrumentation code, which is highlighted in bold, gets a timestamp when the method begins, gets a timestamp when the method ends, and then logs a message providing the name of the user and the time it took to do it. At the beginning, it also checks whether the logger has already been initialized. Notice that the `Log4jLoggingHelper` class is specific to Oracle WebLogic Server, not to log4j. This was used during the test. Even if the code snippet is straightforward, there are some important points to note:

- Start measuring the time, in this case by calling the `currentTimeMillis` method, at the very beginning of the instrumented method. You never know what situations could cause even simple initialization code to consume time.
- The logging framework should support different levels of messages. In the case of log4j, the following levels are available (in increasing order of verbosity): `fatal`, `error`, `warning`, `information`, and `debug`. You set the level by calling one of the following methods: `fatal`, `error`, `warn`, `info`, and `debug`. In other words, each level has its own method. By putting messages in different levels, you're able to explicitly choose the verbosity of the logging by enabling or disabling specific levels. This could be useful if you want to enable part of the instrumentation only while investigating a specific problem.
- Even if the logging facility is aware of which levels the logging is enabled for, it's better to not call the logging method, in the example `info`, if the logging for that specific level isn't enabled, especially to avoid the message construction overhead (and subsequent call to the garbage collector). It's usually much faster to check whether the logging for a specific level is enabled by calling a method like `isInfoEnabled` and then call the logging method only if necessary. That can make a huge difference. For example, on my test server, calling `isInfoEnabled` takes about 7 nanoseconds, while calling `info` and providing the parameters as shown in the previous code snippet takes about 265 nanoseconds (I used the class defined in `LoggingPerf.java` to measure these statistics). Another technique for reducing the overhead of instrumentation would be to remove the logging code at compile time. This is, however, not the preferred technique because it's usually not possible to dynamically recompile the code in case the instrumentation is needed. Further, as you have just seen, the overhead of a line of instrumentation that doesn't generate any message is really very small.
- In this example, the generated message would be something like "Signon(JPS1907) response time 24 ms." This is good for a human being, but if, for example, you plan on parsing the message with another program in order to check a service level agreement, a more structured form, such as XML or JSON, would be more appropriate.

## Database Calls

This section describes how to instrument database calls properly in order to give information to the database engine about the application context in which they'll be executed. Be aware that this type of instrumentation is very different from what you saw in the previous section. In fact, not only should database calls be instrumented like any other part of the application, but the database itself is also already able to generate detailed information about the database calls it executes. You'll see more of this later in Part 2. The aim of the type of instrumentation described here is to provide the database engine with information about the user and/or the application using it. This is necessary since the database engine has generally little, and often no, information about the relationship between application code, sessions, and end users. Consider the following common situations:

- The database engine doesn't know which part of the application code is executing SQL statements through a session. For example, the database engine has no clue about which module, class, or report is running a specific SQL statement through a given session.
- When the application connects to the database engine through a pool of connections opened with a technical user and proxy users aren't utilized, the end-user authentication is usually performed by the application itself. Therefore, the database engine ignores which end user is using which session.

For these reasons, the database engine provides the opportunity to dynamically associate the following attributes to a database session:

- *Client identifier*: This is a string of 64 bytes that identifies a client, albeit not unequivocally.
- *Client information*: This is a string of 64 bytes that describes the client.
- *Module name*: This is a string of 48 bytes that describes the module currently using the session.
- *Action name*: This is a string of 32 bytes that describes the action being processed.

---

■ **Caution** For sessions opened through database links, only the client identifier is automatically propagated to remote sessions. Therefore, for the other attributes, it's necessary to explicitly set them.

---

Their values are externalized through the `v$session` view and the `userenv` context. The following example, which is an excerpt of the output generated by the `session_info.sql` script, shows how to query them:

```
SQL> SELECT sys_context('userenv','client_identifier') AS client_identifier,
  2         sys_context('userenv','client_info') AS client_info,
  3         sys_context('userenv','module') AS module_name,
  4         sys_context('userenv','action') AS action_name
  5  FROM dual;

CLIENT_IDENTIFIER    CLIENT_INFO  MODULE_NAME      ACTION_NAME
-------------------- ------------ ---------------- ------------------------
helicon.antognini.ch Linux x86_64 session_info.sql test session information

SQL> SELECT client_identifier,
  2         client_info,
  3         module AS module_name,
  4         action AS action_name
  5  FROM v$session
  6  WHERE sid = sys_context('userenv','sid');

CLIENT_IDENTIFIER    CLIENT_INFO  MODULE_NAME      ACTION_NAME
-------------------- ------------ ---------------- ------------------------
helicon.antognini.ch Linux x86_64 session_info.sql test session information
```

Note that other views showing SQL statements, for example v$sql, also contain the module and action columns. One word of caution: the attributes are associated to a specific session, but a given SQL statement can be shared between sessions having different module names or action names. The values shown by the dynamic performance views are the ones that were set at hard parse time in the session that first parsed the SQL statement. This can be misleading if you aren't careful.

Now that you have seen what is available, let's take a look at how you can set these values. The first method, PL/SQL, is the only one that doesn't depend on the interface used to connect the database. As a result, it can be used in most situations. The following four—OCI, JDBC, ODP.NET, and PHP—can be utilized only along with the specific interface. Their main advantage is that the values are added to the next database call instead of generating extra round-trips, which is what a call to PL/SQL does. Thus, the overhead of setting the attributes with them is negligible.

## PL/SQL

To set the client identifier, you use the set_identifier procedure in the dbms_session package. In some situations, such as when the client identifier is used along with a global context and connection pooling, it may be necessary to clear the value associated with a given session. If this is the case, you can use the clear_identifier procedure.

To set the client information, the module name, and the action name, you use the set_client_info, set_module, and set_action procedures in the dbms_application_info package. For simplicity, the set_module procedure accepts not only the module name but also the action name.

The following PL/SQL block, which is an excerpt of the session_attributes.sql script, shows an example:

```
BEGIN
  dbms_session.set_identifier(client_id=>'helicon.antognini.ch');
  dbms_application_info.set_client_info(client_info=>'Linux x86_64');
  dbms_application_info.set_module(module_name=>'session_info.sql',
                                   action_name=>'test session information');
END;
```

## OCI

To set the four attributes, you use the OCIAttrSet function. The third parameter specifies the value. The fifth parameter specifies, by means of one of the following constants, which attribute is set:

- OCI_ATTR_CLIENT_IDENTIFIER
- OCI_ATTR_CLIENT_INFO
- OCI_ATTR_MODULE
- OCI_ATTR_ACTION

The following code snippet, which is an excerpt of the session_attributes.c file, shows how to call the OCIAttrSet function to set the client identifier:

```
text client_id[64] = "helicon.antognini.ch";
OCIAttrSet(ses,                         // session handle
           OCI_HTYPE_SESSION,           // type of handle being modified
           client_id,                   // attribute's value
           strlen(client_id),           // size of the attribute's value
           OCI_ATTR_CLIENT_IDENTIFIER,  // attribute being set
           err);                        // error handle
```

## JDBC

There are two ways to set session attributes. The legacy way uses an Oracle extension, but the contemparary way is based on the standard JDBC application programming interface. The contemporary way is available as of version 12.1 of the JDBC drivers provided by Oracle. Because it's based on the same protocol as the legacy way, it can also be used with databases of versions 10.2, 11.1 and 11.2. Note that from version 12.1 onward, the legacy way is deprecated.

### The Legacy Way

To set the client identifier, module name, and action name, you use the `setEndToEndMetrics` method provided by the `OracleConnection` interface. No support is provided to set the client information. One or more attributes are passed to the method with an array of strings. The position in the array, which is defined by the following constants, determines which attribute is set:

- `END_TO_END_CLIENTID_INDEX`
- `END_TO_END_MODULE_INDEX`
- `END_TO_END_ACTION_INDEX`

The following code snippet, which is an excerpt of the `SessionAttributes.java` file, shows how to define the array containing the attributes and how to call the `setEndToEndMetrics` method:

```
metrics = new String[OracleConnection.END_TO_END_STATE_INDEX_MAX];
metrics[OracleConnection.END_TO_END_CLIENTID_INDEX] = "helicon.cha.trivadis.com";
metrics[OracleConnection.END_TO_END_MODULE_INDEX] = "SessionAttributes.java";
metrics[OracleConnection.END_TO_END_ACTION_INDEX] = "test session information";
((OracleConnection)connection).setEndToEndMetrics(metrics, (short)0);
```

### The Contemporary Way

To set the client identifier, module name, and action name, you use the `setClientInfo` method provided by the `Connection` interface. No support is provided to set the client information. The `setClientInfo` method takes as input two `String` parameters: the name of an attribute and its value. You must specify the name as one of the following:

- `OCSID.CLIENTID`
- `OCSID.MODULE`
- `OCSID.ACTION`

The following code snippet, which is an excerpt of the `SessionAttributes12c.java` file, shows how to set the attributes through the `setClientInfo` method:

```
connection.setClientInfo("OCSID.CLIENTID", "helicon.cha.trivadis.com");
connection.setClientInfo("OCSID.MODULE", "SessionAttributes12c.java");
connection.setClientInfo("OCSID.ACTION", "test session information");
```

### ODP.NET

To set the four attributes, you use the ClientId, ClientInfo, ModuleName, and ActionName properties of the OracleConnection class. Note that except for the ClientId property, the others are available only from version 11.1.0.6.20 onward. In order to prevent pooled connections from taking over settings, the properties are set to null when the Close or Dispose method of the OracleConnection class is called. The following code snippet, which is an excerpt of the SessionAttributes.cs file, shows how to set the properties:

```
connection.ClientId = "helicon.antognini.ch";
connection.ClientInfo = "Linux x86_64";
connection ModuleName = "SessionAttributes.cs";
connection.ActionName = "test session information";
```

### PHP

To set the four attributes, you use functions provided in the PECL OCI8 extension. Each function takes two parameters as input (a connection and the attribute's value) and returns a Boolean value (TRUE on success and FALSE on failure). The four functions are:

- oci_set_client_identifier
- oci_set_client_info
- oci_set_module_name
- oci_set_action

The following code snippet, which is an excerpt of the session_attributes.php script, shows how to set all attributes:

```
oci_set_client_identifier($connection, "helicon.antognini.ch");
oci_set_client_info($connection, "Linux x86_64");
oci_set_module_name($connection, "session_attributes.php");
oci_set_action($connection, "test session information");
```

Note that these functions are only available as of OCI8 version 1.4 and when OCI8 is linked with client libraries of version 10.1 or newer.

## On to Part 2

This chapter describes the operations carried out by the database engine when parsing and executing SQL statements. Particular attention is given to the pros and cons related to using bind variables. In addition, the chapter introduces some frequently used terms and describes how to instrument the application code and the database calls.

Part 2 is devoted to answering the first two questions posed in Figure 1-4:

- Where is the time spent?
- How is the time spent?

Simply put, the three chapters in Part 2 describe approaches to finding out where the problem is and what's causing it. Because failing to fix a performance problem is usually not an option, failing to correctly answer these questions isn't an option either. Naturally, if you don't know what's causing a problem, you'll find it impossible to fix.

PART II

# Identification

*Let's work the problem, people. Let's not make things worse by guessin.'*

—Eugene F. Kranz[1]

The first thing to do when an application experiences performance problems is obvious: identify the root cause of the problem. Unfortunately, all too often this is where the real trouble starts. In a typical scenario, where everyone is looking for the source of a performance problem, developers blame the database for poor performance, and the database administrators blame both the developers for misusing the database and the storage subsystem administrators because their very expensive piece of hardware ought to provide much better performance. And as the complexity of the application and infrastructure supporting it increases, so does the mess.

The aim of the chapters in Part II is to describe the key database features that you can use to find out where and how time is spent inside the database. As you read these chapters, keep Chapter 1's advice in mind about collecting detailed information about the processing performed by the database engine *only when* an end-to-end response time measurement points out that there might be a problem in the database layer. Otherwise, you may end up analyzing the wrong component. And if you're analyzing the wrong component, you're likely to fail in identifying the cause of the performance problem that you're trying to troubleshoot. First be sure the problem lies in the database layer. Then what's in Part II can help.

The next thing to consider when dealing with a performance problem is whether you can reproduce the problem at will. If so, then everything is much easier than if you can't. If you can reproduce the problem, it should be quite easy to perform a controlled measurement to pinpoint the problem while the application is having it. That's covered in Chapter 3. If the problem can't be reproduced at will, you have two choices: either wait till the problem occurs again or look into a repository containing historical performance statistics. These two cases are covered in Chapters 4 and 5, respectively.

Independently of whether you can reproduce the problem or not, you must discover what the most time-consuming SQL statements or PL/SQL code invocations are. And for each of those time-consuming statements, you should gather any additional information that can help in understanding the problem.

This additional information often includes the execution plan, key runtime statistics like the number of processed rows and the amount of CPU utilization, and the experienced wait events. Part II doesn't describe what to do with this information—only how to find it. Parts III and IV cover what to do after you've found what you need.

[1]This quote is from the movie *Apollo 13*, directed by Ron Howard. It can be heard about three minutes after the famous line, "Houston, we have a problem."

When your analysis points to a small number of SQL statements or a limited piece of PL/SQL code, you've found the part of your application that needs optimization, and there's likely a straightforward way of fixing it. Otherwise, a response time distributed over a large number of SQL statements or over a large part of the PL/SQL code usually means the problem is due to design decisions; complete reengineering could be necessary. If the design itself isn't a problem, then it's likely that the machine running the application is undersized.

The ultimate, straightforward aim of Part II is to introduce and employ a methodology that avoids guesswork and establishes beyond any doubt where a bottleneck is in an underperforming application. Included in that aim is the gathering of essential information needed to progress into Parts III and IV.

CHAPTER 3

# Analysis of Reproducible Problems

The most efficient way to approach a reproducible problem is to take advantage of one of the available tracing and profiling features to perform a controlled measurement while an application is experiencing the problem. At first, the aim is to categorize the problem into three major classes:

- The database engine spends most of the time executing SQL statements.
- The database engine spends most of the time executing PL/SQL code.
- The database engine is (almost) idle. In other words, the bottleneck is located outside the database tier.

To categorize the problem, you start the analysis by tracing the database calls. If the analysis points to the SQL statements, then the trace files used in the first place to categorize the problem already contain all the necessary information for a detailed analysis. If the PL/SQL code is questioned, you should perform a profiling analysis of the PL/SQL code. Otherwise, the problem isn't due to the database tier, and, therefore, the analysis should continue with a profiling of the application code *not* run by the database engine.

The aim of this chapter isn't only to describe the tracing and profiling capabilities provided by Oracle Database, but also to show examples of tools you can use to support your analysis. Ultimately, the purpose here is to show you how such tools can enhance your ability to quickly and, therefore, efficiently identify performance problems.

## Tracing Database Calls

When the bottleneck is situated in the database tier, it's necessary to take a closer look at the interactions between the application and the database engine. Oracle Database is a highly instrumented piece of software, and thanks to a feature called *SQL trace*, it can provide detailed trace files containing not only a list of executed SQL statements, but also in-depth performance figures about their processing.

Figure 3-1 shows the essential phases involved in the tracing of database calls. The next sections, following an explanation of what SQL trace is, discuss each of these phases in detail.

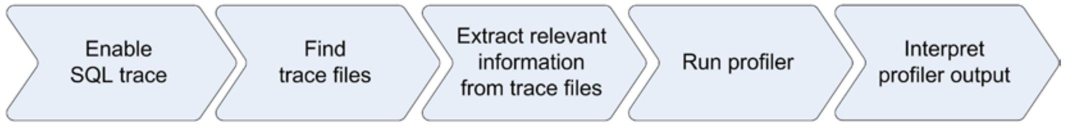

Enable SQL trace
Find trace files
Extract relevant information from trace files
Run profiler
Interpret profiler output

***Figure 3-1.*** *Essential phases involved in the tracing of database calls*

## SQL Trace

As described in Chapter 2, to process a SQL statement, the database engine (specifically, the SQL engine) carries out database calls (for example: parse, execute, and fetch). For each database call, as summarized in Figure 3-2, the SQL engine either

- does some processing itself, by using CPU;
- makes use of other resources (for example, a disk); or
- has to go through a synchronization point needed to guarantee the multiuser capabilities of the database engine (for example, a latch).

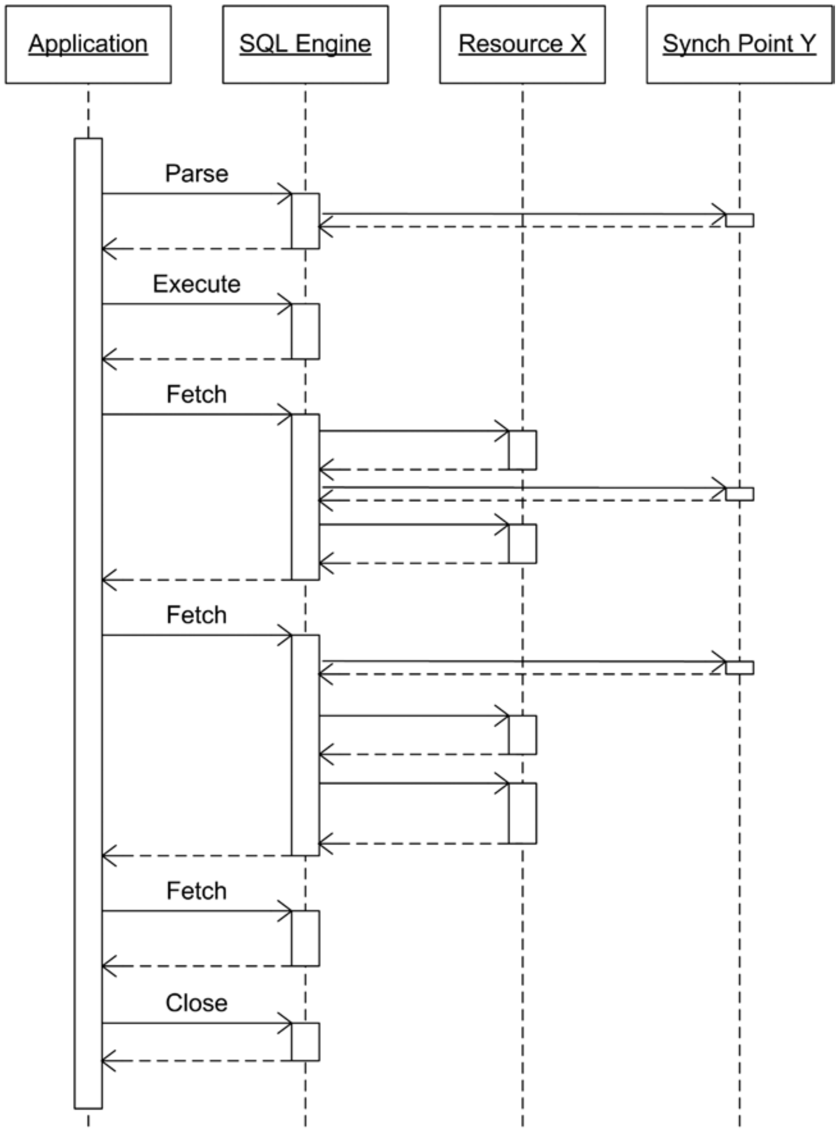


***Figure 3-2.*** *Sequence diagram describing the interactions between the SQL engine and other components*

The aim of SQL trace is twofold: first, to provide information in order to break up the response time between service time and wait time, and second, to give detailed information about the used resources and synchronization points. All this information regarding each interaction between the SQL engine and the other components is written in a trace file. Note that in Figure 3-2, the attribution of CPU, resource X, and synchronization point Y is artificial. The reason for this is to show that every call may use the database engine differently.

Although I cover this in greater detail later in this chapter, let's briefly look at an example of the kind of information provided by SQL trace and that can be extracted by a tool (in this case, TKPROF). This includes the text of the SQL statement, some execution statistics, the waits occurred during the processing, and information about the parsing phase, such as the generated execution plan. Note that such information is provided for each SQL statement executed by the application and recursively by the database engine itself:

```
SELECT CUST_ID, EXTRACT(YEAR FROM TIME_ID), SUM(AMOUNT_SOLD)
FROM SALES
WHERE CHANNEL_ID = :B1
GROUP BY CUST_ID, EXTRACT(YEAR FROM TIME_ID)

call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        1      0.00       0.00          0          0          0           0
Execute      1      0.00       0.00          0          0          0           0
Fetch      164      0.84       1.27       3472       1781          0       16348
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total      166      0.84       1.28       3472       1781          0       16348

Misses in library cache during parse: 1
Misses in library cache during execute: 1
Optimizer mode: ALL_ROWS
Parsing user id: 77  (SH)   (recursive depth: 1)
Number of plan statistics captured: 1

Rows (1st) Rows (avg) Rows (max)  Row Source Operation
---------- ---------- ----------  ------------------------------------------
     16348      16348      16348  HASH GROUP BY
    540328     540328     540328   PARTITION RANGE ALL PARTITION: 1 28
    540328     540328     540328    TABLE ACCESS FULL SALES PARTITION: 1 28

Elapsed times include waiting on following events:
  Event waited on                             Times Waited   Max. Wait  Total Waited
  ----------------------------------------   ------------  ----------  ------------
  Disk file operations I/O                              2        0.00          0.00
  db file sequential read                              29        0.00          0.00
  direct path read                                     70        0.00          0.00
  asynch descriptor resize                             16        0.00          0.00
  direct path write temp                             1699        0.02          0.62
  direct path read temp                              1699        0.00          0.00
```

As mentioned, the preceding sample is generated by a tool named TKPROF. It's not the output generated by SQL trace. In fact, SQL trace outputs text files, storing raw information about the interactions between components. Here's an excerpt of the trace file related to the previous sample. Generally, for each call or wait, there's at least one line in the trace file:

```
...
...
PARSING IN CURSOR #140105537106328 len=139 dep=1 uid=77 oct=3 lid=93 tim=1344867866442114
hv=2959931450 ad='706df490' sqlid='arc3zqqs6ty1u'
SELECT CUST_ID, EXTRACT(YEAR FROM TIME_ID), SUM(AMOUNT_SOLD) FROM SALES WHERE CHANNEL_ID = :B1
GROUP BY CUST_ID, EXTRACT(YEAR FROM TIME_ID)
END OF STMT
PARSE #140105537106328:c=1999,e=1397,p=0,cr=0,cu=0,mis=1,r=0,dep=1,og=1,plh=0, tim=1344867866442113
BINDS #140105537106328:
 Bind#0
  oacdty=02 mxl=22(21) mxlc=00 mal=00 scl=00 pre=00
  oacflg=03 fl2=1206001 frm=00 csi=00 siz=24 off=0
  kxsbbbfp=7f6cdcc6c6e0  bln=22  avl=02  flg=05
  value=3
EXEC #140105537106328:c=7000,e=7226,p=0,cr=0,cu=0,mis=1,r=0,dep=1,og=1,plh=3604305554,
tim=1344867866449493
WAIT #140105537106328: nam='Disk file operations I/O' ela= 45 FileOperation=2 fileno=4 filetype=2
obj#=69232 tim=1344867866450319
WAIT #140105537106328: nam='db file sequential read' ela= 59 file#=4 block#=5009 blocks=1 obj#=69232
tim=1344867866450423
...
...
FETCH #140105537106328:c=0,e=116,p=0,cr=0,cu=0,mis=0,r=48,dep=1,og=1,plh=3604305554,
tim=1344867867730523
STAT #140105537106328 id=1 cnt=16348 pid=0 pos=1 obj=0 op='HASH GROUP BY (cr=1781 pr=3472 pw=1699
time=1206229 us cost=9220 size=4823931 card=229711)'
STAT #140105537106328 id=2 cnt=540328 pid=1 pos=1 obj=0 op='PARTITION RANGE ALL PARTITION: 1 28
(cr=1781 pr=1773 pw=0 time=340163 us cost=1414 size=4823931 card=229711)'
STAT #140105537106328 id=3 cnt=540328 pid=2 pos=1 obj=69227 op='TABLE ACCESS FULL SALES PARTITION: 1
28 (cr=1781 pr=1773 pw=0 time=280407 us cost=1414 size=4823931 card=229711)'
CLOSE #140105537106328:c=0,e=1,dep=1,type=3,tim=1344867867730655
...
...
```

In the previous excerpt, some tokens describing the kind of information provided are highlighted in bold:

- `PARSING IN CURSOR` and `END OF STMT` enclose the text of the SQL statement.
- `PARSE`, `EXEC`, `FETCH`, and `CLOSE` for parse, execute, fetch, and close calls, respectively.
- `BINDS` for the definition and value of bind variables.
- `WAIT` for the wait events that occurred during the processing.
- `STAT` for the execution plans that occurred and associated statistics.

You can find a short description of the trace files' format in the Oracle Support note *Interpreting Raw SQL_TRACE output* (39817.1). If you're interested in a detailed description and discussion of this topic, Millsap's book *The Method R Guide to Mastering Oracle Trace Data* (CreateSpace, 2013) is worth reading.

Internally, SQL trace is based on debugging event 10046. Table 3-1 describes the supported levels, which define the amount of information provided in trace files. When SQL trace is used at a level higher than 1, it's also called *extended SQL trace*.

***Table 3-1.*** *Levels of the Debugging Event 10046*

| Level | Description |
|---|---|
| 0 | The debugging event is disabled. |
| 1 | The debugging event is enabled. For each processed database call, the following information is given: SQL statement, response time, service time, number of processed rows, number of logical reads, number of physical reads and writes, execution plan, and little additional information.<br>In version 10.2, an execution plan is written to the trace file only when the cursor it's associated with is closed. The execution statistics associated with the execution plan are values aggregated over all executions.<br>As of version 11.1, an execution plan is written to the trace file only after the first execution of every cursor. The execution statistics associated with the execution plan are thus from the first execution only. |
| 4 | As in level 1, with additional information about bind variables. Mainly, the data type, its precision, and the value used for each execution. |
| 8 | As in level 1, plus detailed information about wait time. For each wait experienced during the processing, the following information is given: the name of the wait event, the duration, and a few additional parameters identifying the resource that has been waited for. |
| 16 | As in level 1, plus the execution plan's information is written to the trace file after each execution. Available as of version 11.1 only. |
| 32 | As in level 1, but without the execution plan's information. Available as of version 11.1 only. |
| 64 | As in level 1, plus the execution plan's information might be written for executions following the first one. The condition is that, since the last write of the execution plan's information, a particular cursor has consumed at least one additional minute of DB time. This level is useful in two cases. First, when the information about the first execution isn't enough for analysing a specific issue. Second, when the overhead of writing the information about every execution (level 16) is too high. Available as of version 11.2.0.2[1]only. |

In addition to the levels described in Table 3-1, you can also combine levels 4 and 8 with every other level greater than 1. For example:

Level 12 (4 + 8): simultaneously enables level 4 and level 8.

Level 28 (4 + 8 + 16): simultaneously enables level 4, level 8 and level 16.

Level 68 (4 + 64): simultaneously enables level 4 and level 64.

The next sections describe how to enable and disable SQL trace, how to configure the environment to our best advantage, and how to find the trace files it generates.

[1]Or when a patch including the fix for bug 8328200 is installed (for example 11.2.0.1.0 Bundle Patch 7 for Exadata).

**DEBUGGING EVENTS**

A debugging event, which is identified by a numeric value, is the means used to set a type of flag in a running database engine process. The aim is to change its behavior, for example, by enabling or disabling a feature, by testing or simulating a corruption or crash, or by collecting trace or debug information. Some debugging events aren't simple flags and can be enabled at several levels. Each level has its own behavior. In some situations, the level is an address of a block or memory structure.

You should use a debugging event with care and set it only when directed to do so by Oracle Support or if you know and understand what the debugging event is going to change. Debugging events enable specific code paths. Therefore, if a problem occurs when a debugging event is set, it's worth checking whether the same problem can be reproduced without the debugging event set.

Few debugging events are documented by Oracle. If documentation exists, it's usually provided through Oracle Support notes. In other words, debugging events are generally not described in the official Oracle documentation about the database engine. You can find a complete list of the available debugging events in the file $ORACLE_HOME/rdbms/mesg/oraus.msg. Note that this file isn't distributed on all platforms. The range from 10,000 to 10,999 is reserved for debugging events.

## Enabling SQL Trace with ALTER SESSION

The ALTER SESSION statement, as described in the *SQL Language Reference* manual, can be used to enable SQL trace. Here's an example:

```
ALTER SESSION SET sql_trace = TRUE
```

Your only option using the ALTER SESSION statement is to set sql_trace to TRUE, which yields a level 1 trace. In practice, a level 1 trace is usually not enough. In most situations you need to break up the response time completely to understand where the bottleneck is. For this reason, I won't describe this method of enabling SQL trace further. Instead, I cover how to enable SQL trace at any level using the method described in Oracle Support note *EVENT: 10046 "enable SQL statement tracing (including binds/waits)"* (21154.1). To enable and disable SQL trace at any level, you set the events initialization parameter by executing the ALTER SESSION statement. The following SQL statement enables SQL trace at level 12 for the session executing it. Notice how the event number and the level are specified:

```
ALTER SESSION SET events '10046 trace name context forever, level 12'
```

The following SQL statement disables SQL trace for the session executing it. Notice that this is *not* achieved by specifying level 0:

```
ALTER SESSION SET events '10046 trace name context off'
```

You can also set the events initialization parameter by executing the ALTER SYSTEM statement. The syntax is the same as for the ALTER SESSION statement. In any case, not only is there usually no point in enabling SQL trace at the system level, but, in addition, the overhead of doing so can be huge. Also note that it takes effect only for sessions created after it's executed.

## Enabling SQL Trace with DBMS_MONITOR

Oracle Database also provides the dbms_monitor package, which you can use to enable and disable SQL trace. Not only does the package provide a way of enabling extended SQL trace at the session level, but, more importantly, you can enable and disable SQL trace based on the session attributes (see the "Database Calls" section in Chapter 2). These attributes include: client identifier, service name, module name, and action name. This means that if the application is correctly instrumented, you can enable and disable SQL trace independently of the session used to execute the database calls. Nowadays, this is particularly useful because in many situations connection pooling is used, so users aren't tied to a specific session.

When using the dbms_monitor package, you don't directly specify the levels of debugging event 10046. Instead, each procedure that enables SQL trace provides three parameters (binds, waits, and, as of version 11.1, plan_stat) that allow enabling a specific level. Use these parameters as follows:

- To enable level 4, the binds parameter has to be set to TRUE.
- To enable level 8, the waits parameter has to be set to TRUE.
- To enable level 16, the plan_stat parameter has to be set to all_executions.
- To enable level 32, the plan_stat parameter has to be set to never.
- It's not possible to enable level 64 through dbms_monitor.

The waits parameter defaults to TRUE. The binds parameter defaults to FALSE. The plan_stat parameter defaults to NULL (equivalent to first_execution). Therefore, the default level is 8.

The following sections illustrate some examples of using the dbms_monitor package for enabling and disabling SQL trace at the session, client, component, and database levels. Note that, by default, only the users with the dba role enabled are allowed to execute the procedures provided by the dbms_monitor package.

### Session Level

To enable and disable SQL trace for a session, the dbms_monitor package provides the session_trace_enable and session_trace_disable procedures, respectively.

The following PL/SQL call enables SQL trace at level 8 for the session identified by ID 127 and serial number 29:

```
dbms_monitor.session_trace_enable(session_id => 127,
                                  serial_num => 29,
                                  waits      => TRUE,
                                  binds      => FALSE,
                                  plan_stat  => 'first_execution')
```

All parameters have default values. If the two parameters identifying the session aren't specified, SQL trace is enabled for the session executing the PL/SQL call.

When SQL trace has been enabled with the session_trace_enable procedure, the sql_trace, sql_trace_waits, and sql_trace_binds columns of the v$session view are set accordingly. In addition, as of version 11.1, the sql_trace_plan_stats column is also available. Warning: this happens only when the session_trace_enable procedure is used and at least one SQL statement has been executed by the session to be traced. For example, enabling SQL trace with the previous PL/SQL call will result in the following information being given:

```
SQL> SELECT sql_trace, sql_trace_waits, sql_trace_binds, sql_trace_plan_stats
  2  FROM v$session
  3  WHERE sid = 127;
```

```
SQL_TRACE SQL_TRACE_WAITS SQL_TRACE_BINDS SQL_TRACE_PLAN_STATS
--------- --------------- --------------- --------------------
ENABLED   TRUE            FALSE           FIRST EXEC
```

The following PL/SQL call disables SQL trace for the session identified by ID 127 and serial number 29:

```
dbms_monitor.session_trace_disable(session_id => 127,
                                   serial_num => 29)
```

Be aware that both parameters have default values. If they aren't specified, SQL trace is disabled for the session executing the PL/SQL call.

If Real Application Clusters is used, the `session_trace_enable` and `session_trace_disable` procedures have to be executed on the database instance where the session resides.

## Client Level

To enable and disable SQL trace for a client, the `dbms_monitor` package provides the `client_id_trace_enable` and `client_id_trace_disable` procedures, respectively. Naturally, these procedures can be used only if the session attribute client identifier is set.

The following PL/SQL call enables SQL trace at level 12 for all sessions having the client identifier specified as a parameter:

```
dbms_monitor.client_id_trace_enable(client_id => 'helicon.antognini.ch',
                                    waits     => TRUE,
                                    binds     => TRUE,
                                    plan_stat => 'first_execution')
```

The `client_id` parameter has no default value and is case sensitive.

Because the setting is stored in the data dictionary, not only does it persist across database instance restarts, but in addition, in a RAC environment, it applies to all database instances.

The `dba_enabled_traces` and, in a 12.1 multitenant environment, `cdb_enabled_traces` views display which client identifier SQL trace has been enabled for, and which parameters were used to enable it, through the `client_id_trace_enable` procedure. For example, after enabling SQL trace with the previous PL/SQL call, the following information is given:

```
SQL> SELECT primary_id AS client_id, waits, binds, plan_stats
  2  FROM dba_enabled_traces
  3  WHERE trace_type = 'CLIENT_ID';

CLIENT_ID            WAITS BINDS PLAN_STATS
-------------------- ----- ----- ----------
helicon.antognini.ch TRUE  TRUE  FIRST_EXEC
```

The following PL/SQL call disables SQL trace for all sessions having the client identifier specified as a parameter:

```
dbms_monitor.client_id_trace_disable(client_id => 'helicon.antognini.ch')
```

The `client_id_trace_disable` procedure removes the corresponding information added to the data dictionary through the `client_id_trace_enable` procedure. The `client_id` parameter has no default value.

## Component Level

To enable and disable SQL trace for a component specified through a service name, module name, and action name, the dbms_monitor package provides the serv_mod_act_trace_enable and serv_mod_act_trace_disable procedures, respectively. To take full advantage of these procedures, you have to set the session attributes, module name, and action name.

The following PL/SQL call enables SQL trace at level 28 for all sessions using the attributes specified as a parameter:

```
dbms_monitor.serv_mod_act_trace_enable(service_name  => 'DBM11203.antognini.ch',
                                       module_name   => 'mymodule',
                                       action_name   => 'myaction',
                                       waits         => TRUE,
                                       binds         => TRUE,
                                       instance_name => NULL,
                                       plan_stat     => 'all_executions')
```

The only parameter without a default value is the first: service_name.[2] The default values of the module_name and action_name parameters are any_module and any_action, respectively. For both, NULL is a valid value. If the action_name parameter is specified, you must specify the module_name parameter as well. Failing to do so will result in an ORA-13859 being raised. If Real Application Clusters is used, with the instance_name parameter it's possible to restrict the tracing to a single database instance. By default, SQL trace is enabled for all database instances. Be aware that the parameters service_name, module_name, action_name, and instance_name are case sensitive.

Because the setting is stored in the data dictionary, it persists across database instance restarts.

As for SQL trace at the client level, the dba_enabled_traces and, in a 12.1 multitenant environment, cdb_enabled_traces views display which component SQL trace has been enabled for, and which parameters were used to enable it, through the serv_mod_act_trace_enable procedure. After enabling SQL trace with the previous PL/SQL call, you get the following information:

```
SQL> SELECT primary_id AS service_name, qualifier_id1 AS module_name,
  2         qualifier_id2 AS action_name, waits, binds, plan_stats
  3  FROM dba_enabled_traces
  4  WHERE trace_type IN ('SERVICE', 'SERVICE_MODULE', 'SERVICE_MODULE_ACTION');

SERVICE_NAME          MODULE_NAME ACTION_NAME WAITS BINDS PLAN_STATS
--------------------- ----------- ----------- ----- ----- ----------
DBM10203.antognini.ch mymodule    myaction    TRUE  TRUE  ALL_EXEC
```

Note that depending on the attributes (that is, the service name, module name, and action name) specified as parameters to enable SQL trace, the trace_type column is set to either SERVICE, SERVICE_MODULE, or SERVICE_MODULE_ACTION.

The following PL/SQL call disables SQL trace for all sessions using the session attributes specified as parameters:

```
dbms_monitor.serv_mod_act_trace_disable(service_name  => 'DBM11203.antognini.ch',
                                        module_name   => 'mymodule',
                                        action_name   => 'myaction',
                                        instance_name => NULL)
```

[2]The *service name* is a logical name associated to a database. It's configured through the service_names initialization parameter, or through the dbms_service package. One database may have multiple service names.

The serv_mod_act_trace_disable procedure removes the corresponding information added to the data dictionary through the serv_mod_act_trace_enable procedure. All parameters have the same default values and behavior as for the serv_mod_act_trace_enable procedure.

### Database Level

For enabling and disabling SQL trace for all sessions that connect to a database (except those created by background processes), the dbms_monitor package provides the database_trace_enable and database_trace_disable procedures, respectively.

The following PL/SQL call enables SQL trace at level 12 for a single database instance:

```
dbms_monitor.database_trace_enable(waits         => TRUE,
                                   binds         => TRUE,
                                   instance_name => 'DBM11203',
                                   plan_stat     => 'first_execution')
```

All parameters have default values. In the case of Real Application Clusters, by using the instance_name parameter, restricting the tracing to a single database instance is possible. The value for a specific database instance is available in the instance_name column of the gv$instance view. If the instance_name parameter is set to NULL, which is also the default value, SQL trace is enabled for all database instances. Again, note that the instance_name parameter is case sensitive.

Because the setting is stored in the data dictionary, it persists across database instance restarts.

As for SQL trace at the client and component levels, the dba_enabled_traces and, in a 12.1 multitenant environment, cdb_enabled_traces views display which database instance SQL trace has been enabled for, and which parameters it's been enabled with, through the database_trace_enable procedure. For example, after enabling SQL trace with the previous PL/SQL call, the following information is given:

```
SQL> SELECT instance_name, waits, binds, plan_stats
  2  FROM dba_enabled_traces
  3  WHERE trace_type = 'DATABASE';

INSTANCE_NAME WAITS BINDS PLAN_STATS
------------- ----- ----- ----------
DBM11203      TRUE  TRUE  FIRST_EXEC
```

The following PL/SQL call disables SQL trace for a database by removing from the database dictionary the corresponding information added through the database_trace_enable procedure:

```
dbms_monitor.database_trace_disable(instance_name => 'DBM11203')
```

Be aware that it doesn't disable SQL trace enabled at the session, client, or component level. If the instance_name parameter is set to NULL, which is also the default value, SQL trace is disabled for all database instances.

## Enabling SQL Trace with DBMS_SESSION

As pointed out in the previous section, by default the access to the dbms_monitor package is restricted. If you want to enable or disable SQL trace for the session you're connected with, but neither have the privilege to execute the dbms_monitor package nor want to use the ALTER SESSION statement (for example, because its syntax isn't easy to remember), you can use the dbms_session package.

The dbms_session package contains two procedures, session_trace_enable and session_trace_disable, that have the same functionality as those with the same names but provided by dbms_monitor. The only difference is that the ones in dbms_session can only enable and disable SQL trace for the session you're connected with. As a result, any user having the ALTER SESSION privilege can use them.

The following example illustrates how to enable and disable SQL trace with dbms_session. Notice that the v$session view provides output indicating that SQL trace has been enabled:

```
SQL> BEGIN
  2    dbms_session.session_trace_enable(waits     => TRUE,
  3                                      binds     => TRUE,
  4                                      plan_stat => 'all_executions');
  5  END;
  6  /

SQL> SELECT sql_trace, sql_trace_waits, sql_trace_binds, sql_trace_plan_stats
  2  FROM v$session
  3  WHERE sid = sys_context('userenv','sid');

SQL_TRACE SQL_TRACE_WAITS SQL_TRACE_BINDS SQL_TRACE_PLAN_STATS
--------- --------------- --------------- --------------------
ENABLED   TRUE            TRUE            ALL EXEC

SQL> BEGIN
  2    dbms_session.session_trace_disable;
  3  END;
  4  /
```

## Triggering SQL Trace

In the previous sections, you saw different methods of enabling and disabling SQL trace. In the simplest case, you manually execute the shown SQL statements or PL/SQL calls in SQL*Plus. Sometimes, however, it's necessary to automatically trigger SQL trace. *Automatically* here means that code must be added somewhere.

The simplest approach is to create a logon trigger at the database level. To avoid enabling SQL trace for all users, I usually suggest creating a role (named sql_trace in the following example) and temporarily granting it only to the user utilized for the measurement. The following example is an excerpt of the sql_trace_trigger.sql script:

```
CREATE ROLE sql_trace;

CREATE OR REPLACE TRIGGER enable_sql_trace AFTER LOGON ON DATABASE
BEGIN
  IF (dbms_session.is_role_enabled('SQL_TRACE'))
  THEN
    EXECUTE IMMEDIATE 'ALTER SESSION SET timed_statistics = TRUE';
    EXECUTE IMMEDIATE 'ALTER SESSION SET max_dump_file_size = unlimited';
    dbms_session.session_trace_enable;
  END IF;
END;
```

Naturally, it's also possible to define the trigger for a single schema or perform other checks based, for example, on the userenv context. Note that in addition to enabling SQL trace, it's good practice to set the other initialization parameters related to SQL trace as well (more about them later in this chapter).

■ **Note** The ALTER SESSION privilege required to execute the previous trigger can't be granted through a role. Instead, it has to be granted directly to the user executing the trigger.

Another approach is to add code enabling SQL trace directly in the application. Some kind of parameterization triggering that code would need to be added as well. A command-line parameter for a fat-client application or an additional HTTP parameter for a web application are examples of this.

## Timing Information in Trace Files

The `timed_statistics` initialization parameter, which can be set to either `TRUE` or `FALSE`, controls the availability of timing information such as the elapsed time and CPU time in the trace files. If it's set to `TRUE`, timing information is added to the trace files. If it's set to `FALSE`, they should be missing; however, depending on the platform you're working on, they could be partially available as well. The default value of `timed_statistics` depends on another initialization parameter: `statistics_level`. If `statistics_level` is set to `basic`, `timed_statistics` defaults to `FALSE`. Otherwise, `timed_statistics` defaults to `TRUE`.

Generally speaking, if timing information isn't available, the trace files are useless. So, before enabling SQL trace, make sure `timed_statistics` is set to `TRUE`. You can do this, for example, by executing the following SQL statement:

```
ALTER SESSION SET timed_statistics = TRUE
```

### DYNAMIC INITIALIZATION PARAMETERS

Some initialization parameters are static, and others are dynamic. When they're dynamic, it means they can be changed without bouncing the database instance. Among the dynamic initialization parameters, some of them can be changed only at the session level, some only at the system level, and others at the session and system levels. To change an initialization parameter at the session and system levels, you use the `ALTER SESSION` and `ALTER SYSTEM` statements, respectively. Initialization parameters changed at the system level take effect immediately or only for sessions created after the modification. The `v$parameter` view, or more precisely the `isses_modifiable` and `issys_modifiable` columns, provide information about which situation an initialization parameter can be changed in.

## Limiting the Size of Trace Files

Usually, you're not interested in limiting the size of trace files. If it's necessary to do so, however, it's possible to set at the session or system level the `max_dump_file_size` initialization parameter. A numerical value followed by a *K* or *M* suffix specifies, in kilobytes or megabytes, the maximum trace file size. If no limit is wanted, as shown in the following SQL statement, you can set the initialization parameter to the value `unlimited`:

```
ALTER SESSION SET max_dump_file_size = 'unlimited'
```

As of version 11.1, when the limit is reached, a message like the following is written to the alert log:

```
Non critical error ORA-48913 caught while writing to trace file "/u00/app/oracle/diag/rdbms/
dbm11203/DBM11203/trace/DBM11203_ora_6777.trc"
Error message: ORA-48913: Writing into trace file failed, file size limit [512000] reached
Writing to the above trace file is disabled for now on...
```

## Finding Trace Files

Trace files are produced by database engine server processes running on the database server. This means they're written to a disk accessible from the database server. In version 10.2, depending on the type of the process producing the trace files, they're written in two distinct directories:

- Dedicated server processes create trace files in the directory configured through the user_dump_dest initialization parameter.
- Background processes create trace files in the directory configured through the background_dump_dest initialization parameter.

The process type is available in the type column of the v$session view. Note that, strangely enough, not all background processes are listed in the v$bgprocess view.

As of version 11.1, with the introduction of the Automatic Diagnostic Repository, the user_dump_dest and background_dump_dest initialization parameters are deprecated in favor of the diagnostic_dest initialization parameter. Because the new initialization parameter sets the base directory only, you can use the v$diag_info view to get the exact location of the trace files. The following queries show the difference between the value of the initialization parameter and the location of the trace files:

```
SQL> SELECT value FROM v$parameter WHERE name = 'diagnostic_dest';

VALUE
---------------
/u00/app/oracle

SQL> SELECT value FROM v$diag_info WHERE name = 'Diag Trace';

VALUE
-------------------------------------------------
/u00/app/oracle/diag/rdbms/dbm11203/DBM11203/trace
```

Be aware that in a 12.1 multitenant environment, the user_dump_dest, background_dump_dest and diagnostic_dest initialization parameters can't be set at the PDB level.

The name of the trace file itself used to be version and platform dependent. In recent versions, however, it has the following structure:

```
{instance name}_{process name}_{process id}.trc
```

Here's a breakdown:

instance name: This is the value of the instance_name initialization parameter. Notice that especially in a Real Application Clusters environment, this is different from the db_name initialization parameter. It's available in the instance_name column of the gv$instance view.

process name: This is the lowercase value of the name of the process that's producing the trace file. For dedicated server processes, the name ora is used. For shared server processes, it's found in the name column of either the v$dispatcher or v$shared_server view. For parallel slave processes, it's found in the server_name column of the v$px_process view. For most other background processes, it's found in the name column of the v$bgprocess view.

process id: This is the process identifier (thread identifier in Windows) at the operating system level. Its value is found in the spid column of the v$process view.

Based on the information provided here, it's possible to write a query like the one you can find in the map_session_to_tracefile.sql script. Such a query is useful only in version 10.2, though. As of version 11.1, it's much easier to query either the v$diag_info or v$process views, as shown in the following examples:

```
SQL> SELECT value
  2  FROM v$diag_info
  3  WHERE name = 'Default Trace File';

VALUE
--------------------------------------------------------------------
/u00/app/oracle/diag/rdbms/dba111/DBA111/trace/DBA111_ora_23731.trc

SQL> SELECT p.tracefile
  2  FROM v$process p, v$session s
  3  WHERE p.addr = s.paddr
  4  AND s.sid = sys_context('userenv','sid');

TRACEFILE
--------------------------------------------------------------------
/u00/app/oracle/diag/rdbms/dba111/DBA111/trace/DBA111_ora_23731.trc
```

Note that the v$diag_info view provides information for the current session only.

## DO TRACE FILES CONTAIN CONFIDENTIAL INFORMATION?

By default, trace files aren't accessible to everyone. This is good because they may contain confidential information. In fact, both the SQL statements, which may contain data (literals), and the values of bind variables end up in trace files. Basically, this means that every piece of data stored in the database could be written to a trace file as well.

For example, on Unix/Linux database servers, the trace files belong to the user and group running the database engine binaries and by default have 0640 as privileges. In other words, only users in the same group as the user running the database engine can read the trace files.

There is, however, really no good reason for preventing those that already have access to the data in the database from having access to the trace files, if they're required to perform a task. In fact, from a security point of view, trace files are a useful source of information only for those without access to the database. For this reason, the database engine provides an undocumented initialization parameter named _trace_files_public. Per default, it's set to FALSE. If set to TRUE, the trace files are made readable to everyone having access to the database server. Because the initialization parameter isn't dynamic, a database instance bounce is necessary to change its value. Be also aware that in a 12.1 multitenant environment, the _trace_files_public initialization parameter can't be set at the PDB level.

For example, on Unix/Linux with _trace_files_public set to TRUE, the default privileges become 0644. This way, all users that have access to the database server can read the trace files.

From a security point of view, setting the _trace_files_public initialization parameter to TRUE is problematic only when the access to the database server isn't restricted. In providing simple access to the trace files, it's also common to share the directories containing them via SMB, via NFS, or through an HTTP interface. In any case, and for obvious reasons, asking a DBA to manually send a trace file every time one is needed should be avoided as much as possible.

To find the right trace file easily, using the `tracefile_identifier` initialization parameter is also possible. In fact, with that initialization parameter, you can add a custom identifier of up to 255 characters to the trace file name. With it, the trace file name structure becomes the following:

```
{instance name}_{process name}_{process id}_{tracefile identifier}.trc
```

The `tracefile_identifier` initialization parameter can only be set at the session level and only with dedicated server processes. It's also worth noting that every time a session dynamically changes the value of that initialization parameter, a new trace file is automatically created. The value of the `tracefile_identifier` initialization parameter is available in the `traceid` column of the `v$process` view. Be careful in version 10.2, though: this is true only for the very same session that set the initialization parameter. All other sessions see the value `NULL`.

Now that you've seen what SQL trace is and how to configure, enable, and disable it, and where to find the trace files it generates, let's discuss their structure and some tools used to analyze, and consequently leverage, their content.

## Structure of the Trace Files

A trace file contains information about the database calls executed by a specific process. Actually, when the process identifier is reused at the operating system level, a trace file may contain information from several processes as well. Because a process may be used from different sessions (for example, for shared servers or parallel slave processes) and each session may have different session attributes (for example, module name and action name), a trace file can be separated into several logical sections. Figure 3-3 provides an example (both trace files are available along with the other files of this chapter).

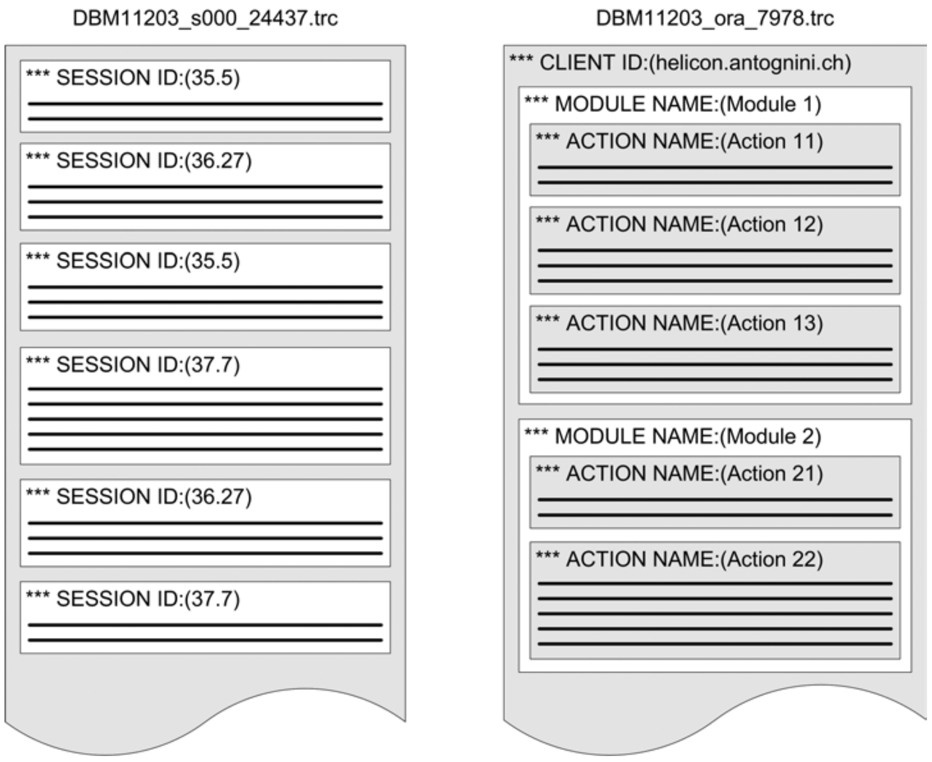

DBM11203_s000_24437.trc
*** SESSION ID:(35.5)
*** SESSION ID:(36.27)
*** SESSION ID:(35.5)
*** SESSION ID:(37.7)
*** SESSION ID:(36.27)
*** SESSION ID:(37.7)
DBM11203_ora_7978.trc
*** CLIENT ID:(helicon.antognini.ch)
*** MODULE NAME:(Module 1)
*** ACTION NAME:(Action 11)
*** ACTION NAME:(Action 12)
*** ACTION NAME:(Action 13)
*** MODULE NAME:(Module 2)
*** ACTION NAME:(Action 21)
*** ACTION NAME:(Action 22)

***Figure 3-3.*** *A trace file may be composed of several logical sections. On the left, a trace file of a shared server containing information from three sessions. On the right, a trace file of a dedicated server containing information from one client with two modules and five actions*

The structure of the trace file shown on the right in Figure 3-3 may be generated with the following PL/SQL block, provided you have previously enabled SQL trace:

```
DECLARE
  l_dummy VARCHAR2(10);
BEGIN
  dbms_session.set_identifier(client_id => 'helicon.antognini.ch');
  dbms_application_info.set_module(module_name => 'Module 1',
                                   action_name => 'Action 11');
  -- code module 1, action 11
  SELECT 'Action 11' INTO l_dummy FROM dual;
  dbms_application_info.set_module(module_name => 'Module 1',
                                   action_name => 'Action 12');
  -- code module 1, action 12
  SELECT 'Action 12' INTO l_dummy FROM dual;
  dbms_application_info.set_module(module_name => 'Module 1',
                                   action_name => 'Action 13');
  -- code module 1, action 13
  SELECT 'Action 13' INTO l_dummy FROM dual;
  dbms_application_info.set_module(module_name => 'Module 2',
                                   action_name => 'Action 21');
  -- code module 2, action 21
  SELECT 'Action 21' INTO l_dummy FROM dual;
  dbms_application_info.set_module(module_name => 'Module 2',
                                   action_name => 'Action 22');
  -- code module 2, action 22
  SELECT 'Action 22' INTO l_dummy FROM dual;
END;
```

The tags beginning with the three stars (***), used in Figure 3-3 to mark a section, are the ones used in the trace files. The difference with the trace files is that not only does the database engine repeat some of them for each section, but in addition, a timestamp is added. The following trace file snippet shows an example of content generated by the previous PL/SQL block:

```
*** CLIENT ID:(helicon.antognini.ch) 2012-11-30 10:05:05.531
*** MODULE NAME:(Module 1) 2012-11-30 10:05:05.531
*** ACTION NAME:(Action 11) 2012-11-30 10:05:05.531
...
...
*** MODULE NAME:(Module 1) 2012-11-30 10:05:05.532
*** ACTION NAME:(Action 12) 2012-11-30 10:05:05.532
...
...
*** MODULE NAME:(Module 1) 2012-11-30 10:05:05.533
*** ACTION NAME:(Action 13) 2012-11-30 10:05:05.533
...
...
*** MODULE NAME:(Module 2) 2012-11-30 10:05:05.533
*** ACTION NAME:(Action 21) 2012-11-30 10:05:05.533
...
...
*** MODULE NAME:(Module 2) 2012-11-30 10:05:05.534
*** ACTION NAME:(Action 22) 2012-11-30 10:05:05.534
```

These logical session tags are very useful because, thanks to them, it's possible to extract information that's relevant to your needs. For example, if you're investigating a performance problem related to a specific action, you can isolate the part of the trace file related to it. You can do this using the tool described in the next section, TRCSESS.

## Using TRCSESS

You can use the command-line tool TRCSESS to extract part of the information contained in one or more trace files, based on the logical sections described in the previous section. If you run TRCSESS without specifying any argument as input, you get a complete list of TRCSESS's arguments including a short description:

```
trcsess [output=<output file name >]  [session=<session ID>] [clientid=<clientid>]
        [service=<service name>] [action=<action name>] [module=<module name>]
        <trace file names>

output=<output file name> output destination default being standard output.
session=<session Id> session to be traced.
Session id is a combination of session Index & session serial number e.g. 8.13.
clientid=<clientid> clientid to be traced.
service=<service name> service to be traced.
action=<action name> action to be traced.
module=<module name> module to be traced.
<trace_file_names> Space separated list of trace files with wild card '*' supported.
```

As you can see, it's possible to specify a session, a client identifier, a service name, a module name, and an action name as an argument. For example, to extract the information about *Action 12* from the `DBM11203_ora_7978.trc` trace file and write the output in a new file named `action12.trc`, you can use the following command:

```
trcsess output=action12.trc action="Action 12" DBM11203_ora_7978.trc
```

Be aware that the `clientid`, `service`, `action`, and `module` arguments are case sensitive.

## Profilers

Once you've identified the correct trace files, or possibly cut off part of some of them with TRCSESS, it's time to analyze the content. For this purpose, you use a *profiler*. Its aim is to generate a formatted output based on the content of raw trace files. Oracle distributes such a profiler with both the database and client binaries. Its name is TKPROF (which stands for *Trace Kernel PROFiler*). Even if the output it provides can be useful in several situations, sometimes it's not adequate for quick identification of performance problems. Strangely, Oracle underestimates the importance of such a tool and, consequently, has only marginally improved it since its introduction in Oracle7. A number of commercial and freeware profilers are available, however. I've also developed my own profiler, which is freeware, named TVD$XTAT. Other profilers you may want to consider are OraSRP,[3] Method R Profiler, and Method R Tools suite.[4] Even Oracle (through Oracle Support) proposes another profiler, named Trace Analyzer.[5]

The next two sections describe two of these profilers. First, I cover TKPROF. In spite of its deficiencies, it's the only one that's always available. In fact, you aren't allowed in all situations to install another profiler on a database server

[3]See `http://www.oracledba.ru/orasrp` for information.
[4]See `http://method-r.com` for information.
[5]See Oracle Support note *TRCANLZR (TRCA): SQL_TRACE/Event 10046 Trace File Analyzer - Tool for Interpreting Raw SQL Traces* (224270.1) for additional information.

or download trace files to another machine. In such situations, it can be useful. After covering TKPROF, I describe my own profiler. The explanations are based on a trace file generated during the execution of the following PL/SQL block:

```
DECLARE
  l_count INTEGER;
BEGIN
  FOR c IN (SELECT extract(YEAR FROM d), id, pad
            FROM t
            ORDER BY extract(YEAR FROM d), id)
  LOOP
    NULL;
  END LOOP;
  FOR i IN 1..10
  LOOP
    SELECT count(n) INTO l_count
    FROM t
    WHERE id < i*123;
  END LOOP;
END;
```

## Using TKPROF

TKPROF is a command-line tool. Its main purpose is to take a raw trace file as input and generate a formatted text file as output. It can also generate a SQL script to load the data in a database, although this feature is hardly ever used.

The simplest analysis is performed by just specifying an input and an output file. In the following example, the input file is `DBM11106_ora_6334.trc`, and the output file is `DBM11106_ora_6334.txt`:

```
tkprof DBM11106_ora_6334.trc DBM11106_ora_6334.txt
```

Even if the default extension of the output file is `prf`, I personally always use `txt`. In my view, it's better to use extensions that mean something to everybody and are usually correctly recognized by any operating system.

An analysis without specifying further arguments is helpful only when analyzing very small trace files. In most situations, to get a better output, you must specify several arguments.

### TKPROF Arguments

If you run TKPROF without arguments, you get a complete list of its arguments with a short description for each of them:

```
Usage: tkprof tracefile outputfile [explain= ] [table= ]
              [print= ] [insert= ] [sys= ] [sort= ]
  table=schema.tablename   Use 'schema.tablename' with 'explain=' option.
  explain=user/password    Connect to ORACLE and issue EXPLAIN PLAN.
  print=integer    List only the first 'integer' SQL statements.
  aggregate=yes|no
  insert=filename  List SQL statements and data inside INSERT statements.
  sys=no           TKPROF does not list SQL statements run as user SYS.
  record=filename  Record non-recursive statements found in the trace file.
  waits=yes|no     Record summary for any wait events found in the trace file.
  sort=option      Set of zero or more of the following sort options:
    prscnt  number of times parse was called
    prscpu  cpu time parsing
    prsela  elapsed time parsing
```

```
prsdsk  number of disk reads during parse
prsqry  number of buffers for consistent read during parse
prscu   number of buffers for current read during parse
prsmis  number of misses in library cache during parse
execnt  number of execute was called
execpu  cpu time spent executing
exeela  elapsed time executing
exedsk  number of disk reads during execute
exeqry  number of buffers for consistent read during execute
execu   number of buffers for current read during execute
exerow  number of rows processed during execute
exemis  number of library cache misses during execute
fchcnt  number of times fetch was called
fchcpu  cpu time spent fetching
fchela  elapsed time fetching
fchdsk  number of disk reads during fetch
fchqry  number of buffers for consistent read during fetch
fchcu   number of buffers for current read during fetch
fchrow  number of rows fetched
userid  userid of user that parsed the cursor
```

The function of each argument is as follows:

- `explain` instructs TKPROF to generate an execution plan for each SQL statement found in the trace file. This is done by executing the `EXPLAIN PLAN` statement (see Chapter 10 for detailed information about this SQL statement). Obviously, to execute a SQL statement, a connection to a database is needed. Consequently, the argument is used to specify the user, password, and, if needed, connect string. The accepted formats are `explain=user/password@connect_string` and `explain=user/password`. Be aware that in order to maximize your chances of getting the right execution plans, you should specify a user with access to the same objects and make sure all query optimizer initialization parameters are set to the same value as the one used to generate the trace file. You should also be wary of initialization parameters changed at runtime by the application or with logon triggers. It goes without saying that if you can use the same user, it's even better. In any case, even if all the previous conditions are met, because the execution plans generated by the `EXPLAIN PLAN` statement don't necessarily match the real ones (the reasons are explained in Chapter 10), I don't recommend specifying the `explain` argument. If an invalid user, password, or connect string is specified, the trace file is processed without any interactive error message. Instead, an error like the following will be found in the output file just after the header:

  ```
  error connecting to database using: scott/lion
  ORA-01017: invalid username/password; logon denied
  EXPLAIN PLAN option disabled.
  ```

- `table` is used only together with the `explain` argument. Its purpose is, in fact, to specify which table is used by the `EXPLAIN PLAN` statement to generate the execution plans. Usually you can avoid specifying it because TKPROF automatically creates and drops a plan table named `prof$plan_table` in the schema used for the analysis. In any case, if the user can't create tables (for example, because the `CREATE TABLE` privilege is lacking), then the `table` argument must be specified. For example, to specify that the `plan_table` table owned by the `system` user must be used, the argument must be set to `table=system.plan_table`. The user performing the analysis must have `SELECT`, `INSERT`, and `DELETE` privileges on the specified table. Also, in this case, errors are made available only in the output file.

- `print` is used to limit the number of SQL statements provided in the output file. Per default there's no limit. It makes sense to specify this argument only together with the `sort` argument (described shortly), to print only the top SQL statements. For example, to get only 10 SQL statements, the argument must be set to `print=10`.
- `aggregate` specifies how TKPROF handles SQL statements having the same text. By default (`aggregate=yes`), all information belonging to a specific SQL statement is aggregated. Version 11.2 adds the further requirement that the execution plan needs to match as well. Thus, in version 11.2, the default is to aggregate information for each execution plan of a given SQL statement. This aggregation is done independently of the number of SQL statements present in the trace file.Therefore, as with any aggregation, there might be a loss of information. Even if the default is good in many cases, it's sometimes better to specify `aggregate=no` and be able to take a look at single SQL statements.
- `insert` instructs TKPROF to generate a SQL script that can be used to store all information in a database. The name of the SQL script is specified by the argument itself, as in `insert=load.sql`.
- `sys` specifies whether SQL statements executed by the `sys` user (typically, recursive queries against the data dictionary during parse operations) are written to the output file. The default value is `yes`, but most of the time I prefer to set it to `no` to avoid having unnecessary information in the output file. It's unnecessary because you usually have no control over the SQL statements executed recursively by the `sys` user.
- `record` instructs TKPROF to generate a SQL script containing all nonrecursive statements found in the trace file. The name of the SQL script is specified by the argument itself (for example, `record=replay.sql`). According to the documentation, this feature could be used to manually replay the SQL statements. Because bind variables aren't handled, this is usually not possible.
- `waits` determines whether information about wait events is added in the output file. Per default, it's added. Personally, I see no good reason for specifying `waits=no` and consequently not having the very important wait events in the output file.
- `sort` specifies the order in which the SQL statements are written to the output file. Per default it's the order in which they're found in the trace file. Basically, by specifying one of the proposed options, you can sort the output according to resource utilization (for example, the number of calls, CPU time, and number of physical reads) or response time (that is, the elapsed time). As you can see for most options (for example, the elapsed time), one value for each type of database call is available: for example, `prsela` for the time spent parsing a cursor, `exeela` for the time spent executing a cursor, and `fchela` for the time spent fetching rows from a cursor. Even if you have many choices and combinations, there's only one sort order that's really useful for investigating performance problems: response time. Therefore, you should specify `sort=prsela,exeela,fchela`. When you specify a comma-separated list of values, TKPROF sums the value of the options passed as arguments. This occurs even if they're based on different units of measurement. Note that when a trace file contains several sessions and the argument `aggregate=no` is specified, the SQL statements are sorted independently for each session.

Based on the information just provided, I personally usually run TKPROF with the arguments shown in the following example:

```
tkprof {input trace file} {output file} sys=no sort=prsela,exeela,fchela
```

Now that you have seen how to run TKPROF, let's take a look at the output file it generates.

## Interpreting TKPROF Output

The analysis was done by specifying the following arguments:

```
tkprof DBM11203_ora_28030.trc DBM11203_ora_28030.txt
       sort=prsela,exeela,fchela print=4 explain=chris/ian aggregate=no
```

Note that this isn't the way you were just advised to do it. This is only to show you a specific output. Both the trace file and the output file are available along with the other files for this chapter.

The output file begins with a header. Most of its information is static. Nevertheless, there's useful information in it: the name of the trace file, the value of the `sort` argument used for the generation of the output file, and a line that identifies the traced session. This last bit of information is available only because the `aggregate=no` argument was specified. Note that when a trace file contains multiple sessions and the `aggregate=no` argument is specified, this header is repeated and used as a separator between the SQL statements belonging to different sessions:

```
TKPROF: Release 11.2.0.3.0 - Development on Fri Nov 30 23:45:57 2012

Copyright (c) 1982, 2011, Oracle and/or its affiliates.  All rights reserved.
Trace file: DBM11203_ora_28030.trc
Sort options: prsela  exeela  fchela
********************************************************************************
count    = number of times OCI procedure was executed
cpu      = cpu time in seconds executing
elapsed  = elapsed time in seconds executing
disk     = number of physical reads of buffers from disk
query    = number of buffers gotten for consistent read
current  = number of buffers gotten in current mode (usually for update)
rows     = number of rows processed by the fetch or execute call
--------------------------------------------------------------------------------

*** SESSION ID: (156.29) 2012-11-30 23:21:45.691
```

Any error that occurred while connecting to the database or generating the execution plans is added just after this header.

After the header, the following information is given for every SQL statement: the text of the SQL statement, the execution statistics, information about parsing, the execution plan, and the wait events. The execution plan and wait events are reported only if they're stored in the trace file. Remember, in version 10.2, an execution plan is written to the trace file only when the cursor it's associated with is closed. This means that if an application reuses cursors without closing them, no execution plan will be written in the trace file for the reused cursors.

The text of the SQL statement in some situations is formatted. Unfortunately, the code responsible for this operation doesn't provide correct formatting in all situations. For instance, in this case the `FROM` keyword of the `extract` function is confused with the `FROM` clause of the `SELECT` statement. Note that the identifier of the SQL statement is available only as of version 11.1.0.6, and the execution plan hash value only as of version 11.1.0.7:

```
SQL ID: 7wd0gdwwgph1r Plan Hash: 961378228

SELECT EXTRACT(YEAR
FROM
 D), ID, PAD FROM T ORDER BY EXTRACT(YEAR FROM D), ID
```

The execution statistics, aggregated by type of database call, provide data in a tabular form. For each of them, the following performance figures are given:

- `count` is the number of times database calls were executed.
- `cpu` is the total CPU time, in seconds, spent processing database calls.
- `elapsed` is the total elapsed time, in seconds, spent processing database calls. If this value is higher than CPU time, the section about wait events found below the execution statistics provides information about the resources or synchronization points waited for.
- `disk` is the number of blocks read with physical reads. Be careful—this isn't the number of physical I/O operations. If this value is larger than the number of logical reads (disk > query + current), it means that blocks spilled into the temporary tablespace. In this case, you can see that at least 33,017 blocks (71,499–38,474–8) were read from it. This fact is confirmed later by the statistics of row source operations and wait events.
- `query` is the number of blocks read with logical reads in consistent mode. Usually, this type of logical read is used by queries.
- `current` is the number of blocks read with logical reads in current mode. Usually this type of logical read is used by `INSERT`, `DELETE`, `MERGE`, and `UPDATE` statements to modify blocks.
- `rows` is the number of rows processed. For queries, this is the number of fetched rows. For `INSERT`, `DELETE`, `MERGE`, and `UPDATE` statements, this is the number of affected rows. In this case, it's worth noting that 1,000,000 rows were fetched in 10,001 fetch calls. This means that on average, each call fetched about 100 rows. Note that 100 is the prefetch size used in PL/SQL. (Chapter 15 provides detailed information about the prefetch size.)

```
call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        1      0.00       0.00          0          0          0           0
Execute      1      0.00       0.00          0          0          0           0
Fetch    10001      6.49      11.92      71499      38474          8     1000000
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total    10003      6.49      11.92      71499      38474          8     1000000
```

The following lines summarize basic information about parsing. The first two values (`Misses in library cache`) provide the number of hard parses that occurred during parse and execute calls. If no hard parse occurred during execute calls, that specific line is missing. The optimizer mode and the user who parsed the SQL statement are shown. Note that the name of the user, in this case `chris`, is provided only when the `explain` argument is specified. Otherwise, only the user identifier (in this case 34) is shown. The last piece of information is the recursive depth. It's provided only for recursive SQL statements. SQL statements directly executed by an application have a depth of 0. A depth of *n* (in this case 1) simply means that another SQL statement with depth *n-1* (in this case 0) has executed this one. In our sample, the SQL statement at depth 0 is a PL/SQL block that was executed by SQL*Plus:

```
Misses in library cache during parse: 1
Misses in library cache during execute: 1
Optimizer mode: ALL_ROWS
Parsing user id: 34    (CHRIS)      (recursive depth: 1)
```

After the general information about parsing, you might see the execution plan. Actually, if the `explain` argument is specified, it may be possible to see two of them. The first one, called `Row Source Operation`, is the execution plan written in the trace file by the server process. The second one, called `Execution Plan`, is generated by TKPROF only when the `explain` argument is specified. Because it's generated later, it doesn't necessarily match the first one. In any case, if you see a difference between the two, the first is the correct one.

Chapter 10 describes how to read an execution plan; here I'm describing only the particularities of TKPROF. Both execution plans provide, for the first execution found in the trace file, the number of rows returned (not processed—be careful) by each operation in the execution plan. In addition to information about the first execution, version 11.2.0.2 and higher also provides the average and maximum number of rows returned over all executions. The number of executions itself is provided by the `Number of plan statistics captured` value.

For each row source operation, the following runtime statistics might also be provided:

- `cr` is the number of blocks read with logical reads in consistent mode.
- `pr` is the number of blocks read with physical reads from the disk.
- `pw` is the number of blocks written with physical writes to the disk.
- `time` is the total elapsed time in microseconds spent processing the operation. Be aware that the value provided by this statistic isn't always very precise. That's because, to reduce the overhead, the server process may use sampling to measure it.
- `cost` is the estimated cost of the operation. This value is available only as of version 11.1.
- `size` is the estimated amount of data (in bytes) returned by the operation. This value is available only as of version 11.1.
- `card` is the estimated number of rows returned by the operation. This value is available only as of version 11.1.

```
Number of plan statistics captured: 1

Rows (1st) Rows (avg) Rows (max)  Row Source Operation
---------- ---------- ----------  ---------------------------------------------------
   1000000    1000000    1000000  SORT ORDER BY (cr=38474 pr=71499 pw=33035 time=11123996 us
                                                 cost=216750 size=264000000 card=1000000)
   1000000    1000000    1000000   TABLE ACCESS FULL T (cr=38474 pr=38463 pw=0 time=5674541 us
                                                        cost=21 size=264000000 card=1000000)

Rows     Execution Plan
-------  ---------------------------------------------------
      0  SELECT STATEMENT   MODE: ALL_ROWS
1000000   SORT (ORDER BY)
1000000    TABLE ACCESS   MODE: ANALYZED (FULL) OF 'T' (TABLE)
```

Note that the runtime statistics, except for the query optimizer estimations, are cumulative—that is, they include the values of the child row source operations. For example, the number of blocks read from the temporary tablespace during the `SORT ORDER BY` operation is 33,036 (71,499 – 38,463). From the previous execution statistics (see the discussion about the `disk` column), you were able to estimate only that there were at least 33,017. Also note that although in older versions these values are related to the first execution, as of version 11.2.0.2 they're averages over all executions; in this case, there's no difference because there was only one execution.

The following section summarizes the wait events for which the SQL statement waited. The following values are provided for each type of wait event:

- `Times Waited` is the number of times a wait event has occurred.
- `Max. Wait` is the maximum wait time in seconds for a single wait event.
- `Total Waited` is the total wait time in seconds for a wait event.

```
Elapsed times include waiting on following events:

Event waited on                             Times Waited   Max. Wait  Total Waited
----------------------------------------  ------------  ----------  ------------
db file sequential read                                2        0.00          0.00
db file scattered read                               530        0.06          2.79
direct path write temp                             11002        0.00          0.51
direct path read temp                              24015        0.00          2.41
```

Ideally, the sum of the wait time for all wait events should be equal to the difference of the elapsed time and CPU time provided by the execution statistics. The difference, if available, is called *unaccounted-for time*.

## UNACCOUNTED-FOR TIME

SQL trace provides information on how much time the database spends on each operation it executes. Ideally, the calculation should be very precise. Unfortunately, it's uncommon to find a trace file that gives exact information with the precision of a fraction of a second. Whenever there's a difference between the real elapsed time and the time accounted for in trace files, you have *unaccounted-for time*:

```
unaccounted-for time = real elapsed time – accounted for time
```

The most common reasons for unaccounted-for time are the following:

- The most obvious is the absence of timing information or wait events in the trace files. The former happens when the `timed_statistics` initialization parameter is set to `FALSE`. The latter happens when SQL trace is activated without level 8. In both cases, the unaccounted-for time is always a positive value. Naturally, correctly enabling extended SQL trace will help you avoid these problems.
- Generally speaking, a process may be in three states: running on a CPU, waiting for the fulfillment of a request (for example, for the fulfillment of a disk I/O operation), or waiting for a CPU in the run queue. The instrumentation code is able to calculate the time spent in the first two states, but has no clue about how much time is spent waiting in the run queue. Therefore, in case of CPU starvation, the unaccounted-for time, which is always a positive value, could be quite long. Basically, you can avoid this problem in only two ways: either by increasing the amount of available CPU or by decreasing the CPU utilization.
- The time measurements performed by the instrumentation code are precise. Nevertheless, there's a small quantization error in every measurement because of the implementation of timers in computer systems. Especially when there are a high number of measured events, these quantization errors could lead to noticeable unaccounted-for time. Owing to their nature, quantization errors could lead to positive as well as negative values for unaccounted-for time. Unfortunately, you're powerless against them. In practice, however, this problem is rarely the source of large unaccounted-for time because positive errors tend to cancel negative errors.

- If you can eliminate the other possible reasons listed here, it's likely that the problem is because the instrumentation code doesn't cover the whole code. For example, the writing of the trace file itself isn't accounted for. This is usually not a problem. But, if the trace files are written to a poorly performing device or the generation of trace information is very high, this can lead to a substantial overhead. In this case, the unaccounted-for time will always be a positive value. To avoid this problem, you should simply write trace files on a device that's able to sustain the necessary throughput. In some rare situations, you may be forced to put the trace files on a RAM disk.

Because the values of the wait events are highly aggregated, they help you know only which type of resource you have been waiting for. For example, according to the previous information, virtually the whole wait time was spent executing physical reads, but due to the aggregation, we can't, for example, see which data files were accessed (this information is included in the raw trace file). In fact, `db file sequential read` is the wait event related to single-block reads, and `db file scattered read` is the wait event related to multiblock reads (additional information about multiblock reads is given in Chapter 9). In addition, the `direct path write temp` and `direct path read temp` waits are related to the spill into the temporary tablespace.

In the analysis of the wait events, the key is knowing which operation they're related to. Fortunately, even if there are hundreds of wait event types, the most recurring ones are usually of only a few types. You can find a short description of most of them in the appendixes of the *Oracle Database Reference* manual.

The analysis continues with the next SQL statement. Because the structure of the information is the same as before, I comment only when something new or inherently different is present in the output file:

```
DECLARE
  l_count INTEGER;
BEGIN
  FOR c IN (SELECT extract(YEAR FROM d), id, pad
            FROM t
            ORDER BY extract(YEAR FROM d), id)
  LOOP
    NULL;
  END LOOP;
  FOR i IN 1..10
  LOOP
    SELECT count(n) INTO l_count
    FROM t
    WHERE id < i*123;
  END LOOP;
END;
```

The execution statistics for a PL/SQL block are limited. No information about physical and logical reads is available. This is because the resources consumed by the recursive SQL statements (for example, the query analyzed earlier) aren't associated to the parent SQL statement. This means that for each SQL statement (or PL/SQL block), you'll see only the resources used by the SQL statement (or PL/SQL block) itself:

```
call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        1      0.00       0.00          0          0          0           0
Execute      1      0.44       0.40          0          0          0           1
Fetch        0      0.00       0.00          0          0          0           0
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total        2      0.45       0.41          0          0          0           1
```

Because the PL/SQL block wasn't executed by the database recursively, the recursive depth isn't shown (the recursive depth is 0). Also, no execution plan is available:

```
Misses in library cache during parse: 1
Optimizer mode: ALL_ROWS
Parsing user id: 34    (CHRIS)
```

The database waits for `SQL*Net message to client` while instructing the network layer to send data to the client (be careful, the real time needed to send the data over the network isn't included), and the database waits for `SQL*Net message from client` while waiting for a message from the client. Consequently, for each round-trip carried out by the SQL*Net layer, you should see a pair of those wait events. Note that the number of round-trips carried out by lower-level layers might be different. For example, it's not uncommon that at the network layer (for example, IP) a larger number of round-trips are performed because of a smaller packet size:

```
Elapsed times include waiting on following events:
  Event waited on                             Times Waited   Max. Wait  Total Waited
  ----------------------------------------  -------------  ----------  ------------
  SQL*Net message to client                              1        0.00          0.00
  SQL*Net message from client                            1        0.00          0.00
```

The next SQL statement is the second one that was executed by the PL/SQL block. The structure of the information is the same as before. What's interesting to point out is that this query was executed ten times. For each execution, the trace file contains the execution statistics (since level 16 was enabled). As a result, the value of `Number of plan statistics captured` is 10, the three `Rows` columns contains different values (122, 676 and 1229), and the runtime statistics at the row source level are averages (for example, 53 disk reads over 10 executions make an average of 5):

```
SQL ID: 7fjjjf0yvd05m Plan Hash: 4270555908

SELECT COUNT(N)
FROM
 T WHERE ID < :B1 *123

call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        1      0.00       0.00          0          0          0           0
Execute     10      0.00       0.00          0          0          0           0
Fetch       10      0.00       0.02         53        303          0          10
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total       21      0.01       0.02         53        303          0          10

Misses in library cache during parse: 1
Misses in library cache during execute: 1
Optimizer mode: ALL_ROWS
Parsing user id: 34  (CHRIS)   (recursive depth: 1)
Number of plan statistics captured: 10
```

```
Rows (1st) Rows (avg) Rows (max)  Row Source Operation
---------- ---------- ----------  ---------------------------------------------------
         1          1          1  SORT AGGREGATE (cr=30 pr=5 pw=0 time=2607 us)
       122        676       1229   TABLE ACCESS BY INDEX ROWID T (cr=30 pr=5 pw=0 time=2045 us
                                                                cost=8 size=1098 card=122)
       122        676       1229    INDEX RANGE SCAN T_PK (cr=4 pr=0 pw=0 time=872 us cost=3
                                                         size=0 card=122)(object id 20991)

Rows     Execution Plan
-------  ---------------------------------------------------
      0  SELECT STATEMENT   MODE: ALL_ROWS
      1   SORT (AGGREGATE)
    122    TABLE ACCESS   MODE: ANALYZED (BY INDEX ROWID) OF 'T' (TABLE)
    122     INDEX   MODE: ANALYZED (RANGE SCAN) OF 'T_PK' (INDEX (UNIQUE)
                )

Elapsed times include waiting on following events:
  Event waited on                             Times   Max. Wait  Total Waited
  ----------------------------------------   Waited  ----------  ------------
  db file sequential read                        53        0.00          0.02
```

The last SQL statement was recursively executed by the database engine in order to get information (for example, object statistics) about the objects being used. Among other things, the query optimizer uses such information to figure out the most efficient execution plan. You have confirmation that this SQL statement was executed by the database engine because the user who parsed it is SYS. Because the recursive depth is 2, you can suppose that this SQL statement is needed to parse one SQL statement at depth 1—for example, the first SQL statement in this output file:

```
SQL ID: 96g93hntrzjtr Plan Hash: 2239883476

select /*+ rule */ bucket_cnt, row_cnt, cache_cnt, null_cnt, timestamp#,
  sample_size, minimum, maximum, distcnt, lowval, hival, density, col#,
  spare1, spare2, avgcln
from
 hist_head$ where obj#=:1 and intcol#=:2


call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        0      0.00       0.00          0          0          0           0
Execute      4      0.00       0.00          0          0          0           0
Fetch        4      0.00       0.01          5         12          0           4
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total        8      0.00       0.01          5         12          0           4

Misses in library cache during parse: 0
Optimizer mode: RULE
Parsing user id: SYS   (recursive depth: 2)
```

Because row source information wasn't written in the trace file in this case, and the user CHRIS has no privileges for the objects referenced in this SQL statement, no information about the execution plan is given (see Chapter 10 for detailed information about the privileges needed to execute the EXPLAIN PLAN statement). The output continues with the wait events:

```
Elapsed times include waiting on following events:
  Event waited on                             Times Waited   Max. Wait  Total Waited
  ----------------------------------------   ------------  ----------  ------------
  db file sequential read                               5        0.00          0.01
```

After the report of all SQL statements, you can see the overall totals for execution statistics as well as parsing and wait events. The only thing of note in this part is that nonrecursive SQL statements are separated from recursive SQL statements:

```
OVERALL TOTALS FOR ALL **NON-RECURSIVE** STATEMENTS

call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        2      0.00       0.00          0          0          0           0
Execute      3      0.45       0.42         20        226          0           3
Fetch        0      0.00       0.00          0          0          0           0
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total        5      0.45       0.42         20        226          0           3

Misses in library cache during parse: 2
Misses in library cache during execute: 1

Elapsed times include waiting on following events:
  Event waited on                             Times Waited   Max. Wait  Total Waited
  ----------------------------------------   ------------  ----------  ------------
  SQL*Net message to client                             2        0.00          0.00
  SQL*Net message from client                           2        0.00          0.00

OVERALL TOTALS FOR ALL **RECURSIVE** STATEMENTS

call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        2      0.00       0.00          0          0          0           0
Execute     29      0.00       0.00          0          0          0           0
Fetch    10037      6.50      11.97      71569      38832          8     1000028
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total    10068      6.50      11.97      71569      38832          8     1000028

Misses in library cache during parse: 2
Misses in library cache during execute: 2

Elapsed times include waiting on following events:
  Event waited on                             Times Waited   Max. Wait  Total Waited
  ----------------------------------------   ------------  ----------  ------------
  db file sequential read                              72        0.00          0.04
  db file scattered read                              530        0.06          2.79
  direct path write temp                            11002        0.00          0.51
  direct path read temp                             24015        0.00          2.41
```

The following lines summarize the number of SQL statements belonging to the current session, how many of them were executed recursively by the database engine, and how many of them the EXPLAIN PLAN statement was executed for:

```
 5  user  SQL statements in session.
13  internal SQL statements in session.
18  SQL statements in session.
 2  statements EXPLAINed in this session.
```

The output file ends by giving overall information about the trace file. At first, you can see the trace file name, its version, and the value of the sort argument used for the analysis. Then, the overall number of sessions and SQL statements are given. In this specific case, because the argument print=4 was specified, you can deduce that 14 (18 – 4) SQL statements are missing in the output file. Information about the table used to execute the EXPLAIN PLAN statement is given as well. At the end, you can see the number of lines the trace file is composed of as well as the overall elapsed time (in seconds) for all SQL statements. I'd personally prefer to see this last piece of information at the beginning of the output file rather than the end. That's because every time I open a TKPROF output file, I glance at this last line before doing everything else. Knowing how much time is spent for the whole trace file is crucial; without it, you can't judge the magnitude of the impact of one SQL statement on the total response time:

```
Trace file: DBM11203_ora_28030.trc
Trace file compatibility: 11.1.0.7
Sort options: prsela    exeela    fchela
       1  session in tracefile.
       5  user  SQL statements in trace file.
      13  internal SQL statements in trace file.
      18  SQL statements in trace file.
      18  unique SQL statements in trace file.
       2  SQL statements EXPLAINed using schema:
           CHRIS.prof$plan_table
             Default table was used.
             Table was created.
             Table was dropped.
   46125  lines in trace file.
      12  elapsed seconds in trace file.
```

## Using TVD$XTAT

Trivadis Extended Tracefile Analysis Tool (TVD$XTAT) is a command-line tool. Like TKPROF, its main purpose is to take a raw trace file as input and generate a formatted file as output. The output file can be an HTML or text file.

The simplest analysis is performed by merely specifying an input and an output file. In the following example, the input file is DBM11106_ora_6334.trc, and the output file is DBM11106_ora_6334.html:

```
tvdxtat -i DBM11106_ora_6334.trc -o DBM11106_ora_6334.html
```

### Why Is TKPROF Not Enough?

In late 1999, I had my first encounter with extended SQL trace, through the Oracle Support note *Interpreting Raw SQL_TRACE and DBMS_SUPPORT.START_TRACE output* (39817.1). From the beginning, it was clear that the information it provided was essential for understanding what an application is doing when it's connected to Oracle Database. At the same time, I was very disappointed that no tool was available for analyzing extended SQL trace files for the purpose of leveraging their content. I should note that TKPROF at that time didn't provide information about wait events. After spending too much time manually extracting information from the raw trace files through command-line tools like awk, I decided to write my own analysis tool: TVD$XTAT.

Currently, TKPROF provides information about wait events, but it still has five major problems that are addressed in TVD$XTAT:

- As soon as the `sort` argument is specified, the relationship between SQL statements is lost.
- Data is provided only in aggregated form. Consequently, useful information is lost.
- No information about bind variables is provided.
- Idle wait events (for example, `SQL*Net message from client`) taking place during the execution of a SQL statement aren't accounted for in the elapsed time shown by TKPROF. As a result, when SQL statements are sorted according to their elapsed time, the output might be misleading or, in extreme cases, very time consuming if not nearly impossible to interpret.
- When a trace file doesn't contain the text of a SQL statement (specifically, the text delimited between the `PARSING IN CURSOR` and `END OF STMT` keywords), TKPROF doesn't report the details about the SQL statement; it just accounts for the resource utilization in the summary at the end of the output file. Note that one case in which the text of a SQL statement isn't stored in the trace file is when SQL trace is activated after the execution has already been started.

## Installation

Here's how to install TVD$XTAT:

1. Download (freeware) TVD$XTAT from `http://top.antognini.ch`.
2. Uncompress the distribution file into an empty directory of your choice.
3. In the shell script used to start TVD$XTAT (either `tvdxtat.cmd` or `tvdxtat.sh`, depending on your operating system), modify the `java_home` and `tvdxtat_home` variables. The former references the directory where a Java Runtime Environment (version 1.4.2 or later) is installed. The latter references the directory where the distribution file was uncompressed.
4. Optionally, change the default value of the command-line arguments. To do that, you need to modify the `tvdxtat.properties` file, which is stored in the `config` subdirectory. By customizing the default configuration, you can avoid specifying all arguments every time you run TVD$XTAT.
5. Optionally, change the logging configuration. To do that, you have to modify the `logging.properties` file, which is stored in the `config` subdirectory. Per default, TVD$XTAT shows errors and warnings. It's not usually necessary to change these default settings, however.

## TVD$XTAT Arguments

If you run TVD$XTAT without arguments, you get a complete list of the available arguments with a short description for each of them. Note that for every argument, there's a short representation (for example, `-c`) and a long representation (for example, `--cleanup`):

```
usage: tvdxtat [-c no|yes] [-f <int>] [-l <int>] [-r 7|8|9|10|11|12]
               [-s no|yes] [-t <template>] [-w no|yes]
               [-x severe|warning|info|fine|finer] -i <input> -o <output>
```

```
-c,--cleanup     remove temporary XML file (no|yes)
-f,--feedback    display progress every x lines (integer number >= 0, no
                 progress = 0)
-h,--help        display this help information and exit
-i,--input       input trace file name (valid extensions: trc|gz|zip)
-l,--limit       limit the size of lists (e.g. number of statements) in
                 the output file (integer number >= 0, unlimited = 0)
-o,--output      output file name (a temporary XML file with the same
                 name but with the extension xml is also created)
-r,--release     major release of the database engine that generated the
                 input trace file (7|8|9|10|11|12)
-s,--sys         report information about SYS recursive statements
                 (no|yes)
-t,--template    name of the XSL template used to generate the output
                 file (html|text)
-v,--version     print product version and exit
-w,--wait        report detailed information about wait events (no|yes)
-x,--logging     logging level (severe|warning|info|fine|finer)
```

The function of each argument is as follows:

- `input` specifies the name of the input file. The input file must be either a trace file (extension `.trc`) or a compressed file (extension `.gz` or `.zip`) that contains one or several trace files. Note, however, that only a single trace file is extracted from `.zip` files.
- `output` specifies the name of the output file. During processing, a temporary XML file is created with the same name as the output file but with the extension `.xml`. Be careful, if another file with the same name as the output file exists, it will be overwritten.
- `cleanup` specifies whether the temporary XML file generated during processing is removed at the end. Generally, it should be set to `yes`. This argument is important only during the development phase to check intermediate results.
- `feedback` specifies whether progress information is displayed. It's useful during the processing of very large trace files to know the current status of the analysis. The argument specifies the interval (number of lines) at which a new message will be generated. If it's set to 0, no progress information is displayed.
- `help` specifies whether to display help information. It can't be used along with other arguments.
- `limit` sets the maximum number of elements present in lists (for example, the lists used for SQL statements, waits, and bind variables) available in the output file. If it's set to 0, there's no limit.
- `release` specifies the major version number (that is, 7, 8, 9, 10, 11, or 12) of Oracle Database that generated the input trace file.
- `sys` specifies whether information about recursive SQL statements that are executed by the `sys` user is available in the output file. It's commonly set to `no`.
- `template` specifies the name of the XSL template used to generate the output file. By default, two templates are available: `html.xsl` and `text.xsl`. The former generates an HTML output file, and the latter generates a text output file. The default templates can be modified and new templates can be written as well. In this way, it's possible to fully customize the output file. The templates must be stored in the `templates` subdirectory.

- `version` specifies whether to display the version number of TVD$XTAT. It can't be used along with other arguments.
- `wait` specifies whether detailed information for wait events is shown. Enabling this feature (that is, setting this argument to `yes`) might have a significant overhead during processing. Therefore, I suggest you set it initially to `no`. Afterward, if the basic wait information isn't enough, you can run another analysis with it set to `yes`.
- `logging` controls the logging level. The following values are available: `severe`, `warning`, `info`, `fine`, and `finer`. This argument is important only when debugging the execution of the tool.

## Interpreting TVD$XTAT Output

This section is based on the same trace file already used earlier with TKPROF. Because the output layout of TVD$XTAT is based on the output layout of TKPROF, I describe only information specific to TVD$XTAT here. To generate the output file, I used the following parameters:

```
tvdxtat -i DBM11203_ora_28030.trc -o DBM11203_ora_28030.txt –s no –w yes -t text
```

Note that both the trace file and the output file in HTML and text format are available along with the other files of this chapter.

The output file begins with overall information about the input trace file. The most important information in this part is the interval covered by the trace file and the number of transactions recorded in it:

```
Database Version
****************
Oracle Database 11g Enterprise Edition Release 11.2.0.3.0 - 64bit Production
With the Partitioning, Automatic Storage Management, Oracle Label Security, OLAP,
Data Mining and Real Application Testing options

Analyzed Trace File
*******************
/u00/app/oracle/diag/rdbms/dbm11203/DBM11203/trace/DBM11203_ora_28030.trc

Interval
********
Beginning 30 Nov 2012 23:21:45.691
End       30 Nov 2012 23:21:58.097
Duration  12.407 [s]

Transactions
************
Committed  0
Rollbacked 0
```

The analysis of the output file starts by looking at the overall resource usage profile. The processing here lasted 12.407 seconds. About 56 percent of this time was spent running on the CPU, about 24 percent was spent reading and writing temporary files (`direct path read temp` and `direct path write temp`), and about 23 percent was spent reading data files (`db file scattered read` and `db file sequential read`). In summary, most of the time is spent on CPU with the rest on disk I/O operations. Notice that the unaccounted-for time is explicitly given:

```
Resource Usage Profile
**********************
                                  Total          Number of Duration per
Component                  Duration [s]       %     Events   Events [s]
--------------------------- ------------ ------- --------- ------------
CPU                               6.969  56.171        n/a          n/a
db file scattered read            2.792  22.502        530        0.005
direct path read temp             2.417  19.479     24,015        0.000
direct path write temp            0.513   4.136     11,002        0.000
db file sequential read           0.041   0.326         72        0.001
SQL*Net message from client       0.001   0.008          2        0.001
SQL*Net message to client         0.000   0.000          2        0.000
unaccounted-for                  -0.325  -2.623        n/a          n/a
--------------------------- ------------ -------
Total                            12.407 100.000
```

■ **Note** TVD$XTAT always sorts lists according to response time. No option is available to change this behavior because this is the only order that makes sense to investigate performance problems.

Knowing how the database engine spent time gives a general overview only. To continue the analysis, it's essential to find out which SQL statements are responsible for that processing time. For this purpose, a list containing all nonrecursive SQL statements is provided after the overall resource usage profile. In this case, you can see that a single SQL statement (actually, a PL/SQL block) is responsible for the whole processing time. Note that in the following list, the total isn't 100 percent because the unaccounted-for time is omitted:

```
The input file contains 18 distinct statements, 15 of which are recursive.
In the following table, only non-recursive statements are reported.

                          Total          Number of  Duration per
Statement ID Type   Duration [s]       % Executions Execution [s]
------------ ------ ------------ ------- ---------- -------------
#1           PL/SQL       12.724 102.561          1        12.724
#5           PL/SQL        0.006   0.045          1         0.006
#9           PL/SQL        0.002   0.016          1         0.002
------------ ------ ------------ -------
Total                     12.732 102.623
```

Naturally, the next step is to get more information about the SQL statement that's responsible for most of the processing time. To reference SQL statements easily, TVD$XTAT generates an identifier (the `Statement ID` column in the previous excerpt) for each SQL statement. In the HTML version of the output file, you can simply click that identifier to locate the SQL statement details. In the text version, however, you have to search for the string "STATEMENT #1".

The following information is then given for every SQL statement: general information about the execution environment, the SQL statement, the execution statistics, the execution plan, the bind variables used for the executions, and the wait events. The execution plan, the bind variables, and the wait events are optional and are obviously reported only if they have been recorded in the trace file.

At first, general information about the execution environment and the text of the SQL statement is given. Note that information about the session attributes is displayed only when available. For example, in this case, the attribute action name isn't displayed because the application didn't set it. Also note that SQL ID is available for trace files generated as of version 11.1 only:

```
Session ID          156.29
Service Name        SYS$USERS
Module Name         SQL*Plus
Parsing User        34
Hash Value          166910891
SQL ID              15p0p084z5qxb

DECLARE
  l_count INTEGER;
BEGIN
  FOR c IN (SELECT extract(YEAR FROM d), id, pad
            FROM t
            ORDER BY extract(YEAR FROM d), id)
  LOOP
    NULL;
  END LOOP;
  FOR i IN 1..10
  LOOP
    SELECT count(n) INTO l_count
    FROM t
    WHERE id < i*123;
  END LOOP;
END;
```

The execution statistics provide data in tabular form, aggregated by type of database call. Because the table layout is based on the one generated by TKPROF, the meaning of the columns is the same. There are, however, two additional columns: `Misses` and `LIO`. The former is the number of hard parses that occurred during each type of call. The latter is just the sum of the `Consistent` and `Current` columns. Also notice that TVD$XTAT provides two tables. The first also includes the statistics about all recursive SQL statements related to the current one. The second, like TKPROF, doesn't include them:

```
Database Call Statistics with Recursive Statements
**************************************************

Call    Count Misses   CPU Elapsed    PIO    LIO Consistent Current Rows
------- ----- ------ ----- ------- ------ ------ ---------- ------- ----
Parse       1      1 0.005   0.006      7     20         20       0    0
Execute     1      0 6.957  12.387 71,562 38,820     38,812       8    1
Fetch       0      0 0.000   0.000      0      0          0       0    0
------- ----- ------ ----- ------- ------ ------ ---------- ------- ----
Total       2      1 6.962  12.393 71,569 38,840     38,832       8    1
```

```
Database Call Statistics without Recursive Statements
*****************************************************

Call    Count Misses   CPU Elapsed    PIO    LIO Consistent Current Rows
------- ----- ------ ----- ------- ------ ------ ---------- ------- ----
Parse       1      1 0.005   0.004      0      0          0       0    0
Execute     1      0 0.448   0.410      0      0          0       0    1
Fetch       0      0 0.000   0.000      0      0          0       0    0
------- ----- ------ ----- ------- ------ ------ ---------- ------- ----
Total       2      1 0.453   0.414      0      0          0       0    1
```

In this case, little time was spent by the current SQL statement according to the execution statistics. This is also shown by the following resource usage profile. In fact, it shows that about 96 percent of the time was spent by recursive SQL statements:

```
Component                   Duration [s]       % Events Events [s]
--------------------------- ------------ ------- ------ ----------
recursive statements              12.271  96.437    n/a        n/a
CPU                                0.453   3.560    n/a        n/a
SQL*Net message from client        0.000   0.003      1      0.000
SQL*Net message to client          0.000   0.000      1      0.000
--------------------------- ------------ -------
Total                             12.724 100.000
```

To show which SQL statements these are, the resource usage profile is followed by a list of the recursive SQL statements. From this list, you can see that SQL statement 2, which is a SELECT statement, was responsible for about 96 percent of the response time. Note that all other SQL statements except for SQL statement 3 were generated by the database engine itself (for example, during the parse phase) and, therefore, are marked with the label SYS recursive:

```
7 recursive statements were executed.
                                              Total
Statement ID Type                      Duration [s]       %
------------ ---------------------- ------------ -------
#2           SELECT                       12.234  96.150
#3           SELECT                        0.033   0.263
#7           SELECT (SYS recursive)        0.003   0.022
#11          SELECT (SYS recursive)        0.000   0.001
#12          SELECT (SYS recursive)        0.000   0.001
#14          SELECT (SYS recursive)        0.000   0.001
#16          SELECT (SYS recursive)        0.000   0.000
------------ ---------------------- ------------ -------
Total                                     12.252  96.286
```

Because SQL statement 2 is responsible for most of the response time, you have to drill down further and get its details. The structure is basically the same as for SQL statement 1. There is, however, additional information. In the part that displays the execution environment, you can see the recursive level (remember, the SQL statements executed by the application are at level 0) and the parent SQL statement identifier. This second piece of information is essential in order to not lose the relationship between SQL statements (as TKPROF does!):

```
Session ID          156.29
Service Name        SYS$USERS
Module Name         SQL*Plus
Parsing User        34
Recursive Level     1
```

```
Parent Statement ID  1
Hash Value           955957303
SQL ID               7wd0gdwwgph1r

SELECT EXTRACT(YEAR FROM D), ID, PAD FROM T ORDER BY EXTRACT(YEAR FROM D), ID
```

Next, you find the execution plan, if it's available in the trace file. Its format is similar to the output generated by TKPROF:

```
Execution Plan
**************

Optimizer Mode       ALL_ROWS
Hash Value           961378228

     Rows Operation
--------- -----------------------------------------------------------
1,000,000 SORT ORDER BY (cr=38474 pr=71499 pw=33035 time=11123996 us
                         cost=216750 size=264000000 card=1000000)
1,000,000   TABLE ACCESS FULL T (cr=38474 pr=38463 pw=0 time=5674541 us
                                 cost=21 size=264000000 card=1000000)
```

As is the case for all SQL statements, the execution plan is followed by the execution statistics, the resource usage profile, and, if available, the recursive SQL statements at level 2 (you're currently looking at a SQL statement at level 1). In this case, you can see that the recursive SQL statements are responsible for less than 1 percent of the response time. In other words, SQL statement 2 is itself responsible for the whole response time:

```
Database Call Statistics with Recursive Statements
**************************************************

Call               Count Misses   CPU Elapsed    PIO    LIO Consistent Current      Rows
----------------- ------ ------ ----- ------- ------ ------ ---------- ------- ---------
Parse                  1      1 0.004   0.010      7     32         32       0         0
Execute                1      1 0.000   0.000      0      0          0       0         0
Fetch             10,001      0 6.492  11.926 71,499 38,482     38,474       8 1,000,000
----------------- ------ ------ ----- ------- ------ ------ ---------- ------- ---------
Total             10,003      2 6.496  11.936 71,506 38,514     38,506       8 1,000,000
Average (per row)      0      0 0.000   0.000      0      0          0       0         1

Database Call Statistics without Recursive Statements
*****************************************************

Call               Count Misses   CPU Elapsed    PIO    LIO Consistent Current      Rows
----------------- ------ ------ ----- ------- ------ ------ ---------- ------- ---------
Parse                  1      1 0.001   0.001      0      9          9       0         0
Execute                1      1 0.000   0.000      0      0          0       0         0
Fetch             10,001      0 6.492  11.926 71,499 38,482     38,474       8 1,000,000
----------------- ------ ------ ----- ------- ------ ------ ---------- ------- ---------
Total             10,003      2 6.493  11.927 71,499 38,491     38,483       8 1,000,000
Average (per row)      0      0 0.000   0.000      0      0          0       0         1
```

```
Resource Usage Profile
**********************

                              Total         Number of Duration per
Component               Duration [s]       %    Events   Events [s]
----------------------- ------------ ------- --------- ------------
CPU                            6.493  53.071       n/a          n/a
db file scattered read         2.792  22.818       530        0.005
direct path read temp          2.417  19.753    24,015        0.000
direct path write temp         0.513   4.194    11,002        0.000
recursive statements           0.020   0.161       n/a          n/a
db file sequential read        0.000   0.002         2        0.000
---------------------- ----------- -------
Total                         12.234 100.000

6 recursive statements were executed.
                                                   Total
Statement ID Type                    Duration [s]       %
------------ --------------------- ------------ -------
#4           SELECT (SYS recursive)       0.015   0.121
#6           SELECT (SYS recursive)       0.004   0.032
#10          SELECT (SYS recursive)       0.001   0.008
#13          SELECT (SYS recursive)       0.000   0.001
#17          SELECT (SYS recursive)       0.000   0.000
#18          SELECT (SYS recursive)       0.000   0.000
------------ --------------------- ----------- -------
Total                                     0.006   0.050
```

In the resource usage profiles shown up to now, wait events are just summarized. To have additional information, a histogram like the following is provided for every component of the resource usage profile. In this case, the statistics are related to the `db file scattered read` wait event of SQL statement 2. Notice how the wait events are grouped by their duration (`Range` column). For example, you see that about 52 percent of the wait events lasted between 4,096 and 8,192 microseconds. Because the `db file scattered read` wait event is associated with multiblock reads, it might also be useful to see the average number of blocks read by a disk I/O operation (`Blocks per Event` column):

```
                            Total         Number of          Duration per        Blocks per
Range [µs]                Duration       %    Events       %   Event [µs] Blocks      Event
------------------------ -------- ------- --------- ------- ------------ ------ ----------
256 □ duration < 512        0.003   0.111         7       1          443     56          8
512 □ duration < 1024       0.008   0.288         9       2          892     72          8
1024 □ duration < 2048      0.033   1.191        18       3        1,847    826         46
2048 □ duration < 4096      0.517  18.525       166      31        3,115 11,627         70
4096 □ duration < 8192      1.465  52.459       264      50        5,547 20,742         79
8192 □ duration < 16384     0.579  20.736        60      11        9,648  4,722         79
16384 □ duration < 32768    0.126   4.496         5       1       25,101    336         67
32768 □ duration < 65536    0.061   2.195         1       0       61,274     81         81
------------------------ -------- ------- --------- ------- ------------ ------ ----------
Total                       2.792 100.000       530 100.000        5,267 38,462         73
```

If the display of detailed information is enabled (the `wait` argument is used for that purpose), further details might be provided in addition to the previous histogram. This strongly depends on the type of wait event. Actually, for many events, no additional information is generated. Wait events related to disk I/O operations typically provide information at the file level. For example, the following table shows the statistics related to the `db file scattered read` wait event of SQL statement 2. In this example, you can see that 530 disk I/O operations were performed on data file 4 in 2.792 seconds. This means that each disk I/O operation lasted 5.267 milliseconds on average (be careful, the table displays this in microseconds):

```
File            Total          Number of                                Duration per
Number   Duration [s]       %     Events       %  Blocks [b]       %    Event [µs]
------- ------------- ------- ---------- ------- ----------- ------- -------------
4               2.792 100.000        530 100.000      38,462 100.000         5,267
```

As you might expect, for all SQL statements, the structure is the same. However, one piece of information is missing in the first two SQL statements. To illustrate, let's have a look at an excerpt of the information provided for a SQL statement using bind variables. As the output for SQL statement 3 shows, if information about bind variables has been recorded in the trace file, TVD$XTAT shows their datatype and value. In addition, if several executions have been performed (in this case, ten), bind variables will be tagged by a number of execution. For example:

```
Session ID          156.29
Service Name        SYS$USERS
Module Name         SQL*Plus
Parsing User        34
Recursive Level     1
Parent Statement ID 1
Hash Value          1035370675
SQL ID              7fjjjf0yvd05m

SELECT COUNT(N) FROM T WHERE ID < :B1 *123

Bind Variables
**************

10 bind variable sets were used to execute this statement.

Number of
Execution  Bind  Datatype  Value
---------- ----- --------- ------
1          1     NUMBER    "1"
2          1     NUMBER    "2"
3          1     NUMBER    "3"
4          1     NUMBER    "4"
5          1     NUMBER    "5"
6          1     NUMBER    "6"
7          1     NUMBER    "7"
8          1     NUMBER    "8"
9          1     NUMBER    "9"
10         1     NUMBER    "10"
```

In summary, even if plenty of SQL statements are executed (18 in total), SQL statement 2 is responsible for most of the response time. Therefore, to improve performance, the execution of that SQL statement should be avoided or optimized.

## Profiling PL/SQL Code

The database engine provides two profilers integrated in the PL/SQL engine for you to use in profiling PL/SQL code. One is a line-level profiler managed through the `dbms_profiler` package. The other is a call-level profiler (also known as *hierarchical profiler*) managed through the `dbms_hprof` package. Table 3-2 summarizes the main advantages of each.

***Table 3-2.*** *Main Advantages of DBMS_HPROF and DBMS_PROFILER*

| DBMS_HPROF | DBMS_PROFILER |
|---|---|
| Imposes very small overhead when enabled | Provides information at the line level |
| Provides information at the call level | Available in releases before version 11.1 |
| Has notion of both “self time” and “total time” | Supported by all major development tools |
| Doesn’t require additional privileges | |
| Supports native-compiled PL/SQL | |

The hierarchical profiler provides runtime statistics that aren’t just more precise than those from the line-level profiler, but also more useful. The one exception is when you actually require information at the line level. Hence, I recommend using the hierarchical profiler, if it’s available, unless you have a specific need for the line-level information provided by the other.

### Using DBMS_HPROF

With the `dbms_hprof` package, introduced in version 11.1, you can enable and disable the hierarchical profiler at the session level. While enabled, the following information is gathered for each PL/SQL and SQL call that’s executed:

- The total number of times the call is executed
- The time spent processing the call
- The time spent processing subcalls
- Information about the call hierarchy

The gathering takes place at session level for all PL/SQL code (for example, in packages and triggers) that a user is able to execute (a limitation is that wrapped PL/SQL code allows you to gather only information about the top-level calls). You need only the `EXECUTE` privilege on the `dbms_hprof` package to enable this profiling.

The data gathered during the profiling is stored in a trace file at the operating-system level. Then, for analysis purposes, the data can be either loaded into the database tables shown in Figure 3-4, or it can be processed with the PLSHPROF utility.

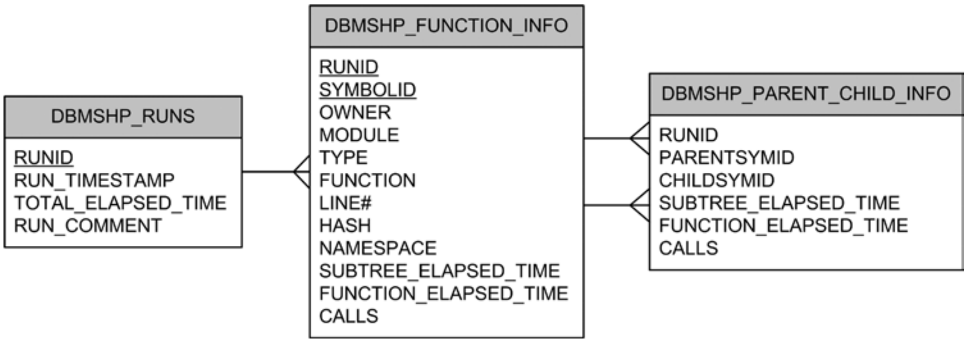


***Figure 3-4.*** *The profiler stores the gathered information in three database tables. Notice that the primary keys consist of the underlined columns*

The `dbmshp_runs` table gives information about which sessions have been profiled. The `dbmshp_function_info` table provides the list of subprograms that have been executed for each run. The `dbmshp_parent_child_info` table gives the parent-child relationship between callers and callees. In other words, it contains information to reconstruct the call hierarchy.

## Installing the Output Tables

The `dbms_hprof` package runs with the privileges of the user executing it. Consequently, the output tables don't necessarily need to be created by the `sys` user. Either the database administrator installs the output tables (by running the `dbmshptab.sql` script) once and provides the necessary synonyms and privileges to use them, or each user installs them in his own schema. In the following example, the database administrator installs the tables once:

```
CONNECT / AS SYSDBA
@?/rdbms/admin/dbmshptab.sql

CREATE PUBLIC SYNONYM dbmshp_runs FOR dbmshp_runs;
CREATE PUBLIC SYNONYM dbmshp_function_info FOR dbmshp_function_info;
CREATE PUBLIC SYNONYM dbmshp_parent_child_info FOR dbmshp_parent_child_info;
CREATE PUBLIC SYNONYM dbmshp_runnumber FOR dbmshp_runnumber;

GRANT SELECT, INSERT, UPDATE, DELETE ON dbmshp_runs TO PUBLIC;
GRANT SELECT, INSERT, UPDATE, DELETE ON dbmshp_function_info TO PUBLIC;
GRANT SELECT, INSERT, UPDATE, DELETE ON dbmshp_parent_child_info TO PUBLIC;
GRANT SELECT ON dbmshp_runnumber TO PUBLIC;
```

## Gathering the Profiling Data

Begin a profiling analysis by calling the `start_profiling` procedure to enable the profiler. The procedure supports three parameters:

> `location` specifies the name of the directory object pointing to an OS-level directory where the trace file containing the profiling data will be stored.

filename specifies the name of the trace file. If the file already exists, it's silently overwritten.

max_depth specifies whether the gathering of profiling data is limited to a specific call depth. By default (NULL), there's no limit.

While the profiler is enabled, profiling data is gathered for code executed by the PL/SQL engine. Profiling is disabled by calling the stop_profiling procedure.

Once a trace file containing the profiling data is available, it can be loaded into the output tables by calling the analyze function. Two parameters are required when calling the analyze function: location and filename. It goes without saying that their purpose is exactly the same as for the identically named parameters in the start_profiling procedure. Hence, you should set them to the same values. Other parameters are supported by the analyze function, and you can view them in the *PL/SQL Packages and Types Reference* manual.

The following example is an excerpt of the output generated by the dbms_hprof.sql script. The example shows a minimal run aimed at profiling an anonymous PL/SQL block. The runid value selected while loading the profiler data into the database is used in the next section to analyze the output of the profiling session:

```
SQL> BEGIN
  2    dbms_hprof.start_profiling(location => 'PLSHPROF_DIR',
  3                               filename => 'dbms_hprof.trc');
  4  END;
  5  /

SQL> DECLARE
  2    l_count INTEGER;
  3  BEGIN
  4    perfect_triangles(1000);
  5    SELECT count(*) INTO l_count
  6    FROM all_objects;
  7  END;
  8  /

SQL> BEGIN
  2    dbms_hprof.stop_profiling;
  3  END;
  4  /

SQL> SELECT dbms_hprof.analyze(location => 'PLSHPROF_DIR',
  2                            filename => 'dbms_hprof.trc') AS runid
  3  FROM dual;

     RUNID
----------
         1
```

Once the profiling data has been loaded into the output tables, it's time to report it. The three main available options are described in the next sections.

## Manually Reporting the Profiling Data

After the load, the profiling data is stored in the output tables. It's then possible to query that data with regular queries, as shown in this section. What follows is an excerpt of the output generated by the dbms_hprof.sql script.

The first query breaks up the profiling data at namespace level. If, as in this case, you see that the PL/SQL code is responsible for a relevant part of the response time (45.1% in this case), it makes sense that you continue looking, in more detail, at the data provided by the profiler. On the other hand, if you see that SQL is responsible for most of the response time, a PL/SQL profiler is the wrong tool of choice; there are better tools, such as SQL trace, to find out which SQL statements are the slower ones.  Either way, the first query provides valuable information, helping you to know where to focus your attention next.

Here's an example of the first query along with some output:

```
SQL> SELECT sum(function_elapsed_time)/1000 AS total_ms,
  2         100*ratio_to_report(sum(function_elapsed_time)) over () AS total_percent,
  3         sum(calls) AS calls,
  4         100*ratio_to_report(sum(calls)) over () AS calls_percent,
  5         namespace AS namespace_name
  6  FROM dbmshp_function_info
  7  WHERE runid = 1
  8  GROUP BY namespace
  9  ORDER BY total_ms DESC;

TOTAL [ms]   TOT%      CALLS   CAL% NAMESPACE_NAME
---------- ------ ---------- ------ ---------------
       565   54.9         89    5.6 SQL
       464   45.1      1,494   94.4 PLSQL
```

The second query, which is very similar to the previous one, breaks up the profiling data at module level. In this case you can see that most of the PL/SQL response time (44.9%) was spent inside the `perfect_triangles` procedure:

```
SQL>  SELECT sum(function_elapsed_time)/1000 AS total_ms,
  2         100*ratio_to_report(sum(function_elapsed_time)) over () AS total_percent,
  3         sum(calls) AS calls,
  4         100*ratio_to_report(sum(calls)) over () AS calls_percent,
  5         namespace,
  6         nvl(nullif(owner || '.' || module, '.'), function) AS module_name,
  7         type
  8  FROM dbmshp_function_info
  9  WHERE runid = 1
 10  GROUP BY namespace, nvl(nullif(owner || '.' || module, '.'), function), type
 11  ORDER BY total_ms DESC;

TOTAL [ms]  TOT%  CALLS  CAL% NAMESPACE MODULE_NAME                TYPE
---------- ----- ------ ----- --------- -------------------------- ------------
       521  50.6      1   0.1 SQL       __static_sql_exec_line5
       462  44.9  1,214  76.7 PLSQL     CHRIS.PERFECT_TRIANGLES    PROCEDURE
        44   4.3     88   5.6 SQL       SYS.XML_SCHEMA_NAME_PRESENT PACKAGE BODY
         1   0.1     44   2.8 PLSQL     SYS.XML_SCHEMA_NAME_PRESENT PACKAGE BODY
         1   0.1      3   0.2 PLSQL     __plsql_vm
         0   0.0      3   0.2 PLSQL     __anonymous_block
         0   0.0     46   2.9 PLSQL     __plsql_vm@1
         0   0.0    179  11.3 PLSQL     SYS.DBMS_OUTPUT            PACKAGE BODY
         0   0.0      1   0.1 PLSQL     SYS.DBMS_UTILITY           PACKAGE BODY
         0   0.0      1   0.1 PLSQL     SYS.DBMS_SESSION           PACKAGE BODY
         0   0.0      1   0.1 PLSQL     SYS.DBMS_APPLICATION_INFO  PACKAGE BODY
         0   0.0      1   0.1 PLSQL     SYS.DBMS_APPLICATION_INFO  PACKAGE SPEC
         0   0.0      1   0.1 PLSQL     SYS.DBMS_HPROF             PACKAGE BODY
```

The aim of the third query, which is hierarchical and at a finer level (including all PL/SQL calls), isn't just to display the call hierarchy, but also to show how much time was spent by the caller and the callees. For example, you can see that from the 463 milliseconds spent calling `perfect_triangles`, 393 milliseconds were spent by the procedure itself. The rest, 69 (64 + 5) milliseconds, were spent inside the callees `sides_are_unique` and `store_dup_sides`, both not reported in the previous query:

```
SQL> SELECT lpad(' ', (level-1) * 2) || nullif(c.owner || '.', '.') ||
            CASE WHEN c.module = c.function
                 THEN c.function
                 ELSE nullif(c.module || '.', '.') || c.function END AS function_name,
            pc.subtree_elapsed_time/1000 AS total_ms,
            pc.function_elapsed_time/1000 AS function_ms,
            pc.calls AS calls
     FROM dbmshp_parent_child_info pc,
          dbmshp_function_info p,
          dbmshp_function_info c
     START WITH pc.runid = 1
     AND p.runid = pc.runid
     AND c.runid = pc.runid
     AND pc.childsymid = c.symbolid
     AND pc.parentsymid = p.symbolid
     AND p.symbolid = 1
     CONNECT BY pc.runid = prior pc.runid
     AND p.runid = pc.runid
     AND c.runid = pc.runid
     AND pc.childsymid = c.symbolid
     AND pc.parentsymid = p.symbolid
     AND prior pc.childsymid = pc.parentsymid
     ORDER SIBLINGS BY total_ms DESC;

FUNCTION NAME                                                TOTAL [ms] FUNCTION [ms]  CALLS
------------------------------------------------------------ ---------- ------------- ------
__static_sql_exec_line5                                             566           521      1
  __plsql_vm@1                                                       45             0     46
    SYS.XML_SCHEMA_NAME_PRESENT.IS_SCHEMA_PRESENT                    45             1     44
      SYS.XML_SCHEMA_NAME_PRESENT.__dyn_sql_exec_line34              22            22     44
      SYS.XML_SCHEMA_NAME_PRESENT.__dyn_sql_exec_line17              22            22     44
CHRIS.PERFECT_TRIANGLES                                             463           393      1
  CHRIS.PERFECT_TRIANGLES.PERFECT_TRIANGLES.SIDES_ARE_UNIQUE         64            64  1,034
  CHRIS.PERFECT_TRIANGLES.PERFECT_TRIANGLES.STORE_DUP_SIDES           5             5    179
  SYS.DBMS_OUTPUT.PUT_LINE                                            0             0    179
SYS.DBMS_SESSION.IS_ROLE_ENABLED                                      0             0      1
  SYS.DBMS_UTILITY.CANONICALIZE                                       0             0      1
SYS.DBMS_APPLICATION_INFO.SET_MODULE                                  0             0      1
SYS.DBMS_APPLICATION_INFO.__pkg_init                                  0             0      1
SYS.DBMS_HPROF.STOP_PROFILING                                         0             0
```

## Using PLSHPROF

You can use the command-line utility PLSHPROF to process a trace file generated by `dbms_hprof`. In doing so, you generate a collection of HTML reports. If you run PLSHPROF without specifying any argument as input, you get a complete list of PLSHPROF's arguments including a short description:

```
Usage: plshprof [<option>...] <tracefile1> [<tracefile2>]
  Options:
    -trace <symbol>    (no default)     specify function name of tree root
    -skip <count>      (default=0)      skip first <count> invokations
    -collect <count>   (default=1)      collect info for <count> invokations
    -output <filename> (default=<symbol>.html or <tracefile1>.html)
    -summary                            print time only
```

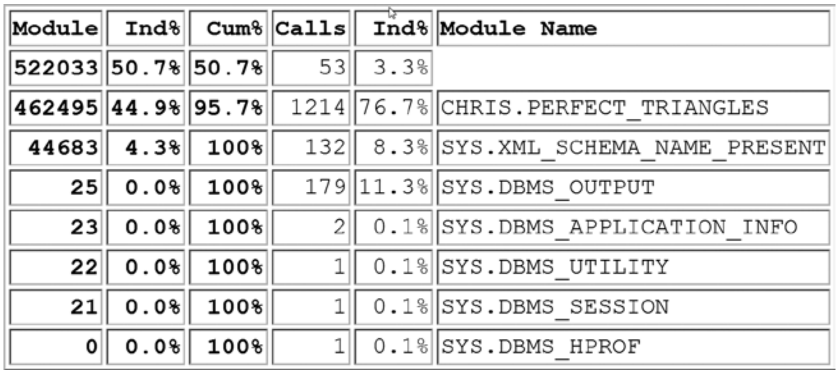
As you can see, it's possible to specify one or two trace files and several options. If a single trace file is specified, PLSHPROF generates the following reports:

- Function elapsed time data sorted according to eight different criteria
- Module elapsed time data sorted according to three different criteria
- Namespace elapsed time data sorted according to three different criteria
- Parents and children elapsed time data

For example, issue the following command to process the `dbms_hprof.trc` trace file and generate a set of reports that can be accessed through a file named `dbms_hprof.html`:

```
plshprof -output dbms_hprof dbms_hprof.trc
```

Note that both the trace file and the HTML reports are available in the `dbms_hprof.zip` file. Figure 3-5 shows one of the available reports.



| Module | Ind% | Cum% | Calls | Ind% | Module Name |
|---|---|---|---|---|---|
| 522033 | 50.7% | 50.7% | 53 | 3.3% | |
| 462495 | 44.9% | 95.7% | 1214 | 76.7% | CHRIS.PERFECT_TRIANGLES |
| 44683 | 4.3% | 100% | 132 | 8.3% | SYS.XML_SCHEMA_NAME_PRESENT |
| 25 | 0.0% | 100% | 179 | 11.3% | SYS.DBMS_OUTPUT |
| 23 | 0.0% | 100% | 2 | 0.1% | SYS.DBMS_APPLICATION_INFO |
| 22 | 0.0% | 100% | 1 | 0.1% | SYS.DBMS_UTILITY |
| 21 | 0.0% | 100% | 1 | 0.1% | SYS.DBMS_SESSION |
| 0 | 0.0% | 100% | 1 | 0.1% | SYS.DBMS_HPROF |

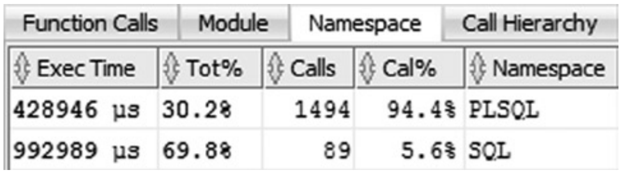
***Figure 3-5.*** *Module Elapsed Time Data Sorted by Total Function Elapsed Time, as generated by PLSHPROF*

Specifying two trace files is useful when you want to compare two runs of the same program. For example, you can specify two trace files and compare them to assess the performance improvement or regression introduced by code change. If the two trace files aren't identical, PLSHPROF generates a collection of reports similar to the ones generated for a single trace file, but pointing out the deltas between the two runs.

## Using a GUI

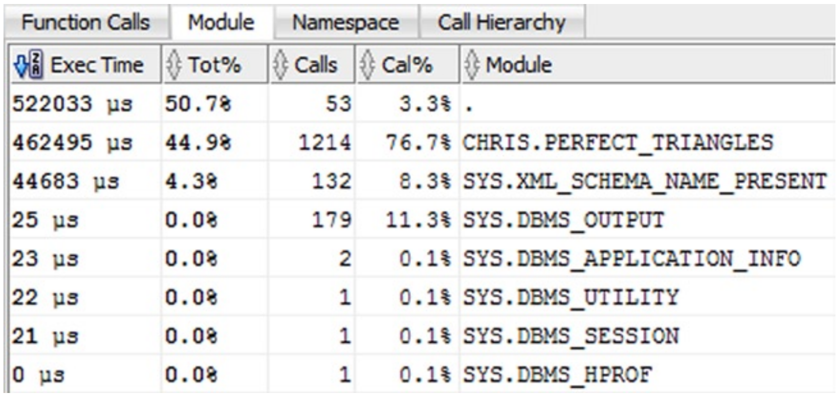
In addition to the methods covered in the previous sections, it's also possible to use one of the graphical interfaces available in third-party products. Such an interface is provided by SQL Developer (Oracle) and Toad (Dell). These tools can be used to profile code, usually by clicking a check box or button before running a test, or by simply analyzing the content of the output tables.

Figures 3-6 through 3-8 show part of the information provided by SQL Developer for the profiling session illustrated in the previous sections.



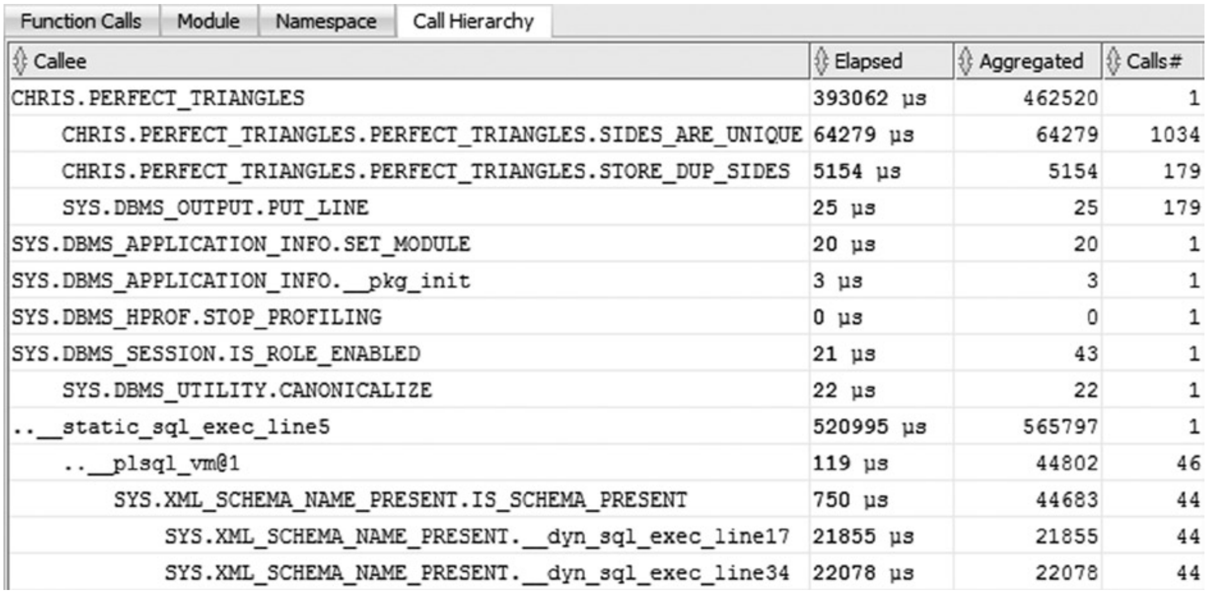
| Exec Time | Tot% | Calls | Cal% | Namespace |
|---|---|---|---|---|
| 428946 µs | 30.2% | 1494 | 94.4% | PLSQL |
| 992989 µs | 69.8% | 89 | 5.6% | SQL |

***Figure 3-6.*** *Profiling data at the namespace level displayed in SQL Developer*



| Exec Time | Tot% | Calls | Cal% | Module |
|---|---|---|---|---|
| 522033 µs | 50.7% | 53 | 3.3% | . |
| 462495 µs | 44.9% | 1214 | 76.7% | CHRIS.PERFECT_TRIANGLES |
| 44683 µs | 4.3% | 132 | 8.3% | SYS.XML_SCHEMA_NAME_PRESENT |
| 25 µs | 0.0% | 179 | 11.3% | SYS.DBMS_OUTPUT |
| 23 µs | 0.0% | 2 | 0.1% | SYS.DBMS_APPLICATION_INFO |
| 22 µs | 0.0% | 1 | 0.1% | SYS.DBMS_UTILITY |
| 21 µs | 0.0% | 1 | 0.1% | SYS.DBMS_SESSION |
| 0 µs | 0.0% | 1 | 0.1% | SYS.DBMS_HPROF |

***Figure 3-7.*** *Profiling data at the module level displayed in SQL Developer*



| Callee | Elapsed | Aggregated | Calls# |
|---|---|---|---|
| CHRIS.PERFECT_TRIANGLES | 393062 µs | 462520 | 1 |
| CHRIS.PERFECT_TRIANGLES.PERFECT_TRIANGLES.SIDES_ARE_UNIQUE | 64279 µs | 64279 | 1034 |
| CHRIS.PERFECT_TRIANGLES.PERFECT_TRIANGLES.STORE_DUP_SIDES | 5154 µs | 5154 | 179 |
| SYS.DBMS_OUTPUT.PUT_LINE | 25 µs | 25 | 179 |
| SYS.DBMS_APPLICATION_INFO.SET_MODULE | 20 µs | 20 | 1 |
| SYS.DBMS_APPLICATION_INFO.__pkg_init | 3 µs | 3 | 1 |
| SYS.DBMS_HPROF.STOP_PROFILING | 0 µs | 0 | 1 |
| SYS.DBMS_SESSION.IS_ROLE_ENABLED | 21 µs | 43 | 1 |
| SYS.DBMS_UTILITY.CANONICALIZE | 22 µs | 22 | 1 |
| ..__static_sql_exec_line5 | 520995 µs | 565797 | 1 |
| ..__plsql_vm@1 | 119 µs | 44802 | 46 |
| SYS.XML_SCHEMA_NAME_PRESENT.IS_SCHEMA_PRESENT | 750 µs | 44683 | 44 |
| SYS.XML_SCHEMA_NAME_PRESENT.__dyn_sql_exec_line17 | 21855 µs | 21855 | 44 |
| SYS.XML_SCHEMA_NAME_PRESENT.__dyn_sql_exec_line34 | 22078 µs | 22078 | 44 |

***Figure 3-8.*** *Profiling data of the call hierarchy displayed in SQL Developer*

## Using DBMS_PROFILER

With the `dbms_profiler` package you can enable and disable the line-level profiler at the session level. While enabled, the following information is gathered for each line of code that's executed:

- The total number of times it's executed
- The total amount of time that's spent executing it
- The minimum and maximum amount of time that's spent executing it

The gathering takes place at session level for all PL/SQL code that's neither wrapped nor natively compiled, and for which the user has the `CREATE` privilege. In other words, the privilege to execute a piece of PL/SQL code isn't enough to use the profiler. Therefore, in practice, the profiling is either done by the owner of the objects to be profiled, or by a user with the `CREATE ANY` privilege.

The profiling data is stored in the database tables shown in Figure 3-9. The `plsql_profiler_runs` table gives information about which profiling sessions have been performed. The `plsql_profiler_units` table provides the list of units that have been executed for each run. And the `plsql_profiler_data` table gives the profiling data, described earlier, for each line of code that has been executed.

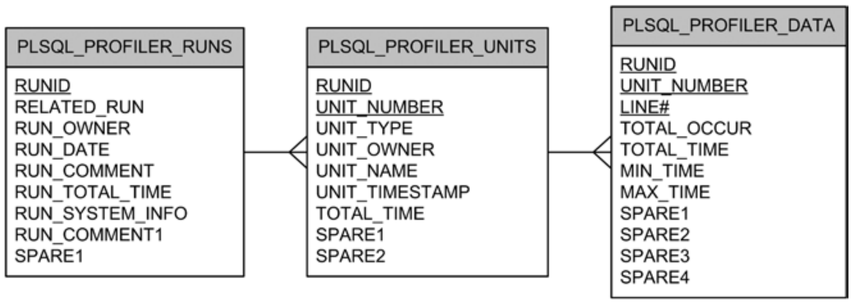


***Figure 3-9.** The profiler stores the gathered information in three database tables. Notice that the primary keys consist of the underlined columns*

## Installing the Output Tables

The package runs with the privileges of the user executing it. Consequently, the output tables don't necessarily need to be created by the sys user. Either the database administrator, as shown here, installs the output tables (by running the `proftab.sql` script) once and provides the necessary synonyms and privileges to use them, or each user installs them in his own schema:

```
CONNECT / AS SYSDBA
@?/rdbms/admin/proftab.sql

CREATE PUBLIC SYNONYM plsql_profiler_runs FOR plsql_profiler_runs;
CREATE PUBLIC SYNONYM plsql_profiler_units FOR plsql_profiler_units;
CREATE PUBLIC SYNONYM plsql_profiler_data FOR plsql_profiler_data;
CREATE PUBLIC SYNONYM plsql_profiler_runnumber FOR plsql_profiler_runnumber;

GRANT SELECT, INSERT, UPDATE, DELETE ON plsql_profiler_runs TO PUBLIC;
GRANT SELECT, INSERT, UPDATE, DELETE ON plsql_profiler_units TO PUBLIC;
GRANT SELECT, INSERT, UPDATE, DELETE ON plsql_profiler_data TO PUBLIC;
GRANT SELECT ON plsql_profiler_runnumber TO PUBLIC;
```

## Gathering the Profiling Data

A profiling analysis starts with enabling the profiler by calling the `start_profiler` routine. While the profiler is enabled, profiling data is gathered for the code executed by the PL/SQL engine. Unless an explicit flush is executed by calling the `flush_data` routine, no profiling data is stored in the output table while the profiler is enabled. The profiler is disabled, and an implicit flush is executed, by calling the `stop_profiler` routine. In addition, it's possible to pause and to resume the profiler by calling the `pause_profiler` and `resume_profiler` routines, respectively. Figure 3-10 shows the states of the profiler and the routines available in `dbms_profiler` that you can use to trigger a change of state.

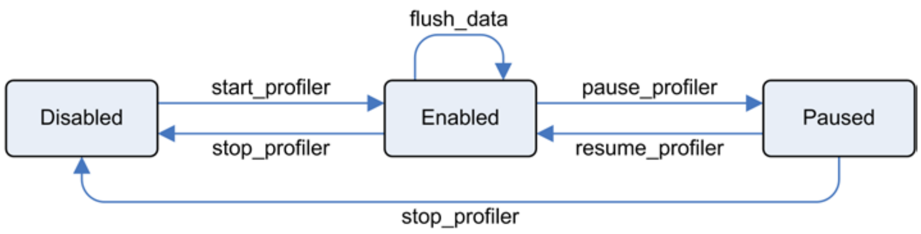


***Figure 3-10.*** *State diagram of the profiler. The `dbms_profiler` package provides routines for changing the state of the profiler: disabled, enabled, or paused*

For each routine shown in Figure 3-10, the package provides a function and a procedure. The functions return the processing result status (0 = successful). The procedures raise an exception in case of error. Except for the `start_profiler` routine, which accepts two comments describing the profiling analysis as a parameter, all other routines are parameterless.

The following example is an excerpt of the output generated by the `dbms_profiler.sql` script. Notice that the `runid` value selected while disabling the profiler is used in the next section to reference the profiling data stored in the output tables:

```
SQL> SELECT dbms_profiler.start_profiler AS status
  2  FROM dual;

    STATUS
----------
         0

SQL> execute perfect_triangles(1000)

SQL> SELECT dbms_profiler.stop_profiler AS status,
  2         plsql_profiler_runnumber.currval AS runid
  3  FROM dual;

    STATUS      RUNID
---------- ----------
         0          1
```

Once the profiling session is over, it's time to report the data generated by the profiler. The following two sections describe the two main methods for doing that.

## Manually Reporting the Profiling Data

Because the profiling data is stored in the output tables, it's possible to query that data with a regular query, as shown in this section. What follows is an excerpt of the output generated by the `dbms_profiler.sql` script. The query provides only the percentage for the response time for two reasons: first, because we're usually interested in spotting the slowest part of the code, and second, because the timing information, especially when the code is CPU bound, isn't very reliable. In fact, for CPU-bound processing, the overhead of the profiler may be very high. In this specific case, which is indeed CPU bound, the processing time increases from less than 1 second to about 7 seconds. Of these 7 seconds, only about 4 seconds are accounted for by the profiler:

```
SQL> SELECT s.line,
  2         round(ratio_to_report(p.total_time) OVER ()*100,1) AS time,
  3         total_occur,
  4         s.text
  5  FROM all_source s,
  6       (SELECT u.unit_owner, u.unit_name, u.unit_type,
  7               d.line#, d.total_time, d.total_occur
  8        FROM plsql_profiler_units u, plsql_profiler_data d
  9        WHERE u.runid = 1
 10        AND d.runid = u.runid
 11        AND d.unit_number = u.unit_number) p
 12  WHERE s.owner = p.unit_owner (+)
 13  AND s.name = p.unit_name (+)
 14  AND s.type = p.unit_type (+)
 15  AND s.line = p.line# (+)
 16  AND s.owner = user
 17  AND s.name = 'PERFECT_TRIANGLES'
 18  AND s.type IN ('PROCEDURE', 'PACKAGE BODY', 'TYPE BODY')
 19  ORDER BY s.line;

LINE#    TIME%      EXEC# CODE
------ -------- ---------- ------------------------------------------------------
     1      0.0          1 PROCEDURE perfect_triangles(p_max IN INTEGER) IS
...
    29     17.7  1,105,793     FOR j IN 1..n
    30                         LOOP
    31     22.3  1,105,614       IF p_long = dup_sides(j).long
    32                              AND
    33                              p_short = dup_sides(j).short
    34                           THEN
    35      0.0        855         RETURN FALSE;
    36                           END IF;
    37                         END LOOP;
...
    44      8.2    501,500     FOR short IN 1..long
    45                         LOOP
    46     21.4    500,500       hyp := sqrt(long*long + short*short);
    47     11.0    500,500       ihyp := floor(hyp);
    48     10.5    500,500       IF hyp-ihyp < 0.01
    49                           THEN
    50      0.2     10,325         IF ihyp*ihyp = long*long + short*short
    51                             THEN
    52      0.1      1,034           IF sides_are_unique(long, short)
    53                               THEN
    54      0.0        179             m := m+1;
    55      0.0        179             unique_sides(m).long := long;
    56      0.0        179             unique_sides(m).short := short;
    57      0.0        179             store_dup_sides(long, short);
    58                                END IF;
    59                              END IF;
    60                            END IF;
    61                          END LOOP;
...
    69       0.0          1 END perfect_triangles;
```

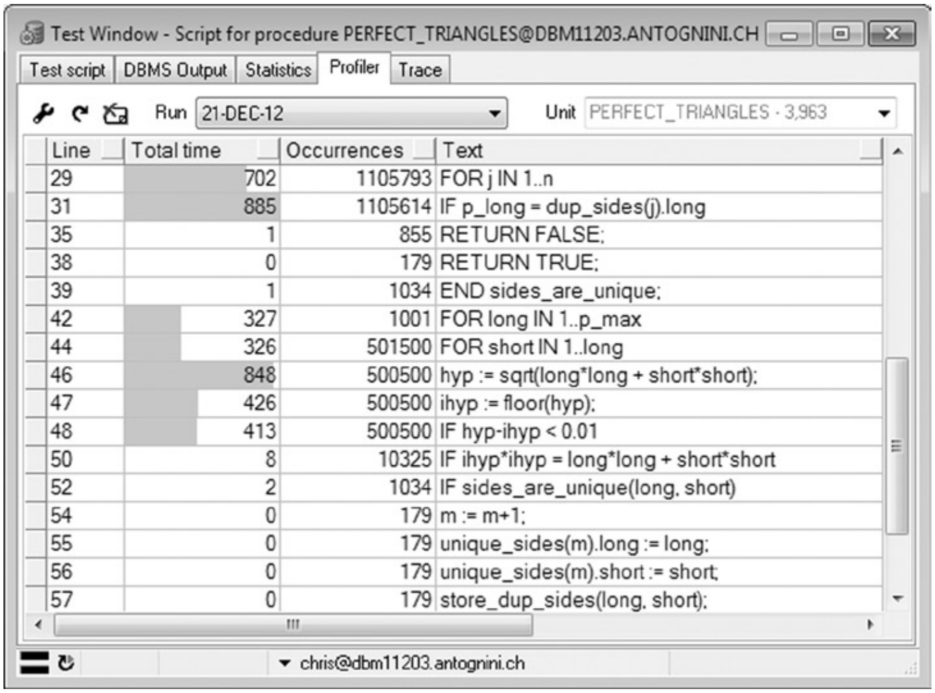
Oracle distributes two scripts that provide examples of queries against the profiling data:

- If the example files are installed (which isn't the case by default), a script called `profrep.sql` is available in the `$ORACLE_HOME/plsql/demo/` directory.
- Through the Oracle Support note *Script to produce HTML report with top consumers out of PL/SQL Profiler DBMS_PROFILER data* (243755.1).

## Using a GUI

In addition to the manual method covered in the previous section, it's also possible to use one of the graphical interfaces available in third-party products. Such an interface is provided by the major players such as PL/SQL Developer (Allround Automations), SQLDetective (Conquest Software Solutions), Toad and SQL Navigator (Dell), or Rapid SQL (Embarcadero). All these tools can be used to profile the code, usually by clicking a check box or button before running a test or by simply analyzing the content of the output tables.

As an example, Figure 3-11 shows the information provided by PL/SQL Developer for the profiling session illustrated in the previous sections. Notice the graphical representation in the "Total time" column that highlights the major time-consuming lines of code.



Test Window - Script for procedure PERFECT_TRIANGLES@DBM11203.ANTOGNINI.CH

Run 21-DEC-12    Unit PERFECT_TRIANGLES - 3,963

| Line | Total time | Occurrences | Text |
|---|---|---|---|
| 29 | 702 | 1105793 | FOR j IN 1..n |
| 31 | 885 | 1105614 | IF p_long = dup_sides(j).long |
| 35 | 1 | 855 | RETURN FALSE; |
| 38 | 0 | 179 | RETURN TRUE; |
| 39 | 1 | 1034 | END sides_are_unique; |
| 42 | 327 | 1001 | FOR long IN 1..p_max |
| 44 | 326 | 501500 | FOR short IN 1..long |
| 46 | 848 | 500500 | hyp := sqrt(long*long + short*short); |
| 47 | 426 | 500500 | ihyp := floor(hyp); |
| 48 | 413 | 500500 | IF hyp-ihyp < 0.01 |
| 50 | 8 | 10325 | IF ihyp*ihyp = long*long + short*short |
| 52 | 2 | 1034 | IF sides_are_unique(long, short) |
| 54 | 0 | 179 | m := m+1; |
| 55 | 0 | 179 | unique_sides(m).long := long; |
| 56 | 0 | 179 | unique_sides(m).short := short; |
| 57 | 0 | 179 | store_dup_sides(long, short); |

chris@dbm11203.antognini.ch

***Figure 3-11.*** *The profiling data displayed in PL/SQL Developer*

### Triggering the Profilers

Both profilers can only be enabled and disabled from within the session that executes the PL/SQL code to be profiled. If the profiling can't be manually started, it's also possible to create database triggers like the following ones to automatically enable and disable the profiler for a whole session:

```
CREATE TRIGGER start_hprof_profiler AFTER LOGON ON DATABASE
BEGIN
  IF (dbms_session.is_role_enabled('HPROF_PROFILE'))
  THEN
    dbms_hprof.start_profiling(
      location => 'PLSHPROF_DIR',
      filename => 'dbms_hprof_'||sys_context('userenv','sessionid')||'.trc'
    );
  END IF;
END;
/

CREATE TRIGGER stop_hprof_profiler BEFORE LOGOFF ON DATABASE
BEGIN
  IF (dbms_session.is_role_enabled('HPROF_PROFILE'))
  THEN
    dbms_hprof.stop_profiling();
  END IF;
END;
/
```

The previous triggers are for the hierarchical profiler. You can find the code to create them for both profilers in the `dbms_hprof_triggers.sql` and `dbms_profiler_triggers.sql` scripts. As the previous triggers show, to avoid enabling the profiler for all users, I usually suggest creating a role (`hprof_profile` in this example) and temporarily granting permission only to the user required for the test. Of course, it's also possible to define the triggers for a single schema or to make other checks based, for example, on the `userenv` context.

## On to Chapter 4

This chapter gives a detailed description of the tracing and profiling features provided by Oracle Database for identifying performance issues of reproducible problems. Specifically, it describes SQL trace with its related tools, and the two PL/SQL profilers externalized through the `dbms_hprof` and `dbms_profiler` packages. With those tools, you can home in without any doubt on the SQL statements or pieces of PL/SQL code that are causing the suboptimal performance that you're attempting to diagnose.

When you can't reproduce a problem or you have to analyze a problem while it's happening, the method described in this chapter is, most of the time, useless. In those cases you can apply the features described next, in Chapter 4.

CHAPTER 4

# Real-Time Analysis of Irreproducible Problems

Dynamic performance views provide key information for real-time analysis of performance problems. Because there are many such views, choosing the right ones and accessing them in the right order is of paramount importance to efficiently pinpoint a performance problem. To choose the right dynamic performance views, you also have to ask yourself one key question:

Do I have access to the features provided by the Diagnostics Pack and Tuning Pack options?

The question isn't technical at all. Nevertheless, answering it is necessary because you're allowed to use certain dynamic performance views, and therefore database features (the key features for real-time analysis are *active session history* and *real-time monitoring*), only when you have licensed the corresponding option. Be aware that the Diagnostics Pack option is a prerequisite for the Tuning Pack option. And because all options aren't available with Standard Edition, Enterprise Edition is a prerequisite as well. For detailed information about licensing, refer to the *Oracle Database Licensing Information* manual.

---

**Tip** If you don't have the license for either the Diagnostics Pack option or the Tuning Pack option, from version 11.1 onward you should set the `control_management_pack_access` initialization parameter accordingly. In Enterprise Edition, the default value is `diagnostic+tuning`. That means both options are enabled. Other valid values are `diagnostic` and `none`. The former enables only the Diagnostics Pack option, and the latter, which is the default value in Standard Edition, disables both options. There are basically two advantages to setting the initialization parameter to the right value. First, some features are disabled and, as a result, don't cause unnecessary overhead. Second, when using the Enterprise Manager, the pages that require one of the two options can't be accessed, which means you don't break your licensing agreement.

---

Independently of whether you're able to use the optional features or not, the steps you go through during an analysis don't change. Let's discuss them with the help of an analysis roadmap.

## Analysis Roadmap

The steps you go through during a perform analysis are summarized in Figure 4-1. Initially, you should check the load on the database server. Specifically, you should carry out two checks. First, find out whether the database server is CPU bound. If it is, many statistics might be artificially inflated. Hence, your first goal would be to find a method to reduce the CPU utilization. Second, check whether there are processes not related to the database instance you're focusing on that consume a lot of CPU. If that is the case, the cause of the performance problem might be due to reasons that you can't find by focusing on the database instance you're looking at.

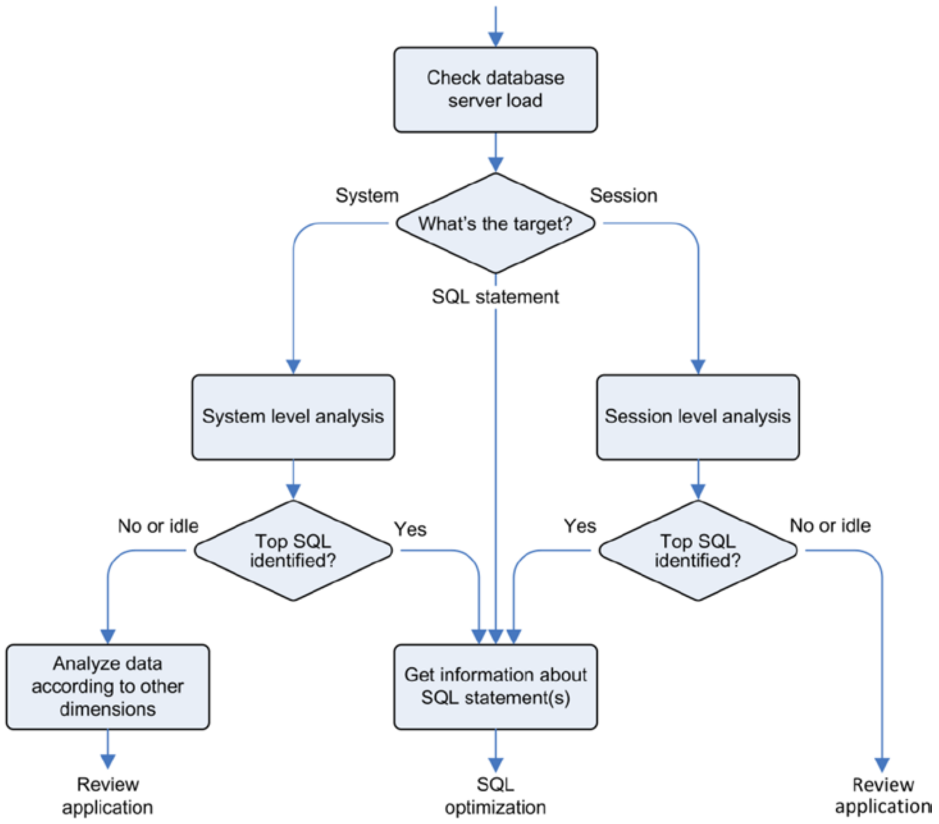


***Figure 4-1.*** *Roadmap for the real-time analysis of irreproducible problems*

After checking the database server load, there are three paths you can take. To determine which one to use, you have to ask yourself the following question:

> Do I have to target a single SQL statement, a single session, or the whole system?

If you already know that a specific SQL statement or session is experiencing performance problems, then you should focus on it. In fact, as described in Chapter 1, you should strive to focus on specific issues as much as possible. However, when no such information is available, you have to continue the analysis at the system level. For example, this happens when a general slowdown is experienced in the production environment and no clear culprit is known yet. In such a case, your goal isn't to troubleshoot a performance problem impacting a specific business task (what you should always strive to do), but to find a way to reduce the load on the system.

If you continue the analysis at the system or session level, your goal is to find whether a small number of SQL statements (let's say, a dozen) are responsible for most of the load. For example, if you see seven SQL statements that cause 85% of the load, you have clearly identified the top SQL statements you should focus on. However, if the top 10 SQL statements are responsible for only 25% of the load, focusing only on them is probably not worth your time.

You should also check whether there are a small number of "components" (for example, sessions, modules, or clients) that are responsible for most of the load. If that's the case, you can continue the analysis by focusing on those "components" only. In any case, if the analysis at the system level doesn't lead to the identification of the top SQL statements, you have to question the application and, therefore, start reviewing it to check whether the processing it does has been implemented in an efficient way. If the application code can't be analyzed (or modified), or the review doesn't lead to satisfying results, the only thing you can still consider is resource management. Simply put, you have two options. First, define parts of the application (for example, some sessions or users) to receive more resources than the others. Second, provide more resources (hardware) to the application. The latter is of course the last thing you should consider!

A special case to consider is when either the system or the session you're analyzing is (almost) idle. By *idle*, I mean that most of the processing time is spent outside the database system. For example, if you see a report that takes 13 minutes to run and, during that time, only 42 seconds are spent by the database engine processing some SQL statements related to it, independently of how many SQL statements were processed, focusing on the database tier is useless. Clearly, the bottleneck isn't to be located in the database tier. Again, the application or other parts of the infrastructure supporting the application should be reviewed.

Independently of whether you focus on a single SQL statement, a single session, or the whole system, for each top SQL statement you find, you have to gather the execution plan, key runtime statistics like number of processed rows and amount of CPU utilization, and the experienced wait events. To describe how to find this data, the next section provides basic information about important dynamic performance views that you need to know. Then the chapter continues by explaining in detail how to carry out a real-time analysis of a performance problem based on the information provided by the dynamic performance views.

## Dynamic Performance Views

Oracle Database provides dynamic performance views to externalize the content of some data structures that reside either in memory or in database files. In other words, even though they seem to be regular tables, the underlying structures that contain the data are completely different. Those structures are exposed in views to give easy access to their data through SQL.

Because the data structures, which the dynamic performance views rely on, are constantly changed by the database engine, the data provided through the dynamic performance views can constantly change as well. Be aware that not every dynamic performance view is updated in the same way. For example, some of them are continuously updated, and others are only updated every 5 seconds.

---

■ **Caution** For queries that access dynamic performance views, read consistency isn't guaranteed. So don't let small errors or inconsistencies confuse you.

---

When describing dynamic performance views, I typically reference the views with the `v$` prefix. If you're working on a RAC environment, be aware that the `v$` views show only information about the database instance you're connected to. If you require information about other database instances, you have to use the corresponding global views that have the `gv$` prefix. The structure of the `gv$` views is equivalent to the one of the `v$` views. In general, the only difference is that `gv$` views have an additional column (`inst_id`) identifying the database instance.

Some of the statistics provided by dynamic performance views depend on the `timed_statistics` initialization parameter, which can be set to either `TRUE` or `FALSE`. If it's set to `TRUE`, timing information is available. If it's set to `FALSE`, those statistics should be missing. However, depending on the platform you're working on, they could be available as well. The default value of timed_statistics depends on another initialization parameter: statistics_level. If statistics_level is set to basic, timed_statistics defaults to FALSE. Otherwise, timed_statistics defaults to TRUE. Because the default value of both initialization parameters is good, I don't recommend changing them at the system level.

There are many dynamic performance views. Let's review the ones that are frequently used for troubleshooting performance problems.

## OS Statistics

In case you don't have access to the database server console, but want to get some key performance figures like CPU and memory utilization at the operating system level, you might find what you're looking for through the v$osstat view. Its content, thanks to the comments column, is self-explanatory. Note that whereas some of the figures provided by the value column are cumulated since the database instance startup (for example, all the statistics providing timing information), others are constant (for example, the values relative to the number of sockets, CPUs and cores) or the current values (for example, the amount of free memory). Also be aware that the actual content depends on the Oracle Database version and platform you run it on. The following query shows an example based on version 12.1 on Linux:

```
SQL> SELECT stat_name, value, comments
  2  FROM v$osstat
  3  WHERE stat_name LIKE '%MEMORY_BYTES';

STAT_NAME                       VALUE COMMENTS
------------------------ ------------ ----------------------------------------------------
NUM_CPUS                            8 Number of active CPUs
IDLE_TIME                    29648458 Time (centi-secs) that CPUs have been in the idle state
BUSY_TIME                     6348349 Time (centi-secs) that CPUs have been in the busy state
USER_TIME                     4942391 Time (centi-secs) spent in user code
SYS_TIME                      1336523 Time (centi-secs) spent in the kernel
IOWAIT_TIME                   3806135 Time (centi-secs) spent waiting for IO
NICE_TIME                       22373 Time (centi-secs) spend in low-priority user code
RSRC_MGR_CPU_WAIT_TIME          14195 Time (centi-secs) processes spent in the runnable state
                                      waiting
LOAD                                1 Number of processes running or waiting on the run queue
NUM_CPU_CORES                       8 Number of CPU cores
NUM_CPU_SOCKETS                     2 Number of physical CPU sockets
PHYSICAL_MEMORY_BYTES     12619522048 Physical memory size in bytes
VM_IN_BYTES                         0 Bytes paged in due to virtual memory swapping
VM_OUT_BYTES                        0 Bytes paged out due to virtual memory swapping
FREE_MEMORY_BYTES          1529409536 Physical free memory in bytes
INACTIVE_MEMORY_BYTES      2112192512 Physical inactive memory in bytes
SWAP_FREE_BYTES            8603631616 Swap free in bytes
TCP_SEND_SIZE_MIN                4096 TCP Send Buffer Min Size
TCP_SEND_SIZE_DEFAULT           16384 TCP Send Buffer Default Size
TCP_SEND_SIZE_MAX             4194304 TCP Send Buffer Max Size
TCP_RECEIVE_SIZE_MIN             4096 TCP Receive Buffer Min Size
TCP_RECEIVE_SIZE_DEFAULT        87380 TCP Receive Buffer Default Size
TCP_RECEIVE_SIZE_MAX          6291456 TCP Receive Buffer Max Size
GLOBAL_SEND_SIZE_MAX          1048576 Global send size max (net.core.wmem_max)
GLOBAL_RECEIVE_SIZE_MAX       4194304 Global receive size max (net.core.rmem_max)
```

This information is especially useful to find out whether there are other applications consuming CPU on the database server. For that purpose, you need to compare the CPU utilization reported by time model statistics with the `BUSY_TIME` statistic. If they're close to each other, you know that most of the CPU is consumed by the database instance you're connected to.

## Time Model Statistics

You can know what kind of processing a database engine is doing on behalf of an application by looking at the *time model* statistics. The purpose of time model statistics is to show the amount of time spent performing key operations like opening new sessions, parsing SQL statements, and processing calls with one of the engines (SQL, PL/SQL, Java and OLAP) provided by Oracle Database. In addition, some figures about background processing are also given.

Time model statistics are based on a small number of figures organized in two independent trees: one for the background processing carried out by the database instance itself, and another for the foreground processing (the processing performed on behalf of an application). Figures 4-2 and 4-3 show not only the statistics associated with background processing and foreground processing, but also the relationship between them. For example, according to Figure 4-3, `parse time elapsed` is a child of `DB time` and the parent of `hard parse elapsed time`. The name of each statistic is basically self-explanatory. For a detailed description, refer to the "V$SESS_TIME_MODEL" section of the *Oracle Database Reference* manual.

```
background elapsed time
└background cpu time
  └RMAN cpu time (backup/restore)
```

***Figure 4-2.*** *The tree formed by time model statistics for background processing*

```
DB time
├DB CPU
├connection management call elapsed time
├sql execute elapsed time
├parse time elapsed
│ ├hard parse elapsed time
│ │ └hard parse (sharing criteria) elapsed time
│ │   └hard parse (bind mismatch) elapsed time
│ └failed parse elapsed time
│   └failed parse (out of shared memory) elapsed time
├repeated bind elapsed time
├sequence load elapsed time
├PL/SQL execution elapsed time
├inbound PL/SQL rpc elapsed time
├PL/SQL compilation elapsed time
├Java execution elapsed time
└OLAP engine elapsed time
  └OLAP engine CPU time
```

***Figure 4-3.*** *The tree formed by time model statistics for foreground processing*

Because time model statistics are separated into two trees, the total amount of processing can be calculated by summing up `DB time` and `background elapsed time`. Note that both statistics include CPU usage as well as the sum of all wait events except for the ones that belongs to the Idle wait class (the next section provides more information about wait classes).

The time reported by a child in the tree is contained within its parent in the tree. But be careful—this doesn't mean that adding up the time reported by all children gives the time reported by their parent. In fact, not only is the time of a specific operation not exclusively associated to a single child, but some operations aren't attributed to any child.

Time model statistics are externalized at the system level and for all connected sessions through the v$sys_time_model and v$sess_time_model views, respectively. In addition, in a 12.1 multitenant environment, the v$con_sys_time_model view shows statistics at the container level. In these dynamic performance views, there are two key columns:

- stat_name identifies the statistic, and
- value provides the amount of time (in microseconds) cumulated since the component they belong to (database instance, session or container) was initiated.

It goes without saying that for the session-level statistics (v$sess_time_model), there's also a column (sid) that identifies the session they belong to. And in a 12.1 multitenant environment, there's also a column (con_id) that identifies the container.

The following query, based on the v$sess_time_model view, shows how a specific session spent its processing time since it was started (97.3 percent of the time was spent executing SQL statements):

```
SQL> WITH
  2    db_time AS (SELECT sid, value
  3                FROM v$sess_time_model
  4                WHERE sid = 42
  5                AND stat_name = 'DB time')
  6  SELECT ses.stat_name AS statistic,
  7         round(ses.value / 1E6, 3) AS seconds,
  8         round(ses.value / nullif(tot.value, 0) * 1E2, 1) AS "%"
  9  FROM v$sess_time_model ses, db_time tot
 10  WHERE ses.sid = tot.sid
 11  AND ses.stat_name <> 'DB time'
 12  AND ses.value > 0
 13  ORDER BY ses.value DESC;

STATISTIC                                SECONDS     %
---------------------------------------- ------- -----
sql execute elapsed time                  99.437  97.3
DB CPU                                      4.46   4.4
parse time elapsed                         0.308   0.3
connection management call elapsed time    0.004   0.0
PL/SQL execution elapsed time              0.000   0.0
repeated bind elapsed time                 0.000   0.0
```

Notice that in this example, the percentages are computed based on the value DB time, which is the overall elapsed time spent by the database engine processing user calls. Because DB time accounts only for the database processing time, the time spent by the database engine waiting on user calls isn't included. As a result, with only the information provided by the time model statistics, you can't know whether the problem is located inside or outside the database. In addition, any difference between elapsed time and CPU time can't be explained based on time model statistics (for example, only 4.4 percent of the time is spent on CPU in the previous example). To know exactly what's going on, information about wait classes and wait events (covered in the next section) is necessary.

**AVERAGE NUMBER OF ACTIVE SESSIONS**

The average number of active sessions (AAS) is the rate at which the DB time increases at the system level. For example, if the DB time of a database instance increases by 1,860 seconds in 60 seconds, the average number of active sessions is 31 (1,860/60). This means that, during the 60-second period, on average 31 sessions were actively processing user calls.

The average number of active sessions at the system level is an important metric because it shows you how much a system is loaded. As a rule of thumb, values much lower than the number of CPU cores mean that the system is almost idle. Inversely, values much higher than the number of CPU cores mean that the system is quite busy. That said, because it's an average, it can hide a lot of information. So, be careful when basing a judgment solely on this statistic. This is especially true when the period of time covers more than a few minutes.

The rate at which the DB time increases can also be computed for a single session. In that case, it's a value between 0 and 1 that indicates the degree to which the session is active. If the value is 0, it means that the session is completely idle. In other words, the database engine is doing no processing. If the value is 1, it means that the session is completely busy processing user calls.

## Wait Classes and Wait Events

Based on time model statistics, you can determine not only how much processing time a database instance (or a session or a container) is consuming, but also how much CPU it's using for that processing. When the two values are equivalent, it means that the database instance isn't experiencing any wait because of things like disk I/O operations, network round-trips, or locks. However, when the difference between the two values is significant, for analyzing a performance issue, you need to know which waits the server processes are waiting for. This information is provided through wait events.

Because the number of wait events is very high (there are more than 1,500 of them in version 12.1), to simplify the production of resource usage profiles, wait events are organized into 13 wait classes (note that there are 12 in version 10.2). Which wait events are available in the version you're using and the wait class they belong to is visible through the `v$event_name` view. For example, the following queries show which wait classes exist in version 12.1.0.1, how many wait events they contain, and the wait events belonging to the wait class with fewer wait events (`Commit`):

```
SQL> SELECT wait_class, count(*)
  2  FROM v$event_name
  3  GROUP BY rollup(wait_class)
  4  ORDER BY wait_class;

WAIT_CLASS     COUNT(*)
-------------- --------
Administrative       57
Application          17
Cluster              57
Commit                4
Concurrency          34
Configuration        26
Idle                119
Network              28
Other              1123
Queueing              9
```

```
Scheduler             10
System I/O            34
User I/O              51
                    1569

SQL> SELECT name
  2  FROM v$event_name
  3  WHERE wait_class = 'Commit';

NAME
----------------------------------
remote log force - commit
log file sync
nologging standby txn commit
enq: BB - 2PC across RAC instances
```

The classes of wait events that have been experienced at the system level and for all connected sessions are externalized through the `v$system_wait_class` and `v$session_wait_class` views, respectively. In addition, in a 12.1 multitenant environment, the `v$con_system_wait_class` view shows statistics at the container level. These dynamic performance views have three key columns:

- `wait_class` identifies the wait class,
- `total_waits` is the number of wait events cumulated since the component they belong to (database instance, session or container) was initiated, and
- `time_waited` is the amount of waiting time (in hundredths of a second) cumulated since the component they belong to (database instance, session or container) was initiated.

It goes without saying that for the session-level statistics (`v$session_wait_class`), there's also a column (`sid`) that identifies the session they belong to, and in a 12.1 multitenant environment, there's a column (`con_id`) that identifies the container. The following example is based on the same session shown in the previous section, the one that spends only 4.4 percent of the DB time on CPU. The example illustrates how to produce a concise resource usage profile for the processing carried out by a session. You can use a similar query for the system level, as well as for the container level. Just change the query to reference the dynamic performance views for the level you are interested in.

```
SQL> SELECT wait_class,
  2         round(time_waited, 3) AS time_waited,
  3         round(1E2 * ratio_to_report(time_waited) OVER (), 1) AS "%"
  4  FROM (
  5    SELECT sid, wait_class, time_waited / 1E2 AS time_waited
  6    FROM v$session_wait_class
  7    WHERE total_waits > 0
  8    UNION ALL
  9    SELECT sid, 'CPU', value / 1E6
 10    FROM v$sess_time_model
 11    WHERE stat_name = 'DB CPU'
 12  )
 13  WHERE sid = 42
 14  ORDER BY 2 DESC;
```

```
WAIT_CLASS    TIME_WAITED     %
------------- ----------- -----
Idle               154.77  60.2
User I/O            96.99  37.7
CPU                  4.46   1.7
Commit               0.85   0.3
Network              0.04   0.0
Configuration        0.03   0.0
Concurrency          0.02   0.0
Application          0.01   0.0
```

The resource usage profile based on wait classes is a beginning, but most of the time you need more precise information. You need the wait events. For this purpose, the database engine also externalizes information about wait events at the system level and for all connected sessions through the `v$system_event` and `v$session_event` views, respectively. In addition, in a 12.1 multitenant environment, the `v$con_system_event` view shows statistics at the container level. The following query, which was executed for the same session as the previous one, illustrates how produce a detailed resource usage profile for the processing carried out by a session (you can use a similar query for the system level as well as the container level, just use the dynamic performance views of the level you're interested in):

```
SQL> SELECT event,
  2         round(time_waited, 3) AS time_waited,
  3         round(1E2 * ratio_to_report(time_waited) OVER (), 1) AS "%"
  4  FROM (
  5    SELECT sid, event, time_waited_micro / 1E6 AS time_waited
  6    FROM v$session_event
  7    WHERE total_waits > 0
  8    UNION ALL
  9    SELECT sid, 'CPU', value / 1E6
 10    FROM v$sess_time_model
 11    WHERE stat_name = 'DB CPU'
 12  )
 13  WHERE sid = 42
 14  ORDER BY 2 DESC;

EVENT                         TIME_WAITED     %
----------------------------- ----------- -----
SQL*Net message from client       154.790  60.2
db file sequential read            96.125  37.4
CPU                                 4.461   1.7
log file sync                       0.850   0.3
read by other session               0.734   0.3
db file parallel read               0.135   0.1
SQL*Net message to client           0.044   0.0
cursor: pin S                       0.022   0.0
enq: TX - row lock contention       0.011   0.0
Disk file operations I/O            0.001   0.0
latch: In memory undo latch         0.001   0.0
```

In the preceding output, it's interesting to notice that the DB time accounts for only 39.8 percent (100 – 60.2) of the total elapsed time. In fact, 60.2 percent is related to an idle wait event (`SQL*Net message from client`). This indicates that for 60.2 percent of the time, the database engine is waiting for the application to submit some work. The other essential information provided by this resource usage profile is that, when the database engine is processing user calls, it's almost always doing disk I/O operations that read a single block (`db file sequential read`). All other wait events and the CPU utilization are negligible.

For some wait events, like the one associated with disk I/O operations, you might want to get information about the average latency. In fact, if you have that information, you can compare the experienced performance with the expected performance (you should know what to expect from the disk I/O subsystem used to store the database). Based on a view like `v$system_event`, you can, for example, run a query to compute the average latency of a specific wait event:

```
SQL> SELECT time_waited_micro/total_waits/1E3 AS avg_wait_ms
  2  FROM v$system_event
  3  WHERE event = 'db file sequential read';

AVG_WAIT_MS
-----------
 9.52927176
```

Given that an average value like the one computed by the preceding query hides a lot of information, Oracle Database provides a view that presents, at the system level, a histogram for each wait event. The view is `v$event_histogram`. It has three key columns:

- `event` is the name of the wait event,
- `wait_time_milli` represents the interval's upper limit (which isn't included) of the histogram's bucket, and
- `wait_count` is the number of wait events associated to the histogram's bucket.

For example, the following query shows that while a plurality (45.7 percent) of the wait events took between 4 and 8 milliseconds, about 24 percent (3.27 + 2.75 + 18.37) took less than 4 milliseconds, and about 10 percent (5.96 + 2.66 + 1.34 + 0.17 + 0.01) took 16 milliseconds or more:

```
SQL> SELECT wait_time_milli, wait_count, 100*ratio_to_report(wait_count) OVER () AS "%"
  2  FROM v$event_histogram
  3  WHERE event = 'db file sequential read';

WAIT_TIME_MILLI WAIT_COUNT      %
--------------- ---------- ------
              1     348528   3.27
              2     293508   2.75
              4    1958584  18.37
              8    4871214  45.70
             16    2106649  19.76
             32     635484   5.96
             64     284040   2.66
            128     143030   1.34
            256      18041   0.17
            512        588   0.01
           1024        105   0.00
           2048          1   0.00
```

In statistics like the previous ones, it's especially interesting to check what the maximum values are. In this example, notice that, according to typical expectations, some disk I/O operations were far too long (between 64 milliseconds and 2 seconds). Even though there weren't many, either the disk I/O subsystem (or one of its components) is undersized or there's a configuration or hardware problem.

## System and Session Statistics

In addition to time model statistics and wait events, the database engine also records hundreds (more than 850 in version 12.1) of additional statistics providing information like the number of times a specific operation has been performed or the amount of data processed by a specific functionality. These statistics are externalized at the system level and for all connected sessions through the `v$sysstat` and `v$sesstat` views, respectively. In addition, in a 12.1 multitenant environment, the `v$con_sysstat` view shows statistics at the container level.

There are two key columns in the `v$sysstat` view:

- `name` identifies the statistic (refer to the *Oracle Database Reference* manual for a short description of most of them), and
- `value` provides the figure associated to the statistic itself. In most cases, the value is cumulated since the database instance startup, but this isn't the case for all statistics.

To show you examples of the kind of information you can retrieve through a dynamic performance view like `v$sysstat`, let's have a look at two queries. The first one retrieves statistics based on counters that are constantly incremented. In this case, those counters represent the number of logons, commits, and in-memory sorts since the database instance startup:

```
SQL> SELECT name, value
  2  FROM v$sysstat
  3  WHERE name IN ('logons cumulative', 'user commits', 'sorts (memory)');

NAME                 VALUE
----------------- --------
logons cumulative     1422
user commits       1298103
sorts (memory)      770169
```

The second query retrieves statistics showing the amount of data processed through disk I/O operations:

```
SQL> SELECT name, value
  2  FROM v$sysstat
  3  WHERE name LIKE 'physical % total bytes';

NAME                             VALUE
-------------------------- -----------
physical read total bytes   9.1924E+10
physical write total bytes  4.2358E+10
```

The structure of the `v$con_sysstat` view is exactly the same as that of `v$sysstat`. However, the `v$sesstat` view is a bit different. Although it does provide a column (`sid`) that's used to identify the session that the statistics belong to, it doesn't provide the `name` column. To get the name of the statistic, it's necessary to join `v$sesstat` with another view that provides a list of all available statistics: `v$statname`. The following query illustrates how to use these two views for retrieving information about the PGA memory utilization for the current session (two values are returned—the amount that is currently allocated and the maximum amount allocated since the session was initialized):

```
SQL> SELECT sn.name, ss.value
  2  FROM v$statname sn, v$sesstat ss
  3  WHERE sn.statistic# = ss.statistic#
  4  AND sn.name LIKE 'session pga memory%'
  5  AND ss.sid = sys_context('userenv','sid');
```

```
NAME                     VALUE
---------------------- --------
session pga memory      1723880
session pga memory max  2313704
```

To avoid the restriction based on the `sid` column shown in the preceding query, you can use the `v$mystat` view. In fact, for the session querying it, it provides exactly the same information as `v$sesstat`. The only difference is that only statistics about the current session are shown.

---

■ **Tip** In most situations, an analysis starts by breaking up the response time into CPU consumption and wait events. If a session is always on CPU and, therefore, doesn't experience any wait event, session statistics might be very useful to understand what the session is doing.

---

## Metrics

Many of the statistics provided by the dynamic performance views described in the previous sections are cumulated values. Based on them, the database engine computes a number of metrics (depending on the version, between 200 and 300) that are particularly useful for monitoring purposes. The available metrics are listed in the `v$metricname` view. As an example, from version 11.2 onward, there's a metric that shows the database server's CPU usage per second (which is based on the OS statistics). The following query shows, for that metric, the content of the `v$metricname` view:

```
SQL> SELECT metric_id, metric_unit, group_id, group_name
  2  FROM v$metricname
  3  WHERE metric_name = 'Host CPU Usage Per Sec';

METRIC_ID METRIC_UNIT             GROUP_ID GROUP_NAME
--------- ---------------------- -------- ----------------------------
     2155 CentiSeconds Per Second        2 System Metrics Long Duration
     2155 CentiSeconds Per Second        3 System Metrics Short Duration
```

As you can see from the output of the previous query, a metric has an ID, a unit of measurement, and is associated to one or several groups (two in the previous case) that have an ID and a name.

Be aware that metrics are computed based on a number of units of measurement. Some of them, like the previous one, express the utilization or the number of events per second. Others are computed per transaction, per request, per call, or are averages of absolute values.

A group to which a metric is associated defines the interval of computation and for how long information about it is provided. If, as in the previous case, a metric is associated to two groups, it means that the database engine computes two separate metrics, which have a specific interval and retention. Information about the groups is provided in the `v$metricgroup` view. The following groups exist in version 12.1 (other versions might have a different number of groups):

```
SQL> SELECT *
  2  FROM v$metricgroup
  3  ORDER BY group_id;
  GROUP_ID NAME                              INTERVAL_SIZE MAX_INTERVAL
---------- ------------------------------- ------------- ------------
         0 Event Metrics                            6000            1
         1 Event Class Metrics                      6000           60
         2 System Metrics Long Duration             6000           60
         3 System Metrics Short Duration            1500           12
         4 Session Metrics Long Duration            6000           60
         5 Session Metrics Short Duration           1500            1
         6 Service Metrics                          6000           60
         7 File Metrics Long Duration              60000            6
         9 Tablespace Metrics Long Duration         6000            0
        10 Service Metrics (Short)                   500           24
        11 I/O Stats by Function Metrics            6000           60
        12 Resource Manager Stats                   6000           60
        13 WCR metrics                              6000           60
        14 WLM PC Metrics                            500           24
```

The interval_size column shows the interval at which the metrics associated to a group are computed in hundredths of a second. For example, the metrics associated to the System Metrics Long Duration group are computed every 60 seconds. The max_interval column shows the maximum number of values that are provided. For example, up to 60 values are provided for the metrics associated to the System Metrics Long Duration group. As a result, for this group, information for the last hour should be available.

The values of the metrics themselves are provided through a number of views. In fact, there are views that are specific for some metric groups. For example, for the metric groups 2 and 3, the v$sysmetric and v$sysmetric_history views show the current value and the history, respectively. Anyway, for simplicity, for most metrics you can also use the v$metric and v$metric_history views. To illustrate, the following query shows the current metrics for the Host CPU Usage Per Sec metric (notice that the first metric is computed over 60 seconds and the second metric over 15 seconds):

```
SQL> SELECT begin_time, end_time, value, metric_unit
  2  FROM v$metric
  3  WHERE metric_name = 'Host CPU Usage Per Sec';

BEGIN_TIME          END_TIME                 VALUE METRIC_UNIT
------------------- ------------------- ---------- -----------------------
2014-04-28 01:56:00 2014-04-28 01:57:00 168.137173 CentiSeconds Per Second
2014-04-28 01:56:45 2014-04-28 01:57:00 159.786951 CentiSeconds Per Second
```

## Current Sessions Status

Through the v$session view, you can not only know what sessions currently exist, but also what they're presently doing. Because this dynamic performance view has many columns (for example, there are 82 in version 10.2.0.5 and 101 in version 12.1.0.1), they're not fully covered here (refer to the *Oracle Database Reference* manual for additional information). The most important information you can extract from the v$session view and which column it's found in, is the following:

- The identification of the session (sid, serial#, saddr and audsid), whether it's a BACKGROUND or USER session (type), and when it was initialized (logon_time).
- The identification of the user that opened the session (username and user#), the current schema (schemaname), and the name of the service used to connect to the database engine (service_name).

- The application using the session (program), which machine it was started on (machine), which process ID it has (process), and the name of the OS user who started it (osuser).
- The type of server-side process (server) which can be either DEDICATED, SHARED, PSEUDO, POOLED or NONE, and its address (paddr).
- The address of the currently active transaction (taddr).
- The status of the session (status) which can be either ACTIVE, INACTIVE, KILLED, SNIPED, or CACHED and how many seconds it's been in that state for (last_call_et). When investigating a performance problem, you're usually interested in the sessions marked as ACTIVE only.
- The type of the SQL statement in execution (command), the identification of the cursor related to it (sql_address, sql_hash_value, sql_id and sql_child_number), when the execution was started (sql_exec_start), and its execution ID (sql_exec_id). The execution ID is an integer value that identifies, along with sql_exec_start, one specific execution. It's necessary because the same cursor can be executed several times per second (note that the datatype of the sql_exec_start column is DATE).
- The identification of the previous cursor that was executed (prev_sql_address, prev_hash_value, prev_sql_id and prev_child_number), when the previous execution was started (prev_exec_start), and its execution ID (prev_exec_id).
- If a PL/SQL call is in execution, the identification of the top-level program and subprogram (plsql_entry_object_id and plsql_entry_subprogram_id) that was called, and the program and subprogram (plsql_object_id and plsql_subprogram_id) that is currently in execution. Note that plsql_object_id and plsql_subprogram_id are set to NULL if the session is executing a SQL statement.
- The session attributes (client_identifier, module, action, and client_info), if the application using the session sets them.
- If the session is currently waiting (in which case the state column is set to WAITING), the name of the wait event it's waiting for (event), its wait class (wait_class and wait_class#), details about the wait event (p1text, p1, p1raw, p2text, p2, p2raw, p3text, p3, and p3raw), and how much time the session has been waiting for that wait event (seconds_in_wait and, from 11.1 onward, wait_time_micro). Be aware that if the state column isn't equal to WAITING, the session is on CPU (provided the status column is equal to ACTIVE). In this case, the columns related to the wait event contain information about the last wait.
- Whether the session is blocked by another session (if this is the case, blocking_session_status is set to VALID) and, if the session is waiting, which session is blocking it (blocking_instance and blocking_session).
- If the session is currently blocked and waiting for a specific row (for example, for a row lock), the identification of the row it's waiting for (row_wait_obj#, row_wait_file#, row_wait_block#, and row_wait_row#). If the session isn't waiting for a locked row, the row_wait_obj# column is equal to the value -1.

In addition to the v$session view, there are other dynamic performance views that are specialized in providing specific information. For example, v$session_wait provides only columns related to wait events, and v$session_blockers provides only columns related to blocked sessions.

## Active Session History

As discussed in the preceding section, through the `v$session` view you can know what the current status of every existing session is. Even though such information might be useful, it's not sufficient to analyze a performance problem. In fact, for a successful analysis, you have to know what a session does over a period of time, and not only at a particular moment. Providing historical information about the status of sessions is the purpose of *active session history* (ASH).

---

■ **Note** To use active session history, the Diagnostics Pack option must be licensed. If the `control_management_pack_access` initialization is set to `none`, active session history is disabled.

---

The key advantage of active session history compared to SQL trace is that the former is always enabled and, therefore, is always available when required. Exactly for that reason, it's useful to analyze performance problems that can't be reproduced. You simply wait until the system experiences it and then analyze the information stored in the active session history.

To build the historical information provided by active session history, a background process (MMNL) carries out the following operations at an interval of one second:

- Sample the status of all sessions (it performs something similar to a query on the `v$session` view).
- Discard the data about sessions that are waiting for an event associated to the Idle wait class.
- Store the remaining data into a memory buffer in the SGA.

The sampling is illustrated in Figure 4-4. Notice, for example, how at time 06:28:12 session 1 is waiting on an event associated to the User I/O class, session 2 is on CPU, and sessions 3 and 4 are idle.

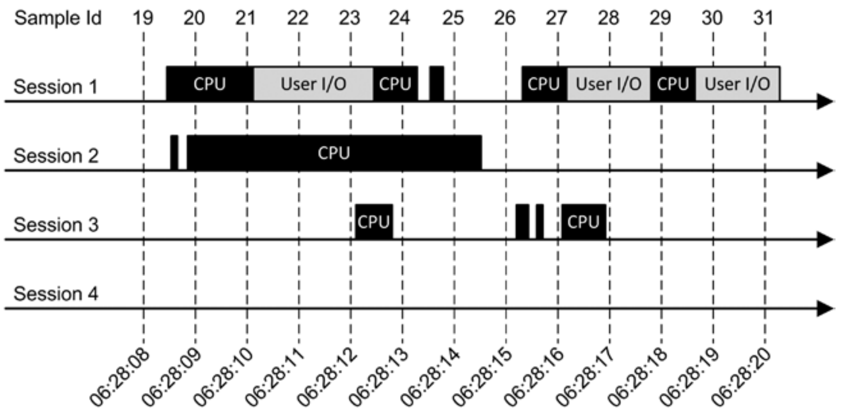


***Figure 4-4.*** *The MMNL process samples the status of all sessions at an interval of one second*

When MMNL samples the sessions shown in Figure 4-4, it produces data similar to what's summarized in Table 4-1. There are two important things to notice. First, operations lasting at least one second are always part of the samples stored in active session history. Second, even though session 3 had some activity, no sample is stored for it in active session history.

***Table 4-1.** The Samples Produced by MMNL for the Load Illustrated by Figure 4-4*

| Sample ID | Timestamp | Session | SQL ID | Activity |
|---|---|---|---|---|
| 20 | 06:28:09 | 1 | gd90ygn1j4026 | CPU |
| 20 | 06:28:09 | 2 | 5m6mu5pd9w028 | CPU |
| 21 | 06:28:10 | 1 | gd90ygn1j4026 | CPU |
| 21 | 06:28:10 | 2 | 5m6mu5pd9w028 | CPU |
| 22 | 06:28:11 | 1 | gd90ygn1j4026 | User I/O |
| 22 | 06:28:11 | 2 | 5m6mu5pd9w028 | CPU |
| 23 | 06:28:12 | 1 | gd90ygn1j4026 | User I/O |
| 23 | 06:28:12 | 2 | 5m6mu5pd9w028 | CPU |
| 24 | 06:28:13 | 1 | 7ztv2z24kw0s0 | CPU |
| 24 | 06:28:13 | 2 | 5m6mu5pd9w028 | CPU |
| 25 | 06:28:14 | 2 | 5m6mu5pd9w028 | CPU |
| 27 | 06:28:16 | 1 | d9gdx5a4gc13y | CPU |
| 28 | 06:28:17 | 1 | 1uaz41wrxw03k | User I/O |
| 29 | 06:28:18 | 1 | 1uaz41wrxw03k | CPU |
| 30 | 06:28:19 | 1 | 1uaz41wrxw03k | User I/O |
| 31 | 06:28:20 | 1 | 1uaz41wrxw03k | User I/O |

Based on the data shown in Table 4-1, it's possible to derive the following information:

- Session 1 was active for about 83% of the time (10 samples out of 12 seconds) and executed at least four distinct SQL statements. Half of the processing time (5 samples out of 10) was spent on CPU, and the other half doing disk I/O operations.
- Session 2 was active for about 50% of the time (6 samples out of 12 seconds). During that time it was always on CPU, executing a SQL statement identified by the ID `5m6mu5pd9w028`.
- Session 3 and session 4 were 100% idle (0 samples out of 12 seconds).

## ACTIVE SESSION HISTORY BUFFER

The buffer used to store active session history in the SGA is automatically dimensioned when the database instance starts. Oracle's design goal is to hold one hour of activity in memory. To know how large the buffer is and how much time it stores information for, you can execute the following queries:

```
SQL> SELECT pool, bytes
  2  FROM v$sgastat
  3  WHERE name = 'ASH buffers';
```

```
POOL            BYTES
----------- ---------
shared pool  14680064

SQL> SELECT max(sample_time) - min(sample_time) AS interval
  2  FROM v$active_session_history;

INTERVAL
-----------------------
+000000000 02:25:30.293
```

The data stored in active session history is externalized through the `v$active_session_history` view. Many of the columns are equivalent to the ones of the `v$session` view (refer to the "Current Session Status" section for information about them). As for the `v$session` view, the columns provided by the `v$active_session_history` view strongly depend on the version you're using. For example, there are 50 in version 10.2.0.5 and 101 in version 12.1.0.1). Compared to the `v$session` view, some of the columns are missing, some have slightly different content, and others are available only in the `v$active_session_history` view. The most important columns that are available only in the `v$active_session_history` view, or that have different contents, are the following (refer to the *Oracle Database Reference* manual for additional information):

- `sample_id` is an identifier associated to all samples taken by MMNL at a given time. Be aware that it doesn't identify a single row in active session history.
- `sample_time` is the timestamp at which MMNL takes the sample. Therefore, based on it, you can reconstruct the activity performed by every session.
- `session_state` expresses what the state of the session is. It's set to either `WAITING` or `ON CPU`.
- If the session is waiting, `time_waited` is the amount of time in microseconds that the session spent waiting. In case one wait event occurrence spans two or more samples, the actual time spent waiting is provided by the last sample and, for all other samples, `time_waited` is set to 0. To illustrate, the output of the following query shows a session that waited for about 7.4 seconds for a lock:

```
SQL> SELECT sample_time, event, time_waited
  2  FROM v$active_session_history
  3  WHERE session_id = 137
  4  ORDER BY sample_time;

SAMPLE_TIME                EVENT                TIME_WAITED
-------------------------- -------------------- -----------
...
27-APR-14 03.10.50.245 PM  enq: TM - contention           0
27-APR-14 03.10.51.245 PM  enq: TM - contention           0
27-APR-14 03.10.52.245 PM  enq: TM - contention           0
27-APR-14 03.10.53.245 PM  enq: TM - contention           0
27-APR-14 03.10.54.245 PM  enq: TM - contention           0
27-APR-14 03.10.55.245 PM  enq: TM - contention           0
27-APR-14 03.10.56.245 PM  enq: TM - contention           0
27-APR-14 03.10.57.245 PM  enq: TM - contention     7390676
...
```

- A number of columns provide information about the execution plan associated to the SQL statement that's in execution. Specifically, its hash value (`sql_plan_hash_value`) and the active operation (`sql_plan_line_id`, `sql_plan_operation` and `sql_plan_options`).
- From 11.1 onward, a number of flags point out what operation, according to the categories defined for time model statistics, is in execution (`in_connection_mgmt`, `in_parse`, `in_hard_parse`, `in_sql_execution`, `in_plsql_execution`, `in_plsql_rpc`, `in_plsql_compilation`, `in_java_execution`, `in_bind`, `in_cursor_close` and, as of version 11.2, `in_sequence_load`).
- For SQL statements executed in parallel, information about the query coordinator is given (`qc_instance_id`, `qc_session_id` and, as of version 11.1 `qc_session_serial#`).

With the information provided through the `v$active_session_history` view, you can perform a statistical analysis on how the database engine spends time during the processing. Let me stress that, because the data is based on sampling, it's a statistical analysis. Therefore, more samples lead to more accurate results. In any case, because only one sample per second and per active session is kept, the available information isn't as accurate as a method not based on sampling (for example, SQL trace). However, in many situations, the analysis provides sufficient information to pinpoint the cause of a performance problem.

The typical query executed against the `v$active_session_history` view is composed of the following parts:

- A restriction on `sample_time` to focus on a specific time period.
- An aggregation based on one or several columns providing context information about the processing like the ID of the session (`session_id`), the SQL statement associated to the cursor in execution (`sql_id`), or the application that carried out the processing (`program`).
- A count of the number of samples. Because every sample accounts for one second, the number of samples is an approximation of DB time in seconds.

---

■ **Caution** To estimate values like the DB time, the CPU time, or the amount of time spent waiting on events, you have to count the number of samples. Be aware that aggregating data with a simple expression like `sum(time_waited)` is the wrong way to do it! That's because the probability of an event being sampled is proportional to the length of the event.

---

For example, the following query retrieves the top 10 SQL statements ordered by their DB time for a period of 5 minutes (notice, for example, that the first SQL statement spent about 1,008 seconds and is responsible for about 29.9% of the total DB time):

```
SQL> SELECT activity_pct,
  2         db_time,
  3         sql_id
  4  FROM (
  5    SELECT round(100 * ratio_to_report(count(*)) OVER (), 1) AS activity_pct,
  6           count(*) AS db_time,
  7           sql_id
  8    FROM v$active_session_history
  9    WHERE sample_time BETWEEN to_timestamp('2014-02-12 22:12:30', 'YYYY-MM-DD HH24:MI:SS')
 10                          AND to_timestamp('2014-02-12 22:17:30', 'YYYY-MM-DD HH24:MI:SS')
 11    AND sql_id IS NOT NULL
 12    GROUP BY sql_id
 13    ORDER BY count(*) DESC
 14  )
 15  WHERE rownum <= 10;
```

```
ACTIVITY_PCT    DB_TIME SQL_ID
------------ ---------- -------------
        29.9       1008 c13sma6rkr27c
        11.3        382 0yas01u2p9ch4
        11.2        376 0y1prvxqc2ra9
         9.5        321 7hk2m2702ua0g
         8.2        277 bymb3ujkr3ubk
         7.8        263 8dq0v1mjngj7t
         5.8        196 8z3542ffmp562
         4.2        142 0bzhqhhj9mpaa
         2.8         93 5mddt5kt45rg3
         1.3         44 0w2qpuc6u2zsp
```

If you have access to Enterprise Manager 12c (Cloud Control or Express), you can also query the content of active session history with the help of *ASH Analytics*. The purpose of ASH Analytics is to allow you to perform an analysis without having to write SQL queries that directly access the `v$active_session_history` view. Simply put, with it you select the period by positioning a slider on a chart providing a timeline and overview of the load experienced by the database instance (illustrated by Figure 4-5), pick out one or several dimensions according to which one you want to aggregate the data to (a dropdown list containing more than two dozen alternatives is available), and choose how to format the displayed data.

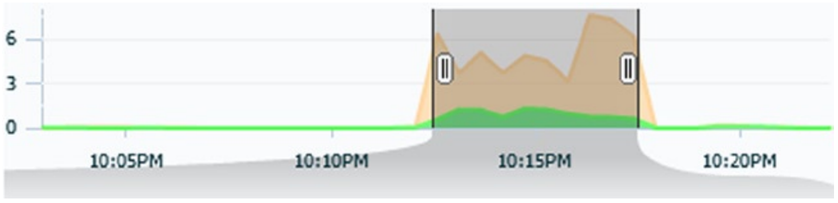


***Figure 4-5.*** *Selecting the time period of the analysis in ASH Analytics*

---

■ **Note** Even though ASH Analytics is an Enterprise Manager feature, it requires some object to be installed database side. Those objects are installed by default only from version 11.2.0.4 onward. In previous versions, an installation is required. When you try to use ASH Analytics on a database without those objects, Enterprise Manager suggests you install them. Be aware that ASH Analytics isn't available in version 10.2.

---

The data can be arranged according to three main formats:

- An *activity chart* shows, for the selected dimension, the variation in the average number of active session over the selected period of time. As an example, Figure 4-6 shows the number of active sessions for the top 10 SQL statements over the 5-minute period selected in Figure 4-5.

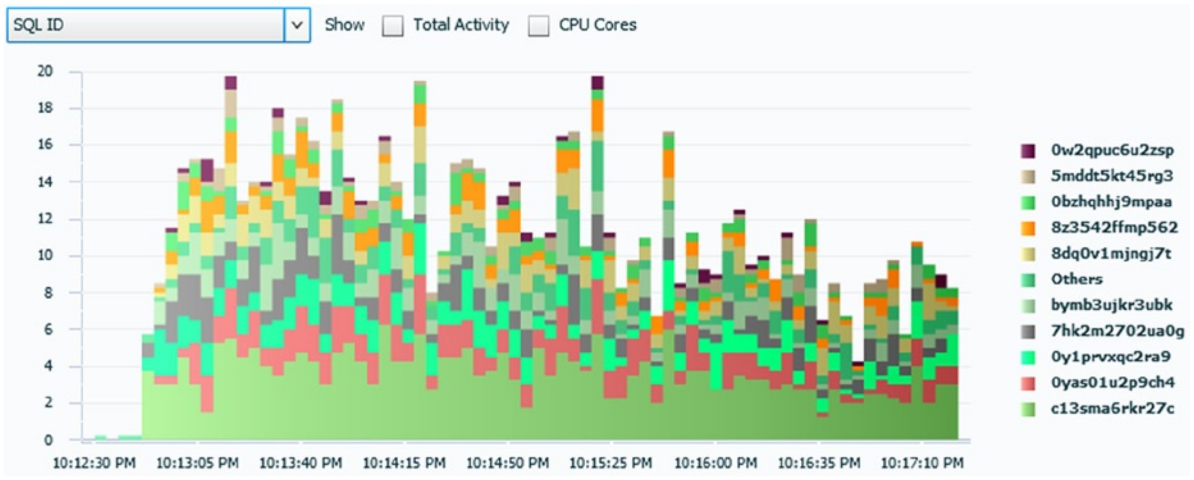
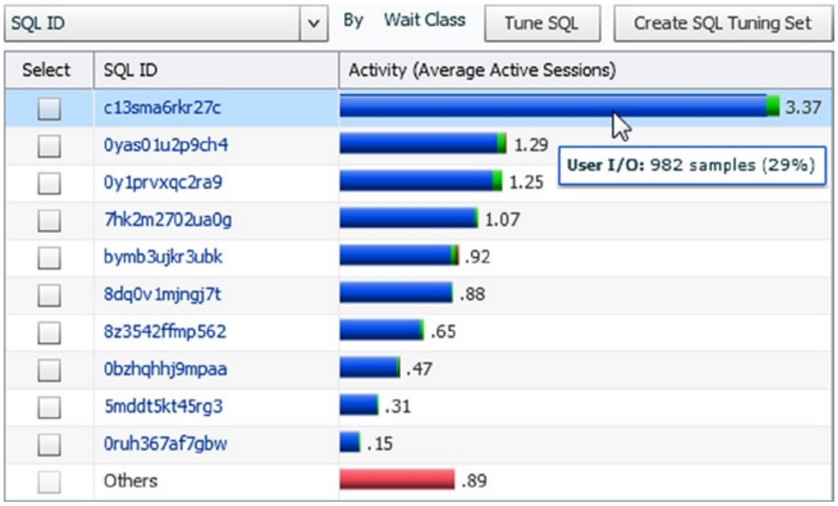


***Figure 4-6.*** *Activity chart showing the top 10 SQL statements for the time period selected in Figure 4-5*

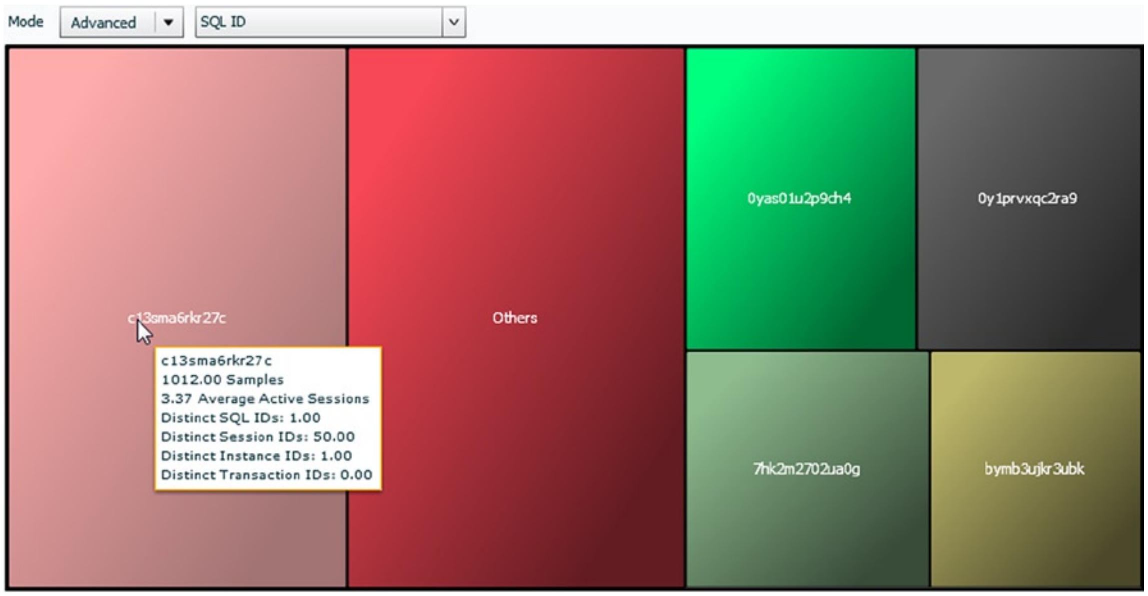
- A *top consumer table* shows, for the selected dimension, the top consumers along with their average number of active sessions over the selected period of time. Note that you can select a dimension different from the one you select for the activity chart. For example, Figure 4-7 shows what the SQL statements are that spent most of the DB time for the 5-minute period selected in Figure 4-5. Notice that by hovering over the activity bar with the pointer, you can also get some information about the kind of activity performed by the SQL statement. For instance, in Figure 4-7, the top SQL statement spent 29% of the time waiting on events associated to the User I/O wait class.



| Select | SQL ID | Activity (Average Active Sessions) |
|---|---|---|
| ☐ | c13sma6rkr27c | 3.37 |
| ☐ | 0yas01u2p9ch4 | 1.29 |
| ☐ | 0y1prvxqc2ra9 | 1.25 |
| ☐ | 7hk2m2702ua0g | 1.07 |
| ☐ | bymb3ujkr3ubk | .92 |
| ☐ | 8dq0v1mjngj7t | .88 |
| ☐ | 8z3542ffmp562 | .65 |
| ☐ | 0bzhqhhj9mpaa | .47 |
| ☐ | 5mddt5kt45rg3 | .31 |
| ☐ | 0ruh367af7gbw | .15 |
| ☐ | Others | .89 |

***Figure 4-7.*** *Top consumer table showing the top 10 SQL statements for the time period selected in Figure 4-5*

- A *load map* shows information similar to the one displayed by the top consumer table, but uses a treemap instead of a table. For example, Figure 4-8 shows the data for the same period as for Figure 4-7. Also, for the load map, you can hover over one of the rectangles to get detailed information about it.



```
c13sma6rkr27c
1012.00 Samples
3.37 Average Active Sessions
Distinct SQL IDs: 1.00
Distinct Session IDs: 50.00
Distinct Instance IDs: 1.00
Distinct Transaction IDs: 0.00
```

***Figure 4-8.*** *Load map showing the top 5 SQL statement for the time period selected in Figure 4-5*

Event though the activity chart and the load map are used to display data, in ASH Analytics, they play another important role. With them you can in fact restrict the analyzed data by selecting one or several top consumers. In other words, with them you can define filters that apply to the charts. You can, for example, do the following:

- Display the top SQL statements and select the most time-consuming (c13sma6rkr27c)
- Display the top wait classes and select the most time-consuming (User I/O)
- Display the top wait events and select the most time-consuming (`db file sequential read`)
- Display the top modules

The result of these operations is shown in Figure 4-9. Notice the filters defined through the load map at the top. Based on that chart, you can infer that the whole load for this particular selection is produced by a single module (New Order).

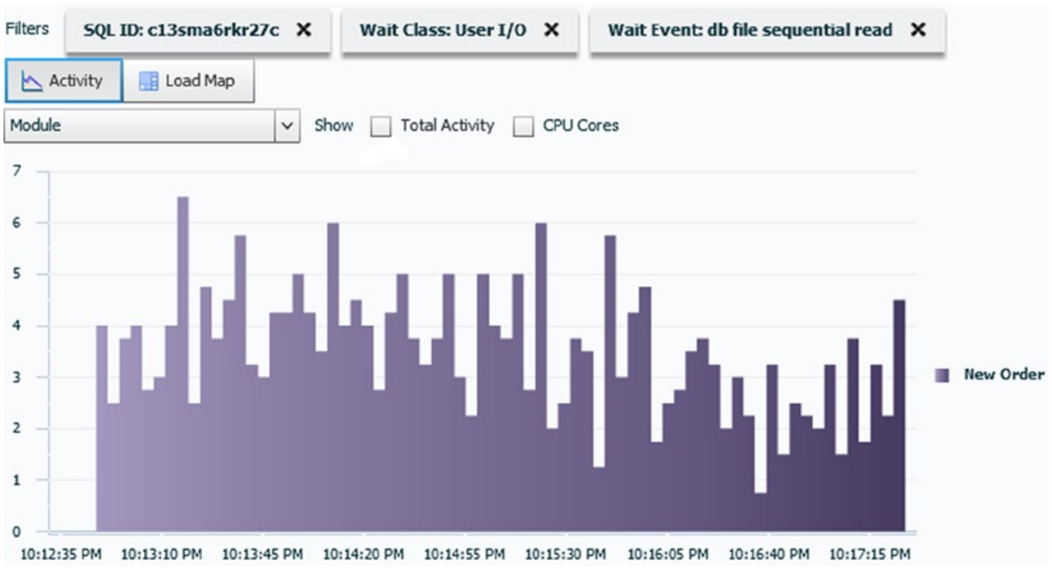


***Figure 4-9.*** *Activity chart showing which module experiences a specific wait event when it executes a specific SQL statement*

Independently of whether you have access to Enterprise Manager, Oracle provides another feature to extract data from active session history without having to manually write SQL statements: *ASH Report*. The purpose of ASH Report is to generate, for a selected period of time, either a text or HTML file that contains information aggregated according to a number of dimensions. In addition, optionally, you can also restrict the analysis to specific sessions, SQL statements, wait classes, services, modules, actions, client ids, or PL/SQL entry points. You can run ASH Report through Enterprise Manager, as well as directly in SQL*Plus, by executing either the `ashrpt.sql` or `ashrpti.sql` script stored under `$ORACLE_HOME/rdbms/admin`. The difference between the two scripts is that the former accepts fewer input parameters. Specifically, when you execute it, you can't restrict the selection to specific components. For example, an execution of the `ashrpt.sql` script asks only for the type of the report (text or HTML), the period of time for which the analysis has to be done, and the report name:

```
SQL> @?/rdbms/admin/ashrpt.sql

Enter 'html' for an HTML report, or 'text' for plain text
Enter value for report_type: text

Enter value for begin_time: 02/12/14 22:12:30

Enter duration in minutes starting from begin time:
Enter value for duration: 5

The default report file name is ashrpt_1_0212_2217.txt.  To use this name,
press <return> to continue, otherwise enter an alternative.
Enter value for report_name:
```

The following is a small excerpt from the report generated by the preceding execution (have a look at the `ashrpt_1_0212_2217.txt` file for the full report):

- General information about the report:

```
          Analysis Begin Time:   12-Feb-14 22:12:30
            Analysis End Time:   12-Feb-14 22:17:30
                 Elapsed Time:         5.0 (mins)
            Begin Data Source:   V$ACTIVE_SESSION_HISTORY
              End Data Source:   V$ACTIVE_SESSION_HISTORY
                 Sample Count:       4,583
      Average Active Sessions:       15.28
  Avg. Active Session per CPU:        3.82
                Report Target:   None specified
```

- The top wait events:

```
                                                Avg Active
Event                       Event Class   % Event   Sessions
----------------------- ------------- ------- ----------
db file sequential read User I/O        65.94      10.07
log file sync           Commit          14.42       2.20
CPU + Wait for CPU      CPU              5.59       0.85
write complete waits    Configuration    1.07       0.16
```

- The activity broken up by engine:

```
                                        Avg Active
Phase of Execution             % Activity   Sessions
------------------------------ ---------- ----------
SQL Execution                       72.31      11.05
PLSQL Execution                      1.46       0.22
```

- The top SQL statements with top wait events (here only the first two are shown, the report contains the top 5):

```
                                                   Sampled #
                 SQL ID             Planhash       of Executions     % Activity
----------------------- ------------------- -------------------- --------------
Event                          % Event Top Row Source                   % RwSrc
------------------------------ ------- --------------------------------- -------
          c13sma6rkr27c           569677903                 1005          21.99
db file sequential read          21.32 TABLE ACCESS - BY INDEX ROWID      15.64
SELECT PRODUCTS.PRODUCT_ID, PRODUCT_NAME, PRODUCT_DESCRIPTION, CATEGORY_ID, WEIG
HT_CLASS, WARRANTY_PERIOD, SUPPLIER_ID, PRODUCT_STATUS, LIST_PRICE, MIN_PRICE, C
ATALOG_URL, QUANTITY_ON_HAND FROM PRODUCTS, INVENTORIES WHERE PRODUCTS.CATEGORY_
ID = :B3 AND INVENTORIES.PRODUCT_ID = PRODUCTS.PRODUCT_ID AND INVENTORIES.WAREHO

          0yas01u2p9ch4                 N/A                  382           8.34
db file sequential read           7.92 ** Row Source Not Available **        7.92
INSERT INTO ORDER_ITEMS(ORDER_ID, LINE_ITEM_ID, PRODUCT_ID, UNIT_PRICE, QUANTITY
) VALUES (:B4 , :B3 , :B2 , :B1 , 1)
```

## SQL Statement Statistics

Information about cursors associated to SQL statements is available at the parent and child level through the v$sqlarea and v$sql views, respectively. In addition, performance statistics at the parent level are also available through v$sqlstats. Even though the v$sqlarea and v$sql views provide more information (columns), there are two good reasons for using the v$sqlstats view. First, it has a greater retention, and therefore even cursors that were already aged out from the library cache may be seen through the v$sqlstats view. Second, accessing the v$sqlstats view requires fewer resources. Because these dynamic performance views have many columns (for example, the v$sql view has 72 columns in version 10.2.0.5 and 91 columns in version 12.1.0.1), they aren't fully covered here (refer to the *Oracle Database Reference* manual for additional information). Here's the most performance-relevant information you can extract from the v$sql view and which column it's found in:

- The identification of the cursor (address, hash_value, sql_id and child_number).
- The type of the SQL statement associated to the cursor (command_type) and the text of the SQL statement (the first 1,000 characters in sql_text and the full text in sql_fulltext).
- The service used to open the session that hard parsed the cursor (service), the schema used for the hard parse (parsing_schema_name and parsing_schema_id), and the session attributes that were in place during the hard parse (module and action).
- If the SQL statement was executed from PL/SQL, the ID of the PL/SQL program and the line number where the SQL statement is located (program_id and program_line#).
- The number of hard parses that took place (loads), how many times the cursor was invalidated (invalidations), when the first and last hard parses took place (first_load_time and last_load_time), the name of the stored outline category (outline_category), SQL profile (sql_profile), SQL patch (sql_patch), SQL plan baseline (sql_plan_baseline) used during the generation of the execution plan, and the hash value of the execution plan associated to the cursor (plan_hash_value).
- The number of parse, execution, and fetch calls (parse_calls, executions, and fetches) that have been carried out and how many rows were processed (rows_processed). For queries, how many times all rows were fetched (end_of_fetch_count).
- The amount of DB time used for the processing (elapsed_time), how much of it has been spent on CPU (cpu_time) or waiting for events belonging to the Application, Concurrency, Cluster and User I/O wait classes (application_wait_time, concurrency_wait_time, cluster_wait_time, and user_io_wait_time), and how much processing has been done by the PL/SQL engine and Java virtual machine (plsql_exec_time and java_exec_time). All values are expressed in microseconds.
- The number of logical reads, physical reads, direct writes, and sorts that have been carried out (buffer_gets, disk_reads, direct_writes, and sorts).

## Real-time Monitoring

Whereas the dynamic performance views described in the last section provide only cumulated statistics about cursors, real-time monitoring provides information that describe how, over time, cursors are executed. There are two important implementation details to be aware of. First, real-time monitoring provides information also during the execution. In other words, you don't have to wait until an execution ends to know how it was carried out. Second, the information related to real-time monitoring is stored independently from the cursor it's based on. Therefore, the information might also be accessible when the cursor itself was already flushed from the library cache. In a way, the goal of real-time monitoring is similar to that of active session history. In fact, whereas active session history provides historical information about the status of the active sessions, real-time monitoring provides historical information about the execution of cursors.

Be aware that to use real-time monitoring, the Tuning Pack option must be licensed. In addition, real-time monitoring is only available from 11.1 onward. If the `control_management_pack_access` initialization isn't set to `diagnostic+tuning`, real-time monitoring is disabled.

Because it makes no sense to monitor all executions, by default the database engine enables monitoring in three specific cases only:

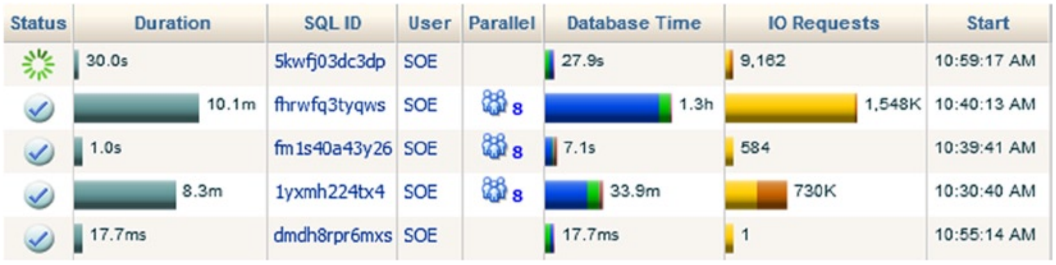
- For executions that consume at least 5 seconds of combined CPU and disk I/O time
- For executions that use parallel processing
- For SQL statements that explicitly enable real-time monitoring by specifying the `monitor` hint (it's also possible to explicitly disable it with the `no_monitor` hint)

---

■ **Caution** In two situations, the database engine can silently disable real-time monitoring for specific executions. First, when an execution plan exceeds 300 lines. Second, when more than 20 concurrent executions per CPU are monitored. To overcome these limitations, you can increase the default value of the `_sqlmon_max_planlines` and `_sqlmon_max_plan` undocumented initialization parameters, respectively. Because higher values can result in higher CPU and memory consumption, you shoudn't unnecessarily set them to very high values without carefully testing the modification.

---

To see which operations were or are currently monitored, you can either directly query the `v$sql_monitor` view or execute the `report_sql_monitor_list` function of the `dbms_sqltune` package. Through them, for each monitored execution, the database engine provides basic information, such as whether the operation is still in execution and the SQL statement related to the monitored operations, as well as key performance figures like DB time utilization. As an example, Figure 4-10 shows part of the information provided by Enterprise Manager.



| Status | Duration | SQL ID | User | Parallel | Database Time | IO Requests | Start |
|---|---|---|---|---|---|---|---|
| ✳ | 30.0s | 5kwfj03dc3dp | SOE | | 27.9s | 9,162 | 10:59:17 AM |
| ✓ | 10.1m | fhrwfq3tyqws | SOE | 8 | 1.3h | 1,548K | 10:40:13 AM |
| ✓ | 1.0s | fm1s40a43y26 | SOE | 8 | 7.1s | 584 | 10:39:41 AM |
| ✓ | 8.3m | 1yxmh224tx4 | SOE | 8 | 33.9m | 730K | 10:30:40 AM |
| ✓ | 17.7ms | dmdh8rpr6mxs | SOE | | 17.7ms | 1 | 10:55:14 AM |

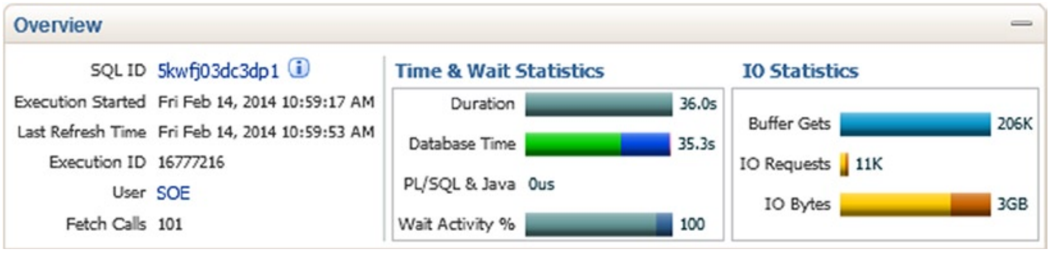
***Figure 4-10.*** *List of the monitored operations*

To get all information gathered by real-time monitoring, you need to generate a report through the `report_sql_monitor` function of the `dbms_sqltune` package. Such an operation can be executed from any tool that can run SQL statements, as well as from several pages in Enterprise Manager (for example, via the SQL Monitoring link in the Performance menu). The `report_sql_monitor` function accepts a number of input parameters and returns a CLOB containing the report. Several of the input parameters can be used to target the monitoring information to display, and others are used to specify the format of the report or the data that has to be displayed. For example, with the `sql_id` parameter, you specify which SQL statement the information is to be displayed for (if `NULL` is specified, the last monitored operation is reported), and with the `type` parameter, you specify the format of the report to be generated (for best results, I advise you to use `active`; `text`, `html`, and `xml` are also available). The following query, an excerpt from the `report_sql_monitor.sql` script, shows an example of how to generate such a report:

```
SELECT dbms_sqltune.report_sql_monitor(sql_id => '5kwfj03dc3dp1',
                                       type   => 'active')
FROM dual
```

In most situations, the report provides all the information you need to understand how a specific execution was carried out. For an active report (note that the formats `text` and `html` provide less information), the following information is provided:

- General information about the execution, as well as a summary of key performance figures (Figure 4-11)



Overview

| | |
|---|---|
| SQL ID | 5kwfj03dc3dp1 |
| Execution Started | Fri Feb 14, 2014 10:59:17 AM |
| Last Refresh Time | Fri Feb 14, 2014 10:59:53 AM |
| Execution ID | 16777216 |
| User | SOE |
| Fetch Calls | 101 |

Time & Wait Statistics

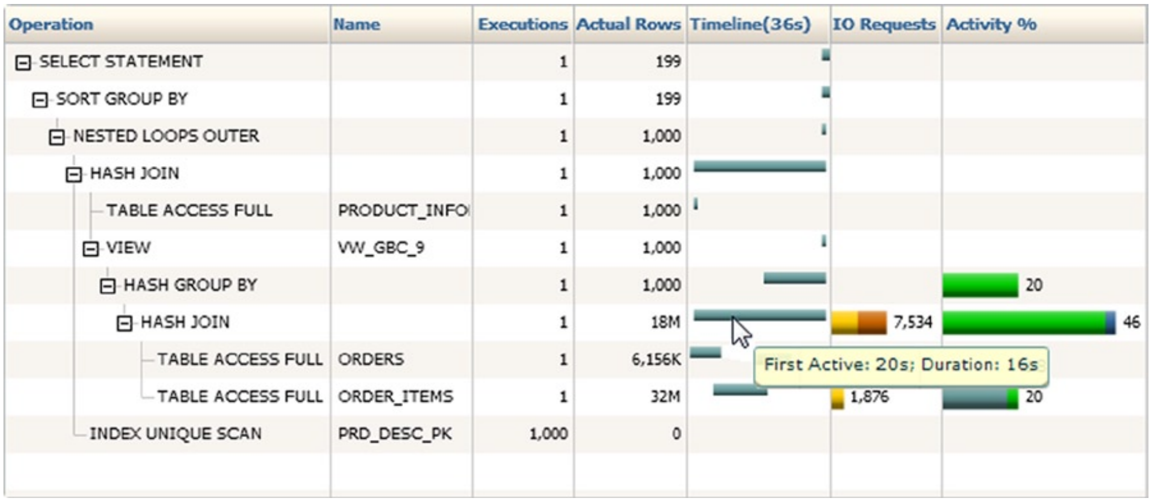
| | |
|---|---|
| Duration | 36.0s |
| Database Time | 35.3s |
| PL/SQL & Java | 0us |
| Wait Activity % | 100 |

IO Statistics

| | |
|---|---|
| Buffer Gets | 206K |
| IO Requests | 11K |
| IO Bytes | 3GB |

*Figure 4-11.* *Overview information about a monitored operation*

- The execution plan, including performance figures at the operation level (Figure 4-12)



| Operation | Name | Executions | Actual Rows | Timeline(36s) | IO Requests | Activity % |
|---|---|---|---|---|---|---|
| SELECT STATEMENT | | 1 | 199 | | | |
| SORT GROUP BY | | 1 | 199 | | | |
| NESTED LOOPS OUTER | | 1 | 1,000 | | | |
| HASH JOIN | | 1 | 1,000 | | | |
| TABLE ACCESS FULL | PRODUCT_INFO | 1 | 1,000 | | | |
| VIEW | VW_GBC_9 | 1 | 1,000 | | | |
| HASH GROUP BY | | 1 | 1,000 | | | 20 |
| HASH JOIN | | 1 | 18M | | 7,534 | 46 |
| TABLE ACCESS FULL | ORDERS | 1 | 6,156K | | | |
| TABLE ACCESS FULL | ORDER_ITEMS | 1 | 32M | | 1,876 | 20 |
| INDEX UNIQUE SCAN | PRD_DESC_PK | 1,000 | 0 | | | |

First Active: 20s; Duration: 16s

*Figure 4-12.* *Execution plan of a monitored operation*

- An activity chart that shows the CPU utilization and the experienced wait events during the execution (Figure 4-13)

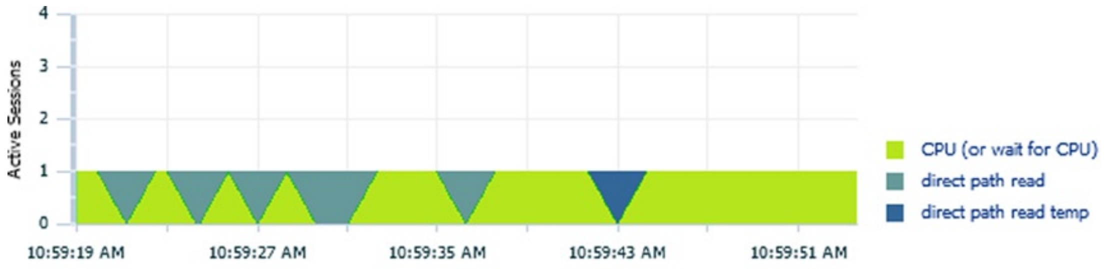


***Figure 4-13.*** *Activity chart of a monitored operation*

- A number of charts that show how specific metrics evolved during the execution (Figure 4-14)

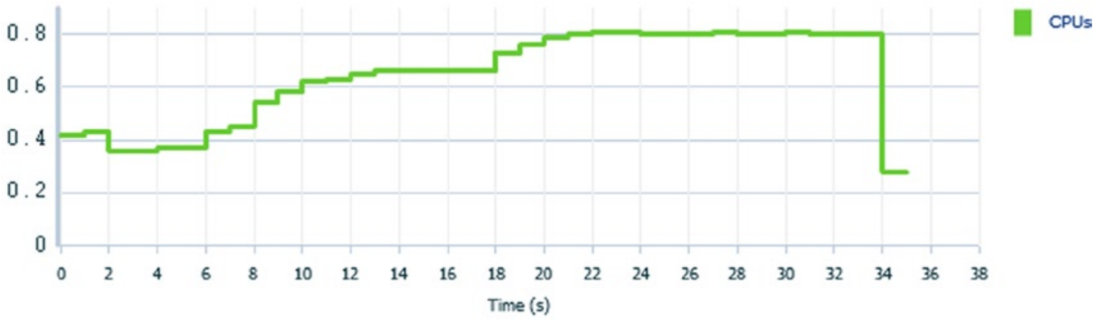


***Figure 4-14.*** *CPU utilization chart of a monitored operation*

To display the text of the SQL statement associated to the monitored operation, you have to click on the icon (i) next to the SQL ID. By doing so, a pop-up showing the text of the SQL statement is opened. In addition, if bind variables are associated to the operation, the name, position, datatype, and value of the bind variables are given.

The Plan Statistics tab is the most interesting part of the report. With it, you can see not only the resource consumption at the execution plan operation level, but also when a specific execution plan operation was performed and how many rows it returned. For example, in Figure 4-12, you can see that the `HASH JOIN` operation (the one highlighted by the cursor) returned 18 million rows, was started 20 seconds after the beginning of the execution, and lasted 16 seconds (you can display this information by hovering the pointer over the bars in the timeline). When you use Enterprise Manager to look at reports, another important feature is that it's possible to follow the execution of active operations in real time.

For every monitored operation, in the Activity tab there's a chart that shows the CPU utilization and the wait events experienced during the execution. For example, Figure 4-13 shows that although at the beginning of the execution the amount of time spent on CPU is more or less equivalent to the time spent performing direct reads, at the end of the execution the operation was fully on CPU. Note that only operations executed in parallel can have a value higher than 1 for the number of active sessions.

The Metrics tab shows a number of charts with metrics for the following performance figures: CPU utilization, number of disk I/O requests, disk I/O throughput, PGA utilization, and temporary tablespace utilization. Figure 4-14 shows the CPU utilization chart. Notice that this chart points out what was already shown in the activity chart: the CPU utilization went up as the execution advanced.

The Plan tab shows the execution plan including all information resulting from the query optimizer estimations. For example, you can use it to see which predicates are applied by which execution plan operation. Finally, for operations executed in parallel, the Parallel tab shows how busy every process involved in the execution was.

## COMPOSITE DATABASE OPERATION

In version 11.1, real-time monitoring works only for SQL statements. For this reason, it's sometimes referred to as real-time SQL monitoring. Since version 11.2, the feature has been extended to support PL/SQL blocks as well. Finally, from version 12.1 onward, with *composite database operation*, it's possible to define that several SQL statements or PL/SQL blocks be considered as a single operation. In other words, you can extend real-time monitoring to user-defined operations that have a meaning from a business point of view. You could, for example, use a composite database operation to define that all SQL statements executed by a batch job are monitored as a single operation.

To define a composite database operation, you have to associate a name to the task you want to monitor. For that purpose, you have the choice between three methods:

- *Generic*: Use the `dbms_sql_monitor` package, specifically the `begin_operation` function and `end_operation` procedure, to delimit the beginning and the end of the operation. This method can be used in any programming language.
- *Java*: Use the `setClientInfo` method of the `java.sql.Connection` interface. This technique is only available from JDBC 4.1 onward.
- *OCI*: Use the `OCIAttrSet` function to set the `OCI_ATTR_DBOP` session attribute.

## Analysis With Diagnostics and Tuning Pack

For an analysis with the Diagnostics Pack, I advise you to use the performance pages provided by Enterprise Manager (whether you use *Database Control*, *Grid Control*, or *Cloud Control* is irrelevant). For this reason, the structure and the examples in this section are based on Enterprise Manager. Note that in this section, all examples as well as all references to Enterprise Manager pages are based on Cloud Control version 12.1.0.3.0. In case you can't use Enterprise Manager, I also describe some scripts that illustrate how to get similar information from the dynamic performance views, which will allow you to perform an equivalent analysis directly with SQL*Plus. Let me stress, though, that if you have access to Enterprise Manager, the analysis is much more straightforward. The only advantage of not using Enterprise Manager is the full flexibility and accessibility of all available data.

### Database Server Load

To assess how much the database server is loaded, you can look at the first chart on the Performance Home page. The chart displays either real-time or historical data. Because this chapter focuses on analyzing performance problems while they're happening (or just after they took place), it goes without saying that you have to use the real-time data. In this mode, data for the last hour should be available. Note that the real-time data displayed by the chart is also available through the `v$metric_history` view.

When looking at the database server load, you should check not only whether the database server is CPU bound (in other words, if all CPU cores are fully used), but also whether there are processes not related to the database instance you're focusing on that consume a lot of CPU time. Figure 4-15 illustrates a case of a database server that was CPU bound for about 5 minutes. However, the database instance background and foreground processes constantly consumed only about two CPU cores. In fact, when the database server was CPU bound, about 6 CPU cores were fully used by processes not related to the database instance. In such a case, it's possible that during those 5 minutes, the database engine experienced a performance problem because of the other processes that were running on the database server.

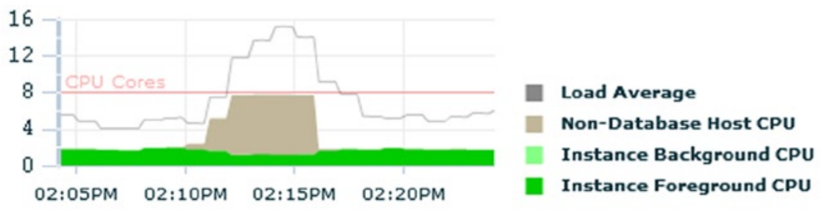


***Figure 4-15.*** *The Runnable Processes chart shows the CPU utilization as well as the load average at the database server level.*

Unfortunately, up to and including version 11.1, the Runnable Processes chart only shows the load average (on a Linux database server that value is equivalent to the load average numbers provided through `/proc/loadavg`). This is because the metrics related to the other values aren't available in previous versions.

---

■ **Caution** When a database server is equipped with CPUs using simultaneous multithreading, the `cpu_count` initialization parameter, as well as the `NUM_CPUS` value in the `v$osstat` view, is set to the total number of threads. As a result, in the Runnable Processes chart the red line represents the number of threads, not the number of CPU cores (as stated). Be aware that using the number of threads to assess whether the database is CPU bound can be misleading. In fact, using all threads at 100% isn't feasible. Also note that virtualization introduces a similar issue.

---

If you don't have access to Enterprise Manager, you can use the `host_load_hist.sql` script to display information that is equivalent to Figure 4-15. Note that the script takes no parameter as input. Because the metrics it displays should be available for one hour, it simply displays all the available data. The following example is an excerpt from its output:

```
SQL> @host_load_hist.sql

BEGIN_TIME DURATION DB_FG_CPU DB_BG_CPU NON_DB_CPU OS_LOAD NUM_CPU
---------- -------- --------- --------- ---------- ------- -------
14:05:00      60.10      1.71      0.03       0.03    4.09       8
14:06:00      60.08      1.62      0.03       0.04    4.13       8
14:07:00      59.10      1.89      0.03       0.04    4.96       8
14:08:00      60.11      1.93      0.03       0.03    5.29       8
14:09:00      60.09      1.73      0.03       0.59    4.60       8
14:10:00      60.10      1.57      0.02       3.64    7.50       8
14:11:00      60.16      1.15      0.02       6.60   11.82       8
14:12:00      60.11      1.21      0.02       6.60   13.77       8
14:13:00      60.28      1.17      0.02       6.62   15.30       8
14:14:00      59.24      1.19      0.02       6.55   14.06       8
14:15:00      60.09      1.59      0.04       0.18    9.19       8
14:16:00      60.09      1.77      0.03       0.03    7.88       8
14:17:00      60.09      1.72      0.03       0.04    5.45       8
14:18:00      60.11      1.87      0.03       0.03    5.28       8
14:19:00      60.09      1.77      0.03       0.03    5.54       8
14:20:00      60.08      1.72      0.03       0.04    4.83       8
```

## System Level Analysis

If you continue the analysis at the system level, you should do it from the Top Activity page. Note that I don't advise to use the Performance Home page, specifically the Average Active Sessions chart, because the drill-down functionalities on that page lead to a fast identification of a performance problem only when most of the response time is consumed by a single wait class.

■ **Tip** If you have access to ASH Analytics, you should use it instead of using the Top Activity page. In some situations, the greater flexibility in selecting the period to analyze, add filters, and aggregate data according a multitude of dimensions allows you to carry out a more straightforward and detailed analysis.

The Top Activity page displays either real-time or historical data. Because this chapter focuses on analyzing performance problems while they're happening (or just after they took place), it goes without saying that you have to use the real-time data. In this mode, data for the last hour should be available. Note that the real-time data displayed by the Top Activity page is taken from the `v$active_session_history` view.

Three sets of data are provided by the Top Activity page:

- An activity chart (Figure 4-16) displays data for the last hour. Specifically, it shows the DB time broken up into CPU utilization as well as wait classes.

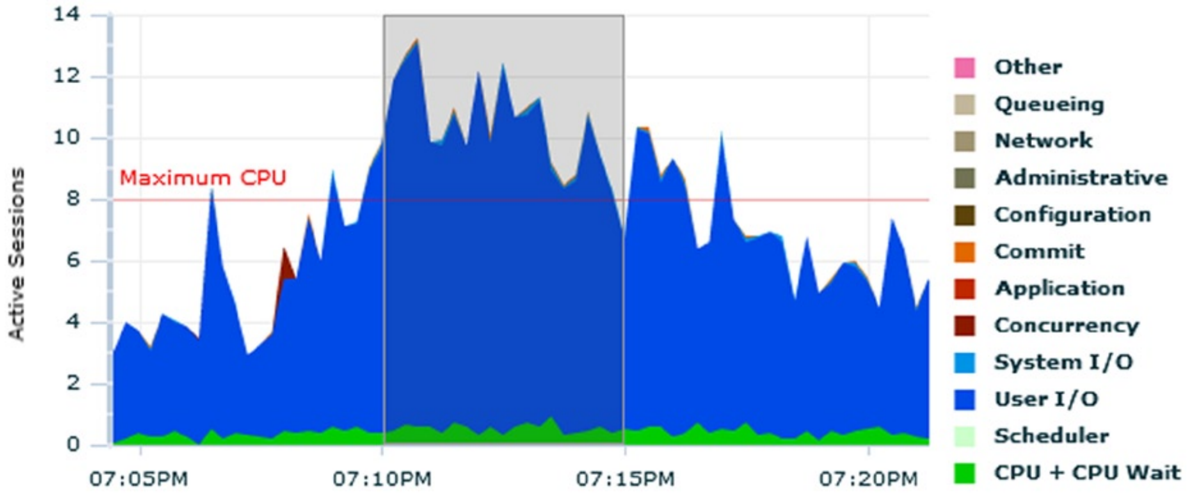
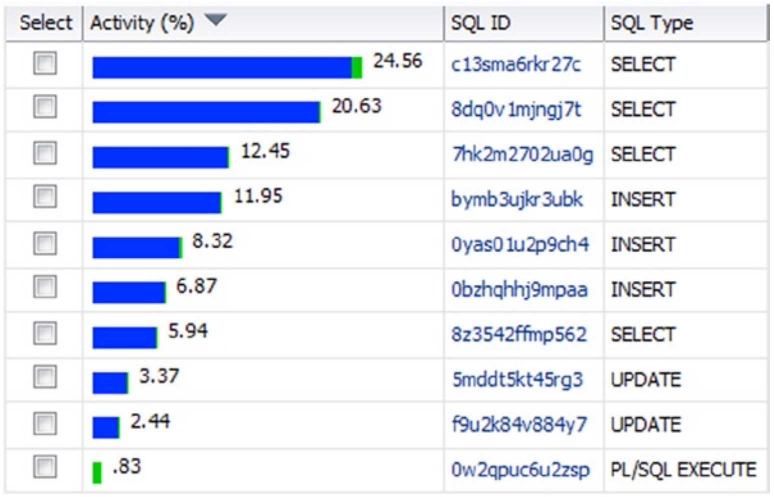


***Figure 4-16.*** *The activity chart shows the CPU utilization as well as wait classes at the system level*

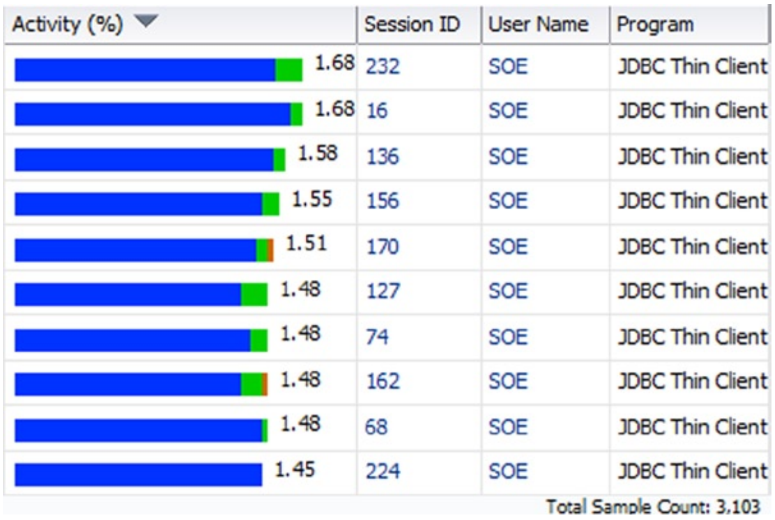
- The Top SQL table (Figure 4-17) shows, for the 5-minute interval selected in the activity chart, the most time consuming SQL statements. For each of them, the overall activity, as well as the SQL ID, is given.



| Select | Activity (%) | SQL ID | SQL Type |
|---|---|---|---|
| ☐ | 24.56 | c13sma6rkr27c | SELECT |
| ☐ | 20.63 | 8dq0v1mjngj7t | SELECT |
| ☐ | 12.45 | 7hk2m2702ua0g | SELECT |
| ☐ | 11.95 | bymb3ujkr3ubk | INSERT |
| ☐ | 8.32 | 0yas01u2p9ch4 | INSERT |
| ☐ | 6.87 | 0bzhqhhj9mpaa | INSERT |
| ☐ | 5.94 | 8z3542ffmp562 | SELECT |
| ☐ | 3.37 | 5mddt5kt45rg3 | UPDATE |
| ☐ | 2.44 | f9u2k84v884y7 | UPDATE |
| ☐ | .83 | 0w2qpuc6u2zsp | PL/SQL EXECUTE |

***Figure 4-17.*** *The Top SQL table shows the most time consuming SQL statements at the system level*

- The Top Sessions table (Figure 4-18) shows, for the 5-minute interval selected in the activity chart, the most time consuming sessions. For each of them, the overall activity, as well as information about the session (the ID, the name of the user, and the program that opened it), is given.



| Activity (%) | Session ID | User Name | Program |
|---|---|---|---|
| 1.68 | 232 | SOE | JDBC Thin Client |
| 1.68 | 16 | SOE | JDBC Thin Client |
| 1.58 | 136 | SOE | JDBC Thin Client |
| 1.55 | 156 | SOE | JDBC Thin Client |
| 1.51 | 170 | SOE | JDBC Thin Client |
| 1.48 | 127 | SOE | JDBC Thin Client |
| 1.48 | 74 | SOE | JDBC Thin Client |
| 1.48 | 162 | SOE | JDBC Thin Client |
| 1.48 | 68 | SOE | JDBC Thin Client |
| 1.45 | 224 | SOE | JDBC Thin Client |

Total Sample Count: 3,103

***Figure 4-18.** The Top Sessions table shows the most time-consuming sessions*

The aim of the activity chart is twofold. First, it allows you to have an idea of what's going on from a database engine point of view. For example, according to Figure 4-16, you can not only infer that the average number of active sessions goes between 3 and 13 (hence, because the number of CPUs is 8, the database engine is moderately loaded), but also that most of the DB time is spent in the User I/O class. Second, you use it to select the 5-minute interval for which you want to display more details about the database load. Hence, if you aren't analyzing something that happened at a specific moment, you usually select the period with the highest number of active sessions. In other words, you select the period with the highest load.

If you don't have access to Enterprise Manager, to display the activity over the last hour you can use the `ash_activity.sql` script. The purpose of the script is to display the average number of active sessions, as well as the percentage of contribution for each wait class. The following output is an example based on the same period as the one shown in Figure 4-16 (note that two parameters set to `all` specify that the script doesn't restrict the output either to a specific sessions or to a specific SQL statement); the only difference compared to Figure 4-16 is that the data is aggregated at minute level:

```
SQL> @ash_activity.sql all all

TIME  AvgActSes   CPU% UsrIO% SysIO%  Conc%  Appl% Commit% Config% Admin%   Net% Queue% Other%
----- --------- ------ ------ ------ ------ ------ ------- ------- ------ ------ ------ ------
19:04       3.8    6.2   93.8    0.0    0.0    0.0     0.0     0.0    0.0    0.0    0.0    0.0
19:05       3.6    8.0   92.0    0.0    0.0    0.0     0.0     0.0    0.0    0.0    0.0    0.0
19:06       5.6    4.8   95.2    0.0    0.0    0.0     0.0     0.0    0.0    0.0    0.0    0.0
19:07       3.4    7.9   92.1    0.0    0.0    0.0     0.0     0.0    0.0    0.0    0.0    0.0
19:08       6.0    5.0   92.5    0.0    2.5    0.0     0.0     0.0    0.0    0.0    0.0    0.0
19:09       7.5    6.9   93.1    0.0    0.0    0.0     0.0     0.0    0.0    0.0    0.0    0.0
19:10      11.2    3.7   96.3    0.0    0.0    0.0     0.0     0.0    0.0    0.0    0.0    0.0
19:11      10.4    4.5   95.5    0.0    0.0    0.0     0.0     0.0    0.0    0.0    0.0    0.0
19:12      10.9    2.9   97.1    0.0    0.0    0.0     0.0     0.0    0.0    0.0    0.0    0.0
19:13       9.8    6.5   93.5    0.0    0.0    0.0     0.0     0.0    0.0    0.0    0.0    0.0
19:14       9.2    3.8   96.2    0.0    0.0    0.0     0.0     0.0    0.0    0.0    0.0    0.0
19:15       8.6    5.2   94.8    0.0    0.0    0.0     0.0     0.0    0.0    0.0    0.0    0.0
19:16       8.0    4.6   95.4    0.0    0.0    0.0     0.0     0.0    0.0    0.0    0.0    0.0
```

```
19:17        7.6    5.1   94.9    0.0     0.0     0.0      0.0      0.0     0.0     0.0     0.0     0.0
19:18        6.1    4.4   95.6    0.0     0.0     0.0      0.0      0.0     0.0     0.0     0.0     0.0
19:19        5.7    5.0   95.0    0.0     0.0     0.0      0.0      0.0     0.0     0.0     0.0     0.0
19:20        6.0    7.2   92.8    0.0     0.0     0.0      0.0      0.0     0.0     0.0     0.0     0.0
19:21        4.8    4.5   95.5    0.0     0.0     0.0      0.0      0.0     0.0     0.0     0.0     0.0
19:22        4.9    5.7   94.3    0.0     0.0     0.0      0.0      0.0     0.0     0.0     0.0     0.0
```

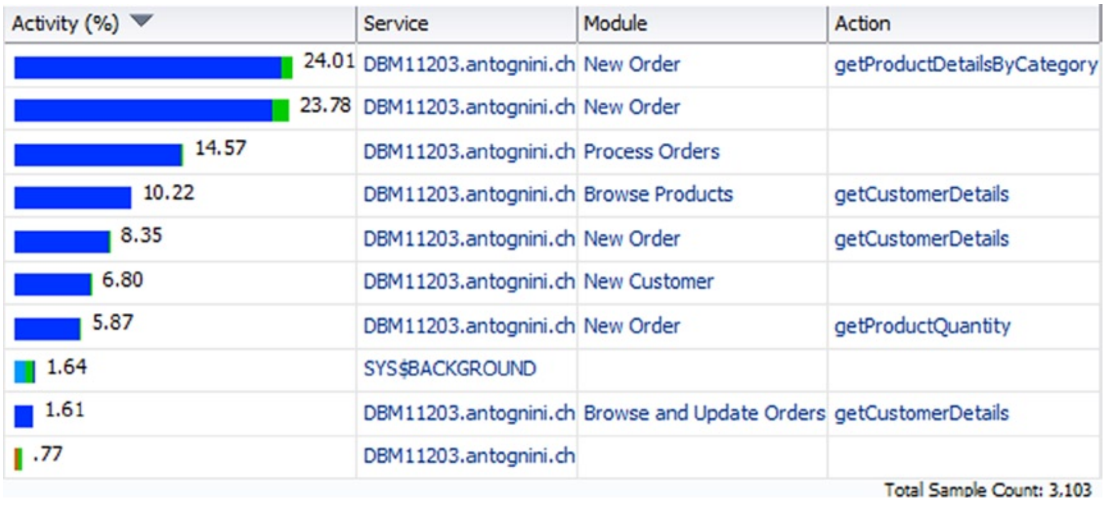
Once you've selected the 5-minute interval you want to focus on, it's time to take a look at the detailed information provided by the Top SQL table (Figure 4-17). If it indicates that few SQL statements are responsible for a large part of the activity (for example, the activity of a single SQL statement is a double-digit percentage), you've identified the SQL statements that you have to further analyze. For example, according to Figure 4-17, seven queries are, in total, responsible for more than 90% of the activity. So, to reduce the load on the system, you have to focus on them.

To display the data shown in Figure 4-17 without Enterprise Manager, you can use the `ash_top_sqls.sql` script. Note that the script requires three parameters as input. The first two specify the period (in this case, beginning and end timestamps) which the data is displayed for. The third specifies whether the script restricts the output to a specific session (this isn't the case when `all` is specified). The following outputs show data that is equivalent to that shown in Figure 4-17:

```
SQL> @ash_top_sqls.sql 2014-02-04_19:10:02.174 2014-02-04_19:15:02.174 all

Activity% DB Time  CPU% UsrIO% Wait% SQL Id        SQL Type
--------- ------- ----- ------ ----- ------------- --------------
     24.6     744   4.2   95.8   0.0 c13sma6rkr27c SELECT
     20.6     625   0.3   99.7   0.0 8dq0v1mjngj7t SELECT
     12.4     377   1.1   98.9   0.0 7hk2m2702ua0g SELECT
     12.0     362   1.9   98.1   0.0 bymb3ujkr3ubk INSERT
      8.3     252   3.6   96.4   0.0 0yas01u2p9ch4 INSERT
      6.9     208   1.4   98.6   0.0 0bzhqhhj9mpaa INSERT
      5.9     180   2.2   97.8   0.0 8z3542ffmp562 SELECT
      3.4     102   5.9   94.1   0.0 5mddt5kt45rg3 UPDATE
      2.4      74   2.7   97.3   0.0 f9u2k84v884y7 UPDATE
      0.8      25 100.0    0.0   0.0 0w2qpuc6u2zsp PL/SQL EXECUTE
```

If no particular SQL statement stands out, it obviously means that the activity is produced by many SQL statements. Hence, it's a sign that major changes in the applications might be necessary to improve performance. When you see such a case, I advise you to take a look at the activity aggregated according to other dimensions. By default, the Top Activity page shows the Top Sessions table (Figure 4-18). But with the dropdown list at the top of that table, you can also aggregate the data according to other dimensions, like Top Services, Top Modules, Top Actions (Figure 4-19), and Top Clients. Sometimes this helps in identifying the part of the application or the client causing high load. Note, however, that some of these dimensions provide useful information only when the application you're analyzing has been correctly instrumented by defining the session attributes described in Chapter 2.



| Activity (%) | Service | Module | Action |
|---|---|---|---|
| 24.01 | DBM11203.antognini.ch | New Order | getProductDetailsByCategory |
| 23.78 | DBM11203.antognini.ch | New Order | |
| 14.57 | DBM11203.antognini.ch | Process Orders | |
| 10.22 | DBM11203.antognini.ch | Browse Products | getCustomerDetails |
| 8.35 | DBM11203.antognini.ch | New Order | getCustomerDetails |
| 6.80 | DBM11203.antognini.ch | New Customer | |
| 5.87 | DBM11203.antognini.ch | New Order | getProductQuantity |
| 1.64 | SYS$BACKGROUND | | |
| 1.61 | DBM11203.antognini.ch | Browse and Update Orders | getCustomerDetails |
| .77 | DBM11203.antognini.ch | | |

Total Sample Count: 3,103

***Figure 4-19.*** *The Top Actions table shows the most time-consuming components aggregated by service, module, and action name*

---

■ **Note** The *Total Sample Count* value displayed by Enterprise Manager tells you how many active session history samples were used to build the chart you're looking at. For example, Figure 4-18 is based on 3,103 samples.

---

You have to look for components that are responsible for a large part of the activity also when considering the activity displayed by a table like Top Sessions or Top Actions. For example, although according to Figure 4-18 no session is responsible for more than 2% of the activity, Figure 4-19 shows that few modules/actions are responsible for most of the load. Hence, independently of whether the Top SQL tables points out something interesting, you might want to review those modules/actions as well.

To display the data aggregated according to a specific dimension without Enterprise Manager, you can use one of the following scripts: `ash_top_sessions.sql`, `ash_top_services.sql`, `ash_top_modules.sql`, `ash_top_actions.sql`, `ash_top_clients.sql`, `ash_top_files.sql`, `ash_top_objects.sql`, and `ash_top_plsql.sql`. The dimension they work on is clearly mentioned in the script name. Note that all scripts require two parameters as input (in the following case, beginning and end timestamps) to specify the period which the data is to be displayed for. The following two examples show the output generated with the `ash_top_sessions.sql` and `ash_top_actions.sql` scripts that are equivalent to Figure 4-18 and Figure 4-19:

```
SQL> @ash_top_sessions.sql 2014-02-04_19:10:02.174 2014-02-04_19:15:02.174

Activity% DB Time CPU% UsrIO% Wait% Session Id User Name Program
--------- ------- ---- ------ ----- ---------- --------- ----------------
      1.7      52  9.6   90.4   0.0        232 SOE       JDBC Thin Client
      1.7      52  3.8   96.2   0.0         16 SOE       JDBC Thin Client
      1.6      49  4.1   95.9   0.0        136 SOE       JDBC Thin Client
      1.5      48  6.3   93.8   0.0        156 SOE       JDBC Thin Client
      1.5      47  4.3   93.6   2.1        170 SOE       JDBC Thin Client
      1.5      46 10.9   89.1   0.0        127 SOE       JDBC Thin Client
      1.5      46  6.5   93.5   0.0         74 SOE       JDBC Thin Client
      1.5      46  8.7   89.1   2.2        162 SOE       JDBC Thin Client
      1.5      46  2.2   97.8   0.0         68 SOE       JDBC Thin Client
      1.5      45  8.9   91.1   0.0         77 SOE       JDBC Thin Client
```

```
SQL> @ash_top_actions.sql 2014-02-04_19:10:02.174 2014-02-04_19:15:02.174

Activity% DB Time CPU% UsrIO% Wait% Module                   Action
--------- ------- ---- ------ ----- ------------------------ ---------------------------
     24.0     745  4.3   95.7   0.0 New Order                getProductDetailsByCategory
     23.8     738  6.0   94.0   0.0 New Order
     14.6     452  1.5   98.5   0.0 Process Orders
     10.2     317  0.0  100.0   0.0 Browse Products          getCustomerDetails
      8.3     259  1.2   98.8   0.0 New Order                getCustomerDetails
      6.8     211  2.8   97.2   0.0 New Customer
      5.9     182  3.3   96.7   0.0 New Order                getProductQuantity
      1.6      51 37.3    3.9  58.8
      1.6      50  0.0  100.0   0.0 Browse and Update Orders getCustomerDetails
      0.8      24 45.8    0.0  54.2
```

## Session Level Analysis

If you carry out an analysis at the session level, the starting point depends on whether the session still exists. If the session exists, you can access information about it by searching the session through the Search Sessions menu item in the Performance menu. Otherwise, you can follow one of the Session ID links on the Top Activity page, specifically from the Top Sessions table.

Three sets of data are provided by the session-level activity page:

- The activity chart (Figure 4-20) shows the DB time broken up into CPU and wait events. It shows up to one hour of data. If the activity is 0 percent, it means that the session is completely idle. If the activity reaches 100 percent, it means that the session is completely busy processing user calls.

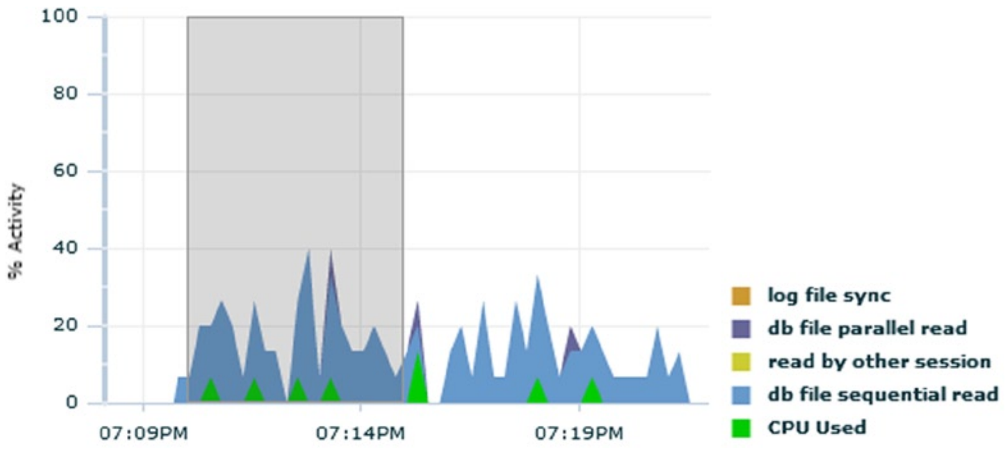
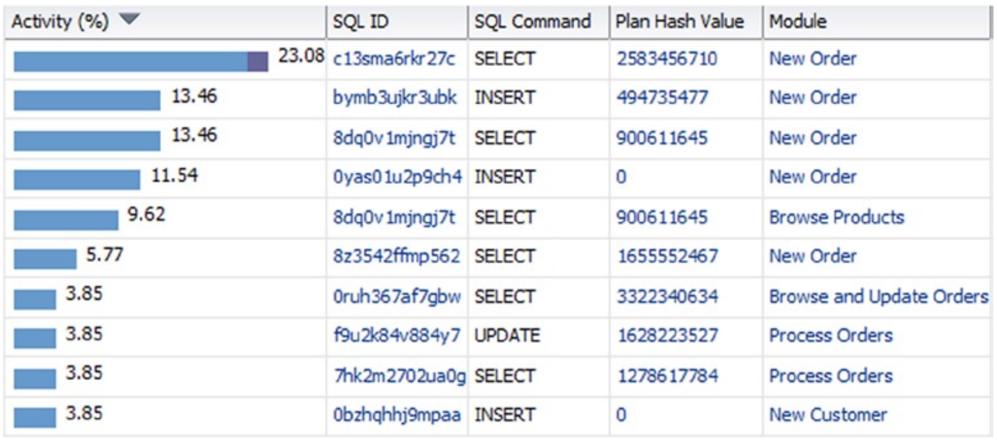


***Figure 4-20.** The session-level activity chart shows the CPU utilization and wait events related to a single session*

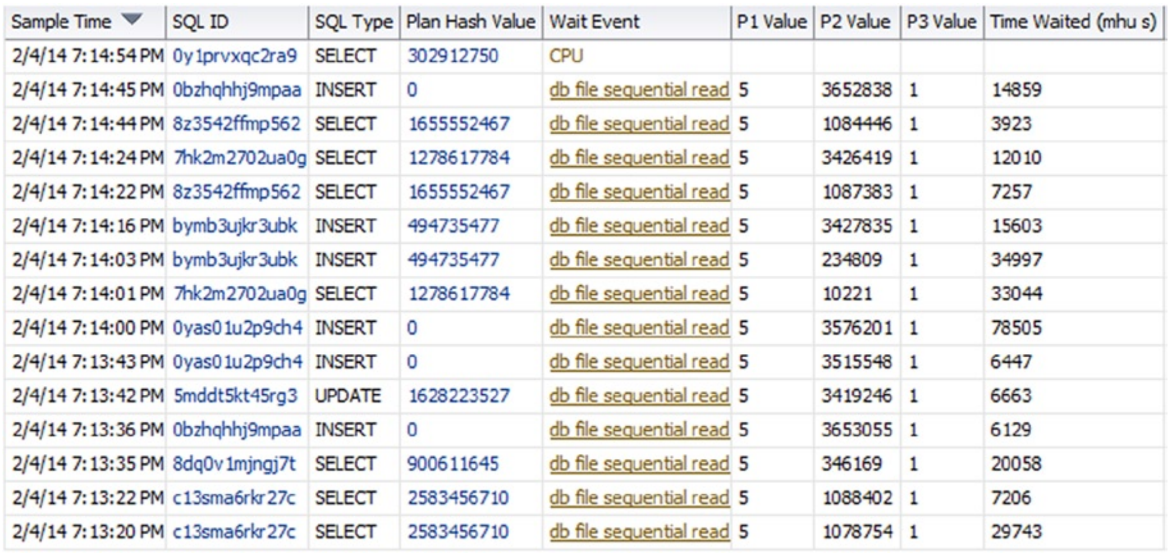
- The active session history aggregated data (Figure 4-21) shows, for the 5-minute interval selected in the activity chart, the most time-consuming SQL statements. For each of them, the overall activity, the SQL ID, the execution plan hash value, as well as some session properties are given.



| Activity (%) | SQL ID | SQL Command | Plan Hash Value | Module |
|---|---|---|---|---|
| 23.08 | c13sma6rkr27c | SELECT | 2583456710 | New Order |
| 13.46 | bymb3ujkr3ubk | INSERT | 494735477 | New Order |
| 13.46 | 8dq0v1mjngj7t | SELECT | 900611645 | New Order |
| 11.54 | 0yas01u2p9ch4 | INSERT | 0 | New Order |
| 9.62 | 8dq0v1mjngj7t | SELECT | 900611645 | Browse Products |
| 5.77 | 8z3542ffmp562 | SELECT | 1655552467 | New Order |
| 3.85 | 0ruh367af7gbw | SELECT | 3322340634 | Browse and Update Orders |
| 3.85 | f9u2k84v884y7 | UPDATE | 1628223527 | Process Orders |
| 3.85 | 7hk2m2702ua0g | SELECT | 1278617784 | Process Orders |
| 3.85 | 0bzhqhhj9mpaa | INSERT | 0 | New Customer |

***Figure 4-21.*** *The active session history aggregated data shows the most time-consuming SQL statements at the session level*

- The active session history raw data (Figure 4-22) shows, for the 5-minute interval selected in the activity chart, detailed information about the samples.



| Sample Time | SQL ID | SQL Type | Plan Hash Value | Wait Event | P1 Value | P2 Value | P3 Value | Time Waited (mhu s) |
|---|---|---|---|---|---|---|---|---|
| 2/4/14 7:14:54 PM | 0y1prvxqc2ra9 | SELECT | 302912750 | CPU | | | | |
| 2/4/14 7:14:45 PM | 0bzhqhhj9mpaa | INSERT | 0 | db file sequential read | 5 | 3652838 | 1 | 14859 |
| 2/4/14 7:14:44 PM | 8z3542ffmp562 | SELECT | 1655552467 | db file sequential read | 5 | 1084446 | 1 | 3923 |
| 2/4/14 7:14:24 PM | 7hk2m2702ua0g | SELECT | 1278617784 | db file sequential read | 5 | 3426419 | 1 | 12010 |
| 2/4/14 7:14:22 PM | 8z3542ffmp562 | SELECT | 1655552467 | db file sequential read | 5 | 1087383 | 1 | 7257 |
| 2/4/14 7:14:16 PM | bymb3ujkr3ubk | INSERT | 494735477 | db file sequential read | 5 | 3427835 | 1 | 15603 |
| 2/4/14 7:14:03 PM | bymb3ujkr3ubk | INSERT | 494735477 | db file sequential read | 5 | 234809 | 1 | 34997 |
| 2/4/14 7:14:01 PM | 7hk2m2702ua0g | SELECT | 1278617784 | db file sequential read | 5 | 10221 | 1 | 33044 |
| 2/4/14 7:14:00 PM | 0yas01u2p9ch4 | INSERT | 0 | db file sequential read | 5 | 3576201 | 1 | 78505 |
| 2/4/14 7:13:43 PM | 0yas01u2p9ch4 | INSERT | 0 | db file sequential read | 5 | 3515548 | 1 | 6447 |
| 2/4/14 7:13:42 PM | 5mddt5kt45rg3 | UPDATE | 1628223527 | db file sequential read | 5 | 3419246 | 1 | 6663 |
| 2/4/14 7:13:36 PM | 0bzhqhhj9mpaa | INSERT | 0 | db file sequential read | 5 | 3653055 | 1 | 6129 |
| 2/4/14 7:13:35 PM | 8dq0v1mjngj7t | SELECT | 900611645 | db file sequential read | 5 | 346169 | 1 | 20058 |
| 2/4/14 7:13:22 PM | c13sma6rkr27c | SELECT | 2583456710 | db file sequential read | 5 | 1088402 | 1 | 7206 |
| 2/4/14 7:13:20 PM | c13sma6rkr27c | SELECT | 2583456710 | db file sequential read | 5 | 1078754 | 1 | 29743 |

***Figure 4-22.*** *The active session history raw data shows detailed information about the samples*

The analysis performed at the session level is similar to the one described in the preceding section at the system level. The only major difference is that at the session level, Enterprise Manager doesn't give you the choice of aggregating the data according to several dimensions. You focus on a single session, and in most situations, only the top SQL statements are relevant.

If you don't have access to Enterprise Manager, you can use the `ash_activity.sql` and `ash_top_sqls.sql` scripts to display data equivalent to Figure 4-20 and Figure 4-21, respectively. Refer to the preceding section for a short description of the scripts. To use them, just specify the ID of the session you're focusing on. To display the data in Figure 4-22, you can simply query the `v$active_session_history` view. The following example illustrates how to use the `ash_activity.sql` script to display data similar to Figure 4-20 (notice that the first parameter specifies the session ID):

```
SQL> @ash_activity.sql 232 all

TIME  AvgActSes   CPU% UsrIO% SysIO%  Conc%  Appl% Commit% Config% Admin%   Net% Queue% Other%
----- --------- ------ ------ ------ ------ ------ ------- ------- ------ ------ ------ ------
19:10       0.2   11.1   88.9    0.0    0.0    0.0     0.0     0.0    0.0    0.0    0.0    0.0
19:11       0.2    8.3   91.7    0.0    0.0    0.0     0.0     0.0    0.0    0.0    0.0    0.0
19:12       0.1    0.0  100.0    0.0    0.0    0.0     0.0     0.0    0.0    0.0    0.0    0.0
19:13       0.2    7.1   92.9    0.0    0.0    0.0     0.0     0.0    0.0    0.0    0.0    0.0
19:14       0.2    0.0  100.0    0.0    0.0    0.0     0.0     0.0    0.0    0.0    0.0    0.0
19:15       0.1   33.3   66.7    0.0    0.0    0.0     0.0     0.0    0.0    0.0    0.0    0.0
19:16       0.1    0.0  100.0    0.0    0.0    0.0     0.0     0.0    0.0    0.0    0.0    0.0
19:17       0.2    0.0  100.0    0.0    0.0    0.0     0.0     0.0    0.0    0.0    0.0    0.0
19:18       0.2    0.0  100.0    0.0    0.0    0.0     0.0     0.0    0.0    0.0    0.0    0.0
19:19       0.2   10.0   90.0    0.0    0.0    0.0     0.0     0.0    0.0    0.0    0.0    0.0
19:20       0.1    0.0  100.0    0.0    0.0    0.0     0.0     0.0    0.0    0.0    0.0    0.0
```

## SQL Statement Information

When you focus on a specific SQL statement, you can display detailed information about it either by clicking on its SQL ID in one of the tables showing the top SQL statements (for example, Figures 4-17 and 4-21) or by searching for it via the Search SQL link in the Performance menu. By doing so, you'll access the SQL Detail page associated to the selected SQL statement. Be aware that when several execution plans exist, you can select one of them with the Plan Hash Value dropdown list positioned between the text of the SQL statement and the tabs. The SQL Detail page provides, in addition to the text of the SQL statement, the following tabs:

- The Statistics tab shows a chart with the average number of active sessions executing the SQL statement (Figure 4-23), the execution statistics (Figure 4-24), and general information associated to the cursor associated to SQL statement. Note that the execution statistics are cumulated values starting from the initial load of the cursor in the library cache. This information can only be displayed if it hasn't been aged out from the library cache.

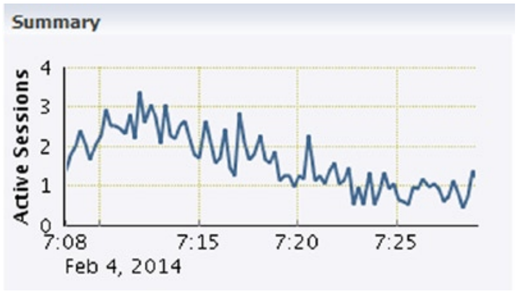
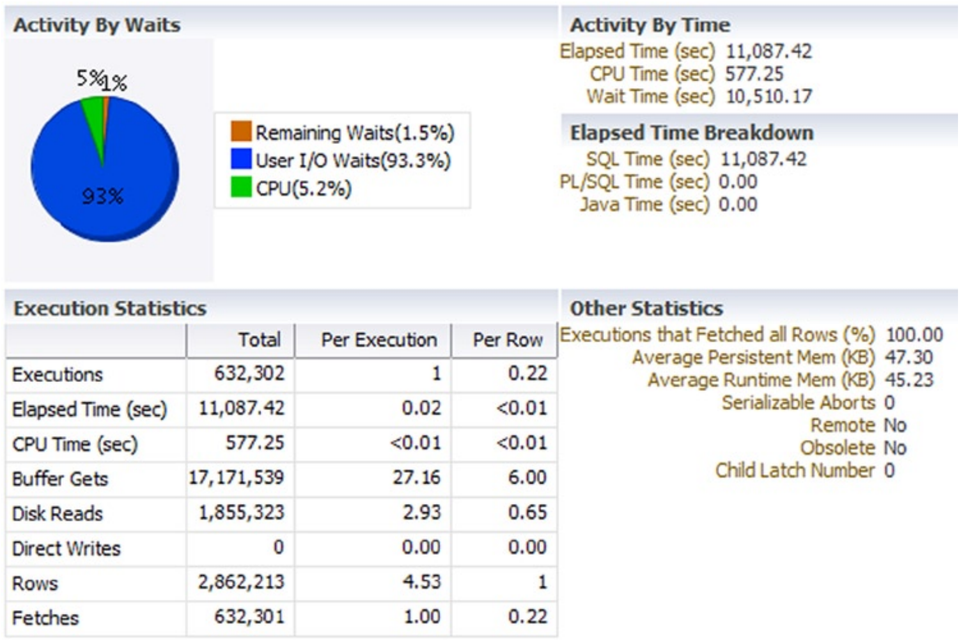


***Figure 4-23.*** *The Summary chart of the Statistics tab shows the average active sessions related to a single SQL statement*



**Activity By Waits**

5% 1%

93%

| | |
|---|---|
| 🟧 | Remaining Waits(1.5%) |
| 🟦 | User I/O Waits(93.3%) |
| 🟩 | CPU(5.2%) |

**Activity By Time**

| | |
|---|---|
| Elapsed Time (sec) | 11,087.42 |
| CPU Time (sec) | 577.25 |
| Wait Time (sec) | 10,510.17 |

**Elapsed Time Breakdown**

| | |
|---|---|
| SQL Time (sec) | 11,087.42 |
| PL/SQL Time (sec) | 0.00 |
| Java Time (sec) | 0.00 |

**Execution Statistics**

| | Total | Per Execution | Per Row |
|---|---|---|---|
| Executions | 632,302 | 1 | 0.22 |
| Elapsed Time (sec) | 11,087.42 | 0.02 | <0.01 |
| CPU Time (sec) | 577.25 | <0.01 | <0.01 |
| Buffer Gets | 17,171,539 | 27.16 | 6.00 |
| Disk Reads | 1,855,323 | 2.93 | 0.65 |
| Direct Writes | 0 | 0.00 | 0.00 |
| Rows | 2,862,213 | 4.53 | 1 |
| Fetches | 632,301 | 1.00 | 0.22 |

**Other Statistics**

| | |
|---|---|
| Executions that Fetched all Rows (%) | 100.00 |
| Average Persistent Mem (KB) | 47.30 |
| Average Runtime Mem (KB) | 45.23 |
| Serializable Aborts | 0 |
| Remote | No |
| Obsolete | No |
| Child Latch Number | 0 |

***Figure 4-24.*** *The execution statistics of the Statistics tab shows information about the runtime behavior of a single SQL statement*

- The Activity tab (Figure 4-25) shows the DB time broken into CPU utilization as well as wait events in an activity chart. It shows up to one hour of data.

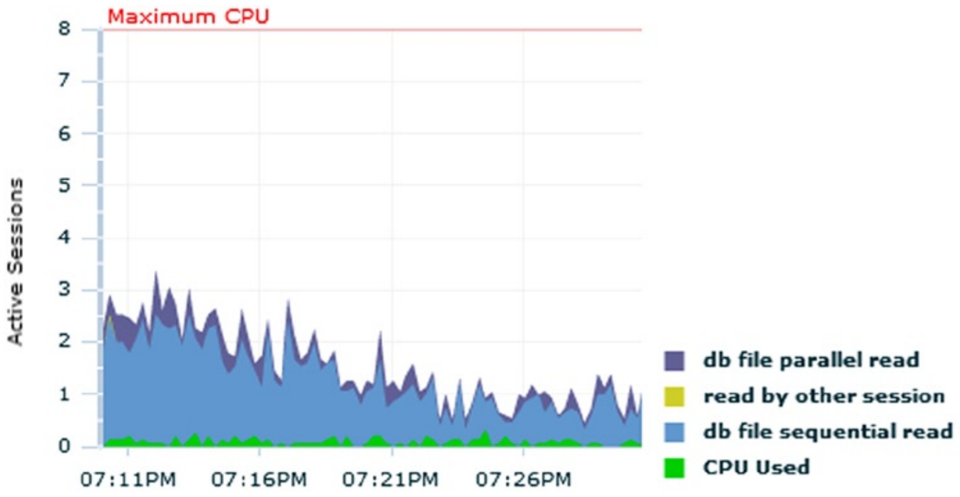


***Figure 4-25.*** *The SQL statement level activity chart shows the CPU utilization and wait events related to a single SQL statement*

- The Plan tab shows the execution plans associated to the SQL statement. This information can only be displayed if it has not been aged out from the shared pool. Note that Chapter 10 provides detailed information on how to read an execution plan and to judge whether it's efficient.

- The Plan Control tab shows whether there are objects, like SQL profiles and SQL plan baselines, associated to the SQL statement. Such objects are discussed in Chapter 11.
- The Tuning History tab shows information generated by the SQL Tuning advisor. The SQL Tuning advisor is also briefly covered in Chapter 11.
- The SQL Monitoring tab, available from version 11.1 onward, shows information related to real-time monitoring. If this information isn't available, the tab can't be selected.

If the version of Oracle Database you're using is 11.2 or newer, and you have the license of the Tuning Pack option, from the SQL Detail page you can also generate a SQL Details Active Report. Because it doesn't provide additional information compared to the SQL Detail page, I only use it when I want to save the information I'm seeing in an HTML file. That way, I can consult it later on or send it to somebody else. If required, you can also generate the same report without using Enterprise Manager. To do that, use the `report_sql_detail` function of the `dbms_sqltune` package. The function accepts several input parameters and returns a CLOB containing the report. Whereas several input parameters can be used to change the data that is displayed in the report, with the `sql_id` parameter you specify which SQL statement the information is displayed for. The following query, an excerpt from the `report_sql_detail.sql` script, shows an example of how to generate such a report:

```
SELECT dbms_sqltune.report_sql_detail(sql_id => 'c13sma6rkr27c')
FROM dual
```

To display the execution statistics (Figure 4-24), as well as the general information about a SQL statement without Enterprise Manager, you can use one of the following scripts: `sqlarea.sql`, `sql.sql`, and `sqlstats.sql`. As their names imply, they extract data provided through `v$sqlarea`, `v$sql`, and `v$sqlstats`, respectively. Refer to the "SQL Statement Information" part of the "Analysis Without Diagnostics Pack" section for additional information.

---

■ **Tip** The `sqlarea.sql`, `sql.sql`, and `sqlstats.sql` scripts provide a feature that isn't available in Enterprise Manager. They can not only show the cumulated statistics since the cursor was loaded as Enterprise Manager does, but also the statistics about the last *n* seconds. This is useful to know what the statistics about the current executions are. In fact, for cursors that stay in the library cache for a long period of time, the picture provided by the cumulated statistics might be misleading.

---

To display the activity data about a single SQL statement without Enterprise Manager, you can use the `ash_activity.sql` script by specifying `all` as first parameter (that means no restriction at the session level) and the ID of the SQL statement as second parameter. The following example shows data similar to Figure 4-25:

```
SQL> @ash_activity.sql all c13sma6rkr27c

TIME  AvgActSes CPU% UsrIO% SysIO% Conc% Appl% Commit% Config% Admin% Net% Queue% Other%
----- --------- ---- ------ ------ ----- ----- ------- ------- ------ ---- ------ ------
19:10       2.6  3.2   96.8    0.0   0.0   0.0     0.0     0.0    0.0  0.0    0.0    0.0
19:11       2.4  5.5   94.5    0.0   0.0   0.0     0.0     0.0    0.0  0.0    0.0    0.0
19:12       2.9  2.3   97.7    0.0   0.0   0.0     0.0     0.0    0.0  0.0    0.0    0.0
19:13       2.4  5.6   94.4    0.0   0.0   0.0     0.0     0.0    0.0  0.0    0.0    0.0
19:14       2.4  4.2   95.8    0.0   0.0   0.0     0.0     0.0    0.0  0.0    0.0    0.0
19:15       2.1  6.3   93.7    0.0   0.0   0.0     0.0     0.0    0.0  0.0    0.0    0.0
19:16       1.8  4.7   95.3    0.0   0.0   0.0     0.0     0.0    0.0  0.0    0.0    0.0
19:17       2.1  1.6   98.4    0.0   0.0   0.0     0.0     0.0    0.0  0.0    0.0    0.0
19:18       1.9  5.4   94.6    0.0   0.0   0.0     0.0     0.0    0.0  0.0    0.0    0.0
```

```
19:19       1.2  6.8   93.2   0.0   0.0   0.0    0.0    0.0    0.0  0.0    0.0    0.0
19:20       1.5  8.9   91.1   0.0   0.0   0.0    0.0    0.0    0.0  0.0    0.0    0.0
19:21       1.2  2.7   97.3   0.0   0.0   0.0    0.0    0.0    0.0  0.0    0.0    0.0
19:22       1.2  8.2   91.8   0.0   0.0   0.0    0.0    0.0    0.0  0.0    0.0    0.0
19:23       0.8 10.2   89.8   0.0   0.0   0.0    0.0    0.0    0.0  0.0    0.0    0.0
19:24       1.0 15.3   84.7   0.0   0.0   0.0    0.0    0.0    0.0  0.0    0.0    0.0
19:25       0.8 11.1   88.9   0.0   0.0   0.0    0.0    0.0    0.0  0.0    0.0    0.0
19:26       1.0  6.6   93.4   0.0   0.0   0.0    0.0    0.0    0.0  0.0    0.0    0.0
19:27       0.9  9.3   90.7   0.0   0.0   0.0    0.0    0.0    0.0  0.0    0.0    0.0
19:28       0.8 10.0   90.0   0.0   0.0   0.0    0.0    0.0    0.0  0.0    0.0    0.0
```

## Analysis Without Diagnostics Pack

The main operations you have to carry out for the analysis of performance problems without the Diagnostics Pack option are basically the same as those described in the preceding section. However, it goes without saying that the goal is to carry out the analysis without taking advantage of dynamic performance views that are licensed through the Diagnostics Pack option. There are two challenges here: first, you can't use Enterprise Manager, and second, most of the dynamic performance views you can use provide only cumulated statistics. Specifically, a replacement for active session history doesn't exist. As a result, you can't look at what's happened over the last few minutes, nor can you directly query a history of the operations carried out by a session. The only dynamic performance views that don't provide cumulated statistics are those providing metrics. But because metrics are focused on ratios and counters, they're of limited use for analyzing performance problems. Therefore, in general, the analysis must be based on dynamic performance views that provide cumulative statistics.

To effectively analyze performance problems based on dynamic performance views that provide cumulative statistics, you need utilities that sample the information they provide. Such utilities might be simple scripts or complex tools like Enterprise Manager. Even though several third-party tools provide features similar to those available in Enterprise Manager, this section focuses on a set of scripts that are freely available, so they can be used on any system. Because most of these scripts work on cumulated statistics, their aim is to find out the rate at which a specific statistic changes over a short period of time. For that purpose, they repeatedly select the same dynamic performance view and compute deltas between each selection.

### Database Server Load

To assess how much the database server is loaded, you can't use the metric's history (to use it, you need the Diagnostics Pack option). Instead, you can look at the current metric through the `v$metric` view. To do that, I use the `host_load.sql` script. It takes as input a single parameter that specifies in minutes for how long the database server load is shown. The following example is an excerpt of the output it generates (notice that the data is equivalent to that displayed in Figure 4-15):

```
SQL> @host_load.sql 16

BEGIN_TIME DURATION DB_FG_CPU DB_BG_CPU NON_DB_CPU OS_LOAD NUM_CPU
---------- -------- --------- --------- ---------- ------- -------
14:05:00      60.10      1.71      0.03       0.03    4.09       8
14:06:00      60.08      1.62      0.03       0.04    4.13       8
14:07:00      59.10      1.89      0.03       0.04    4.96       8
14:08:00      60.11      1.93      0.03       0.03    5.29       8
14:09:00      60.09      1.73      0.03       0.59    4.60       8
14:10:00      60.10      1.57      0.02       3.64    7.50       8
14:11:00      60.16      1.15      0.02       6.60   11.82       8
```

```
14:12:00      60.11      1.21      0.02      6.60   13.77       8
14:13:00      60.28      1.17      0.02      6.62   15.30       8
14:14:00      59.24      1.19      0.02      6.55   14.06       8
14:15:00      60.09      1.59      0.04      0.18    9.19       8
14:16:00      60.09      1.77      0.03      0.03    7.88       8
14:17:00      60.09      1.72      0.03      0.04    5.45       8
14:18:00      60.11      1.87      0.03      0.03    5.28       8
14:19:00      60.09      1.77      0.03      0.03    5.54       8
14:20:00      60.08      1.72      0.03      0.04    4.83       8
```

## System Level Analysis

When you do an analysis at the system level, at first you should look at some system-wide statistics to check how much the database instance is loaded. Such statistics are provided by the `v$system_wait_class` view. Specifically, you should use a script (or tool) that samples the content of the `v$system_wait_class` view to get information similar to what is shown in Figure 4-16. In other words, the average number of active sessions and the portion of time they spent for every wait class. Here I show you an example based on the script `system_activity.sql`. To use it, you have to specify two parameters:

- The first parameter specifies the sampling interval in seconds. Because the database engine doesn't update the statistics in real-time, specifying less than 10–15 seconds is usually pointless.
- The second parameter specifies the number of samples to be taken.

The following example is an excerpt of the output generated by the `system_activity.sql` script when gathering 20 samples of 15 seconds each (notice that the data is equivalent to what is displayed in Figure 4-16):

```
SQL> @system_activity.sql 15 20

Time     AvgActSess Other% Net% Adm% Conf% Comm% Appl% Conc% SysIO% UsrIO% Sched%   CPU%
-------- ---------- ------ ---- ---- ----- ----- ----- ----- ------ ------ ------ ------
19:10:11        9.7    0.0  0.0  0.0   0.0   0.4   0.0   0.0    0.9   94.8    0.0    3.8
19:10:26       10.0    0.0  0.0  0.0   0.0   0.5   0.0   0.0    1.0   94.6    0.0    3.9
19:10:41       10.0    0.0  0.0  0.0   0.0   0.4   0.0   0.0    1.0   94.8    0.0    3.8
19:10:56        9.9    0.0  0.0  0.0   0.0   0.4   0.0   0.0    1.0   94.6    0.0    4.0
19:11:11        9.8    0.0  0.0  0.0   0.2   1.0   0.0   0.0    1.2   93.7    0.0    4.0
19:11:26        9.5    0.0  0.0  0.0   0.0   0.4   0.0   0.0    0.9   94.8    0.0    3.9
19:11:41        9.6    0.0  0.0  0.0   0.0   0.4   0.0   0.0    0.9   94.8    0.0    3.8
19:11:56        9.8    0.0  0.0  0.0   0.0   0.5   0.0   0.0    1.0   94.6    0.0    3.9
19:12:11        9.7    0.0  0.0  0.0   0.0   0.3   0.0   0.0    0.8   94.8    0.0    4.1
19:12:26        9.5    0.0  0.0  0.0   0.0   0.4   0.0   0.0    1.0   94.5    0.0    4.0
19:12:42        9.9    0.0  0.0  0.0   0.0   0.4   0.0   0.0    0.9   95.1    0.0    3.6
19:12:57        9.8    0.0  0.0  0.0   0.9   0.4   0.0   0.1    0.9   93.7    0.0    3.9
19:13:12        9.4    0.0  0.0  0.0   0.0   0.4   0.0   0.0    1.0   94.7    0.0    4.0
19:13:27        9.7    0.0  0.0  0.0   0.0   0.4   0.0   0.0    0.9   94.7    0.0    4.0
19:13:42        9.8    0.0  0.0  0.0   0.0   0.4   0.0   0.0    1.1   94.6    0.0    3.9
19:13:57       10.1    0.0  0.0  0.0   0.0   0.4   0.0   0.0    1.0   94.9    0.0    3.7
19:14:12        9.9    0.0  0.0  0.0   0.0   0.4   0.0   0.0    0.9   94.9    0.0    3.7
19:14:27        9.6    0.0  0.0  0.0   0.0   0.4   0.0   0.0    1.0   94.5    0.0    4.0
19:14:42        9.6    0.0  0.0  0.0   0.7   0.4   0.0   0.0    0.9   94.0    0.0    4.0
19:14:57        9.8    0.0  0.0  0.0   0.0   0.4   0.0   0.0    0.8   94.8    0.0    4.0
```

Another useful system-wide perspective on the load of a system is given by the time model statistics, specifically from the data provided by the v$sys_time_model view. Its content tells you which engine is doing most of the processing and, in case the SQL engine is responsible for it, it also informs you whether operations like parses might be impacting the performance. Also in this case, I advise you to use a script (or tool) that samples the content of the dynamic performance view. For example, the time_model.sql script shows, for a given period of time, the deltas of all time model statistics. To use it, you have to specify two parameters:

- The first parameter specifies the sampling interval in seconds. Because the database engine doesn't update time model statistics in real time, specifying less than 10–15 seconds is usually pointless.
- The second parameter specifies the number of samples to be taken.

The following example shows the output generated by the time_model.sql script when gathering two samples of 15 seconds each (notice that the script displays only statistics that change during the sampling period):

```
SQL> @time_model.sql 15 2

Time     Statistic                                           AvgActSess Activity%
-------- --------------------------------------------------- ---------- ---------
19:14:49 DB time                                                    9.8      98.6
         .DB CPU                                                    0.3       3.4
         .sql execute elapsed time                                  9.7      97.3
         .PL/SQL execution elapsed time                             0.1       1.2
         background elapsed time                                    0.1       1.4
         .background cpu time                                       0.0       0.4

Time     Statistic                                           AvgActSess Activity%
-------- --------------------------------------------------- ---------- ---------
19:15:04 DB time                                                    9.8      98.8
         .DB CPU                                                    0.3       3.5
         .sql execute elapsed time                                  9.7      97.8
         .parse time elapsed                                        0.0       0.3
         ..hard parse elapsed time                                  0.0       0.3
         .PL/SQL execution elapsed time                             0.1       1.2
         background elapsed time                                    0.1       1.2
         .background cpu time                                       0.0       0.3
```

You can also use time model statistics to check whether specific sessions are actually responsible for most of the observed activity. For that purpose, you need the session level statistics provided by the v$sess_time_model view. Also in this case, you should use a script (or tool) that samples the content of the dynamic performance view. Here I show you an example based on the active_sessions.sql script. Its purpose is to show, for a given period of time, how much DB time is spent by the top sessions. To use it, you have to specify three parameters:

- The first parameter specifies the sampling interval in seconds. Because the database engine doesn't update time model statistics in real time, specifying less than 10–15 seconds is usually pointless.
- The second parameter specifies the number of samples to be taken.
- The third parameter specifies how many sessions are listed in the output. It's usually pointless to specify more than 10–20 sessions.

The following example shows the output generated by the `active_sessions.sql` script when gathering the top 10 sessions during a single sample of 15 seconds (notice that the data is similar to Figure 4-18):

```
SQL> @active_sessions.sql 15 1 10

Time     #Sessions #Logins SessionId       Username             Program          Activity%
-------- --------- ------- --------------- -------------------- ---------------- ---------
19:14:49       117       0 195             SOE                  JDBC Thin Client       1.8
                           224             SOE                  JDBC Thin Client       1.5
                           225             SOE                  JDBC Thin Client       1.5
                           232             SOE                  JDBC Thin Client       1.5
                           7               SOE                  JDBC Thin Client       1.5
                           227             SOE                  JDBC Thin Client       1.4
                           74              SOE                  JDBC Thin Client       1.4
                           16              SOE                  JDBC Thin Client       1.4
                           171             SOE                  JDBC Thin Client       1.4
                           68              SOE                  JDBC Thin Client       1.4
                           Top-10 Total                                               14.9
```

Notice that the preceding output also shows, for every interval, the number of open sessions and logins. This information is essential because the script can't detect the work performed by sessions that terminate during the sample interval. So, watch out if you see a decreasing number of sessions or a high number of logins without a proportional increment in the number of sessions.

If, according to the output of the script, few sessions are responsible for a large part of the activity (for example, the activity of a single session is at least a double-digit percentage), you have identified sessions that you might want to further analyze. If, as in the example above, no particular session stands out, it obviously means that the activity is due to many sessions. Hence, it may be necessary to look at performance statistics by aggregating them according to a dimension which is different from the session ID. To perform this task, I advise you to use a script developed by Tanel Põder. Its name is Snapper[1] (`snapper.sql`). Its key functionality is to sample the `v$session` view at a frequency that is inversely proportional to the sampling period. During the sampling, Snapper checks the status of the specified sessions and, for active sessions, it gathers information about their activity (for example, which SQL statement is in execution). Because Snapper is a very flexible and powerful script that accepts many parameters, I don't describe it fully here. I limit myself here to describing the basics and showing you a few examples. For additional information, read the header of the script.

Snapper requires four parameters:

- The first parameter specifies which dynamic performance views to be sampled. When the constant `ash` is specified, the sampling is performed against `v$session`. The idea is to gather data similar to what is provided by active session history. When this parameter is specified, it's also possible to define which columns of the `v$session` view the data is to be aggregated according to. For example, `ash=username+sql_id` means that data is aggregated according to the `username` and `sql_id` columns of the `v$session` view (any column of the view can be specified). When the constant `stats` is specified, the sampling is performed against `v$sesstat`, `v$sess_time_model`, and `v$session_event`.
- The second parameter specifies the sampling period in seconds.
- The third parameter specifies the number of samples to be taken.

[1]The script is available at http://blog.tanelpoder.com/files/scripts/snapper.sql.

- The fourth parameter specifies which sessions are sampled. It's possible to specify a single session ID, a list of several session ids (separated by a comma), the constant `all` to sample all sessions, a query that returns the ID of the sessions to be sampled, or one of a number of supported expressions (for example, `user=chris` to query data for all sessions opened by the specified user—see the script's header for a full list of available expressions).

The first example based on Snapper shows how to gather information similar to that shown in Figure 4-17. The four parameters are set as follows:

- The first parameter (`ash=sql_id`) specifies to query the `v$session` view and aggregate the resulting data by SQL ID.
- The second parameter (15) specifies to use a sampling period of 15 seconds.
- The third parameter (1) specifies to take a single sample.
- The fourth parameter (`all`) specifies to sample all sessions.

```
SQL> @snapper.sql ash=sql_id 15 1 all

-------------------------
Active% | SQL_ID
-------------------------
   196% | c13sma6rkr27c
   186% | 8dq0v1mjngj7t
   122% | bymb3ujkr3ubk
   107% | 7hk2m2702ua0g
    82% | 0yas01u2p9ch4
    63% | 8z3542ffmp562
    62% | 0bzhqhhj9mpaa
    30% | 5mddt5kt45rg3
    26% |
    26% | f9u2k84v884y7
```

Note that as shown in the previous example, the Active% column might be greater than 100%. This can happen when the sampling is performed on several sessions. For example, in the previous output, during the sampling period, the top SQL statement (c13sma6rkr27c) was executed, on average, by 1.96 sessions.

The second example shows how to gather information similar to that shown in Figure 4-19. For that purpose, compared to the preceding example, only the first parameter has to be changed. It has to specify to aggregate data according to the session attributes module and action (`ash=module+action`):

```
SQL> @snapper.sql ash=module+action 15 1 all

---------------------------------------------------------------
Active% | MODULE                       | ACTION
---------------------------------------------------------------
    97% | New Order                    | getProductDetailsByCatego
    94% | New Order                    |
    86% | Process Orders               |
    58% | Browse Products              | getCustomerDetails
    32% | New Order                    | getCustomerDetails
    28% | New Customer                 |
    22% | New Order                    | getProductQuantity
     9% |                              |
     8% | Browse and Update Orders     | getCustomerDetails
     3% | Browse Products              | getProductDetails
```

## Session Level Analysis

The previous examples based on Snapper show you how to analyze the activity of the whole system (in other words, the activity carried out by all sessions). However, with Snapper it's also possible to target a specific session. For that, the fourth parameter, instead of being set to `all`, has to specify a session ID. The following two examples show how to get information similar to what is shown in Figures 4-20 and 4-21, respectively:

```
SQL> @snapper.sql ash=event 15 1 172

---------------------------------------------
Active% | EVENT
---------------------------------------------
    22% | db file sequential read
     1% | ON CPU
     1% | db file parallel read

SQL> @snapper.sql ash=sql_id+module+action 15 1 172

---------------------------------------------------------------------------------
Active% | SQL_ID          | MODULE                       | ACTION
---------------------------------------------------------------------------------
     7% | c13sma6rkr27c   | New Order                    | getProductDetailsByCatego
     3% | 8dq0v1mjngj7t   | New Order                    | getCustomerDetails
     3% | 0yas01u2p9ch4   | New Order                    |
     1% | 7hk2m2702ua0g   | Process Orders               |
     1% | 8dq0v1mjngj7t   | Browse Products              | getCustomerDetails
     1% | 8dq0v1mjngj7t   | Browse and Update Orders     | getCustomerDetails
     1% | bymb3ujkr3ubk   | New Order                    |
     1% | 8z3542ffmp562   | New Order                    | getProductQuantity
     1% | 0bzhqhhj9mpaa   | New Customer                 |
```

## SQL Statement Information

When you've identified a SQL statement that's responsible for a large part of the activity, to display information about it you can use one of the following scripts: `sqlarea.sql`, `sql.sql`, and `sqlstats.sql`. As their names imply, they extract data provided through `v$sqlarea`, `v$sql`, and `v$sqlstats`, respectively. The three scripts require two parameters as input:

- The first parameter specifies the ID of the SQL statement you're interested in.
- The second parameter specifies whether the script displays either the cumulated statistics since the cursor was loaded in the library cache, or how much the statistics are currently increasing. When the parameter is set to a numerical value greater than 0, the latter mode is enabled. For that purpose, the statistics are queried twice with the wait in-between (number of seconds) specified as a parameter. For any other value, the former are shown.

The following example shows how to use the `sqlstats.sql` script to show, for the SQL statement identified by the ID `c13sma6rkr27c`, the statistics of the last 15 seconds:

```
SQL> @sqlstats.sql c13sma6rkr27c 15

-------------------------------------------------------------------------------------------
Identification
-------------------------------------------------------------------------------------------
SQL Id                                                                        c13sma6rkr27c
Execution Plan Hash Value                                                        1640444070
-------------------------------------------------------------------------------------------
Shared Cursors Statistics
-------------------------------------------------------------------------------------------
Total Parses                                                                              0
Loads / Hard Parses                                                                       0
Invalidations                                                                             0
Cursor Size / Shared (bytes)                                                              0
-------------------------------------------------------------------------------------------
Activity by Time
-------------------------------------------------------------------------------------------
Elapsed Time (seconds)                                                               33.559
CPU Time (seconds)                                                                    0.568
Wait Time (seconds)                                                                  32.991
-------------------------------------------------------------------------------------------
Activity by Waits
-------------------------------------------------------------------------------------------
Application Waits (%)                                                                 0.000
Concurrency Waits (%)                                                                 0.000
Cluster Waits (%)                                                                     0.000
User I/O Waits (%)                                                                   97.994
Remaining Waits (%)                                                                   0.313
CPU (%)                                                                               1.692
-------------------------------------------------------------------------------------------
Elapsed Time Breakdown
-------------------------------------------------------------------------------------------
SQL Time (seconds)                                                                   33.559
PL/SQL Time (seconds)                                                                 0.000
Java Time (seconds)                                                                   0.000
-------------------------------------------------------------------------------------------
Execution Statistics                                Total     Per Execution         Per Row
-------------------------------------------------------------------------------------------
Elapsed Time (milliseconds)                        33,559                23           5.133
CPU Time (milliseconds)                               568                 0           0.087
Executions                                          1,436                 1           0.220
Buffer Gets                                        43,305                30           6.624
Disk Reads                                          4,292                 3           0.656
Direct Writes                                           0                 0           0.000
Rows                                                6,538                 5           1.000
Fetches                                             1,440                 1           0.220
Average Fetch Size                                      5
```

```
-------------------------------------------------------------------------------------------
Other Statistics
-------------------------------------------------------------------------------------------
Executions that Fetched All Rows (%)                                                    100
Serializable Aborts                                                                       0
-------------------------------------------------------------------------------------------
```

## On to Chapter 5

This chapter provides an analysis road map for identifying performance problems as they occur. It also describes several tools and techniques that can be used with it. Even though the presented analysis road map is helpful, it's only one approach among many. In any case, the most important point is that only a methodical approach to performance problems leads to a quick and successful identification of the problems. I can't stress this enough.

This chapter describes how to analyze performance problems as they happen. But what can you do if a problem that occurred in the past is no longer experienced? Can you find out what happened and, as a result, prevent it from happenning again? Chapter 5 describes how to give answers to these questions by using a repository containing historical performance statistics.

CHAPTER 5

# Postmortem Analysis of Irreproducible Problems

This chapter describes how to analyze a performance problem that you can neither reproduce nor observe while it's happening. In other words, this chapter explains what you can do when you are trying to analyze a problem that happened in the past and, as a result, you can't take advantage of the information provided by SQL Trace and dynamic performance views. In such a situation, the only way you can make a quantitative analysis is by using a repository containing performance statistics covering the period of time you want to analyze.

## Repositories

Oracle Database provides two repositories that store information that you can use to analyze a performance problem that happened in the past:

- Automatic Workload Repository (AWR)
- Statspack

Since AWR is an evolution of Statspack, it is based on the same three basic concepts(as are the utilities provided with them):

- At regular intervals (for example, 30 minutes), the content of a bunch of dynamic performance views (for example, the views discussed in Chapter 4) is dumped into a set of tables. The resulting data is called a *snapshot* and can be identified by an integer value called a *snapshot ID*. For some dynamic performance views, all the data they provide is dumped. For others, only part of the data is dumped. For example, information about SQL statements is dumped only for the top consumers.
- Through a script provided by Oracle or, for AWR, with a tool (for example, Enterprise Manager or SQL Developer), you can find out how much the statistics stored in the repository changed over a period of time delimited by two snapshots.
- In general, snapshots aren't preserved indefinitely. Instead, they are purged after a specific retention period has elapsed. Snapshots of specific time periods can be marked as *baselines* and, as a result, be excluded from the purging process. Baselines are useful for comparative purposes. For example, if you preserve a baseline for a period of time in which the system performs as expected, you can compare that period with a baseline when a performance problem occurs.

Be aware that the length of the interval at which snapshots are taken is of paramount importance. In fact, intervals of one hour or longer are generally less useful than shorter ones. There are two main reasons. First, computing a rate or an average over a too-long period of time can be very misleading. Second, since the information provided by some dynamic performance views is highly volatile, at the time a snapshot is taken, useful information might no longer be available. For example, a SQL statement that consumes a lot of resources might be missing from a snapshot because it gets removed from the library cache before the snapshot is taken. Therefore, I usually advise using an interval of 20 or 30 minutes.

The key differences between AWR and Statspack are summarized in Table 5-1. Since AWR is more powerful than Statspack, you should use AWR except when licensing requirements prevent you from doing so.

***Table 5-1.*** *Key Differences Between AWR and Statspack*

| Automatic Workload Repository | Statspack |
|---|---|
| Tightly integrated with the database engine and, therefore, automatically installed and managed. | Manually installed and managed by the DBA. |
| Stores system-level and SQL statement–level information, as well as session-level information based on Active Session History. | Stores system-level and SQL statement–level information only. |
| Content can be leveraged through Enterprise Manager. | No Enterprise Manager integration. |
| Content is used by advisories to automatically diagnose performance issues. | Content isn't used by advisories. |
| Requires Oracle Diagnostics Pack option and, therefore, Enterprise Edition. | Freely available in all editions. |
| Can't be used on a standby database open in read-only mode. | From version 11.1 onward, can be used on a standby database open in read-only mode. |

## Automatic Workload Repository

This section describes how to configure AWR, take snapshots, and manage baselines. How to take advantage of the information stored in AWR is described later, in the "Analysis With Diagnostics Pack" section. For the moment, it's important to know only that information stored in AWR is exposed through data dictionary views that have the `dba_hist` prefix (in a 12.1 multitenant environment, views with the `cdb_hist` prefix also exist).

### Performing Configuration

AWR is automatically installed and configured on every database. As a result, from the very beginning of its existence, the database engine takes snapshots describing the workload that the database is subject to.

■ **Caution** When the `statistics_level` initialization parameter is set to `basic`, the database engine doesn't automatically take snapshots.

The configuration is based on three parameters:

- *Snapshot interval*: The interval (in minutes) between two snapshots. The minimum and maximum values are 10 minutes and 100 years, respectively. The default value is 1 hour.

- *Retention period*: How long (in minutes) the snapshots are retained. The minimum and maximum values are 1 day and 100 years, respectively. If 0 is specified, snapshots are retained permanently. The default value is 7 days in version 10.2, and 8 days from version 11.1 onward.
- *Top SQL statements*: The number of SQL statements that are considered top consumers for every snapshot. Since several categories of consumers are considered (for example, top elapsed time, top CPU utilization, and top parse calls), the actual number of SQL statements that are stored for every snapshot can be higher than the value specified with this parameter. The parameter accepts values between 30 and 50,000 as well as DEFAULT and MAXIMUM. Note that DEFAULT means either 30 or 100, depending on the flush level used for creating the snapshot (refer to the "Taking Snapshots" section that follows).

## COLORED SQL ID

In case you want to make sure that information about a specific SQL statement is captured in every snapshot (that is, independent of whether it's a top consumer), from version 11.1 onward you can mark that statement's SQL ID as *colored*. To mark and unmark a SQL ID as colored, the dbms_workload_repository package provides the add_colored_sql and remove_colored_sql procedures, respectively. Note that both procedures accept a parameter specifying the SQL ID on which the operation has to be carried out.

To know which SQL statements were marked as colored and when that happened, you can query dba_hist_colored_sql and, in a 12.1 multitenant environment, cdb_hist_colored_sql views.

The following query shows how to display the current value of the parameters (notice that these are the default values used from version 11.1 onward):

```
SQL> SELECT snap_interval, retention, topnsql
  2  FROM dba_hist_wr_control;

SNAP_INTERVAL     RETENTION         TOPNSQL
----------------- ----------------- -------
+00000 01:00:00.0 +00008 00:00:00.0 DEFAULT
```

You can change the default configuration by calling the modify_snapshot_settings procedure of the dbms_workload_repository package. For example, the following call sets the interval to 20 minutes and the retention to 35 days:

```
dbms_workload_repository.modify_snapshot_settings(
  interval  => 20,
  retention => 35*60*24,
  topnsql   => 'DEFAULT'
);
```

The data belonging to AWR is stored in the sysaux tablespace. The amount of storage required for it strongly depends on how the three parameters are set. In general, though, every snapshot takes at least 1 megabyte. If you want to know how much space is currently used, you can run the following query:

```
SELECT space_usage_kbytes
FROM v$sysaux_occupants
WHERE occupant_name = 'SM/AWR'
```

## Taking Snapshots

In addition to the snapshots that are automatically created by the database engine, you can manually take snapshots. This is useful when you want to store information about a specific period of time. To take a snapshot, you have to call one of the `create_snapshot` subroutines of the `dbms_workload_repository` package. Two subroutines exist: a function and a procedure. Both of them accept a parameter that is used to specify the flush level (either `TYPICAL` or `ALL`, the former being the default value). If `TYPICAL` is used, the top 30 SQL statements for each category are stored. If `ALL` is used, the top 100 are stored. The only difference between the function and the procedure is that the function returns the snapshot ID. The following query shows how to create a snapshot with the flush level `ALL` and display the snapshot ID associated with it:

```
SQL> SELECT dbms_workload_repository.create_snapshot(flush_level => 'ALL') AS snap_id
  2  FROM dual;

   SNAP_ID
----------
       738
```

The snapshots stored in AWR are visible in the `dba_hist_snapshot` and, in a 12.1 multitenant environment, `cdb_hist_snapshot` views:

```
SQL> SELECT begin_interval_time, end_interval_time,
  2         decode(snap_level, 1, 'TYPICAL', 2, 'ALL', snap_level) AS snap_level
  3  FROM dba_hist_snapshot
  4  WHERE snap_id = 738;

BEGIN_INTERVAL_TIME       END_INTERVAL_TIME         SNAP_LEVEL
------------------------- ------------------------- ----------
22-APR-14 04.00.22.234 PM 22-APR-14 04.06.58.230 PM ALL
```

## Managing Baselines

A baseline is composed of several consecutive snapshots that are marked as a baseline. Two types of baselines exist:

- *Fixed baseline*: A set of consecutive snapshots delimited by a static start snapshot ID and a static end snapshot ID. You can create as many fixed baselines as necessary.
- *Moving window baseline*: A set of consecutive snapshots covering a specific amount of time (specified in days) and ending with the most current snapshot. Every database has one moving window baseline that is used by the database engine for adaptive thresholds (refer to the *Performance Tuning Guide* manual for additional information). This type of baseline is available from version 11.1 onward.

### Managing Fixed Baselines

To create baselines, the `dbms_workload_repository` package provides several functions and procedures named `create_baseline`. They all implement the same basic functionality, although they differ in two aspects. First, the start and the end of the baseline can be specified through either two IDs or two dates (the latter is available only from version 11.1 onward, though). Second, only the functions return the ID associated with the new baseline. Note that all subroutines require a parameter that specifies the name of the baseline and, optionally, a parameter that specifies after how many days the baseline is automatically dropped (the default value, NULL, indicates that there's no expiration). For example, the following call shows how to create a baseline named TEST that expires after 30 days:

```
dbms_workload_repository.create_baseline(
  start_snap_id => 738,
  end_snap_id   => 739,
  baseline_name => 'TEST',
  expiration    => 30
);
```

The baselines stored in AWR are visible in the dba_hist_baseline and, in a 12.1 multitenant environment, cdb_hist_baseline views:

```
SQL> SELECT start_snap_id, start_snap_time, end_snap_id, end_snap_time
  2  FROM dba_hist_baseline
  3  WHERE baseline_name = 'TEST'
  4  AND baseline_type = 'STATIC';

START_SNAP_ID START_SNAP_TIME           END_SNAP_ID END_SNAP_TIME             EXPIRATION
------------- ------------------------- ----------- ------------------------- ----------
          738 22-APR-14 04.06.58.230 PM         739 22-APR-14 04.12.50.933 PM         30
```

To display the metrics associated with a baseline (this also works for the moving window baseline), the dbms_workload_repository package provides the select_baseline_metric function. The following query illustrates how to display the metrics associated with the TEST baseline:

```
SQL> SELECT metric_name, metric_unit, minimum, average, maximum
  2  FROM table(dbms_workload_repository.select_baseline_metric('TEST'))
  3  ORDER BY metric_name;

METRIC_NAME              METRIC_UNIT            MINIMUM    AVERAGE    MAXIMUM
------------------------ ---------------------- ------- ---------- ----------
Active Parallel Sessions Sessions                     0          0          0
Active Serial Sessions   Sessions                     0 1.42857143          7
Average Active Sessions  Active Sessions              0 .413742101 3.49426268
...
...
User Transaction Per Sec Transactions Per Second      0 6.98898086 49.2425504
VM in bytes Per Sec      bytes per sec                0          0          0
VM out bytes Per Sec     bytes per sec                0          0          0
```

To rename baselines, the dbms_workload_repository package provides the rename_baseline procedure. As parameters, the procedure requires the old and the new name. For example, the following call shows how to rename the baseline from TEST to TEST1:

```
dbms_workload_repository.rename_baseline(
  old_baseline_name => 'TEST',
  new_baseline_name => 'TEST1'
);
```

Finally, to drop baselines, the dbms_workload_repository package provides the drop_baseline procedure. As parameters, the procedure requires the name of the baseline and, optionally, a flag that specifies whether the snapshots associated with the baseline are to be dropped (by default, they aren't). For example, the following call shows how to drop the baseline named TEST1 as well as the snapshots associated with it:

```
dbms_workload_repository.drop_baseline(
  baseline_name => 'TEST1',
  cascade       => TRUE
);
```

### Managing the Moving Window Baseline

There's not much to manage for the moving window baseline. In fact, you can modify only the window size by calling the modify_baseline_window_size procedure of the dbms_workload_repository package. As a parameter, the procedure requires the new size specified in number of days. For example, the following call shows how to set it to 30 days:

```
dbms_workload_repository.modify_baseline_window_size(window_size => 30);
```

The only requirement to fulfill when calling the modify_baseline_window_size procedure is that the new window size can't be greater than the retention period used for the snapshots. If that requirement isn't fulfilled, the database engine raises the following exception: `ORA-13541: system moving window baseline size greater than retention`.

To display the current window size, you can use the following query (notice that these are the default values used from version 11.1 onward):

```
SQL> SELECT baseline_name, moving_window_size
  2  FROM dba_hist_baseline
  3  WHERE baseline_type = 'MOVING_WINDOW';

BASELINE_NAME        MOVING_WINDOW_SIZE
-------------------- ------------------
SYSTEM_MOVING_WINDOW                  8
```

## Statspack

This section describes how to install and configure Statspack, take snapshots, and manage baselines. How to take advantage of the information stored in the Statspack repository is described later, in the "Analysis Without Diagnostics Pack" section.

---

■ **Tip** Oracle Database manuals no longer provide information about Statspack. You can find detailed information about installing, configuring, managing, and using Statspack in the spdoc.txt file, which is available (along with installation scripts and other utilities) under the $ORACLE_HOME/rdbms/admin directory. The Oracle Support note "Installing and Using Standby Statspack in 11g" (454848.1) provides information about using Statspack with a standby database open in read-only mode.

---

## Performing Installation

To install Statspack, you have to connect as user `sys` and run the `spcreate.sql` script, which is available in the `$ORACLE_HOME/rdbms/admin` directory. The script creates a user named `perfstat` that contains most of the objects required to run Statspack. In addition, it creates a number of public synonyms and some views in the `sys` schema. During execution, the script asks for the password of the `perfstat` user, the temporary tablespace that has to be used for it, and the tablespace that the tables and indexes have to be stored in. Note that both tablespaces have to already exist before executing the script. If you don't want to create new tablespaces, just select the default temporary tablespace and the `sysaux` tablespace.

## Configuring the Repository

The Statspack configuration, which is stored in the `stats$statspack_parameter` table of the `perfstat` schema, is based on three types of parameters:

- *Snapshot level*: Defines the data that is stored when a snapshot is taken. Table 5-2 briefly describes the available snapshot levels. Also, the `stats$level_description` table contains a short description of each level.

***Table 5-2.*** *Statspack's Snapshot Levels*

| Level | Description |
|---|---|
| 0 | Captures general performance statistics. |
| 5 | Captures general performance statistics (as level 0), as well as statistics about SQL statements that cross a threshold. This is the default level. |
| 6 | Captures all statistics gathered at lower levels, as well as execution plans (including usage statistics). |
| 7 | Captures all statistics gathered at lower levels, as well as segment-level statistics (for example, the number of logical and physical reads) for the segments that cross a threshold. |
| 10 | Captures all statistics gathered at lower levels, as well as statistics about latches. |

- *SQL statement thresholds*: Six thresholds (number of executions, number of parse calls, number of physical reads, number of logical reads, amount of sharable memory, and number of child cursors) that are used to determine whether a SQL statement is captured. The capture of a SQL statement takes place only when at least one of them is crossed.
- *Segment statistics thresholds*: Seven thresholds (number of logical reads, number of physical reads, number of buffer busy waits, number of row lock waits, number of ITL waits, number of global cache-consistent read blocks, and number of global cache current blocks) that are used to determine the segments for which statistics are captured. The capture of statistics for a given segment takes place only when at least one of them is crossed.

The following query shows the actual values (in this case, the default values in place after the installation) of each parameter:

```
SQL> SELECT parameter, value
  2  FROM stats$statspack_parameter
  3  UNPIVOT (
  4    value FOR
  5    parameter IN (snap_level, executions_th, parse_calls_th, disk_reads_th, buffer_gets_th,
  6                  sharable_mem_th, version_count_th, seg_phy_reads_th, seg_log_reads_th,
  7                  seg_buff_busy_th, seg_rowlock_w_th, seg_itl_waits_th, seg_cr_bks_rc_th,
  8                  seg_cu_bks_rc_th)
  9  );

PARAMETER          VALUE
---------------- -------
SNAP_LEVEL             5
EXECUTIONS_TH        100
PARSE_CALLS_TH      1000
DISK_READS_TH       1000
BUFFER_GETS_TH     10000
SHARABLE_MEM_TH  1048576
VERSION_COUNT_TH      20
SEG_PHY_READS_TH    1000
SEG_LOG_READS_TH   10000
SEG_BUFF_BUSY_TH     100
SEG_ROWLOCK_W_TH     100
SEG_ITL_WAITS_TH     100
SEG_CR_BKS_RC_TH    1000
SEG_CU_BKS_RC_TH    1000
```

To modify the value of one of the parameters, the `statspack` package provides the `modify_statspack_parameter` procedure. For each Statspack parameter, the procedure has an input parameter. For example, the following call changes the snapshot level to 6:

```
statspack.modify_statspack_parameter(i_snap_level => 6)
```

## Taking and Purging Snapshots

To take a snapshot, you have to call one of the `snap` subroutines of the `statspack` package. Two subroutines exist: a function and a procedure. Both of them accept an input parameter for every configuration parameter described in the previous section. All parameters are optional, so you can simply take a snapshot without specifying any parameter, like this:

```
perfstat.statspack.snap();
```

To purge one or several snapshots, provided that they aren't defined as baselines, you have to call one of the `purge` subroutines of the `statspack` package. Eight subroutines exist. On the one hand, the same functionality is implemented as a function and as a procedure. On the other hand, four methods are available to specify which snapshots have to be deleted:

- All snapshots between a begin and end snapshot ID (parameters `i_begin_snap` and `i_end_snap`)
- All snapshots created between a begin and end date (parameters `i_begin_date` and `i_end_date`)

- All snapshots created prior to a specific date (parameter i_purge_before_date)
- All snapshots older than a specific number of days (parameter i_num_days)

Note that by default, data about SQL statements and execution plans isn't purged. To purge it, an *extended purge* must be activated by setting the i_extended_purge parameter to TRUE. For example, the following call purges all snapshots (including SQL statements and execution plans) taken before April 2014:

```
statspack.purge(
  i_purge_before_date => to_date('2014-04-01','YYYY-MM-DD'),
  i_extended_purge    => TRUE
);
```

Since, out of the box, snapshots are neither automatically taken nor purged after a specific period of time, you should schedule two jobs that carry out those tasks. The following are examples (note that both jobs should be created with the perfstat user):

- Take a snapshot every 15 minutes.

```
dbms_scheduler.create_job(
  job_name        => 'TAKE_STATSPACK_SNAPSHOT',
  job_type        => 'PLSQL_BLOCK',
  job_action      => 'perfstat.statspack.snap();',
  start_date      => sysdate,
  repeat_interval => 'FREQ = HOURLY; BYMINUTE = 0,15,30,45',
  enabled         => TRUE,
  comments        => 'take STATSPACK shapshot'
);
```

- Purge snapshots created more than 35 days ago.

```
dbms_scheduler.create_job(
  job_name        => 'PURGE_STATSPACK_SNAPSHOTS',
  job_type        => 'PLSQL_BLOCK',
  job_action      => 'statspack.purge(i_num_days => 35, i_extended_purge => TRUE);',
  start_date      => sysdate,
  repeat_interval => 'FREQ = HOURLY; BYMINUTE = 50',
  enabled         => TRUE,
  comments        => 'purge STATSPACK shapshots'
);
```

You can find other examples in two scripts distributed by Oracle: spauto.sql and sppurge.sql. Both are available in the $ORACLE_HOME/rdbms/admin directory.

---

■ **Note** In a Real Application Clusters environment, it's necessary to separately schedule the jobs on every database instance.

---

### Managing Baselines

A *baseline* is a regular snapshot that is marked, by setting a flag, as a baseline. As a result, it's excluded from the purging process. To manage that flag, the `statspack` package provides two sets of procedures and functions. The first set, composed by the subroutines named `make_baseline`, is used to set the flag for one or several snapshots. The second one, composed by the subroutines named `clear_baseline`, is used to unset the flag for one or several snapshots. The `statspack` package provides both functions and procedures that implement the same functionality. The only difference between them is that only the functions return the number of snapshots for which the flag has been modified. To identify the snapshots to be altered, you can specify either a range of IDs or a period of time. For example, the following call marks all snapshots between the two specified timestamps as baselines:

```
perfstat.statspack.make_baseline(
  i_begin_date => to_date('2014-04-02 17:00:00','YYYY-MM-DD HH24:MI:SS'),
  i_end_date   => to_date('2014-04-02 17:59:59','YYYY-MM-DD HH24:MI:SS')
);
```

In a similar way, the following call unsets the flag for all snapshots between the two specified timestamps:

```
perfstat.statspack.clear_baseline(
  i_begin_date => to_date('2014-04-02 00:00:00','YYYY-MM-DD HH24:MI:SS'),
  i_end_date   => to_date('2014-04-02 23:59:59','YYYY-MM-DD HH24:MI:SS')
);
```

## Analysis With Diagnostics Pack

For an analysis with the Diagnostics Pack, I advise using the performance pages provided by Enterprise Manager (it's irrelevant whether you use Database Control, Grid Control, or Cloud Control). As described in Chapter 4, the Performance Home and Top Activity pages display either real-time or historical information. Since this chapter focuses on analyzing performance problems that occurred in the past, it goes without saying that historical information has to be used. In this mode, by default, information about the last week is available.

The analysis with historical information is basically the same as the one with real-time data. Hence, refer to Chapter 4 for detailed information. Here are the most relevant differences:

- With historical data, a 30-minute interval (instead of 5) is used to select the period for which you want to display more details about the database load. (For more flexibility, if possible, you should use ASH Analytics.)
- Data is less detailed or even missing because not all real-time data is stored in AWR. Nevertheless, you should be able to have access to enough information about the top consumers.
- You aren't able to search for a specific session. In fact, the Search Sessions item in the Performance menu isn't available. Sessions can be accessed only through the Top Sessions table.

In addition to its integration inside Enterprise Manager, AWR provides a series of reports that can be used to assess the load during a specific period of time. Here are the three most commonly used:

- The AWR report summarizes the operations performed during a period of time specified as input. It can be produced either through Enterprise Manager or by executing the `$ORACLE_HOME/rdbms/admin/awrrpt.sql` script. Since this report is based on the Statspack report, take a look at the next section for information about its interpretation.

- The Compare Periods report compares the operations performed during two distinct periods of time specified as input. It's especially useful to find out the differences between a baseline period and a period when the database engine suffered from performance problems. It can be produced either through Enterprise Manager or by executing the `$ORACLE_HOME/rdbms/admin/awrddrpt.sql` script.
- The SQL Statement report provides detailed information about a SQL statement. Refer to Chapter 10, specifically the "Automatic Workload Repository and Statspack" section, for additional information.

## Analysis Without Diagnostics Pack

Provided you aren't focusing on a single SQL statement, with Statspack, the analysis of a performance issue starts by executing the `$ORACLE_HOME/rdbms/admin/spreport.sql` script. The script, after asking for the period of time the report has to be generated for (I advise you to reference two consecutive snapshots), writes the report in an output file. Even though the aim of this section is to describe how to read such a report, it's neither feasible nor sensible to cover all sections. In fact, a report not only might be very long (a typical length is 2,000–3,000 lines), but also might contain content that most of the time is not necessary for assessing what's going on. In fact, plenty of content is there just in case you might need it.

■ **Tip** AWR is an evolution of Statspack. Therefore, the explanations on how to interpret a Statspack report also apply to the AWR report.

The analysis starts by looking at the initial part of the report (about 100 lines). The information it contains is neither ordered in the best possible way nor always very interesting. However, since the initial part of the report isn't very long, it makes sense to go through it sequentially.

A Statspack report starts by providing some self-explanatory information about the instance and the server running it:

```
Database    DB Id    Instance     Inst Num  Startup Time    Release     RAC
~~~~~~~~ ----------- ------------ -------- --------------- ----------- ---
          2532911053 DBM11203            1 23-Apr-14 16:33 11.2.0.3.0  NO

Host Name            Platform                CPUs Cores Sockets   Memory (G)
~~~~ ---------------- ---------------------- ----- ----- ------- ------------
     helicon          Linux x86 64-bit           8     8       2          7.8
```

From the previous excerpt, keep in mind the number of CPU cores that the database server has. Later, that information is required in order to assess whether the database engine is CPU bound.

■ **Caution** When a database server is equipped with CPUs using simultaneous multithreading, the `NUM_CPUS` value in the `v$osstat` view, which is the value shown in the Statspack report in the `CPUs` column, is set to the total number of threads. Be aware that using the number of threads to assess whether the database is CPU bound is probably misleading. In fact, using all threads at 100% isn't feasible. Even though basing the check on the number of CPU cores might be considered too conservative, that's the capacity you know you can use for sure (provided you aren't working in an environment where virtualization is playing a role).

The report continues by providing information about the period of time it covers (beginning, end, and duration), the number of sessions at the beginning and at the end of the period, as well as, from version 11.1 onward, the DB time (106.55) and the CPU utilization (28.35). In this part, you should carefully check that the report covers the period of time you have to analyze. Note that the average number of active sessions (7.1), which is available as of version 11.1.0.7 only, is computed by dividing the DB time by the elapsed time:

```
Snapshot       Snap Id     Snap Time      Sessions Curs/Sess Comment
~~~~~~~~    ---------- ----------------- -------- --------- ------------------
Begin Snap:        548 23-Apr-14 18:30:40       57       1.6
  End Snap:        549 23-Apr-14 18:45:40       59       1.5
   Elapsed:      15.00 (mins) Av Act Sess:      7.1
   DB time:     106.55 (mins)      DB CPU:      28.35 (mins)
```

The report goes on to provide the size of the most important SGA components. Be aware that if either the buffer cache or the shared pool were resized during the observed period, the value at the end of the period would be shown. Otherwise, as in the following example, only the value at the beginning of the period is shown:

```
Cache Sizes            Begin        End
~~~~~~~~~~~       ---------- ----------
    Buffer Cache:       728M            Std Block Size:         8K
     Shared Pool:       260M                Log Buffer:     7,992K
```

Next, you see plenty of metrics about the processing performed per second, per transaction, and, for a couple of metrics, per execution and per call:

```
Load Profile              Per Second    Per Transaction    Per Exec    Per Call
~~~~~~~~~~~~      ------------------  ----------------- ----------- -----------
      DB time(s):                7.1                0.0        0.01        0.00
       DB CPU(s):                1.9                0.0        0.00        0.00
       Redo size:          392,163.4            1,928.3
   Logical reads:          406,805.1            2,000.3
   Block changes:            2,822.9               13.9
  Physical reads:              579.7                2.9
 Physical writes:              377.8                1.9
      User calls:            2,895.2               14.2
          Parses:                0.6                0.0
     Hard parses:                0.0                0.0
W/A MB processed:                0.1                0.0
          Logons:                0.0                0.0
        Executes:            1,364.1                6.7
       Rollbacks:                0.0                0.0
    Transactions:              203.4
```

Most of the previous metrics can't be used directly to assess whether the database instance was experiencing a performance problem. Their main purpose is twofold. First, they give you a general feeling about the load. For example, in the previous case, we see that there are 203.4 transactions per second and that each of them, on average, carries out 6.7 executions. So, you know that the system is for sure doing some processing. Second, they help determine whether the system is doing more or less work than expected, and more important, you can compare these values with a baseline. Nevertheless, two of the metrics can be used to judge the load of the database instance directly (unfortunately, both values aren't available in version 10.2.):

- The `DB time Per Second` (7.1) is equivalent to the average number of active sessions. Based on the rule of thumb described in Chapter 4, with an average number of active sessions that is equivalent to the number of CPU cores (8), the system can be considered fairly busy.
- The `DB CPU per second` (1.9) tells you whether the instance is CPU bound. In fact, by comparing it with the number of CPU cores (8), you can determine the average CPU utilization. In this case, the database instance consumed only about 24% (1.9/8*100) of the total CPU capacity. Hence, if the server wasn't shared with other database instances or applications, the CPU capacity was adequate for the load.

The report continues with a set of ratios that are of very limited usage. The only sensible thing you can do with them is to compare them with a baseline to determine whether something has changed:

```
Instance Efficiency Indicators
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
            Buffer Nowait %:  100.00       Redo NoWait %:   99.98
            Buffer  Hit   %:   99.86  Optimal W/A Exec %:  100.00
            Library Hit   %:  100.01        Soft Parse %:   98.99
         Execute to Parse %:   99.96         Latch Hit %:   99.96
Parse CPU to Parse Elapsd %:  103.03     % Non-Parse CPU:   98.84

 Shared Pool Statistics        Begin   End
                               ------  ------
             Memory Usage %:   66.47   66.55
    % SQL with executions>1:   71.80   71.85
  % Memory for SQL w/exec>1:   72.70   72.88
```

The next part shows a resource usage profile of the top 5 events (including the CPU utilization). Simply put, in this table, the DB time is broken up to show how it was spent. For example, according to the following excerpt, 71.6% of the time was spent doing single-block reads:

```
Top 5 Timed Events                                              Avg %Total
~~~~~~~~~~~~~~~~~~                                             wait   Call
Event                                       Waits    Time (s)   (ms)   Time
----------------------------------------- ------------ ----------- ------ ------
db file sequential read                        520,240       4,567      9   71.6
CPU time                                                     1,620          25.4
log file sync                                  182,275          94      1    1.5
log file parallel write                        178,406          39      0     .6
read by other session                            2,693          27     10     .4
```

Contrary to the AWR report, the list of top 5 events doesn't show the wait classes (for example, `User I/O`, `System I/O`, `Commit`, or `Concurrency`). If you see an event for which you ignore the wait class it belongs to, you can execute a query like the following to find it:

```
SQL> SELECT wait_class
  2  FROM v$event_name
  3  WHERE name = 'db file parallel read';

WAIT_CLASS
----------
User I/O
```

The report continues by giving OS-level information about the CPU utilization (be aware that up to and including version 11.1.0.6, the `Instance CPU` section provides only the percentage values and uses different labels):

```
Host CPU  (CPUs: 8  Cores: 8  Sockets: 2)
~~~~~~~~              Load Average
                      Begin     End      User  System    Idle     WIO     WCPU
                    ------- -------   ------- ------- ------- ------- --------
                       5.98    7.09     23.83    0.89   74.75   21.86

Instance CPU
~~~~~~~~~~~~                                       % Time (seconds)
                                              -------- --------------
                     Host: Total time (s):                  7,001.7
                  Host: Busy CPU time (s):                  1,768.2
                   % of time Host is Busy:        25.3
             Instance: Total CPU time (s):                  1,739.9
          % of Busy CPU used for Instance:        98.4
        Instance: Total Database time (s):                  6,500.3
  %DB time waiting for CPU (Resource Mgr):         0.0
```

The purpose of the previous excerpt is twofold:

- To check whether the host (not the database instance) is CPU bound. The `% of time Host is Busy` value provides you the information you need. If it's low (as in this case), everything is fine. If its value is high (close to 100%, provided the CPUs aren't using simultaneous multithreading), the resource usage profile of the top 5 events, as well as all other statistics about waits events, might be misleading. In fact, in the case of a CPU shortage, many statistics might be artificially inflated. Hence, your first goal would be to find a method to reduce the CPU utilization.
- To determine whether the CPU utilization at the OS level is primarily due to the database instance you are looking at. This is an essential piece of information when a server runs several database instances or even other applications. The most important value to check, which shows how much of the total CPU utilization is due to the database instance you are looking at, is `% of Busy CPU used for Instance`. If you see values of less than 80–90%, it means that other applications are using a non-negligible amount of CPU. For example, in the previous excerpt, notice that almost all of the CPU utilization (98.4%) is due to this database instance. This means there's nothing else using a lot of CPU on this server.

After the OS-level information about the CPU utilization, OS-level information about the memory utilization is shown. With it, you can know how much memory is available on the host (8 GB) and what percentage is used for the SGA and PGA of the database instance you are looking at (15.1%):

```
Memory Statistics                          Begin          End
~~~~~~~~~~~~~~~~~                   ------------ ------------
                  Host Mem (MB):        7,974.6      7,974.6
                   SGA use (MB):        1,019.4      1,019.4
                   PGA use (MB):          175.1        183.1
    % Host Mem used for SGA+PGA:           15.0         15.1
```

The last information from the first 100 lines provides time model statistics. As discussed in Chapter 4, based on this data, you can know the amount of time spent by the database engine processing key operations. In the following example, the statistics point out that most of the time (95.5%) is spent executing SQL statements:

```
Statistic                                       Time (s) % DB time
----------------------------------- ------------------- ---------
sql execute elapsed time                         6,103.6      95.5
DB CPU                                           1,701.1      26.6
parse time elapsed                                  26.8        .4
sequence load elapsed time                           0.1        .0
PL/SQL execution elapsed time                        0.0        .0
repeated bind elapsed time                           0.0        .0
DB time                                          6,392.9
background elapsed time                            107.5
background cpu time                                 38.8
```

Based on the information provided in the first 100 lines of the report, you should have a fairly good understanding of what's going on—for example, to what extent the system is loaded and the main operations it performs. The next step is to find out what the top SQL statements are. For this purpose, the report contains several lists ordered by different criteria (CPU, elapsed time, number of logical reads, number of physical reads, number of executions, and number of parse calls). Since the elapsed time is, in most situations, the most important criteria, I advise you to continue the analysis by looking at the section called `SQL ordered by Elapsed time`.

Before looking at the list itself, it's important to check whether the captured SQL statements account for a relevant part of the DB time. If, as in the following excerpt, the captured SQL statements account for most of the DB time (95.4%), the list contains useful information:

```
-> Total DB Time (s):            6,393
-> Captured SQL accounts for   95.4% of Total DB Time
-> SQL reported below exceeded  1.0% of Total DB Time
```

However, in case the captured SQL statements account for a small percentage of the DB time (for example, 10–20%), the list is almost useless. In fact, either SQL statements that had a major contribution in the DB time were flushed from the library cache before the snapshot was taken, or there's no SQL statement that consumed a significant part of the DB time. In the former case, the repository doesn't contain enough information to completely analyze the situation. In the latter, many SQL statements are responsible for the load. Hence, as explained in the "Analysis Roadmap" section in Chapter 4, it doesn't make sense to focus on the top SQL statements.

As illustrated in the following excerpt, for each SQL statement, you can see not only information about the total and average elapsed time, but also figures about CPU utilization and physical reads:

```
  Elapsed                Elap per           CPU                        Old
  Time (s)   Executions  Exec (s)  %Total   Time (s)  Physical Reads Hash Value
---------- ------------ ---------- ------ ---------- --------------- ----------
   1861.91      124,585       0.01   29.1      32.06         194,485 3739063178
Module: Swingbench User Thread
select customer_id, cust_first_name ,cust_last_name ,nls_languag
e ,nls_territory ,credit_limit ,cust_email ,account_mgr_id  from
 customers where customer_id = :1
```

```
   1354.48        7,087       0.19   21.2    1241.76           7,834 1481390170
Module: Swingbench User Thread
SELECT /*+  first_rows index(customers, customers_pk) index(orde
rs, order_status_ix) */  o.order_id, line_item_id, product_id, u
nit_price, quantity, order_mode, order_status, order_total, sale
s_rep_id, promotion_id, c.customer_id, cust_first_name, cust_las

    648.93       36,705       0.02   10.2      20.58          70,411 3476971243
Module: Swingbench User Thread
insert into orders(ORDER_ID, ORDER_DATE, CUSTOMER_ID, WAREHOUSE_
ID) values (:1 , :2 , :3 , :4 )
```

If, as in the previous excerpt, a limited number of SQL statements is responsible for most of the DB time (the top 3 were responsible for more than 60% of the DB time), you have found the SQL statement that you have to focus on. Then, based on the hash value, you can use the `sprepsql.sql` script to get all available information about the SQL statement (refer to Chapter 10, specifically in the "Automatic Workload Repository and Statspack" section).

All other sections in the report might be useful to get more-detailed information about a specific behavior or configuration of the system. For example, in a case like the one described in this section, where the system is disk I/O bound, for each top event related to disk I/O operations, you should check the histograms provided in the `Wait Event Histogram` section. Based on them, and on the knowledge of the I/O subsystem configuration, you should be able to determine whether the disk I/O operations perform as expected. According to the following excerpt, you see that while the log writes almost always take less than 1 millisecond, the reads are up to an order of magnitude slower.

```
                           Total ----------------- % of Waits ------------------
Event                      Waits  <1ms  <2ms  <4ms  <8ms <16ms <32ms  <=1s   >1s
-------------------------- ----- ----- ----- ----- ----- ----- ----- ----- -----
db file sequential read     520K   2.1   2.9  19.8  49.1  18.2   4.4   3.4
log file parallel write     178K  98.7    .3    .3    .2    .3    .2    .0
```

## On to Part 3

This chapter describes how to install, configure, and manage the two repositories that Oracle Database provides to analyze performance problems that happened in the past: AWR and Statspack. In addition, it outlines how to go through a Statspack report, which is very similar to an AWR report. These are the main checks you should carry out.

In general, the aim isn't to investigate performance problems, but to avoid them in the first place. Based on my experience, there are two major causes of performance problems: not designing databases and/or applications for performance, and poor configuration of the query optimizer. The latter is critical, because every SQL statement executed by the database engine goes through the query optimizer. For this reason, the next part not only explains how the query optimizer works, but also provides information on how to correctly configure it.

PART III

# Query Optimizer

*Make the best use of what is in your power, and take the rest as it happens.*

—Epictetus[1]

Every single SQL statement sent to the database before being processed by the SQL engine must be turned into an execution plan. In fact, an application specifies only what data must be processed through SQL statements, not how to process it. The aim of the query optimizer isn't only to produce execution plans describing how to process data but also, and most important, to deliver efficient execution plans. Failing to do so may lead to abysmal performance. Precisely for this reason, a book about database performance must deal with the query optimizer.

The aim of this part, however, isn't to cover the internal workings of the query optimizer. Instead, a very pragmatic approach is presented here, aimed at describing the essential features of the query optimizer you have to know. Chapter 6 introduces basic concepts and the architecture of the query optimizer. Chapter 7 and 8 discuss the statistics used by the query optimizer. Chapter 9 describes the initialization parameters influencing the behaviour of the query optimizer and how to set them. Finally, Chapter 10 outlines different methods of obtaining execution plans, as well as how to read them and recognize inefficient ones.

Two main query optimizers are available in Oracle Database, the *rule-based optimizer* (RBO) and the *cost-based optimizer* (CBO). As of Oracle Database 10g, the use of the rule-based optimizer is no longer supported and, therefore, won't be covered here. Throughout this book, when you read the term *query optimizer*, I always mean the cost-based optimizer.

[1]http://www.quotationspage.com/quote/2525.html

CHAPTER 6

# Introducing the Query Optimizer

The query optimizer is one of the building blocks of the SQL engine. Its purpose is to produce efficient execution plans in a timely manner. The time constraint is essential because in most situations it isn't sensible to spend too much time on the optimization phase. What does "too much time" mean? In general, the parse phase, which contains the work performed by the query optimizer, should be much shorter than the execution phase. The only situation in which having a parse phase longer than the execution phase is acceptable is when a cursor can be reused for many executions. As discussed in Chapter 2, the ability to cache the shared SQL area associated with a cursor in the SGA was also introduced for this very same purpose.

The aim of this chapter is to provide an overview of the information used by the query optimizer to carry out its work, to describe the architecture of the SQL engine, and explain how its components interact to process SQL statements. It also provides information about the query transformations carried out by the query optimizer.

## Fundamentals

The query optimizer, to choose an execution plan, has to answer questions like the following:

- Which is the optimal access path to extract data from each table referenced in the SQL statement?
- Which are the optimal join methods and join orders through which the data of the referenced tables will be processed?
- When should aggregations and/or sorts be processed during SQL statement execution?
- Is it beneficial to use parallel processing?

In practice, however, the query optimizer doesn't directly answer these questions. Instead, it explores the so-called *search space*, which consists of all potential execution plans, in pursuit of the optimal execution plan. To decide which execution plan is the optimal one, the query optimizer estimates the cost of a number of execution plans and picks the one with the lowest cost. For example, the search space of the following query is composed of more than a hundred possible execution plans. With the `search_space.sql` script I was able to reproduce 122 of them. Their cost goes from 20 to more than 100,000:

```
SELECT *
FROM t1 JOIN t2 ON t1.id = t2.t1_id
WHERE t1.n = 1 AND t2.n = 2
```

Because the goal of the query optimizer is to find the cheapest execution plan as quickly as possible, it is useful that the query optimizer doesn't evaluate all execution plans for any but the simplest of SQL statements. In other words, the query optimizer explores only a subset of the search space. Simply put, based on a heuristic selection, the query optimizer starts evaluating the most promising execution plan and then considers different execution plans until the cheapest one is found or too many alternatives are probed. It implements a *branch-and-bound* algorithm. A *branch* is an alternative (for example, an access path or a join method), and the *bound* is the cost of the best execution plan found so far—the query optimizer discards a branch (and possibly all its subbranches) as soon as its current cost gets higher than the bound.

Figure 6-1 shows that to estimate the cost of an execution plan, the query optimizer considers not only the SQL statement to be optimized, but also a number of other inputs. Some of those other inputs are stored in the data dictionary and hardly ever change, or are not expected to frequently change, at runtime. They can be considered as a static environment in which an application runs. Other inputs, though, not only might change frequently or even for every execution, but might not even be known until runtime. Because of such inputs, a different execution plan might be generated every time the query optimizer processes a given SQL statement.

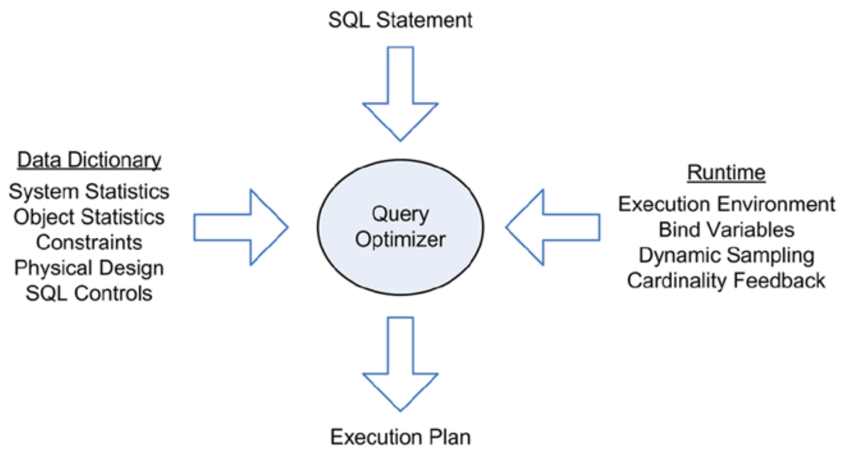


***Figure 6-1.*** *The query optimizer considers a number of inputs to produce an execution plan*

Some of the inputs in Figure 6-1 are used to determine which options are available. Others are used to estimate the cost of potential execution plans. The following list briefly describes these inputs and, when possible, points out which part of the book provides more information about them:

> **System statistics:** The query optimizer must know the power of the system it's running on to provide accurate estimates. For that purpose, system statistics describe both the machine running the database engine and the performance figures of the storage subsystem. Chapter 7 describes which system statistics are available, how to manage them, and how the query optimizer uses them to improve its estimations.
>
> **Object statistics:** Table, index, and column statistics, which are stored in the data dictionary, are essential because they describe the data stored in the database. For example, a query optimizer that's aware only of the SQL statements to be processed and the structure of the referenced objects can't provide efficient execution plans. To generate efficient execution plans, the query optimizer has to be able to quantify the amount of data to be processed and the cost of processing it by the various alternatives that the query optimizer has to choose from. Chapter 8 describes which object statistics are available and how to manage them.

**Constraints:** The query optimizer takes advantage of `NOT NULL` constraints, unique key constraints, primary key constraints, foreign key constraints, and some check constraints. As described later in this chapter, constraints are also central to assessing whether applying certain query transformations is possible or sensible. In addition, as described in Chapter 13, constraints also enable additional access paths. For these reasons, it's advisable to create all known constraints when defining the objects to be stored in the database.

**Physical design:** There are three main physical design areas that have an effect on the query optimizer. First, Oracle Database offers five strategies to store data: heap-organized tables (this is the default), index-organized tables, external tables, index clusters, and hash clusters. In addition, heap-organized tables and index-organized tables can be partitioned. One or several access paths are associated to each strategy. Chapter 13 covers those access paths in detail. Second, for each data storage strategy, with the exception of external tables, Oracle Database can manage various types of indexes. Each of the index types (described in Chapter 13) adds specific access paths. In addition, all storage strategies support materialized views that, with query rewrite, give the query optimizer additional ways to optimize queries. This topic is covered in Chapter 15. Third, even though column order in tables don't impact available access paths, it influences some of the costs computed by the query optimizer. The reasons behind this are explained in Chapters 7 and 16.

**SQL controls:** In most situations, the query optimizer is able to generate optimal execution plans. But there are cases where it's unable to do so, and Oracle Database provides a number of features to ameliorate trouble when those cases occur. Chapter 11 covers those features in detail. For the moment, it's important only to know that features like stored outlines, SQL profiles, and SQL plan baselines allow you to store in the data dictionary information that influences the decisions taken by the query optimizer while generating execution plans.

**Execution environment:** A set of initialization parameters controls the behavior of the query optimizer. Such parameters are set at the system level through the initialization or server parameter file of the database engine.When required, they can be overridden at the session level by issuing the `ALTER SESSION` statement. As described in Chapter 11, some of them can even be changed at the SQL statement level. There are parameters you can configure at the operating system level server side as well as client side. The National Language Support (NLS) parameters are one example—they can be configured on both sides of the connection. In fact, NLS parameters can also be set with environment variables or, on Windows boxes, through the registry. Especially with the client-side settings, you must be very careful: it's often forgotten, for client/server applications, that some client-side environment variables affect the query optimizer. Chapter 9 discusses the most important initialization parameters that control the behavior of the query optimizer. Some NLS parameters are covered in Chapter 13.

**Bind variables:** Bind variables are fully described in Chapter 2. Besides the value, the definition (i.e., datatype) of bind variables also has a strong influence on the execution plans generated by the query optimizer.

**Dynamic sampling:** Based on object statistics stored in the data dictionary, the query optimizer can't always accurately estimate the cost of an operation or predicate. When the query optimizer recognizes such a case, in some situations it can dynamically gather additional statistics during query optimization. To do so, the query optimizer executes recursive queries against the objects referenced by the SQL statement to be optimized. This feature is described in Chapter 9.

**Cardinality feedback (also called statistics feedback):** Either because of complex predicates or missing input information, the query optimizer can't always compute accurate estimations. When the query optimizer recognizes that it's computing low-quality estimates for a SQL statement, the generated execution plan is annotated. The accuracy of the estimations is then checked after the SQL statement's execution. If the actual and estimated values differ significantly, information about the correct values is stored, and a re-optimization is forced at the next SQL statement execution. Note that the re-optimization is forced to let the query optimizer take advantage of the information gained during the first execution, thereby increasing the chances of producing an optimal execution plan. This feature is available as of version 11.2 only.

The version of the Oracle Database software in use also determines which query optimizer features are available. It's essential to recognize that even knowing the five numbers of the Oracle Database version is insufficient to exactly know which features are available. You also need to know whether the release in use is Enterprise Edition or Standard Edition. In addition, note that some patches might also control the availability of specific query optimizer features and their behavior.

## Architecture

The query optimizer, which can be decomposed in the *logical optimizer* and the *physical optimizer*, is just one of the building blocks of the SQL engine.

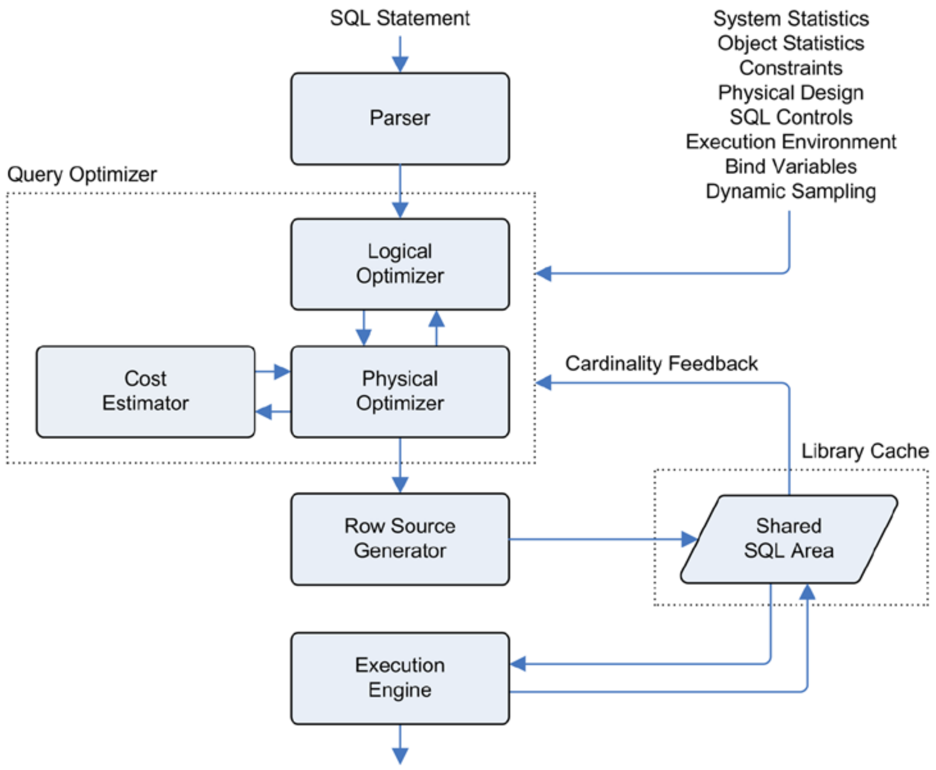


***Figure 6-2.*** *The architecture of the SQL engine*

As shown in Figure 6-2, the following are the key components:

**Parser:** This is the first component involved in the execution of a SQL statement. Its purpose is to deliver to the query optimizer a parsed representation of the SQL statement. Additional information about the work the parser performs is provided in Chapter 2, specifically in the "How Parsing Works" section.

**Logical optimizer:** During the logical optimization phase, the query optimizer produces new and semantically equivalent SQL statements by applying different query transformation techniques. The purpose of the logical optimizer is to select the best combination of query transformations. In doing so, the search space is increased, and execution plans can be explored that wouldn't be considered without such query transformations. Later in this chapter, in the "Query Transformations" section, I provide additional information about the work performed by this component.

**Physical optimizer:** During the physical optimization phase, several operations are performed. At first, several execution plans for each SQL statement resulting from the logical optimization are generated. Then every one of them is passed to the cost estimator to let it calculate a cost. Finally, the execution plan with the lowest cost is selected. Simply put, the physical optimizer explores the search space to find the most efficient execution plan.

**Cost estimator:** Based on the inputs introduced in Figure 6-1, the cost estimator calculates the cost for the execution plan submitted by the physical optimizer.

**Row source generator:** The execution plan produced by the query optimizer can't be directly executed by the execution engine. It has to be converted into a tree of row source operations and stored in the library cache.

**Execution engine:** This component executes the row source operations produced by the row source generator. If monitoring for cardinality feedback is active, the execution engine verifies (after execution) whether actual and estimated values differ significantly. If a significant difference is found, information about the correct values is stored in the shared SQL area, and a re-optimization on the next execution is forced.

## Query Transformations

The query optimizer uses a multitude of query transformations to produce new and semantically equivalent SQL statements. Among those query transformations, depending on the method used to decide whether they are applied, two approaches can be distinguished:

**Heuristic-based query transformations** are applied when specific conditions are met. They're expected to lead to better execution plans in most situations.

**Cost-based query transformations** are applied when, according to the cost computed by the cost estimator, they lead to execution plans with lower costs than the original statement.

The following sections introduce two dozen query transformations and provide examples of their use. The aim is not to describe them extensively but simply to give you an idea of what's going on under the hood during the logical optimization phase. Therefore, no detailed information about prerequisites or limitations is given. Also note that some query transformations are more powerful or only available in recent releases. Hence, not every query transformation described in this chapter is available in earlier releases, such as 10.2 and 11.1.

■ **Note** I tried to keep the following examples as simple as possible. As a result, at first glance, some query transformations may seem useful only when the query optimizer has to process SQL statements of bad quality. By *bad quality* I mean SQL statements that include, for example, redundant or conflicting operations. But you have to consider that the query optimizer will have to process SQL statements that are much more complex than in my examples. Just think about the case of a query referencing several views that in turn reference other views, or a query generated by general-purpose and ad-hoc query tools. When the query optimizer puts everything together, it's not uncommon that, for example, redundant or conflicting operations appear. In addition, it's a good thing that the query optimizer recognizes odd situations and avoids performing unnecessary processing. In addition, some query transformations allow you to write SQL statements in the most natural, readable way, without exchanging clarity for performance. In fact, some query transformations are (quite) common SQL optimization techniques that, when applied manually, produce less readable SQL statements.

## Count Transformation

The purpose of *count transformation* is to transform `count(column)` expressions to `count(*)`. This query transformation was introduced because a `count(*)` can be processed using a larger selection of indexes than a `count(column)`. Chapter 13 talks more about this. Count transformation is a heuristic-based query transformation that can be applied when the column referenced in the `count` function has an associated `NOT NULL` constraint (check constraints can't be used for this purpose, though). Note that count transformation also transforms `count(1)` expressions to `count(*)`.

The following example, based on the `count_transformation.sql` script, illustrates this query transformation. Notice that the orginal query contains the `count(n2)` expression:

```
SELECT count(n2)
FROM t
```

If a `NOT NULL` constraint is defined on the `n2` column, count transformation translates the `count(n2)` in `count(*)` and produces the following query:

```
SELECT count(*)
FROM t
```

## Common Sub-Expression Elimination

The purpose of *common sub-expression elimination* is to remove duplicate predicates and thereby avoid processing the same operation several times. This is a heuristic-based query transformation.

In the following example, based on the `common_subexpr_elimination.sql` script, notice how the two disjunctive predicates are overlapping. In fact, all rows that fulfill the first one necessarily also fulfill the second one:

```
SELECT *
FROM t
WHERE (n1 = 1 AND n2 = 2) OR (n1 = 1)
```

Common sub-expression elimination removes the redundant predicate and produces the following query:

```
SELECT *
FROM t
WHERE n1 = 1
```

Are you surprised at the predicate that's left? If you think about it, you'll see that `n1 = 1` is enough to satisfy the query, whereas `n2 = 2` requires that `n1 = 1` also be looked at.

## Or Expansion

The purpose of *or expansion* is to transform a query with a `WHERE` clause containing disjunctive predicates into a compound query that uses one or several `UNION ALL` set operators. In general, each disjunctive predicate is transformed into a component query. This is a cost-based query transformation that is applied, most of the time, to enable additional index access paths. In fact, as Chapter 13 explains, disjunctive predicates and indexes don't always go well together. Also note that this query transformation supports function-based indexes as of version 11.2.0.2 only.

---

■ **Note** Even though or expansion is a cost-based query transformation, the query optimizer checks some heuristics before attempting it. If the query transformation is disallowed, an execution plan with a lower cost can be missed.

---

In the following example, based on the `or_expansion.sql` script, notice that the `WHERE` clause contains two disjunctive predicates. Because of them, the query optimizer evaluates whether the cost of a single access based on a table scan is higher than the cost of two distinct accesses based on index scans:

```
SELECT pad
FROM t
WHERE n1 = 1 OR n2 = 2
```

If the cost of the two index scans is lower, or expansion produces the following query. Notice that the `lnnvl(n1 = 1)` predicate is added to avoid duplicates. The `lnnvl` function returns `TRUE` when the condition passed as a parameter is `FALSE` or `NULL`. Hence, a row is returned by the second component query only when the first component query didn't already return it:

```
SELECT pad
FROM t
WHERE n1 = 1
UNION ALL
SELECT pad
FROM t
WHERE n2 = 2 AND lnnvl(n1 = 1)
```

Some disjunctive predicates can never be sensibly transformed with or expansions. The following query shows an example. All predicates reference the column named `n1`, enabling the `WHERE` clause to be processed like an `IN` condition:

```
SELECT *
FROM t
WHERE n1 = 1 OR n1 = 2 OR n1 = 3 OR n1 = 4
```

## View Merging

The purpose of *view merging* is to reduce the number of query blocks due to views and inline views by merging several of them together. This query transformation was introduced because, without it, the query optimizer would process each query block separately. When processing query blocks separately, the query optimizer can't always find an execution plan that is optimal for the SQL statement as a whole. In addition, the query block resulting from view merging might enable further query transformations to be considered.

### QUERY BLOCKS

Simply put, top-level SQL statements and each additional portion of a SQL statement with its own `SELECT` clause are query blocks. Simple SQL statements have a single query block. Multiple query blocks exist whenever views or constructs such as subqueries, inline views, and set operators are used. For example, the following query has two query blocks (I'm using the subquery factoring clause instead of defining a real view for illustration purposes only). The first query block is the top-level query, the one that references the `dept` table. The second query block is the query defined in the `WITH` clause, the one that references the `emp` table:

```
WITH emps AS (SELECT deptno, count(*) AS cnt
              FROM emp
              GROUP BY deptno)
SELECT dept.dname, emps.cnt
FROM dept, emps
WHERE dept.deptno = emps.deptno
```

View merging has two subcategories:

- **Simple view merging** is used for merging plain, select-project-join query blocks.[1] Because of the simplicity of the cases it handles, simple view merging is a heuristic-based query transformation. It can't be applied to views or inline views that contain constructs like aggregations, set operators, hierarchical queries, the `MODEL` clause, or subqueries in the `SELECT` list.
- **Complex view merging** is used for merging query blocks that contain aggregations. It is a cost-based query transformation that can't be applied to views or inline views that, for example, either appear in hierarchical queries or contain `GROUPING SETS`, `ROLLUP`, `PIVOT`, or `MODEL` clauses.

Note that complex view merging is a cost-based query transformation because applying it isn't always beneficial. In fact, when it's applied, the aggregation present in the merged view or inline view is postponed and, therefore, might be executed on a larger result set.

View merging can introduce security issues. To prevent them, the concept of *secure view merging*, which is controlled by the `optimizer_secure_view_merging` initialization parameter, is available. Chapter 9 describes this feature in detail.

[1]A select-project-join query block is made up of three basic operations: a selection that extracts rows fulfilling specific predicates, a projection that extracts specific columns from the referenced tables, and a join that puts together data extracted from several tables. Filter and join predicates are based on simple operators like equalities. Example: `SELECT t1.id, t2.n FROM t1 JOIN t2 ON t1.id = t2.id WHERE t1.n = 42`.

### Simple View Merging

In the following example, based on the simple_view_merging.sql script, the query is based on three query blocks: the top-level query and two inline views. Notice that the two inline views are plain, select-project-join query blocks:

```
SELECT *
FROM (SELECT t1.*
      FROM t1, t2
      WHERE t1.id = t2.t1_id) t12,
     (SELECT *
      FROM t3
      WHERE id > 6) t3
WHERE t12.id = t3.t1_id
```

Because the inline views can be merged, simple view merging produces the following query:

```
SELECT t1.*, t3.*
FROM t1, t2, t3
WHERE t1.id = t3.t1_id AND t1.id = t2.t1_id AND t3.id > 6
```

Simple view merging can't always be performed when outer joins are involved. For example, view merging *can* be performed on the previous query if the predicate of the top-level query is modified to t12.id = t3.t1_id(+), but *not* if the predicate is changed to t12.id(+) = t3.t1_id.

### Complex View Merging

The following example, based on the complex_view_merging.sql script, shows an inline view having a GROUP BY clause. Such a query is executed in the following manner: the table referenced in the inline view is accessed, the GROUP BY clause and the sum function are evaluated, and finally the result set of the inline view is joined to the table referenced in the top-level query:

```
SELECT t1.id, t1.n, t1.pad, t2.sum_n
FROM t1, (SELECT n, sum(n) AS sum_n
          FROM t2
          GROUP BY n) t2
WHERE t1.n = t2.n
```

When it's beneficial to postpone the evaluation of the GROUP BY clause until after the join, complex view merging generates the following query:

```
SELECT t1.id, t1.n, t1.pad, sum(n) AS sum_n
FROM t1, t2
WHERE t1.n = t2.n
GROUP BY t1.id, t1.n, t1.pad, t1.rowid, t2.n
```

## Select List Pruning

The purpose of *select list pruning* is to remove unnecessary columns or expressions from the SELECT clause of subqueries, inline views, or regular views. Top-level SELECT clauses aren't considered by this query transformation. Columns or expressions are considered unnecessary when they're not referenced outside the SELECT clause where they're referenced or defined. This is a heuristic-based query transformation.

In the following example, based on the `select_list_pruning.sql` script, notice how the subquery references two columns (`n2` and `n3`) that aren't referenced in the main query:

```
SELECT n1
FROM (SELECT n1, n2, n3
      FROM t)
```

Because the two columns `n2` and `n3` are unnecessary, select list pruning removes them and produces the following query:

```
SELECT n1
FROM (SELECT n1
      FROM t)
```

With view merging, the query can be simplified further. As a result, the following query is produced:

```
SELECT n1
FROM t
```

## Predicate Push Down

The purpose of *predicate push down* is to push predicates inside views or inline views that can't be merged. The predicates that can be pushed have to be contained in the query block that contains the unmergeable view or inline view. There are three main reasons for applying this query transformation:

- To enable additional access paths (typically index scans)
- To enable additional join methods and/or join orders
- To ensure that predicates are applied as soon as possible, thereby avoiding unnecessary processing

Predicate push down has two subcategories: *filter push down* and *join predicate push down*. The difference between the two is given by the type of predicates they work on.

### Filter Push Down

The purpose of *filter push down* is to push restrictions (filtering conditions) inside views or inline views that can't be merged. This is a heuristic-based query transformation. Be careful—this query transformation doesn't push join conditions. Pushing join conditions is done by the query transformation presented in the next section.

The following example is based on the `filter_push_down.sql` script. The `UNION` set operator prevents the inline view from being merged with the top-level query:

```
SELECT *
FROM (SELECT *
      FROM t1
      UNION
      SELECT *
      FROM t2)
WHERE id = 1
```

Filter push down pushes the restriction (`id = 1`) inside the inline view and produces the following query. Not only can the two tables now be accessed through an index, but it's also guaranteed that the sort operation required for the `UNION` set operator processes as few rows as possible:

```
SELECT *
FROM (SELECT *
      FROM t1
      WHERE id = 1
      UNION
      SELECT *
      FROM t2
      WHERE id = 1)
```

Simple view merging then eliminates the top-level query block as well.

## Join Predicate Push Down

The purpose of *join predicate push down* is to push join conditions inside views or inline views that can't be merged. This is a cost-based query transformation.

The following example is based on the `join_predicate_push_down.sql` script. The `UNION` set operator prevents the inline view from being merged with the top-level query. Notice that the inline view is the same (the tables have a different name, though) as the one used in the example in the preceding section. In this case, though, the inline view is joined to another table:

```
SELECT *
FROM t1, (SELECT *
          FROM t2
          UNION
          SELECT *
          FROM t3) t23
WHERE t1.id = t23.id
```

Join predicate push down pushes the join condition (`t1.id = t23.id`) inside the inline view and produces the following query. This query shares the same advantages as the one described in the previous section (index accesses enabled, and reduced amount of data to sort). In this specific case, the additional access paths also enable the query optimizer to freely choose between all available join methods and join orders:

```
SELECT *
FROM t1, (SELECT *
          FROM t2
          WHERE t2.id = t1.id
          UNION
          SELECT *
          FROM t3
          WHERE t3.id = t1.id) t23
```

Even though the previous SQL statement isn't valid (the `t1.id` column isn't visible inside the inline view), the SQL engine can process something similar to it. To support such a query, as of version 12.1 lateral inline views are available. For instance, the following is a valid query in version 12.1:

```
SELECT *
FROM t1, lateral(SELECT *
                 FROM t2
                 WHERE t2.id = t1.id
                 UNION
                 SELECT *
                 FROM t3
                 WHERE t3.id = t1.id) t23
```

## Predicate Move Around

The purpose of *predicate move around* is to pull up, move across, and push down restrictions (filtering conditions) inside views or inline views that can't be merged. Even though it's similar to predicate push down, this query transformation can also move predicates across query blocks that aren't contained in each other. The main reasons for applying this heuristic-based query transformation are enabling additional access paths (typically, index scans) and making sure that predicates are applied as soon as possible.

The two inline views in the following example, based on the `predicate_move_around.sql` script, can't be merged with the top-level query because of the `DISTINCT` operator. The first inline view contains a restriction on the column (`n`) used as join condition between the two inline views (`t1.n = t2.n`):

```
SELECT t1.pad, t2.pad
FROM (SELECT DISTINCT n, pad
      FROM t1
      WHERE n = 1) t1,
     (SELECT DISTINCT n, pad
      FROM t2) t2
WHERE t1.n = t2.n
```

In this case, predicate move around is performed in three main steps:

1. Pull up the restriction (`n = 1`) from the first inline view into the top-level query block.
2. Apply transitivity between the restriction (`t1.n = 1`) and the join condition (`t1.n = t2.n`) and generate a new predicate (`t2.n = 1`).
3. Push the new predicate inside the second inline view.

As a result, predicate move around generates the following query. The predicate added in the second inline view not only enables table `t2` access through an index, it also likely reduces the amount of data to be processed by the `DISTINCT` operator:

```
SELECT t1.pad, t2.pad
FROM (SELECT DISTINCT n, pad
      FROM t1
      WHERE n = 1) t1,
     (SELECT DISTINCT n, pad
      FROM t2
      WHERE n = 1) t2
WHERE t1.n = t2.n
```

## Distinct Placement

The purpose of *distinct placement* is to eliminate duplicates as soon as possible. This is a cost-based query transformation that's available as of version 11.2 only.

The DISTINCT operator eliminates duplicate rows from a result set. When it's specified in a query along with one or several joins, conceptually the database engine should process the DISTINCT operator *after* the joins have already been resolved. To achieve best performance, though, some cases call for eliminating duplicates before processing joins. Eliminating duplicates earlier keeps intermediate result sets as small as possible, and less processing is required for joins.

In the following example, based on the distinct_placement.sql script, there is a parent-child relationship between the two tables. Furthermore, you can presume that the child table (t2) contains many more rows than the parent table (t1):

```
SELECT DISTINCT t1.n, t2.n
FROM t1, t2
WHERE t1.id = t2.t1_id
```

When the number of distinct values of t2.n is much lower than the number of rows stored in the child table, distinct placement produces the following query. Notice the additional DISTINCT operator that, applied to the data of the child table, eliminates the duplicates before joining the parent table:

```
SELECT DISTINCT t1.n, vw_dtp.n
FROM t1, (SELECT DISTINCT t2.t1_id, t2.n
          FROM t2) vw_dtp
WHERE t1.id = vw_dtp.t1_id
```

## Distinct Elimination

The purpose of *distinct elimination*, which is available as of version 10.2.0.4, is to remove DISTINCT operators that aren't required to guarantee that the result set doesn't contain duplicates. This is a heuristic-based query transformation that can be applied when a SELECT clause references, without modifying them, all columns of a primary key, all columns of a unique key that's not nullable, or the rowid.

The following example is based on the distinct_elimination.sql script. Notice that the primary key of the table is defined on the column named id:

```
SELECT DISTINCT id, n
FROM t
```

The presence of the id column, which is the primary key of the table, in the SELECT clause guarantees the uniqueness of the rows. Hence, distinct elimination produces the following query:

```
SELECT id, n
FROM t
```

## Group-by Placement

The purpose of *group-by placement* is basically the same as that of *distinct placement*. The only obvious difference is the types of queries to which they're applied. Whereas the former is used for queries containing a GROUP BY clause, the latter is used for queries containing a DISTINCT operator. Group-by placement is a cost-based query transformation that's available as of version 11.1.

In the following example, based on the group_by_placement.sql script, there's a parent-child relationship between the two tables, and the child table (t2) contains many more rows than the parent table (t1):

```
SELECT t1.n, t2.n, count(*)
FROM t1, t2
WHERE t1.id = t2.t1_id
GROUP BY t1.n, t2.n
```

When the number of distinct values of t2.n is much lower than the number of rows stored in the child table, group-by placement produces the following query. An additional GROUP BY clause is applied to the data of the child table to eliminate duplicates before joining the parent table. Also notice that in the top-level SELECT clause the count function is replaced by the sum function:

```
SELECT t1.n, vw_gb.n, sum(vw_gb.cnt)
FROM t1, (SELECT t2.t1_id, t2.n, count(*) AS cnt
          FROM t2
          GROUP BY t2.t1_id, t2.n) vw_gb
WHERE t1.id = vw_gb.t1_id
GROUP BY t1.n, vw_gb.n
```

## Order-By Elimination

The purpose of *order-by elimination* is to remove superfluous ORDER BY clauses from subqueries, inline views, and regular views. Top-level SELECT clauses are obviously not considered by this heuristic-based query transformation. ORDER BY clauses can be considered unnecessary when an ORDER BY is followed by an operation that doesn't guarantee that it will return the rows ordered, or that it will return the rows in a different order; for example, another ORDER BY or an aggregation.

In the following example, based on the order_by_elimination.sql script, there's not only an ORDER BY in the inline view, but also a GROUP BY in the top-level query block:

```
SELECT n2, count(*)
FROM (SELECT n1, n2
      FROM t
      ORDER BY n1)
GROUP BY n2
```

Because the GROUP BY in the top-level query block doesn't guarantee the order of the returned rows, order-by elimination removes the ORDER BY and produces the following query:

```
SELECT n2, count(*)
FROM (SELECT n1, n2
      FROM t)
GROUP BY n2
```

The query optimizer, with select list pruning and simple view merging, further transforms the query into the following:

```
SELECT n2, count(*)
FROM t
GROUP BY n2
```

## Subquery Unnesting

The purpose of *subquery unnesting* is to inject semi- (IN, EXISTS), anti-join (NOT IN, NOT EXISTS), and scalar subqueries into the FROM clause of the containing query block, and to transform them into inline views. Some unnestings are performed as heuristic-based query transformations, and others are carried out as cost-based query transformations. The main reason for applying this query transformation is to enable all available join methods. In fact, without subquery unnesting, a subquery might have to be executed once for every row returned by the containing query block (Chapter 10 provides more information about this). Subquery unnesting can't always be applied, though. For example, unnesting isn't possible if a subquery contains some types of aggregation, or if it contains the rownum pseudocolumn. Semi- and anti-join subqueries containing set operators can only be unnested as of version 11.2. In addition, from version 12.1 onward, scalar subquery unnesting has been improved to process scalar subqueries in SELECT clauses.

The following example, based on the subquery_unnesting.sql script, illustrates how this query transformation works:

```
SELECT *
FROM t1
WHERE EXISTS (SELECT 1
              FROM t2
              WHERE t2.id = t1.id
              AND t2.pad IS NOT NULL)
```

Unnesting a subquery can be summarized in two steps. The first step, as shown in the following query, is to rewrite the subquery as an inline view. Note that what follows isn't a valid SQL statement, because the operator implementing the semi-join (s=) isn't available in the SQL syntax (it's used internally by the SQL engine only):

```
SELECT *
FROM t1, (SELECT id
          FROM t2
          WHERE pad IS NOT NULL) sq
WHERE t1.id s= sq.id
```

The second step, as shown here, is to rewrite the inline view as a regular join:

```
SELECT t1.*
FROM t1, t2
WHERE t1.id s= t2.id AND t2.pad IS NOT NULL
```

Although the preceding example is based on a semi-join, this query transformation is also used for anti-joins. The only difference is that instead of using the semi-join operator (s=), it uses the anti-join operator (a=). This is another operator that's used internally by the SQL engine only.

## Subquery Coalescing

The purpose of *subquery coalescing* is to combine equivalent semi- and anti-join subqueries into a single query block. The main reason for applying this heuristic-based query transformation, which is available as of version 11.2, is to reduce the number of table accesses, and thus to reduce the number of joins.

The following example, based on the subquery_coalescing.sql script, illustrates how this query transformation works. Notice that the two correlated subqueries process the same data. Only the restrictions differ:

```
SELECT *
FROM t1
WHERE EXISTS (SELECT 1
              FROM t2
              WHERE t2.id = t1.id AND t2.n > 10)
OR EXISTS (SELECT 1
           FROM t2
           WHERE t2.id = t1.id AND t2.n < 100)
```

Subquery coalescing combines the two subqueries and produces the following query:

```
SELECT *
FROM t1
WHERE EXISTS (SELECT 1
              FROM t2
              WHERE t2.id = t1.id AND (t2.n > 10 OR t2.n < 100))
```

With subquery unnesting, the query can be transformed further as follows. Notice that, as explained in the "Subquery Unnesting" section, this next query uses a special operator (s=) to implement the semi-join. That operator, available internally to the SQL engine, isn't part of the syntax that you can specify when writing a SQL statement:

```
SELECT t1.*
FROM t1, t2
WHERE t1.id s= t2.id AND (t2.n > 10 OR t2.n < 100)
```

## Subquery Removal Using Window Functions

The purpose of *subquery removal using window functions* is to replace subqueries containing aggregate functions with window functions. This heuristic-based query transformation can be applied when a query block contains all the tables and predicates appearing in a subquery.

The following example, based on the subquery_removal.sql script, illustrates how this query transformation works. Notice that the table t2 is referenced in the top-level query block as well as in the subquery:

```
SELECT t1.id, t1.n, t2.id, t2.n
FROM t1, t2
WHERE t1.id = t2.t1_id
AND t2.n = (SELECT max(n)
            FROM t2
            WHERE t2.t1_id = t1.id)
```

The query transformation removes the subquery and produces the following query. Notice how the CASE expression is used to generate a kind of flag by which to recognize the rows fulfilling the restriction that, in the original query, is specified by the subquery:

```
SELECT t1_id, t1_n, t2_id, t2_n
FROM (SELECT t1.id AS t1_id, t1.n AS t1_n, t2.id AS t2_id, t2.n AS t2_n,
             CASE t2.n
```

```
                WHEN max(t2.n) OVER (PARTITION BY t2.t1_id) THEN 1
             END AS max
      FROM t2, t1
      WHERE t1.id = t2.t1_id) vw_wif
WHERE max IS NOT NULL
```

## Join Elimination

The purpose of *join elimination* is to remove redundant joins—in other words, to completely avoid executing a join even if a SQL statement explicitly calls for it. The key information used by the query optimizer to decide whether it's sensible to implement this query transformation is the availability of a foreign key that's either enforced or marked RELY. In addition, as of version 11.2, self-joins based on the primary key are also considered. This heuristic-based query transformation is especially useful when views containing joins are used. Note, however, that join elimination can also be applied to SQL statements without views.

Let's take a look at an example based on the join_elimination.sql script. The following SQL statement defines a view. Note that between the two tables there is a parent-child relationship. In fact, table t2, with its column t1_id, references the primary key of table t1:

```
CREATE VIEW v AS
SELECT t1.id AS t1_id, t1.n AS t1_n, t2.id AS t2_id, t2.n AS t2_n
FROM t1, t2
WHERE t1.id = t2.t1_id
```

When a simple SELECT * FROM v is executed, simple view merging can be done, and the query is transformed as follows:

```
SELECT t1.id AS t1_id, t1.n AS t1_n, t2.id AS t2_id, t2.n AS t2_n
FROM t1, t2
WHERE t1.id = t2.t1_id
```

However, as illustrated in the next example, when only columns defined in the child tables are referenced (for example, SELECT t2_id, t2_n FROM v), the query optimizer is able to eliminate the join to the parent table. It can do so because there's a foreign key constraint that guarantees that all rows in table t2 reference one and only one row in table t1:

```
SELECT t2.id AS t2_id, t2.n AS t2_n
FROM t2
```

## Join Factorization

The purpose of *join factorization*, which is available as of version 11.2, is to recognize whether part of the processing of a compound query can be shared across component queries, with the goal being to avoid repetitive data accesses and joins. In fact, without this query transformation, all component queries would be executed idependently before applying the set operator. This is a cost-based query transformation that the query optimizer applies only to compound queries based on the UNION ALL set operator.

The following is an example based on the join_factorization.sql script. Notice that both component queries not only access the same tables, they also apply a restriction to the same table (t2). Without this query transformation, both component queries would be executed idependently, and both tables would be accessed twice:

```
SELECT *
FROM t1, t2
WHERE t1.id = t2.id AND t2.id < 10
UNION ALL
SELECT *
FROM t1, t2
WHERE t1.id = t2.id AND t2.id > 990
```

To avoid the repetitive processing of accessing each table twice, join factorization can transform the query, as shown in the following example. Because table t1 is factorized, it's accessed only once. Depending on the table's size and the access path chosen to extract data from it, the savings in term of I/O and CPU utilization can be huge:

```
SELECT t1.*, vw_jf.*
FROM t1, (SELECT *
          FROM t2
          WHERE id < 10
          UNION ALL
          SELECT *
          FROM t2
          WHERE id > 990) vw_jf
WHERE t1.id = vw_jf.id
```

## Outer Join to Inner Join

The purpose of *outer join to inner join* is to convert superfluous outer joins into inner joins. This is done because outer joins might prevent the query optimizer from choosing a specific join method or order. This is a heuristic-based query transformation.

The following example, based on the outer_to_inner.sql script, illustrates this query transformation. Notice that the restriction (t2.id IS NOT NULL) conflicts with the outer join condition (t1.id = t2.t1_id(+)):

```
SELECT *
FROM t1, t2
WHERE t1.id = t2.t1_id(+) AND t2.id IS NOT NULL
```

The query transformation removes the outer join operator as well as the redundant predicate and produces the following query:

```
SELECT *
FROM t1, t2
WHERE t1.id = t2.t1_id
```

## Full Outer Join

The purpose of *full outer join*, which is a heuristic-based query transformation, is to translate a full outer join into a compound query that uses the UNION ALL set operator to combine the rows returned by two joins: an outer join and an anti-join. In addition, if the predicate specified in the ON clause references not-nullable columns that define a foreign key constraint which is either enforced or marked RELY, this query transformation can even translate the full outer join into a query that at runtime will be executed as a left outer join.

---

■ **Note** Even though the full outer join syntax is available as of version 9.0, prior to version 11.1 the SQL engine wasn't able to natively execute a full outer join. As a result, queries containing full outer joins are translated into something that the SQL engine can work on. As of version 11.1, with the introduction of native full outer joins, this is no longer the case.

---

The following example, based on the full_outer_join.sql script, illustrates how this query transformation works. Notice that the query uses the FULL OUTER JOIN syntax in the FROM clause:

```
SELECT *
FROM t1 FULL OUTER JOIN t2 ON t1.n = t2.n
```

The query transformation produces the following query. Notice that only the first of the two joins defined in the inline view is an outer join. The second one contains, as explained earlier in the "Subquery Unnesting" section, a special operator (a=) to implement the anti-join:

```
SELECT id1 AS id, n1 AS n, pad1 AS pad, id, t1_id, n, pad
FROM (SELECT t1.id AS id1, t1.n AS n1, t1.pad AS pad1, t2.id, t2.t1_id, t2.n, t2.pad
      FROM t1, t2
      WHERE t1.n = t2.n(+)
      UNION ALL
      SELECT NULL, NULL, NULL, t2.id, t2.t1_id, t2.n, t2.pad
      FROM t1, t2
      WHERE t1.n a= t2.n) vw_foj
```

## Table Expansion

The purpose of *table expansion*, which is available as of version 11.2, is to enable the use of as many index scans as possible by also leveraging partially unusable indexes. The essential thing to recognize is that this cost-based query transformation is only considered when three basic conditions are fulfilled:

- A partitioned table is involved.
- The partitioned table has a local index having unusable partitions or, as of version 12.1, a partial index.
- The SQL statement to be optimized has to process data underlying both usable and unusable index partitions.

Prior to version 11.2 in such a situation, an index scan based on the partially unusable index isn't possible, and the whole index is completely ignored. Instead, a full table scan has to be used. But with table expansion, the query optimizer can take advantage of usable index partitions when they exist and fall back to a full partition scan for unusable index partitions.

In the following example, based on the `table_expansion.sql` script, there's a range-partitioned table with one partition per quarter of 2014. Notice that the local index it creates is unusable. Later, a usable index partition is built for a single table partition only:

```
CREATE TABLE t (
  id NUMBER PRIMARY KEY,
  d DATE NOT NULL,
  n NUMBER NOT NULL,
  pad VARCHAR2(4000) NOT NULL
)
PARTITION BY RANGE (d) (
  PARTITION t_q1_2014 VALUES LESS THAN (to_date('2014-04-01','yyyy-mm-dd')),
  PARTITION t_q2_2014 VALUES LESS THAN (to_date('2014-07-01','yyyy-mm-dd')),
  PARTITION t_q3_2014 VALUES LESS THAN (to_date('2014-10-01','yyyy-mm-dd')),
  PARTITION t_q4_2014 VALUES LESS THAN (to_date('2015-01-01','yyyy-mm-dd'))
);

CREATE INDEX i ON t (n) LOCAL UNUSABLE;

ALTER INDEX i REBUILD PARTITION t_q4_2014;
```

Prior to version 11.2, an index scan is completely avoided when a query such as the following is optimized. Even though the restriction is based on an indexed column, not all required index partitions are usable. Thus, even for partitions that have a usable index, no index scan is performed:

```
SELECT *
FROM t
WHERE n = 8
```

As of version 11.2, to enable the use of the usable index partitions, the query is transformed into a compound query in which one component query accesses the partitions with the usable index, and the other component query accesses the partitions with the unusable index. Notice how this is achieved by adding to both component queries a predicate based on the partition key:

```
SELECT *
FROM (SELECT *
      FROM t
      WHERE n = 8
      AND d < to_date('2014-10-01','yyyy-mm-dd')
      UNION ALL
      SELECT *
      FROM t
      WHERE n = 8
      AND d >= to_date('2014-10-01','yyyy-mm-dd')
      AND d < to_date('2015-01-01','yyyy-mm-dd')) vw_te
```

## Set to Join Conversion

The purpose of *set to join conversion* is to avoid sort operations in compound queries involving `INTERSECT` and `MINUS`. This query transformation also postpones the elimination of duplicates to the end of processing for such queries.

A compound query based on the INTERSECT and MINUS set operators is basically carried out in the following way:

1. Every component query is independently executed, the result set is sorted, and duplicates are eliminated.
2. Then the set operations are executed, and the final result set is determined.

This way of executing a query involving an INTERSECT or MINUS operation isn't always efficient. For example, when the component queries return a lot of data, but the majority of data is eliminated by the set operator, most of the data that's later eliminated ends up being unnecessarily sorted. The *set to join conversion* avoids that inefficiency by transforming the query in a way that allows rows to be thrown out *prior* to the sort rather than after it. In addition, since the set operator is replaced by a join, additional access paths are enabled. This is a heuristic-based query transformation that, by default, isn't enabled.[2] The set_to_join hint has to be specified to take advantage of it.

The following is an example based on the set_to_join.sql script. The compound query is based on the INTERSECT set operator:

```
SELECT *
FROM t1
WHERE n > 500
INTERSECT
SELECT *
FROM t2
WHERE t2.pad LIKE 'A%'
```

The query transformation converts the set operator to a join. Furthermore, to ensure that only the required rows are returned, the query transformation also adds several predicates and a DISTINCT operator. Here is the final query after set to join conversion is complete:

```
SELECT DISTINCT t1.*
FROM t1, t2
WHERE t1.id = t2.id AND t1.n = t2.n AND t1.pad = t2.pad
AND t1.n > 500 AND t1.pad LIKE 'A%'
AND t2.n > 500 AND t2.pad LIKE 'A%'
```

## Star Transformation

*Star transformation* is a cost-based query transformation used for queries extracting data from a star schema. Chapter 14 provides detailed information about star schemas and optimizing the joins of queries against such schemas.

## Query Rewrite with Materialized Views

*Query rewrite with materialized views* is an optimization technique allowing the database engine to access data stored in materialized views even when no materialized view is directly referenced in the query to be optimized. Chapter 15 discusses this type of query transformation in detail.

[2]By default the _convert_set_to_join initialization parameter is set to FALSE.

## On to Chapter 7

This chapter describes fundamental information about the query optimizer. You've read about the inputs the query optimizer uses to generate execution plans. You've also learned about the key components that compose the SQL engine and how they interact with each other. And you've learned about query transformations and how they're applied to increase the chances of finding the best possible execution plan for a given SQL statement.

System statistics is one of the inputs introduced in this chapter. Their purpose is to describe both the system running the database engine and the runtime behavior of the storage subsystem. Chapter 7 describes in detail what system statistics are, how to manage them, and how the query optimizer uses them to improve its estimations.

CHAPTER 7

# System Statistics

The query optimizer used to base its cost estimations on the number of physical reads needed to execute SQL statements. That method is known as the *I/O cost model*. The main drawback of this method is that single-block reads and multiblock reads are equally costly.[1] Consequently, multiblock read operations, such as full table scans, are artificially favored. Before system statistics were introduced, especially in OLTP systems, the `optimizer_index_caching` and `optimizer_index_cost_adj` initialization parameters were used to work around this problem (see Chapter 9 for coverage of both parameters). In fact, the default values used to be appropriate for reporting systems and data warehouses only. Currently, a new costing method, known as the *CPU cost model*, is used to address this flaw. To use the CPU cost model, additional information about the performance of the system where the database engine runs, called *system statistics*, has to be provided to the query optimizer. Essentially, system statistics supply the following information:

- Performance of the disk I/O subsystem
- Performance of the CPU

Despite its name, the CPU cost model takes into consideration the cost of physical reads as well. But instead of basing the I/O costs on the number of physical reads only, the performance of the disk I/O subsystem is also considered. Don't let the name mislead you.

A default set of system statistics is always available. As a result, by default, the CPU cost model is used. Actually, the only way to use the I/O cost model is to specify the `no_cpu_costing` hint at the SQL statement level or by setting an undocumented initialization parameter. In all other cases, the query optimizer uses the CPU cost model.

## The dbms_stats Package

The `dbms_stats` package provides a comprehensive set of procedures to manage system statistics. By default, the package modifies the data dictionary. However, it's possible with most of the package procedures to work instead on a user-defined table stored outside the data dictionary. Such a table is what I call the *backup table*. This table is mainly used in two situations:

- For gathering system statistics without having to store them in the data dictionary and, as a result, without immediately making them available to the query optimizer
- For moving system statistics between two databases

Moving statistics between databases is done by exporting the system statistics into a backup table on the source, moving the backup table to another database, and importing its content into the target data dictionary. Moving system statistics between databases is a way to make sure that all databases related to a given application (for example, development, test and production) are using the same system statistics.

[1]Common sense suggests that reading a single block should be faster than reading multiple blocks. Strangely enough, this isn't always true in reality. In any case, the important thing is to recognize that there's a difference.

---

■ **Note** The gathering of system statistics doesn't invalidate the cursors stored in the library cache. As a result, the new system statistics will only be used for SQL statements that will need to be hard parsed.

---

The dbms_stats package provides the following subprograms (see Figure 7-1):

- gather_system_stats gathers system statistics and stores them in either the data dictionary or in a backup table.
- delete_system_stats deletes system statistics stored in either the data dictionary or a backup table.
- restore_system_stats restores system statistics to the data dictionary.
- export_system_stats moves system statistics from the data dictionary to a backup table.
- import_system_stats moves system statistics from a backup table to the data dictionary.
- get_system_stats extracts system statistics stored in either the data dictionary or a backup table.
- set_system_stats modifies system statistics stored in either the data dictionary or a backup table.

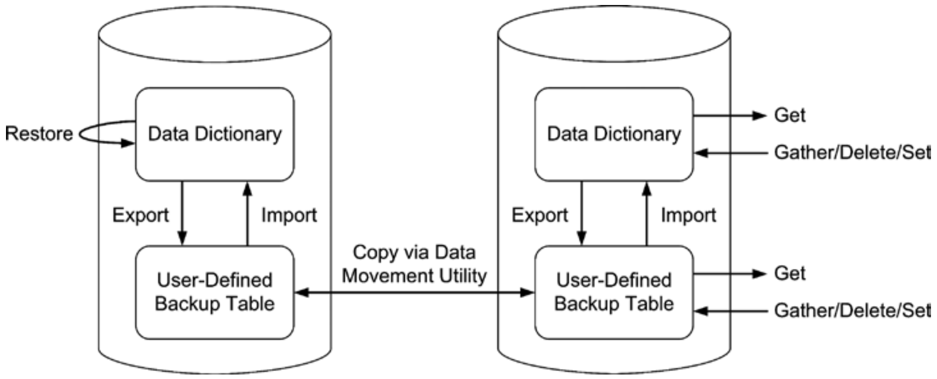


***Figure 7-1.*** *The dbms_stats package provides a comprehensive set of features to manage system statistics*

By default, permission to execute the dbms_stats package is granted to public. As a result, every user can gather system statistics. Nevertheless, only those holding the privileges provided by the gather_system_statistics role are able to change the system statistics stored in the data dictionary. Unprivileged users can only store them in a backup table. Per default, the gather_system_statistics role is provided through the dba role.

## What System Statistics Are Available?

There are two kinds of system statistics: *noworkload statistics* and *workload statistics*. The main difference between the two is the method used to measure the performance of the disk I/O subsystem. Whereas the former runs a synthetic benchmark, the latter uses an application benchmark. In both cases the performance of the CPU is computed with a synthetic benchmark. Before discussing in detail the difference between these two approaches, let's see how system statistics are stored in the data dictionary.

## APPLICATION VS. SYNTHETIC BENCHMARK

An *application benchmark*, also called a *real benchmark*, is based on the workload produced by the normal operation of a real application. Although it usually provides very good information about the real performance of the system running it, because of its nature, applying it in a controlled manner isn't always possible.

A *synthetic benchmark* is a workload produced by a program that does no real work. The main idea is that it should simulate (model) an application workload by executing similar operations. Although it can be easily applied in a controlled manner, usually it doesn't produce performance figures that are as good as an application benchmark. Nevertheless, it could be useful for comparing different systems.

System statistics are stored in the aux_stats$ data dictionary table. Unfortunately, no data dictionary view is available to externalize them. In this table, up to three sets of rows are differentiated by the following values of the sname column:

- SYSSTATS_INFO is the set containing the status of system statistics and when they were gathered. If they're correctly gathered, STATUS is set to COMPLETED. If there's a problem during the gathering of statistics, STATUS is set to BADSTATS, in which case the system statistics aren't used by the query optimizer. Two more values may be seen during the gathering of workload statistics: MANUALGATHERING and AUTOGATHERING. The attribute FLAGS takes the following values: 0 if the system statistics were set to the default values by calling the delete_system_stats procedure; 1 if the system statistics were regularly gathered or set; 128 if the system statistics were restored by calling the restore_system_stats procedure.

```
SQL> SELECT pname, pval1, pval2
  2  FROM sys.aux_stats$
  3  WHERE sname = 'SYSSTATS_INFO';

PNAME    PVAL1 PVAL2
------- ------ -----------------
DSTART         10-25-2013 23:26
DSTOP          10-25-2013 23:28
FLAGS        1
STATUS         COMPLETED
```

- SYSSTATS_MAIN is the set containing the system statistics themselves. Detailed information about them is provided in the next section.

```
SQL> SELECT pname, pval1
  2  FROM sys.aux_stats$
  3  WHERE sname = 'SYSSTATS_MAIN';

PNAME              PVAL1
----------- ------------
CPUSPEEDNW        1991.0
IOSEEKTIM           10.0
IOTFRSPEED        4096.0
SREADTIM             1.6
MREADTIM             7.8
```

```
CPUSPEED          1992.0
MBRC                21.0
MAXTHR       659158016.0
SLAVETHR      34201600.0
```

- `SYSSTATS_TEMP` is the set containing values used for the computation of system statistics. It's available only while gathering workload statistics.

## Gathering System Statistics

As just mentioned, the database engine supports two types of system statistics: noworkload statistics and workload statistics. This section describes not only how they're gathered, but also what information they provide to the query optimizer and how to decide whether noworkload statistics or workload statistics should be used.

Because a single set of statistics exists for a single database, all instances of a RAC system use the same system statistics. Therefore, if the nodes aren't equally sized or loaded, you must carefully decide which node the system statistics are to be gathered on.

### Noworkload Statistics

Noworkload statistics are always available. If you explicitly delete them, they're automatically gathered during the next database start-up. You can gather noworkload statistics on an idle system because the database engine uses a synthetic benchmark to generate the load used to measure the performance of the system. To measure the CPU speed, most likely some kind of calibrating operation is executed in a loop. To measure the disk I/O performance, some reads of different sizes are performed on several data files of the database.

To gather noworkload statistics, you set the `gathering_mode` parameter of the `gather_system_stats` procedure to `noworkload`, as shown in this example:

```
dbms_stats.gather_system_stats(gathering_mode => 'noworkload')
```

In addition, to better support systems having high disk I/O throughput (like Exadata), from version 11.2.0.4 onward (or when a patch implementing the enhancement associated to bug 10248538 is installed) there is another method for gathering noworkload statistics: by setting the `gathering_mode` parameter to `exadata`, as shown here:

```
dbms_stats.gather_system_stats(gathering_mode => 'exadata')
```

In both cases, the gathering usually takes a few minutes, and the statistics listed in Table 7-1 are computed. Oddly, sometimes it's necessary to repeat the gathering of statistics more than once; otherwise, the default values, which are also available in Table 7-1, are used. That is because the measured statistics must pass a sanity check before being stored. If they don't pass, they're discarded and replaced with the default statistics. Unfortunately, no information is provided to you when that happens.

***Table 7-1.*** *Noworkload Statistics Stored in the Data Dictionary*

| Name | Description |
|---|---|
| CPUSPEEDNW | The number of operations per second (in millions) that one CPU is able to process. There's no default value because CPUSPEEDNW is always based on the result of the synthetic benchmark used for assessing the speed of the CPU. |
| IOSEEKTIM | Average time (in milliseconds) needed to locate data on disk. The default value is 10. |
| IOTFRSPEED | Average number of bytes per millisecond that can be transferred from disk. The default value is 4,096. |
| MBRC | Number of blocks read during multiblock read operations. This statistic is set in exadata mode only (in other words, when gathering_mode is set to exadata). There is no default value because MBRC is always set to the value of the db_file_multiblock_read_count initialization parameter. |

The only difference between the regular and the exadata noworkload statistics is that in the latter the mbrc statistic is also set. Specifically, it's set to the value of the db_file_multiblock_read_count initialization parameter. The aim is to inform the query optimizer that the database engine can efficiently perform large disk I/O operations and, therefore, can reduce the cost of full scans. Using the exadata gathering mode is essential only when the db_file_multiblock_read_count initialization parameter isn't explicitly set. When it isn't set, the query optimizer uses a value of 8 to cost full scans (this behaviour is described in Chapter 9). With such a value, the cost of full scans is usually much higher than it should be. When using the exadata mode, the situation is completely different. In fact, mbrc is set to 128 on most systems—therefore the cost of full scans is much lower. It's especially advisable to use the exadata mode on systems with high disk I/O throughput (like Exadata).

## Workload Statistics

Workload statistics are available only when explicitly gathered. To gather them, you can't use an idle system because the database engine has to take advantage of the regular database load to measure the performance of the disk I/O subsystem. On the other hand, the method for noworkload statistics is used to measure the speed of the CPU. As shown in Figure 7-2, gathering workload statistics is a three-step activity. The idea is that to compute the average time taken by an operation, it's necessary to know how many times that operation was performed and how much time was spent executing it. For example, with the following SQL statements, I was able to compute the average time for single-block reads (6.2 milliseconds) from one of my test databases, in the same way the dbms_stats package would:

```
SQL> SELECT sum(singleblkrds) AS count, sum(singleblkrdtim)*10 AS time_ms
  2  FROM v$filestat;

     COUNT    TIME_MS
---------- ----------
     22893      36760

SQL> REMARK run a benchmark to generate some disk I/O operations...

SQL> SELECT sum(singleblkrds) AS count, sum(singleblkrdtim)*10 AS time_ms
  2  FROM v$filestat;

     COUNT    TIME_MS
---------- ----------
     54956     236430
```

```
SQL> SELECT round((236430-36760)/(54956-22893),1) AS avg_tim_singleblkrd
  2  FROM dual;

AVG_TIM_SINGLEBLKRD
-------------------
                6.2
```

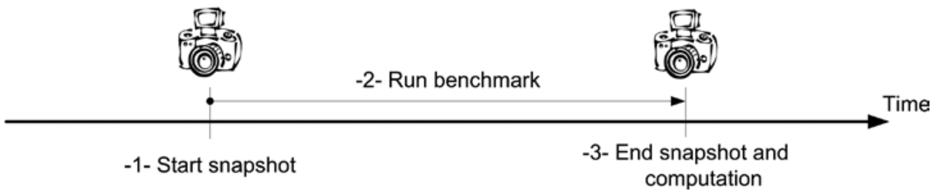


***Figure 7-2.*** *To gather (compute) system statistics, two snapshots of several performance figures are used*

The three steps illustrated in Figure 7-2 are as follows:

1. A snapshot of several performance figures is taken and stored in the aux_stats$ data dictionary table (for these rows, the sname column is set to SYSSTATS_TEMP). This step is carried out by setting the gathering_mode parameter of the gather_system_stats procedure to start, as shown in the following command:

   ```
   dbms_stats.gather_system_stats(gathering_mode => 'start')
   ```

2. The database engine doesn't control the database load. Consequently, enough time to cover a representative load has to elapse before taking another snapshot. It's difficult to provide general advice about this waiting time, but it's common to wait at least 5–10 minutes.
3. A second snapshot is taken. This step is carried out by setting the gathering_mode parameter of the gather_system_stats procedure to stop, as shown in the following command:

   ```
   dbms_stats.gather_system_stats(gathering_mode => 'stop')
   ```

4. The system statistics listed in Table 7-2 are computed, based on the performance statistics of the two snapshots. If one of the disk I/O statistics can't be computed, that statistic is set to NULL. Inability to compute a statistic can occur if the workload did not use either single-block reads, multiblock reads or parallel processing. For example, if the workload did not perform any multiblock reads, mbrc and mreadtim are set to NULL.

***Table 7-2.*** *Workload Statistics Stored in the Data Dictionary*

| Name | Description |
|---|---|
| CPUSPEED | The number of operations per second (in millions) that one CPU is able to process |
| SREADTIM | Average time (in milliseconds) needed to perform a single-block read operation |
| MREADTIM | Average time (in milliseconds) needed to perform a multiblock read operation |
| MBRC | Average number of blocks read during multiblock read operations |
| MAXTHR | Maximum disk I/O throughput (in bytes per second) for the whole system |
| SLAVETHR | Average disk I/O throughput (in bytes per second) for a single parallel processing slave |

To avoid manually taking the ending snapshot, it's also possible to set the gathering_mode parameter of the gather_system_stats procedure to interval. With this parameter, the starting snapshot is immediately taken, and the ending snapshot is scheduled to be executed after the number of minutes specified by a second parameter named interval. The following command specifies that the gathering of statistics should last 10 minutes:

```
dbms_stats.gather_system_stats(gathering_mode => 'interval',
                               interval       => 10)
```

Note that the execution of the preceding command doesn't take 10 minutes. It just takes the starting snapshot and schedules a job to take the ending snapshot. You can see the job by querying, for example, the user_scheduler_jobs view.

---

■ **Caution** Because of bug 9842771, workload system statistics, specifically the values of sreadtim and mreadtim, are broken in version 11.2.0.1 and version 11.2.0.2. To fix the problem, you can install patch 9842771. If you can't install that patch, then as a workaround you can set the value of sreadtim and mreadtim manually (the code example later in this section shows how to set those values).

---

The main problem in gathering workload statistics is choosing the gathering period. In fact, most systems experience a load that is anything but constant, and therefore, the evolution of workload statistics, except for cpuspeed, is equally inconstant. Figure 7-3 shows the evolution of workload statistics that I measured on a production system. To produce the charts, I gathered workload statistics for about four days at intervals of one hour. Consult the system_stats_history.sql and system_stats_history_job.sql scripts for examples of the SQL statements I used for that purpose.

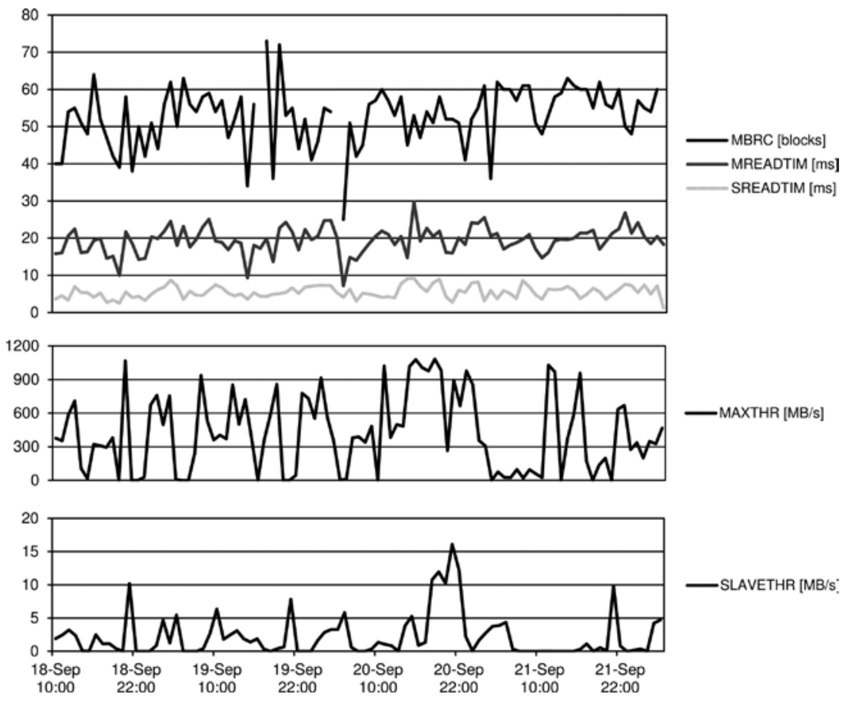


***Figure 7-3.*** *On most systems, the evolution of workload statistics is anything but constant*

To avoid gathering workload statistics during a period that provides values that are unrepresentative of the load, I see only two approaches. Either gather workload statistics over a period of several days or produce charts based on much shorter periods (for example, 10 minutes), as in Figure 7-3, to get values that make sense. I usually advise the latter because averages computed over several days may be very misleading when the workload changes considerably over that same period of time. In addition, with shorter periods, you also get a useful view of the system performance at the same time.

Another advantage of using the approach based on short intervals gathered for several days or weeks is that it forces you to *not* immediately change the system statistics in the data dictionary. In fact, when gathering system statistics, it's much better to gather them in a backup table and check them for consistency. Then, if they're okay, import them into the data dictionary.

For example, based on the charts shown in Figure 7-3, I suggest using the average values for `mbrc`, `mreadtim`, and `sreadtim` and using the maximum values for `maxthr` and `slavethr`. Then a PL/SQL block like the following one might be used to manually set the workload statistics. Note that before setting the workload statistics with the `set_system_stats` procedure, the old set of system statistics is deleted with the `delete_system_stats` procedure:

```
BEGIN
  dbms_stats.delete_system_stats();
  dbms_stats.set_system_stats(pname => 'CPUSPEED', pvalue => 772);
  dbms_stats.set_system_stats(pname => 'SREADTIM', pvalue => 5.5);
  dbms_stats.set_system_stats(pname => 'MREADTIM', pvalue => 19.4);
  dbms_stats.set_system_stats(pname => 'MBRC',     pvalue => 53);
  dbms_stats.set_system_stats(pname => 'MAXTHR',   pvalue => 1136136192);
  dbms_stats.set_system_stats(pname => 'SLAVETHR', pvalue => 16870400);
END;
```

This method of manually setting system statistics could also be used if different sets of workload statistics are needed for different periods of the day or week. It must be said, however, that I have never come across a case that required more than one set of workload statistics.

## Choosing Between Noworkload Statistics and Workload Statistics

Choosing between the two types of available system statistics is about choosing between simplicity and control. If simplicity is key, you may want to choose noworkload statistics. This is because, as described in the previous sections, noworkload statistics are much easier to gather.

---

■ **Note** The absolute simplest approach is to choose the default statistics, which you can do with a call to `delete_system_stats`. For some databases, these default statistics may be all you need.

---

However, by choosing the simple approach of using noworkload statistics, you lose control of two specific features:

- When using noworkload statistics, the value of the `db_file_multiblock_read_count` initialization parameter may impact some of the estimations performed by the query optimizer. As described in Chapter 9, this is suboptimal. With workload statistics, the role played by that parameter is replaced by the `mbrc` statistic.
- Only with workload statistics, through the `maxthr` and `slavethr` statistics, can you control the costing of parallel operations.

These two features are available only with workload system statistics. For this reason, I consider workload statistics to be superior and usually recommend them over the noworkload option. The additional effort you put into gathering workload statistics usually pays off in the long term.

## Restoring System Statistics

Whenever system statistics are changed through the `dbms_stats` package, instead of simply overwriting current statistics with the new statistics, the current statistics are saved in another data dictionary table (`wri$_optstat_aux_history`) that keeps a history of all changes occurring within a retention period. The purpose is to be able to restore old statistics in case new statistics lead to inefficient execution plans.

For purpose of restoring old statistics, the `dbms_stats` package provides the `restore_system_stats` procedure. This procedure accepts a single parameter: a timestamp specifying the target time. Statistics are restored to those that were in use at that specific time. For example, the following PL/SQL block restores the system statistics that were in use one day ago.

```
BEGIN
  dbms_stats.delete_system_stats();
  dbms_stats.restore_system_stats(as_of_timestamp => systimestamp - INTERVAL '1' DAY);
END;
```

---

■ **Caution** To make sure you restore exactly the same system statistics that were in use at a specific time, you have to delete the current system statistics before the restore. Otherwise, the statistics being restored are in fact *merged* with whatever statistics are current.

---

System statistics (as well as object statistics, because they're maintained by the same underlying functionality) are kept in the history for an interval specified by a retention period. The default value is 31 days. You can display the current value by calling the `get_stats_history_retention` function in the `dbms_stats` package, as shown here:

```
SELECT dbms_stats.get_stats_history_retention() AS retention FROM dual
```

To change the retention period, the `dbms_stats` package provides the `alter_stats_history_retention` procedure. Here's an example where the call sets the retention period to 14 days:

```
dbms_stats.alter_stats_history_retention(retention => 14)
```

Note that with the `alter_stats_history_retention` procedure, the following values have a special meaning:

- `NULL` sets the retention period to the default value.
- 0 disables the history.
- –1 disables the purging of the history.

When the `statistics_level` initialization parameter is set to `typical` (the default value) or `all`, statistics older than the retention period are automatically purged. Whenever manual purging is necessary, the `dbms_stats` package provides the `purge_stats` procedure. The following call purges all statistics placed in the history more than 14 days ago:

```
dbms_stats.purge_stats(before_timestamp => systimestamp - INTERVAL '14' DAY)
```

To execute the alter_stats_history_retention and purge_stats procedures, you need to have the analyze any and analyze any dictionary system privileges.

## Working with a Backup Table

Most dbms_stats procedures used for managing system statistics are able to work with either the data dictionary or a backup table. But there's one procedure limited to working only with the data dictionary: the restore_system_stats procedure.

While all operations are performed against the data dictionary by default, procedures supporting a backup table provide three parameters for you to use if you want to work with a backup table instead. The three parameters are as follows:

> stattab specifies the name of a table outside the data dictionary where the statistics are stored. The default value is NULL.
>
> statown specifies the owner of the table specified with the stattab parameter. The default value is NULL, and therefore the current user is used.
>
> statid is an optional identifier used to recognize multiple sets of statistics stored in the backup table, specified with the stattab and statown parameters. Only valid Oracle identifiers[2] are supported.

For example, the following call gathers noworkload statistics and stores them in the backup table named mystats, which is owned by the system user:

```
dbms_stats.gather_system_stats(gathering_mode => 'noworkload',
                               statown        => 'system',
                               stattab        => 'mystats')
```

To create a backup table, invoke the create_stat_table procedure in the dbms_stats package. Creation is a matter of specifying the owner (with the ownname parameter) and the name (with the stattab parameter) of the backup table. In addition, the optional tblspace parameter specifies the tablespace in which the table is created. If the tblspace parameter isn't specified, the table ends up in the default tablespace of the user. Following is an example:

```
dbms_stats.create_stat_table(ownname  => user,
                             stattab  => 'mystats',
                             tblspace => 'users')
```

The dbms_stats package provides the drop_stat_table procedure to drop a backup table. You can also drop a backup table with a regular DROP TABLE statement. For example:

```
dbms_stats.drop_stat_table(ownname => user,
                           stattab => 'mystats')
```

## Logging of Management Operations

All dbms_stats procedures used for managing system statistics, except for restore_system_stats, log some information about their activities into the data dictionary. This information is externalized through the dba_optstat_operations view and, from version 12.1 onward, also in the dba_optstat_operation_tasks view. Note that in a multitenant environment,

[2]Refer to the *SQL Language Reference* manual in the Oracle documentation for the definition of an identifier.

the cdb views are also available. The following excerpt of the output generated by the `system_stats_logging.sql` script shows an example of querying the view. By querying the view, you can find out to know which operations were performed, when they were started, and how long they took:

```
SQL> VARIABLE now VARCHAR2(14)

SQL> BEGIN
  2    SELECT to_char(sysdate,'YYYYMMDDHH24MISS') INTO :now FROM dual;
  3    dbms_stats.delete_system_stats();
  4    dbms_stats.gather_system_stats('noworkload');
  5  END;
  6  /

SQL> SELECT operation, start_time,
  2         (end_time-start_time) DAY(1) TO SECOND(0) AS duration
  3  FROM dba_optstat_operations
  4  WHERE start_time > to_date(:now,'YYYYMMDDHH24MISS')
  5  ORDER BY start_time;

OPERATION            START_TIME                        DURATION
-------------------- --------------------------------- -----------
delete_system_stats  25-SEP-13 16.59.47.679829 +02:00  +0 00:00:00
gather_system_stats  25-SEP-13 16.59.47.688208 +02:00  +0 00:00:02
```

In addition, as of version 12.1, you can see the parameters with which an operation was executed. The following query illustrates this:

```
SQL> SELECT x.*
  2  FROM dba_optstat_operations o,
  3       XMLTable('/params/param'
  4                PASSING XMLType(notes)
  5                COLUMNS name VARCHAR2(20) PATH '@name',
  6                        value VARCHAR2(20) PATH '@val') x
  7  WHERE start_time > to_date(:now,'YYYYMMDDHH24MISS')
  8  AND operation = 'gather_system_stats';

NAME                 VALUE
-------------------- ----------
gathering_mode       noworkload
interval             60
statid
statown
stattab
```

In version 12.1 it's also possible to extract details about a specific operation through the `report_single_stats_operation` function of the `dbms_stats` package. Different formats (text, HTML, and XML) are supported as output. The following query illustrates how to generate a text report:

```
SQL> SELECT dbms_stats.report_single_stats_operation(opid         => id,
  2                                                  detail_level => 'all',
  3                                                  format       => 'text')
  4  FROM dba_optstat_operations
  5  WHERE operation = 'gather_system_stats'
  6  AND start_time > to_date(:now,'YYYYMMDDHH24MISS');

------------------------------------------------------------------------------------------
| Operation | Operation           | Start Time       | End Time         | Additional Info  |
| Id        |                     |                  |                  |                  |
------------------------------------------------------------------------------------------
| 4928      | gather_system_stats | 25-SEP-13        | 25-SEP-13        | Parameters:      |
|           |                     | 16.28.35.528238  | 16.28.37.105673  | [gathering_mode: |
|           |                     | +02:00           | +02:00           | noworkload]      |
|           |                     |                  |                  | [interval: 60]   |
|           |                     |                  |                  | [statid: ]       |
|           |                     |                  |                  | [statown: ]      |
|           |                     |                  |                  | [stattab: ]      |
------------------------------------------------------------------------------------------
```

Be aware that log information is purged by the same mechanism as the statistics history described earlier. Both, therefore, have the same retention period.

## Impact on the Query Optimizer

System statistics have a direct impact on the costs estimated by the query optimizer. Most of the statistics, when available, are always used. Some, however, are used by the query optimizer only for estimating the costs of particular execution plans. Specifically, `mbrc` is used only when multiblock reads are involved; `maxthr` and `slavethr` are used only for SQL statements that are considered to be executed in parallel.

This section illustrates some usages. Others are covered in Chapter 9 in the discussion of how the query optimizer estimates the cost of full table scans.

---

■ **Caution** The formulas provided in this section, with a single exception, aren't published by Oracle. Several tests show that they're able to describe how the query optimizer estimates the cost of a given operation. In any case, they can't be claimed to be precise or correct in all situations. They're provided here to give you an idea of how system statistics influence query optimizer estimations.

System statistics as described in this chapter are available as of 10.1 only. If you set the `optimizer_features_enable` initialization parameter to a value through 9.2.0.8, the query optimizer doesn't always behave as explained here. Because such a configuration isn't common at all, no further information about the differences in behavior is provided. Refer to Chapter 9 for information about `optimizer_features_enable`.

---

When system statistics are available, the query optimizer computes two costs: I/O and CPU. Chapter 9 describes how I/O costs are computed for the most important access paths. Very little information is available about the computation of CPU costs. Nevertheless, we can imagine that the query optimizer associates a cost to every operation in terms of CPU. For example, Formula 7-1 is used to compute the CPU cost of accessing a column.

***Formula 7-1.*** *The estimated CPU cost to access a column depends on its position in the table. This formula gives the cost of accessing one row. If several rows are accessed, the CPU cost increases proportionally. Chapter 16 provides further information on why the position of a column is relevant.*

$$cpu\_cost = column\_position \cdot 20$$

The following example, which is an excerpt of the `cpu_cost_column_access.sql` script, shows Formula 7-1 in action. A table with nine columns is created, one row is inserted, and then with the `EXPLAIN PLAN` statement, the CPU cost of independently accessing the nine columns is displayed. See Chapter 10 for detailed information about this SQL statement. Notice how there's an initial CPU cost of 35,757 to access the table, and then for each subsequent column, a CPU cost of 20 is added. At the same time, the I/O cost is constant. This makes sense because all columns are stored in the very same database block, and therefore the number of physical reads required to read them is the same for all queries:

```
SQL> CREATE TABLE t (c1 NUMBER, c2 NUMBER, c3 NUMBER,
  2                  c4 NUMBER, c5 NUMBER, c6 NUMBER,
  3                  c7 NUMBER, c8 NUMBER, c9 NUMBER);

SQL> INSERT INTO t VALUES (1, 2, 3, 4, 5, 6, 7, 8, 9);

SQL> EXPLAIN PLAN SET STATEMENT_ID 'c1' FOR SELECT c1 FROM t;
SQL> EXPLAIN PLAN SET STATEMENT_ID 'c2' FOR SELECT c2 FROM t;
SQL> EXPLAIN PLAN SET STATEMENT_ID 'c3' FOR SELECT c3 FROM t;
SQL> EXPLAIN PLAN SET STATEMENT_ID 'c4' FOR SELECT c4 FROM t;
SQL> EXPLAIN PLAN SET STATEMENT_ID 'c5' FOR SELECT c5 FROM t;
SQL> EXPLAIN PLAN SET STATEMENT_ID 'c6' FOR SELECT c6 FROM t;
SQL> EXPLAIN PLAN SET STATEMENT_ID 'c7' FOR SELECT c7 FROM t;
SQL> EXPLAIN PLAN SET STATEMENT_ID 'c8' FOR SELECT c8 FROM t;
SQL> EXPLAIN PLAN SET STATEMENT_ID 'c9' FOR SELECT c9 FROM t;

SQL> SELECT statement_id, cpu_cost AS total_cpu_cost,
  2         cpu_cost-lag(cpu_cost) OVER (ORDER BY statement_id) AS cpu_cost_1_coll,
  3         io_cost
  4  FROM plan_table
  5  WHERE id = 0
  6  ORDER BY statement_id;

STATEMENT_ID TOTAL_CPU_COST CPU_COST_1_COLL IO_COST
------------ -------------- --------------- -------
c1                    35757                       3
c2                    35777              20       3
c3                    35797              20       3
c4                    35817              20       3
c5                    35837              20       3
c6                    35857              20       3
c7                    35877              20       3
c8                    35897              20       3
c9                    35917              20       3
```

The I/O and CPU costs are expressed with different units of measurement. Obviously then, the overall cost of a SQL statement can't be calculated simply by summing up the costs. To solve this problem, the query optimizer uses Formula 7-2 with workload statistics. Simply put, the CPU cost is divided by `cpuspeed` to get the estimated elapsed time and then divided by `sreadtim` to express the cost in the same unit of measurement as `io_cost`.

***Formula 7-2.*** *The overall costs are based on the I/O costs and the CPU costs.*

$$cost \approx io\_cost + \frac{cpu\_cost}{cpuspeed \cdot sreadtim \cdot 1000}$$

To compute the overall cost with noworkload statistics, in Formula 7-2 `cpuspeed` is replaced by `cpuspeednw`, and `sreadtim` is computed using Formula 7-3. Simply put, to compute `sreadtim`, Formula 7-3 adds the time needed to locate a block on disk with the time needed to transfer it to the database instance.

***Formula 7-3.*** *If necessary, sreadtim is computed based on noworkload statistics and the default block size of the database.*

$$sreadtim \approx ioseektim + \frac{db\_block\_size}{iotfrspeed}$$

Generally speaking, if workload statistics are available, the query optimizer uses them and ignores noworkload statistics. You should be aware that the query optimizer performs several sanity checks that could disable or partially replace workload statistics. You can look into this behavior through the `system_stats_sanity_checks.sql` script. The following are some items to watch for:

- When `mbrc` isn't available or set to 0, the query optimizer ignores workload statistics. The result is that noworkload statistics are used instead.
- When `sreadtim` isn't available or set to 0, the query optimizer recomputes the value of `sreadtim` and `mreadtim` using Formula 7-3 and Formula 7-4, respectively.
- When `mreadtim` isn't available, or when it isn't greater than `sreadtim`, the query optimizer recomputes the value of `sreadtim` and `mreadtim` using Formula 7-3 and Formula 7-4, respectively.

***Formula 7-4.*** *The computation of* `mreadtim` *based on noworkload statistics and the default block size of the database*

$$mreadtim \approx ioseektim + \frac{mbrc \cdot db\_block\_size}{iotfrspeed}$$

A particular case where Formula 7-3 and Formula 7-4 are used is when only noworkload statistics gathered in exadata mode are available. In fact, with this kind of statistics, all estimations are based on `mbrc`, `ioseektim`, and `iotfrspeed`.

What's the role played by `slavethr` and `maxthr` for the estimations related to SQL statements that are considered to be executed in parallel? Simply put, while the former can increase the cost of parallel executions, the latter can cap the cost of parallel executions with a high degree of parallelism. I'll now discuss the impact of these two statistics in detail.

When `slavethr` and `maxthr` aren't set, the query optimizer, as shown by Formula 7-5, considers that the cost of an operation executed in parallel is inversely proportional to the degree of parallelism used to execute it. As a result, the query optimizer considers that whatever the degree of parallelism is, every slave process running in parallel is able to sustain the throughput computed by Formula 7-6.

***Formula 7-5.*** *The parallel I/O cost is inversely proportional to the degree of parallelism. Note that the constant 0.9 is a fudge factor that probably takes into account the inevitable contention due to parallel processing.*

$$parallel\_io\_cost \approx \frac{serial\_io\_cost}{dop \cdot 0.9}$$

***Formula 7-6.*** *The computation of the expected throughput (in bytes per second) of a single server process based on workload statistics and the default block size of the database*

$$mreadthr \approx \frac{mbrc \cdot db\_block\_size}{mreadtim} \cdot 1000$$

In case the query optimizer is too optimistic in estimating the costs of parallel operations, with `slavethr` you can increase the estimations. To do so, you set `slavethr` to a value lower than `mreadthr`, which is the value computed with Formula 7-6. In other words, you inform the query optimizer that the throughput that each slave process can sustain is lower than the default value. Note that the opposite—decreasing the cost by setting `slavethr` to a value higher than `mreadthr`—is not possible. In fact, when the ratio of `slavethr` to `mreadthr` is higher than 0.9 (the fudge factor used in Formula 7-5), it has no impact on the cost estimated by the query optimizer. Figure 7-4 shows, for a full table scan, the impact of setting `slavethr` to half of the value of `mreadthr`.

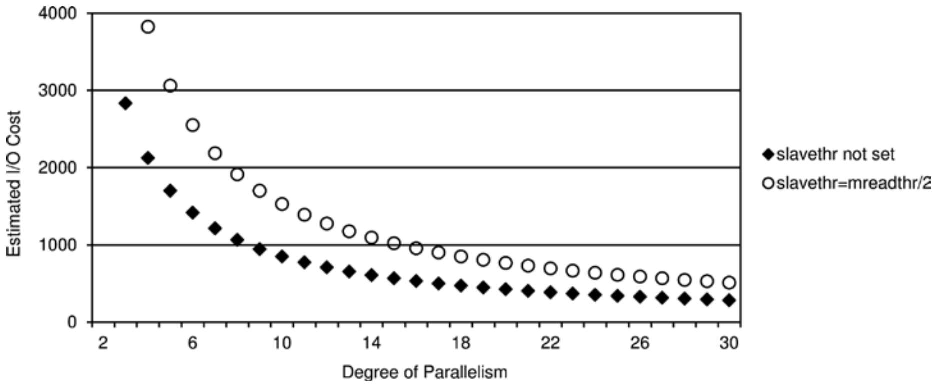


***Figure 7-4.*** *Comparison of the estimated I/O costs with and without* `slavethr` *(data generated by the* `parallel_fts_costing.sql` *script)*

The adjustment due to `slavethr` is shown in Formula 7-7. Notice the difference with Formula 7-5: the fudge factor (0.9) is only used when it's higher than the ratio of `slavethr` to `mreadthr`.

***Formula 7-7.*** *An adjustment (increase) of the parallel I/O costs take place when the ratio of* `slavethr` *to* `mreadthr` *is lower than 0.9 (note that k is defined in the note following the formula).*

$$parallel\_io\_cost \approx \frac{serial\_io\_cost}{dop \cdot \text{least}\left(0.9\ ,\ \frac{slavethr \cdot k}{mreadthr}\right)}$$

---

■ **Note** In Formulas 7-7 and 7-8, the factor *k* depends on the database release. Up to version 11.2.0.3, it has the value 1,000. From version 11.2.0.4 onward, it has the value 1. As a result, through version 11.2.0.3 only very small values of `slavethr` compared to `mreadthr` actually impact the estimations of the query optimizer. For this reason, in practice, most of the time no impact is observed through version 11.2.0.3.

---

As Formula 7-7 clearly shows, the costs stay inversely proportional to the degree of parallelism, and therefore `slavethr` can only be used to increase the costs, not cap them. In reality, the actual resource consumption isn't always inversely proportional to the degree of parallelism. In fact, because database servers don't scale infinitely, for high degrees of parallelism the estimated costs are too low. This is exactly why `maxthr` is available. Figure 7-5 shows, for the same case illustrated in Figure 7-4, the impact of setting `maxthr` to prevent the costs from falling below a specific threshold. Notice that although the minimum cost is independent of the `slavethr` value, the cap takes place at a different degree of parallelism.

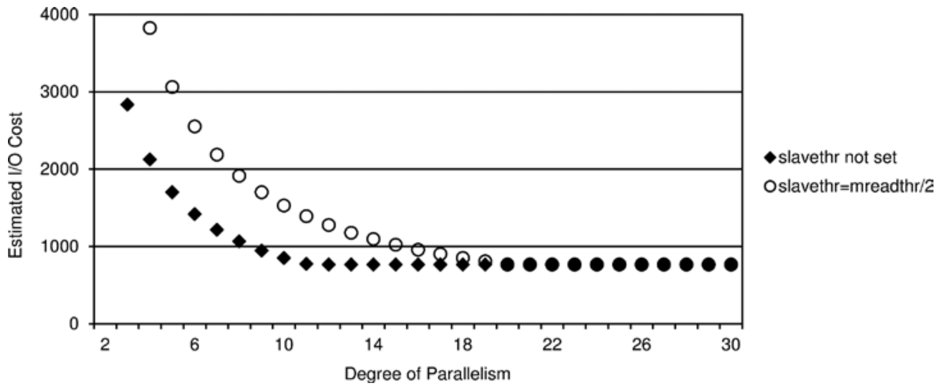


***Figure 7-5.*** *Comparison of the estimated I/O costs with* `maxthr` *set (data generated by the* `parallel_fts_costing.sql` *script)*

As illustrated in Figure 7-5, the value of `maxthr` stops the cost of decreasing when the degree of parallelism is getting too high. Simply put, the query optimizer computes, based on Formula 7-8, that the cost can't be lower than a specific value.

***Formula 7-8.*** *The ratio between the expected throughput of a single server process and the maximum disk I/O throughput for the whole system limits the parallel I/O cost (note that k is defined in the note of the previous page)*

$$minimum\_parallel\_io\_cost \approx serial\_io\_cost \cdot \frac{mreadthr}{maxthr \cdot k}$$

As discussed in this section, system statistics make the query optimizer aware of the system where the database engine is running. This means they're essential for a successful configuration. I recommend freezing them in order to have some stability in the generation of execution plans. In other words, I consider them as initialization parameters.

Of course, in case of major hardware or software changes, system statistics should be recomputed, and as a result, the whole configuration should be checked. For checking purposes, it's also possible to regularly gather them in a backup table (in other words, using the `statown` and `stattab` parameters of the `gather_system_stats` procedure) and verify whether there's a major difference between the current values and the values stored in the data dictionary.

## On to Chapter 8

This chapter describes what system statistics are and why the query optimizer needs them. Simply put, they provide performance information about the CPU and the disk I/O subsystem. The chapter also coveres how to manage system statistics with the `dbms_stats` package, and where to find them in the data dictionary.

System statistics aren't sufficient though, to fully describe the environment in which the query optimizer operates. The query optimizer also needs insight about the data stored in the database. For that purpose, another type of statistics is available: object statistics. The next chapter provides full coverage of this second type of statistics.

CHAPTER 8

# Object Statistics

Object statistics describe the data stored in the database. For example, they tell the query optimizer how many rows are stored in tables. Without this quantitative information, the query optimizer could never make right decisions, such as finding the right join method for small or large tables (result sets). To illustrate this, consider the following example. Let's say I ask you what's the fastest method of transportation for me to get home from a particular place. Would it be by car, by train, or by plane? Why not by bike? The point is, without considering my actual position and where my home is, you can't arrive at a meaningful answer. Without object statistics, the query optimizer has the same problem. It simply can't generate optimal execution plans.

This chapter begins by describing which object statistics are available and where to find them in the data dictionary. Then it presents the features of the `dbms_stats` package used to gather, restore, lock, compare, and delete statistics. Finally, it describes a few strategies that I use to manage object statistics, making full use of the available features. What the query optimizer does with object statistics is described here for only a few cases. The purpose of most statistics is explained in Chapter 9. Because the query optimizer uses statistics and initialization parameters at the same time, it makes sense to describe them together in the same chapter.

**Note** The database engine, through the `ASSOCIATE STATISTICS` statement, allows user-defined statistics to be associated with columns, functions, packages, types, domain indexes, and index types. When needed, this SQL statement is a very powerful feature, although in practice this technique is rarely used. For this reason, `ASSOCIATE STATISTICS` isn't covered here. For information about it, refer to the *Oracle Database Data Cartridge Developer's Guide* manual and to Chapter 7 of *Expert Oracle Practices* (Apress, 2010).

## The dbms_stats Package

It used to be that object statistics were gathered with the `ANALYZE` statement. This is no longer the case. For gathering object statistics, the `ANALYZE` statement is available, but only for purposes of backward compatibility. As of Oracle9*i*, it's recommended that you use the `dbms_stats` package. In fact, not only does the `dbms_stats` package provide many more features, but in some situations it provides better statistics as well. For example, the `ANALYZE` statement provides less control over the gathering of statistics, doesn't support external tables, and for partitioned objects, gathers statistics only for each segment and derives (usually poorly) the statistics at the table/index level. For these reasons, I don't cover the `ANALYZE` statement in this chapter.

It's important to recognize that the `dbms_stats` package provides a comprehensive set of procedures and functions to manage object statistics. Because there are a lot of objects in a database, it's important to be able to manage their statistics at different granularities. You have the choice between managing the object statistics for the whole database, for the data dictionary, for a single schema, for a single table, for a single index, or for a single table or index partition.

By default, the dbms_stats package modifies the data dictionary directly. Nevertheless, with many of its procedures and functions, it's also possible to work on a user-defined table stored outside the data dictionary. This is what I call the *backup table*.

Because managing statistics means much more than simply gathering them, the dbms_stats package provides the following key features (see Figure 8-1):

- Gathering object statistics and, optionally, storing the current statistics in a backup table before overwriting them
- Locking and unlocking object statistics stored in the data dictionary
- Copying object statistics from one partition or subpartition to another
- Restoring object statistics in the data dictionary
- Deleting object statistics stored in the data dictionary or a backup table
- Exporting object statistics from the data dictionary to a backup table
- Importing object statistics from a backup table to the data dictionary
- Getting (extracting) object statistics stored in the data dictionary or a backup table
- Setting (modifying) object statistics stored in the data dictionary or a backup table

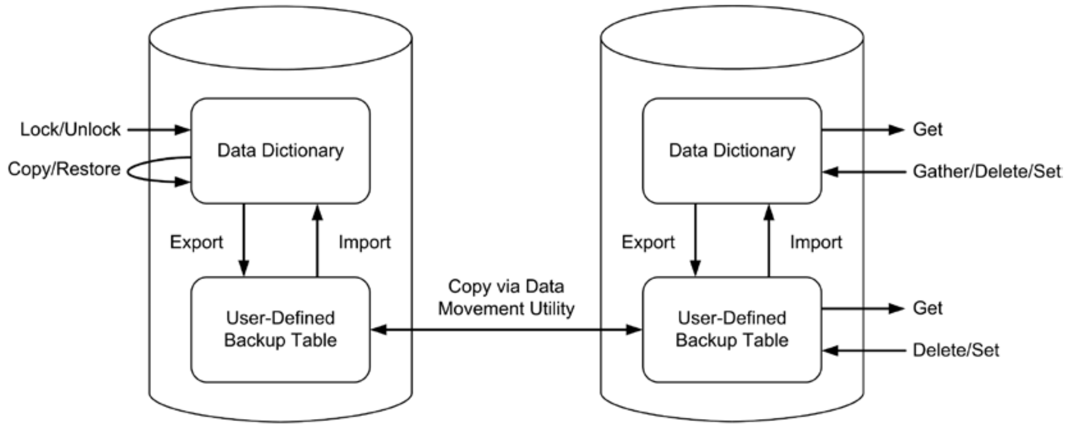


***Figure 8-1.*** *The dbms_stats package provides a comprehensive set of features to manage object statistics*

Note that moving statistics between databases is performed by means of a generic data movement utility (for example, Data Pump), not with the dbms_stats package itself.

Depending on the granularity and the operation you want to execute, Table 8-1 shows that the dbms_stats package provides different procedures and functions. For example, if you want to operate on a single schema, the dbms_stats package provides gather_schema_stats, delete_schema_stats, lock_schema_stats, unlock_schema_stats, restore_schema_stats, export_schema_stats, and import_schema_stats.

***Table 8-1.*** *Features Provided by the `dbms_stats` Package*

| Feature | Database | Dictionary | Schema | Table* | Index* |
|---|---|---|---|---|---|
| Gather/Delete | ✓ | ✓ | ✓ | ✓ | ✓ |
| Lock/Unlock | | | ✓ | ✓ | |
| Copy | | | | ✓ | |
| Restore | ✓ | ✓ | ✓ | ✓ | |
| Export/Import | ✓ | ✓ | ✓ | ✓ | ✓ |
| Get/Set | | | | ✓ | ✓ |

**For partitioned objects, limiting the processing to a single partition is possible.*

## What Object Statistics Are Available?

There are three types of object statistics: table statistics, column statistics, and index statistics. For each type, there are up to three subtypes: table/index-level statistics, partition-level statistics, and subpartition-level statistics. It may be obvious that partition and subpartition statistics exist only when an object is partitioned and subpartitioned, respectively.

Object statistics are shown in the data dictionary views reported in Table 8-2. Of course, for each view there are `dba`, `all`, and, in a 12.1 multitenant environment, `cdb` versions as well—for example, `dba_tab_statistics`, `all_tab_statistics` and `cdb_tab_statistics`.

***Table 8-2.*** *Data Dictionary Views Showing Object Statistics of Relational Tables*

| Object | Table/Index-Level Statistics | Partition-Level Statistics | Subpartition-Level Statistics |
|---|---|---|---|
| Tables | `user_tab_statistics` | `user_tab_statistics` | `user_tab_statistics` |
| Columns | `user_tab_col_statistics` `user_tab_histograms` | `user_part_col_statistics` `user_part_histograms` | `user_subpart_col_statistics` `user_subpart_histograms` |
| Indexes | `user_ind_statistics` | `user_ind_statistics` | `user_ind_statistics` |

The rest of this section describes the most important object statistics available in the data dictionary. For this purpose, I created a test table with the following SQL statements. These SQL statements, as well as all other queries in this section, are available in the `object_statistics.sql` script:

```
CREATE TABLE t
AS
SELECT rownum AS id,
       50+round(dbms_random.normal*4) AS val1,
       100+round(ln(rownum/3.25+2)) AS val2,
       100+round(ln(rownum/3.25+2)) AS val3,
       dbms_random.string('p',250) AS pad
FROM dual
CONNECT BY level <= 1000
ORDER BY dbms_random.value;
```

```
UPDATE t SET val1 = NULL WHERE val1 < 0;

ALTER TABLE t ADD CONSTRAINT t_pk PRIMARY KEY (id);

CREATE INDEX t_val1_i ON t (val1);

CREATE INDEX t_val2_i ON t (val2);

BEGIN
  dbms_stats.gather_table_stats(
    ownname          => user,
    tabname          => 'T',
    estimate_percent => 100,
    method_opt       => 'for columns size skewonly id, val1 size 15, val2, val3 size 5, pad',
    cascade          => TRUE
  );
END;
/
```

## Table Statistics

The following query shows how to get the most important table statistics for a table:

```
SQL> SELECT num_rows, blocks, empty_blocks, avg_space, chain_cnt, avg_row_len
  2  FROM user_tab_statistics
  3  WHERE table_name = 'T';

NUM_ROWS BLOCKS EMPTY_BLOCKS AVG_SPACE CHAIN_CNT AVG_ROW_LEN
-------- ------ ------------ --------- --------- -----------
    1000     44            0         0         0         266
```

Here is an explanation of the table statistics returned by this query:

- `num_rows` is the number of rows in the table.
- `blocks` is the number of blocks below the high watermark in the table.
- `empty_blocks` is the number of blocks above the high watermark in the table. This value isn't computed by the `dbms_stats` package. It's set to 0 (unless another value is already stored in the data dictionary).
- `avg_space` is the average free space (in bytes) in the table's data blocks. This value isn't computed by the `dbms_stats` package. It's set to 0 (unless another value is already stored in the data dictionary).
- `chain_cnt` is the sum of the rows in the table that are chained or migrated to another block (chained and migrated rows are described in Chapter 16). Even though the query optimizer uses this value, the `dbms_stats` package doesn't compute it. It's set to 0 (unless another value is already stored in the data dictionary).
- `avg_row_len` is the average size (in bytes) of a row in the table.

### HIGH WATERMARK

The *high watermark* is the boundary between used and unused space in a segment. The used blocks are below the high watermark, and therefore, the unused blocks are above the high watermark. Blocks above the high watermark have never been used or initialized.

In normal circumstances, operations requiring space (for example, `INSERT` statements) increase the high watermark only if there is no more free space available below the high watermark. A common exception to this is due to direct-path inserts because they exclusively use blocks above the high watermark (refer to Chapter 15).

Operations releasing space (for example, `DELETE` statements) don't decrease the high watermark. They simply make space available to other operations. If the free space is released at a rate equal to or lower than the rate the space is reused, the use of the blocks below the high watermark should be optimal. Otherwise, the free space below the high watermark would increase steadily. Long-term, this would cause not only an unnecessary increase in the size of the segment but also suboptimal performance. In fact, full scans access all blocks below the high watermark. This occurs even if they're empty. The segment should be reorganized to solve such a problem.

## Column Statistics

The following query shows how to get the most important column statistics for a table:

```
SQL> SELECT column_name AS "NAME",
  2         num_distinct AS "#DST",
  3         low_value,
  4         high_value,
  5         density AS "DENS",
  6         num_nulls AS "#NULL",
  7         avg_col_len AS "AVGLEN",
  8         histogram,
  9         num_buckets AS "#BKT"
 10  FROM user_tab_col_statistics
 11  WHERE table_name = 'T';

NAME  #DST LOW_VALUE           HIGH_VALUE             DENS #NULL AVGLEN HISTOGRAM        #BKT
---- ----- ------------------- ------------------- ------- ----- ------ --------------- -----
ID    1000 C102                C20B                 .00100     0      4 NONE                1
VAL1    22 C128                C140                 .03884     0      3 HYBRID             15
VAL2     6 C20202              C20207               .00050     0      4 FREQUENCY           6
VAL3     6 C20202              C20207               .00050     0      4 TOP-FREQUENCY       5
PAD   1000 202623436F294373342 7E79514A202D4946493  .00100     0    251 HYBRID            254
           37B426574336E4A5B30 66C744E253F36264C69
           2E4F4B53236932303A2 27557A57737C6D4B225
           1215F462B7667457032 9414C442D2544364130
           694174782F7749393B6 612F5B3447405A4E714
           5735646366D20736939 A403B6237592B3D7B67
           335D712B233B3F      7D4D594E766B57
```

Here is an explanation of the column statistics returned by this query:

- `num_distinct` is the number of distinct values in the column.
- `low_value` is the lowest value in the column. It's shown in the internal representation. Note that for string columns (in the example, the `pad` column), only the first 32 bytes (64 bytes as of version 12.1) are used.
- `high_value` is the highest value in the column. It's shown in the internal representation. Notice that for string columns (in the example, the `pad` column), only the first 32 bytes (64 bytes as of version 12.1) are used.

## LOW_VALUE AND HIGH_VALUE FORMAT

Unfortunately, the columns `low_value` and `high_value` aren't easily decipherable. In fact, they display the values using the binary internal representation used by the database engine to store data. To convert them to human-readable values, there are two possibilities.

First, the `utl_raw` package provides the functions `cast_to_binary_double`, `cast_to_binary_float`, `cast_to_binary_integer`, `cast_to_number`, `cast_to_nvarchar2`, `cast_to_raw`, and `cast_to_varchar2`. As the names of the functions suggest, for each specific datatype, there is a corresponding function used to convert the internal value to the actual value. For instance, to get the low and high value of the `val1` column, you can use the following query:

```
SQL> SELECT utl_raw.cast_to_number(low_value) AS low_value,
  2         utl_raw.cast_to_number(high_value) AS high_value
  3  FROM user_tab_col_statistics
  4  WHERE table_name = 'T'
  5  AND column_name = 'VAL1';

LOW_VALUE HIGH_VALUE
--------- ----------
       39         63
```

Second, the `dbms_stats` package provides the procedures `convert_raw_value` (which is overloaded several times), `convert_raw_value_nvarchar`, and `convert_raw_value_rowid`. Notice that to avoid using a PL/SQL block, the following query uses the version 12.1 possibility to declare PL/SQL functions and procedures in the `WITH` clause. The purpose of the query is the same as the previous one (in the `object_statistics.sql` script, you find a variation of this query that supports all the most common datatypes):

```
SQL> WITH
  2    FUNCTION convert_raw_value(p_value IN RAW) RETURN NUMBER IS
  3      l_ret NUMBER;
  4    BEGIN
  5     dbms_stats.convert_raw_value(p_value, l_ret);
  6     RETURN l_ret;
  7    END;
  8  SELECT convert_raw_value(low_value) AS low_value,
  9         convert_raw_value(high_value) AS high_value
 10  FROM user_tab_col_statistics
 11  WHERE table_name = 'T'
 12  AND column_name = 'VAL1'
 13  /

LOW_VALUE HIGH_VALUE
--------- ----------
       39         63
```

- `density` is a decimal number between 0 and 1. Values close to 0 indicate that a restriction on that column filters out the majority of the rows. Values close to 1 indicate that a restriction on that column filters almost no rows. If no histogram is present, `density` is `1/num_distinct`. If a histogram is present, the computation differs and depends on the type of histogram. In any case, as of version 10.2.0.4, for columns with histograms, this value is only used for backward compatibility when the `optimizer_features_enable` initialization parameter is set to an older release.
- `num_nulls` is the number of `NULL` values stored in the column.
- `avg_col_len` is the average column size in bytes.
- `histogram` indicates whether a histogram is available for the column and, if it's available, which type it is. Valid values are `NONE` (meaning no histogram), `FREQUENCY`, `HEIGHT BALANCED`, and, as of version 12.1, `TOP-FREQUENCY` and `HYBRID`.
- `num_buckets` is the number of buckets in the histogram. A bucket, or *category* as it's called in statistics, is a group of values of the same kind. As described in the next section, histograms are composed of at least one bucket. If no histogram is available, it's set to 1. The maximum number of buckets is 254 up to version 11.2, and 2,048 as of version 12.1.

## Histograms

The query optimizer starts from the principle that data is uniformly distributed. An example of a uniformly distributed set of data is the one stored in the `id` column in the test table used throughout the previous sections. In fact, it stores all integers from 1 up to 1,000 exactly once. In such a case, to produce a good estimate of the number of rows filtered out by a predicate based on that column (for example, `id BETWEEN 6 AND 19`), the query optimizer requires only the object statistics described in the preceding section: the minimum value, the maximum value, and the number of distinct values.

If data isn't uniformly distributed, the query optimizer can't compute acceptable estimations without additional information. For example, given the data set stored in the `val2` column (see the output of the following query), how could the query optimizer make a meaningful estimation for a predicate like `val2=105`? It can't, because it has no clue that about 50 percent of the rows fulfill that predicate:

```
SQL> SELECT val2, count(*)
  2  FROM t
  3  GROUP BY val2
  4  ORDER BY val2;

      VAL2   COUNT(*)
---------- ----------
       101          8
       102         25
       103         68
       104        185
       105        502
       106        212
```

The additional information needed by the query optimizer to get information about the nonuniform distribution of data is called a *histogram*. Prior to version 12.1, two types of histograms are available: *frequency histograms* and *height-balanced histograms*. Oracle Database 12.1 introduces two additional types to replace height-balanced histograms: *top frequency histograms* and *hybrid histograms*.

---

■ **Caution** The `dbms_stats` package builds top frequency histograms and hybrid histograms only when the sampling used for gathering the object statistics is based on `dbms_stats.auto_sample_size` (later in this chapter, the "Gathering Options" section covers this topic).

---

## Frequency Histograms

The frequency histogram is what most people understand by the term *histogram*. Figure 8-2 is an example of this type, which shows a common graphical representation of the data returned by the previous query.

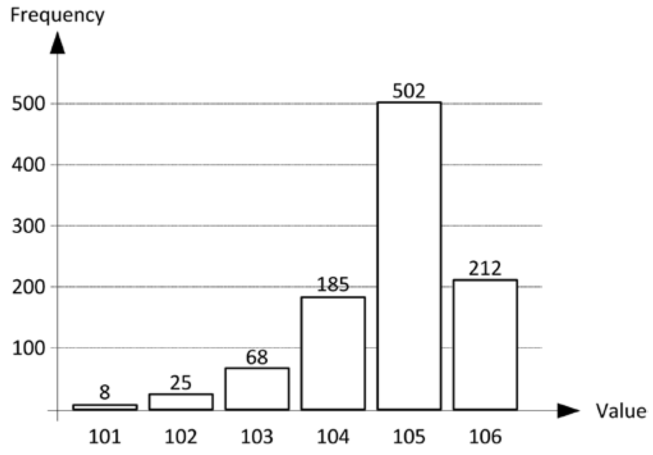


***Figure 8-2.*** *Graphical representation of a frequency histogram based on the set of data stored in the* `val2` *column*

The frequency histogram stored in the data dictionary is similar to this representation. The main difference is that instead of the frequency, the cumulated frequency is used. The following query turns the cumulated frequency into the frequency by computing the difference between two consecutive bucket values (notice that the `endpoint_number` column is the cumulated frequency):

```
SQL> SELECT endpoint_value, endpoint_number,
  2         endpoint_number - lag(endpoint_number,1,0)
  3                           OVER (ORDER BY endpoint_number) AS frequency
  4  FROM user_tab_histograms
  5  WHERE table_name = 'T'
  6  AND column_name = 'VAL2'
  7  ORDER BY endpoint_number;
```

```
ENDPOINT_VALUE ENDPOINT_NUMBER FREQUENCY
-------------- --------------- ---------
           101               8         8
           102              33        25
           103             101        68
           104             286       185
           105             788       502
           106            1000       212
```

The essential characteristics of a frequency histogram are the following:

- The number of buckets (in other words, the number of categories) is the same as the number of distinct values. A row in available for each bucket in a view like `user_tab_histograms`.
- The `endpoint_value` column provides a numerical representation of the value itself. Hence, for non-numerical datatypes, the actual values must be encoded in a number. Depending on the data, the datatype, and the version, the actual values might be visible in the `endpoint_actual_value` column (not shown in the previous output). It's essential to know that values stored in histograms are distinguished based only on their leading 32 bytes (64 bytes as of version 12.1). As a result, long fixed prefixes might jeopardize the effectiveness of histograms. This is especially true for multibyte character sets where each character might take up to three bytes.
- The `endpoint_number` column provides the cumulated frequency of the value. To get the frequency itself, the value of the `endpoint_number` column of the previous row must be subtracted.

---

■ **Caution** In case sampling is used to build a histogram, the frequency information is scaled proportionally to the sample size. To know the scaling factor, divide the sample size (`sample_size`) by the number of rows (`num_rows`). Both columns are provided by views like `user_tab_statistics`.

---

The following example shows how the query optimizer takes advantage of the frequency histogram to estimate precisely the number of rows returned by a query (the *cardinality*) that has a predicate on the `val2` column (detailed information about the `EXPLAIN PLAN` statement is provided in Chapter 10):

```
SQL> EXPLAIN PLAN SET STATEMENT_ID '101' FOR SELECT * FROM t WHERE val2 = 101;
SQL> EXPLAIN PLAN SET STATEMENT_ID '102' FOR SELECT * FROM t WHERE val2 = 102;
SQL> EXPLAIN PLAN SET STATEMENT_ID '103' FOR SELECT * FROM t WHERE val2 = 103;
SQL> EXPLAIN PLAN SET STATEMENT_ID '104' FOR SELECT * FROM t WHERE val2 = 104;
SQL> EXPLAIN PLAN SET STATEMENT_ID '105' FOR SELECT * FROM t WHERE val2 = 105;
SQL> EXPLAIN PLAN SET STATEMENT_ID '106' FOR SELECT * FROM t WHERE val2 = 106;

SQL> SELECT statement_id, cardinality
  2  FROM plan_table
  3  WHERE id = 0;
```

```
STATEMENT_ID CARDINALITY
------------ -----------
101                    8
102                   25
103                   68
104                  185
105                  502
106                  212
```

In the preceding example, all predicates reference only values that are represented in the histogram. But what happens when other values are used? Up to and including 10.2.0.3, the query optimizer used 1 as frequency. From version 10.2.0.4 onward, there are two distinct situations to consider. First, if the value is between the minimum and the maximum values, the query optimizer takes the lowest frequency of all values represented in the histogram and divides it by two. Second, if the value is outside the range covered by the histogram, the frequency depends on the distance from the lowest/maximum value. The following example illustrates:

```
SQL> EXPLAIN PLAN SET STATEMENT_ID '096' FOR SELECT * FROM t WHERE val2 = 96;
SQL> EXPLAIN PLAN SET STATEMENT_ID '098' FOR SELECT * FROM t WHERE val2 = 98;
SQL> EXPLAIN PLAN SET STATEMENT_ID '100' FOR SELECT * FROM t WHERE val2 = 100;
SQL> EXPLAIN PLAN SET STATEMENT_ID '103.5' FOR SELECT * FROM t WHERE val2 = 103.5;
SQL> EXPLAIN PLAN SET STATEMENT_ID '107' FOR SELECT * FROM t WHERE val2 = 107;
SQL> EXPLAIN PLAN SET STATEMENT_ID '109' FOR SELECT * FROM t WHERE val2 = 109;
SQL> EXPLAIN PLAN SET STATEMENT_ID '111' FOR SELECT * FROM t WHERE val2 = 111;

SQL> SELECT statement_id, cardinality
  2  FROM plan_table
  3  WHERE id = 0
  4  ORDER BY statement_id;

STATEMENT_ID CARDINALITY
------------ -----------
096                    1
098                    2
100                    3
103.5                  4
107                    3
109                    2
111                    1
```

## Height-Balanced Histograms

When the number of distinct values is greater than the maximum number of allowed buckets (with the dbms_stats package, there is both a hard limit and the possibility to specify an even lower value), you can't use frequency histograms because they support a single value per bucket. This is where height-balanced histograms become useful.

To create a height-balanced histogram, think of the following procedure. First, a frequency histogram is created. Then, as shown in Figure 8-3, the values of the frequency histogram are stacked in a pile. Finally, the pile is divided into several buckets of exactly the same height. For example, in Figure 8-3 the pile is split into five buckets.

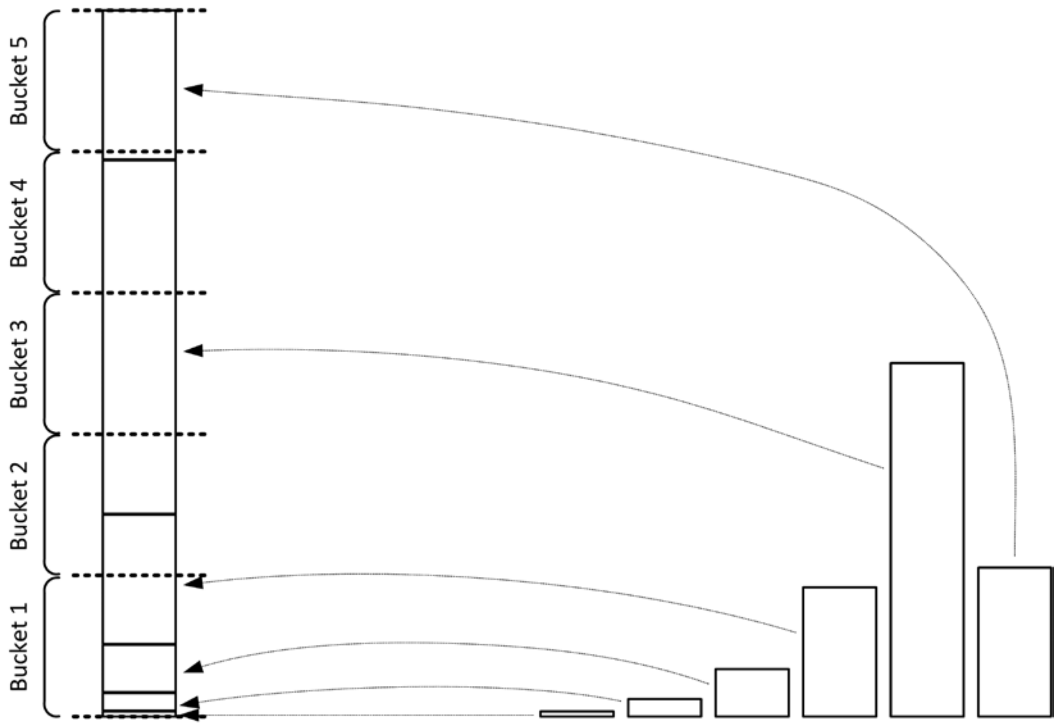


***Figure 8-3.** Transformation of a frequency histogram into a height-balanced histogram*

The following query is an example of how to produce such a height-balanced histogram for the `val2` column. Figure 8-4 shows a graphical representation of the data it returns. Notice how the endpoint value of each bucket is the value at the point where the split occurs. In addition, a bucket 0 is added to store the minimum value:

```
SQL> SELECT count(*), max(val2) AS endpoint_value, endpoint_number
  2  FROM (
  3    SELECT val2, ntile(5) OVER (ORDER BY val2) AS endpoint_number
  4    FROM t
  5  )
  6  GROUP BY endpoint_number
  7  ORDER BY endpoint_number;

  COUNT(*) ENDPOINT_VALUE ENDPOINT_NUMBER
---------- -------------- ---------------
       200            104               1
       200            105               2
       200            105               3
       200            106               4
       200            106               5
```

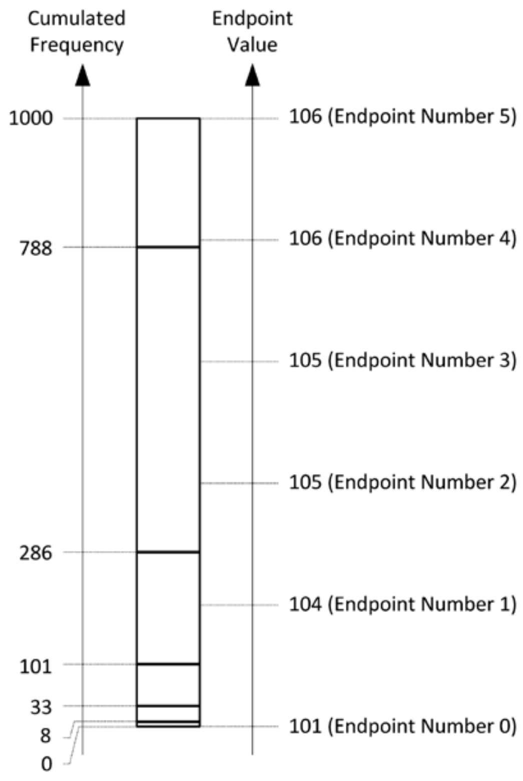


***Figure 8-4.*** *Graphical representation of a height-balanced histogram based on the set of data stored in the* `val2` *column*

For the case of Figure 8-4, the following query shows the height-balanced histogram stored in the data dictionary. Interestingly enough, not all buckets are stored. This doesn't happen because several adjacent buckets with the same endpoint value are of no use. In fact, from this data it's possible to infer that bucket 2 has an endpoint value of 105 and bucket 4 has an endpoint value of 106. The result is a kind of compression. The values appearing several times in the histogram are called *popular values* and are especially handled by the query optimizer:

```
SQL> SELECT endpoint_value, endpoint_number
  2  FROM user_tab_histograms
  3  WHERE table_name = 'T'
  4  AND column_name = 'VAL2'
  5  ORDER BY endpoint_number;

ENDPOINT_VALUE ENDPOINT_NUMBER
-------------- ---------------
           101               0
           104               1
           105               3
           106               5
```

Here are the main characteristics of height-balanced histograms:

- The number of buckets is less than the number of distinct values. For each bucket, except when they're compressed, a row with the endpoint number is available in a view like `user_tab_histograms`. In addition, the endpoint number 0 indicates the minimum value.
- The `endpoint_value` column gives a numerical representation of the value itself. For additional information about this column, refer to the description provided in the "Height-Balanced Histograms" section.
- The `endpoint_number` column gives the bucket number.
- The histogram doesn't store the frequency of the values.

The following example shows the estimations performed by the query optimizer when the height-balanced histogram is in place. Note the lower precision compared to the frequency histogram:

```
SQL> EXPLAIN PLAN SET STATEMENT_ID '101' FOR SELECT * FROM t WHERE val2 = 101;
SQL> EXPLAIN PLAN SET STATEMENT_ID '102' FOR SELECT * FROM t WHERE val2 = 102;
SQL> EXPLAIN PLAN SET STATEMENT_ID '103' FOR SELECT * FROM t WHERE val2 = 103;
SQL> EXPLAIN PLAN SET STATEMENT_ID '104' FOR SELECT * FROM t WHERE val2 = 104;
SQL> EXPLAIN PLAN SET STATEMENT_ID '105' FOR SELECT * FROM t WHERE val2 = 105;
SQL> EXPLAIN PLAN SET STATEMENT_ID '106' FOR SELECT * FROM t WHERE val2 = 106;

SQL> SELECT statement_id, cardinality
  2  FROM plan_table
  3  WHERE id = 0;

STATEMENT_ID CARDINALITY
------------ -----------
101                   50
102                   50
103                   50
104                   50
105                  400
106                  300
```

■ **Note** You might expect the cardinality estimation of the values 105 and 106 to be identical (400, because both popular values cover 2/5 of the buckets). However, for the value 106, this isn't the case. That's because the query optimizer adjusts its estimations when a popular value is also the maximum value represented in the histogram.

Also for this type of histogram, let's have a look at what happens when values not represented in the histogram are used. There are two distinct situations to consider. First, if the value is between the minimum and the maximum values, the query optimizer uses the same frequency as for the other non-popular values. Second, if the value is outside the range covered by the histogram, the frequency depends on the distance from the lowest/maximum value. The following example illustrates:

```
SQL> EXPLAIN PLAN SET STATEMENT_ID '096' FOR SELECT * FROM t WHERE val2 = 96;
SQL> EXPLAIN PLAN SET STATEMENT_ID '098' FOR SELECT * FROM t WHERE val2 = 98;
SQL> EXPLAIN PLAN SET STATEMENT_ID '100' FOR SELECT * FROM t WHERE val2 = 100;
SQL> EXPLAIN PLAN SET STATEMENT_ID '103.5' FOR SELECT * FROM t WHERE val2 = 103.5;
SQL> EXPLAIN PLAN SET STATEMENT_ID '107' FOR SELECT * FROM t WHERE val2 = 107;
SQL> EXPLAIN PLAN SET STATEMENT_ID '109' FOR SELECT * FROM t WHERE val2 = 109;
SQL> EXPLAIN PLAN SET STATEMENT_ID '111' FOR SELECT * FROM t WHERE val2 = 111;

SQL> SELECT statement_id, cardinality
  2  FROM plan_table
  3  WHERE id = 0
  4  ORDER BY statement_id;

STATEMENT_ID CARDINALITY
------------ -----------
096                    1
098                   20
100                   40
103.5                 50
107                   40
109                   20
111                    1
```

Considering these essential characteristics of the two types of histograms, it's apparent that frequency histograms are more accurate than height-balanced histograms. The main problem with height-balanced histograms is not only that the precision is lower, but also that sometimes it might be by chance that a value is recognized as popular or not. For example, in the histogram illustrated in Figure 8-4, the split point between buckets 4 and 5 occurs very close to the point where the value changes from 105 to 106.

Therefore, even a small change in the data distribution might lead to a different histogram and to different estimations. The following example, where only 20 rows are updates (this is 2% of the total number of rows), illustrates such a case:

```
SQL> UPDATE t SET val2 = 105 WHERE val2 = 106 AND rownum <= 20;

SQL> REMARK at this point object statistics are gathered

SQL> SELECT endpoint_value, endpoint_number
  2  FROM user_tab_histograms
  3  WHERE table_name = 'T'
  4  AND column_name = 'VAL2'
  5  ORDER BY endpoint_number;

ENDPOINT_VALUE ENDPOINT_NUMBER
-------------- ---------------
           101               0
           104               1
           105               4
           106               5

SQL> EXPLAIN PLAN SET STATEMENT_ID '101' FOR SELECT * FROM t WHERE val2 = 101;
SQL> EXPLAIN PLAN SET STATEMENT_ID '102' FOR SELECT * FROM t WHERE val2 = 102;
SQL> EXPLAIN PLAN SET STATEMENT_ID '103' FOR SELECT * FROM t WHERE val2 = 103;
SQL> EXPLAIN PLAN SET STATEMENT_ID '104' FOR SELECT * FROM t WHERE val2 = 104;
SQL> EXPLAIN PLAN SET STATEMENT_ID '105' FOR SELECT * FROM t WHERE val2 = 105;
SQL> EXPLAIN PLAN SET STATEMENT_ID '106' FOR SELECT * FROM t WHERE val2 = 106;
SQL> SELECT statement_id, cardinality
  2  FROM plan_table
  3  WHERE id = 0;

STATEMENT_ID CARDINALITY
------------ -----------
101                   80
102                   80
103                   80
104                   80
105                  600
106                   80
```

Therefore, in practice, height-balanced histograms may not only be misleading but may also lead to instability in query optimizer estimations. For this reason, as of version 12.1, top frequency histograms and hybrid histograms replace height-balanced histograms.

## Top Frequency Histograms

One of the key characteristics of a frequency histogram is that every value is represented in the histogram. Even though they're accurate, because of the limit in the number of buckets, sometimes they can't be created. The concept behind top frequency histograms is that, in case some of the values represent a small percentage of the data, they can be safely discarded because they're statistically insignificant. And, if it's possible to discard enough values to void surpassing the limit of the number of buckets, a top frequency histogram, which is based only on the top-n values, may be created.

To determine whether a histogram with *n* buckets is sufficiently accurate, the database engine checks whether those *n* values represent at least *p* percent of the rows, where *p* is computed with Formula 8-1. For example, a top frequency histogram like the one gathered on the val3 column, with its 5 buckets, has to represent at least 80% (100 - 100/5) of the rows.

***Formula 8-1.*** Minimum percentage of rows that have to be represented by the top-n values

$$p = 100 - \frac{100}{n}$$

In the case of the val3 column, five buckets are sufficient because, as the output of the following query shows, the top-3 values already account for more than 80% of the rows:

```
SQL> SELECT val3, count(*) AS frequency, ratio_to_report(count(*)) OVER ()*100 AS percent
  2  FROM t
  3  GROUP BY val3
  4  ORDER BY val3;

      VAL3 FREQUENCY    PERCENT
---------- --------- ----------
       101         8        0.8
       102        25        2.5
       103        68        6.8
       104       185       18.5
       105       502       50.2
       106       212       21.2
```

The following is the histogram stored in the data dictionary for the val3 column:

```
SQL> SELECT endpoint_value, endpoint_number,
  2         endpoint_number - lag(endpoint_number,1,0)
  3                           OVER (ORDER BY endpoint_number) AS frequency
  4  FROM user_tab_histograms
  5  WHERE table_name = 'T'
  6  AND column_name = 'VAL3'
  7  ORDER BY endpoint_number;

ENDPOINT_VALUE ENDPOINT_NUMBER FREQUENCY
-------------- --------------- ---------
           101               1         1
           103              69        68
           104             254       185
           105             756       502
           106             968       212
```

Compared to the frequency histogram of the val2 column, there are two differences. First, the bucket for the value 102 doesn't exist. And that, even though the frequency of value 102 is higher than the frequency of value 101. In other words, the histogram doesn't represent the top-5 values. Second, the bucket for value 101, even though there are eight rows with that value, has endpoint_number equal to 1. The fact is that a histogram must always contain the minimum and maximum values. If, as in this case, one of the two values should be discarded because it's not part of the top-n values, another value is discarded (the one with the lowest frequency), and the frequency of the minimum/maximum value is set to 1. Note that after such an operation, the rule based on Formula 8-1 must be reevaluated.

The following example shows that, as you might expect, the estimations performed by the query optimizer with the top frequency histogram are different from the ones performed with the frequency histogram only for the values without frequency information (101 and 102):

```
SQL> EXPLAIN PLAN SET STATEMENT_ID '101' FOR SELECT * FROM t WHERE val3 = 101;
SQL> EXPLAIN PLAN SET STATEMENT_ID '102' FOR SELECT * FROM t WHERE val3 = 102;
SQL> EXPLAIN PLAN SET STATEMENT_ID '103' FOR SELECT * FROM t WHERE val3 = 103;
SQL> EXPLAIN PLAN SET STATEMENT_ID '104' FOR SELECT * FROM t WHERE val3 = 104;
SQL> EXPLAIN PLAN SET STATEMENT_ID '105' FOR SELECT * FROM t WHERE val3 = 105;
SQL> EXPLAIN PLAN SET STATEMENT_ID '106' FOR SELECT * FROM t WHERE val3 = 106;

SQL> SELECT statement_id, cardinality
  2  FROM plan_table
  3  WHERE id = 0
  4  ORDER BY statement_id;

STATEMENT_ID CARDINALITY
------------ -----------
101                   32
102                   32
103                   68
104                  185
105                  502
106                  212
```

Notice that for values 101 and 102, the frequency is the lowest frequency of all values represented in the histogram divided by two. Be aware that it is 32, and not 34 (68/2) as you might expect, because not all values are represented in the histogram.

Let's have a look at what happens when values not represented in the histogram are used. Simply put, it's the same as for the frequency histograms. The following example illustrates:

```
SQL> EXPLAIN PLAN SET STATEMENT_ID '096' FOR SELECT * FROM t WHERE val3 = 96;
SQL> EXPLAIN PLAN SET STATEMENT_ID '098' FOR SELECT * FROM t WHERE val3 = 98;
SQL> EXPLAIN PLAN SET STATEMENT_ID '100' FOR SELECT * FROM t WHERE val3 = 100;
SQL> EXPLAIN PLAN SET STATEMENT_ID '103.5' FOR SELECT * FROM t WHERE val3 = 103.5;
SQL> EXPLAIN PLAN SET STATEMENT_ID '107' FOR SELECT * FROM t WHERE val3 = 107;
SQL> EXPLAIN PLAN SET STATEMENT_ID '109' FOR SELECT * FROM t WHERE val3 = 109;
SQL> EXPLAIN PLAN SET STATEMENT_ID '111' FOR SELECT * FROM t WHERE val3 = 111;

SQL> SELECT statement_id, cardinality
  2  FROM plan_table
  3  WHERE id = 0
  4  ORDER BY statement_id;

STATEMENT_ID CARDINALITY
------------ -----------
096                    1
098                   13
100                   26
103.5                 32
107                   26
109                   13
111                    1
```

If the rule based on Formula 8-1 isn't fulfilled, and therefore, neither a frequency histogram nor a top frequency histogram can be built, the database engine creates a hybrid histogram.

## Hybrid Histograms

Hybrid histograms combine some of the characteristics of both frequency and height-balanced histograms. The process to build them starts in the same way as for height-balanced histograms. Then two important improvements take place:

- Every distinct value is associated to a single bucket (in other words, the concept of popular value as defined for height-balanced histograms no longer exists). For that purpose, the bucket's limits are shifted. As a result, every bucket may be based on a different number of rows.
- A frequency is added to the endpoint value of every bucket. Hence, for the endpoint values, and only for the endpoint values, a kind of frequency histogram is available.

The test table has two hybrid histograms. As an example, let's have a look at the one created for the `val1` column (note that it has 22 distinct values). The output of the following query shows the data set it contains:

```
SQL> SELECT val1, count(*), ratio_to_report(count(*)) OVER ()*100 AS percent
  2  FROM t
  3  GROUP BY val1
  4  ORDER BY val1;
```

```
      VAL1   COUNT(*) PERCENT
---------- ---------- -------
        39          2     0.2
        41          4     0.4
        42         13     1.3
        43         21     2.1
        44         26     2.6
        45         54     5.4
        46         66     6.6
        47         86     8.6
        48         81     8.1
        49         97     9.7
        50        102    10.2
        51        103    10.3
        52         80     8.0
        53         64     6.4
        54         76     7.6
        55         50     5.0
        56         30     3.0
        57         21     2.1
        58         12     1.2
        59          6     0.6
        60          5     0.5
        63          1     0.1
```

If the database engine is instructed to create a histogram with ten buckets, neither a frequency histogram nor a top frequency histogram can be used. The former isn't applicable because the number of buckets is greater than the number of distict values. The latter isn't applicable because the top-10 values represent only about 80% of the rows (according to Formula 8-1, 90% is required; 90 = 100 – 100/10). As a result, a hybrid histogram that provides the following information is built:

```
SQL> SELECT endpoint_value, endpoint_number,
  2         endpoint_number - lag(endpoint_number,1,0)
  3                           OVER (ORDER BY endpoint_number) AS count,
  4         endpoint_repeat_count
  5  FROM user_tab_histograms
  6  WHERE table_name = 'T'
  7  AND column_name = 'VAL1'
  8  ORDER BY endpoint_number;

ENDPOINT_VALUE ENDPOINT_NUMBER      COUNT ENDPOINT_REPEAT_COUNT
-------------- --------------- ---------- ---------------------
            39               2          2                     2
            44              66         64                    26
            45             120         54                    54
            46             186         66                    66
            47             272         86                    86
            48             353         81                    81
            49             450         97                    97
            50             552        102                   102
            51             655        103                   103
```

```
        52            735         80                     80
        53            799         64                     64
        54            875         76                     76
        56            955         80                     30
        59            994         39                      6
        63           1000          6                      1
```

In the preceding output, notice that on the one hand, the `endpoint_number` column provides information about the number of rows associated to a bucket, and on the other hand, the `endpoint_repeat_count` column provides the frequency of the endpoint value. Based on this information, the estimations performed by the query optimizer for the endpoint values can be accurate. Here's an example:

```
SQL> EXPLAIN PLAN SET STATEMENT_ID '44' FOR SELECT * FROM t WHERE val1 = 44;
SQL> EXPLAIN PLAN SET STATEMENT_ID '50' FOR SELECT * FROM t WHERE val1 = 50;
SQL> EXPLAIN PLAN SET STATEMENT_ID '56' FOR SELECT * FROM t WHERE val1 = 56;

SQL> SELECT statement_id, cardinality
  2  FROM plan_table
  3  WHERE id = 0
  4  ORDER BY statement_id;

STATEMENT_ID CARDINALITY
------------ -----------
44                    26
50                   102
56                    30
```

■ **Tip** The information provided by an hybrid histogram is much better than that provided by a height-balanced histogram. For this reason, from version 12.1 onward, height-balanced histograms can and should be completely avoided.

## No Histogram

It's useful to note that the `user_tab_histograms` view shows two rows for each column without a histogram. This is because the minimum and maximum values are stored in endpoint numbers 0 and 1, respectively. For example, the content for the `id` column, which has no histogram, is the following:

```
SQL> SELECT endpoint_value, endpoint_number
  2  FROM user_tab_histograms
  3  WHERE table_name = 'T'
  4  AND column_name = 'ID';

ENDPOINT_VALUE ENDPOINT_NUMBER
-------------- ---------------
             1               0
          1000               1
```

## Extended Statistics

The column statistics and histograms described in the previous sections are helpful only when column values are used without being modified in predicates. For example, if the predicate `country='Switzerland'` is used, with column statistics and a histogram in place for the `country` column, the query optimizer should be able to correctly estimate its selectivity. This is because column statistics and the histogram describe the values of the `country` column itself. On the other hand, if the predicate `upper(country)='SWITZERLAND'` is used, the query optimizer is no longer able to directly infer the selectivity from the object statistics and the histogram. A similar problem occurs when a predicate references several columns. For example, if I apply the predicate `country='Denmark' AND language='Danish'` to a table containing people from all over the world, it's likely that the two restrictions apply to the same rows for most rows in such a table. In fact, most people speaking Danish live in Denmark, and most people living in Denmark speak Danish. In other words, the two restrictions are almost redundant. Such columns are commonly called *correlated columns* and challenge the query optimizer. This is because no object statistics or histograms describe such a dependency between data—or put another way, the query optimizer actually assumes that data stored in different columns isn't interdependent.

As of version 11.1, it's possible to gather object statistics and histograms on expressions or on groups of columns to solve these kinds of problems. These new statistics are called *extended statistics*. Basically, what happens is that a hidden column, called an *extension*, is created, based on either an expression or a group of columns. Then regular object statistics and histograms are gathered on it.

The definition is carried out with the `create_extended_stats` function of the `dbms_stats` package. For example, two extensions are created with the following query. The first one is on `upper(pad)`, and the second one is a column group made up of the columns `val2` and `val3`. In the test table, these contain exactly the same values; in other words, the columns are highly (actually, perfectly) correlated. For the definition, as shown next, the expression or group of columns must be enclosed in parentheses. Notice that the function returns a system-generated name for the extension (a 30-byte name starting with `SYS_STU`):

```
SQL> SELECT dbms_stats.create_extended_stats(ownname   => user,
  2                                          tabname   => 'T',
  3                                          extension => '(upper(pad))') AS ext1,
  4         dbms_stats.create_extended_stats(ownname   => user,
  5                                          tabname   => 'T',
  6                                          extension => '(val2,val3)') AS ext2
  7  FROM dual;

EXT1                           EXT2
------------------------------ ------------------------------
SYS_STUOKSQX64#IO1CKJ5FPGFK3W9 SYS_STUPS77EFBJCOTDFMHM8CHP7Q1
```

---

■ **Note** A group of columns can't reference expressions or virtual columns.

---

Obviously, once the extensions are created, the data dictionary provides information about them. The following query, based on the `user_stat_extensions` view, shows the existing extensions of the test table. There are `dba`, `all`, and, in a 12.1 multitenant environment, `cdb` versions as well:

```
SQL> SELECT extension_name, extension
  2  FROM user_stat_extensions
  3  WHERE table_name = 'T';
```

```
EXTENSION_NAME                 EXTENSION
------------------------------ ---------------
SYS_STU0KSQX64#I01CKJ5FPGFK3W9 (UPPER("PAD"))
SYS_STUPS77EFBJCOTDFMHM8CHP7Q1 ("VAL2","VAL3")
```

As shown in the output of the next query, the hidden columns have the same name as the extensions. Also notice how the definition of the extension is added to the column:

```
SQL> SELECT column_name, data_type, hidden_column, data_default
  2  FROM user_tab_cols
  3  WHERE table_name = 'T'
  4  ORDER BY column_id;

COLUMN_NAME                    DATA_TYPE HIDDEN DATA_DEFAULT
------------------------------ --------- ------ -----------------------------------
ID                             NUMBER    NO
VAL1                           NUMBER    NO
VAL2                           NUMBER    NO
VAL3                           NUMBER    NO
PAD                            VARCHAR2  NO
SYS_STU0KSQX64#I01CKJ5FPGFK3W9 VARCHAR2  YES    UPPER("PAD")
SYS_STUPS77EFBJCOTDFMHM8CHP7Q1 NUMBER    YES    SYS_OP_COMBINED_HASH("VAL2","VAL3")
```

■ **Caution** Because the extended statistics for a group of columns are based on a hash function (`sys_op_combined_hash`), they work only with predicates based on equality. In other words, the query optimizer can't take advantage of them for predicates based on operators like `BETWEEN` and `<` or `>`. Extended statistics for a group of columns are also used to estimate the cardinality of `GROUP BY` clauses and, from version 11.2.0.3 onward, for the `DISTINCT` operator in `SELECT` clauses.

To drop an extension, the `dbms_stats` package provides you with the `drop_extended_stats` procedure. In the following example, the PL/SQL block drops the two extensions previously created:

```
BEGIN
  dbms_stats.drop_extended_stats(ownname   => user,
                                 tabname   => 'T',
                                 extension => '(upper(pad))');
  dbms_stats.drop_extended_stats(ownname   => user,
                                 tabname   => 'T',
                                 extension => '(val2,val3)');
END;
```

It's not necessarily a trivial thing deciding which group of columns it's sensible to create an extension on. The following approach can be used (a full example is available in the `seed_col_usage.sql` script) as of version 11.2.0.2:

1. Invoke the `seed_col_usage` procedure of the `dbms_stats` package to instruct the query optimizer to record information about predicates specified in `WHERE` clauses, about columns referenced in `GROUP BY` clauses, and, from version 11.2.0.3 onward, about the `DISTINCT` operator in `SELECT` clauses. The recording is done either for all SQL statements of the SQL tuning set specified in the `sqlset_name` and `owner_name` parameters or, as shown in the following example, for all SQL statements that are hard parsed (no execution is needed, hence, an `EXPLAIN PLAN` statement is sufficient) over a period of time specified in seconds by the `time_limit` parameter:

```
SQL> BEGIN
  2    dbms_stats.seed_col_usage(sqlset_name => NULL,
  3                              owner_name  => NULL,
  4                              time_limit  => 30);
  5  END;
  6  /
```

2. Once the recording is over, invoke the `report_col_usage` function of the `dbms_stats` package to report the columns' usage. Each column's utilization pattern is reported. For example, in the following output, the `val1` and `val2` columns were part of a single-table predicate based on equalities:

```
SQL> SELECT dbms_stats.report_col_usage(ownname => user, tabname => 't')
  2  FROM dual;

DBMS_STATS.REPORT_COL_USAGE(OWNNAME=>USER,TABNAME=>'T')
------------------------------------------------------------------------
LEGEND:
.......

EQ         : Used in single table EQuality predicate
RANGE      : Used in single table RANGE predicate
LIKE       : Used in single table LIKE predicate
NULL       : Used in single table is (not) NULL predicate
EQ_JOIN    : Used in EQuality JOIN predicate
NONEQ_JOIN : Used in NON EQuality JOIN predicate
FILTER     : Used in single table FILTER predicate
JOIN       : Used in JOIN predicate
GROUP_BY   : Used in GROUP BY expression
........................................................................
```

```
###############################################################################

COLUMN USAGE REPORT FOR CHRIS.T
...............................

1. VAL1                              : EQ
2. VAL2                              : EQ
3. VAL3                              : EQ
4. (VAL1, VAL2)                      : FILTER
5. (VAL1, VAL3)                      : FILTER
6. (VAL2, VAL3)                      : GROUP_BY
###############################################################################
```

3. Create extensions using the `create_extended_stats` procedure of the `dbms_stats` package. Note that if the definition of the extensions themselves isn't passed as a parameter, the definition is taken from the information stored during the recording. Therefore, only the schema and table names are required. Notice how, in the following example, three extensions are created with a single call to `create_extended_stats`:

```
SQL> SELECT dbms_stats.create_extended_stats(ownname => user, tabname => 't')
  2  FROM dual;

DBMS_STATS.CREATE_EXTENDED_STATS(OWNNAME=>USER,TABNAME=>'T')
-------------------------------------------------------------------------------
###############################################################################

EXTENSIONS FOR CHRIS.T
......................

1. (VAL1, VAL2)                      : SYS_STU4K1K3JNH1Z9#_L_V93K3DT4 created
2. (VAL1, VAL3)                      : SYS_STUS574STTDWYBF6PGQN#XHGGJ created
3. (VAL2, VAL3)                      : SYS_STUPS77EFBJCOTDFMHM8CHP7Q1 created
###############################################################################
```

4. After creating the extensions, regather the object statistics of the modified table.

In version 12.1, extensions can also be automatically created by the database engine. In fact, for SQL statements that take advantage of statistics feedback, the query optimizer can create a SQL plan directive whose aim is to instruct the database engine to create an extension. As a result, future reoptimization due to statistics feedback can be avoided. A full example is available in the `seed_col_usage.sql` script. There are two essential things to be aware of. First, extensions can and will be automatically created. Second, the extensions are created only when object statistics are gathered. In other words, the interval between the creation of the SQL plan directive and the creation of the extension depends on the frequency of the object statistics gathering.

It's interesting to note that extended statistics are based on another feature, introduced in version 11.1, called *virtual columns*. A virtual column is a column that doesn't store data but simply generates its content with an expression based on other columns. This is helpful in case an application makes frequent usage of given expressions. A typical example is applying the `upper` function to a `VARCHAR2` column or the `trunc` function to a `DATE` column. If these expressions are frequently used, it makes sense to define them directly in the table as shown in the following example:

```
SQL> CREATE TABLE persons (
  2    name VARCHAR2(100),
  3    name_upper AS (upper(name))
  4  );
```

```
SQL> INSERT INTO persons (name) VALUES ('Michelle');

SQL> SELECT name
  2  FROM persons
  3  WHERE name_upper = 'MICHELLE';

NAME
----------
Michelle
```

As you will see in Chapter 13, virtual columns can also be indexed.

The main issue of virtual columns, compared to extended statistics, is that they change the behavior of some SQL statements (for examples, of `SELECT *` statements and `INSERT` statements without the column list). In other words, because extended statistics are based on hidden columns, they are completely transparent to the application.

It's important to recognize that, independently of how virtual columns are defined (either explicitly by the user or implicitly through extended statistics), object statistics and histograms are normally gathered on them. This way, the query optimizer gets additional statistics about the data.

## SQL PLAN DIRECTIVES

SQL plan directives are a new concept introduced in version 12.1. Their purpose is to help the query optimizer cope with misestimates. To do so, they store in the data dictionary information about the expressions that cause misestimates. Because they aren't associated to a specific SQL statement, not only can several of them be used for a single SQL statement, but, in addition, a single SQL plan directive can be applied to multiple SQL statements.

In some cases, SQL plan directives instruct the database engine to automatically create extended statistics (specifically, column groups). When extended statistics can't be created, they instruct the query optimizer to use dynamic sampling.

SQL plan directives are enabled when the `optimizer_adaptive_features` initialization parameter is set to `TRUE` (this is the default value). When activated, the database engine automatically maintains (for example, creates and purges) SQL plan directives. Some management operations can also be executed manually through the `dbms_spd` package.

Information about the available SQL plan directives is provided through the `dba_sql_plan_directives` and `dba_sql_plan_dir_objects` views (`cdb` versions of these views also exist).

## Index Statistics

Before describing index statistics, let's briefly review the structure of an index based on Figure 8-5. The block at the top is called the *root block*. This is the block where every lookup starts from. The root block references the *branch blocks*. Note that the root block is also considered a branch block. Each branch block in turn references either another level of branch blocks or, as in Figure 8-5, the *leaf blocks*. The leaf blocks store the keys (in this case, some numeric values between 6 and 89) and the rowids that reference the data. For a given index, there are always the same number of branch blocks between the root block and every leaf block. In other words, the index is always balanced. Note that to support efficient lookups over ranges of values (for example, all values between 25 and 45), the leaf blocks are chained.

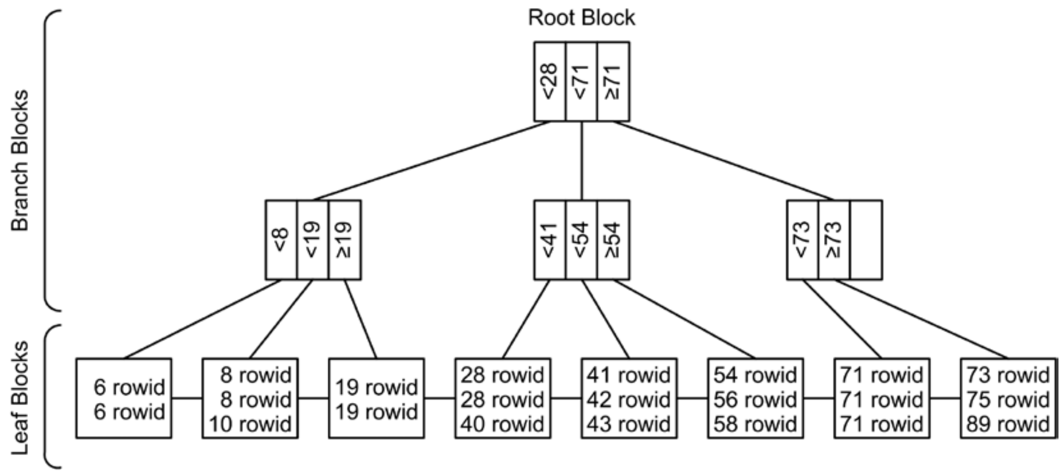


***Figure 8-5.*** *The structure of an index is based on a B+-tree*

Not all indexes have the three types of blocks. In fact, the branch blocks exist only if the root block isn't able to store the references of all the leaf blocks. In addition, if the index is very small, it consists of a single block containing all the data usually stored in the root block and the leaf blocks.

The following query shows how to get the most important index statistics for a table:

```
SQL> SELECT index_name AS name,
  2         blevel,
  3         leaf_blocks AS leaf_blks,
  4         distinct_keys AS dst_keys,
  5         num_rows,
  6         clustering_factor AS clust_fact,
  7         avg_leaf_blocks_per_key AS leaf_per_key,
  8         avg_data_blocks_per_key AS data_per_key
  9  FROM user_ind_statistics
 10  WHERE table_name = 'T';

NAME     BLEVEL LEAF_BLKS DST_KEYS NUM_ROWS CLUST_FACT LEAF_PER_KEY DATA_PER_KEY
-------- ------ --------- -------- -------- ---------- ------------ ------------
T_PK          1         2     1000     1000        979            1            1
T_VAL1_I      1         2      431      497        478            1            1
T_VAL2_I      1         3        6     1000        175            1           29
```

The index statistics returned by this query are as follows:

- `blevel` is the number of branch blocks to be read, including the root block, in order to access a leaf block.
- `leaf_blocks` is the number of leaf blocks of the index.
- `distinct_keys` is the number of distinct keys in the index.
- `num_rows` is the number of keys in the index. For primary keys, this is the same as `distinct_keys`.

- `clustering_factor` indicates how many adjacent index entries don't refer to the same data block in the table. If the table and the index are sorted similarly, the clustering factor is low. The minimum value is the number of nonempty data blocks in the table. If the table and the index are sorted differently, the clustering factor is high. The maximum value is the number of keys in the index. I discuss the way it's computed and its performance impact in detail in Chapter 13. It's worth mentioning that for bitmap indexes, no real clustering factor is computed. Actually, it's set to the number of keys in the index.
- `avg_leaf_blocks_per_key` is the average number of leaf blocks that store a single key. This value is derived from other statistics using Formula 8-2.

***Formula 8-2.*** Computation of the average number of leaf blocks that store a single key

$$avg\_leaf\_blocks\_per\_key \approx \frac{leaf\_blocks}{distinct\_keys}$$

- `avg_data_blocks_per_key` is the average number of data blocks in the table referenced by a single key. This value is derived from other statistics using Formula 8-3.

***Formula 8-3.*** Computation of the average number of data blocks referenced by a single key

$$avg\_data\_blocks\_per\_key \approx \frac{clustering\_factor}{distinct\_keys}$$

## Statistics for Partitioned Objects

Partitioned objects are logical constructs composed of sets of segments. For example, the following SQL statement creates a partitioned table with 16 segments, as shown in Figure 8-6. While the 16 segments are objects that actually hold data in a tablespace, the four partitions and the table are metadata-only objects. They only exist in the data dictionary:

```
CREATE TABLE t (id NUMBER, tstamp DATE, pad VARCHAR2(1000))
PARTITION BY RANGE (tstamp)
SUBPARTITION BY HASH (id)
SUBPARTITION TEMPLATE
(
  SUBPARTITION sp1,
  SUBPARTITION sp2,
  SUBPARTITION sp3,
  SUBPARTITION sp4
)
(
  PARTITION q1 VALUES LESS THAN (to_date('2014-04-01','YYYY-MM-DD')),
  PARTITION q2 VALUES LESS THAN (to_date('2014-07-01','YYYY-MM-DD')),
  PARTITION q3 VALUES LESS THAN (to_date('2014-10-01','YYYY-MM-DD')),
  PARTITION q4 VALUES LESS THAN (to_date('2015-01-01','YYYY-MM-DD'))
)
```

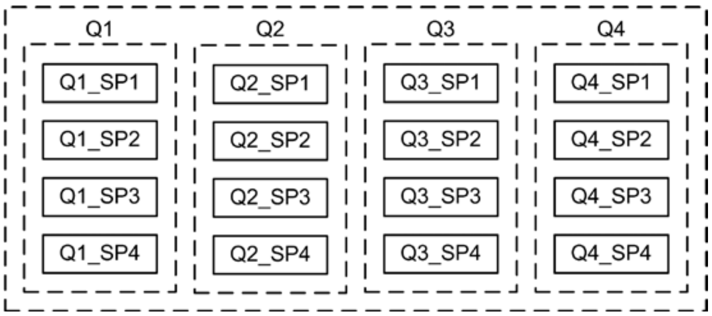


***Figure 8-6.*** *Range-hash partitioned table with 16 segments*

For partitioned objects, the database engine is able to handle all object statistics discussed in the previous sections (in other words, table statistics, column statistics, histograms and index statistics) at the table/index-level as well as at the partition and subpartition levels. Having object statistics at all levels is useful because, depending on the SQL statement to be processed, the query optimizer considers the object statistics that most closely describe the segments to be accessed. Simply put, the query optimizer uses the partition and subpartition statistics only when, during the parse phase, it can determine whether a specific partition or subpartition is accessed. Otherwise, the query optimizer generally uses the table/index-level statistics. (There are some situations in which the query optimizer uses, at the same time, table/index statistics *as well as* partition and subpartition statistics).

## Gathering Object Statistics

To gather object statistics, the dbms_stats package contains several procedures. There are several procedures because, depending on the situation, the process of gathering object statistics should occur for the whole database, for the data dictionary, for a schema, or for a single object:

- gather_database_stats gathers object statistics for a whole database.
- gather_dictionary_stats gathers object statistics for the data dictionary. Note that the data dictionary isn't only composed of the objects stored in the sys schema, but also includes the other schemas installed by Oracle for optional components.
- gather_fixed_objects_stats gathers object statistics for particular objects called *fixed tables* (also known as x$ *tables*) and *fixed indexes* that are part of the data dictionary. The fixed tables, which are commonly used in dynamic performance views, are in-memory structures only. For this reason, they require special handling. To know which tables are relevant for this procedure, you can use the following query. Note that object statistics aren't gathered for every fixed table, though:

```
SELECT name
FROM v$fixed_table
WHERE type = 'TABLE'
```

- gather_schema_stats gathers object statistics for a whole schema.
- gather_table_stats gathers object statistics for one table including its columns and, optionally, for its indexes.
- gather_index_stats gathers object statistics for one index.

■ **Note** The `dbms_stats` package isn't the only feature that gathers object statistics. In fact, the `CREATE INDEX` and `ALTER INDEX` statements automatically gather object statistics while building an index. In addition, from version 12.1 onward, CTAS statements and direct-path inserts into empty tables automatically gather object statistics. Be aware that the object statistics computed by the `dbms_stats` package can be superior to ones that are automatically gathered. Therefore, you can't always rely, in every case, on the automatically gathered statistics.

The procedures provided by the `dbms_stats` package take several parameters that can be grouped into three main categories. With the first group you specify target objects, with the second group you specify gathering options, and with the third group you specify whether to back up the current statistics before overwriting them. Table 8-3 summarizes which parameter is available with which procedure. The next three sections describe the scope and use of each parameter in detail.

***Table 8-3.*** *Parameters of the Procedures Used for Gathering Object Statistics*

| Parameter | Database | Dictionary | Fixed Objects | Schema | Table | Index |
|---|---|---|---|---|---|---|
| **Target Objects** | | | | | | |
| `ownname` | | | | ✓ | ✓ | ✓ |
| `indname` | | | | | | ✓ |
| `tabname` | | | | | ✓ | |
| `partname` | | | | | ✓ | ✓ |
| `comp_id` | | ✓ | | | | |
| `granularity` | ✓ | ✓ | | ✓ | ✓ | ✓ |
| `cascade` | ✓ | ✓ | | ✓ | ✓ | |
| `gather_fixed` | ✓ | | | ✓ | | |
| `gather_sys` | ✓ | | | | | |
| `gather_temp` | ✓ | | | ✓ | | |
| `options` | ✓ | ✓ | | ✓ | ✓* | |
| `objlist` | ✓ | ✓ | | ✓ | | |
| `force` | | | | ✓ | ✓ | ✓ |
| `obj_filter_list` | ✓ | ✓ | | ✓ | | |
| **Gathering Options** | | | | | | |
| `estimate_percent` | ✓ | ✓ | | ✓ | ✓ | ✓ |
| `block_sample` | ✓ | ✓ | | ✓ | ✓ | |
| `method_opt` | ✓ | ✓ | | ✓ | ✓ | |
| `degree` | ✓ | ✓ | | ✓ | ✓ | ✓ |
| `no_invalidate` | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |

(*continued*)

***Table 8-3.*** (*continued*)

| Parameter | Database | Dictionary | Fixed Objects | Schema | Table | Index |
|---|---|---|---|---|---|---|
| **Backup Table** | | | | | | |
| `stattab` | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| `statid` | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| `statown` | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |

**Available as of version 12.1.*

## Target Objects

Target object parameters specify which objects you gather object statistics for:

- `ownname` specifies the name of the schema to be processed. This parameter is mandatory.
- `indname` specifies the name of the index to be processed. This parameter is mandatory.
- `tabname` specifies the name of the table to be processed. This parameter is mandatory.
- `partname` specifies the name of the partition or subpartition to be processed. If no value is specified, object statistics for all partitions and subpartitions might be gathered, depending on the value of the `granularity` parameter (see below). The default value is `NULL`.
- `comp_id` specifies the ID of the component to be processed. Because the ID of a component can't be used for gathering statistics, it's transformed internally into a list of schema. To know which schemas are processed for a given component, you can use the following query.[1] Note that the output of the query depends on several factors, such as the version and the actual components that are installed. Independently of this parameter, `sys` and `system` schemas are always processed. If an invalid value is specified, no error message is returned, and the `sys` and `system` schemas are processed regularly. With the default value `NULL`, all components are processed:

```
SQL> SELECT u.username AS schema_name, r.cid AS comp_id, r.cname AS comp_name
  2  FROM dba_users u,
  3       (SELECT schema#, cid, cname
  4        FROM sys.registry$
  5        WHERE status IN (1, 3, 5)
  6        AND namespace = 'SERVER'
  7        UNION ALL
  8        SELECT s.schema#, s.cid, cname
  9        FROM sys.registry$ r, sys.registry$schemas s
 10        WHERE r.status IN (1,3,5)
 11        AND r.namespace = 'SERVER'
 12        AND r.cid = s.cid) r
 13  WHERE u.user_id = r.schema#
 14  ORDER BY r.cid, u.username;
```

[1]Unfortunately, Oracle doesn't make all necessary information available through data dictionary views. As a result, this query is based on internal tables. The `select any dictionary` system privilege provides access to the necessary tables.

```
SCHEMA_NAME        COMP_ID COMP_NAME
------------------ ------- ----------------------------------
SYS                APS     OLAP Analytic Workspace
SYS                CATALOG Oracle Database Catalog Views
SYS                CATJAVA Oracle Database Java Packages
APPQOSSYS          CATPROC Oracle Database Packages and Types
DBSNMP             CATPROC Oracle Database Packages and Types
DIP                CATPROC Oracle Database Packages and Types
GSMADMIN_INTERNAL  CATPROC Oracle Database Packages and Types
ORACLE_OCM         CATPROC Oracle Database Packages and Types
OUTLN              CATPROC Oracle Database Packages and Types
SYS                CATPROC Oracle Database Packages and Types
SYSTEM             CATPROC Oracle Database Packages and Types
CTXSYS             CONTEXT Oracle Text
SYS                JAVAVM  JServer JAVA Virtual Machine
LBACSYS            OLS     Oracle Label Security
MDSYS              ORDIM   Oracle Multimedia
ORDDATA            ORDIM   Oracle Multimedia
ORDPLUGINS         ORDIM   Oracle Multimedia
ORDSYS             ORDIM   Oracle Multimedia
SI_INFORMTN_SCHEMA ORDIM   Oracle Multimedia
WMSYS              OWM     Oracle Workspace Manager
MDSYS              SDO     Spatial
ANONYMOUS          XDB     Oracle XML Database
XDB                XDB     Oracle XML Database
XS$NULL            XDB     Oracle XML Database
SYS                XML     Oracle XDK
SYS                XOQ     Oracle OLAP API
```

- `granularity` specifies at which level statistics for partitioned objects are processed. This parameter accepts the value listed in Table 8-4. The default value is `auto` (this default value can be changed—see the "Configuring the `dbms_stats` Package" section later in this chapter). For additional information about managing object statistics for partitioned objects, refer to the "Working with Partitioned Objects" section later in this chapter.

***Table 8-4.*** *Values Accepted by the `granularity` Parameter*

| Value | Meaning |
|---|---|
| `all` | Table/index, partition, and subpartition statistics are gathered. |
| `auto` | Table/index and partition statistics are gathered. Subpartition statistics are gathered only if the table is subpartitioned by list or range. |
| `global` | Only table/index statistics are gathered. |
| `global and partition` | Table/index and partition statistics are gathered. |
| `approx_global and partition` | Similar to `global and partition`, but uses derived statistics at the table/index level. Available in version 10.2.0.5 and from version 11.1.0.7 onward. |
| `partition` | Only partition statistics are gathered. |
| `subpartition` | Only subpartition statistics are gathered. |

- cascade specifies whether indexes are processed. This parameter accepts the values TRUE, FALSE, and dbms_stats.auto_cascade. The latter, which is a constant evaluating to NULL, lets the database engine decide whether to gather the index statistics. The default value is dbms_stats.auto_cascade (this default value can be changed—see the section "Configuring the dbms_stats Package" later in this chapter).
- gather_fixed specifies whether object statistics for fixed tables are gathered. This parameter accepts the values TRUE and FALSE. The default value is FALSE.
- gather_sys specifies whether the sys schema is processed. This parameter accepts the values TRUE and FALSE. The default value is FALSE.
- gather_temp specifies whether temporary tables are processed. This parameter accepts the values TRUE and FALSE. The default value is FALSE. Refer to the "Working with Global Temporary Tables" section for additional information.
- options specifies which, and whether, objects are processed. This parameter accepts the value listed in Table 8-5. However, when used with the gather_table_stats procedure, only gather and gather auto are supported. The default value is gather.

***Table 8-5.*** *Values Accepted by the options Parameter*

| Value | Meaning |
|---|---|
| gather | All objects are processed. |
| gather auto | Let the procedure determine not only which objects are to be processed but also how they're processed. When this value is *not* used with the gather_table_stats procedure, all parameters except ownname, objlist, stattab, statid, and statown are ignored. |
| gather stale | Only objects having stale object statistics are processed. Be careful: objects without object statistics aren't considered stale. |
| gather empty | Only objects without object statistics are processed. |
| list auto | Lists objects that would be processed with the gather auto option. |
| list stale | Lists objects that would be processed with the gather stale option. |
| list empty | Lists objects that would be processed with the gather empty option. |

## STALENESS OF OBJECT STATISTICS

To recognize whether object statistics are stale, the database engine counts (approximately), for each table, the number of rows modified through SQL statements. The result of that counting is externalized through the data dictionary views all_tab_modifications, dba_tab_modifications (this view exists as of version 11.2 only), user_tab_modifications, and, in a 12.1 multitenant environment, cdb_tab_modifications. The following query is an example:

```
SQL> SELECT inserts, updates, deletes, truncated
  2  FROM user_tab_modifications
  3  WHERE table_name = 'T';
```

```
INSERTS UPDATES DELETES TRUNCATED
------- ------- ------- ---------
    775   14200      66 NO
```

Based on this information, the `dbms_stats` package is able to determine whether the object statistics associated with a specific object are stale. In version 10.2, object statistics are considered stale if at least 10 percent of the rows have been modified. As of version 11.1, you can configure the threshold through the `stale_percent` preference. Its default value is 10 percent. Later in this chapter, the section “Configuring the dbms_stats Package” shows how to change it.

Be careful, because in version 10.2.0.5, 11.2.0.1, and 11.2.0.2 rows loaded into an empty table through Data Pump are incorrectly counted as regular inserts. As a result, after an import, the object statistics are considered stale.

Counting is controlled databasewide by the `statistics_level` initialization parameter. If it’s set to either `typical` (which is the default value) or `all`, counting is enabled.

- `objlist` returns, depending on the value of the `options` parameter, the list of objects that were processed or that would be processed. This is an output parameter based on a type defined in the `dbms_stats` package. For instance, the following PL/SQL block shows how to display the list of processed objects:

```
SQL> DECLARE
  2    l_objlist dbms_stats.objecttab;
  3    l_index PLS_INTEGER;
  4  BEGIN
  5    dbms_stats.gather_schema_stats(ownname => 'HR',
  6                                   objlist => l_objlist);
  7    l_index := l_objlist.FIRST;
  8    WHILE l_index IS NOT NULL
  9    LOOP
 10      dbms_output.put(l_objlist(l_index).ownname || '.');
 11      dbms_output.put_line(l_objlist(l_index).objname);
 12      l_index := l_objlist.next(l_index);
 13    END LOOP;
 14  END;
 15  /
HR.COUNTRIES
HR.DEPARTMENTS
HR.EMPLOYEES
HR.JOBS
HR.JOB_HISTORY
HR.LOCATIONS
HR.REGIONS
```

- `force` specifies whether locked statistics are overwritten. If this parameter is set to `FALSE` while a procedure that’s designed to process a single table or index is being executed on locked statistics, an error (ORA-20005) is raised. This parameter accepts the values `TRUE` and `FALSE`. You find more information about locked statistics in the section “Locking Object Statistics” later in this chapter.

- `obj_filter_list` specifies to gather statistics only for objects fulfilling at least one of the filters passed as a parameter. It's based on the `objecttab` type defined in the `dbms_stats` package itself and is available as of version 11.1 only. The following PL/SQL block shows how to gather statistics for all tables of the HR schema and all tables of the SH schema that have a name starting with the letter C:

```
DECLARE
  l_filter dbms_stats.objecttab := dbms_stats.objecttab();
BEGIN
  l_filter.extend(2);
  l_filter(1).ownname := 'HR';
  l_filter(2).ownname := 'SH';
  l_filter(2).objname := 'C%';
  dbms_stats.gather_database_stats(obj_filter_list => l_filter,
                                   options         => 'gather');
END;
```

## Gathering Options

The gathering option parameters listed in Table 8-2 specify how the gathering of statistics takes place, which kinds of column statistics are gathered, and whether dependent SQL cursors are invalidated. The options are as follows:

- `estimate_percent` specifies whether sampling is used for gathering statistics. Valid values are decimal numbers between 0.000001 and 100. The value 100, as well as the value `NULL`, means no sampling. The `dbms_stats.auto_sample_size` constant, which is the default value (this default value can be changed—see the section "Configuring the dbms_stats Package" later in this chapter), lets the procedure determine the sample size. As of version 11.1, it's recommended to use this value. In fact, in most cases, using the default value not only gathers statistics that are more accurate than a sampling of, say, 10%, but it results in their being gathered more quickly. This behavior is possible because version 11.1 introduces a completely new algorithm used only when `dbms_stats.auto_sample_size` is specified. Also note that, because this new algorithm requires a full scan of the table on which the statistics are gathered, on a system with a relatively slow disk I/O subsystem that might take too long. Also note that some features (top frequency histograms, hybrid histograms, and incremental statistics) only work when `dbms_stats.auto_sample_size` is specified. It's important to understand that when a decimal number is passed as parameter, the value specified by the `estimate_percent` parameter is only the minimum percentage used for gathering statistics. In fact, as shown in the following example, the `dbms_stats` package may automatically increase the specified `estimate_percent` value if the package considers the parameter's value to be too small. Provided you don't work with `dbms_stats.auto_sample_size` , to speed up the gathering of object statistics use small estimate percentages; values less than 10 percent are usually good. For large tables, even 0.5 percent, 0.1 percent, or less could be fine. The actual optimal value depends on data distribution. If you're uncertain about your choice, simply try different estimate percentages and compare the gathered statistics. In that way, you may find the best compromise between performance and accuracy. Note that small estimate percentages might not generate deterministic statistics. Because values that are too small are automatically increased if the gathering of statistics is performed at the database or schema level, the estimate percentage should be chosen for the biggest tables. In passing, note that sampling on external tables isn't supported:

```
SQL> BEGIN
  2    dbms_stats.gather_schema_stats(ownname          => user,
  3                                   estimate_percent => 0.5);
  4  END;
  5  /

SQL> SELECT table_name, sample_size, num_rows,
  2         round(sample_size/num_rows*100,1) AS "%"
  3  FROM user_tables
  4  WHERE num_rows > 0
  5  ORDER BY table_name;

TABLE_NAME             SAMPLE_SIZE NUM_ROWS      %
---------------------- ----------- -------- ------
CAL_MONTH_SALES_MV              48       48  100.0
CHANNELS                         5        5  100.0
COSTS                         4975    81391    6.1
COUNTRIES                       23       23  100.0
CUSTOMERS                     5435    55002    9.9
FWEEK_PSCAT_SALES_MV          4742    11001   43.1
PRODUCTS                        72       72  100.0
PROMOTIONS                     503      503  100.0
SALES                         4639   927800    0.5
SALES_TRANSACTIONS_EXT      916039   916039  100.0
TIMES                         1826     1826  100.0
```

- `block_sample` specifies whether row sampling or block sampling is used for the gathering of statistics. Although row sampling is more accurate, block sampling is faster. Therefore, block sampling should be used only when it's sure that data is randomly distributed. This parameter accepts the values `TRUE` and `FALSE`. The default value is `FALSE`. As of version 11.1, this parameter is only pertinent when the `estimate_percent` parameter isn't set to `dbms_stats.auto_sample_size`.
- `method_opt` specifies whether and how column statistics and histograms are gathered. There are three typical use cases:[2]
    - Gathering column statistics and histograms for all[3] columns. All histograms are created with the very same value for the `size_clause` parameter. If `size 1` is specified, no histograms are created. The syntax is shown in Figure 8-7. For example, with the `for all columns size 254` value, a histogram with up to 254 buckets is created for every column.

[2]For simplicity, I'm not describing all the possibilities because many of them are redundant or of limited use in practice.
[3]Actually, with the options *indexed* and *hidden*, it's possible to restrict the statistics gathering to indexed and hidden columns only. As a rule, object statistics should be available for all columns. For this reason, both these options should be avoided (hence they're shown in Figure 8-7 in gray). If for some columns there's no need to have object statistics, you should use the syntax described in Figure 8-8 instead. One sensible reason for using *hidden* is to gather statistics on a virtual column used for an extension that was just added to a table.

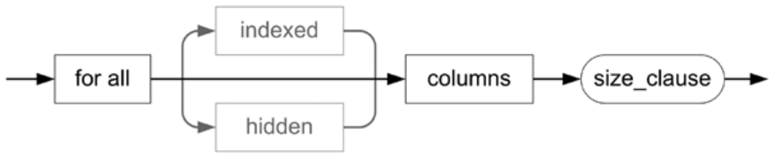


***Figure 8-7.*** *Gathering column statistics and histograms for all columns with a single value for the `size_clause` parameter (see Table 8-5)*

- Gathering column statistics on all columns and histograms on a subset of columns. All histograms are created with the very same value for the `size_clause` parameter. The syntax is a combination of that in Figure 8-7 and Figure 8-8: the former specifies the gathering of column statistics, and the latter specifies the gathering of histograms. For example, specifying `for all columns size 1 for columns size 254 col1` causes column statistics to be gathered for every column and a histogram with up to 254 buckets to be gathered for the `col1` column only.

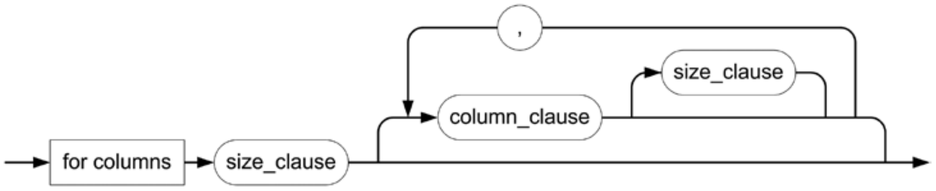


***Figure 8-8.*** *Gathering column statistics and histograms only for a subset of columns but with different values of the `size_clause` parameter (see Table 8-6). For the columns that don't explicitly specify a `size_clause`, the default `size_clause` (the first one) is used. If no columns are specified, no column statistics are gathered at all. The `column_clause` can be a column name, an extension name, or an extension. If an extension that doesn't exist is specified, a new extension is automatically created. This syntax is available only when the `gather_table_stats` procedure is used*

***Table 8-6.*** *Values Accepted by the `size_clause` Parameter*

| Value | Meaning |
|---|---|
| `size n` | The value specifies the maximum number of buckets. If `size 1` is specified, no histograms are created. In any case, column statistics are gathered normally. |
| `size skewonly` | Histograms are gathered only for columns with skewed data. The number of buckets is determined automatically. |
| `size auto` | Histograms are gathered only for columns with skewed data, such as `skewonly`, and, in addition, for those that have been referenced in `WHERE` clauses. This second condition is based on the column usage history. The number of buckets is determined automatically. |
| `size repeat` | Refreshes available histograms. |

- Gathering column statistics and histograms only for a subset of columns and with different values for the `size_clause` parameter. The syntax is shown in Figure 8-8. For example, specifying `for columns size 1 id, col1 size 100, col2 size 5, col3` causes column statistics to be gathered for the four columns, but histograms with up to 100 and respectively 5 buckets are gathered only for the columns `col1` and `col2`.

The default value is `for all columns size auto` (this default value can be changed—see the section "Configuring the dbms_stats Package" later in this chapter). For simplicity, use either `size skewonly` or `size auto`. If it's too slow or the chosen number of buckets isn't good (or the needed histogram isn't created at all), manually specify the list of columns. If `NULL` is specified, `for all columns size 1` is used.

## COLUMN USAGE HISTORY

The `dbms_stats` package relies on column usage history to determine on which columns a histogram is useful for. To gather the history, while generating a new execution plan, the query optimizer tracks which columns are referenced in the `WHERE` clause and stores the information it finds in the SGA. Then, at regular intervals, the database engine stores this information in the data dictionary table `col_usage$`. With a query based on internal data dictionary tables such as the following (which is available in the `col_usage.sql` script), it's possible to know which columns were referenced in `WHERE` clauses and for which kind of predicates. The `timestamp` column indicates the last usage. The other columns are counts of the number of hard parses (actually, hard parses providing the same information and executed in rapid succession aren't counted). Columns that were never referenced in a `WHERE` clause have no rows in the `col_usage$` table, so all columns in the output except `name` are `NULL`.

```
SQL> SELECT c.name, cu.timestamp,
  2         cu.equality_preds AS equality, cu.equijoin_preds AS equijoin,
  3         cu.nonequijoin_preds AS noneequijoin, cu.range_preds AS range,
  4         cu.like_preds AS "LIKE", cu.null_preds AS "NULL"
  5  FROM sys.col$ c, sys.col_usage$ cu, sys.obj$ o, dba_users u
  6  WHERE c.obj# = cu.obj# (+)
  7  AND c.intcol# = cu.intcol# (+)
  8  AND c.obj# = o.obj#
  9  AND o.owner# = u.user_id
 10  AND o.name = 'T'
 11  AND u.username = user
 12  ORDER BY c.col#;

NAME TIMESTAMP EQUALITY EQUIJOIN NONEEQUIJOIN RANGE  LIKE  NULL
---- --------- -------- -------- ------------ ----- ----- -----
ID   27-MAY-14        1        1            0     0     0     0
VAL1 27-MAY-14        1        0            0     0     0     0
VAL2
VAL3 27-MAY-14        1        1            0     0     0     0
PAD  27-MAY-14        0        0            0     1     0     0
```

As of version 11.2.0.2, the `report_col_usage` function of the `dbms_stats` package makes the selection of `col_usage$` information easier. Note that this is same function already discussed in the "Extended Statistics" section. But be aware that if the function `seed_col_usage` isn't used, the report returned by the `report_col_usage` function won't contain information about potential column groups. The following query shows an example and an excerpt of the output:

```
SQL> SELECT dbms_stats.report_col_usage(ownname => user, tabname => 't')
  2  FROM dual;
```

```
COLUMN USAGE REPORT FOR CHRIS.T
...............................

1. ID                                  : EQ EQ_JOIN
2. PAD                                 : RANGE
3. VAL1                                : EQ
4. VAL3                                : EQ EQ_JOIN
```

In addition, as of version 11.2.0.2, the `dbms_stats` package also provides a way to reset the content of `col_usage$`. You can do this through the `reset_col_usage` procedure.

- `degree` specifies the degree of parallelism used while gathering statistics for a single object. To use the degree of parallelism defined at the table/index level, specify the value `NULL`. To let the procedure determine the degree of parallelism, specify the `dbms_stats.default_degree` constant. The default value is `NULL` (this default value can be changed—see the section "Configuring the dbms_stats Package" later in this chapter). Note that the processing of several objects is serialized except when concurrent statistics gathering is used. This means parallelization is useful only for speeding up the gathering of statistics on large objects. To parallelize the processing of several objects at the same time, manual parallelization (that is, starting several jobs) is necessary. Refer to Chapter 15 for detailed information about parallel processing. Parallel gathering of object statistics is available only with the Enterprise Edition.

## CONCURRENT STATISTICS GATHERING

By default, the `dbms_stats` package only parallelizes the gathering at the table or partition level (based on the `degree` parameter). In other words, at any given time, only a single table or partition is processed. If the database server has plenty of free resources and a number of tables or partitions have to be processed, it may be sensible to simultaneously process them. For that purpose, as of version 11.2.0.2, Oracle Database provides a new gathering mode called *concurrent statistics gathering*.

Concurrent statistics gathering is implemented in the `gather_*_stats` procedures. To control it, the `concurrent` preference is available. Depending on the version you're running, it can be set to the following values:

- 11.2: `FALSE` to disable the feature (this is the default value), or `TRUE` to enable the feature.
- 12.1: `OFF` to disable the feature (this is default value), `MANUAL` to enable the feature only for manual statistics gathering, `AUTOMATIC` to enable the feature only for automatic statistics gathering, or `ALL` to enable the feature for all statistics gathering.

To take advantage of concurrent statistics gathering, the following requirements should be fulfilled:

- The `job_queue_processes` initialization parameter should be set to at least 4. This is necessary because to simultaneously process several tables or partitions, the `dbms_stats` package submits a number of jobs to the Scheduler.
- The Resource Manager should be enabled. Without it, because the `dbms_stats` package doesn't control how many concurrent jobs are run simulteneously, system load could go out of control. In fact, concurrent statistics gathering relies on the Scheduler and the Resource Manager to produce an optimal load.

- The user who submits the gathering must have either the `dba` role or the following privileges: `CREATE JOB`, `MANAGE SCHEDULER`, and `MANAGE ANY QUEUE`.
- `no_invalidate` specifies whether cursors depending on the processed objects are invalidated and, therefore, whether their future usage is prevented. This parameter accepts the values `TRUE`, `FALSE`, and `dbms_stats.auto_invalidate`. When the parameter is set to `TRUE`, the cursors depending on the changed object statistics aren't invalidated and, therefore, remain available for future executions. On the other hand, if it's set to `FALSE`, all the cursors are immediately invalidated. With the value `dbms_stats.auto_invalidate`, which is a constant evaluating to `NULL`, the cursors are invalidated over a period of time. This last possibility is good for avoiding reparsing spikes. The default value is `dbms_stats.auto_invalidate` (this default value can be changed—see the section "Configuring the dbms_stats Package" later in this chapter).

When `dbms_stats.auto_invalidate` is used, the `dbms_stats` package marks all cursors that depend on the changed statistics for delayed invalidation. The package sets a timestamp at the cursor level specifying when the cursor should no longer be used. Note that the timestamp is different for every cursor and is based on a random value of up to five hours from the moment the cursor is marked. The real invalidation is performed by the server processes that try to reuse a cursor that's marked for delayed invalidation. As a result, if a cursor marked for delayed invalidation is never reparsed, it will never be invalidated. The only exceptions are cursors related to parallel SQL statements. Such cursors are immediately invalidated by the `dbms_stats` package.

## Backup Table

The backup table parameters listed in Table 8-2 are supported by all procedures used for gathering object statistics. They instruct the `dbms_stats` package to back up current statistics in a backup table before overwriting them with the new ones in the data dictionary. The parameters are as follows:

- `stattab` specifies a backup table outside the data dictionary where the statistics are stored. If `NULL` (the default value) is specified, no backup table is used.
- `statid` is an optional ID used to recognize multiple sets of object statistics stored in the backup table specified with the `stattab` parameter. Only valid Oracle identifiers[4] are supported. If `NULL` (the default value) is specified, no ID is associated to the object statistics.
- `statown` specifies the owner of the table specified with the `stattab` parameter. The default value is `NULL`, and therefore the current user is used.

To create the backup table, the `dbms_stats` package provides the `create_stat_table` procedure. As shown in the following example, its creation is a matter of specifying the owner (with the `ownname` parameter) and the name (with the `stattab` parameter) of the backup table. In addition, the optional `tblspace` parameter specifies in which tablespace the table is created. If the `tblspace` parameter isn't specified, by default, the table ends up in the default tablespace of the user:

```
dbms_stats.create_stat_table(ownname  => user,
                             stattab  => 'MYSTATS',
                             tblspace => 'USERS')
```

[4]Refer to the *SQL Language Reference* manual of the Oracle documentation for the definition of *identifier*.

Because the backup table is used to store different kinds of information, most of its columns are generic. For example, in version 11.2 there are 12 columns to store numeric values (named n1 . . . n12), 5 columns to store character string values (named c1 . . . c5), 2 columns to store bit string values (named r1 and r2) and 1 column to store datatime values (named d1).

Note that over the years the backup table's structure has changed. As a result, you may need to upgrade the backup table after upgrading to a new database release or moving a backup table between different database releases. Otherwise, you might not be able to use the table. For that purpose, the dbms_stats package provides the upgrade_stat_table procedure. To use it, specify the owner (with the ownname parameter) and the name (with the stattab parameter) of the backup table. For example:

```
dbms_stats.upgrade_stat_table(ownname => user,
                              stattab => 'MYSTATS')
```

To drop a backup table, the dbms_stats package provides the drop_stat_table procedure:

```
dbms_stats.drop_stat_table(ownname => user,
                           stattab => 'MYSTATS')
```

You can also drop the backup table with a regular DROP TABLE statement.

## Configuring the dbms_stats Package

The dbms_stats package provides two sets of subprograms for configuring the default values of some of the parameters described in the preceding section. The first set should be used only in version 10.2. In fact, its subprograms are obsolete as of version 11.1. Therefore, starting with version 11.1, the subprograms provided by the second set of subprograms should be used.

### The Legacy Way

In version 10.2, you can change the global default values of the parameters cascade, estimate_percent, degree, method_opt, no_invalidate, and granularity. This is possible because the default values aren't hard-coded in the signature of the procedures, but rather extracted from the data dictionary at runtime. The set_param procedure of the dbms_stats package is available for setting default values. To execute it, you need the system privileges analyze any dictionary and analyze any. The get_param function of the dbms_stats package is for getting default values. The following example shows how use them. Note that pname is the parameter name and pval is the parameter value:

```
SQL> execute dbms_output.put_line(dbms_stats.get_param(pname => 'CASCADE'))

DBMS_STATS.AUTO_CASCADE

SQL> execute dbms_stats.set_param(pname => 'CASCADE', pval =>'TRUE')

SQL> execute dbms_output.put_line(dbms_stats.get_param(pname => 'CASCADE'))

TRUE
```

Another parameter that can be set with this method is autostats_target. This parameter is used only by the gather_stats_job job (described later in this chapter) to determine which objects the gathering of statistics has to process. Table 8-7 lists the available values. The default value is auto.

***Table 8-7.*** *Values Accepted by the `autostats_target` Parameter*

| Value | Meaning |
|---|---|
| `all` | All objects are processed. Up to and including version 11.2, fixed tables are excluded. However, from version 12.1 onward, fixed tables are included. |
| `auto` | The job determines which objects are processed. |
| `oracle` | Only objects belonging to the data dictionary, except fixed tables, are processed. |

To get the default value of all parameters without executing the `get_param` function several times, you can use the following query:[5]

```
SQL> SELECT sname AS parameter, nvl(spare4,sval1) AS default_value
  2  FROM sys.optstat_hist_control$
  3  WHERE sname IN ('CASCADE','ESTIMATE_PERCENT','DEGREE','METHOD_OPT',
  4                  'NO_INVALIDATE','GRANULARITY','AUTOSTATS_TARGET');

PARAMETER        DEFAULT_VALUE
---------------- ---------------------------
CASCADE          DBMS_STATS.AUTO_CASCADE
ESTIMATE_PERCENT DBMS_STATS.AUTO_SAMPLE_SIZE
DEGREE           NULL
METHOD_OPT       FOR ALL COLUMNS SIZE AUTO
NO_INVALIDATE    DBMS_STATS.AUTO_INVALIDATE
GRANULARITY      AUTO
AUTOSTATS_TARGET AUTO
```

To restore the default values to the original setting, the `dbms_stats` package provides the `reset_param_defaults` procedure.

## The Contemporary Way

As of version 11.1, the concept of setting default values for parameters, which are called *preferences*, is strongly enhanced compared to version 10.2. In fact, not only can you set the global defaults, you can also set defaults at the table level. One consequence of these enhancements is that the `get_param` function and the `set_param` and `reset_param_defaults` procedures described in the preceding section are obsolete.

You can change the default values of the parameters `autostats_target`, `cascade`, `concurrent`, `estimate_percent`, `degree`, `method_opt`, `no_invalidate`, `granularity`, `publish`, `incremental`, `stale_percent`, `table_cached_blocks` (as of version 11.2.0.4), and, as of version 12.1, `global_temp_table_stats`, `incremental_staleness`, and `incremental_level`. To change them, the following procedures are available in the `dbms_stats` package:

- `set_global_prefs` sets the global preferences. It replaces the `set_param` procedure.
- `set_database_prefs` sets the database preferences. The difference between global and database preferences is that the latter aren't used for the data dictionary objects. In other words, database preferences are used only for user-defined objects.

[5]Unfortunately, Oracle doesn't externalize this information through a data dictionary view, meaning this query is based on an internal table. The system privilege `select any dictionary` provides access to that table.

- `set_schema_prefs` sets the preferences for a specific schema.
- `set_table_prefs` sets the preferences for a specific table.

Note that the parameters `autostats_target` and `concurrent` can only be modified with the `set_global_prefs` procedure.

---

■ **Caution** The procedures `set_database_prefs` and `set_schema_prefs` don't directly store preferences in the data dictionary. Instead, they're converted into table preferences for all objects *presently* available in the database or schema at the time the procedure is called. In other words, only global and table preferences exist. The procedures `set_database_prefs` and `set_schema_prefs` are simple wrappers around the `set_table_prefs` procedure. This means that global preferences will be used for new tables created after these two procedures have been called.

---

The following PL/SQL blocks show how to set different values for the `cascade` parameter. Note that `pname` is the parameter name, `pvalue` is the parameter value, `ownname` is the owner, and `tabname` is the table name. Once again, be careful because the order of the calls is capital in such a PL/SQL block. In fact, every call overwrites some of the definition made by the previous call:

```
BEGIN
  dbms_stats.set_database_prefs(pname  => 'CASCADE',
                                pvalue => 'DBMS_STATS.AUTO_CASCADE');
  dbms_stats.set_schema_prefs(ownname => 'SCOTT',
                              pname   => 'CASCADE',
                              pvalue  => 'FALSE');
  dbms_stats.set_table_prefs(ownname => 'SCOTT',
                             tabname => 'EMP',
                             pname   => 'CASCADE',
                             pvalue  => 'TRUE');
END;
```

To get the current setting, the `get_prefs` function, which replaces the `get_param` function, is available. The following query shows the effect of the setting performed on the previous PL/SQL blocks. Note that `pname` is the parameter name, `ownname` is the owner name, and `tabname` is the table name. As you can see, depending on which parameters are specified, the function returns the values at a specific level. This searching for preferences is carried out as shown in Figure 8-9:

```
SQL> SELECT dbms_stats.get_prefs(pname   => 'cascade') AS global,
  2         dbms_stats.get_prefs(pname   => 'cascade',
  3                              ownname => 'SCOTT',
  4                              tabname =>'DEPT') AS dept,
  5         dbms_stats.get_prefs(pname   => 'cascade',
  6                              ownname => 'SCOTT',
  7                              tabname =>'EMP') AS emp
  8  FROM dual;

GLOBAL                  DEPT  EMP
----------------------- ----- ----
DBMS_STATS.AUTO_CASCADE FALSE TRUE
```

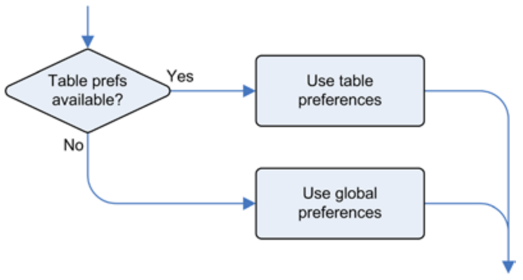


***Figure 8-9.*** *In searching for preferences, table settings take precedence over global preferences*

To get the global preferences without executing the get_param function several times, as just described in the "The Legacy Way" section, it's possible to query the internal data dictionary table optstat_hist_control$. To get preferences for tables, you can also run the following query. Notice that even if in the preceding PL/SQL block the configuration was performed at the schema level, the dba_tab_stat_prefs view shows the setting:

```
SQL> SELECT table_name, preference_name, preference_value
  2  FROM dba_tab_stat_prefs
  3  WHERE owner = 'SCOTT'
  4  AND table_name IN ('EMP', 'DEPT')
  5  ORDER BY table_name, preference_name;

TABLE_NAME PREFERENCE_NAME PREFERENCE_VALUE
---------- --------------- ----------------
DEPT       CASCADE         FALSE
EMP        CASCADE         TRUE
```

To get rid of preferences, the dbms_stats package provides the following procedures:

- reset_global_pref_defaults resets the global preferences to the default values.
- delete_database_prefs deletes preferences at the database level.
- delete_schema_prefs deletes preferences at the schema level.
- delete_table_prefs deletes preferences at the table level.

The following call shows how to delete the preferences related to the cascade parameter for all tables currently contained in the scott schema:

```
dbms_stats.delete_schema_prefs(ownname => 'SCOTT', pname => 'CASCADE')
```

To execute the procedures at the global and database levels, you need to have both the system privileges analyze any dictionary and analyze any. To execute the procedures at a schema or table level, you need to be connected as owner or have the system privilege analyze any.

## Working with Global Temporary Tables

Up to and including version 11.2, for the global temporary tables, the dbms_stats package provides only the gather_temp parameter of the gather_database stats and gather_schema stats procedures. With that parameter, you can only control whether global temporary tables are processed. How the gathering is carried out is no different from what is done for "regular" tables. As a result, most of the time and independently of how the object statistics are gathered, no object statistics are available for global temporary tables. The reason is twofold. First, the dbms_stats package executes a COMMIT at the beginning of the processing, and therefore, temporary tables created with the on commit delete rows option (which is the default) are always empty. Second, if the gathering, as usually happens, takes place in a job like the default gathering job, the global temporary tables are also empty. In general, the only way to get meaningful object statistics is to manually set them. But even if you set them manually, it might not be possible to find a set of object statistics that's good for everyone. In fact, every session can store a completely different amount of data in those tables.

Finally, version 12.1 introduces a feature to handle global temporary tables correctly. The idea is that you can choose between *shared statistics* (the only kind available in versions up to and including 11.2) and *session statistic*s. If session statistics are used (which is the default for global temporary tables), every session can gather a set of object statistics that won't be visible to other sessions. The gathering itself, as usual, is carried out with the gather_table_stats procedure of the dbms_stats package. This means that to take advantage of this feature, applications have to be modified to include a call to the gather_table_stats procedure just after loading the global temporary table. Note that to make this feature work, the COMMIT executed at the beginning of the processing by the dbms_stats package was removed. Here's an example (based on the gtt.sql script) that illustrates how it works:

```
SQL> CREATE GLOBAL TEMPORARY TABLE t (id NUMBER, pad VARCHAR2(1000));

SQL> INSERT INTO t SELECT rownum, rpad('*',1000,'*') FROM dual CONNECT BY level <= 1000;

SQL> execute dbms_stats.gather_table_stats(ownname => user, tabname => 't')

SQL> SELECT num_rows, blocks, avg_row_len, scope
  2  FROM user_tab_statistics
  3  WHERE table_name = 'T';

NUM_ROWS     BLOCKS AVG_ROW_LEN SCOPE
-------- ---------- ----------- -------
                                SHARED
    1000        147        1005 SESSION

SQL> SELECT count(*)
  2  FROM t
  3  WHERE id BETWEEN 10 AND 100;

  COUNT(*)
----------
        91

SQL> SELECT * FROM table(dbms_xplan.display_cursor);

---------------------------------------------------------------------------
| Id  | Operation          | Name | Rows  | Bytes | Cost (%CPU)| Time     |
---------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |      |       |       |    42 (100)|          |
|   1 |  SORT AGGREGATE    |      |     1 |     4 |            |          |
|*  2 |   TABLE ACCESS FULL| T    |    92 |   368 |    42   (0)| 00:00:01 |
---------------------------------------------------------------------------
```

```
Predicate Information (identified by operation id):
---------------------------------------------------

   2 - filter(("ID"<=100 AND "ID">=10))

Note
-----
   - Global temporary table session private statistics used
```

To control whether shared statistics or session statistics are used, you can set the `global_temp_table_stats` preference. Two values are supported: `shared` and `session`. The default value is `session`.

## Working with Pending Object Statistics

Usually, as soon as the gathering is finished, the object statistics are published (that is, made available) to the query optimizer. This means that it's not possible (for testing purposes, for instance) to gather statistics without overwriting the current object statistics. Of course, test databases should be used for testing purposes, but sometimes it's not possible to do so; you might want to do it in production. An example of this is when the data stored in the test database isn't the same as the data in the production database.

As of version 11.1, it's possible to separate gathering statistics from publishing them, and it's possible to use objects statistics that are unpublished, which are called *pending statistics*, for testing purposes. Here is the procedure (a full example is provided in the `pending_object_statistics.sql` script):

1. Disable automatic publishing by setting the `publish` preference to `FALSE` (the default value is `TRUE`). As described in the preceding section, you can do this at the global, database, schema, or table level. The following example shows how to do it for a table belonging to the current user:

   ```
   dbms_stats.set_table_prefs(ownname => user,
                              tabname => 'T',
                              pname   => 'PUBLISH',
                              pvalue  => 'FALSE')
   ```

2. Gather object statistics. Because the `publish` preference is set to `FALSE` for this table, the newly gathered object statistics aren't published. A set of pending statistics is created instead. This means the query optimizer keeps using the statistics available before the gathering. At the same time, cursors depending on that table aren't invalidated:

   ```
   dbms_stats.gather_table_stats(ownname => user, tabname => 'T')
   ```

3. To test the impact of the pending statistics on an application or a set of SQL statements, you can either set the `optimizer_use_pending_statistics` initialization parameter to `TRUE` at the session level or use the `opt_param('optimizer_use_pending_statistics' 'true')` hint at the SQL statement level.

4. If the test is successful, the pending statistics can be published (in other words, made available to all sessions) by calling the `publish_pending_stats` procedure. The following example shows how to do it for a single table. If the `tabname` parameter is set to `NULL`, all pending statistics of the specified schema are published. This procedure also has two additional parameters. The third, `no_invalidate`, controls the invalidation of the cursors depending on the modified object statistics, as previously described. The fourth, `force`,

is used to override a potential lock of the object statistics (the "Locking Object Statistics" section later in this chapter describes such locks). Its default value is FALSE, which means that locks are honored by default:

```
dbms_stats.publish_pending_stats(ownname => user, tabname => 'T')
```

5. If the test isn't successful, you can delete the pending statistics by calling the `delete_pending_stats` procedure. If the `tabname` parameter isn't specified or set to `NULL`, pending statistics for the whole schema specified by the `ownname` parameter are deleted:

```
dbms_stats.delete_pending_stats(ownname => user, tabname => 'T')
```

6. Enable automatic publishing by setting the `publish` preference to `TRUE`. This step is required to revert the change carried out in step 1:

```
dbms_stats.set_table_prefs(ownname => user,
                           tabname => 'T',
                           pname   => 'PUBLISH',
                           pvalue  => 'TRUE')
```

To execute the procedures `publish_pending_stats` and `delete_pending_stats`, you need to be connected as owner or have the `analyze any` system privilege.

If you're interested in knowing the values of the pending statistics, the following data dictionary views provide all the necessary information. For each view, there are `dba`, `all` and, in a 12.1 multitenant environment, `cdb` versions as well:

- `user_tab_pending_stats` shows pending table statistics.
- `user_ind_pending_stats` shows pending index statistics.
- `user_col_pending_stats` shows pending column statistics.
- `user_tab_histgrm_pending_stats` shows pending histograms.

The content and structure of these data dictionary views are similar to `user_tab_statistics`, `user_ind_statistics`, `user_tab_col_statistics`, and `user_tab_histograms`, respectively.

## Working with Partitioned Objects

Gathering object statistics for partitioned tables and indexes poses specific challenges. This section describes what the challenges are and introduces two techniques to tackle them.

### Challenges

The `dbms_stats` package uses two main approaches for gathering object statistics for partitioned tables and indexes:

- Gather object statistics at the object, partition, and, if available, subpartition level by means of queries that are independently executed at each level.
- Gather object statistics at the physical level only (either the partition or subpartition level) and use those results to derive the object statistics for the other levels.

There are two key differences in these approaches:

- The time and resources required to gather the object statistics with the first approach are, in general, much higher. In fact, when gathering the object statistics at the table/index level, all segments have to be accessed. The same thing happens with partition level statistics for subpartitioned objects. For example, think about the case of a weekly partitioned table containing data for several years. If a single partition is changed, then all partitions must be accessed to update the table/index statistics. Even though only the data of one partition was modified, all must be accessed.

- The second approach consumes much fewer resources, but it's able to produce accurate statistics only at the physical level. That's because it's not possible to derive the number of distinct values and the histograms from the underlying partitions and subpartitions. By the way, all other statistics *can* be derived.

Object statistics gathered with the first approach are called *global statistics*. Those gathered with the second approach are called *derived statistics* (or sometimes *aggregated statistics*). To recognize which type is in place, you can check whether the `global_stats` column in the data dictionary views listed in Table 8-2 (except for the views providing detailed information about histograms) is set to `YES` or `NO`. Whenever possible, the `dbms_stats` package gathers global statistics. It gathers derived statistics only when, for example, the gathering granularity is explicitly limited to the subpartition level and no object statistics are available at the partition and table/index level.

The following example, based on the output generated by the `global_stats.sql` script, shows a case where, for a table partitioned by range and subpartitioned by hash, derived statistics aren't accurate. Notice that not only are the number of distinct values at the table and partition level wrong, but the `global_stats` column is set to `NO`:

- A gathering at the subpartition level is performed on a table without object statistics (notice that no sampling is involved):

```
SQL> BEGIN
  2    dbms_stats.delete_table_stats(ownname => user,
  3                                  tabname => 't');
  4    dbms_stats.gather_table_stats(ownname          => user,
  5                                  tabname          => 't',
  6                                  estimate_percent => 100,
  6                                  granularity      => 'subpartition');
  7  END;
  8  /
```

- The number of distinct values at the table level), because they were gathered as derived statistics, is wrong:

```
SQL> SELECT count(DISTINCT sp)
  2  FROM t;

   COUNT(DISTINCTSP)
--------------------
                 100
SQL>
SQL> SELECT num_distinct, global_stats
  2  FROM user_tab_col_statistics
  3  WHERE table_name = 'T'
  4  AND column_name = 'SP';
```

```
NUM_DISTINCT GLOBAL_STATS
------------ ------------
          28 NO
```

- The number of distinct values at the partition level (here for a single partition), because they were gathered as derived statistics, is also wrong:

```
SQL> SELECT count(DISTINCT sp)
  2  FROM t PARTITION (q1);

   COUNT(DISTINCTSP)
--------------------
                 100
SQL>
SQL> SELECT num_distinct, global_stats
  2  FROM user_part_col_statistics
  3  WHERE table_name = 'T'
  4  AND partition_name = 'Q1'
  5  AND column_name = 'SP';

NUM_DISTINCT GLOBAL_STATS
------------ ------------
          28 NO
```

- The statistics about the number of distinct values at the subpartition level (here for a single partition) are right:

```
SQL> SELECT 'Q1_SP1' AS part_name, count(DISTINCT sp) FROM t SUBPARTITION (q1_sp1)
  2  UNION ALL
  3  SELECT 'Q1_SP2', count(DISTINCT sp) FROM t SUBPARTITION (q1_sp2)
  4  UNION ALL
  5  SELECT 'Q1_SP3', count(DISTINCT sp) FROM t SUBPARTITION (q1_sp3)
  6  UNION ALL
  7  SELECT 'Q1_SP4', count(DISTINCT sp) FROM t SUBPARTITION (q1_sp4);

PART_NAME COUNT(DISTINCTSP)
--------- -----------------
Q1_SP1                   20
Q1_SP2                   28
Q1_SP3                   25
Q1_SP4                   27

SQL> SELECT subpartition_name, num_distinct, global_stats
  2  FROM user_subpart_col_statistics
  3  WHERE table_name = 'T'
  4  AND column_name = 'SP'
  5  AND subpartition_name LIKE 'Q1%';
```

```
SUBPARTITION_NAME NUM_DISTINCT GLOBAL_STATS
----------------- ------------ ------------
Q1_SP1                      20 YES
Q1_SP2                      28 YES
Q1_SP3                      25 YES
Q1_SP4                      27 YES
```

■ **Caution** Object statistics at the table/index level can only be derived from the underlying partitions if all of those underlying partitions have object statistics in place. The same is also true for deriving partition statistics from subpartition statistics. In addition, be aware that the dbms_stats package doesn't replace global statistics with derived statistics. Both cases can be reproduced with the `global_stats.sql` script.

In summary, global statistics are more accurate than derived statistics, but require more time and resources to be gathered. Derived statistics might sometimes be enough. In practice, therefore, for big tables it's important to find a good balance between the required accuracy and the time and resources needed to achieve it. For this reason, the next two sections describe techniques that are available to manage object statistics for tables large enough to preclude the repeated gathering of full, global statistics.

## Incremental Statistics

As discussed in the preceding section, the ability to gather global statistics comes with pros and cons. The main pro is the accuracy of the object statistics at the table level and, if subartitioning is used, at the partition level. The main con is the time and resources that are needed to gather them.

The goal of incremental statistics is to offer the same accuracy by lowering the time and resources required to gather object statistics. How is that possible? The key idea is to leverage additional information (called synopses) stored in the data dictionary during the gathering of object statistics at the partition level, to accurately derive object statistics at the table level.

The following requirements have to be fulfilled to take advantage of incremental statistics:

- You're running version 11.1 or later.
- For the table being processed, the `incremental` preference is set to `TRUE` (the default is `FALSE`) :

```
dbms_stats.set_table_prefs(ownname => user,
                           tabname => 't',
                           pname   => 'incremental',
                           pvalue  => 'TRUE');
```

- For the table being processed, the `publish` preference is set to `TRUE` (this is the default value).
- For the table being processed, the `estimate_percent` parameter is set to `dbms_stats.auto_sample_size` (this is the default value).
- Additional space is available in the `sysaux` tablespace.

The gathering itself is performed as usual—for example, through a call to the `gather_table_stats` procedure of the `dbms_stats` package. The only thing to be careful about is that, to take advantage of incremental statistics, synopses need to be present at the partition level. As a result, after setting the `incremental` preference, you have to gather new object statistics on all partitions. You can consider this operation of gathering new object statistics on all partitions as the one that finally enables incremental statistics. In other words, fulfilling only the requirements listed above isn't enough.

Once all the synopses are in place, the dbms_stats package uses the monitoring information to know which partition (or subpartition) was modified and, therefore, requires new object statistics. Therefore, when using incremental statistics, you should *not* target the partitions (or subpartition) that changed. Instead, you let the dbms_stats package find out what it needs to do. The following example, based on the incremental_stats.sql script, illustrates (take a closer look at the last_analyzed timestamp to know which objects the statistics were gathered on):

```
SQL> SELECT object_type || ' ' || nvl(subpartition_name, partition_name) AS object,
  2         object_type, num_rows, blocks, avg_row_len,
  3         to_char(last_analyzed, 'HH24:MI:SS') AS last_analyzed
  4  FROM user_tab_statistics
  5  WHERE table_name = 'T'
  6  ORDER BY partition_name, subpartition_name;

OBJECT              OBJECT_TYPE  NUM_ROWS  BLOCKS AVG_ROW_LEN LAST_ANALYZED
------------------- ------------ -------- ------- ----------- -------------
SUBPARTITION Q1_SP1 SUBPARTITION     1786      46         116 14:52:22
SUBPARTITION Q1_SP2 SUBPARTITION     2173      46         116 14:52:22
PARTITION Q1        PARTITION        3959      92         116 14:52:22
SUBPARTITION Q2_SP1 SUBPARTITION     1804      46         116 14:52:22
SUBPARTITION Q2_SP2 SUBPARTITION     2200      46         116 14:52:22
PARTITION Q2        PARTITION        4004      92         116 14:52:22
SUBPARTITION Q3_SP1 SUBPARTITION     1815      46         116 14:52:22
SUBPARTITION Q3_SP2 SUBPARTITION     2233      46         116 14:52:22
PARTITION Q3        PARTITION        4048      92         116 14:52:22
SUBPARTITION Q4_SP1 SUBPARTITION     1795      46         116 14:52:22
SUBPARTITION Q4_SP2 SUBPARTITION     2194      46         116 14:52:22
PARTITION Q4        PARTITION        3989      92         116 14:52:22
TABLE               TABLE           16000     368         117 14:52:22

SQL> INSERT INTO t SELECT * FROM t SUBPARTITION (q1_sp1);

SQL> execute dbms_stats.gather_table_stats(ownname => user, tabname => 't', granularity=>'all')

SQL> SELECT object_type || ' ' || nvl(subpartition_name, partition_name) AS object,
  2         object_type, num_rows, blocks, avg_row_len,
  3         to_char(last_analyzed, 'HH24:MI:SS') AS last_analyzed
  4  FROM user_tab_statistics
  5  WHERE table_name = 'T'
  6  ORDER BY partition_name, subpartition_name;

OBJECT              OBJECT_TYPE  NUM_ROWS BLOCKS AVG_ROW_LEN LAST_ANALYZED
------------------- ------------ -------- ------ ----------- -------------
SUBPARTITION Q1_SP1 SUBPARTITION     3572    110         116 14:54:39
SUBPARTITION Q1_SP2 SUBPARTITION     2173     46         116 14:52:22
PARTITION Q1        PARTITION        5745    156         116 14:54:40
SUBPARTITION Q2_SP1 SUBPARTITION     1804     46         116 14:52:22
SUBPARTITION Q2_SP2 SUBPARTITION     2200     46         116 14:52:22
PARTITION Q2        PARTITION        4004     92         116 14:52:22
SUBPARTITION Q3_SP1 SUBPARTITION     1815     46         116 14:52:22
SUBPARTITION Q3_SP2 SUBPARTITION     2233     46         116 14:52:22
PARTITION Q3        PARTITION        4048     92         116 14:52:22
```

```
SUBPARTITION Q4_SP1 SUBPARTITION     1795     46          116 14:52:22
SUBPARTITION Q4_SP2 SUBPARTITION     2194     46          116 14:52:22
PARTITION Q4        PARTITION        3989     92          116 14:52:22
TABLE               TABLE           17786    432          117 14:54:40
```

As the example shows, the object statistics associated to partitions (or subpartitions) that experience any modification are considered stale. From version 12.1 onward, there is a preference, `incremental_staleness`, that lets you control this behavior. With the default value, `NULL`, the behavior is the same as in the previous version (any modification makes a partition stale). If you set the value `use_stale_percent`, the object statistics associated to partitions (or subpartitions) are considered stale only when the number of modifications crosses the threshold set through the `stale_percent` preference. In addition, with the value `use_locked_stats`, you can define that object statistics associated to partitions (or subpartitions) with locked statistics are never considered stale. Note that `use_stale_percent` and `use_locked_stats` can be enabled at the same time. Here's an example:

```
dbms_stats.set_table_prefs(ownname => user,
                           tabname => 't',
                           pname   => 'incremental_staleness',
                           pvalue  => 'use_stale_percent, use_locked_stats');
```

Only in version 12.1 can the `dbms_stats` package create synompses on nonpartitioned tables (for that purpose, the `incremental_level` preference has to be set to `table`). As a result, only in version 12.1 can partition exchange take advantage of incremental statistics.

---

■ **Tip** The Oracle Support note *How To Collect Statistics On Partitioned Table in 10g and 11g* (1417133.1) provides a list of the most important bugs and patches related to incremental statistics. Check the note to know whether the version you're using requires particular attention, and check any further notes referenced by the first one.

---

## Copying Statistics

In situations where partitions are frequently added and their content changes significantly over time, keeping a representative set of partition level statistics requires very frequent gatherings. These frequent gatherings represent significant overhead in terms of resource utilization. In addition, under normal conditions, it's not good to leave a recently added partition without object statistics. Doing so leads to dynamic sampling, a feature covered in Chapter 9. To cope with such issues, the `dbms_stats` package, through the `copy_table_stats` procedure, provides the functionality to copy object statistics from one partition or subpartition to another. Note that the copy takes care of column statistics as well as dependent objects like subpartitions and local indexes.

The following command illustrates how to perform a copy (a full example is provided in the `copy_table_stats.sql` script). The `ownname` and `tabname` parameters specify the table the command is executed on. The `srcpartname` and `dstpartname` parameters specify the source and destination partition (or subpartition), respectively:

```
dbms_stats.copy_table_stats(ownname      => user,
                            tabname      => 't',
                            srcpartname  => 'p_2014_q1',
                            dstpartname  => 'p_2015_q1',
                            scale_factor => 1);
```

It's important to point out that the `copy_table_stats` procedure doesn't perform a simple, one-to-one copy. Instead, it's able to change the minimum and maximum values according to the way the partitions are defined. For example, and given a range partitioned table, the package can derive the minimum and maximum value from the partition bounds of the destination partition and of the partition preceding it. In addition, as of version 10.2.0.5, you can scale up or down the number of rows and blocks by setting a `scale_factor` parameter to a value different than 1 (the default).

If the table on which the copy is performed has derived statistics at the table/index level, then the table/index level statistics are also amended during the copy. For example, the number of rows is increased, and the maximum value is set accordingly. A similar thing happens at the partition level when statistics are copied between subpartitions.

---

■ **Tip** All versions up to 11.2.0.3 contain some bugs in the `copy_table_stats` procedure. Some of those bugs are corner cases that you might never hit, but others, depending on the version you're using, impact core functionality. Search for "copy table stats" on the Oracle Support website to learn whether the version you are using requires particular attention.

---

## Scheduling Object Statistics Gathering

The query optimizer requires object statistics to correctly carry out its duties. Thus, when a new database is created, a job calling the `gather_database_stats_job_proc` procedure in the `dbms_stats` package is set up by default. The `gather_database_stats_job_proc` procedure performs essentially the same thing that would happen were you to invoke the `gather_database_stats` procedure of the `dbms_stats` package with the `options` parameter set to `gather_stale` and `gather_empty`. Note that although in version 10.2 a regular job is used, as of version 11.1 the gathering is integrated in the automated maintenance tasks. In both cases, the job is scheduled using the `dbms_scheduler` package, not with the `dbms_job` package.

---

■ **Caution** Up to and including version 11.2, by default the job targets all objects except for fixed tables. As a result, you have to take care of fixed tables yourself by gathering object statistics for them when the database engine is at peak load. I advise you to do it at peak load because their content is strongly dependent on the load. Think, for example, about `x$ksuse`, which contains one row for each session.

---

The aim of the next two sections is to provide detailed information about the configuration used for scheduling the default job. The first section covers 10g. The next section covers later versions.

### The 10g Way

The `gather_stats_job` job is automatically setup in 10g. The current configuration, which in the following example is the default configuration of version 10.2, can be displayed with the following queries. The output was generated with the `dbms_stats_job_10g.sql` script:

```
SQL> SELECT program_name, schedule_name, enabled, state
  2  FROM dba_scheduler_jobs
  3  WHERE owner = 'SYS'
  4  AND job_name = 'GATHER_STATS_JOB';
```

```
PROGRAM_NAME      SCHEDULE_NAME            ENABLED STATE
----------------- ----------------------- ------- ---------
GATHER_STATS_PROG MAINTENANCE_WINDOW_GROUP TRUE    SCHEDULED

SQL> SELECT program_action, number_of_arguments, enabled
  2  FROM dba_scheduler_programs
  3  WHERE owner = 'SYS'
  4  AND program_name = 'GATHER_STATS_PROG';

PROGRAM_ACTION                            NUMBER_OF_ARGUMENTS ENABLED
----------------------------------------- ------------------- -------
dbms_stats.gather_database_stats_job_proc                   0 TRUE

SQL> SELECT w.window_name, w.repeat_interval, w.duration, w.enabled
  2  FROM dba_scheduler_jobs j, dba_scheduler_wingroup_members m,
  3       dba_scheduler_windows w
  4  WHERE j.schedule_name = m.window_group_name
  5  AND m.window_name = w.window_name
  6  AND j.owner = 'SYS'
  7  AND j.job_name = 'GATHER_STATS_JOB';

WINDOW_NAME      REPEAT_INTERVAL                       DURATION      ENABLED
---------------- ------------------------------------- ------------- -------
WEEKNIGHT_WINDOW freq=daily;byday=MON,TUE,WED,THU,FRI; +000 08:00:00 TRUE
                 byhour=22;byminute=0; bysecond=0
WEEKEND_WINDOW   freq=daily;byday=SAT;byhour=0;byminut +002 00:00:00 TRUE
                 e=0;bysecond=0
```

In summary, the configuration is the following:

- The job executes the gather_stats_prog program and is can run within the maintenance_window_group window group.
- The gather_stats_prog program calls, without parameters, the gather_database_stats_job_proc procedure in the dbms_stats package. Because no parameters are passed to it, the only way to change its behavior is to change the default configuration of the dbms_stats package, as explained in the "Configuring the dbms_stats Package" section earlier in this chapter. Note that this procedure is undocumented and tagged "for internal use only."
- The maintenance_window_group window group has two members: the weeknight_window window and the weekend_window window. The former opens for eight hours every night from Monday to Friday. The latter is open Saturday and Sunday. The gathering of the object statistics takes place when one of these two windows is open.
- The job, the program, and the windows are enabled.

The opening and duration of the default scheduling should be checked and, whenever necessary, changed to match the expected statistics gathering frequency. If possible, they should match the low-utilization periods.

Every time the job has to be stopped because the window is closed, a trace file, containing the list of all objects that have not been processed, is written in the directory referenced by the background_dump_dest initialization parameter. The following is an excerpt of such a trace file:

```
GATHER_STATS_JOB: Stopped by Scheduler.
Consider increasing the maintenance window duration if this happens frequently.
The following objects/segments were not analyzed due to timeout:
TABLE: "SH"."SALES"."SALES_1995"
TABLE: "SH"."SALES"."SALES_1996"
TABLE: "SH"."SALES"."SALES_H1_1997"
...
TABLE: "SYS"."WRI$_OPTSTAT_AUX_HISTORY".""
TABLE: "SYS"."WRI$_ADV_OBJECTS".""
TABLE: "SYS"."WRI$_OPTSTAT_HISTGRM_HISTORY".""
error 1013 in job queue process
ORA-01013: user requested cancel of current operation
```

To enable or disable the gather_stats_job job, the following PL/SQL calls are available:

```
dbms_scheduler.enable(name => 'sys.gather_stats_job')

dbms_scheduler.disable(name => 'sys.gather_stats_job')
```

Per default, only the sys user is able to execute them. Other users need the alter object privilege. For example, after executing the following SQL statement, the system user can not only alter but also drop the gather_stats_job job:

```
GRANT ALTER ON gather_stats_job TO system
```

## The 11g and 12c Way

As of version 11.1, the gathering of object statistics is integrated into the automated maintenance tasks. As a result, the gather_stats_job job described in the preceding section no longer exists. The current configuration, which in the following example is the default configuration of version 11.2, can be viewed with the following queries. The output was generated with the dbms_stats_job_11g.sql script:

```
SQL> SELECT task_name, status
  2  FROM dba_autotask_task
  3  WHERE client_name = 'auto optimizer stats collection';

TASK_NAME         STATUS
----------------- -------
gather_stats_prog ENABLED

SQL> SELECT program_action, number_of_arguments, enabled
  2  FROM dba_scheduler_programs
  3  WHERE owner = 'SYS'
  4  AND program_name = 'GATHER_STATS_PROG';
```

```
PROGRAM_ACTION                           NUMBER_OF_ARGUMENTS ENABLED
---------------------------------------- ------------------- -------
dbms_stats.gather_database_stats_job_proc                  0 TRUE

SQL> SELECT window_group
  2  FROM dba_autotask_client
  3  WHERE client_name = 'auto optimizer stats collection';

WINDOW_GROUP
--------------
ORA$AT_WGRP_OS

SQL> SELECT w.window_name, w.repeat_interval, w.duration, w.enabled
  2  FROM dba_autotask_window_clients c, dba_scheduler_windows w
  3  WHERE c.window_name = w.window_name
  4  AND c.optimizer_stats = 'ENABLED';

WINDOW_NAME      REPEAT_INTERVAL                                      DURATION      ENABLED
---------------- ---------------------------------------------------- ------------- -------
MONDAY_WINDOW    freq=daily;byday=MON;byhour=22;byminute=0; bysecond=0 +000 04:00:00 TRUE
TUESDAY_WINDOW   freq=daily;byday=TUE;byhour=22;byminute=0; bysecond=0 +000 04:00:00 TRUE
WEDNESDAY_WINDOW freq=daily;byday=WED;byhour=22;byminute=0; bysecond=0 +000 04:00:00 TRUE
THURSDAY_WINDOW  freq=daily;byday=THU;byhour=22;byminute=0; bysecond=0 +000 04:00:00 TRUE
FRIDAY_WINDOW    freq=daily;byday=FRI;byhour=22;byminute=0; bysecond=0 +000 04:00:00 TRUE
SATURDAY_WINDOW  freq=daily;byday=SAT;byhour=6;byminute=0; bysecond=0  +000 20:00:00 TRUE
SUNDAY_WINDOW    freq=daily;byday=SUN;byhour=6;byminute=0; bysecond=0  +000 20:00:00 TRUE
```

In summary, the configuration is the following:

- The gather_stats_prog program calls, without parameters, the gather_database_stats_job_proc procedure in the dbms_stats package. Because no parameters are passed to it, the only way to change its behavior is to change the default configuration of the dbms_stats package, as explained in the “Configuring the dbms_stats Package” section earlier in this chapter. Note that this procedure is undocumented and tagged “for internal use only.”
- The window group used by the automatic maintenance task has seven members, one for each day of the week. From Monday to Friday it’s open four hours a day. For Saturday and Sunday, it’s open 20 hours a day. The gathering of the object statistics takes place when one of these windows is open. Note that when a window is open for a long time, such as on the weekend, the gather_stats_prog program is restarted every four hours.
- The maintenance task, the program, and the windows are enabled.

The opening and duration of the default scheduling should be checked and, whenever necessary, changed to match the expected statistics gathering frequency. If possible, they should match the low-utilization periods.

To completely enable or disable the maintenance task, the following PL/SQL calls are available. By setting the windows_name parameter to a non-NULL value, you can also enable or disable the maintenance task for a single window only.

```
dbms_auto_task_admin.enable(client_name => 'auto optimizer stats collection',
                            operation   => NULL,
                            window_name => NULL)
```

```
dbms_auto_task_admin.disable(client_name => 'auto optimizer stats collection',
                             operation   => NULL,
                             window_name => NULL)
```

■ **Caution** As of version 11.2, setting the `job_queue_processes` initialization parameter to 0 disables the automatic statistics job (and everything else scheduled through the Scheduler).

## Restoring Object Statistics

Whenever object statistics are gathered through the `dbms_stats` package or, as of version 11.2, through the `ALTER INDEX` statement, instead of simply overwriting current statistics with the new statistics, the current statistics are saved in other data dictionary tables that keep a history of all changes occurring within a retention period. The purpose is to be able to restore old statistics in case new statistics lead to inefficient execution plans.

Object statistics (as well as system statistics, because they're maintained by the same underlying functionality) are kept in the history for an interval specified by a retention period. The default value is 31 days. You can display the current value by calling the `get_stats_history_retention` function in the `dbms_stats` package, as shown here:

```
SELECT dbms_stats.get_stats_history_retention() AS retention FROM dual
```

To change the retention period, the `dbms_stats` package provides the `alter_stats_history_retention` procedure. Here's an example where the call sets the retention period to 14 days:

```
dbms_stats.alter_stats_history_retention(retention => 14)
```

Note that with the `alter_stats_history_retention` procedure, the following values have a special meaning:

- `NULL` sets the retention period to the default value.
- 0 disables the history.
- -1 disables the purging of the history.

When the `statistics_level` initialization parameter is set to `typical` (the default value) or `all`, statistics older than the retention period are automatically purged. Whenever manual purging is necessary, the `dbms_stats` package provides the `purge_stats` procedure. The following call purges all statistics placed in the history more than 14 days ago:

```
dbms_stats.purge_stats(before_timestamp => systimestamp - INTERVAL '14' DAY)
```

To execute the `alter_stats_history_retention` and `purge_stats` procedures, you need to have the `analyze any` and `analyze any dictionary` system privileges.

If you're interested in knowing when object statistics for a given table were modified, the `user_tab_stats_history` data dictionary view provides all the necessary information. Of course, there are `dba`, `all` and, in a 12.1 multitenant environment, `cdb` versions of that view as well. Here's an example. With the following query, it's possible to display when the object statistics of the `tab$` table in the `sys` schema were modified:

```
SQL> SELECT stats_update_time
  2  FROM dba_tab_stats_history
  3  WHERE owner = 'SYS' and table_name = 'TAB$';
```

```
STATS_UPDATE_TIME
--------------------------------
26-MAR-14 22.03.03.104730 +01:00
27-MAR-14 22.01.14.193033 +01:00
13-APR-14 14.14.57.461660 +02:00
```

Whenever it may be necessary, statistics can be restored from the history. For that purpose, the `dbms_stats` package provides the following procedures:

- `restore_database_stats` restores object statistics for the whole database.
- `restore_dictionary_stats` restores object statistics for the data dictionary.
- `restore_fixed_objects_stats` restores object statistics for fixed tables and their indexes.
- `restore_schema_stats` restores object statistics for a single schema.
- `restore_table_stats` restores object statistics for a single table.

In addition to the parameters specifying the target (for example, the schema and table names for the `restore_table_stats` procedure), all these procedures provide the following parameters:

- `as_of_timestamp` specifies to restore the statistics that were in use at a specific time.
- `force` specifies whether locked statistics should be overwritten. Note that locks on statistics are part of the history. This means the information about whether statistics are locked or not is also restored. The default value is `FALSE`.
- `no_invalidate` specifies whether cursors depending on the overwritten statistics are invalidated. This parameter accepts the values `TRUE`, `FALSE`, and `dbms_stats.auto_invalidate`. The default value is `dbms_stats.auto_invalidate`.

The following call restores the object statistics of the `SH` schema to the values that were in use one day ago. Because the `force` parameter is set to `TRUE`, the restore is done even if statistics are currently locked:

```
dbms_stats.restore_schema_stats(ownname         => 'SH',
                                as_of_timestamp => systimestamp - INTERVAL '1' DAY,
                                force           => TRUE)
```

## Locking Object Statistics

In some situations, you want to make sure that object statistics for part of the database aren't available or can't be changed, either because you want to use dynamic sampling (see Chapter 9), because you have to use object statistics that aren't up-to-date (for example, because the content of some tables changes very frequently and you want to carefully gather stats only when the table contains a representative set of rows), or because gathering statistics isn't possible (for example, because of bugs).

It's possible to explicitly lock object statistics by executing one of the following procedures in the `dbms_stats` package. Note that these locks have nothing to do with regular database locks. They are, in fact, simple flags set at the table level in the data dictionary:

- `lock_schema_stats` locks object statistics for all tables belonging to a schema:

  ```
  dbms_stats.lock_schema_stats(ownname => user)
  ```

- `lock_table_stats` locks object statistics for a single table:

  ```
  dbms_stats.lock_table_stats(ownname => user, tabname => 'T')
  ```

Naturally, it's also possible to remove the locks, which you can do by executing one of the following procedures:

- `unlock_schema_stats` removes locks from object statistics for all tables belonging to a schema. Even locks that were set with the `lock_table_stats` procedure are removed:

  ```
  dbms_stats.unlock_schema_stats(ownname => user)
  ```

- `unlock_table_stats` removes the lock from object statistics for a single table:

  ```
  dbms_stats.unlock_table_stats(ownname => user, tabname => 'T')
  ```

To execute these four procedures, you need to be connected as owner or have the `analyze any` system privilege.

When the object statistics of a table are locked, all the object statistics related to that table (including table statistics, column statistics, histograms, and index statistics on all dependent indexes) are considered to be locked.

When the object statistics of a table are locked, procedures in the `dbms_stats` package that modify object statistics of a single table (for example `gather_table_stats`) raise an error (ORA-20005). In contrast, procedures that operate on multiple tables (for example `gather_schema_stats`) skip the locked table. Most procedures that modify object statistics can override the lock by setting the `force` parameter to `TRUE`. The following example demonstrates that behavior (a full example is provided in the `lock_statistics.sql` script):

```
SQL> BEGIN
  2    dbms_stats.lock_schema_stats(ownname => user);
  3  END;
  4  /

SQL> BEGIN
  2    dbms_stats.gather_schema_stats(ownname => user);
  3  END;
  4  /

SQL> BEGIN
  2    dbms_stats.gather_table_stats(ownname => user,
  3                                  tabname => 'T');
  4  END;
  5  /
BEGIN
*
ERROR at line 1:
ORA-20005: object statistics are locked (stattype = ALL)
ORA-06512: at "SYS.DBMS_STATS", line 33859
ORA-06512: at line 2

SQL> BEGIN
  2    dbms_stats.gather_table_stats(ownname => user,
  3                                  tabname => 'T',
  4                                  force   => TRUE);
  5  END;
  6  /
```

To know which tables object statistics are locked for, you can use a query like the following:

```
SQL> SELECT table_name
  2  FROM user_tab_statistics
  3  WHERE stattype_locked IS NOT NULL;

TABLE_NAME
----------
T
```

Be aware that the dbms_stats package isn't the only one that gathers object statistics and, therefore, is affected by locks on object statistics. In fact, the ANALYZE, CREATE INDEX, and ALTER INDEX statements—as well as, from version 12.1 onward, CTAS statements and direct-path inserts into empty tables—also gather object statistics. The first gathers object statistics when it's explicitly instructed to do so. But, as stated at the beginning of this chapter, you should no longer use it for that purpose. The others automatically gather object statistics while carrying out the task they're design for. This is useful because the overhead associated with the gathering of statistics while executing these SQL statements is negligible. Consequently, when the object statistics of a table are locked, these SQL statements may behave differently or even fail. The following example, a continuation of the previous one, shows this behavior:

```
SQL> ANALYZE TABLE t COMPUTE STATISTICS;
ANALYZE TABLE t COMPUTE STATISTICS
*
ERROR at line 1:
ORA-38029: object statistics are locked

SQL> ANALYZE TABLE t VALIDATE STRUCTURE;

SQL> ALTER INDEX t_pk REBUILD COMPUTE STATISTICS;
ALTER INDEX t_pk REBUILD COMPUTE STATISTICS
*
ERROR at line 1:
ORA-38029: object statistics are locked

SQL> ALTER INDEX t_pk REBUILD;

SQL> CREATE INDEX t_i ON t (pad) COMPUTE STATISTICS;
CREATE INDEX t_i ON t (pad) COMPUTE STATISTICS
                    *
ERROR at line 1:
ORA-38029: object statistics are locked

SQL> CREATE INDEX t_i ON t (pad);
```

Notice that the SQL statement CREATE INDEX and ALTER INDEX fails only when the deprecated COMPUTE STATISTICS clause is specified. Because these SQL statements gather object statistics by default, it's pointless to use the COMPUTE STATISTICS clause.

## Comparing Object Statistics

In the following three common situations, you end up with several sets of object statistics for the very same object:

- When you instruct the `dbms_stats` package (through the parameters `statown`, `stattab`, and `statid`) to save the current statistics in a backup table.
- Whenever the `dbms_stats` package is used to gather object statistics. In fact, as already described in the "Restoring Object Statistics" section, the package automatically keeps a history of the object statistics instead of simply overwriting them when a new set is gathered.
- As of version 11.1, when you gather pending statistics.

It's not unusual to want to know the differences between two sets of object statistics. As of version 10.2.0.4, you're no longer required to write queries yourself to make such a comparison. You can simply take advantage of new functions in the `dbms_stats` package.

The following example, an excerpt of the output generated by the `comparing_object_statistics.sql` script, shows the kind of report you can get:

```
###############################################################################

STATISTICS DIFFERENCE REPORT FOR:
.................................

TABLE         : T
OWNER         : CHRIS
SOURCE A      : Statistics as of 10-APR-13 20.05.07.106712 +02:00
SOURCE B      : Current Statistics in dictionary
PCTTHRESHOLD  : 10
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
TABLE / (SUB)PARTITION STATISTICS DIFFERENCE:
.............................................

OBJECTNAME                  TYP SRC ROWS       BLOCKS     ROWLEN     SAMPSIZE
...............................................................................

T                           T   A   10088      110        37         5865
                                B   12691      253        37         5036
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
COLUMN STATISTICS DIFFERENCE:
.............................

COLUMN_NAME     SRC NDV     DENSITY    HIST NULLS   LEN  MIN   MAX   SAMPSIZ
...............................................................................

ID              A   9862    .000101399 NO   0       4    C103  C2646 5734
                B   12645   .000079082 NO   0       5    C108  C3026 5018
VAL1            A   3203    .000454959 YES  0       5    3D382 C2240 5779
                B   2990    .000489236 YES  0       5    3D421 C2251 4926
VAL2            A   9       .000049759 YES  0       3    C10C  C114  5842
                B   9       .000039438 YES  0       3    C10C  C114  5031
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

```
INDEX / (SUB)PARTITION STATISTICS DIFFERENCE:
.............................................

OBJECTNAME      TYP SRC ROWS    LEAFBLK DISTKEY LF/KY DB/KY CLF     LVL SAMPSIZ
...............................................................................

                                   INDEX: T_PK
                                   ...........

T_PK            I   A   10000   20      10000   1     1     9901    1   10000
                    B   12500   27      12500   1     1     12300   1   12500
###############################################################################
```

Notice how in the first part, you can see the parameters used for the comparison: the schema and the table name, the definition of two sources (A and B), and a threshold. This last parameter specifies whether to display only the object statistics for which the difference (in percent) between the two sets of statistics exceeds the specified threshold. For example, if you have the two values 100 and 115, they're recognized as different only if the threshold is set to 15 or less. The default value is 10. To display all object statistics, the value 0 can be used.

The following are the functions available in the `dbms_stats` package to generate such a report:

- `diff_table_stats_in_stattab` compares the object statistics found in a backup table (specified with the parameters `ownname` and `tabname`) with the current object statistics or another set found in another backup table. The parameters `stattab1`, `statid1`, and `stattab1own` are provided to specify the first backup table. The second backup table (which is optional) is specified with the parameters `stattab2`, `statid2`, and `stattab2own`. If the parameters of the second backup table aren't specified, or if they're set to `NULL`, the current object statistics are compared with the object statistics in the first backup table. The following example compares the current object statistics with a set of object statistics named `set1` and stored in the `mystats` backup table:

```
dbms_stats.diff_table_stats_in_stattab(ownname      => user,
                                       tabname      => 'T',
                                       stattab1     => 'MYSTATS',
                                       statid1      => 'SET1',
                                       stattab1own  => user,
                                       pctthreshold => 10)
```

- `diff_table_stats_in_history` compares, for one table, either the current object statistics with object statistics from the history or two sets of object statistics from the history. The parameters `time1` and `time2` are provided to specify which statistics are used. If the parameter `time2` isn't specified, or set to `NULL`, the current object statistics are compared to another set from the history. The following example compares the current object statistics with the object statistics of one day ago (for example, prior to a gathering of statistics that was executed during the night):

```
dbms_stats.diff_table_stats_in_history(ownname      => user,
                                       tabname      => 'T',
                                       time1        => systimestamp - 1,
                                       time2        => NULL,
                                       pctthreshold => 10));
```

- diff_table_stats_in_pending compares, for one table, either the current object statistics or a set from the history, with the pending statistics. To specify object statistics stored in the history, the parameter time_stamp is provided. If this parameter is set to NULL (default), current object statistics are compared to pending statistics. The following example compares the current statistics with the pending statistics:

```
dbms_stats.diff_table_stats_in_pending(ownname => user,
                                       tabname => 'T',
                                       time_stamp => NULL,
                                       pctthreshold => 10));
```

## Deleting Object Statistics

You can delete object statistics from the data dictionary. Except for testing purposes, this is usually not necessary. Nevertheless, it might happen that a table shouldn't have object statistics because you want to take advantage of dynamic sampling (covered in Chapter 9). In that case, the following procedures are available in the dbms_stats package: delete_database_stats, delete_dictionary_stats, delete_fixed_objects_stats, delete_schema_stats, delete_table_stats, delete_column_stats, and delete_index_stats.

As you can see, for each gather_*_stats procedure, there is a corresponding delete_*_stats procedure. The former ones gather object statistics, and the latter ones delete object statistics. The only exception is the delete_column_stats procedure. As its name suggests, it's used for deleting column statistics and histograms.

Table 8-8 summarizes the parameters available for each of these procedures. Most of them are the same and, therefore, have the same meaning as the parameters used by the gather_*_stats procedures. I describe here only the parameters that have not already been described with the earlier procedures:

- cascade_parts specifies whether statistics for all underlying partitions are deleted. This parameter accepts the values TRUE and FALSE. The default value is TRUE.
- cascade_columns specifies whether column statistics are deleted as well. This parameter accepts the values TRUE and FALSE. The default value is TRUE.
- cascade_indexes specifies whether index statistics are deleted as well. This parameter accepts the values TRUE and FALSE. The default value is TRUE.
- col_stat_type specifies which statistics are deleted. If it's set to ALL, column statistics and histograms are deleted. If it's set to HISTOGRAM, only histograms are deleted. The default value is ALL. This parameter is available as of version 11.1.
- stat_category specifies which category of statistics is deleted. It accepts a comma-separated list of values. If OBJECT_STATS is specified, object statistics (table statistics, column statistics, histograms, and index statistics) are deleted. If SYNOPSES is specified, only information supporting incremental statistics is deleted. By default, both object statistics and synopses are deleted. This parameter is available as of version 12.1.

***Table 8-8.*** *Parameters of the Procedures Used for Deleting Object Statistics*

| Parameter | Database | Dictionary | Fixed Objects | Schema | Table | Column | Index |
|---|---|---|---|---|---|---|---|
| **Target Objects** | | | | | | | |
| ownname | | | | ✓ | ✓ | ✓ | ✓ |
| indname | | | | | | | ✓ |
| tabname | | | | | ✓ | ✓ | |
| colname | | | | | | ✓ | |
| partname | | | | | ✓ | ✓ | ✓ |
| cascade_parts | | | | | ✓ | ✓ | ✓ |
| cascade_columns | | | | | ✓ | | |
| cascade_indexes | | | | | ✓ | | |
| stat_category | ✓ | ✓ | | ✓ | ✓ | | |
| col_stat_type | | | | | | ✓ | |
| force | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| **Deleting Options** | | | | | | | |
| no_invalidate | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| **Backup Table** | | | | | | | |
| stattab | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| statid | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| statown | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |

The following call shows how to delete the histogram of one single column (a full example is found in the `delete_histogram.sql` script) without modifying the other statistics:

```
dbms_stats.delete_column_stats(ownname      => user,
                               tabname      => 'T',
                               colname      => 'VAL',
                               col_stat_type => 'HISTOGRAM')
```

## Exporting, Importing, Getting, and Setting Object Statistics

As illustrated earlier in Figure 8-1, the `dbms_stats` package provides several procedures and functions that are available in addition to those used for gathering statistics. I won't describe them here because they're not frequently used in practice. For information about them, refer to the *Oracle Database PL/SQL Packages and Types Reference* manual. Nevertheless, I want to share the following piece of information that you don't find in the documentation.

■ **Caution** The export and import procedures provided by the dbms_stats package process locks set with the techniques described in the “Locking Object Statistics” section earlier in this chapter. The only exception, which applies only to releases prior to version 11.2.0.2, is related to tables without object statistics. For such tables, locks disappear when their object statistics are exported and imported.

## Logging of Management Operations

Many procedures in the dbms_stats package log information about their execution in the data dictionary. This logging information is externalized through the dba_optstat_operations and, in a 12.1 multitenant environment, cdb_optstat_operations views. Basically, you can know which operations were performed, when they were started, and how long they took. As of version 12.1, information about the status,[6] the session, and (optionally) the job associated with the operation is also available. The following example, an excerpt taken from a production database, shows that the gather_database_stats procedure was started every day and took between 9 and 18 minutes to run (note that April 5-6, 2014 is a weekend):

```
SQL> SELECT operation, start_time,
  2         (end_time-start_time) DAY(1) TO SECOND(0) AS duration
  3  FROM dba_optstat_operations
  4  ORDER BY start_time DESC;

OPERATION                   START_TIME                          DURATION
--------------------------- ----------------------------------- -----------
gather_database_stats(auto) 09-APR-14 10.00.08.877925 PM +02:00 +0 00:10:28
gather_database_stats(auto) 08-APR-14 10.00.02.899209 PM +02:00 +0 00:09:30
gather_database_stats(auto) 07-APR-14 10.00.04.119250 PM +02:00 +0 00:12:45
gather_database_stats(auto) 06-APR-14 10.05.00.173419 PM +02:00 +0 00:00:55
gather_database_stats(auto) 06-APR-14 06.04.46.957190 PM +02:00 +0 00:00:50
gather_database_stats(auto) 06-APR-14 02.04.32.438573 PM +02:00 +0 00:00:53
gather_database_stats(auto) 06-APR-14 10.04.16.208319 AM +02:00 +0 00:01:27
gather_database_stats(auto) 06-APR-14 06.00.09.299059 AM +02:00 +0 00:04:55
gather_database_stats(auto) 05-APR-14 10.03.28.888807 PM +02:00 +0 00:00:58
gather_database_stats(auto) 05-APR-14 06.03.14.637546 PM +02:00 +0 00:00:43
gather_database_stats(auto) 05-APR-14 02.02.59.997594 PM +02:00 +0 00:01:06
gather_database_stats(auto) 05-APR-14 10.02.46.052860 AM +02:00 +0 00:01:13
gather_database_stats(auto) 05-APR-14 06.00.03.801439 AM +02:00 +0 00:06:05
gather_database_stats(auto) 04-APR-14 10.00.03.068541 PM +02:00 +0 00:17:32
gather_database_stats(auto) 03-APR-14 10.00.02.781440 PM +02:00 +0 00:06:59
gather_database_stats(auto) 02-APR-14 10.00.02.702294 PM +02:00 +0 00:12:45
gather_database_stats(auto) 01-APR-14 10.00.03.254860 PM +02:00 +0 00:12:48
```

[6]Possible values are: PENDING, IN PROGRESS, COMPLETED, FAILED, SKIPPED, and TIMED OUT.

In addition, as of version 12.1, you can see the parameters with which an operation was executed. For example, the following query shows which parameters were used by the default gathering job during the last execution:

```
SQL> SELECT x.*
  2  FROM dba_optstat_operations o,
  3       XMLTable('/params/param'
  4                PASSING XMLType(notes)
  5                COLUMNS name VARCHAR2(20) PATH '@name',
  6                        value VARCHAR2(30) PATH '@val') x
  7  WHERE operation = 'gather_database_stats (auto)'
  8  AND start_time = (SELECT max(start_time)
  9                    FROM dba_optstat_operations
 10                    WHERE operation = 'gather_database_stats (auto)');

NAME                 VALUE
-------------------- ------------------------------
block_sample         FALSE
cascade              NULL
concurrent           FALSE
degree               NULL
estimate_percent     DEFAULT_ESTIMATE_PERCENT
granularity          DEFAULT_GRANULARITY
method_opt           DEFAULT_METHOD_OPT
no_invalidate        DBMS_STATS.AUTO_INVALIDATE
reporting_mode       FALSE
stattype             DATA
```

Be aware that log information is purged by the same mechanism as the statistics history described earlier. Both, therefore, have the same retention period.

## Strategies for Keeping Object Statistics Up-to-Date

The `dbms_stats` package provides many features for managing object statistics. The question is, how and when should you use them to achieve a successful configuration? Answering this question is difficult. Probably no definitive answer exists. In other words, there is no single method that can be implemented in all situations. Let's examine how to approach the problem.

The general rule, and probably the most important one, is that the query optimizer needs object statistics that describe the data stored in the database. As a result, when data changes, object statistics should change as well. As you may know, I am an advocate of gathering object statistics regularly. Those who are opposed to this practice argue that if a database is running well, there is no need to regather object statistics. The problem with that approach is that more often than not, some of the object statistics are dependent on the actual data. For example, one statistic that commonly changes is the low/high value of columns that contain data such as a timestamp associated to a transaction, a sale, or a phone call. True, not many of them change in typical tables, but usually those that do change are critical because they're used over and over again in the application. In practice, I run into many more problems caused by object statistics that aren't up-to-date than the other way around.

Obviously, it makes no sense to gather object statistics on data that never changes. Only stale object statistics should be regathered. Therefore, it's essential to take advantage of the feature that logs the number of modifications occurring to each table. In this way, you regather object statistics only for those tables experiencing substantial modifications. By default, a table is considered stale when more than 10% of the rows change. This is a good default value. As of version 11.1, this can be changed if necessary.

The frequency of gathering of statistics is also a matter of opinion. I have seen everything from hourly to monthly or even less frequently as being successful. It really depends on your data. In any case, when the staleness of the tables is used as a basis to regather object statistics, intervals that are too long can lead to an excess of stale objects, which in turn leads to excessive time required for statistics gathering and a peak in resource usage. For this reason, I like to schedule them frequently (to spread out the load) and keep single runs as short as possible. If your system has daily or weekly low-utilization periods, then scheduling runs during those periods is usually a good thing. If your system is a true 7×24 system, then it's usually better to use very frequent schedules (many times per day) to spread out the load as much as possible and avoid peaks.

If you have jobs that load or modify lots of data (for example, the ETL jobs in a data warehouse environment), you shouldn't wait for a scheduled gathering of object statistics. Simply make the gathering of statistics for the modified objects part of the job itself. In other words, if you know that something has substantially changed, trigger the gathering of statistics immediately.

If for some good reason object statistics shouldn't be gathered on some tables, you should lock them. In this way, the job that regularly gathers object statistics will simply skip them. This is much better than completely deactivating the job for the whole database.

You should take advantage of the default gathering job as much as possible. For this to meet your requirements, you should check, and if necessary change, the default configuration. Because a configuration at the object level is possible only as of version 11.1, if you have particular requirements for some tables in prior versions, you should schedule a job that processes them before the default job. In this way, the default job, which processes only those tables with stale statistics, will simply skip them. Locks might also be helpful to ensure that only a specific job is regathering object statistics on those critical tables.

If you're thinking about completely disabling the default gathering job instead, you should set the `autostats_target` preference to `oracle`. That way, you let the database engine take care of the data dictionary, and for the other tables, you set up a specific job that does what you expect.

If gathering statistics leads to inefficient execution plans, you can do two things. The first is to fix the problem by restoring the object statistics that were successfully in use before gathering statistics. The second is to find out why inefficient execution plans are generated by the query optimizer with the new object statistics. To do this, you should first check whether the newly gathered statistics correctly describe the data. For example, it's possible that sampling along with a new data distribution will lead to different histograms. If object statistics aren't good, then the gathering itself, or possibly a parameter used for their gathering, is the problem. If the object statistics are in fact good, there are two more possible causes. Either the query optimizer isn't correctly configured or the query optimizer is making a mistake. You have little control over the latter, but you should be able to find a solution for the former. In any case, you should avoid thinking too hastily that gathering object statistics is inherently problematic and, as a result, stop gathering them regularly.

The best practice is to gather object statistics with the `dbms_stats` package. However, there are situations where the correct object statistics may be misleading for the query optimizer. A common example is data for which a history must be kept online for a long time (for instance, in Switzerland some types of data must be kept for at least ten years). In such cases, if the data distribution hardly changes over the time, the object statistics gathered by the `dbms_stats` package should be fine. In contrast, if the data distribution is strongly dependent on the period and the application frequently accesses only a subset of the data, it could make sense to manually modify (that is, fudge) object statistics to describe the most relevant data. In other words, if you know something that the `dbms_stats` package ignores or isn't able to discover, it's legitimate to inform the query optimizer by fudging the object statistics.

## On to Chapter 9

This chapter describes what table statistics, column statistics, histograms, and index statistics are and how they in turn describe the data stored in the database. It also covers how to gather object statistics with the `dbms_stats` package and where to find them in the data dictionary.

This chapter gives little information about the utilization of object statistics. Such information is provided in Chapter 9, along with a description of the initialization parameters that configure the query optimizer. After reading that chapter, you should be able to correctly configure the query optimizer and, as a result, get efficient execution plans from it most of the time.

CHAPTER 9

# Configuring the Query Optimizer

The query optimizer is directly responsible for the performance of SQL statements. For this reason, it makes sense to take some time to configure it correctly. In fact, without an optimal configuration, the query optimizer may generate inefficient execution plans that lead to poor performance.

The configuration of the query optimizer consists not only of several initialization parameters but also of system statistics and object statistics (system and object statistics are described in Chapter 7 and Chapter 8, respectively). This chapter describes how these initialization parameters and statistics influence the query optimizer and presents a straightforward and pragmatic road map that will help you achieve a successful configuration.

---

**Caution** The formulas provided in this chapter, with a single exception, aren't published by Oracle. Several tests show that they're able to describe how the query optimizer estimates the cost of a given operation. In any case, they neither claim to be precise nor claim to be correct in all situations. They're provided here to give you an idea of how an initialization parameter or a statistic influences query optimizer estimations.

---

## To Configure or Not to Configure...

Adapting a Kenyan proverb[1] to our situation here, I'd say, "Configuring the query optimizer is costly, but it's worth the expense." In practice, I have seen too many sites that underestimate the importance of a good configuration. From time to time I even have heated discussions with people who say to me, "We don't need to spend time on individually configuring the query optimizer for each database. We already have a set of initialization parameters that we use over and over again on all our databases." My first reply is, frequently, something like this: "Why would Oracle introduce almost two dozen initialization parameters that are specific to the query optimizer if a single set works well on all databases? They know what they're doing. If such a magic configuration existed, they would provide it by default and make the initialization parameters undocumented." I then continue by carefully explaining that such a magic configuration doesn't exist because of these two reasons:

- Each application has its own requirements and workload profile.
- Each system, which is composed of different hardware and software components, has its own characteristics.

[1] The Kenyan proverb is "Peace is costly, but it is worth the expense." You can find this quote at www.quotationspage.com/quote/38863.html.

That said the query optimizer works well, meaning it generates good execution plans for most[2] SQL statements. Be careful, though, because this is true only on the condition that the query optimizer is correctly configured and the database has been designed to take advantage of all its features. I can't stress this enough. Also note that the configuration of the query optimizer includes not only the initialization parameters but the system statistics and object statistics as well.

## Configuration Road Map

Because there's no such thing as a magic configuration, we need a solid and reliable procedure to help us. Figure 9-1 sums up the main steps I go through. Their description is the following:

1. Two initialization parameters should always be adjusted: `optimizer_mode` and `db_file_multiblock_read_count`. As you'll see later in this chapter, the latter isn't always relevant for the query optimizer itself. Nevertheless, the performance of some operations may strongly depend on it.
2. Because the default values of the initialization parameters adjusted in this step are generally good, this step is optional. In any case, the aim of this step is to enable or disable specific features of the query optimizer.
3. Because system statistics and object statistics provide vital information to the query optimizer, they must be gathered.
4. By setting the `workarea_size_policy` initialization parameter, the choice is made between manual and automatic sizing of work areas provided to operations storing data in memory. Depending on the method chosen, other initialization parameters are set either in step 5 or in step 6.
5. If the sizing of work areas is automatic, the `pga_aggregate_target` initialization parameter is set. Optionally, as of version 12.1, the `pga_aggregate_limit` initialization parameter can be changed as well.
6. If the sizing of work areas is manual, the actual size depends on the type of operation using the memory. Basically, a specific initialization parameter is set for each type of operation.
7. When the first part of the configuration is in place, it's time to test the application. During the test, the execution plans are collected for the components that don't provide the required performance. By analyzing these execution plans, you should be able to infer what the problem is. Note that at this stage, it's important to recognize general, not individual, behavior. For example, you may notice that the query optimizer uses too many or too few indexes or doesn't recognize restrictions correctly.
8. If the query optimizer generates efficient execution plans for most SQL statements, the configuration is good. If not, you proceed to step 9.
9. If the query optimizer tends to use too many or too few indexes or nested loops, it's usually possible to adjust the `optimizer_index_caching` and `optimizer_index_cost_adj` initialization parameters to fix the problem. If the query optimizer makes big mistakes in the estimation of cardinalities, it's possible that some histograms are missing or need to be adjusted. Also adjusting dynamic sampling might help. As of version 11.1, extended statistics might also help.

[2] Perfection is unrealizable in software development as in almost any other activity you can imagine. This rule, even if neither you nor Oracle likes it, applies to the query optimizer as well. You should therefore expect that a small percentage of the SQL statements will require manual intervention (this topic is covered in Chapter 11).

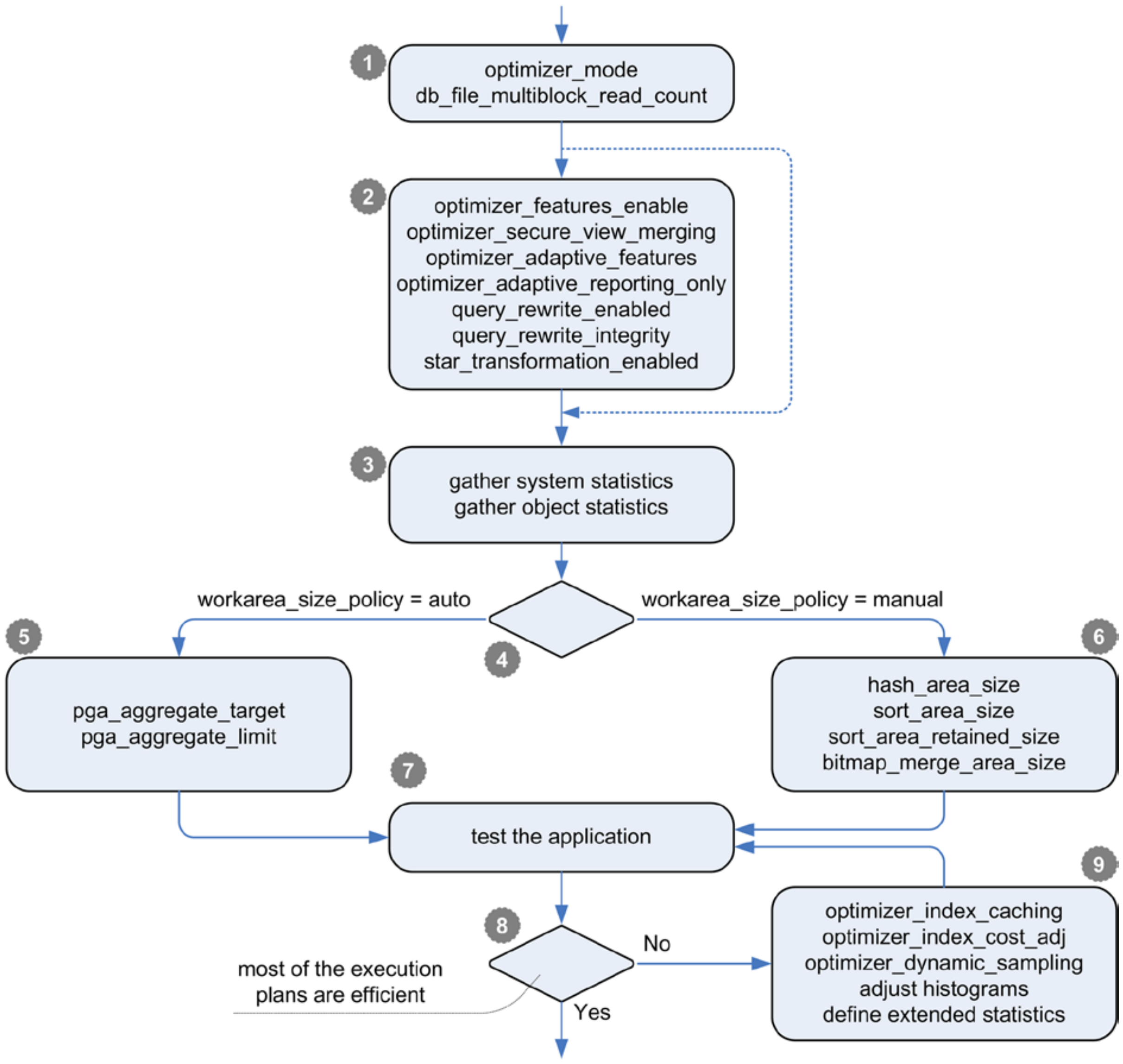

1
optimizer_mode
db_file_multiblock_read_count
2
optimizer_features_enable
optimizer_secure_view_merging
optimizer_adaptive_features
optimizer_adaptive_reporting_only
query_rewrite_enabled
query_rewrite_integrity
star_transformation_enabled
3
gather system statistics
gather object statistics
workarea_size_policy = auto
workarea_size_policy = manual
4
5
pga_aggregate_target
pga_aggregate_limit
6
hash_area_size
sort_area_size
sort_area_retained_size
bitmap_merge_area_size
7
test the application
8
No
Yes
most of the execution plans are efficient
9
optimizer_index_caching
optimizer_index_cost_adj
optimizer_dynamic_sampling
adjust histograms
define extended statistics

***Figure 9-1.*** *Main steps of the configuration road map*

According to Figure 9-1, the initialization parameters set in steps 1–6 can't be changed afterward. Of course, this isn't written in stone. If you can't achieve good results by adjusting the index-related initialization parameters or the histograms in step 9, it may be necessary to start over from the beginning. It's also worth mentioning that because several initialization parameters have an impact on system statistics, after changing them, it may be necessary to recompute the system statistics.

## Set the Right Parameter!

It's obvious that Oracle doesn't just randomly provide new initialization parameters. Instead, each initialization parameter is introduced to control a very specific feature or behavior of the query optimizer. At the risk of repeating myself, I must remind you that Oracle's introduction of a parameter implies that no single value can be applied to all situations. Thus, for each initialization parameter, you must infer a sensible value from both the application workload profile and the system where the database engine runs.

To perform a successful configuration of the query optimizer, it's essential to understand how it works and the impact each initialization parameter has on it. With this knowledge, instead of tweaking the configuration randomly or copying the "good values" from an article you found recently on the Internet, you should do the following:

- Understand the current situation. For example, why has the query optimizer chosen an execution plan that is suboptimal?
- Determine the goal to be achieved. In other words, what execution plan do you want to achieve?
- Find out which initialization parameters, or possibly which statistics, should be rectified to achieve the goal you set. Of course, in some situations it isn't enough to set initialization parameters only. It may be necessary to modify the SQL statement and/or the database design as well.

The following sections describe how some of the initialization parameters referenced by the configuration road map of Figure 9-1 work and also give advice on how to find good values for your system. The parameters that aren't described in this chapter are covered elsewhere in the book while covering the feature they control. Parameters are divided into two groups: one where only the operation of the query optimizer is affected, and the other where parameters have to do with the *program global area* (PGA).

## Query Optimizer Parameters

The following sections describe a number of parameters related to the operation of the query optimizer.

### optimizer_mode

The `optimizer_mode` initialization parameter is essential because with it you specify what the word *efficient* means to the query optimizer. Generally speaking, it may mean "faster," "using fewer resources," or perhaps something else. Because with a database you're processing data, you'll usually want to process it as fast as possible. Therefore, the meaning of *efficient* should be "the fastest way to execute a SQL statement without wasting unnecessary resources." What that means for a SQL statement that's always completely executed (for example, an `INSERT` statement) is clear. For a query, on the other hand, there are subtle differences. An application, for example, isn't obliged to fetch all rows returned by a query. In other words, queries might be only partially executed.

Let me give you an example unrelated to Oracle Database. When I Google the term *query optimizer*, I get the matching pages in best-ranking order, beginning with a page listing the first ten results (the ten best ones). On the same page, I'm informed that there are about 986,000 pages in the result set and that the search took 0.26 seconds. This is a good example of processing that is optimized to deliver the initial data as fast as possible, because the first few pages are almost always the only ones that will be actually accessed by the users. To access one of the pages, I then click the corresponding link. At this point I'm usually not interested in getting only the first few lines. I want the whole page to be available and correctly formatted, at which point I'll start reading. In this case, the processing should be optimized to provide all data as fast as possible and not only pieces. Every application (or part of it) falls into one of these two categories. Either fast delivery of the first part of the result set is important or the fast delivery of the whole result set (that is equivalent to the fast delivery of the last row) is important.

To choose the value of the `optimizer_mode` initialization parameter, you have to ask yourself whether it's more important for the query optimizer to produce execution plans for the fast delivery of the first row or the fast delivery of the last row:

- If fast delivery of the last row is important, you should use the `all_rows` value. This is the most commonly used configuration.
- If fast delivery of the first row(s) is important, you should use the `first_rows_`*n* value (where *n* is 1, 10, 100, or 1,000 rows). This configuration should be used only when the application partially fetches result sets that are larger than the number of rows specified with the parameter. For existing applications, you can check this by comparing the `executions` and `end_of_fetch_count` columns in the `v$sqlarea` view. Notice that the older implementation of the first row optimizer (that is, configured through the value `first_rows`) should no longer be used. In fact, it's provided for backward compatibility only.

The default value is `all_rows`. Also note that `INSERT`, `DELETE`, `MERGE` and `UPDATE` statements are always optimized with `all_rows`. It makes sense to do so because those SQL statements must process all rows before they return control to the caller.

---

■ **Caution** The key idea of the first row optimization is to avoid blocking operations (that is, operations that don't produce rows until they have run to completion). For this purpose, nested loop joins are generally preferred over hash joins (which block until a hash table has been built) and merge joins (which block until both inputs have been sorted). In addition, in some situations, `ORDER BY` operations (which block until the rows have been sorted) are replaced by index range scans. For large result sets, it's unlikely that the first row optimization leads to optimal performance. So it's of paramount importance to use first row optimization only when large result sets are only partially fetched by the calling application.

---

The `optimizer_mode` initialization parameter is dynamic and can be changed at the instance and session levels. In a 12.1 multitenant environment, it can also be set at the PDB level. In addition, with one of the following hints, it's possible to set it at the statement level:

- `all_rows`
- `first_rows(`*n*`)` where *n* is any natural number greater than 0

## optimizer_features_enable

In every database version, Oracle introduces or enables new features in the query optimizer. If you're upgrading to a new database version and want to keep the old behavior of the query optimizer, it's possible to set the `optimizer_features_enable` initialization parameter to the database version from which you're upgrading. Unfortunately, not all new features are disabled by this initialization parameter. For example, if you set it to 10.2.0.4 in version 11.2, you won't get exactly the 10.2.0.4 query optimizer. For this reason, I usually advise using the default value, which is the same as the version number used for the database binary files. Also Oracle Support, through the *Use Caution if Changing the OPTIMIZER_FEATURES_ENABLE Parameter After an Upgrade* note (1362332.1), provides similar advice.

---

■ **Tip** Changing the default value of the `optimizer_features_enable` initialization parameter is only a short-term workaround. Sooner or later the application should be adapted (optimized) for the new database version.

---

Valid values for the `optimizer_features_enable` initialization parameter are database versions such as 10.2.0.5, 11.1.0.7, or 11.2.0.3. Because the documentation (specifically the *Oracle Database Reference* manual) isn't up-to-date for each patch level for this parameter, it's possible to generate the actual supported values with the following SQL statement:

```
SQL> SELECT value
  2  FROM v$parameter_valid_values
  3  WHERE name = 'optimizer_features_enable';

VALUE
----------
8.0.0
8.0.3
8.0.4
...
11.2.0.2
11.2.0.3
11.2.0.3.1
```

The `optimizer_features_enable` initialization parameter is dynamic and can be changed at the instance and session levels. In a 12.1 multitenant environment, it can also be set at the PDB level. In addition, it's possible to specify a value at the statement level with the `optimizer_features_enable` hint. The following two examples show the hint used to specify the default value and a specific value, respectively (refer to Chapter 11 for more information about hints):

- `optimizer_features_enable(default)`
- `optimizer_features_enable('10.2.0.5')`

## db_file_multiblock_read_count

The maximum disk I/O size used by the database engine during multiblock reads (for example, full table scans or index fast full scans) is determined by multiplying the values of the `db_block_size` and `db_file_multiblock_read_count` initialization parameters. Thus, the maximum number of blocks read during multiblock reads is determined by dividing the maximum disk I/O size by the block size of the tablespace being read. In other words, for the default block size, the `db_file_multiblock_read_count` initialization parameter specifies the maximum number of blocks read. This is "only" a maximum because there are at least three common situations leading to multiblock reads that are smaller than the value specified by this initialization parameter:

- Segment headers and other blocks containing only segment metadata like extent maps are read with single-block reads.
- Physical reads—except for a special case related to direct reads performed against a tablespace using auto segment space management—never span several extents.
- Blocks already in the buffer cache, except for direct reads, aren't reread from the disk I/O subsystem.

To illustrate, Figure 9-2 shows the structure of a segment stored in a tablespace using manual segment space management. Like any segment, it's composed of extents (in this example, 2), and each extent is composed of blocks (in this example, 16). The first block of the first extent is the segment header. Some blocks (4, 9, 10, 19, and 21) are cached in the buffer cache. A database engine process executing buffer cache reads of this segment can't perform a single physical multiblock read, even if the `db_file_multiblock_read_count` initialization parameter is set to a value greater than or equal to 32.

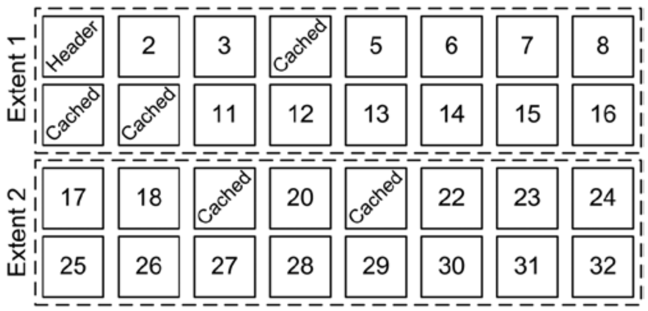


***Figure 9-2.*** *Structure of a data segment*

If the `db_file_multiblock_read_count` initialization parameter is set to 8, the following buffer cache reads are performed:

- One single-block read of the segment header (block 1).
- One multiblock read of two blocks (2 and 3). More blocks can't be read because block 4 is cached.
- One multiblock read of four blocks (from 5 to 8). More blocks can't be read because block 9 is cached.
- One multiblock read of six blocks (from 11 to 16). More blocks can't be read because block 16 is the last one of the extent.
- One multiblock read of two blocks (17 and 18). More blocks can't be read because block 19 is cached.
- One single-block read of block 20. More blocks can't be read because block 21 is cached.
- One multiblock read of eight blocks (from 22 to 29). More blocks can't be read because the `db_file_multiblock_read_count` initialization parameter is set to 8.
- One multiblock read of three blocks (from 30 to 32).

In summary, the process performs two single-block reads and six multiblock reads. The average number of blocks read by a multiblock read is about four. The fact that the average size is smaller than eight explains why Oracle introduced the `mbrc` value in system statistics.

The `db_file_multiblock_read_count` initialization parameter is dynamic and can be changed at the instance and session levels. In a 12.1 multitenant environment, it can also be set at the PDB level.

At this point, it's also important to discuss how the query optimizer computes the cost of multiblock read operations (for example, full table scans or index fast full scans).

When workload system statistics are available, the I/O cost isn't dependent on the value of the `db_file_multiblock_read_count` initialization parameter. It's computed by Formula 9-1. Note that `mreadtim` is divided by `sreadtim` because the query optimizer normalizes the costs according to single-block reads, as already discussed in Chapter 7 (Formula 7-2).

***Formula 9-1.*** I/O cost of multiblock read operations with workload statistics

$$io\_cost \approx \frac{blocks}{mbrc} \cdot \frac{mreadtim}{sreadtim}$$

In Formula 9-1, with noworkload statistics, the variables are replaced as follows:

- Provided that the db_file_multiblock_read_count initialization parameter is explictly set, mbrc is replaced by the value of the db_file_multiblock_read_count initialization parameter; otherwise, 8 is used.
- sreadtim is replaced with the value computed by Formula 7-3.
- mreadtim is replaced with the value computed by Formula 7-4.

This means that the db_file_multiblock_read_count initialization parameter has a direct impact on the cost of multiblock read operations only when noworkload statistics are used. This also means values that are too high may lead to excessive full scans or at least an underestimation of the cost of multiblock read operations. Further, this is another situation where workload statistics are superior to noworkload statistics.

Now that you've seen the costing formulas, you need to know how to find out the value the db_file_multiblock_read_count initialization parameter should be set to. The most important thing is to recognize that multiblock reads have a big impact on performance. Therefore, the db_file_multiblock_read_count initialization parameter should be carefully set to achieve best performance. Even though values that lead to a disk I/O size of 1MB usually provide performance close to the best, sometimes higher or lower values are better. In addition, higher values usually require less CPU to issue the disk I/O operations. A simple full table scan with different values gives useful information about the impact of this initialization parameter and, therefore, helps find an optimal value. The following PL/SQL code snippet, which is an excerpt of the assess_dbfmbrc.sql script, could be used for that purpose:

```
BEGIN
  dbms_output.put_line('dbfmbrc blocks seconds cpu');
  FOR i IN 0..10
  LOOP
    l_dbfmbrc := power(2,i);

    EXECUTE IMMEDIATE 'ALTER SESSION SET db_file_multiblock_read_count = '||l_dbfmbrc;
    EXECUTE IMMEDIATE 'ALTER SYSTEM FLUSH BUFFER_CACHE';

    SELECT sum(decode(name, 'physical reads', value)),
           sum(decode(name, 'CPU used by this session', value))
    INTO l_starting_blocks, l_starting_cpu
    FROM v$mystat ms JOIN v$statname USING (statistic#)
    WHERE name IN ('physical reads','CPU used by this session');

    l_starting_time := dbms_utility.get_time();

    SELECT count(*) INTO l_count FROM t;

    l_ending_time := dbms_utility.get_time();

    SELECT sum(decode(name, 'physical reads', value)),
           sum(decode(name, 'CPU used by this session', value))
    INTO l_ending_blocks, l_ending_cpu
    FROM v$mystat ms JOIN v$statname USING (statistic#)
    WHERE name IN ('physical reads','CPU used by this session');

    l_time := round((l_ending_time-l_starting_time)/100,1);
    l_blocks := l_ending_blocks-l_starting_blocks;
    l_cpu := l_ending_cpu-l_starting_cpu;
    dbms_output.put_line(l_dbfmbrc||' '||l_blocks||' '||to_char(l_time)||' '||to_char(l_cpu));
  END LOOP;
END;
```

As you can see, it isn't that difficult to do. In any case, be careful not to cache the test table at the operating system and disk I/O subsystem levels, because that would render the test useless. The easiest way to avoid that is to use a table larger than the largest cache available in your system. For systems expected to use parallel processing, it's worth extending such a test to execute parallel queries as well (Chapter 15 describes parallel processing).

Figure 9-3 shows the characteristics of my test system measured with the previous PL/SQL block executed against an 11.2 database with all the initialization parameters set to the default. Here are the characteristics to note:

The throughput increases from about 200MB/s for small values of the `db_file_multiblock_read_count` initialization parameter up to more than 600MB/s for very large values.

The CPU utilization decreases from about 1.5 seconds for small values of the `db_file_multiblock_read_count` initialization parameter down to less than 0.5 seconds for very large values.

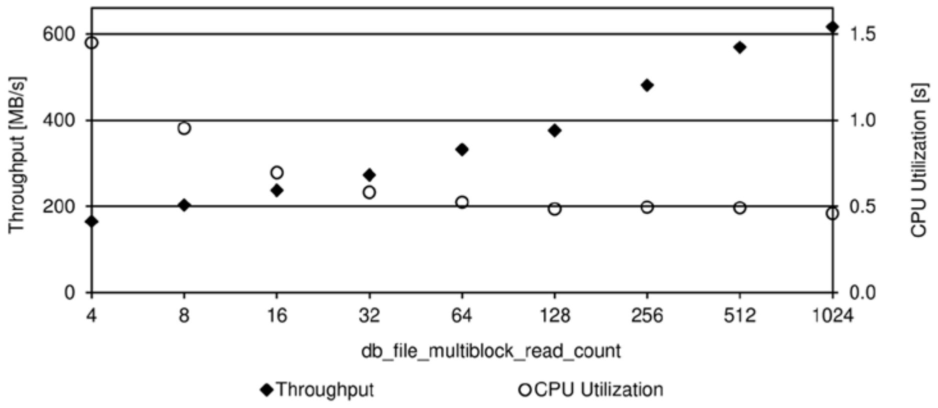


***Figure 9-3.*** *Impact of disk I/O size on the performance of a full table scan on four different systems*

It's also possible to instruct the database engine to automatically configure the value of the `db_file_multiblock_read_count` initialization parameter. To use this feature, simply don't set it. As shown in Formula 9-2, the database engine will then try to set it to a value that allows 1MB physical reads. At the same time, however, a kind of sanity check is applied to reduce the value if the size of the buffer cache is quite small compared to the number of sessions supported by the database.

***Formula 9-2.*** Default value of the `db_file_multiblock_read_count` initialization parameter

$$db\_file\_multiblock\_read\_count \approx \text{least}\left(\frac{1048576}{db\_block\_size}, \frac{db\_cache\_size}{sessions \cdot db\_block\_size}\right)$$

Because, as described earlier, physical reads of 1MB aren't always the ones that perform better, I'd advise against using this feature. It's better to find out the optimal value case by case.

Be aware that if noworkload statistics are used with this automatic configuration, `mbrc` isn't replaced by the automatically configured value in Formula 9-1. Instead, the value 8 is used.

## optimizer_dynamic_sampling

The query optimizer used to base its estimations solely on object statistics stored in the data dictionary. With *dynamic sampling*, that has changed. In fact, some statistics may be dynamically gathered during the parse phase as well.

This means that to gather additional information, some (sampling) queries are executed against the referenced objects. Unfortunately, the statistics gathered by dynamic sampling are neither stored in the data dictionary nor stored elsewhere. The only way to virtually reuse them is to reuse the shared cursor itself. Also note that the statistics gathered by dynamic sampling aren't necessarily used. In fact, the query optimizer performs several sanity checks to decide whether they should be used or not.

---

■ **Note** As of version 12.1, the term *dynamic statistics* is used instead of dynamic sampling. In this book I always use the old name.

---

The value (also called *level*) of the `optimizer_dynamic_sampling` initialization parameter specifies how and when dynamic sampling is used. Table 9-1 summarizes the accepted values and their meanings. Note that the default value depends on the `optimizer_features_enable` initialization parameter:

- If `optimizer_features_enable` is set to 10.0.0 or higher, the default is level 2.
- If `optimizer_features_enable` is set to 9.2.0, the default is level 1.
- If `optimizer_features_enable` is set to 9.0.1 or lower, dynamic sampling is disabled.

***Table 9-1.*** *Dynamic Sampling Levels and Their Meaning*

| Level | When Is Dynamic Sampling Used? | Number of Blocks* |
|---|---|---|
| 0 | Dynamic sampling is disabled. | 0 |
| 1 | Dynamic sampling is used for tables without object statistics. However, this occurs only if the following three conditions are met: the table has no index, it's part of a join (also subquery or nonmergeable view), and it has more blocks below the high watermark than the number of blocks used for the sampling. | 32 |
| 2 | Dynamic sampling is used for all tables without object statistics. | 64 |
| 3 | Dynamic sampling is used for all tables fulfilling the level-2 criterion and, in addition, for which a guess is used to estimate the selectivity of a predicate. | 32 or 64 |
| 4 | Dynamic sampling is used for all tables fulfilling the level-3 criterion and, in addition, having two or more columns referenced in a `WHERE` clause. | 32 or 64 |
| 5 | The same as level 4. | 64 |
| 6 | The same as level 4. | 128 |
| 7 | The same as level 4. | 256 |
| 8 | The same as level 4. | 1024 |
| 9 | The same as level 4. | 4096 |
| 10 | The same as level 4. | All |
| 11 | The query optimizer decides when and how to use dynamic sampling. This level is available as of version 12.1 only. | Automatically determined |

** This is the number of blocks used for sampling when dynamic sampling is activated with the initialization parameter or the hint using the statement-level syntax. For levels 3 and 4, if object statistics are available, then 32 blocks are sampled; otherwise, 64 blocks are sampled. When a hint using the object-level syntax is used, and for levels from 1 to 9, the number of blocks is computed with the following formula:* $32 \cdot 2^{(level-1)}$.

The `optimizer_dynamic_sampling` initialization parameter is dynamic and can be changed at the instance level and at session level. In a 12.1 multitenant environment, it can also be set at the PDB level. In addition, it's possible to specify a value at the statement level with the `dynamic_sampling` hint. The hint supports two syntaxes:

- The statement-level syntax overrides the value of the `optimizer_dynamic_sampling` initialization parameter: `dynamic_sampling(`*`level`*`)`
- The object-level syntax activates dynamic sampling for a specific table only: `dynamic_sampling(table_alias` *`level`*`)`

---

■ **Caution** When dynamic sampling is activated through the hint using the object-level syntax, the sampling always takes place. In other words, the query optimizer doesn't check whether the rules mentioned in Table 9-1 are fulfilled. However, depending on whether object statistics are already available, the sampled statistics might be discarded. All this can be an unnecessary overhead, so I don't recommend using the object-level syntax.

---

As of version 11.2, if the `optimizer_dynamic_sampling` initialization parameter is set to the default, then the query optimizer automatically decides how and when dynamic sampling is used for SQL statements executed in parallel. This is done because parallel SQL statements are expected to consume a lot of resources, and it's therefore critical to have the best possible execution plan.

The query optimizer can use dynamic sampling to gather two types of statistics. The first type includes the following:

- The number of blocks below the high watermark of a segment
- The number of rows in a table
- The number of distinct values in a column
- The number of `NULL` values in a column

As you can see, the statistics of the first type are equivalent to corresponding object statistics that should already be available in the data dictionary. As a result, the statistics gathered by dynamic sampling would only be useful if the object statistics are either missing or inaccurate (stale). But be aware: by default, the statistics of the first type are gathered only for objects that are missing object statistics. However, you can force the matter by specifying the `dynamic_sampling_est_cdn(table_alias)` hint. You might do so if the statistics are present but inaccurate. The hint forces them to be gathered when they would otherwise not be.

The second type of statistics gathered by dynamic sampling includes the following:

- The selectivity of a predicate
- The cardinality of a join (from version 12.1 onward only)
- The cardinality of an aggregation (from version 12.1 onward only)

Because these statistics go beyond the information provided through object statistics (despite the fact that in some cases the selectivity of a predicate can be obtained through extended statistics), they're intended to increase the information that object statistics can provide. With them, the query optimizer might be able to perform better estimations.

Here are some examples (excerpts of the output generated by the `dynamic_sampling_levels.sql` script run in version 11.2.0.3) illustrating in which situations the values between 1 and 4 lead to dynamic sampling. The tables used for the tests are created with the following SQL statements. Initially, they have no object statistics. Note that the only difference between the `t_noidx` table and the `t_idx` table is that the latter has a primary key (and therefore an index):

```
CREATE TABLE t_noidx (id, n1, n2, pad) AS
SELECT rownum,
       rownum,
       cast(round(dbms_random.value(1,100)) AS VARCHAR2(100)),
       cast(dbms_random.string('p',1000) AS VARCHAR2(1000))
FROM dual
CONNECT BY level <= 1000

CREATE TABLE t_idx (id CONSTRAINT t_idx_pk PRIMARY KEY, n1, n2, pad) AS
SELECT *
FROM t_noidx
```

The following are the first test queries. The only difference between them is that the first one references the `t_noidx` table, and the second references the `t_idx` table:

```
SELECT *
FROM t_noidx t1, t_noidx t2
WHERE t1.id = t2.id AND t1.id < 19

SELECT *
FROM t_idx t1, t_idx t2
WHERE t1.id = t2.id AND t1.id < 19
```

If the level is set to 1, dynamic sampling is performed only for the first query because the table referenced by the second one is indexed. The following is the recursive query used to gather the statistics for the `t_noidx` table on my test database. Some hints have been removed and bind variables replaced with literals to make it easier to read. Note that SQL trace was activated before executing the test queries. Then all I needed to do was inspect the generated trace file to find out which recursive SQL statements were executed:

```
SELECT NVL(SUM(C1),0),
       NVL(SUM(C2),0),
       COUNT(DISTINCT C3),
       NVL(SUM(CASE WHEN C3 IS NULL THEN 1 ELSE 0 END),0)
FROM (
  SELECT 1 AS C1,
         CASE WHEN "T1"."ID"<19 THEN 1 ELSE 0 END AS C2,
         "T1"."ID" AS C3
  FROM "CHRIS"."T_NOIDX" SAMPLE BLOCK (20 , 1) SEED (1) "T1"
) SAMPLESUB
```

Here are the significant points to notice:

- The query optimizer counts the total number of rows, the number of rows in the range specified in the `WHERE` clause (`id < 19`), and the number of distinct values and `NULL` values of the `id` column.
- The values used in the query must be known. If bind variables are used, the query optimizer must be able to peek the values to perform dynamic sampling.
- The `SAMPLE` clause is used to perform the sampling. `t_noidx` table has 155 blocks on my database, so the sampling percentage is 20 percent (32/155).

---

■ **Caution** Depending on the data you're working with, level 6 or 7 might be required to ensure that dynamic sampling generates useful insights. After all, even at level 7, at most 256 blocks are sampled. Depending on the amount of data and its distribution, sampling a small number of blocks might not correctly represent the whole content of a table.

---

If the level is set to 2, dynamic sampling is performed for both test queries because, at that level, dynamic sampling is always used when object statistics are missing. The recursive query used to gather the statistics for both tables is the same as the one shown earlier. The sampling percentage increases because, at that level, it's based on 64 blocks instead of 32. In addition, for the `t_idx` table, the following recursive query is executed as well. Its aim is to scan the index instead of the table as in the previous query. This is done because a quick sampling on the table may miss the rows in the range specified by the predicate present in the `WHERE` clause. Instead, a quick scan of the index will certainly locate them, if they exist:

```
SELECT NVL(SUM(C1),0),
       NVL(SUM(C2),0),
       NVL(SUM(C3),0)
FROM (
  SELECT 1 AS C1,
         1 AS C2,
         1 AS C3
  FROM "CHRIS"."T_IDX" "T1"
  WHERE "T1"."ID"<19
  AND ROWNUM <= 2500
) SAMPLESUB
```

The next level of dynamic sampling is 3. Starting with that level, dynamic sampling is also used when object statistics are available in the data dictionary. Before executing further tests, object statistics were gathered with the following PL/SQL block:

```
BEGIN
  dbms_stats.gather_table_stats(ownname    => user,
                                tabname    => 't_noidx',
                                method_opt => 'for all columns size 1');
  dbms_stats.gather_table_stats(ownname    => user,
                                tabname    => 't_idx',
                                method_opt => 'for all columns size 1',
                                cascade    => true);
END;
```

If the level is set to 3 or higher, the query optimizer performs dynamic sampling to estimate the selectivity of a predicate by measuring the selectivity over a sample of the table rows, instead of using the statistics from the data dictionary and possibly hard-coded values. The following two queries illustrate this:

```
SELECT *
FROM t_idx
WHERE id = 19

SELECT *
FROM t_idx
WHERE round(id) = 19
```

For the first one, the query optimizer is able to estimate the selectivity of the `id=19` predicate based on the column statistics and histograms. Thus, no dynamic sampling is necessary. Instead, for the second one (except if extended statistics for the `round(id)` expression are in place), the query optimizer can't infer the selectivity of the `round(id)=19` predicate. In fact, the column statistics and histograms provide information only about the `id` column itself, not about its rounded values. The following query is the one used for the dynamic sampling. As you can see, it has the same structure as one discussed earlier. The `c2` and `c3` columns are different because the `WHERE` clause of the

SQL statement leading to dynamic sampling is different. Because an expression is applied to the indexed column (`id`), even with the `t_idx` table, no sampling on the index is performed in this specific case:

```
SELECT NVL(SUM(C1),0),
       NVL(SUM(C2),0),
       COUNT(DISTINCT C3)
FROM (
  SELECT 1 AS C1,
         CASE WHEN ROUND("T_IDX"."ID")=19 THEN 1 ELSE 0 END AS C2,
         ROUND("T_IDX"."ID") AS C3
  FROM "CHRIS"."T_IDX" SAMPLE BLOCK (20 , 1) SEED (1) "T_IDX"
) SAMPLESUB
```

If the level is set to 4 or higher, the query optimizer performs dynamic sampling also when two or more columns of the same table are referenced in the `WHERE` clause. This is useful for improving estimations in the case of correlated columns. The following query provides an example of this. If you look back at the SQL statements used to create the test tables, you'll notice that the `id` and `n1` columns contain the same data:

```
SELECT *
FROM t_idx
WHERE id < 19 AND n1 < 19
```

Also in this case, the query optimizer performs dynamic sampling with a query that has the same structure as the previous ones. Once more, the main difference is because of the `WHERE` clause of the SQL statement that causes dynamic sampling:

```
SELECT NVL(SUM(C1),0),
       NVL(SUM(C2),0)
FROM (
  SELECT 1 AS C1,
         CASE WHEN "T_IDX"."ID"<19 AND "T_IDX"."N1"<19 THEN 1 ELSE 0 END AS C2
  FROM "CHRIS"."T_IDX" SAMPLE BLOCK (20 , 1) SEED (1) "T_IDX"
) SAMPLESUB
```

Summing up, you can see that level 1 and 2 are usually not very useful. In fact, tables and indexes should have up-to-date object statistics. A common exception is when temporary tables—implemented either as global temporary tables or as regular tables—containing temporary data are accessed. In fact, frequently no object statistics are available for them. An exception to this is when, in version 12.1, you take advantage of the session-level statistics. In any case, be aware that a session can share a cursor parsed by another session even if, at the moment it's used, the segment associated to the temporary table contains very different sets of data. Level 3 and higher levels are useful for improving selectivity estimations of "complex" predicates. Therefore, if the query optimizer can't make correct estimations because of "complex" predicates, set the `optimizer_dynamic_sampling` initialization parameter to 4 or higher values. Otherwise, leave it at the default value. In addition, as mentioned in Chapter 8, as of version 11.1 it's possible to gather statistics on expressions and groups of columns. So in some situations, it should be possible to avoid dynamic sampling.

## optimizer_index_cost_adj

The `optimizer_index_cost_adj` initialization parameter is used to change the cost of table accesses through index scans. Valid values go from 1 to 10,000. The default is 100. Values greater than 100 make index scans more expensive and as a result favor full table scans. Values less than 100 make index scans less expensive.

To understand the effect of this initialization parameter on the costing formula, it's useful to describe how the query optimizer computes costs related to table accesses based on *index range scans*.

An index range scan is an index lookup of several keys. As shown in Figure 9-4, the following operations are carried out:

1. Access the root block of the index.
2. Go through the branch blocks to locate the leaf block containing the first keys.
3. For each key fulfilling the search criteria, do the following:
    a. Extract the rowid referencing the data block.
    b. Access the data block referenced by the rowid.

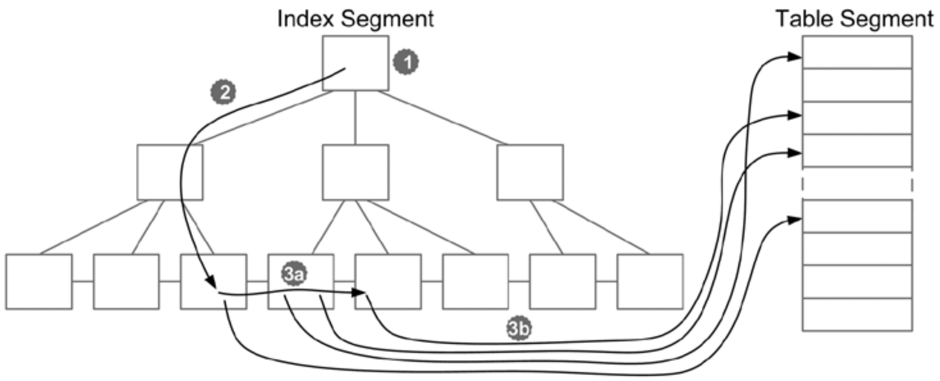


***Figure 9-4.*** *Operations carried out during table accesses based on index range scans*

The number of physical reads performed by an index range scan is equal to the number of branch blocks accessed to locate the leaf block containing the first key (which is the `blevel` statistic), plus the number of leaf blocks that are scanned (the `leaf_blocks` statistic multiplied by the selectivity of the operation), plus the number of data blocks accessed via rowid (the `clustering_factor` statistic multiplied by the selectivity of the operation). This gives you Formula 9-3, where, in addition, the correction applied by the `optimizer_index_cost_adj` initialization parameter is taken into consideration.

***Formula 9-3.*** I/O cost of table accesses based on index range scans

$$io\_cost \approx \left(blevel + \left(leaf\_blocks + clustering\_factor\right) \cdot selectivity\right) \cdot \frac{optimizer\_index\_cost\_adj}{100}$$

■ **Note** In Formula 9-3, the same selectivity is used to compute the cost of the index access (operation 3a in Figure 9-4) and the cost of the table access (operation 3b). In reality, the query optimizer might use two distinct selectivities for these two distinct costs. This is necessary when only part of the filter is applied through the index access. For example, this happens when an index is composed of three columns and the second one has no restriction.

In summary, you see that the optimizer_index_cost_adj initialization parameter has a direct impact on the I/O cost of an index access. When it's set to a value less than the default, all costs decrease proportionally. In some cases this might be a problem because the query optimizer rounds off the results of its estimations. This means that, even if the object statistics of several indexes are different, they may have the same cost as far as the query optimizer is concerned. If several costs are equal in value, the query optimizer decides based on the name of the indexes! It simply takes the first one in alphabetical order. This problem is demonstrated in the following example. Notice how the index used for the INDEX RANGE SCAN operation changes when the optimizer_index_cost_adj initialization parameter and the index name changes. This is an excerpt of the output generated by the optimizer_index_cost_adj.sql script:

```
SQL> ALTER SESSION SET OPTIMIZER_INDEX_COST_ADJ = 100;

SQL> SELECT * FROM t WHERE val1 = 11 AND val2 = 11;

-----------------------------------------------
| Id  | Operation                   | Name     |
-----------------------------------------------
|   0 | SELECT STATEMENT            |          |
|*  1 |  TABLE ACCESS BY INDEX ROWID| T        |
|*  2 |   INDEX RANGE SCAN          | T_VAL2_I |
-----------------------------------------------

   1 - filter("VAL1"=11)
   2 - access("VAL2"=11)

SQL> ALTER SESSION SET OPTIMIZER_INDEX_COST_ADJ = 10;

-----------------------------------------------
| Id  | Operation                   | Name     |
-----------------------------------------------
|   0 | SELECT STATEMENT            |          |
|*  1 |  TABLE ACCESS BY INDEX ROWID| T        |
|*  2 |   INDEX RANGE SCAN          | T_VAL1_I |
-----------------------------------------------

   1 - filter("VAL2"=11)
   2 - access("VAL1"=11)

SQL> ALTER INDEX t_val1_i RENAME TO t_val3_i;

SQL> SELECT * FROM t WHERE val1 = 11 AND val2 = 11;

-----------------------------------------------
| Id  | Operation                   | Name     |
-----------------------------------------------
|   0 | SELECT STATEMENT            |          |
|*  1 |  TABLE ACCESS BY INDEX ROWID| T        |
|*  2 |   INDEX RANGE SCAN          | T_VAL2_I |
-----------------------------------------------

   1 - filter("VAL1"=11)
   2 - access("VAL2"=11)
```

To avoid this kind of instability, I usually don't recommend setting the `optimizer_index_cost_adj` initialization parameter to low values. It's also important to mention that system statistics provide an adjustment of the cost associated to full table scans. This means that if system statistics are in place, the default value is usually good. Also notice that system statistics don't have the same drawbacks as this parameter does, because they increase the costs instead of decreasing them.

The `optimizer_index_cost_adj` initialization parameter is dynamic and can be changed at the instance and session levels. In a 12.1 multitenant environment, it can also be set at the PDB level.

## optimizer_index_caching

The `optimizer_index_caching` initialization parameter is used to specify the expected amount (in percent) of index blocks cached in the buffer cache during the execution of in-list iterators and nested loop joins. It's important to note that the value of this initialization parameter is used by the query optimizer to adjust its estimations only. In other words, it doesn't specify how much of each of the indexes should be cached by the database engine. Valid values range from 0 to 100. The default is 0. Values greater than 0 decrease the cost of index scans performed for in-list iterators and in the inner loop of nested loop joins. Because of this, the `optimizer_index_caching` parameter is used to increase the utilization of these operations.

Formula 9-4 shows the correction applied to the index range scan costing formula presented in the previous section (Formula 9-3).

***Formula 9-4.*** I/O cost of table accesses based on index range scans

$$io\_cost \approx \left\{ \left( blevel + leaf\_blocks \cdot selectivity \right) \cdot \left( 1 - \frac{optimizer\_index\_caching}{100} \right) + clustering\_factor \cdot selectivity \right\} \cdot \frac{optimizer\_index\_cost\_adj}{100}$$

This initialization parameter shares some of the drawbacks described in the previous section about the `optimizer_index_cost_adj` initialization parameter. Nevertheless, its impact is less widespread mainly because of two reasons. First, it's used only for nested loops and in-list iterators. Second, it has no impact on the clustering factor part of the costing formula used for index range scans (Formula 9-4). Because the clustering factor is frequently the biggest factor in the costing formula, it's less likely that this initialization parameter leads to wrong decisions. In summary, this initialization parameter has less impact on the query optimizer than the `optimizer_index_cost_adj` initialization parameter does. That said, the default value is usually good.

The `optimizer_index_caching` initialization parameter is dynamic and can be changed at the instance and session levels. In a 12.1 multitenant environment, it can also be set at the PDB level.

## optimizer_secure_view_merging

The `optimizer_secure_view_merging` initialization parameter is available to control query transformations like view merging and predicate move around (refer to Chapter 6 for a description of these query transformations). It can be set to either `FALSE` or `TRUE`. The default is `TRUE`.

- `FALSE` allows the query optimizer to apply query transformations without checking whether doing so could lead to security issues.
- `TRUE` allows the query optimizer to apply query transformations only when doing so won't lead to security issues.

■ **Note** Because of its name, you might think that the `optimizer_secure_view_merging` initialization parameter is only relevant for view merging. However, it controls all query transformations that might lead to security issues. It has that name for a very simple reason: when it was implemented, it controlled only view merging.

To understand the impact of this initialization parameter, let's look at an example that shows why view merging could be dangerous from a security point of view (the full example is provided in the `optimizer_secure_view_merging.sql` script).

Say you have a very simple table with one primary key and two more columns:

```
CREATE TABLE t (
  id NUMBER(10) PRIMARY KEY,
  class NUMBER(10),
  pad VARCHAR2(10)
)
```

For security reasons, you want to provide access to this table through the following view. Notice the filter that is applied with the function to partially show the content of the table. How this function is implemented and what it does exactly isn't important:

```
CREATE OR REPLACE VIEW v AS
SELECT *
FROM t
WHERE f(class) = 1
```

Let's say, for example, that a user who has access to the view creates the following PL/SQL function. As you can see, it will just display the value of the input parameters through a call to the `dbms_output` package:

```
CREATE OR REPLACE FUNCTION spy (id IN NUMBER, pad IN VARCHAR2) RETURN NUMBER AS
BEGIN
  dbms_output.put_line('id='||id||' pad='||pad);
  RETURN 1;
END;
```

With the `optimizer_secure_view_merging` initialization parameter set to `FALSE`, you can run two test queries. Both return only the values that the user is allowed to see. In the second one, however, thanks to view merging, the function added to the query is executed before the function implementing the security check. As a result, you're able to see data that you shouldn't be able to access:

```
SQL> SELECT id, pad
  2  FROM v
  3  WHERE id BETWEEN 1 AND 5;

        ID PAD
---------- ----------
         1 DrMLTDXxxq
         4 AszBGEUGEL

SQL> SELECT id, pad
  2  FROM v
  3  WHERE id BETWEEN 1 AND 5
  4  AND spy(id, pad) = 1;
```

```
        ID PAD
---------- ----------
         1 DrMLTDXxxq
         4 AszBGEUGEL
id=1 pad=DrMLTDXxxq
id=2 pad=XOZnqYRJwI
id=3 pad=nlGfGBTxNk
id=4 pad=AszBGEUGEL
id=5 pad=qTSRnFjRGb
```

With the optimizer_secure_view_merging initialization parameter set to TRUE, the second query returns the following output. As you can see, the function and the query display the same data:

```
SQL> SELECT id, pad
  2  FROM v
  3  WHERE id BETWEEN 1 AND 5
  4  AND spy(id, pad) = 1;

        ID PAD
---------- ----------
         1 DrMLTDXxxq
         4 AszBGEUGEL
id=1 pad=DrMLTDXxxq
id=4 pad=AszBGEUGEL
```

Note that if the view is owned by the same user that issues the query, the optimizer_secure_view_merging initialization parameter is ignored (because there's no point in preventing a user from seeing information that he can already see by reading the tables referenced in the view directly).

A similar example, but showing the impact on predicate move around applied to Virtual Private Database (VPD) predicates, is available in the optimizer_secure_view_merging_vpd.sql script.

In summary, with the optimizer_secure_view_merging initialization parameter set to TRUE, the query optimizer checks whether query transformations could lead to security issues. If this is the case, those transformations won't be performed, and performance could be suboptimal as a result. For this reason, if you're neither using views nor VPD for security purposes, I advise you to set the optimizer_secure_view_merging initialization parameter to FALSE.

The optimizer_secure_view_merging initialization parameter is dynamic and can be changed at the instance level. In a 12.1 multitenant environment, it can also be set at the PDB level. It can't be changed at the session level. Instead, users having either the MERGE VIEW object privilege or the MERGE ANY VIEW system privilege aren't subject to the restrictions imposed by this initialization parameter. Be aware that by default the dba role provides the MERGE ANY VIEW system privilege.

## PGA Management

SQL operations that store data in memory (for example, sort operations and hash joins) use *work areas* in order to be executed. These work areas are allocated in the private memory of each server process (PGA). This section describes the initialization parameters devoted to the configuration of these work areas.

Usually, larger work areas provide better performance. Therefore, you should devote the unused memory that's available on the system to the allocation of work areas. Be careful, though, when changing it. The size of the work areas has an influence on the estimations of the query optimizer as well. You should expect changes not only in performance but also in execution plans. In other words, any modification should be carefully tested if you want to avoid surprises.

Generally speaking, this section doesn't provide "good" values for the initialization parameter it describes. The only way to find good values for a specific application is to test and measure how much PGA is required to achieve good performance. In fact, the amount of memory has an impact only on performance and not on how an operation works.

## workarea_size_policy

The `workarea_size_policy` initialization parameter specifies how the sizing of the work areas is performed. It can be set to one of the following two values:

- `auto`: The sizing of the single work areas is delegated to the *memory manager*. Through the `pga_aggregate_target` initialization parameter, only the amount of PGA for the whole system is specified. This is the default value.
- `manual`: Through the `hash_area_size`, `sort_area_size`, `sort_area_retained_size`, and `bitmap_merge_area_size` initialization parameters, you have full control over the size of the work areas.

In most situations, the memory manager does a good job, so it's highly recommended to delegate the PGA management to it. Only in rare cases does manual fine-tuning provide better results than automatic PGA management.

The `workarea_size_policy` initialization parameter is dynamic and can be changed at the instance and session levels. It's therefore possible to enable automatic PGA management at the system level and then, for special requirements, to switch to manual PGA management at the session level. In a 12.1 multitenant environment, it can also be set at the PDB level.

## pga_aggregate_target

If automatic PGA management is enabled, the `pga_aggregate_target` initialization parameter specifies (in bytes) the total amount of PGA dedicated to one database instance. Values from 10MB up to 4TB are supported. The default value is 20 percent of the size of the *system global area* (SGA). It's difficult to give any specific advice on what value should be used. On all systems, however, at least a few megabytes per concurrent session are needed.

---

■ **Note** As of version 11.1, the `memory_target` and `memory_max_target` initialization parameters can be used to specify the total amount of memory (that is, the SGA plus the aggregate PGA) used by a database instance. When they're set, the database engine automatically redistributes memory as needed between the SGA and the PGA. In such a configuration, the `pga_aggregate_target` initialization parameter is used to set the minimum size of the PGA only.

---

To illustrate how the memory manager works, in version 11.2.0.3 I executed a query requiring about 60MB of PGA with an increasing number of concurrent sessions (from 1 to 50). For each iteration, I checked the maximum amount of PGA allocated by the database instance and looked at the average amount of PGA allocated by the sessions executing the query. The `pga_aggregate_target` initialization parameter was set to 1GB. This means that, if the target is honored, at most 17 sessions (1GB/60MB) should be able to get the PGA necessary to execute the whole statement in memory. Figure 9-5 shows the results of the test. As you can see, the maximum PGA allocated by the database instance increased, as configured, up to 1GB. Notice that the system PGA increased almost proportionally with the number of sessions only up to 19 concurrent sessions. With more than 17 sessions, the system started reducing the amount of the PGA provided to each session.

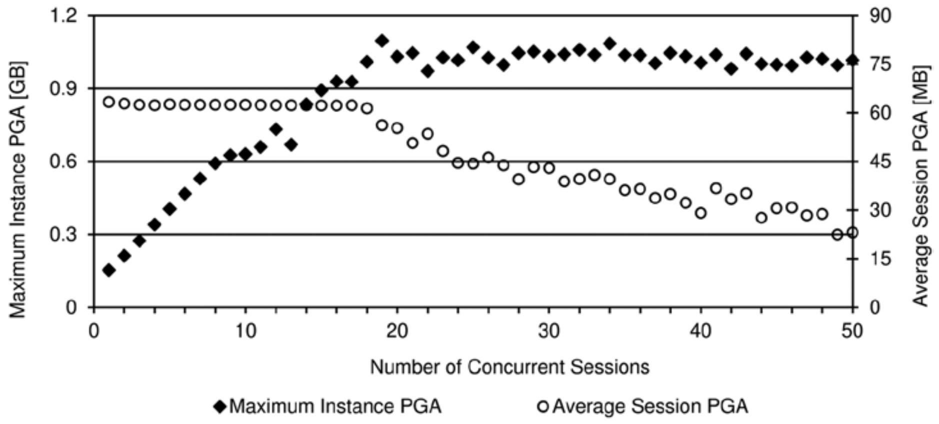


***Figure 9-5.*** *The memory manager automatically adjusts the amount of the PGA provided to sessions*

It's important to understand that the value of the `pga_aggregate_target` initialization parameter isn't a hard limit, but is rather a target. As a result, the database engine is free to allocate more memory than the value specified in the event the value specified is too low. This is allowed because otherwise operations requiring memory that can't be allocated would fail. You can set a hard limit though, using the `pga_aggregate_limit` initialization parameter (covered in the next section). That parameter becomes available in version 12.1. It's not available in prior releases.

To show a case of a database instance overallocating the PGA, I executed the previous test by setting the `pga_aggregate_target` initialization parameter to 128MB. In other words, I specified a value that's way too low to run 50 concurrent sessions requiring about 60MB each. Figure 9-6 shows the results of the test. As you can see, even a single session wasn't able to get sufficient PGA to execute the query in memory. In fact, that session got only half the required memory. As the number of concurrent sessions increased, more and more PGA was allocated. At 50 sessions, about 400MB of PGA was in use—more than three times the configured target.

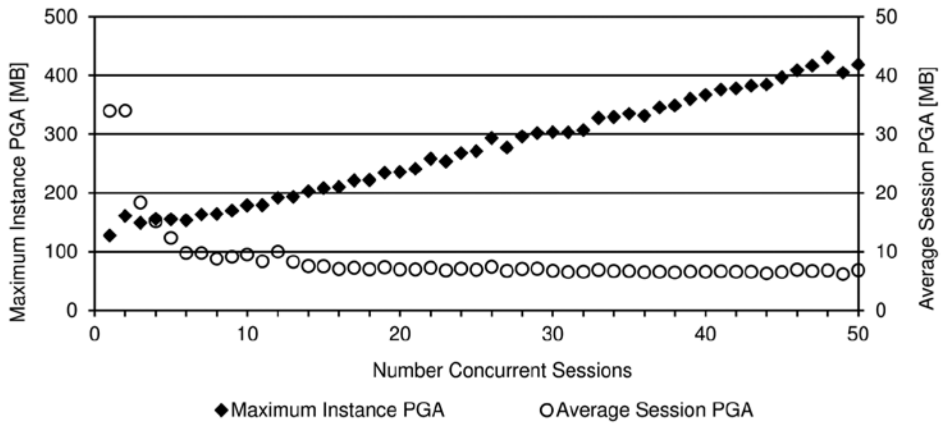


***Figure 9-6.*** *If the target set through the `pga_aggregate_target` initialization parameter (128MB in this case) is too low, the memory manager doesn't honor it*

To know whether a system experienced an overallocation of PGA, you can use the following query against the v$pgastat view. (Note that the query's output shows the final status of the database instance after running the test shown in Figure 9-6). If, as shown, the value of maximum PGA allocated is much higher than the value of aggregate PGA target parameter, then the value of the pga_aggregate_target initialization parameter isn't suitable. In such a case, it's important to know at which frequency the overallocation happens. For that purpose, the over allocation count statistic indicates the number of times since the last database instance start-up that the system had to allocate more PGA than specified through the pga_aggregate_target initialization parameter. Ideally, that value should be 0:

```
SQL> SELECT name, value, unit
  2  FROM v$pgastat
  3  WHERE name IN ('aggregate PGA target parameter',
  4                 'maximum PGA allocated',
  5                 'over allocation count');

NAME                                VALUE UNIT
------------------------------ ---------- -----
aggregate PGA target parameter 134217728 bytes
maximum PGA allocated          418658304 bytes
over allocation count                 94
```

You can also get information through the v$pgastat view about the amount of currently allocated PGA and how much of it's used for auto or manual work areas. The following query illustrates that. Notice that although the statistics with the total prefix provide the current utilization, the statistics with the maximum prefix provide the maximum utilization since the last database instance start-up:

```
SQL> SELECT name, value, unit
  2  FROM v$pgastat
  3  WHERE name LIKE '% PGA allocated' OR name LIKE '% workareas';

NAME                                       VALUE UNIT
------------------------------------- ---------- -----
total PGA allocated                    999358464 bytes
maximum PGA allocated                 1015480320 bytes
total PGA used for auto workareas      372764672 bytes
maximum PGA used for auto workareas    614833152 bytes
total PGA used for manual workareas            0 bytes
maximum PGA used for manual workareas          0 bytes
```

Also notice in this output that only part of the allocated PGA is used for work areas. Obviouly, something else is stored in the PGA. The point is that every process that requires some memory to execute a SQL statement or a PL/SQL program is able to allocate part of the PGA configured through the pga_aggregate_target initialization parameter. And that can be done even though the memory isn't used for a work area. Because the memory manager has no control over the size of these additional memory structures (also known as *untunable memory*), part of the PGA is also not under the memory manager's control. As a result, the amount of PGA available for work areas changes over time, depending on the system load. At any given moment you can see the amount of available memory through the aggregate PGA auto target statistics. The following example, which is an excerpt of the output generated by the pga_auto_target.sql script, shows how a collection defined through a PL/SQL call can allocate 500MB of PGA and, as a result, reduce the amount of memory available for work areas:

```
SQL> SELECT name, value, unit
  2  FROM v$pgastat
  3  WHERE name LIKE 'aggregate PGA %';
```

```
NAME                                  VALUE UNIT
------------------------------ ----------- -----
aggregate PGA target parameter  1073741824 bytes
aggregate PGA auto target        910411776 bytes

SQL> execute pga_pkg.allocate(500000)

SQL> SELECT name, value, unit
  2  FROM v$pgastat
  3  WHERE name LIKE 'aggregate PGA %';

NAME                                  VALUE UNIT
------------------------------ ----------- -----
aggregate PGA target parameter  1073741824 bytes
aggregate PGA auto target        375754752 bytes

SQL> execute dbms_session.reset_package;

SQL> SELECT name, value, unit
  2  FROM v$pgastat
  3  WHERE name LIKE 'aggregate PGA %';

NAME                                  VALUE UNIT
------------------------------ ----------- -----
aggregate PGA target parameter  1073741824 bytes
aggregate PGA auto target        910411776 bytes
```

The pga_aggregate_target initialization parameter is dynamic and can be changed only at the instance level. In a 12.1 multitenant environment, it can't be set at the PDB level.

## pga_aggregate_limit

The pga_aggregate_limit initialization parameter is new in version 12.1. It sets a hard limit to the amount of PGA a database instance can use. The parameter is useful because, as described in the previous section, the value set through the pga_aggregate_target initialization parameter is just a target, not a hard limit. In version 12.1, you can now also specify a hard limit should the need arise.

The default value of the pga_aggregate_limit initialization parameter is set to the greater of the following values:

- 2GB
- Twice the value of the pga_aggregate_target initialization parameter
- 3MB times the value of the processes initialization parameter

As a result, a limit is imposed by default. To avoid the limit, the parameter has to be set to 0. Setting the parameter to a value lower than the default, except for 0, isn't possible. The following error is raised when you attempt to set a limit lower than the default:

```
SQL> ALTER SYSTEM SET pga_aggregate_limit = 1G;
ALTER SYSTEM SET pga_aggregate_limit = 1G
*
ERROR at line 1:
ORA-02097: parameter cannot be modified because specified value is invalid
ORA-00093: pga_aggregate_limit must be between 2048M and 100000G
```

When the limit is reached, the database engine terminates calls or even kills sessions. To choose the session to deal with, the database engine doesn't consider the maximum PGA utilization. Instead, the database engine considers the session using the highest amount of untunable memory. When a call is terminated, the following error is raised:

```
ORA-04036: PGA memory used by the instance exceeds PGA_AGGREGATE_LIMIT
```

When a session is killed, a typical ORA-03113 is raised:

```
ORA-03113: end-of-file on communication channel
Process ID: 5125
Session ID: 17 Serial number: 39
```

In addition, a corresponding message like the following is written to the alert.log:

```
PGA memory used by the instance exceeds PGA_AGGREGATE_LIMIT of 2048 MB
Immediate Kill Session#: 17, Serial#: 39
Immediate Kill Session: sess: 0x77eb7478  OS pid: 5125
```

The pga_aggregate_limit initialization parameter is dynamic and can be changed only at the instance level. In a 12.1 multitenant environment, it can't be set at the PDB level.

## sort_area_size

If manual PGA management is enabled, the sort_area_size initialization parameter specifies (in bytes) the size of the work areas used for merge joins, sorts, and aggregations (including hash group-bys). Be careful: this is the size of one work area, and a single session may allocate several work areas (Chapter 14 provides some examples of this behavior). As a result, the total amount of PGA used for the whole system depends on the number of allocated work areas and not on the number of sessions. The default value is 64KB. Even though it's practically impossible to give general advice regarding the suggested values, the default is very small, and usually at least 512KB/1MB should be used. Significantly, work areas aren't always fully allocated. In other words, the value specified by the sort_area_size initialization parameter is only a limit. Consequently, specifying a value larger than is really needed isn't necessarily a problem.

The sort_area_size initialization parameter is dynamic and can be changed at the instance and session levels. In a 12.1 multitenant environment, it can also be set at the PDB level.

## sort_area_retained_size

In the previous section, you saw that the sort_area_size initialization parameter specifies the maximum size of the work areas used for sort operations. Strictly speaking, though, the sort_area_size initialization parameter specifies only the amount of memory used while the sorting operation takes place. After the last row has been obtained and included in the sorted result stored in the work area, memory is still required only as a buffer to return the sorted result to the parent operation. The sort_area_retained_size initialization parameter specifies (in bytes) the amount of memory retained for that read buffer. This initialization parameter is used only when manual PGA management is enabled. Even though the default value is derived from the sort_area_size initialization parameter, the v$parameter view shows 0.

To set this initialization parameter, you must be aware that if it's set to a value lower than the sort_area_size initialization parameter, and the result set doesn't fit into the retained memory, data is spilled into a temporary segment when the sort operation is completed. This might occur even if the sort operation itself is completely executed in memory! Consequently, it's advisable to use the default value for better performance. Only when the system is really short on memory does it make sense to set this parameter.

The `sort_area_retained_size` initialization parameter is dynamic and can be changed at the instance and session levels. In a 12.1 multitenant environment, it can also be set at the PDB level.

### hash_area_size

If manual PGA management is enabled, the `hash_area_size` initialization parameter specifies (in bytes) the size of the work areas used for hash joins. Be aware that this is the size of one work area and that a single session may allocate several work areas. That means the total amount of the PGA used for the whole system depends on the number of allocated work areas and not on the number of sessions. The default value is twice the value of the `sort_area_size` initialization parameter. Again, suggesting specific values is difficult. In any case, for values up to 4MB, it should be set to at least four to five times the value of the `sort_area_size` initialization parameter. If not, the query optimizer may overestimate the cost of hash joins and, as a result, favor merge joins to them. Again, the work areas aren't always fully allocated. In other words, the value specified by the `hash_area_size` initialization parameter is only a limit. Specifying a value larger than is really required isn't necessarily a problem.

The `hash_area_size` initialization parameter is dynamic and can be changed at the instance and session levels. In a 12.1 multitenant environment, it can't be set at the PDB level.

### bitmap_merge_area_size

If manual PGA management is enabled, the `bitmap_merge_area_size` initialization parameter specifies (in bytes) the size of the work areas used for merging bitmaps related to bitmap indexes. The default value is 1MB. Once again, it's practically impossible to give general advice regarding suggested values. Clearly, larger values might improve performances if a lot of bitmap indexes (for example, because of star transformation—see Chapter 14) are used.

The `bitmap_merge_area_size` initialization parameter is static and can't be changed at the system or session level. A database instance bounce is therefore necessary to change it. In a 12.1 multitenant environment, it can't be set at the PDB level.

## On to Chapter 10

This chapter describes how to achieve a good configuration of the query optimizer by setting initialization parameters. For this purpose, it's essential to understand not only how initialization parameters work but also how object and system statistics influence the query optimizer.

Even with the best configuration in place, the query optimizer may fail to find an efficient execution plan. When the performance of a SQL statement is questioned, the first thing to do is to review the execution plan. Chapter 10 discusses how to obtain execution plans and, more important, how to interpret them. I present some rules on how to recognize inefficient execution plans as well.

CHAPTER 10

# Execution Plans

An *execution plan* describes the operations carried out by the engine to execute a SQL statement. Every time you have to analyze a performance problem related to a SQL statement, or simply question the decisions taken by the query optimizer, you must know the execution plan. Without it, you're like a blind man with his cane in the middle of the Sahara Desert, groping around trying to find his way. I can't stress enough that the first thing to do while analyzing or questioning the performance of a SQL statement is to get its execution plan.

Whenever you deal with an execution plan, you carry out three basic actions: you obtain it, you interpret it, and you judge its efficiency. The aim of this chapter is to describe in detail how you should perform these three actions.

## Obtaining Execution Plans

Basically, Oracle Database provides five methods to obtain the execution plan associated with a SQL statement:

- Execute the `EXPLAIN PLAN` statement and then query the table where the output was written.
- Query a dynamic performance view showing the execution plans cached in the library cache.
- Use Real-time Monitoring to get information about SQL statements being executed or that just completed.
- Query an Automatic Workload Repository (AWR) or Statspack table, showing the execution plans stored in the repository.
- Activate a tracing facility providing execution plans.

Even though there are other methods of obtaining execution plans (for example, as described in Chapter 11, with features related to SQL profiles and SQL plan baselines), those methods can't be used to directly obtain the execution plan associated with a given SQL statement. Hence, they're not covered in this chapter. Because all tools displaying execution plans take advantage of one of the five methods just listed, the following sections describe just the basics rather than focus on specific tools such as Oracle Enterprise Manager, PL/SQL Developer, or Toad. I don't discuss such tools here also because, more often than not, they don't provide all the information you need for a thorough analysis. Note that Real-time Monitoring is described in Chapter 4.

### The EXPLAIN PLAN Statement

The aim of the `EXPLAIN PLAN` statement is to take a SQL statement as input and provide its execution plan and some information related to it as output in the *plan table*. In other words, with it you can ask the query optimizer which execution plan would be used to execute a given SQL statement.

Figure 10-1 shows the syntax of the EXPLAIN PLAN statement. The following parameters are available:

- statement specifies for which SQL statement the execution plan should be provided. The following SQL statements are supported: SELECT, INSERT, UPDATE, MERGE, DELETE, CREATE TABLE, CREATE INDEX, and ALTER INDEX.
- id specifies a name to distinguish between several execution plans stored in the plan table. Any string of up to 30 characters is supported. This parameter is optional. The default value is NULL.
- table specifies the name of the plan table where the information about the execution plan is inserted. This parameter is optional. The default value is plan_table. Whenever necessary, it's possible to specify a schema name as well as a database link name with the usual syntax: schema.table@dblink.

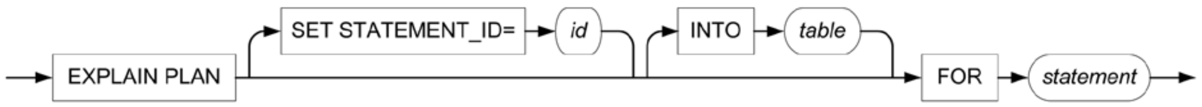


***Figure 10-1.*** *Syntax of the EXPLAIN PLAN statement*

It's important to recognize that the EXPLAIN PLAN statement is a DML statement, not a DDL statement. This means it doesn't perform an implicit commit of the current transaction. It simply inserts rows into the plan table.

To execute the EXPLAIN PLAN statement, the privileges to execute the SQL statement that is passed as a parameter are needed. Note that when working with views, appropriate privileges on all underlying tables and views are required as well. Because that's counterintuitive, look at the following example. Notice how the user is able to execute a query referencing the user_objects view but isn't able to execute the EXPLAIN PLAN statement for the very same query:

```
SQL> SELECT count(*) FROM user_objects;

  COUNT(*)
----------
        29

SQL> EXPLAIN PLAN FOR SELECT count(*) FROM user_objects;
EXPLAIN PLAN FOR SELECT count(*) FROM user_objects
                                      *
ERROR at line 1:
ORA-01039: insufficient privileges on underlying objects of the view
```

As pointed out by the error message, the user lacks the SELECT privilege on one or several data dictionary tables referenced by the user_objects view.

## The Plan Table

The plan table is where the EXPLAIN PLAN statement writes the output. If the plan table doesn't exist, an error is raised. A default plan table is owned by SYS, and a public synonym named plan_table exposes the table to all users. Whenever a private table is needed, it's good practice to manually create it with the utlxplan.sql script, available under the directory $ORACLE_HOME/rdbms/admin. If a plan table is manually created, you shouldn't forget to drop it and recreate it again in case of a database upgrade. In fact, it happens that new attributes are added with new releases.

It's interesting to note that the default plan table is a global temporary table that stores data up to the end of the session.[1] In this way, several concurrent users working with it don't interfere with each other.

To use a plan table with the EXPLAIN PLAN statement, you need at least INSERT and SELECT privileges. Even though you can perform basic operations without it, the DELETE privilege is usually granted as well.

I don't describe the plan table fully here for the simple reason that you usually don't need to query it directly. For a detailed description of its columns, refer to the *Performance Tuning Guide* (up to and including version 11.2), or the *SQL Tuning Guide* (beginning with version 12.1).

## Querying the Plan Table

It may be obvious that you can obtain the execution plan by running queries against the plan table directly. However, it's easier to use the display function in the dbms_xplan package, as shown in the following example. As you can see, its utilization is simple. In fact, it's enough to call the function in order to display the execution plan generated by the EXPLAIN PLAN statement. Notice how the return value of the function, which is a collection, is converted with the table function:

```
SQL> EXPLAIN PLAN FOR SELECT * FROM emp WHERE deptno = 10 ORDER BY ename;

SQL> SELECT * FROM table(dbms_xplan.display);

PLAN_TABLE_OUTPUT
---------------------------------------------------------------------------
Plan hash value: 150391907

---------------------------------------------------------------------------
| Id  | Operation          | Name | Rows  | Bytes | Cost (%CPU)| Time     |
---------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |      |     3 |   114 |     3  (34)| 00:00:01 |
|   1 |  SORT ORDER BY     |      |     3 |   114 |     3  (34)| 00:00:01 |
|*  2 |   TABLE ACCESS FULL| EMP  |     3 |   114 |     2   (0)| 00:00:01 |
---------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   2 - filter("DEPTNO"=10)
```

The display function isn't limited to being used without parameters. For this reason, later in this chapter I cover the dbms_xplan package, exploring all the possibilities, including a description of the generated output.

[1]In other words, it's a global temporary table created with the on commit preserve rows option.

## Bind Variables Trap

The most common mistake I come across in using the EXPLAIN PLAN statement is to specify a SQL statement that is different from the one to be analyzed. Naturally, that could lead to the wrong execution plan. Because the formatting itself has no impact on the execution plan, the difference is usually caused by replacing bind variables. Let's examine the execution plan used by the query in the following PL/SQL procedure:

```
CREATE OR REPLACE PROCEDURE p (p_value IN NUMBER) IS
BEGIN
  FOR i IN (SELECT * FROM emp WHERE empno = p_value)
  LOOP
    NULL; -- do something
  END LOOP;
END;
```

A commonly used technique is to copy/paste the query by replacing the PL/SQL variable with a literal. You execute a SQL statement like this:

```
EXPLAIN PLAN FOR SELECT * FROM emp WHERE empno = 7788
```

The problem is that by replacing the bind variable with a literal, you submit a different SQL statement to the query optimizer. This change—because, for example, of the presence of SQL profiles, stored outlines, SQL plan baselines (more about these topics in Chapter 11), or the method used by the query optimizer to estimate the selectivity of the predicate used in the WHERE clause (literals and bind variables aren't handled in the same way)—might have an impact on the decisions taken by the query optimizer.

The correct approach is to use the same SQL statement. This is possible because bind variables can be used with the EXPLAIN PLAN statement. You should, as an example, execute a SQL statement like the following one (notice that the p_value PL/SQL variable was replaced by the :B1 bind variable because this is what the PL/SQL engine would do):

```
EXPLAIN PLAN FOR SELECT * FROM emp WHERE empno = :B1
```

Nonetheless, using bind variables with the EXPLAIN PLAN statement has two problems. The first is that, by default, bind variables are declared as VARCHAR2. As a result, the database engine might automatically add implicit conversions, and that could change the execution plan. You can check this with the information about predicates shown at the end of the output generated by the display function in the dbms_xplan package. In the following output example, the to_number function is used for that purpose:

```
SQL> SELECT * FROM table(dbms_xplan.display);

PLAN_TABLE_OUTPUT
-------------------------------------------------------------------------------------
Plan hash value: 4024650034

-------------------------------------------------------------------------------------
| Id  | Operation                    | Name   | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |        |     1 |    38 |     1   (0)| 00:00:01 |
|   1 |  TABLE ACCESS BY INDEX ROWID| EMP    |     1 |    38 |     1   (0)| 00:00:01 |
|*  2 |   INDEX UNIQUE SCAN         | EMP_PK |     1 |       |     0   (0)| 00:00:01 |
-------------------------------------------------------------------------------------
```

```
Predicate Information (identified by operation id):
---------------------------------------------------

   2 - access("EMPNO"=TO_NUMBER(:B1))
```

It's generally good practice to check whether datatypes are correctly handled, for example, by using explicit conversion for all bind variables of the original SQL statement that aren't of VARCHAR2 type.

The second problem with using bind variables with the EXPLAIN PLAN statement is that no bind variable peeking is used. Because there's no solution for that problem, it isn't guaranteed that the execution plan generated by the EXPLAIN PLAN statement will be the one chosen at runtime. In other words, whenever bind variables are involved, the output generated by the EXPLAIN PLAN statement is unreliable.

## Dynamic Performance Views

Four dynamic performance views show information about the cursors present in the library cache:

- v$sql_plan provides basically the same information as the plan table. In other words, it provides execution plans and other related information provided by the query optimizer. The only notable differences between this view and the plan table are due to some columns identifying the cursor related to the execution plan in the library cache.
- v$sql_plan_statistics provides execution statistics, such as the elapsed time and the number of produced rows, for each operation in the v$sql_plan view. Essentially, it provides the runtime behavior of an execution plan. This is an important piece of information because the v$sql_plan view shows only the estimations and decisions taken by the query optimizer at parse time. Because the collection of execution statistics might cause a non-negligible overhead (depending on the execution plan and the operating system of the database server, the overhead can also be significant), by default they aren't collected. To activate the collection, either the statistics_level initialization parameter must be set to all or the gather_plan_statistics hint must be specified in the SQL statement. Be aware that, because of the possible overhead, I don't recommend changing the default value of the statistics_level initialization parameters at the system level.
- v$sql_workarea provides information about the memory work areas needed to execute a cursor. It gives runtime memory utilization as well as estimations about the amount of memory needed to efficiently execute operations.
- v$sql_plan_statistics_all shows in a single view all the information provided by the v$sql_plan, v$sql_plan_statistics, and v$sql_workarea views. By using it, you simply avoid manually joining several views.

The cursors in the library cache (and therefore in these dynamic performance views) are identified by two columns: address and child_number. With the address column, you identify the parent cursors. With the two columns, you identify the child cursors. It's more common to use the sql_id column instead of the address column to identify cursors. The advantage of using the sql_id column is that its value depends only on the SQL statement itself. In other words, it never changes for a given SQL statement (in fact, it's the result of a hash function applied to the text of the SQL statement). On the other hand, the address column is a pointer to the handle of the SQL statement in memory and can change over time.

To identify a cursor, basically you're confronted with two search methods. Either you know the session executing a SQL statement or you know the text of the SQL statement. In both cases, once the child cursor is identified, you can display information about it.

## Identifying Child Cursors

The first common situation you have to face is trying to get information about a SQL statement that is related to a session currently connected to the instance. In this case, you execute the search on the v$session view. The currently executed SQL statement is identified by the sql_id (or sql_address) and sql_child_number columns. The last-executed SQL statement is identified by the prev_sql_id (or prev_sql_addr) and prev_child_number columns. To illustrate the use of this method, let's say that a user, Curtis, calls you and complains that he's waiting on a request submitted with an application just a few minutes ago. For this problem, it's useful to query the v$session view directly, as shown in the following example. With that output, you know he's currently running a SQL statement (otherwise, the status wouldn't be ACTIVE) and which cursor is related to his session:

```
SQL> SELECT status, sql_id, sql_child_number
  2  FROM v$session
  3  WHERE username = 'CURTIS';

STATUS  SQL_ID        SQL_CHILD_NUMBER
------- ------------- ----------------
ACTIVE  1scu79x31qavt                1
```

The second common situation is when you do know the text of the SQL statement that you want to find more information about. In this case, you execute the search on the v$sql view. The text associated with a cursor is available in the sql_text and sql_fulltext columns. The difference between the two columns is that the first shows only the first part of the text through a VARCHAR2(1000), while the second shows the whole text through a CLOB. For example, if you know that the SQL statement you're looking for contains a literal with the text "online discount," you can use the following query to find out the identifiers of the cursor:

```
SQL> SELECT sql_id, child_number, sql_text
  2  FROM v$sql
  3  WHERE sql_fulltext LIKE '%online discount%'
  4  AND sql_text NOT LIKE '%v$sql%';

SQL_ID        CHILD_NUMBER SQL_TEXT
------------- ------------ ---------------------------------------------------
1hqjydsjbvmwq            0 SELECT SUM(AMOUNT_SOLD) FROM SALES S, PROMOTIONS P
                           WHERE S.PROMO_ID = P.PROMO_ID AND PROMO_SUBCATEGORY
                            = 'online discount'
```

## Querying Dynamic Performance Views

To obtain the execution plan, you can run queries directly against the v$sql_plan and v$sql_plan_statistics_all views. However, there's an easier and much better way to do it: you can use the display_cursor function in the dbms_xplan package. As shown in the following example, its use is similar to calling the display function previously discussed. The only difference is that two parameters identifying the child cursor to be displayed are passed to the function:

```
SQL> SELECT * FROM table(dbms_xplan.display_cursor('1hqjydsjbvmwq', 0));
```

```
PLAN_TABLE_OUTPUT
-------------------------------------------------------------------------------------
SQL_ID  1hqjydsjbvmwq, child number 0
-------------------------------------
SELECT SUM(AMOUNT_SOLD) FROM SALES S, PROMOTIONS P WHERE S.PROMO_ID =
P.PROMO_ID AND PROMO_SUBCATEGORY = 'online discount'

Plan hash value: 265338492

-------------------------------------------------------------------------------------
| Id  | Operation              | Name       | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT       |            |       |       |   139 (100)|          |
|   1 |  SORT AGGREGATE        |            |     1 |    30 |            |          |
|*  2 |   HASH JOIN            |            |  913K|    26M|   139  (33)| 00:00:01 |
|*  3 |    TABLE ACCESS FULL   | PROMOTIONS |    23 |   483 |     4   (0)| 00:00:01 |
|   4 |    PARTITION RANGE ALL|            |  918K|  8075K|   123  (27)| 00:00:01 |
|   5 |     TABLE ACCESS FULL | SALES      |  918K|  8075K|   123  (27)| 00:00:01 |
-------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   2 - access("S"."PROMO_ID"="P"."PROMO_ID")
   3 - filter("PROMO_SUBCATEGORY"='online discount')
```

The `display_cursor` function isn't limited to being used with two parameters identifying a child cursor. For this reason, later in this chapter I cover the `dbms_xplan` package, exploring all possibilities, including a description of the generated output.

## Automatic Workload Repository and Statspack

When a snapshot is taken, Automatic Workload Repository (AWR) and Statspack are able to collect execution plans. To get execution plans, queries against the dynamic performance views described in the previous section are executed. Once available, the execution plans may be displayed in reports, by Oracle Enterprise Manager or other tools. For both AWR and Statspack, the repository table storing the execution plans has a structure very similar to the one of the `v$sql_plan` view. Because of this, the techniques described in the previous section apply to it as well.

The execution plans stored in AWR are available through the `dba_hist_sql_plan` view (from version 12.1 onward, `cdb_hist_sql_plan` is also available). To query them, the `dbms_xplan` package provides the `display_awr` function. As for the other functions of this package, its use is straightforward. The following query is an example (note that the parameter passed to the `display_awr` function identifies the SQL statement through its `sql_id`):

```
SQL> SELECT * FROM table(dbms_xplan.display_awr('1hqjydsjbvmwq'));

PLAN_TABLE_OUTPUT
-------------------------------------------------------------------------------------
SQL_ID 1hqjydsjbvmwq
--------------------
SELECT SUM(AMOUNT_SOLD) FROM SALES S, PROMOTIONS P WHERE S.PROMO_ID =
P.PROMO_ID AND PROMO_SUBCATEGORY = 'online discount'
```

```
Plan hash value: 265338492

----------------------------------------------------------------------------------
| Id  | Operation             | Name       | Rows  | Bytes | Cost (%CPU)| Time     |
----------------------------------------------------------------------------------
|   0 | SELECT STATEMENT      |            |       |       |   139 (100)|          |
|   1 |  SORT AGGREGATE       |            |     1 |    30 |            |          |
|   2 |   HASH JOIN           |            |  913K|   26M|   139  (33)| 00:00:01 |
|   3 |    TABLE ACCESS FULL  | PROMOTIONS |    23 |   483 |     4   (0)| 00:00:01 |
|   4 |    PARTITION RANGE ALL|            |  918K|  8075K|   123  (27)| 00:00:01 |
|   5 |     TABLE ACCESS FULL | SALES      |  918K|  8075K|   123  (27)| 00:00:01 |
----------------------------------------------------------------------------------
```

The display_awr function isn't limited to being used with one parameter identifying a SQL statement. For this reason, later in this chapter I cover the dbms_xplan package, exploring all possibilities, including a description of the generated output.

Statspack stores execution plans in the stats$sql_plan repository table when a level equal to or greater than 6 is used for taking the snapshots. Even though no specific function is provided by the dbms_xplan package to query that repository table, it's possible to take advantage of the display function to show the execution plans it contains. An example can be found in the display_statspack.sql script.

In addition, for both AWR and Statspack, Oracle Database provides useful scripts for highlighting execution plan changes and resource consumption variation during a period of time for a specific SQL statement. Their names are awrsqrpt.sql and sprepsql.sql, respectively. You find them under the directory $ORACLE_HOME/rdbms/admin. The following is an excerpt of the output generated by the awrsqrpt.sql script. According to the output, the execution plan of the SQL statement changed during the analyzed period. The average elapsed time went from about 8.3 seconds (16,577/2/1,000) for the first one to about 3.7 seconds (14,736/4/1,000) for the second one:

```
SQL ID: 1hqjydsjbvmwq              DB/Inst: DBM11203/DBM11203  Snaps: 576-577
-> 1st Capture and Last Capture Snap IDs
   refer to Snapshot IDs witin the snapshot range
-> SELECT SUM(AMOUNT_SOLD) FROM SALES S, PROMOTIONS P WHERE S.PROMO_ID = ...

    Plan Hash           Total Elapsed                 1st Capture   Last Capture
#   Value                    Time(ms)    Executions       Snap ID        Snap ID
--- ---------------- ---------------- ------------- ------------- --------------
1   2446651477                 16,577             2           577            577
2   265338492                  14,736             4           577            577
--- --------------------------------------------------------------------------

Plan 1(PHV: 2446651477)
-----------------------

Stat Name                                Statement   Per Execution % Snap
---------------------------------------- ---------- -------------- -------
Elapsed Time (ms)                            16,577         8,288.6    50.2
CPU Time (ms)                                16,071         8,035.3    50.9
Executions                                        2             N/A     N/A
Buffer Gets                                 163,606        81,803.0    90.1
Disk Reads                                  161,900        80,950.0    96.0
Parse Calls                                       2             1.0     1.0
Rows                                              2             1.0     N/A
-----------------------------------------------------------------------
```

```
-----------------------------------------------------------------------------------
| Id  | Operation             | Name       | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------
|   0 | SELECT STATEMENT      |            |       |       |  2798 (100)|          |
|   1 |  SORT AGGREGATE       |            |     1 |    30 |            |          |
|   2 |   NESTED LOOPS        |            |  913K|    26M|  2798  (27)| 00:00:12 |
|   3 |    TABLE ACCESS FULL  | PROMOTIONS |    23 |   483 |     4   (0)| 00:00:01 |
|   4 |    PARTITION RANGE ALL|            | 39950 |   351K|   121  (27)| 00:00:01 |
|   5 |     TABLE ACCESS FULL | SALES      | 39950 |   351K|   121  (27)| 00:00:01 |
-----------------------------------------------------------------------------------

Plan 2(PHV: 265338492)
----------------------

Stat Name                                Statement   Per Execution % Snap
---------------------------------------- ---------- -------------- -------
Elapsed Time (ms)                            14,736        3,684.0    44.6
CPU Time (ms)                                14,565        3,641.2    46.1
Executions                                        4            N/A     N/A
Buffer Gets                                   6,755        1,688.8     3.7
Disk Reads                                    6,485        1,621.3     3.8
Parse Calls                                       1            0.3     0.5
Rows                                              4            1.0     N/A
--------------------------------------------------------------------------

-----------------------------------------------------------------------------------
| Id  | Operation             | Name       | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------
|   0 | SELECT STATEMENT      |            |       |       |   139 (100)|          |
|   1 |  SORT AGGREGATE       |            |     1 |    30 |            |          |
|   2 |   HASH JOIN           |            |  913K|    26M|   139  (33)| 00:00:01 |
|   3 |    TABLE ACCESS FULL  | PROMOTIONS |    23 |   483 |     4   (0)| 00:00:01 |
|   4 |    PARTITION RANGE ALL|            |  918K|  8075K|   123  (27)| 00:00:01 |
|   5 |     TABLE ACCESS FULL | SALES      |  918K|  8075K|   123  (27)| 00:00:01 |
-----------------------------------------------------------------------------------
```

## Tracing Facilities

Several tracing facilities provide information about execution plans. Unfortunately, except for SQL trace (see Chapter 3), none of them is either officially supported or documented. But they may turn out to be useful, so I cover two of them briefly.

### Event 10053

If you're in serious difficulty because of the decisions made by the query optimizer and you want to understand what is going on, a query optimizer trace may help you. Let me warn you, though, that reading the trace files it generates isn't an easy task. Luckily, you won't need to read those often—and then only if you're really interested in the internal workings of the query optimizer.

If you want to generate a trace file for one SQL statement at a time and you're able to manually execute it, it's common to embed it between the following two SQL statements, thus enabling and disabling the event 10053. Just be aware that the trace file is generated only when a hard parse is performed:

```
ALTER SESSION SET events '10053 trace name context forever'

ALTER SESSION SET events '10053 trace name context off'
```

If manually executing a SQL statement isn't an option, as of version 11.1 you can instruct the query optimizer to generate a trace file the next time a SQL statement identified by a specific `sql_id` is hard parsed. To enable and disable that behavior, you use SQL statements like the following (it goes without saying that you have to change the `sql_id` passed as a parameter). The advantage of this method is that you can let the application issue the SQL statement. That in turn gives you a trace file for the real SQL statement as executed in the live execution environment. Note that this method works at the session level as well. Simply replace the `ALTER SYSTEM` statement with an `ALTER SESSION` statement:

```
ALTER SYSTEM SET events 'trace[rdbms.SQL_Optimizer.*][sql:9s5u1k3vshsw4]'

ALTER SYSTEM SET events 'trace[rdbms.SQL_Optimizer.*][sql:9s5u1k3vshsw4] off'
```

If you want to analyze the SQL statement associated with a cursor stored in the library cache, from version 11.2 onward you can take advantage of the `dump_trace` procedure in the `dbms_sqldiag` package. This method requires neither the execution of the SQL statement nor knowledge of the actual environment in which the SQL statement was parsed. You also don't need to know the values of the bind variables that are associated to the cursor. The procedure takes everything it needs from the library cache and instructs the query optimizer to reoptimize the SQL statement and dump a trace file. The following illustrates how to call it:

```
dbms_sqldiag.dump_trace(
  p_sql_id       => '30g1nn8wdymh3',
  p_child_number => 0,
  p_component    => 'Optimizer',
  p_file_id      => 'test'
);
```

The procedure has the following input parameters:

- `p_sql_id` specifies the parent cursor to be processed.
- `p_child_number` specifies the child number that, along with `p_sql_id`, identifies the child cursor to be processed. This parameter is optional and defaults to 0.
- `p_component` specifies whether the procedure dumps the `Optimizer` or `Compiler` trace. Simply put, while the former is analog to setting event 10053, the latter writes even more information to the trace file.
- `p_file_id` specifies a value for the `tracefile_identifier` initialization parameter. This parameter is optional and defaults to `NULL`.

Independently of how you enabled the tracing, the query optimizer generates a trace file containing plenty of information about the work it carries out. In it you'll find the execution environment determined by initialization parameters, system statistics, and object statistics, as well as the estimations performed for the purpose of finding out the most efficient execution plan. Describing the content of the trace file generated by this event is beyond the scope of this book. If necessary, please refer to the following sources:

- Wolfgang Breitling's paper *A Look under the Hood of CBO: The 10053 Event*
- Oracle Support note *CASE STUDY: Analyzing 10053 Trace Files* (338137.1)
- Chapter 14 of Jonathan Lewis's book *Cost-Based Oracle Fundamentals*, (Apress, 2006)

Each server process writes all data about the SQL statements it parses in its own trace file. This means not only that a trace file can contain information about several SQL statements, but also that several trace files will be used whenever the generation of the trace file is enabled in several sessions. For information about the name and location of trace files, refer to the "Finding Trace Files" section in Chapter 3.

## Event 10132

You can use event 10132 to cause a trace file to be generated, containing the execution plan related to every hard parse. This may be useful if you want to keep a history of all execution plans for a specific module or application. The following example shows the kind of information stored in the trace file for every SQL statement, principally the SQL statement and its execution plan (which includes information about predicates). Notice that from this output I cut out, in two different places, long lists of parameters and bug fixes that provide information about the execution environment:

```
----- Current SQL Statement for this session (sql_id=gbxvdrz7jvt80) -----
SELECT count(n) FROM t WHERE n BETWEEN 6 AND 19
----- Explain Plan Dump -----

-------------------------------------+-----------------------------------+
| Id  | Operation            | Name    | Rows  | Bytes | Cost  | Time      |
-------------------------------------+-----------------------------------+
| 0   | SELECT STATEMENT     |         |       |       |     2 |           |
| 1   |  SORT AGGREGATE      |         |     1 |    13 |       |           |
| 2   |   TABLE ACCESS FULL  | T       |    14 |   182 |     2 |  00:00:01 |
-------------------------------------+-----------------------------------+
Predicate Information:
----------------------
2 - filter(("N">=6 AND "N"<=19))

Content of other_xml column
===========================
  db_version     : 11.2.0.3
  parse_schema   : CHRIS
  dynamic_sampling: 2
  plan_hash      : 2966233522
  plan_hash_2    : 1071362934
  Outline Data:
  /*+
    BEGIN_OUTLINE_DATA
      IGNORE_OPTIM_EMBEDDED_HINTS
      OPTIMIZER_FEATURES_ENABLE('11.2.0.3')
      DB_VERSION('11.2.0.3')
      ALL_ROWS
      OUTLINE_LEAF(@"SEL$1")
      FULL(@"SEL$1" "T"@"SEL$1")
    END_OUTLINE_DATA
  */

Optimizer state dump:
Compilation Environment Dump
optimizer_mode_hinted               = false
optimizer_features_hinted           = 0.0.0
...
_px_numa_support_enabled            = true
total_processor_group_count         = 1
Bug Fix Control Environment
    fix  3834770 = 1
    fix  3746511 = enabled
...
End of Optimizer State Dump
```

The lists of initialization parameters and bug fixes are especially long. For this reason, depending on the release you're using, about 10–30KB of data is written to the trace file even for the simplest SQL statement. The generation of such a trace file might be a significant overhead. You should therefore only activate the event 10132 if you really need it.

The event 10132 can be enabled and disabled in the following ways:

- Enable and disable the event for the current session.

  ```
  ALTER SESSION SET events '10132 trace name context forever'

  ALTER SESSION SET events '10132 trace name context off'
  ```

- Enable and disable the event for the whole database. Warning: this setting doesn't take effect immediately, but only for sessions created after the modification.

  ```
  ALTER SYSTEM SET events '10132 trace name context forever'

  ALTER SYSTEM SET events '10132 trace name context off'
  ```

Each server process writes all data about the SQL statements it parses in its own trace file. This means not only that a trace file can contain information about several SQL statements but also that several trace files will be used whenever the event is enabled in several sessions. For information about the name and location of trace files, refer to the "Finding Trace Files" section in Chapter 3.

## The dbms_xplan Package

You saw earlier in this chapter that the dbms_xplan package can be used to display execution plans stored in several places: among others, in the plan table, in the library cache, in AWR, and in the Statspack repository. The following sections describe the functions available in the package for that purpose. To begin with, let's take a look at the output they generate.

### Output

The aim of this section is to explain the information contained in the output that's returned by some of the functions in the dbms_xplan package. To do so, I use a sample output, generated by the dbms_xplan_output.sql script, that contains most of the available sections. Because one book page isn't wide enough to show all the information, not everything for each section is shown. I show key information only. If something is missing, I point it out. Also note that for most of the information provided in this section, examples and further explanations are given either later on in this chapter or in Part 4. The first section of output is as follows:

```
SQL_ID  dwnnunj9nuztb, child number 0
-------------------------------------
SELECT t2.* FROM t1, t2 WHERE t1.n = t2.n AND t1.id > :t1_id AND
t2.id BETWEEN :t2_id_min AND :t2_id_max
```

This section gives the following information about the SQL statement:

- The sql_id identifies the parent cursor. This information is available only when the output is generated by the display_cursor and display_awr functions.
- The child number, along with the sql_id, identifies the child cursor. This information is available only when the output is generated by the display_cursor function.
- The text of the SQL statement is available only when the output is generated by the display_cursor and display_awr functions.

The second section shows the hash value of the execution plan and, in a table, the execution plan itself. Here's the excerpt:

```
Plan hash value: 2539808735

---------------------------------------------------------------------------------------
| Id  | Operation                     | Name  | Rows  | Bytes | Cost (%CPU)| Time     |
---------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT              |       |       |       |    15 (100)|          |
|*  1 |  FILTER                       |       |       |       |            |          |
|*  2 |   HASH JOIN                   |       |    14 |  7756 |    15   (7)| 00:00:01 |
|   3 |    TABLE ACCESS BY INDEX ROWID| T2    |    14 |  7392 |     4   (0)| 00:00:01 |
|*  4 |     INDEX RANGE SCAN          | T2_PK |    14 |       |     2   (0)| 00:00:01 |
|*  5 |    TABLE ACCESS FULL          | T1    |   876 | 22776 |    23   (0)| 00:00:01 |
---------------------------------------------------------------------------------------
```

In the table, estimations and execution statistics for each operation are provided. The number of columns in the table depends directly on the amount of available information. For example, information about partitioning, parallel processing, or execution statistics is shown only when available. For this reason, two outputs generated by the same function with exactly the same parameters may be different. In this case, you see the columns available by default. Table 10-1 summarizes all the columns you might see.

***Table 10-1.*** *Columns of the Table Containing the Execution Plan*

| Column | Description |
|---|---|
| **Basics (Always Available)** | |
| Id | The identifier of each operation (line) in the execution plan. If the number is prefixed by an asterisk, it means that predicate information for that line is available later. |
| Operation | The operation to be executed. This is also known as the *row source operation*. |
| Name | The object on which the operation is executed. |
| **Query Optimizer Estimations** | |
| Rows and E-Rows | The estimated number of rows returned by the operation. |
| Bytes and E-Bytes | The estimated amount of data returned by the operation. |
| TempSpc and E-Temp | The estimated amount of temporary space (in bytes) required by the operation. |
| Cost (%CPU) | The estimated cost of the operation. The percentage of CPU cost is given in parentheses. Note that this value is cumulated through the execution plan. In other words, the cost of parent operations contains the cost of their child operations. |
| Time and E-Time | The estimated amount of time needed to execute the operation (HH:MM:SS). |
| **Partitioning** | |
| Pstart | The number of the first partition to be accessed. If that number isn't known at parse time, either KEY or INVALID. KEY is used when the first partition is determined during the execution phase. |
| Pstop | The number of the last partition to be accessed. If that number isn't known at parse time, either KEY or INVALID. KEY is used when the last partition is determined during the execution phase. |
| **Parallel and Distributed Processing** | |
| Inst | For distributed processing, the name of the database link used by the operation. |
| TQ | For parallel processing, the table queue used for the communication between slave processes. |
| IN-OUT | The relationship between parallel or distributed operations. |
| PQ Distrib | For parallel processing, the distribution used by producers to send data to consumers. |

(*continued*)

***Table 10-1.*** *(continued)*

| Column | Description |
|---|---|
| **Runtime Statistics*** | |
| `Starts` | The number of times a specific operation was executed. In some special cases, this statistic shows the number of times a specific memory structure was accessed (as shown later on in the “Unrelated-Combine Operations” section). |
| `A-Rows` | The actual number of rows returned by the operation. |
| `A-Time` | The actual amount of time spent executing the operation (HH:MM:SS.FF). |
| **I/O Statistics*** | |
| `Buffers` | The number of logical reads performed during the execution. |
| `Reads` | The number of physical reads performed during the execution. |
| `Writes` | The number of physical writes performed during the execution. |
| **Memory Utilization Statistics** | |
| `OMem` | The estimated amount of memory (in bytes) needed for an optimal execution. |
| `1Mem` | The estimated amount of memory (in bytes) needed for a one-pass execution. |
| `O/1/M` | The number of times the execution was performed in optimal/one-pass/multipass mode. |
| `Used-Mem` | The amount of memory (in bytes) used by the operation during the last execution. |
| `Used-Tmp` | The amount of temporary space (in kilobytes) used by the operation during the last execution. This value must be multiplied by 1,024 to be consistent with the other memory utilization columns (for example, 32K means 32MB). |
| `Max-Tmp` | The maximum amount of temporary space (in kilobytes) used by the operation. This value has to be multiplied by 1,024 to be consistent with the other memory utilization columns (for example, 32K means 32MB). |

**Available only when execution statistics are enabled.*

The next section shows the query block names and the object aliases:

```
Query Block Name / Object Alias (identified by operation id):
-------------------------------------------------------------

   1 - SEL$1
   3 - SEL$1 / T2@SEL$1
   4 - SEL$1 / T2@SEL$1
   5 - SEL$1 / T1@SEL$1
```

For each operation in the execution plan, you see which query block it's related to and, optionally, which object it's executed on. This information is essential when the SQL statement references the same table several times. Query block names are discussed in more detail along with hints in Chapter 11.

The fourth section shows the set of hints, called *outline*, that should be sufficient to reproduce that particular execution plan. Be aware that the outline doesn't always contain all the necessary hints. Chapter 11 explains why some outlines aren't sufficient to reproduce an execution plan and also describes how it's possible to store and take advantage of such an outline with, for example, stored outlines and SQL plan baselines:

```
Outline Data
-------------

  /*+
      BEGIN_OUTLINE_DATA
      IGNORE_OPTIM_EMBEDDED_HINTS
      OPTIMIZER_FEATURES_ENABLE('11.2.0.4')
      DB_VERSION('11.2.0.4')
      ALL_ROWS
      OUTLINE_LEAF(@"SEL$1")
      INDEX_RS_ASC(@"SEL$1" "T2"@"SEL$1" ("T2"."ID"))
      FULL(@"SEL$1" "T1"@"SEL$1")
      LEADING(@"SEL$1" "T2"@"SEL$1" "T1"@"SEL$1")
      USE_HASH(@"SEL$1" "T1"@"SEL$1")
      END_OUTLINE_DATA
  */
```

The following section is only displayed when the query optimizer takes advantage of bind variable peeking. Each bind variable's data type and value are provided:

```
Peeked Binds (identified by position):
--------------------------------------

   1 - :T1_ID (NUMBER): 6
   2 - :T2_ID_MIN (NUMBER): 6
   3 - :T2_ID_MAX (NUMBER): 19
```

The next section shows which predicates are applied. For each, it's shown where (line) and how (`access`, `filter` or `storage`) they're applied:

```
Predicate Information (identified by operation id):
---------------------------------------------------

   1 - filter(:T2_ID_MIN<=:T2_ID_MAX)
   2 - access("T1"."N"="T2"."N")
   4 - access("T2"."ID">=:T2_ID_MIN AND "T2"."ID"<=:T2_ID_MAX)
   5 - filter("T1"."ID">:T1_ID)
```

Although an *access predicate* is used to locate rows by taking advantage of an efficient access structure (for example, a hash table in memory, like for operation 2, or an index, like for operation 4), a *filter predicate* is applied only after the rows have already been extracted from the structure storing them. In addition, when an Exadata storage server is used, a *storage predicate* points out that a specific filter is offloaded to the underlying storage subsystem.

Note that this section contains both the predicate present in the SQL statement itself and the predicates that may be generated by the query optimizer or by Virtual Private Database policies. In the preceding example, you have the following predicates:

- The operation in line 1 checks whether the value of the bind variables lead to an empty result set or not. The query can return rows only if the `:T2_ID_MIN<=:T2_ID_MAX` predicate is fullfiled. If the predicate isn't fulfilled, the remainder of the query operations aren't executed.
- The hash join at line 2 uses the `"T1"."N"="T2"."N"` predicate to join the two tables. In other words, the access predicate might show a join condition as well. In this specific case, the access predicate is used to specify that the hash table in memory containing the data of the `t1` table, whose hash key is `t1.n`, is probed with the values of column `t2.n` returned by accessing the `t2` table (how hash joins work is described in detail in Chapter 14).
- The index scan at line 4 accesses the `t_pk` index to look up the `id` column of the `t1` table. In this case, the access predicate shows on which key the lookup is performed.
- At line 5, all rows in the `t1` table are read through a full scan. Then, when the rows have been extracted from the blocks, the `"T1"."ID">:T1_ID` predicate is applied to filter them.

The following section shows which columns are returned as output when each operation is executed. Here's the excerpt:

```
Column Projection Information (identified by operation id):
-----------------------------------------------------------

   1 - "T2"."N"[NUMBER,22], "T2"."ID"[NUMBER,22], "T2"."PAD"[VARCHAR2,1000]
   2 - (#keys=1) "T2"."N"[NUMBER,22], "T2"."ID"[NUMBER,22],
       "T2"."PAD"[VARCHAR2,1000]
   3 - "T2"."ID"[NUMBER,22], "T2"."N"[NUMBER,22], "T2"."PAD"[VARCHAR2,1000]
   4 - "T2".ROWID[ROWID,10], "T2"."ID"[NUMBER,22]
   5 - "T1"."N"[NUMBER,22]
```

In this case, it's significant to note that while the table access at line 3 returns the `id`, `n`, and `pad` columns, the table access at line 5 returns only the `n` column. For this reason, the estimated amount of data (`Bytes`) returned from line 3 for each row (7,392/14 = 528 bytes) is much greater than for line 5 (22,776/876 = 26 bytes). Chapter 16 talks more about the ability of the database engine to partially read a row and why, from a performance point of view, doing so is sensible.

Finally, there's a section providing notes and warnings about the optimization phase, the environment, or the SQL statement itself:

```
Note
-----
   - dynamic sampling used for this statement (level=2)
```

Here you're informed that the query optimizer used dynamic sampling to gather object statistics.

## The display Function

The `display` function returns execution plans stored in a plan table. The return value is an instance of the `dbms_xplan_type_table` collection. The elements of the collection are instances of the `dbms_xplan_type` object type. The only attribute of this object type, named `plan_table_output`, is of type `VARCHAR2(300)`. The function has the following input parameters:

- `table_name` specifies the name of the plan table. The default value is `plan_table`. If `NULL` is specified, the default value is used.
- `statement_id` specifies the SQL statement name, optionally given as a parameter, when the `EXPLAIN PLAN` statement is executed. The default value is `NULL`. If the default value is used, the execution plan most recently inserted into the plan table is displayed (provided the `filter_preds` parameter isn't specified).
- `format` specifies which information is provided in the output. There are primitive values (`basic`, `typical`, `serial`, `all`, and `advanced`) and, for finer control, additional modifiers (`adaptive`, `alias`, `bytes`, `cost`, `note`, `outline`, `parallel`, `partition`, `peeked_binds`, `predicate`, `projection`, `remote`, `report`, and `rows`) that can be added to them. If information should be added, modifiers are optionally prefixed by the + character (for example, `basic +predicate`). If information should be removed, modifiers have to be prefixed by the - character (for example, `typical -bytes`). Multiple modifiers can be specified at the same time (for example, `typical +alias -bytes -cost`). Table 10-2 and Table 10-3 fully describe the primitive values and the modifiers, respectively. The default value is `typical`.

***Table 10-2.*** *Primitive Values Accepted by the `format` Parameter*

| Value | Description |
|---|---|
| `basic` | Displays only the minimum amount of information, basically only the operations and the objects on which they're executed. |
| `typical` | Displays the most common information, basically everything except for aliases, outline, peeked bind variables, subplans, column projection, and reporting mode information. |
| `serial` | Like `typical`, except that information about parallel processing isn't displayed. |
| `all` | Displays all available information except the outline, the peeked bind variables, subplans, and reporting mode information. |
| `advanced` | Displays all available information except for subplans and reporting mode information. |

***Table 10-3.*** *Modifiers Accepted by the `format` Parameter*

| Value | Description |
|---|---|
| `adaptive` | Controls the display of subplans. This section isn't shown in the previous examples. Refer to the "Adaptive Execution Plans" section later on for an example. This modifier is available from version 12.1 onward only. |
| `alias` | Controls the display of the section containing query block names and object aliases. |
| `bytes` | Controls the display of the `Bytes` column in the execution plan table. |
| `cost` | Controls the display of the `Cost` column in the execution plan table. |

(*continued*)

***Table 10-3.*** (*continued*)

| Value | Description |
|---|---|
| note | Controls the display of the section containing the notes. |
| outline | Controls the display of the section containing the outline. |
| parallel | Controls the display of parallel processing information, specifically, the TQ, IN-OUT, and PQ Distrib columns in the execution plan table. These columns aren't shown in the previous examples. |
| partition | Controls the display of partitioning information, specifically the Pstart and Pstop columns in the execution plan table. These columns aren't shown in the previous examples. |
| peeked_binds | Controls the display of the section containing peeked bind variables. |
| predicate | Controls the display of the section containing filter, access and storage predicates. |
| projection | Controls the display of the section containing column projection information. |
| remote | Controls the display of SQL statements executed remotely. This section isn't shown in the previous examples. |
| report | Controls the activation of the reporting mode. When enabled, additional information about adaptive and reoptimized execution plans is displayed. This section isn't shown in the previous examples. Refer to the "Adaptive Execution Plans" section later on for an example. This modifier is available from version 12.1 onward only. |
| rows | Controls the display of the Rows column in the execution plan table. |

- `filter_preds` specifies a restriction applied while querying the plan table. The restriction is a regular SQL predicate based on one of the columns of the plan table (for example, `statement_id = 'test3'`). The default value is `NULL`. If the default value is used, the execution plan most recently inserted into the plan table is displayed.

To use the `display` function, the caller requires only the `EXECUTE` privilege on the package and the `SELECT` privilege on the plan table.

The following queries, which display the same execution plan, show the main differences between the primitive values `basic`, `typical`, and `advanced`. Here's an excerpt of the output generated by the `display.sql` script:

```
SQL> SELECT * FROM table(dbms_xplan.display(NULL, NULL, 'basic'));

PLAN_TABLE_OUTPUT
-----------------------------------

Plan hash value: 2966233522

-----------------------------------
| Id  | Operation          | Name |
-----------------------------------
|   0 | SELECT STATEMENT   |      |
|   1 |  SORT AGGREGATE    |      |
|   2 |   TABLE ACCESS FULL| T    |
-----------------------------------
```

```
SQL> SELECT * FROM table(dbms_xplan.display(NULL, NULL, 'typical'));

PLAN_TABLE_OUTPUT
---------------------------------------------------------------------------

Plan hash value: 2966233522

---------------------------------------------------------------------------
| Id  | Operation          | Name | Rows  | Bytes | Cost (%CPU)| Time     |
---------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |      |     1 |     4 |     2   (0)| 00:00:01 |
|   1 |  SORT AGGREGATE    |      |     1 |     4 |            |          |
|*  2 |   TABLE ACCESS FULL| T    |    15 |    60 |     2   (0)| 00:00:01 |
---------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   2 - filter("N"<=19 AND "N">=6)

SQL> SELECT * FROM table(dbms_xplan.display(NULL, NULL, 'advanced'));

PLAN_TABLE_OUTPUT
---------------------------------------------------------------------------

Plan hash value: 2966233522

---------------------------------------------------------------------------
| Id  | Operation          | Name | Rows  | Bytes | Cost (%CPU)| Time     |
---------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |      |     1 |     4 |     2   (0)| 00:00:01 |
|   1 |  SORT AGGREGATE    |      |     1 |     4 |            |          |
|*  2 |   TABLE ACCESS FULL| T    |    15 |    60 |     2   (0)| 00:00:01 |
---------------------------------------------------------------------------

Query Block Name / Object Alias (identified by operation id):
-------------------------------------------------------------

   1 - SEL$1
   2 - SEL$1 / T@SEL$1

Outline Data
-------------

  /*+
      BEGIN_OUTLINE_DATA
      FULL(@"SEL$1" "T"@"SEL$1")
      OUTLINE_LEAF(@"SEL$1")
      ALL_ROWS
      DB_VERSION('11.2.0.3')
      OPTIMIZER_FEATURES_ENABLE('11.2.0.3')
      IGNORE_OPTIM_EMBEDDED_HINTS
      END_OUTLINE_DATA
  */
```

```
Predicate Information (identified by operation id):
---------------------------------------------------

   2 - filter("N"<=19 AND "N">=6)

Column Projection Information (identified by operation id):
-----------------------------------------------------------

   1 - (#keys=0) COUNT(*)[22]
```

The following queries show how to use modifiers to add or remove information from the default output generated by the primitive values `basic` and `typical`. Because they're based on the same query as the previous examples, you can compare the outputs to see what's different. This is an excerpt of the output generated by the `display.sql` script:

```
SQL> SELECT * FROM table(dbms_xplan.display(NULL, NULL, 'basic +predicate'));

PLAN_TABLE_OUTPUT
-----------------------------------

Plan hash value: 2966233522

-----------------------------------
| Id  | Operation          | Name |
-----------------------------------
|   0 | SELECT STATEMENT   |      |
|   1 |  SORT AGGREGATE    |      |
|*  2 |   TABLE ACCESS FULL| T    |
-----------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   2 - filter("N"<=19 AND "N">=6)

SQL> SELECT * FROM table(dbms_xplan.display(NULL, NULL, 'typical -bytes -note'));

PLAN_TABLE_OUTPUT
-------------------------------------------------------------------

Plan hash value: 2966233522

-------------------------------------------------------------------
| Id  | Operation          | Name | Rows  | Cost (%CPU)| Time     |
-------------------------------------------------------------------
|   0 | SELECT STATEMENT   |      |     1 |     2   (0)| 00:00:01 |
|   1 |  SORT AGGREGATE    |      |     1 |            |          |
|*  2 |   TABLE ACCESS FULL| T    |    15 |     2   (0)| 00:00:01 |
-------------------------------------------------------------------
```

```
Predicate Information (identified by operation id):
---------------------------------------------------

   2 - filter("N"<=19 AND "N">=6)
```

If you use the EXPLAIN PLAN statement and the display function when the current_schema session parameter is set to a schema that owns a plan table that has the default name, you have to add the schema name to both the INTO clause of the EXPLAIN PLAN statement and the table_name parameter. Failing to do so causes the display function to raise an error message. The following example illustrates:

```
SQL> ALTER SESSION SET current_schema = franco;

SQL> EXPLAIN PLAN FOR SELECT * FROM t;

SQL> SELECT * FROM table(dbms_xplan.display);

PLAN_TABLE_OUTPUT
-----------------------------------------------------

Error: cannot fetch last explain plan from PLAN_TABLE

SQL> EXPLAIN PLAN INTO franco.plan_table FOR SELECT * FROM t;

SQL> SELECT * FROM table(dbms_xplan.display(table_name=>'franco.plan_table'));

PLAN_TABLE_OUTPUT
--------------------------------------------------------------------------

Plan hash value: 3956160932

--------------------------------------------------------------------------
| Id  | Operation         | Name | Rows  | Bytes | Cost (%CPU)| Time     |
--------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |      |    14 |  1218 |     3   (0)| 00:00:01 |
|   1 |  TABLE ACCESS FULL| T    |    14 |  1218 |     3   (0)| 00:00:01 |
--------------------------------------------------------------------------
```

It's also possible with the display function to query a plan table that has a structure based on the v$sql_plan_statistics_all view. This feature is useful when you want to persist information that, by design, is only temporarly available in the library cache. Since such a plan table contains additional information, when querying it through the display function, the format parameter supports the additional modifiers described in the next section, specifically in Table 10-4. The following example shows how you could take advantage of this feature to persist information about the last SQL statement to be executed:

```
SQL> SELECT /*+ gather_plan_statistics */ count(*) FROM t;

  COUNT(*)
----------
      1000
```

```
SQL> CREATE TABLE my_plan_table
  2  AS
  3  SELECT cast(1 AS VARCHAR2(30)) AS plan_id, p.*
  4  FROM v$sql_plan_statistics_all p
  5  WHERE (sql_id, child_number) = (SELECT prev_sql_id, prev_child_number
  6                                  FROM v$session
  7                                  WHERE sid = sys_context('userenv','sid'));

SQL> SELECT * FROM table(dbms_xplan.display('my_plan_table', NULL, 'iostats'));

PLAN_TABLE_OUTPUT
-------------------------------------------------------------------------------------------

Plan hash value: 2966233522

-------------------------------------------------------------------------------------------
| Id  | Operation          | Name | Starts | E-Rows | A-Rows |   A-Time   | Buffers | Reads  |
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |      |      2 |        |      2 |00:00:00.01 |      10 |      4 |
|   1 |  SORT AGGREGATE    |      |      2 |      1 |      2 |00:00:00.01 |      10 |      4 |
|   2 |   TABLE ACCESS FULL| T    |      2 |   1000 |   2000 |00:00:00.01 |      10 |      4 |
-------------------------------------------------------------------------------------------
```

***Table 10-4.*** *Modifiers Accepted by the `format` Parameter*

| Value | Description |
|---|---|
| `allstats` | This is a shortcut for `iostats memstats`. |
| `iostats` | Controls the display of runtime statistics (columns `Starts`, `A-Rows`, `A-Time`), the estimated number of rows (column `E-Rows`), and disk I/O statistics (columns `Buffers`, `Reads` and `Writes`). |
| `last` | Per default, the `allstats`, `iostats`, `memstats`, and `rowstats` modifiers display the cumulated statistics of all executions. If this value is added to them, only the statistics of the last execution are displayed. Specifying this modifier for SQL statements executed with parallel processing doesn't work as you might expect. More information about this in the "Parallel Processing" section in Chapter 15. |
| `memstats` | Controls the display of memory utilization statistics (columns `OMem`, `1Mem`, `O/1/M`, `Used-Mem`, `Used-Tmp` and `Max-Tmp`). |
| `rowstats` | Controls the display of row count statistics (columns `Starts`, `E-Rows and A-Rows`). This modifier is available as of version 11.2.0.4 only. |
| `runstats_last` | Same as `iostats last`. This modifier is deprecated and available for backward compatibility only. |
| `runstats_tot` | Same as `iostats`. This modifier is deprecated and available for backward compatibility only. |

## The display_cursor Function

The `display_cursor` function returns execution plans stored in the library cache. Note that, in a Real Application Clusters environment, it's not possible to get an execution plan stored in a remote instance. As for the `display` function, the return value is an instance of the `dbms_xplan_type_table` collection. The function has the following input parameters:

- `sql_id` specifies the parent cursor whose execution plan is returned. The default value is `NULL`. If the default value is used, the execution plan of the last SQL statement executed by the current session is returned.
- `cursor_child_no` specifies the child number that, along with `sql_id`, identifies the child cursor whose execution plan is returned. The default value is 0. If `NULL` is specified, all child cursors of the parent cursor identified by the `sql_id` parameter are returned.
- `format` specifies which information is displayed. The same values are supported as in the parameter format of the `display` function. In addition, if execution statistics are available (in other words, if the `statistics_level` initialization parameter is set to `all` or the `gather_plan_statistics` hint is specified in the SQL statement), the modifiers described in Table 10-4 are also supported. The default value is `typical`.

---

■ **Caution** As pointed out in Chapter 2, sometimes the `sql_id` and `child_number` columns aren't sufficient to identify a child cursor in the `v$sql` view. In such a case, because of bug 14585499, and up to and including version 11.2.0.3, the `display_cursor` function returns wrong data. To recognize this problem, look for the following error message in the `display_cursor` function output:

```
An uncaught error happened in prepare_sql_statement : ORA-01422: exact fetch returns more than requested
number of rows
```

You can use the `display_cursor_ora-01422.sql` script to reproduce the bug.

---

To use the `display_cursor` function, the caller requires the `SELECT` privilege on the following dynamic performance views: `v$session`, `v$sql`, `v$sql_plan`, and `v$sql_plan_statistics_all`. The `select_catalog_role` role and the `select any dictionary` system privilege provide these privileges, among others.

---

■ **Note** The modifiers listed in Table 10-4 have the side effect of removing from the output the following columns related to the query optimizer estimations: `Bytes`, `TempSpc`, `Cost (%CPU)`, `Time`. If you want one of these columns in the output, you have to explicitly specify it either through a primitive value or a modifier.

---

The following example shows a query that uses the `gather_plan_statistics` hint to enable the generation of the execution statistics. The `display_cursor` function is then instructed to display the disk I/O statistics for the last execution. Because no physical read or writes took place, only logical reads (`Buffers`) are displayed. Here's an excerpt of the output generated by the `display_cursor.sql` script:

```
SQL> SELECT /*+ gather_plan_statistics */ count(pad)
  2  FROM (SELECT rownum AS rn, pad FROM t ORDER BY n)
  3  WHERE rn = 1;
```

```
COUNT(PAD)
----------
         1

SQL> SELECT * FROM table(dbms_xplan.display_cursor('d5v0dt28fp5fh', 0, 'iostats last'));

PLAN_TABLE_OUTPUT
-----------------------------------------------------------------------------------------

SQL_ID  d5v0dt28fp5fh, child number 0
-------------------------------------
SELECT /*+ gather_plan_statistics */ count(pad) FROM (SELECT rownum AS
rn, pad FROM t ORDER BY n) WHERE rn = 1

Plan hash value: 2545006537

-----------------------------------------------------------------------------------------
| Id  | Operation             | Name | Starts | E-Rows | A-Rows |   A-Time   | Buffers |
-----------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT      |      |      1 |        |      1 |00:00:00.02 |     147 |
|   1 |  SORT AGGREGATE       |      |      1 |      1 |      1 |00:00:00.02 |     147 |
|*  2 |   VIEW                |      |      1 |   1000 |      1 |00:00:00.02 |     147 |
|   3 |    SORT ORDER BY      |      |      1 |   1000 |   1000 |00:00:00.02 |     147 |
|   4 |     COUNT             |      |      1 |        |   1000 |00:00:00.01 |     145 |
|   5 |      TABLE ACCESS FULL| T    |      1 |   1000 |   1000 |00:00:00.01 |     145 |
-----------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   2 - filter("RN"=1)
```

## The display_awr Function

The `display_awr` function returns execution plans stored in AWR. As for the `display` function, the return value is an instance of the `dbms_xplan_type_table` collection. The function has the following input parameters:

- `sql_id` specifies the SQL statement whose execution plan is returned. The parameter has no default value.
- `plan_hash_value` specifies the hash value of the execution plan to be returned. The default value is `NULL`. If the default value is used, all execution plans related to the SQL statement identified by the `sql_id` parameter are returned.
- `db_id` specifies which database the execution plan to be returned was executed on. The default value is `NULL`. If the default value is used, the current database is used.
- `format` specifies which information is displayed. Even though the same values as the `format` parameter of the `display` function are supported, not all information can be displayed. For example, information about predicates is missing because AWR doesn't store it. The default value is `typical`.

To use the display_awr function, the caller requires at least the SELECT privilege on the following data dictionary views: dba_hist_sql_plan and dba_hist_sqltext. If the db_id parameter isn't specified, the SELECT privilege on the v$database view is necessary as well. The select_catalog_role role provides these privileges, among others.

The following queries show the usefulness of the plan_hash_value parameter whenever several execution plans exist for a given SQL statement. Note that whereas the first query returns two execution plans, the second query returns only one. Here's an excerpt of the output generated by the display_awr.sql script:

```
SQL> SELECT * FROM table(dbms_xplan.display_awr('48vuyqjwpf9wg', NULL, NULL, 'basic'));

PLAN_TABLE_OUTPUT
-----------------------------------

SQL_ID 48vuyqjwpf9wg
--------------------
SELECT COUNT(N) FROM T

Plan hash value: 2966233522

-----------------------------------
| Id  | Operation          | Name |
-----------------------------------
|   0 | SELECT STATEMENT   |      |
|   1 |  SORT AGGREGATE    |      |
|   2 |   TABLE ACCESS FULL| T    |
-----------------------------------

SQL_ID 48vuyqjwpf9wg
--------------------
SELECT COUNT(N) FROM T

Plan hash value: 3776247601

--------------------------------------
| Id  | Operation             | Name |
--------------------------------------
|   0 | SELECT STATEMENT      |      |
|   1 |  SORT AGGREGATE       |      |
|   2 |   INDEX FAST FULL SCAN| I    |
--------------------------------------

SQL> SELECT * FROM table(dbms_xplan.display_awr('48vuyqjwpf9wg', 2966233522, NULL, 'basic'));

PLAN_TABLE_OUTPUT
-----------------------------------------------------------------------------------------

SQL_ID 48vuyqjwpf9wg
--------------------
SELECT COUNT(N) FROM T
```

```
Plan hash value: 2966233522

-----------------------------------
| Id  | Operation         | Name |
-----------------------------------
|   0 | SELECT STATEMENT   |      |
|   1 |  SORT AGGREGATE    |      |
|   2 |   TABLE ACCESS FULL| T    |
-----------------------------------
```

Several situations lead to multiple execution plans for a given SQL statement, such as when an index has been added or simply because data (and therefore its object statistics) has changed. Basically, each time the environment that the query optimizer evolves in changes, different execution plans may be generated. Such an output is useful when you're questioning the performance of a SQL statement that you think has been running without problems for some time. The idea is to check whether a SQL statement has been executed with several execution plans over a period of time. If this is the case, infer what could be the reason leading to the change based on the available information.

## Interpreting Execution Plans

I've always found it surprising how little documentation there is about how to read execution plans, especially since there seem to be so many people who are unable to correctly read them. I attempt to address this problem by describing the approach I use when reading an execution plan. Note that details about the different operations aren't provided here; rather, I provide the basics you need in order to understand how to walk through execution plans. I give detailed information about the most common operations in Part 4.

■ **Caution** Parallel processing makes the interpretation of execution plans more difficult. The reason is quite simple: several operations run concurrently. This section, to keep the description as simple as possible, doesn't pretend to cover parallel processing. Information about execution plans processed in parallel is provided in Chapter 15.

### Parent-Child Relationship

An execution plan is a tree describing not only which order the SQL engine executes operations in but also what the relationship between operations is. Each node in the tree is a *row source operation* (actually implemented as a function written in C)—for example, a table access, a join, or a sort. Between operations (nodes), there's a parent-child relationship. Understanding those relationships is essential to correctly reading an execution plan. When an execution plan is displayed in a textual form, the rules governing the parent-child relationship are the following:

- A parent has one or multiple children.
- A child has a single parent.
- The only operation without a parent is the root of the tree.
- Children are indented to the right, with respect to their parent. Depending on the method used to display the execution plan, the indentation could be a single space character, two spaces, or something else. It doesn't really matter. The essential point is that all children of a specific parent have the very same indentation.
- A parent is placed before its children (the ID of the parent is less than the ID of the children). If, for a child, there are several preceding operations with the same indentation as the parent, the nearest operation is the parent.

The following is a sample execution plan generated by the `relationship.sql` script. Note that although only the `Operation` column is needed to walk through an execution plan, the `Id` column is shown here to help you identify the operations more easily. The SQL statement used to generate it has been left out intentionally because it doesn't serve our purposes for this section:

```
---------------------------------------------
| Id  | Operation                      | Name |
---------------------------------------------
|   0 | UPDATE STATEMENT               |      |
|   1 |  UPDATE                        | T    |
|   2 |   NESTED LOOPS                 |      |
|   3 |    TABLE ACCESS FULL           | T    |
|   4 |    INDEX UNIQUE SCAN           | T_PK |
|   5 |   SORT AGGREGATE               |      |
|   6 |    TABLE ACCESS BY INDEX ROWID| T    |
|   7 |     INDEX FULL SCAN            | I    |
|   8 |   TABLE ACCESS BY INDEX ROWID | T    |
|   9 |    INDEX UNIQUE SCAN           | T_PK |
---------------------------------------------
```

Figure 10-2 provides a graphical representation of the execution plan. Using the rules described earlier, you can conclude the following:

- Operation 0 is the root of the tree. It informs you about the type of SQL statement to which the execution plan is associated. Operation 0 has one child: Operation 1.
- Operation 1 has three children: 2, 5, and 8.
- Operation 2 has two children: 3 and 4.
- Operations 3 and 4 have no children.
- Operation 5 has one child: 6.
- Operation 6 has one child: 7.
- Operation 7 has no children.
- Operation 8 has one child: 9.
- Operation 9 has no children.

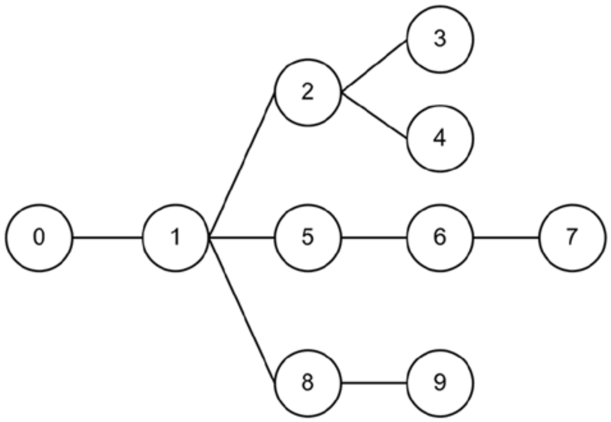


***Figure 10-2.** Parent-child relationships between execution plan operations*

Knowing the parent-child relationship is essential to understanding the order in which operations of an execution plan are executed. In fact, parent operations, to fulfill their task, require data that is provided by their child operations. As a result, even though execution starts at the root of the tree, the first operation to be fully executed is one that has no child and, therefore, is a leaf of the tree. To illustrate, let's take a look at the following execution plan:

```
---------------------------------------------
| Id  | Operation                    | Name |
---------------------------------------------
|   0 | SELECT STATEMENT             |      |
|   1 |  SORT ORDER BY               |      |
|   2 |   TABLE ACCESS BY INDEX ROWID| T    |
|   3 |    INDEX RANGE SCAN          | T_PK |
---------------------------------------------
```

The operations are executed as follows:

1. The entry point of the execution plan is operation 0, the root of the tree. However, operation 0, a SELECT statement, has no data to work on. Hence, it has to call its child (1).
2. Operation 1, a sort, has no data to work on. Hence, it has to call its child (2).
3. Operation 2, a table access, requires rowids to access the t table. Hence, it has to call its child (3).
4. Operation 3, an index access, requires no data from another operation (it has no child). Therefore, it carries out the index range scan on the t_pk index and passes the rowids it finds to its parent (2).
5. Operation 2 uses the list of rowids it receives from its child (3) to access the t table. Then, it passes the resulting data to its parent (1).
6. Operation 1 sorts the data passed by its child (2) and passes the resulting data to its parent (0).
7. Operation 0 passes the data received from its child (1) to the caller.

---

■ **Note** Even though the first operation being executed is always the root of the tree, parent operations (three in the previous case) may do nothing other than to invoke a child operation. So, for simplicity, I usually say that the execution starts with the first operation that can do some real work (this is operation 3 in the previous case).

---

The following three general rules summarize the behavior just described:

- Parent operations call child operations.
- Child operations are fully executed before their parent operations.
- Child operations pass data to their parent operations.

## Types of Operations

There are hundreds of different operations. Of course, to fully understand an execution plan, you should know what each operation it's made of does. For our purpose of walking through an execution plan, you need to consider only four major types of operations: *stand-alone operations*, *iterative operations*, *unrelated-combine operations*, and *related-combine operations*. Basically, each type has a particular behavior, and knowing it is sufficient for reading execution plans.

---

■ **Caution** I coined the terms used here for the four types of operations while writing a presentation about the query optimizer in 2007. Don't expect to find these terms used elsewhere.

---

In addition to these four types, operations can be separated into blocking operations and nonblocking operations. Simply put, blocking operations process data in bulk, whereas nonblocking operations process data row by row. For example, a sort operation is blocking because it's able to return the output rows only when all input rows have been fully processed (sorted)—since the first output row could be anywhere in the input set. On the other hand, a filter applying a simple restriction is nonblocking because it evaluates each row independently. It goes without saying that for blocking operations, data must be buffered either in memory (PGA) or on disk (temporary tablespace). For simplicity, while walking through an execution plan, you can consider all operations to be blocking. Remember, though, that most of the operations are in fact nonblocking and that the SQL engine, for obvious reasons, tries to avoid the buffering of data as much as possible.

## Stand-Alone Operations

I identify as *stand-alone operations* all operations having at most one child and that are not *iterative operations* (covered in the next section). Most operations are stand-alone. This makes the interpretation of execution plans easier because less than two dozen operations aren't of this type. The rules governing the working of stand-alone operations are the ones described in the "Parent-Child Relationship" section, with the following addition:

- A child operation is executed at most once.

Here's an example of a query and its execution plan based on the output generated by the `stand-alone.sql` script (Figure 10-3 provides a graphical representation of its parent-child relationships):

```
SELECT deptno, count(*)
FROM emp
WHERE job = 'CLERK' AND sal < 1200
GROUP BY deptno

------------------------------------------------------------------
| Id  | Operation                    | Name     | Starts | A-Rows |
------------------------------------------------------------------
|   0 | SELECT STATEMENT             |          |      1 |      2 |
|   1 |  HASH GROUP BY               |          |      1 |      2 |
|*  2 |   TABLE ACCESS BY INDEX ROWID| EMP      |      1 |      3 |
|*  3 |    INDEX RANGE SCAN          | EMP_JOB_I |      1 |      4 |
------------------------------------------------------------------

   2 - filter("SAL"<1200)
   3 - access("JOB"='CLERK')
```

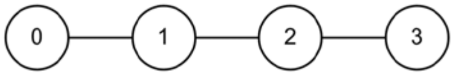


***Figure 10-3.*** *Parent-child relationships between stand-alone operations*

This execution plan consists only of stand-alone operations. By applying the rules described earlier, you find out that the execution plan carries out the operations as follows:

1. Operations 0, 1 and 2 have a single child each (1, 2 and 3, respectively); they can't be the first operations being executed. Therefore, the execution starts with operation 3.
2. Operation 3 scans the `emp_job_i` index by applying the `"JOB"='CLERK'` access predicate. In doing so, it extracts four rowids (this information is given in the `A-Rows` column) from the index and passes them to its parent (2).
3. Operation 2 accesses the `emp` table through the four rowids passed from operation 3. For each rowid, a row is read. Then, it applies the `"SAL"<1200` filter predicate. This filter leads to the exclusion of one row. The data of the remaining three rows is passed to its parent (1).
4. Operation 1 performs a `GROUP BY` on the rows passed from operation 2. The resulting set is reduced to two rows and passed to its parent (0).
5. Operation 0 sends the data to the caller.

Notice how the `Starts` column clearly shows that each operation is executed only once.

One of the rules states that child operations are entirely executed before parent operations. This is generally true, but there are situations where smart optimizations are introduced. What can happen is that a parent decides that it makes no sense to completely execute a child or even that it makes no sense to execute it at all. In other words, parents control the execution of children. Let's take a look at two common cases. Note that both examples are excerpts of the output generated by the `stand-alone.sql` script.

## Optimization of the COUNT STOPKEY Operation

The `COUNT STOPKEY` operation is commonly used to execute top-n queries. Its aim is to stop the processing as soon as the required number of rows has been returned to the caller. For example, the aim of the following query is to return only the first ten rows found in the `emp` table:

```
SELECT *
FROM emp
WHERE rownum <= 10

-----------------------------------------------------
| Id  | Operation          | Name | Starts | A-Rows |
-----------------------------------------------------
|   0 | SELECT STATEMENT   |      |      1 |     10 |
|*  1 |  COUNT STOPKEY     |      |      1 |     10 |
|   2 |   TABLE ACCESS FULL| EMP  |      1 |     10 |
-----------------------------------------------------

   1 - filter(ROWNUM<=10)
```

The important thing to notice in this execution plan is that the number of rows returned by operation 2 is limited to ten. This is true even if operation 2 is a full table scan of a table containing more than 10 rows (actually the table contains 14 rows). What happens is that operation 1 stops the processing of operation 2 as soon as the necessary number of rows has been processed. Be careful, though, because blocking operations can't be stopped. In fact, they need to be fully processed before returning rows to their parent operation. For example, in the following query, all rows (14) of the emp table are read because of the ORDER BY clause:

```
SELECT *
FROM (
  SELECT *
  FROM emp
  ORDER BY sal DESC
)
WHERE rownum <= 10

----------------------------------------------------------
| Id  | Operation              | Name | Starts | A-Rows |
----------------------------------------------------------
|   0 | SELECT STATEMENT       |      |      1 |     10 |
|*  1 |  COUNT STOPKEY         |      |      1 |     10 |
|   2 |   VIEW                 |      |      1 |     10 |
|*  3 |    SORT ORDER BY STOPKEY|     |      1 |     10 |
|   4 |     TABLE ACCESS FULL  | EMP  |      1 |     14 |
----------------------------------------------------------

   1 - filter(ROWNUM<=10)
   3 - filter(ROWNUM<=10)
```

## Optimization of the FILTER Operation

The FILTER operation not only applies a filter when its child passes data to it, but in addition, it could decide to completely avoid the execution of a child and all the dependent operations (grandchild and so on) as well. For example, in the following query, a filter predicate applied from operation 1 checks whether the value of the bind variables leads to an empty result set or not. In fact, the query can return rows only if the :SAL_MIN<=:SAL_MAX filter predicate is fulfilled:

```
SELECT *
FROM emp
WHERE sal BETWEEN :sal_min AND :sal_max

-----------------------------------------------------
| Id  | Operation          | Name | Starts | A-Rows |
-----------------------------------------------------
|   0 | SELECT STATEMENT   |      |      1 |      0 |
|*  1 |  FILTER            |      |      1 |      0 |
|*  2 |   TABLE ACCESS FULL| EMP  |      0 |      0 |
-----------------------------------------------------

   1 - filter(:SAL_MIN<=:SAL_MAX)
   2 - filter(("SAL"<=:SAL_MAX AND "SAL">=:SAL_MIN))
```

According to the rules described earlier, operation 2 should be the first fully executed operation in the execution plan shown. In reality, looking at the `Starts` column tells you that only operations 0 and 1 are executed. The optimization simply avoids processing operation 2, because the data has no chance of going through the filter applied by operation 1 anyway.

## Iterative Operations

I identify all operations that have at most one child that can be executed more than once as *iterative operations*. You can consider them as operations that implement a sort of loop in an execution plan. The `INLIST ITERATOR` and most of the operations that have the `PARTITION` suffix (for example, `PARTITION RANGE ITERATOR`; refer to Chapter 13 for a detailed description of these operations) are of this type. The rules governing the working of iterative operations are the ones described in the "Parent-Child Relationship" section with, in addition, the following:

- A child operation may be executed several times or not executed at all.

Here's an example of a query and its execution plan based on the output generated by the `iterative.sql` script:

```
SELECT *
FROM emp
WHERE job IN ('CLERK', 'ANALYST')

--------------------------------------------------------------------
| Id  | Operation                   | Name      | Starts | A-Rows |
--------------------------------------------------------------------
|   0 | SELECT STATEMENT            |           |      1 |      6 |
|   1 |  INLIST ITERATOR            |           |      1 |      6 |
|   2 |   TABLE ACCESS BY INDEX ROWID| EMP      |      2 |      6 |
|*  3 |    INDEX RANGE SCAN         | EMP_JOB_I |      2 |      6 |
--------------------------------------------------------------------

   3 - access(("JOB"='ANALYST' OR "JOB"='CLERK'))
```

The execution plan is similar to the one discussed for stand-alone operations. The only difference is that part of the execution plan, because of the `INLIST ITERATOR` operation, can be executed several times. Specifically, the child of an iterative operation can be executed several times. In this case, operations 2 and 3 are executed once for every distinct value specified in the `IN` condition.

## Unrelated-Combine Operations

I call all operations having multiple children that are independently executed *unrelated-combine operations*. The following operations are of this type: `AND-EQUAL`, `BITMAP AND`, `BITMAP OR`, `BITMAP MINUS`, `CONCATENATION`, `CONNECT BY WITHOUT FILTERING`, `HASH JOIN`, `INTERSECTION`, `MERGE JOIN`, `MINUS`, `MULTI-TABLE INSERT`, `SQL MODEL`, `TEMP TABLE TRANSFORMATION`, and `UNION-ALL`. The rules governing the working of unrelated-combine operations are the ones described in the "Parent-Child Relationship" section, with the following additions:

- Children are executed sequentially, starting from the one with the smallest ID and going to the one with the highest ID. Before starting the processing of a subsequent child, the current one must be completely executed.
- A child is executed at most once and independently from all other children.

■ **Note** There are specific cases where the children of the `MERGE JOIN` operation aren't exactly executed according to the two rules just mentioned. The "Merge Joins" section in Chapter 14 provides information about such specifc cases.

Here's a sample query and its execution plan based on the output generated by the `unrelated-combine.sql` script (see Figure 10-4 for a graphical representation of its parent-child relationships):

```
SELECT ename FROM emp
UNION ALL
SELECT dname FROM dept
UNION ALL
SELECT '%' FROM dual

-----------------------------------------------------
| Id  | Operation           | Name | Starts | A-Rows |
-----------------------------------------------------
|   0 | SELECT STATEMENT    |      |      1 |     19 |
|   1 |  UNION-ALL          |      |      1 |     19 |
|   2 |   TABLE ACCESS FULL| EMP  |      1 |     14 |
|   3 |   TABLE ACCESS FULL| DEPT |      1 |      4 |
|   4 |   FAST DUAL         |      |      1 |      1 |
-----------------------------------------------------
```

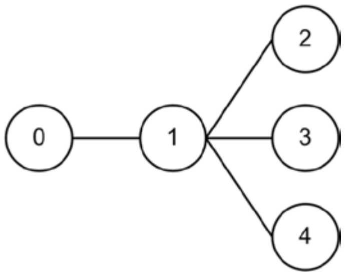


***Figure 10-4.*** *Parent-child relationships of the `UNION-ALL` unrelated-combine operation*

In this execution plan, the unrelated-combine operation is the `UNION-ALL`. The other three are stand-alone operations. By applying the rules given earlier, you see that the execution plan carries out the operations as follows:

1. Operation 0 has one child (1). It can't be the first one being executed.
2. Operation 1 has three children, and, among them, operation 2 is the first in ascending order. Therefore, the execution starts with operation 2.
3. Operation 2 scans the `emp` table and returns 14 rows to its parent (1).
4. When operation 2 is completely executed, operation 3 is started.
5. Operation 3 scans the `dept` table and returns four rows to its parent (1).
6. When operation 3 is completely executed, operation 4 is started.

7. Operation 4 scans the dual table and returns one row to its parent (1).

8. Operation 1 builds a single result set of 19 rows based on all data received from its children and returns them to its parent (0).

9. Operation 0 sends the data to the caller.

Notice how the Starts column clearly shows that each operation is executed only once.

In Table 10-1 I mention that there are cases where the meaning of the Starts column is a bit different. Instead of the number of executions, the column sometimes provides the number of times a specific memory structure is accessed. The MERGE JOIN operation can be used to show such a case, as the following example illustrates. Notice how the Starts column has the value 4 for operation 4. However, it makes no sense to sort the data four times. What is happening is that the memory structure is being accessed four times, and hence the value 4 in the Starts column. The structure is accessed once for each row extracted from the dept table (Chapter 14 explains in detail how merge joins are executed):

```
---------------------------------------------------------------
| Id  | Operation            | Name | Starts | E-Rows | A-Rows |
---------------------------------------------------------------
|   0 | SELECT STATEMENT     |      |      1 |        |     14 |
|   1 |  MERGE JOIN          |      |      1 |     14 |     14 |
|   2 |   SORT JOIN          |      |      1 |      4 |      4 |
|   3 |    TABLE ACCESS FULL| DEPT |      1 |      4 |      4 |
|*  4 |   SORT JOIN          |      |      4 |     14 |     14 |
|   5 |    TABLE ACCESS FULL| EMP  |      1 |     14 |     14 |
---------------------------------------------------------------

   4 - access("E"."DEPTNO"="D"."DEPTNO")
       filter("E"."DEPTNO"="D"."DEPTNO")
```

All other operations listed earlier have the same behavior as the UNION-ALL operation shown in this section. In short, an unrelated-combine operation sequentially executes its children once. Obviously, the processing performed by the unrelated-combine operation itself is different.

## Related-Combine Operations

I refer to all operations having multiple children where one of the children controls the execution of all other children as *related-combine operations*. The following operations are of this type: NESTED LOOPS, FILTER, UPDATE, CONNECT BY WITH FILTERING, UNION ALL (RECURSIVE WITH), and BITMAP KEY ITERATION. The rules governing the working of related-combine operations are the ones described in the "Parent-Child Relationship" section with the following additions:

- The child with the smallest ID controls the execution of the other children.
- Children are executed going from the one with the smallest ID to the one with the highest ID. Contrary to unrelated-combine operations, however, they aren't executed sequentially. Instead, a kind of interleaving is performed.
- Only the first child is executed at most once. All other children may be executed several times or not executed at all.

Even if the operations of this type share the same characteristics, each of them has, to some extent, its own behavior. Let's take a look at an example for each of them (except for `BITMAP KEY ITERATION`, which is covered in Chapter 14). Note that all examples provided in the following subsections are excerpts of the output generated by the `related-combine.sql` script.

## The NESTED LOOPS Operation

This operation is used to join two sets of rows. Consequently, it always has two children, no more, no less. The child with the smallest ID is called the *outer loop* or *driving row source*. The second child is called the *inner loop*. The particular characteristic of this operation is that the inner loop is executed once for each row returned by the outer loop (Chapter 14 explains in detail how nested loops joins are executed).

The following query and its execution plan are an example (Figure 10-5 shows a graphical representation of its parent-child relationships):

```
SELECT *
FROM emp, dept
WHERE emp.deptno = dept.deptno
AND emp.comm IS NULL
AND dept.dname != 'SALES'
```

```
-----------------------------------------------------------------
| Id  | Operation                    | Name    | Starts | A-Rows |
-----------------------------------------------------------------
|   0 | SELECT STATEMENT             |         |      1 |      8 |
|   1 |  NESTED LOOPS                |         |      1 |      8 |
|*  2 |   TABLE ACCESS FULL          | EMP     |      1 |     10 |
|*  3 |   TABLE ACCESS BY INDEX ROWID| DEPT    |     10 |      8 |
|*  4 |    INDEX UNIQUE SCAN         | DEPT_PK |     10 |     10 |
-----------------------------------------------------------------

   2 - filter("EMP"."COMM" IS NULL)
   3 - filter("DEPT"."DNAME"<>'SALES')
   4 - access("EMP"."DEPTNO"="DEPT"."DEPTNO")
```

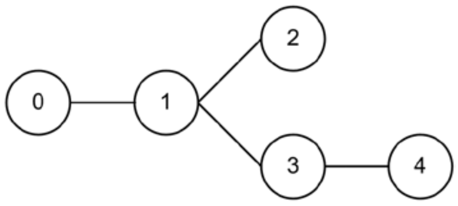


***Figure 10-5.*** *Parent-child relationships of the* `NESTED LOOPS` *operation*

In this execution plan, both children of the `NESTED LOOPS` operation are stand-alone operations. By applying the rules described earlier, you can see that the execution plan carries out the operations as follows:

1. Operation 0 has one child (1). It can't be the first one being executed.
2. Operation 1 has two children (2 and 3), and among them, operation 2 is the first in ascending order. Therefore, operation 2 (the outer loop) is the first one being executed.

3. Operation 2 scans the emp table, applies the "EMP"."COMM" IS NULL filter predicate and passes the data of ten rows to its parent (1).
4. For each row returned by operation 2, the second child of the NESTED LOOPS operation, the inner loop, is executed once. This is confirmed by comparing the A-Rows column of operation 2 with the Starts column of operations 3 and 4.
5. The inner loop is composed of two stand-alone operations. Based on the rules that apply to this type of operation, operation 4 is executed before operation 3.
6. Operation 4 scans the dept_pk index by applying the "EMP"."DEPTNO"= "DEPT"."DEPTNO"access predicate. In doing so, it extracts ten rowids from the index over the ten executions and passes them to its parent (3).
7. Operation 3 accesses the dept table through the ten rowids passed from operation 4. For each rowid, a row is read. Then it applies the "DEPT"."DNAME"<>'SALES' filter predicate. This filter leads to the exclusion of two rows. It passes the data of the remaining eight rows to its parent (1).
8. Operation 1 passes the eight rows to its parent (0)
9. Operation 0 sends the data to the caller.

## The FILTER Operation

The particular characteristic of this operation is that it supports a varying number of children. If it has a single child, it's considered a stand-alone operation. If it has two or more children, its function is similar to the NESTED LOOPS operation. The first child drives the execution of the other children.

To illustrate, the following query and its execution plan are given (Figure 10-6 shows a graphical representation of its parent-child relationships):

```
SELECT *
FROM emp
WHERE NOT EXISTS (SELECT 0
                  FROM dept
                  WHERE dept.dname = 'SALES' AND dept.deptno = emp.deptno)
AND NOT EXISTS (SELECT 0
                FROM bonus
                WHERE bonus.ename = emp.ename)
```

```
------------------------------------------------------------------
| Id  | Operation                    | Name    | Starts | A-Rows |
------------------------------------------------------------------
|   0 | SELECT STATEMENT             |         |      1 |      8 |
|*  1 |  FILTER                      |         |      1 |      8 |
|   2 |   TABLE ACCESS FULL          | EMP     |      1 |     14 |
|*  3 |   TABLE ACCESS BY INDEX ROWID| DEPT    |      3 |      1 |
|*  4 |    INDEX UNIQUE SCAN         | DEPT_PK |      3 |      3 |
|*  5 |   TABLE ACCESS FULL          | BONUS   |      8 |      0 |
------------------------------------------------------------------
```

```
   1 - filter( NOT EXISTS (SELECT 0 FROM "DEPT" "DEPT" WHERE "DEPT"."DEPTNO"=:B1
               AND "DEPT"."DNAME"='SALES') AND  NOT EXISTS (SELECT 0 FROM "BONUS"
               "BONUS" WHERE "BONUS"."ENAME"=:B2))
   3 - filter("DEPT"."DNAME"='SALES')
   4 - access("DEPT"."DEPTNO"=:B1)
   5 - filter("BONUS"."ENAME"=:B1)
```

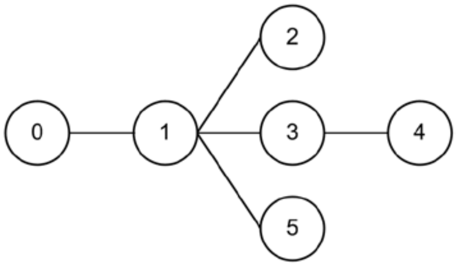


***Figure 10-6.*** *Parent-child relationships of the* FILTER *operation*

■ **Caution** The `display_cursor` function in the `dbms_xplan` package sometimes shows wrong predicates. The problem, though, isn't the package. It's actually caused by the `v$sql_plan` and `v$sql_plan_statistics_all` views that show wrong information. In this case, `EXPLAIN PLAN` shows the correct predicates shown above, but the views show a wrong predicate for operation 1:

```
1 - filter(( IS NULL AND  IS NULL))
```

Note that according to Oracle, this isn't a bug. It's just a limitation of the current implementation.

In this execution plan, the three children of the `FILTER` operation are stand-alone operations. Applying the rules described earlier, you see that the execution plan carries out the operations in the following manner:

1. Operation 0 has one child (1). It can't be the first one being executed.
2. Operation 1 has three children (2, 3 and 5), and operation 2 is the first of them in ascending order. Therefore, the execution starts with operation 2.
3. Operation 2 scans the `emp` table and returns 14 rows to its parent (1).
4. For each row returned by operation 2, the second and third children of the `FILTER` operation should be executed once. In reality, a kind of caching is implemented to reduce executions to a minimum. This is confirmed by comparing the `A-Rows` column of operation 2 with the `Starts` column of operations 3 and 5. Operation 3 is executed three times, once for each distinct value in the `deptno` column in the `emp` table. Operation 5 is executed eight times, once for each distinct value in the `ename` column in the `emp` table after applying the filter imposed by the operation 3. The following query shows that the number of starts matches the number of distinct values:

```
SQL> SELECT deptno, dname, count(*)
  2  FROM emp NATURAL JOIN dept
  3  GROUP BY deptno, dname;
```

```
DEPTNO DNAME      COUNT(*)
------ ---------- --------
    10 ACCOUNTING        3
    20 RESEARCH          5
    30 SALES             6
```

5. According to the rules for stand-alone operations, operation 4, which is executed before operation 3, scans the `dept_pk` index by applying the `"DEPT"."DEPTNO"=:B1` access predicate. The bind variable (`B1`) is used to pass the value that's to be checked by the subquery. By doing so over the three executions, it extracts three rowids from the index and passes them to its parent (3).
6. Operation 3 accesses the `dept` table through the rowids passed from its child (4) and applies the `"DEPT"."DNAME"='SALES'` filter predicate. Because this operation is used only to apply a restriction, it returns no data to its parent (1). It just informs its parent whether the condition is fulfilled. In any case, it's important to note that only one row satisfying the filter predicate was found. Since a `NOT EXISTS` is used, this matching row is discarded.
7. Operation 5 scans the `bonus` table and applies the `"BONUS"."ENAME"=:B1` filter predicate. The bind variable (`B1`) is used to pass the value to be checked by the subquery. Because this operation is used only to apply a restriction, it returns no data to its parent (1). It's important, however, to notice that no row satisfying the filter predicate was found. Since a `NOT EXISTS` is used, no rows are discarded.
8. Operation 1, after applying the filter predicate implemented with operations 3 and 5, passes the resulting rows to its parent (0).
9. Operation 0 sends the data to the caller.

## The UPDATE Operation

This operation is used when an `UPDATE` statement is executed. Its particular characteristic is that it supports a varying number of children. Most of the time, it has a single child and therefore is considered a stand-alone operation. Two or more children are available only when subqueries are used in the `SET` clause. If it has more than one child, the first child drives the execution of the other children.

Here's a sample SQL statement and its execution plan (see Figure 10-7 for a graphical representation of its parent-child relationships):

```
UPDATE emp e1
SET sal = (SELECT avg(sal) FROM emp e2 WHERE e2.deptno = e1.deptno),
    comm = (SELECT avg(comm) FROM emp e3)

---------------------------------------------------------------
| Id  | Operation             | Name | Starts | E-Rows | A-Rows |
---------------------------------------------------------------
|   0 | UPDATE STATEMENT      |      |      1 |        |      0 |
|   1 |  UPDATE               | EMP  |      1 |        |      0 |
|   2 |   TABLE ACCESS FULL   | EMP  |      1 |     14 |     14 |
|   3 |   SORT AGGREGATE      |      |      3 |      1 |      3 |
|*  4 |    TABLE ACCESS FULL| EMP  |      3 |      5 |     14 |
|   5 |   SORT AGGREGATE      |      |      1 |      1 |      1 |
|   6 |    TABLE ACCESS FULL| EMP  |      1 |     14 |     14 |
---------------------------------------------------------------
```

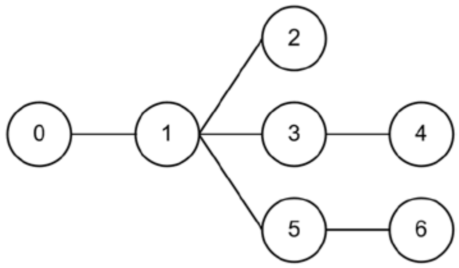


***Figure 10-7.*** *Parent-child relationships of the UPDATE operation*

```
   4 - filter("E2"."DEPTNO"=:B1)
```

In this execution plan, all three children of the UPDATE related-combine operation are stand-alone operations. The rules described earlier indicate that the execution plan carries out the operations as follows:

1. Operation 0 has one child (1). It can't be the first operation being executed.
2. Operation 1 has three children (2, 3, and 5), and operation 2 is the first of the three in ascending order. Therefore, the execution starts with operation 2.
3. Operation 2 scans the emp table and returns 14 rows to its parent (1).
4. The second and third child (3 and 5) might be executed several times (at most, as many times as the number of rows returned by operation 2). Since both these operations are stand-alone, and each has a child, their execution starts with the children (4 and 6).
5. For each distinct value in the deptno column returned by operation 2, operation 4 scans the emp table and applies the "E2"."DEPTNO"=:B1 filter predicate. In doing so over the three executions, it extracts 14 rows and passes them to its parent (3).
6. Operation 3 computes the average salary of the rows passed to it from operation 4 and returns the result to its parent (1).
7. Operation 6 scans the emp table, extracts 14 rows, and passes them to its parent (5). Note that this subquery is executed only once because it's not correlated to the main query.
8. Operation 5 computes the average commission of the rows passed to it from operation 6 and returns the result to its parent (1).
9. Operation 1 updates each row passed by operation 2 with the value returned by its other children (3 and 5) and passes to its parent (0) the number of rows that were updated. Note that even if the UPDATE statement modifies the 14 rows, A-Rows column shows 0 for this operation.
10. Operation 0 sends the number of rows that were updated to the caller.

## The CONNECT BY WITH FILTERING Operation

This operation is used to process hierarchical queries. It's characterized by two child operations. The first one is used to get the root(s) of the hierarchy, and the second one is executed once for each level in the hierarchy.

Here's a sample query and its execution plan (Figure 10-8 shows a graphical representation of its parent-child relationships). Note that the execution plan was generated on version 11.2 (the reason is explained later):

```
SELECT level, rpad('-',level-1,'-')||ename AS ename, prior ename AS manager
FROM emp
START WITH mgr IS NULL
CONNECT BY PRIOR empno = mgr
```

```
---------------------------------------------------------------------
| Id  | Operation                      | Name      | Starts | A-Rows |
---------------------------------------------------------------------
|   0 | SELECT STATEMENT               |           |      1 |     14 |
|*  1 |  CONNECT BY WITH FILTERING     |           |      1 |     14 |
|*  2 |   TABLE ACCESS FULL            | EMP       |      1 |      1 |
|   3 |   NESTED LOOPS                 |           |      4 |     13 |
|   4 |    CONNECT BY PUMP             |           |      4 |     14 |
|   5 |    TABLE ACCESS BY INDEX ROWID| EMP       |     14 |     13 |
|*  6 |     INDEX RANGE SCAN           | EMP_MGR_I |     14 |     13 |
---------------------------------------------------------------------

   1 - access("MGR"=PRIOR "EMPNO")
   2 - filter("MGR" IS NULL)
   6 - access("connect$_by$_pump$_002"."PRIOR empno"="MGR")
       filter("MGR" IS NOT NULL)
```

---

■ **Caution** The preceding query represents another situation where the `v$sql_plan` and `v$sql_plan_statistics_all` views give wrong information. In this case, `EXPLAIN PLAN` shows the correct predicates shown above, the wrongly displayed predicate is the one associated with operation 1:

```
1 - access("MGR"=PRIOR NULL)
```

Furthermore, the access predicate associated with operation 6 is wrong up to and including version 11.1.

---

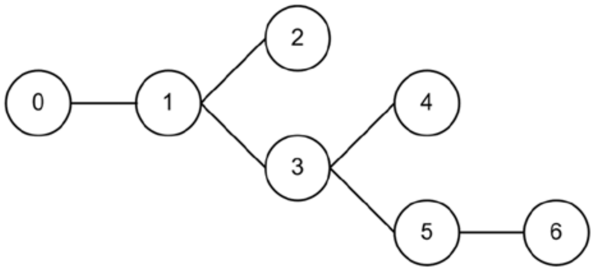


***Figure 10-8.*** *Parent-child relationships of the* `CONNECT BY WITH FILTERING` *operation*

In this execution plan, the first child of the `CONNECT BY WITH FILTERING` operation is a stand-alone operation. Instead, the second child is itself a related-combine operation. To read an execution plan in such a situation, you simply apply the rules recursively while descending the tree.

To help you understand the execution plan with a hierarchical query more easily, it's useful to look at the data returned by the query as well:

```
LEVEL ENAME    MANAGER
----- -------- -------
    1 KING
    2 -JONES   KING
    3 --SCOTT  JONES
    4 ---ADAMS SCOTT
    3 --FORD   JONES
    4 ---SMITH FORD
    2 -BLAKE   KING
    3 --ALLEN  BLAKE
    3 --WARD   BLAKE
    3 --MARTIN BLAKE
    3 --TURNER BLAKE
    3 --JAMES  BLAKE
    2 -CLARK   KING
    3 --MILLER CLARK
```

Applying the rules described earlier, you can see that the execution plan carries out the operations as follows:

1. Operation 0 has one child (1). It can't be the first operation being executed.
2. Operation 1 has two children (2 and 3), and operation 2 is the first of them in ascending order. Therefore, the execution starts with operation 2.
3. Operation 2 scans the `emp` table, applies the `"MGR" IS NULL` filter predicate, and returns the root of the hierarchy (KING) to its parent (1).
4. Operation 3 is the second child of operation 1. It's therefore executed for each level of the hierarchy—in this case, four times. Naturally, the rules previously discussed for the `NESTED LOOPS` operation apply for operation 3. The first child (4) is executed and, for each row it returns, the inner loop (composed of operations 5 and 6) is executed once. Notice, as expected, the match between the `A-Rows` column of operation 4 with the `Starts` column of operations 5 and 6.
5. For the first execution, operation 4 gets the root of the hierarchy through the `CONNECT BY PUMP` operation. In this case, there's a single row (KING) at level 1. With the value in the `mgr` column, operation 6 does a scan of the `emp_mgr_i` index by applying the `"MGR"=PRIOR "EMPNO"` access predicate (shown as `"connect$_by$_pump$_002"."PRIOR empno"="MGR"`), applies the filter predicate `"MGR" IS NOT NULL`, extracts the rowids, and returns them to its parent (5). Operation 5 accesses the `emp` table with the rowids and returns the rows to its parent (3).
6. For the second execution of operation 4, everything works the same as for the first execution. The only difference is that the data from level 2 (JONES, BLAKE, and CLARK) is passed to operation 4 for the processing (one by one, each row causing a start of operation 4).
7. For the third execution of operation 4, everything works like in the first one. The only difference is that level 3 data (SCOTT, FORD, ALLEN, WARD, MARTIN, TURNER, JAMES, and MILLER) is passed to operation 4 for the processing.
8. For the fourth and last execution of operation 4, everything works like in the first one. The only difference is that level 4 data (ADAMS and SMITH) is passed to operation 4 for the processing.

9. Operation 3 gets the rows passed from its children and returns them to its parent (1).
10. Operation 1 applies the "MGR" IS NOT NULL filter predicate. Then, the operation returns 14 rows to its parent (0).
11. Operation 0 sends the data to the caller.

The execution plan generated up to and including version 10.2.0.3 is slightly different. As can be seen in the following example, the CONNECT BY WITH FILTERING operation has a third child (operation 8). In this case, it wasn't executed, however. The value in the Starts column for operation 8 confirms this. Actually, the third child is executed only when the CONNECT BY WITH FILTERING operation uses temporary space. When that happens, performance might degrade considerably. This problem, which is fixed as of version 10.2.0.4, is known as bug 5065418:

```
---------------------------------------------------------------------
| Id  | Operation                      | Name      | Starts | A-Rows |
---------------------------------------------------------------------
|*  1 |  CONNECT BY WITH FILTERING     |           |      1 |     14 |
|*  2 |   TABLE ACCESS FULL            | EMP       |      1 |      1 |
|   3 |   NESTED LOOPS                 |           |      4 |     13 |
|   4 |    BUFFER SORT                 |           |      4 |     14 |
|   5 |     CONNECT BY PUMP            |           |      4 |     14 |
|   6 |    TABLE ACCESS BY INDEX ROWID| EMP       |     14 |     13 |
|*  7 |     INDEX RANGE SCAN           | EMP_MGR_I |     14 |     13 |
|   8 |   TABLE ACCESS FULL            | EMP       |      0 |      0 |
---------------------------------------------------------------------
```

## The UNION ALL (RECURSIVE WITH) Operation

The UNION ALL (RECURSIVE WITH) operation is available as of version 11.2. It was added to implement the recursive subquery factoring clause. Hence, it's used for hierarchical queries. Note that there are actually two operations that are related:

```
UNION ALL (RECURSIVE WITH) BREADTH FIRST
UNION ALL (RECURSIVE WITH) DEPTH FIRST
```

As their names suggest, the difference is due to the search clause that you can specify as either BREADTH FIRST BY or DEPTH FIRST BY.

Here's a sample query and its execution plan:

```
WITH
  e (xlevel, empno, ename, job, mgr, hiredate, sal, comm, deptno)
  AS (
    SELECT 1, empno, ename, job, mgr, hiredate, sal, comm, deptno
    FROM emp
    WHERE mgr IS NULL
    UNION ALL
    SELECT mgr.xlevel+1, emp.empno, emp.ename, emp.job, emp.mgr, emp.hiredate, emp.sal,
    FROM emp, e mgr
    WHERE emp.mgr = mgr.empno
  )
SELECT *
FROM e
```

```
---------------------------------------------------------------------------------
| Id  | Operation                                  | Name      | Starts | A-Rows |
---------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                           |           |      1 |     14 |
|   1 |  VIEW                                      |           |      1 |     14 |
|   2 |   UNION ALL (RECURSIVE WITH) BREADTH FIRST|           |      1 |     14 |
|*  3 |    TABLE ACCESS FULL                       | EMP       |      1 |      1 |
|   4 |    NESTED LOOPS                            |           |      4 |     13 |
|   5 |     NESTED LOOPS                           |           |      4 |     13 |
|   6 |      RECURSIVE WITH PUMP                   |           |      4 |     14 |
|*  7 |      INDEX RANGE SCAN                      | EMP_MGR_I |     14 |     13 |
|   8 |     TABLE ACCESS BY INDEX ROWID            | EMP       |     13 |     13 |
---------------------------------------------------------------------------------

   3 - filter("MGR" IS NULL)
   7 - access("EMP"."MGR"="MGR"."EMPNO")
       filter("EMP"."MGR" IS NOT NULL)
```

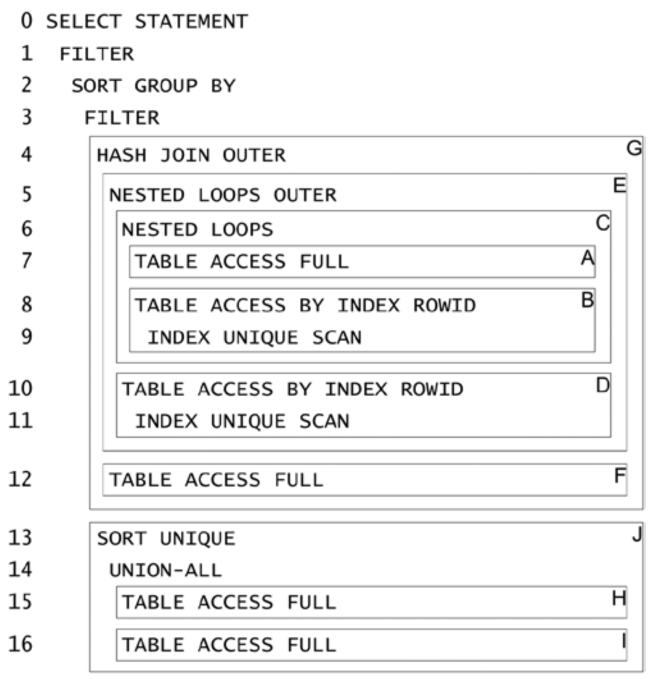
Reading an execution plan containing the UNION ALL (RECURSIVE WITH) operation is the same as reading one containing the CONNECT BY WITH FILTERING operation. As a matter of fact, the purpose of both operations is basically the same. Just notice that the PUMP operation used in the execution plan also differs. While in the former it's called RECURSIVE WITH PUMP, in the latter it's called CONNECT BY PUMP. In any case, the difference, for the purpose of reading the execution plan, doesn't matter.

## Divide and Conquer

In the previous sections, you saw how to read the three types of operations execution plans are composed of. All the execution plans you've seen so far were quite easy (short). More often than not, though, you have to deal with complex (long) execution plans. That's not because most SQL statements are complex but because it's likely that simple SQL statements are correctly optimized by the query optimizer, and as a result, you never have to question their performance.

The essential thing to recognize is that reading long execution plans is no different from reading short ones. All you need is to methodically apply the rules provided in the previous sections. With them, it doesn't matter how many lines an execution plan has. You simply proceed in the same way.

To show you how to proceed with an execution plan that's longer than a few lines, let's take a look at the operations carried out by the execution plan shown in Figure 10-9 (Figure 10-10 shows a graphical representation of its parent-child relationships). I'm not providing the SQL statement used to generate it intentionally. For our purposes, you're simply not interested in the SQL statement. The execution plan, on the other hand, is the key.



```
 0 SELECT STATEMENT
 1  FILTER
 2   SORT GROUP BY
 3    FILTER
 4     HASH JOIN OUTER                              G
 5      NESTED LOOPS OUTER                          E
 6       NESTED LOOPS                               C
 7        TABLE ACCESS FULL                         A
 8        TABLE ACCESS BY INDEX ROWID               B
 9         INDEX UNIQUE SCAN
10       TABLE ACCESS BY INDEX ROWID                D
11        INDEX UNIQUE SCAN
12      TABLE ACCESS FULL                           F
13     SORT UNIQUE                                  J
14      UNION-ALL
15       TABLE ACCESS FULL                          H
16       TABLE ACCESS FULL                          I
```

***Figure 10-9.*** *An execution plan decomposed in blocks. The numbers on the left identify the operations. The letters on the right identify the blocks*

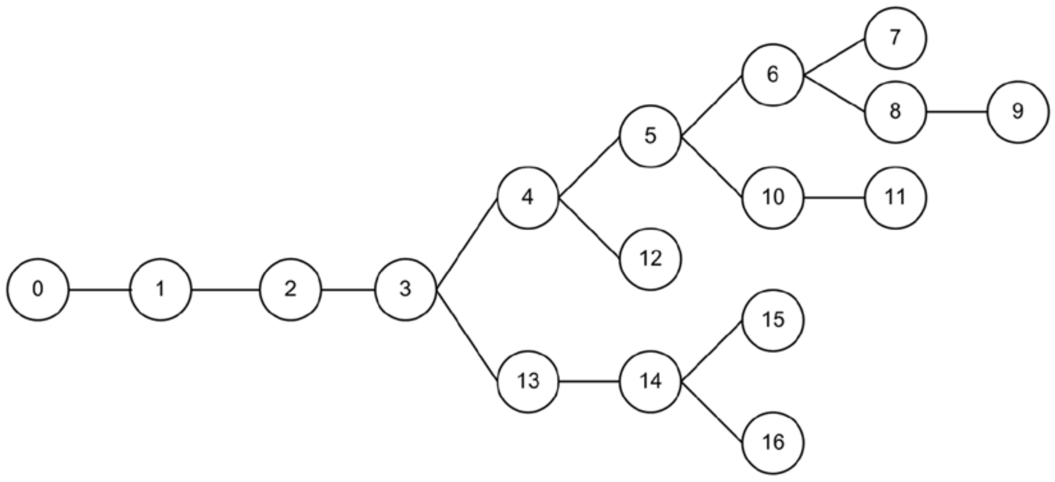


***Figure 10-10.*** *Parent-child relationships of the execution plan shown in Figure 10-9*

Initially, it's necessary to both decompose the execution plan into basic blocks and recognize the order of execution. To do so, you carry out the following steps. To read an execution plan at first, you have to identify the combine operations (both related and unrelated) it's composed of. In other words, you identify each operation having more than one child. In the example in Figure 10-9, the combine operations are the following: 3, 4, 5, 6, and 14. Then, for each child operation of each combine operation, you define a block. Because in Figure 10-9 you have five combine operations, and each of them has two children, you have a total of ten blocks. For example, for operation 3, the first child consists of the lines from 4 to 12 (block G), and the second child consists of the lines from 13 to 16 (block J). Note that in Figure 10-9, each block is delimited by a frame. Finally, you need to find out in what order the blocks are executed. To see how this is done, let's walk through the execution plan shown in Figure 10-9 and apply the rules discussed previously:

1. Operation 0 is a stand-alone operation, and its child (1) is executed before it.
2. Operation 1 is a stand-alone operation, and its child (2) is executed before it.
3. Operation 2 is a stand-alone operation, and its child (3) is executed before it.
4. Operation 3 is a related-combine operation, and its children are executed before it. Since the first child block (G) is executed before the second child block (J), let's continue with the first operation (4) of the first child block (G).
5. Operation 4 is an unrelated-combine operation, and its children are executed before it. Since the first child block (E) is executed before the second child block (F), let's continue with the first operation (5) of the first child block (E).
6. Operation 5 is a related-combine operation, and its children are executed before it. Since the first child block (C) is executed before the second child block (D), let's continue with the first operation (6) of the first child block (C).
7. Operation 6 is a related-combine operation, and its children are executed before it. Since the first child block (A) is executed before the second child block (B), let's continue with the first operation (7) of the first child block (A).
8. Operation 7 is a stand-alone operation and has no children. This means that you have finally found the first operation to be executed (hence it's in block A). The operation scans a table and returns the rows to its parent operation (6).
9. Block B is executed for each row returned by block A. In this block, operation 9 scans an index at first, and operation 8 accesses a table with the returned rowids and finally returns the rows to its parent operation (6).
10. Operation 6 performs the join between the rows returned by blocks A and B and then returns the result to its parent operation (5).
11. Block D is executed for each row returned by block C. In other words, it's executed for each row returned by operation 6 to its parent operation (5). In this block, operation 11 scans an index initially. Then, operation 10 accesses a table with the returned rowids and returns the rows to its parent operation (5).
12. Operation 5 performs the join between the rows returned by the blocks C and D and then returns the result to its parent operation (4).
13. Operation 12 (block F) is executed only once. It scans a table and returns the result to its parent operation (4).
14. Operation 4 performs the join between the rows returned by the blocks E and F and then returns the result to its parent operation (3).

15. Block J is basically executed for each row returned by block G. In other words, it's executed for each row returned by operation 4 to its parent operation (3). In this block, operation 15 scans a table at first and returns the rows to its parent operation (14). Then, operation 16 scans a table and returns the rows to its parent operation (14). After that, operation 14 puts the rows returned by its children together and returns the result to its parent operation (13). Finally, operation 13 removes some duplicate rows. Note that this block doesn't return data to its parent. In fact, the parent is a `FILTER` operation, and the second child is used to apply a restriction only.
16. Once operation 3 has applied the filter with the block J, it returns the result to its parent operation (2).
17. Operation 2 performs a `GROUP BY` and returns the result to its parent operation (1).
18. Operation 1 applies a filter and returns the result to the caller.

In summary, note that blocks are executed according to their identifier (from A up to J). Some blocks (A, C, E, F, and G) are executed once at most, and others (B, D, H, I, and J) might be executed several times (or never), depending on how many rows are returned by the operations driving them.

## Special Cases

The rules described in the previous sections apply to almost all execution plans. Nevertheless, there are some special cases. Usually you can find out what's going on by looking at the operations, the predicates they apply, the tables on which they're executed, and their runtime behavior (especially the `Starts` and `A-Rows` columns). The following subsections cover three examples that are taken from among many possible cases. Note that all examples provided in the following subsections are excerpts of the output generated by the `special_cases.sql` script.

### Subquery in the SELECT Clause

This example shows what an execution plan for a query containing a subquery in the `SELECT` clause looks like. The query and its execution plan are the following:

```
SELECT ename, (SELECT dname
               FROM dept
               WHERE dept.deptno = emp.deptno)
FROM emp

-----------------------------------------------------------------
| Id  | Operation                    | Name    | Starts | A-Rows |
-----------------------------------------------------------------
|   0 | SELECT STATEMENT             |         |      1 |     14 |
|   1 |  TABLE ACCESS BY INDEX ROWID| DEPT    |      3 |      3 |
|*  2 |   INDEX UNIQUE SCAN         | DEPT_PK |      3 |      3 |
|   3 |  TABLE ACCESS FULL          | EMP     |      1 |     14 |
-----------------------------------------------------------------

   2 - access("DEPT"."DEPTNO"=:B1)
```

The strange thing in this execution plan is that operation 0 has several children. If you look carefully at the Starts column, you'll notice that although operations 1 and 2 are executed three times, operation 3 is executed only once. Also notice that operations 1 and 2, since they reference the dept table, implement the subquery. This unusual execution plan carries out the operations as follows:

1. Operation 3, which is the first one being executed, scans the emp table and returns all rows to its parent (0).
2. For each row returned by the operation 3, the subquery should be executed once. However, also in this case the SQL engine caches the results, and therefore the subquery is executed only once for each distinct value in the deptno column.
3. To execute the subquery, operation 2 scans the dept_pk index by applying the "DEPT"."DEPTNO"=:B1 access predicate, extracts the rowids, and returns them to its parent (1). The bind variable (B1) is used to pass the value to be checked to the subquery. Then operation 1 accesses the dept table with those rowids and passes the data to its parent (0).
4. Operation 0 sends the data to the caller.

## Subquery in the WHERE Clause #1

This example shows a particular execution plan related to a query containing a subquery in the WHERE clause. The query and its execution plan are the following:

```
SELECT deptno
FROM dept
WHERE deptno NOT IN (SELECT deptno FROM emp)

--------------------------------------------------------
| Id  | Operation          | Name    | Starts | A-Rows |
--------------------------------------------------------
|   0 | SELECT STATEMENT   |         |      1 |      1 |
|*  1 |  INDEX FULL SCAN   | DEPT_PK |      1 |      1 |
|*  2 |   TABLE ACCESS FULL| EMP     |      4 |      3 |
--------------------------------------------------------

   1 - filter( NOT EXISTS (SELECT 0 FROM "EMP" "EMP" WHERE
              LNNVL("DEPTNO"<>:B1)))
   2 - filter(LNNVL("DEPTNO"<>:B1))
```

■ **Caution** This query is another case where the v$sql_plan and v$sql_plan_statistics_all views show wrong information. In this case, EXPLAIN PLAN shows the correct predicates shown above. The wrongly displayed predicate is the one associated with operation 1:

```
1- filter( IS NULL)
```

At first sight, this execution plan is composed from two stand-alone operations. If you carefully look at the `Starts` column, you'll notice something strange. In fact, although the parent (1) is executed only once, the child (2) is executed four times. Actually, this execution plan carries out the operations as follows:

1. Operation 1, which is the first being executed, scans the `dept_pk` index. For each value in the `deptno` column, it executes operation 2. As shown by the filter predicates, operation 2 applies the `NOT EXISTS (SELECT 0 FROM "EMP" "EMP" WHERE LNNVL("DEPTNO"<>:B1))` subquery. Notice that the query optimizer transformed the `NOT IN` into a `NOT EXISTS`. The bind variable (`B1`) is used to pass the value to be checked to the subquery.
2. Operation 2 scans the `emp` table, applies the `LNNVL("DEPTNO"<>:B1)` filter predicate, and returns the data to its parent (1).
3. For each row fulfilling the filter predicates operation 1 passes the data to its parent (0).
4. Operation 0 sends the data to the caller.

Another execution plan that can be generated by the query optimizer for the very same query is the following one. However, because it uses a feature (`NULL`-aware anti-join) that's only available as of version 11.1, don't expect to see this type of execution plan in version 10.2. (In my opinion, this execution plan is far more readable than the previous one).

```
---------------------------------------------------------
| Id  | Operation          | Name    | Starts | A-Rows |
---------------------------------------------------------
|   0 | SELECT STATEMENT   |         |      1 |      1 |
|*  1 |  HASH JOIN ANTI NA |         |      1 |      1 |
|   2 |   INDEX FULL SCAN  | DEPT_PK |      1 |      4 |
|   3 |   TABLE ACCESS FULL| EMP     |      1 |     14 |
---------------------------------------------------------

   1 - access("DEPTNO"="DEPTNO")
```

## Subquery in the WHERE Clause #2

This example is an extension of the previous one. It also involves a subquery in the `WHERE` clause. I show it, because I want to call attention to the fact that the query optimizer is able to generate execution plans like the one discussed in the previous section, even when the code implementing the subquery is more complex than a plain lookup. The query and its execution plan are as follows:

```
SELECT *
FROM t1
WHERE n1 = 8 AND n2 IN (SELECT t2.n1
                        FROM t2, t3
                        WHERE t2.id = t3.id AND t3.n1 = 4);

---------------------------------------------------------------
| Id  | Operation                   | Name | Starts | A-Rows |
---------------------------------------------------------------
|   0 | SELECT STATEMENT            |      |      1 |      7 |
|   1 |  TABLE ACCESS BY INDEX ROWID| T1   |      1 |      7 |
|*  2 |   INDEX RANGE SCAN          | I1   |      1 |      7 |
|*  3 |    HASH JOIN                |      |     13 |      1 |
|*  4 |     TABLE ACCESS FULL       | T3   |     13 |   1183 |
|*  5 |     TABLE ACCESS FULL       | T2   |     13 |    910 |
---------------------------------------------------------------
```

```
   2 - access("N1"=8)
       filter( EXISTS (SELECT /*+ PUSH_SUBQ LEADING ("T3" "T2") FULL ("T3")
              USE_HASH ("T2") FULL ("T2") */ 0 FROM "T3" "T3","T2" "T2" WHERE
              "T2"."ID"="T3"."ID" AND "T2"."N1"=:B1 AND "T3"."N1"=4))
   3 - access("T2"."ID"="T3"."ID")
   4 - filter("T3"."N1"=4)
   5 - filter("T2"."N1"=:B1)
```

---

■ **Caution** This query is another case where the `v$sql_plan` and `v$sql_plan_statistics_all` views show wrong information. In this case, `EXPLAIN PLAN` shows the correct predicates shown above. The wrongly displayed predicate is the one associated with the filter of operation 2:

```
2 - access("N1"=8)
    filter( IS NOT NULL)
```

---

Also in this case, if you carefully look at the `Starts` column, you'll notice something strange. The operations up to 2 are executed only once, and the operations from 3 to 5 are executed 13 times. This execution plan carries out the operations as follows:

1. Operation 2, which is the first one being executed, applies the `"N1"=8` access predicate by scanning the `i1` index. It extracts from the index, for the key fulfilling the access predicate, not only the rowid but also the value of the `n2` column. For each distinct value of the `n2` column, the subquery (operations 3 to 5) is executed once. This is done to apply the filter predicate. Notice that the query optimizer transformed the `IN` into an `EXISTS`. The join carried out by the subquery is implemented with a hash join, an unrelated-combine operation.
2. Operation 4, the first child of the hash join, reads the `t3` table with a full scan and passes the rows that fulfill the `"T3"."N1"=4` filter predicate to its parent (3).
3. Operation 5, the second child of the hash join, reads the `t2` table with a full scan and passes the rows fulfilling the `"T2"."N1"=:B1` filter predicate to its parent (3). The bind variable (`B1`) is used to pass the value to be checked to the subquery.
4. Operation 3 joins the two sets of rows passed by operations 3 and 4. It passes the data to its parent (2) when at least one row is found.
5. Operation 2, for each row fulfilling the condition implemented by the subquery, passes one rowid to its parent (1).
6. Operation 1 uses the rowids it receives from its child (2) to access the `t1` table and extract its columns. It passes the data to its parent (0).
7. Operation 0 sends the data to the caller.

## Adaptive Execution Plans

Object statistics don't always provide all the information the query optimizer needs to find an optimal execution plan. To improve this situation, the query optimizer, during the parse phase, can take advantage of dynamic sampling to get additional insights into the data to be processed. (Dynamic sampling is described in Chapter 9). In addition, as of version 12.1, the query optimizer is able to postpone some decisions until the execution phase. The idea is to leverage information that can be collected while executing part of an execution plan to determine how another part should be carried out. For this purpose, the query optimizer adds what are called *subplans*. Also added are operations that are responsible for determining which subplans to activate.

---

■ **Note** Adaptive execution plans are available only in Enterprise Edition.

---

From version 12.1 onward, the query optimizer can use adaptive execution plans in the following situations:

- To switch the join method from a nested loops join to a hash join, and vice versa.
- To switch the distribution method from hash to broadcast for SQL statements executed in parallel.

The case related to parallel processing is covered in Chapter 15. The following example illustrates how switching the join method works. The example shows an excerpt of the output generated by the `adaptive_plan.sql` script. The query is a simple join between two tables. It's carried out with a regular nested loops join:

```
SQL> EXPLAIN PLAN FOR
  2  SELECT *
  3  FROM t1, t2
  4  WHERE t1.id = t2.id
  5  AND t1.n = 666;

SQL> SELECT * FROM table(dbms_xplan.display(format=>'basic +predicate +note'));

PLAN_TABLE_OUTPUT
----------------------------------------------
Plan hash value: 1837274416

----------------------------------------------
| Id  | Operation                   | Name  |
----------------------------------------------
|   0 | SELECT STATEMENT            |       |
|   1 |  NESTED LOOPS               |       |
|   2 |   NESTED LOOPS              |       |
|*  3 |    TABLE ACCESS FULL        | T1    |
|*  4 |    INDEX UNIQUE SCAN        | T2_PK |
|   5 |   TABLE ACCESS BY INDEX ROWID| T2    |
----------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   3 - filter("T1"."N"=666)
   4 - access("T1"."ID"="T2"."ID")

Note
-----
   - this is an adaptive plan
```

Notice that the Note section in the previous excerpt points out that the execution plan is adaptive. In the execution plan itself, however, there's no sign of anything peculiar. The fact is that, by default, the display function of the dbms_xplan package shows the *default execution plan* only. Simply put, this is the execution plan that the query optimizer would choose without considering adaptive execution plans. If you want to see the *full execution plan* containing the subplans, you have to specify the adaptive modifier when using the dbms_xplan package. In this case, three additional operations are shown in the execution plan:

```
SQL> SELECT * FROM table(dbms_xplan.display(format=>' basic +predicate +note +adaptive'));

PLAN_TABLE_OUTPUT
-------------------------------------------------
Plan hash value: 1837274416

-------------------------------------------------
|   Id  | Operation                     | Name  |
-------------------------------------------------
|     0 | SELECT STATEMENT              |       |
|- *  1 |  HASH JOIN                    |       |
|     2 |   NESTED LOOPS                |       |
|     3 |    NESTED LOOPS               |       |
|-    4 |     STATISTICS COLLECTOR      |       |
|  *  5 |      TABLE ACCESS FULL        | T1    |
|  *  6 |     INDEX UNIQUE SCAN         | T2_PK |
|     7 |    TABLE ACCESS BY INDEX ROWID| T2    |
|-    8 |   TABLE ACCESS FULL           | T2    |
-------------------------------------------------

   1 - access("T1"."ID"="T2"."ID")
   5 - filter("T1"."N"=666)
   6 - access("T1"."ID"="T2"."ID")

Note
-----
   - this is an adaptive plan (rows marked '-' are inactive)
```

Such an execution plan isn't very readable because, in fact, it contains two different execution plans. First, the default execution plan based on a nested loops join:

```
----------------------------------------------
| Id  | Operation                     | Name  |
----------------------------------------------
|   0 | SELECT STATEMENT              |       |
|   1 |  NESTED LOOPS                 |       |
|   2 |   NESTED LOOPS                |       |
|   3 |    TABLE ACCESS FULL          | T1    |
|   4 |    INDEX UNIQUE SCAN          | T2_PK |
|   5 |   TABLE ACCESS BY INDEX ROWID| T2    |
----------------------------------------------
```

Next, an alternative execution plan based on a hash join:

```
-----------------------------------
| Id  | Operation          | Name |
-----------------------------------
|   0 | SELECT STATEMENT   |      |
|   1 |  HASH JOIN         |      |
|   2 |   TABLE ACCESS FULL| T1   |
|   3 |   TABLE ACCESS FULL| T2   |
-----------------------------------
```

Basically, the first execution plan is better than the second one when the scan of the `t1` table returns a small number of rows. Hence, to decide which execution plan should be used, the query optimizer estimates the maximum number of rows (called *inflection point*) that can be efficiently processed with the nested loops join. To determine, during the execution phase, which execution plan has to be used, the `STATISTICS COLLECTOR` operation buffers and counts the number of rows the scan of the `t1` table returns. Then—and only if the number is lower than the inflection point—is the nested loops join executed. Otherwise, the hash join is executed. The execution plan that is actually used is called *final execution plan*. Once the final execution plan has been determined, the `STATISTICS COLLECTOR` operation is disabled and, therefore, no further buffering takes place. In addition, the operations related to the inefficient join method are also disabled.

■ **Note** Be aware that the execution plan switch is only performed during the first execution of a child cursor. All successive executions use the final execution plan.

The `v$sql` dynamic performance view provides a new column to help you know whether, for a specific child cursor, the final execution plan was already selected. That column is `is_resolved_adaptive_plan`. It's set to one of the following values:

- `NULL` means that the execution plan associated to the cursor isn't adaptive.
- `N` means that the final execution plan hasn't been determined. This value can be observed only until the final execution plan has been determined.
- `Y` means that the final execution plan was determined.

Two initialization parameters control adaptive execution plans:

- `optimizer_adaptive_features` fully enables or disables the feature. Adaptive execution plans are disabled when the parameter is set to `FALSE`. The default value is `TRUE`.
- `optimizer_adaptive_reporting_only` enables or disables adaptive execution plans in reporting mode. This mode can be useful to assess whether an execution plan would change because of adaptive execution plans. When set to `TRUE`, adaptive execution plans are generated, and the SQL engine checks the inflection point, but the SQL engine uses only the default execution plan. Then, with the reporting feature shown in the following example, you can check which execution plan would be used if the feature were to be fully enabled. The default value is `FALSE`:

```
SQL> ALTER SESSION SET optimizer_adaptive_reporting_only = TRUE;
SQL> SELECT *
  2  FROM t1, t2
  3  WHERE t1.id = t2.id
  4  AND t1.n = 666;

SQL> SELECT *
  2  FROM table(dbms_xplan.display_cursor(format=>'basic +predicate +note +adaptive +report'));

EXPLAINED SQL STATEMENT:
------------------------
SELECT * FROM t1, t2 WHERE t1.id = t2.id AND t1.n = 666

Plan hash value: 1837274416

-------------------------------------------------
|   Id  | Operation                     | Name  |
-------------------------------------------------
|     0 | SELECT STATEMENT              |       |
|- *  1 |  HASH JOIN                    |       |
|     2 |   NESTED LOOPS                |       |
|     3 |    NESTED LOOPS               |       |
|-    4 |     STATISTICS COLLECTOR      |       |
|  *  5 |      TABLE ACCESS FULL        | T1    |
|  *  6 |     INDEX UNIQUE SCAN         | T2_PK |
|     7 |    TABLE ACCESS BY INDEX ROWID| T2    |
|-    8 |   TABLE ACCESS FULL           | T2    |
-------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   1 - access("T1"."ID"="T2"."ID")
   5 - filter("T1"."N"=666)
   6 - access("T1"."ID"="T2"."ID")

Note
-----
   - this is an adaptive plan (rows marked '-' are inactive)

Adaptive plan:
-------------
This cursor has an adaptive plan, but adaptive plans are enabled for
reporting mode only.  The plan that would be executed if adaptive plans
were enabled is displayed below.

Plan hash value: 1837274416
```

```
-----------------------------------
| Id  | Operation         | Name |
-----------------------------------
|   0 | SELECT STATEMENT   |      |
|*  1 |  HASH JOIN         |      |
|*  2 |   TABLE ACCESS FULL| T1   |
|   3 |   TABLE ACCESS FULL| T2   |
-----------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   1 - access("T1"."ID"="T2"."ID")
   2 - filter("T1"."N"=666)

Note
-----
   - this is an adaptive plan
```

## Recognizing Inefficient Execution Plans

The sad truth is, the only way to be sure that an execution plan isn't the most efficient one is to find another one that's better. Nevertheless, simple checks might reveal clues that suggest an inefficient execution plan. The following sections describe two checks that I use for that purpose. In Chapter 13, another check used for assessing the efficiency of access paths is introduced.

### Wrong Estimations

The idea behind this check is very simple. The query optimizer computes costs to decide which access paths, join orders, and join methods should be used to get an efficient execution plan. If the computation of the cost is wrong, it's likely that the query optimizer picks out a suboptimal execution plan. In other words, wrong estimations easily lead to making a mistake in the choice of an execution plan.

Judging the cost directly of a SQL statement itself isn't feasible in practice. It's much easier to check other estimations performed by the query optimizer, which the computation of the cost is based on: the estimated number of rows returned by an operation (the cardinality). Checking the estimated cardinality is quite easy because you can, with the `display_cursor` function in the `dbms_xplan` package, for example, directly compare it with the actual cardinality. As you've just seen, only if the two cardinalities are close did the query optimizer do a good job. One of the central characteristics of this method is that no information about the SQL statement or the database structure is necessary to judge the quality of the execution plan. You simply concentrate on comparing the estimations with the actual data.

Let me illustrate this concept with an example. The following excerpt of the output produced by the `wrong_estimations.sql` script shows an execution plan with its estimated (`E-Rows`) and actual (`A-Rows`) cardinalities. As you can see, the estimation of operation 4 (and consequently of operations 2 and 3) is completely wrong. The query optimizer estimated, for operation 4, a return of only 32 rows instead of 80,016. To make things worse, operations 2 and 3 are related-combine operations. This means that operations 6 and 7, instead of being executed only 32 times as estimated, are in fact executed 80,016 and 75,808 times, respectively. This is confirmed by the values in the `Starts` column. It's important to note that the estimations for operations 6 and 7 are correct. In fact, before making the comparison, the actual cardinality (`A-Rows`) must be divided by the number of executions (`Starts`):

```
SELECT count(t2.col2)
FROM t1 JOIN t2 USING (id)
WHERE t1.col1 = 666
```

```
----------------------------------------------------------------------------
| Id  | Operation                      | Name    | Starts | E-Rows | A-Rows |
----------------------------------------------------------------------------
|   0 | SELECT STATEMENT               |         |      1 |        |      1 |
|   1 |  SORT AGGREGATE                |         |      1 |      1 |      1 |
|   2 |   NESTED LOOPS                 |         |      1 |        |  75808 |
|   3 |    NESTED LOOPS                |         |      1 |     32 |  75808 |
|   4 |     TABLE ACCESS BY INDEX ROWID| T1      |      1 |     32 |  80016 |
|*  5 |      INDEX RANGE SCAN          | T1_COL1 |      1 |     32 |  80016 |
|*  6 |     INDEX UNIQUE SCAN          | T2_PK   |  80016 |      1 |  75808 |
|   7 |    TABLE ACCESS BY INDEX ROWID | T2      |  75808 |      1 |  75808 |
----------------------------------------------------------------------------

   5 - access("T1"."COL1"=666)
   6 - access("T1"."ID"="T2"."ID")
```

To understand the problem, you have to carefully analyze why the query optimizer can't compute good estimations. The cardinality is computed by multiplying the selectivity by the number of rows in the table. Therefore, if the cardinality is wrong, the problem can have only three possible causes: a wrong selectivity, a wrong number of rows, or a bug in the query optimizer.

In this case, our analysis should start by looking at the estimation performed for operation 5. In other words, the estimation related to the "T1"."COL1"=666 predicate. Because the query optimizer bases its estimation on object statistics, let's see whether they describe the current data. With the following query, you're able to get the object statistics for the t1_col1 index used for operation 5. At the same time, it's possible to compute the average number of rows per key. This is basically the value used for the query optimizer when no histogram is available:

```
SQL> SELECT num_rows, distinct_keys, num_rows/distinct_keys AS avg_rows_per_key
  2  FROM user_indexes
  3  WHERE index_name = 'T1_COL1';

NUM_ROWS DISTINCT_KEYS AVG_ROWS_PER_KEY
-------- ------------- ----------------
  160000          5000               32
```

It's useful to notice that in this case, the average number of rows, 32, is the same as the estimated cardinality in the previous execution plan. To check whether these object statistics are good, you have to compare them with the actual data. So, let's execute the following query on the t1 table. As you can see, the query not only computes the object statistics of the previous query but also counts the number of rows for which the col1 column is different from 666:

```
SQL> SELECT count(*) AS num_rows, count(DISTINCT col1) AS distinct_keys,
  2         count(nullif(col1,666)) AS rows_per_key_666
  3  FROM t1;

NUM_ROWS DISTINCT_KEYS ROWS_PER_KEY_666
-------- ------------- ----------------
  160000          5000            79984
```

From the output, you can confirm that not only are the object statistics correct but that the data is also strongly skewed. As a result, a histogram is absolutely essential for correct estimations. With the following query, you can confirm that no histogram exists in this case:

```
SQL> SELECT histogram, num_buckets
  2  FROM user_tab_col_statistics
  3  WHERE table_name = 'T1' AND column_name = 'COL1';

HISTOGRAM NUM_BUCKETS
--------- -----------
NONE                1
```

After gathering the missing histogram, the query optimizer managed to correctly estimate cardinalities and, as a result, considered another execution plan to be the most efficient:

```
---------------------------------------------------------------
| Id  | Operation          | Name | Starts | E-Rows | A-Rows |
---------------------------------------------------------------
|   0 | SELECT STATEMENT    |      |      1 |        |      1 |
|   1 |  SORT AGGREGATE     |      |      1 |      1 |      1 |
|*  2 |   HASH JOIN         |      |      1 |  80000 |  75808 |
|*  3 |    TABLE ACCESS FULL| T1   |      1 |  80000 |  80016 |
|   4 |    TABLE ACCESS FULL| T2   |      1 |   151K|   151K|
---------------------------------------------------------------

   2 - access("T1"."ID"="T2"."ID")
   3 - filter("T1"."COL1"=666)
```

Note that when the `wrong_estimations.sql` script is executed in version 12.1 without disabling adaptive execution plans, the query optimizer generates an adaptive execution plan and, as a result, automatically detects at runtime that, for this particular query, a hash join is better than a nested loop.

## Restriction Not Recognized

I must warn you that the check presented in the previous section is superior to this one. I usually use this second check only in addition to the first one. The idea of this check is to verify whether the query optimizer correctly recognized the restriction in the SQL statement and, as a result, applied it as soon as possible. In other words, you check whether the execution plan leads to unnecessary processing.

Let me illustrate this concept with an example based on the following excerpt of the output produced by the `restriction_not_recognized.sql` script. From it, you can see that the query optimizer decided to start joining the `t1` and `t2` tables. This first join returned a result set of 40,000 rows. Later, the result set was joined to the `t3` table. A result set of only 100 rows was generated, even though the operation reading the `t3` table returned 80,000 rows. This simply means that the query optimizer didn't recognize the restriction and applied it too late when a lot of processing was already being performed. Estimating join cardinalities is, *en passant*, one of the most difficult tasks the query optimizer has to perform:

```
SELECT count(t1.pad), count(t2.pad), count(t3.pad)
FROM t1, t2, t3
WHERE t1.id = t2.t1_id AND t2.id = t3.t2_id
```

```
----------------------------------------------------------------
| Id  | Operation           | Name | Starts | E-Rows | A-Rows |
----------------------------------------------------------------
|   0 | SELECT STATEMENT     |      |      1 |        |      1 |
|   1 |  SORT AGGREGATE      |      |      1 |      1 |      1 |
|*  2 |   HASH JOIN          |      |      1 |  79800 |    100 |
|*  3 |    HASH JOIN         |      |      1 |  40000 |  40000 |
|   4 |     TABLE ACCESS FULL| T1   |      1 |  20000 |  20000 |
|   5 |     TABLE ACCESS FULL| T2   |      1 |  40000 |  40000 |
|   6 |    TABLE ACCESS FULL | T3   |      1 |  80000 |  80000 |
----------------------------------------------------------------

   2 - access("T2"."ID"="T3"."T2_ID")
   3 - access("T1"."ID"="T2"."T1_ID")
```

When you encounter such problems, there's little you can do. In fact, there are no object statistics describing the relationship between two tables. One possible way to correct a situation of this type is to use a SQL profile. Applying one in this case would give you the following execution plan. (I cover what a SQL profile is, and how it works, in Chapter 11.) For the moment, it's just important to realize that a solution exists. Notice that not only has the order of the join changed (t2 ➤ t3 ➤ t1), but the access to the `t1` table is also different:

```
--------------------------------------------------------------------------
| Id  | Operation                     | Name  | Starts | E-Rows | A-Rows |
--------------------------------------------------------------------------
|   0 | SELECT STATEMENT              |       |      1 |        |      1 |
|   1 |  SORT AGGREGATE               |       |      1 |      1 |      1 |
|   2 |   NESTED LOOPS                |       |      1 |        |    100 |
|   3 |    NESTED LOOPS               |       |      1 |    100 |    100 |
|*  4 |     HASH JOIN                 |       |      1 |    100 |    100 |
|   5 |      TABLE ACCESS FULL        | T2    |      1 |  40000 |  40000 |
|   6 |      TABLE ACCESS FULL        | T3    |      1 |  80000 |  80000 |
|*  7 |     INDEX UNIQUE SCAN         | T1_PK |    100 |      1 |    100 |
|   8 |    TABLE ACCESS BY INDEX ROWID| T1    |    100 |      1 |    100 |
--------------------------------------------------------------------------

   4 - access("T2"."ID"="T3"."T2_ID")
   7 - access("T1"."ID"="T2"."T1_ID")
```

## On to Part 4

This chapter describes how to obtain execution plans through the `EXPLAIN PLAN` statement, dynamic performance views, Automatic Workload Repository, Statspack, and some tracing facilities. As discussed for the first four techniques, the `dbms_xplan` package is the tool of choice for extracting and formatting execution plans. With it, you're able to get all the information you need simply, enabling you to understand execution plans. Some rules for interpreting execution plans and for recognizing whether they're efficient are discussed as well.

Clearly, inefficient execution plans that cause performance problems should be optimized. The first chapter of Part 4 starts covering this topic by describing the SQL optimization techniques available for that purpose. Note that there are several techniques, since each of them can be applied in specific circumstances or just for the optimization of particular problems.

PART IV

# Optimization

*Engineering isn't about perfect solutions; it's about doing the best you can with limited resources.*

— Randy Pausch, *The Last Lecture*. 2008.

Only once you have identified the root cause of a performance problem should you try to solve it. Regardless of the problem you are facing, the essential goal to achieve is reducing—or, even better, eliminating—the time spent by the most time-consuming operation. Note that a single operation may be composed of many actions that are executed one by one. For example, many fetches are necessary in order to fully process a query returning many rows.

Chapter 11 describes the available SQL optimization techniques, and goes on to explain how to choose between them. Chapter 12 describes how parsing works, how to identify parsing problems, and how to minimize parsing's impact without jeopardizing performance. Chapter 13 describes how to take advantage of available access structures in order to efficiently access data stored in a single table. Chapter 14 goes beyond accessing a single table by describing how to join data from several tables together. Chapter 15 deals with parallel processing and the techniques used for speeding up stream inserts, and for minimizing the interactions between components. Finally, Chapter 16 describes how physical storage parameters have an observable impact on performance. Simply put, the aim of the chapters in this part is to show how to improve the response time of operations interacting with the SQL engine by taking advantage of the many features provided by Oracle Database for that purpose.

CHAPTER 11

# SQL Optimization Techniques

Whenever the query optimizer is unable to automatically generate an efficient execution plan, some manual optimization is required. For that purpose, Oracle Database provides several techniques, summarized in Table 11-1. The goal of this chapter isn't only to describe these techniques in detail but to explain what each technique can do for you and in which situations you can take advantage of them. To choose one of them, it's essential to ask yourself three basic questions:

- Is the SQL statement known and static?
- Should the measures to be taken have an impact on a single SQL statement or on all SQL statements executed by a single session (or even on the whole system)?
- Is it possible to change the SQL statement?

***Table 11-1.*** *SQL Optimization Techniques and Their Impacts*

| Technique | System | Session | SQL Statement | Availability |
|---|---|---|---|---|
| Altering the access structures | ✓ | | | All versions |
| Altering the SQL statement | | | ✓* | All versions |
| Hints | | | ✓* | All versions |
| Altering the execution environment | | ✓ | ✓* | All versions |
| Stored outlines | | | ✓ | All versions |
| SQL profiles | | | ✓ | All versions† |
| SQL Plan Management | | | ✓ | As of version 11.1‡ |

**You have to change the SQL statement to use this technique.*

*†The Tuning Pack and, therefore, Enterprise Edition are required.*

*‡Enterprise Edition is required.*

Let me explain why these three questions are so important. First, sometimes the SQL statements are simply unknown because they're generated at runtime and change virtually for each execution. In other situations, the query optimizer can't correctly deal with specific constructs (such as a restriction in the `WHERE` clause that can't be applied through an index) that are used by lots of SQL statements. In both cases, you have to use techniques that solve the problem at the session or system level, not at the SQL statement level. This fact leads to two main problems. On one hand, as summarized in Table 11-1, several techniques can be used only for specific SQL statements. They're simply not applicable at the session or system level. On the other hand, as explained in Chapter 9, whenever your database schema is good and the configuration of the query optimizer is correctly performed, you usually have to optimize a small number of SQL statements. Therefore, you want to avoid techniques impacting the SQL statements for which the query optimizer automatically provides an efficient execution plan. Second, whenever you deal with an application for which you have no control over the SQL statements (either because the code isn't available, like in a packaged application, or because it generates SQL statements at runtime), you can't use techniques that require changes to the code. In summary, more often than not, your choice is restricted.

The aim of this chapter isn't to describe how to find out what the best execution plan for a given SQL statement is, for example, by explaining in which situation a specific access or join method should be used. This analysis is covered in the chapters in Part 4. The purpose of this chapter is solely to describe the available SQL optimization techniques.

Each section that describes a SQL optimization technique is organized in the same way. A short introduction is followed by a description of how the technique works and when you should use it. All sections end with a discussion of some common pitfalls and fallacies.

## Altering the Access Structures

This technique isn't tied to a specific feature. It's simply a fact that the response time of a SQL statement is strongly dependent not only on how the processed data is stored but also on how the processed data can be accessed.

### How It Works

The first thing you have to do while questioning the performance of a SQL statement is verify which access structures are in place. Based on the information you find in the data dictionary, you should answer the following questions:

- What is the organization type of the tables involved? Is it heap, index-organized, or external? Or is the table stored in a cluster?
- Are materialized views containing the needed data available?
- What indexes exist on the tables, clusters, and materialized views? Which columns do the indexes contain and in what order?
- How are all these segments partitioned?

Next you have to assess whether the available access structures are adequate to efficiently process the SQL statement you're optimizing. For example, during this analysis, you may discover that an additional index is necessary to efficiently support the WHERE clause of the SQL statement. Let's say that you're investigating the performance of the following query:

```
SELECT *
FROM emp
WHERE empno = 7788
```

Basically, the following execution plans can be considered by the query optimizer to execute it. While the first execution plan performs a full table scan, the second one accesses the table through an index. Naturally, the second can be considered only if the index exists:

```
----------------------------------
| Id  | Operation         | Name |
----------------------------------
|   0 | SELECT STATEMENT  |      |
|   1 |  TABLE ACCESS FULL| EMP  |
----------------------------------
```

```
----------------------------------------------
| Id  | Operation                   | Name   |
----------------------------------------------
|   0 | SELECT STATEMENT            |        |
|   1 |  TABLE ACCESS BY INDEX ROWID| EMP    |
|   2 |   INDEX UNIQUE SCAN         | EMP_PK |
----------------------------------------------
```

No more information about this topic is provided here because several chapters in Part 4 cover in detail when and how the different access structures should be used. For the moment, it's just important to recognize that this is a fundamental SQL optimization technique.

### When to Use It

Without the necessary access structures in place, it may be impossible to optimize a SQL statement. Therefore, you should consider using this technique whenever you're able to change the access structures. Unfortunately, this isn't always possible, such as when you're working with a packaged application and the vendor doesn't support altering the access structures.

### Pitfalls and Fallacies

When altering the access structures, it's essential to carefully consider possible side effects. Generally speaking, every altered access structure introduces both positive and negative consequences. In fact, it's unlikely that the impact of such a measure is restricted to a single SQL statement. There are very few situations where this isn't the case. For instance, if you add an index like in the previous example, you have to consider that the index will slow down the execution of every `INSERT` and `DELETE` statement on the indexed table as well as every `UPDATE` statement that modifies the indexed columns. You should also check whether the necessary space is available to add access structures. All things considered, you need to carefully determine whether the pros outweigh the cons before altering access structures.

## Altering the SQL Statement

SQL is a very powerful and flexible query language. Frequently, you're able to submit the very same request in many different ways. For developers, this is particularly useful. For the query optimizer, however, it's a real challenge to provide efficient execution plans for all sorts of SQL statements. Remember, flexibility is the enemy of performance.

### How It Works

Let's say you're selecting all departments without employees in the `scott` schema. The following four SQL statements, which can be found in the `depts_wo_emps.sql` script, return the information you're looking for:

```
SELECT deptno
FROM dept
WHERE deptno NOT IN (SELECT deptno FROM emp)

SELECT deptno
FROM dept
WHERE NOT EXISTS (SELECT 1 FROM emp WHERE emp.deptno = dept.deptno)
```

```
SELECT deptno FROM dept
MINUS
SELECT deptno FROM emp

SELECT dept.deptno
FROM dept, emp
WHERE dept.deptno = emp.deptno(+) AND emp.deptno IS NULL
```

The purpose of these four SQL statements is the same. The results they return are the same as well. Therefore, you might expect the query optimizer to provide the same execution plan in all cases. This is, however, not what happens. In fact, only the second and the fourth use the same execution plan. The others are quite different. Note these execution plans were generated on version 12.1. Other versions can generate different execution plans:

```
-------------------------------------
| Id  | Operation         | Name    |
-------------------------------------
|   0 | SELECT STATEMENT  |         |
|   1 |  HASH JOIN ANTI NA |        |
|   2 |   INDEX FULL SCAN | DEPT_PK |
|   3 |   TABLE ACCESS FULL| EMP    |
-------------------------------------

-------------------------------------
| Id  | Operation         | Name    |
-------------------------------------
|   0 | SELECT STATEMENT  |         |
|   1 |  HASH JOIN ANTI   |         |
|   2 |   INDEX FULL SCAN | DEPT_PK |
|   3 |   TABLE ACCESS FULL| EMP    |
-------------------------------------

--------------------------------------
| Id  | Operation          | Name    |
--------------------------------------
|   0 | SELECT STATEMENT   |         |
|   1 |  MINUS             |         |
|   2 |   SORT UNIQUE NOSORT|        |
|   3 |    INDEX FULL SCAN | DEPT_PK |
|   4 |   SORT UNIQUE      |         |
|   5 |    TABLE ACCESS FULL| EMP    |
--------------------------------------

-------------------------------------
| Id  | Operation         | Name    |
-------------------------------------
|   0 | SELECT STATEMENT  |         |
|   1 |  HASH JOIN ANTI   |         |
|   2 |   INDEX FULL SCAN | DEPT_PK |
|   3 |   TABLE ACCESS FULL| EMP    |
-------------------------------------
```

Basically, although the method used to access the data is always the same, the method used to combine the data to produce the result set is different. In this specific case, the two tables are very small, and consequently, you wouldn't notice any real performance difference with these execution plans. Naturally, if you're dealing with much bigger tables, that may not necessarily be the case. Generally speaking, whenever you process a large amount of data, every small difference in the execution plan could lead to substantial differences in the response time or resource utilization.

The key point here is to realize that the very same data can be extracted by means of different SQL statements. Whenever you're optimizing a SQL statement, you should ask yourself whether other equivalent SQL statements exist. If they do, compare their execution plans carefully to assess which one provides the best performance.

### When to Use It

Whenever you're able to change the SQL statement, you should consider this technique. There is no reason for not doing it.

### Pitfalls and Fallacies

SQL statements are code. The first rule of writing code is to make it maintainable. In the first place, this means that it should be readable and concise. Unfortunately, with SQL, because of the reasons explained earlier, the simplest or most readable way of writing a SQL statement doesn't always lead to the most efficient execution plan. Consequently, in some situations you may be forced to give up readability and conciseness for performance, although only when it's really necessary and profitable to do so.

## Hints

According to the Merriam-Webster online dictionary, a *hint* is an indirect or summary suggestion. In Oracle's parlance, the definition of a hint is a bit different. Simply put, hints are directives added to SQL statements to influence the query optimizer's decisions. In other words, a hint is something that impels toward an action, rather than merely suggests one. It seems to me that Oracle's choice of this word wasn't the best when naming this feature. In any case, the name isn't that important. What hints can do for you is important. Just don't let the name mislead you.

■ **Caution** Just because a hint is a directive, it doesn't mean that the query optimizer will always use it. Or, seeing it the other way around, just because a hint isn't used by the query optimizer, it doesn't imply that a hint is merely a suggestion. As I describe in a moment, there are cases where a hint is simply not relevant or legal, and therefore has no influence over the execution plan generated by the query optimizer.

### How It Works

The following sections describe what hints are, which categories of hints exist, and how to use them. The essential thing to note before looking at the details is that using hints isn't as trivial as you might think. Actually, in practice, it's quite common to see hints incorrectly applied.

## What Are Hints?

While working on a SQL statement, the query optimizer may have to take a lot of execution plans into account. In theory, it should consider all possible execution plans. In practice, except for simple SQL statements, it's not feasible to consider too many combinations in order to keep the optimization time reasonable. Consequently, the query optimizer excludes some of the execution plans *a priori*. Of course, the decision to completely ignore some of them may be critical, and the query optimizer's credibility is at stake in doing so.

Whenever you specify a hint, your goal is to either change the execution environment, activate or deactivate a specific feature, or reduce the number of execution plans considered by the query optimizer. Except when you change the execution environment, with a hint you tell the query optimizer which operations should or shouldn't be considered for a specific SQL statement. For instance, let's say the query optimizer has to produce the execution plan for the following query:

```
SELECT *
FROM emp
WHERE empno = 7788
```

If the emp table is a heap table and its empno column is indexed, the query optimizer considers at least two execution plans. The first is to completely read the emp table through a full table scan:

```
----------------------------------
| Id  | Operation         | Name |
----------------------------------
|   0 | SELECT STATEMENT  |      |
|   1 |  TABLE ACCESS FULL| EMP  |
----------------------------------
```

The second is to do an index lookup based on the predicate in the WHERE clause (empno = 7788) and then, through the rowid found in the index, to access the table:

```
----------------------------------------------
| Id  | Operation                   | Name   |
----------------------------------------------
|   0 | SELECT STATEMENT            |        |
|   1 |  TABLE ACCESS BY INDEX ROWID| EMP    |
|   2 |   INDEX UNIQUE SCAN         | EMP_PK |
----------------------------------------------
```

In such a case, to control the execution plan provided by the query optimizer, you could add a hint specifying to use either the full table scan or the index scan. The important thing to understand is that you can't tell the query optimizer, "I want a full table scan on the emp table, so search for an execution plan containing it." However, you can tell it, "If you have to decide between a full table scan and an index scan on the emp table, take a full table scan." This is a slight but fundamental difference. Hints can allow you to influence the query optimizer when it has to choose between several possibilities.

To further emphasize this essential point, let's take an example based on the decision tree shown in Figure 11-1. Note that even if the query optimizer works with decision trees, this is a general example not directly related to Oracle Database. In Figure 11-1, the aim is to descend the decision tree by starting at the root node (1) and ending at a leaf node (111–123). In other words, the goal is to choose a path going from point A to point B. Let's say that, for some reason, it's necessary to go through node 122. To do so, two hints, in the Oracle parlance, are added to prune the paths from node 12 to the nodes 121 and 123. In this way, the only path going on from node 12 leads to the node 122. But this isn't enough to ensure that the path goes through node 122. In fact, if at node 1 it goes through node 11 instead of node 12, the two hints would never have an effect. Therefore, to lead the path through node 122, you should add another hint pruning the path from node 1 to node 11.

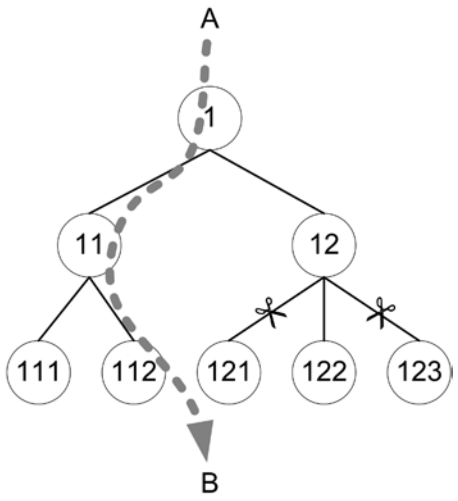


***Figure 11-1.*** *Pruning of a decision tree*

Something similar may happen with the query optimizer as well. In fact, hints are evaluated only when they apply to a decision that the query optimizer has to take. No more, no less. For this reason, as soon as you specify a hint, you may be forced to add several of them to ensure it works. And, in practice, as the complexity of the execution plans increases, it's more and more difficult to find all the necessary hints that lead to the desired execution plan.

## Specifying Hints

Hints are an Oracle extension. To not jeopardize the compatibility of the SQL statements with other database engines, Oracle decided to add them as a special kind of comment. The only differences between comments and hints are the following:

- Hints must follow immediately after `DELETE`, `INSERT`, `MERGE`, `SELECT`, and `UPDATE` keywords. In other words, they can't be specified anywhere in the SQL statement like comments can.
- The first character after the comment delimiter must be a plus sign (+).

Syntactical errors in hints don't raise errors. If the parser doesn't manage to parse them, they're simply considered real comments. Sometimes mixing comments and hints is also possible. Here are two examples that show how to force a full table scan on the `emp` table for the query discussed in the previous section:

```
SELECT /*+ full(emp) */ *
FROM emp
WHERE empno = 7788
```

```
SELECT /*+ full(emp) you can add a real comment after the hint */ *
FROM emp
WHERE empno = 7788
```

However, mixing comments and hints don't always work. For example, a comment added before a hint invalidates it. The following query shows such a case:

```
SELECT /*+ but this one does not work full(emp) */ *
FROM emp
WHERE empno = 7788
```

Because comments can invalidate hints, I don't advise you to mix comments and hints. It's much better to separate them.

## Categories of Hints

There are several methods (points of view) of categorizing hints. Personally, I like to group them in the following categories:

- *Initialization parameter hints* overwrite the setting of some initialization parameters defined at the system or session level. I classify the following hints in this category: `all_rows`, `cursor_sharing_exact`, `dynamic_sampling`, `first_rows`, `gather_plan_statistics`, `optimizer_features_enable`, and `opt_param`. I cover these hints in the section "Altering the Execution Environment" later in this chapter, and I cover the `gather_plan_statistics` hint in Chapter 10. Note that these hints always overwrite the values set at the instance or session level when they're specified.
- *Query transformation hints* control the utilization of query transformation techniques during the logical optimization. I put the following hints in this category: (no_)`eliminate_join`, `no_expand`, (no_)`expand_table`, (no_)`fact`, (no_)`merge`, (no_)`outer_join_to_inner`, (no_)`push_pred`, (no_)`push_subq`, (no_)`native_full_outer_join`, `no_query_transformation`, (no_)`rewrite`, (no_)`star_transformation`, (no_)`unnest`, `no_xmlindex_rewrite`, `no_xml_query_rewrite`, and `use_concat`. I cover some of these hints later in this chapter and some others in Chapters 14 and 15.
- *Access path hints* control the method used to access data (for example, whether an index is used). I classify the following hints in this category: `cluster`, `full`, `hash`, (no_)`index`, `index_asc`, `index_combine`, `index_desc`, (no_)`index_ffs`, `index_join`, (no_)`index_ss`, `index_ss_asc`, and `index_ss_desc`. I cover these hints along with access methods in Chapter 13.
- *Join hints* control not only the join method but also the order used to join tables. I put the following hints in this category: `leading`, (no_)`nlj_batching`, `ordered`, (no_)`swap_join_inputs`, (no_)`use_cube`, (no_)`use_hash`, (no_)`use_merge`, `use_merge_cartesian`, (no_)`use_nl`, and `use_nl_with_index`. I cover these hints along with the join methods in Chapter 14.

- *Parallel processing hints* control how and whether parallel processing is used. I classify the following hints in this category: (no_)parallel, (no_)parallel_index, (no_)pq_concurrent_union, pq_distribute, pq_filter, (no_)pq_skew, (no_)px_join_filter and (no_)statement_queuing. I cover these hints along with parallel processing in Chapter 15. A possible utilization of the pq_distribute hint is provided along with partition-wise joins in Chapter 14.
- *Other hints* control the utilization of other features that aren't related to the previous categories. I classify the following hints in this category: (no)append, append_values, (no_)bind_aware, (no_)result_cache, (no)cache, change_dupkey_error_index, driving_site, (no_)gather_optimizer_statistics, ignore_row_on_dupkey_index, inline, materialize, (no_)monitor, model_min_analysis, (no_)monitor, qb_name and retry_on_row_change. I cover the qb_name hint later in this chapter and some others throughout the book.

Although through this book I describe or show plenty of examples of hints, I provide neither real references nor their full syntax. Such references are given in Chapter 2 of *Oracle Database SQL Language Reference* manual.

It's worth pointing out that a significant number of hints disabling a specific operation or feature (the hints prefixed by no_) are available. This is good because it's sometimes easier to specify which operation or feature should not be used rather than specify which one should be used.

The list of hints just provided isn't complete; it just contains the hints documented in the *SQL Reference Guide* manual. There are other hints that aren't documented. You will see some examples in the "SQL Profiles" section later in this chapter. As of version 11.1, you can select the v$sql_hint view to get what is an almost complete list of all hints.

## Validity of Hints

Simple SQL statements have a single query block. Multiple query blocks exist whenever views or constructs such as subqueries, in-line views, and set operators are used. For example, the following query has two query blocks (I'm using the subquery factoring clause instead of defining a real view for illustration purposes only). The first is the main query that references the dept table. The second is the subquery that references the emp table:

```
WITH
  emps AS (SELECT deptno, count(*) AS cnt
           FROM emp
           GROUP BY deptno)
SELECT dept.dname, emps.cnt
FROM dept, emps
WHERE dept.deptno = emps.deptno
```

In general, initialization parameter hints are valid for the whole SQL statement (for example, dynamic_sampling is an exception). Most other hints are valid for a single query block only (exceptions are, for example, bind_aware and monitor). Hints valid for a single query block have to be specified in the block they control. For instance, if you want to specify an access path hint for both tables in the previous query, one hint has to be added into the main query and the other in the subquery. The validity of both is restricted to the query block where they're defined:

```
WITH
  emps AS (SELECT /*+ full(emp) */ deptno, count(*) AS cnt
           FROM emp
           GROUP BY deptno)
SELECT /*+ full(dept) */ dept.dname, emps.cnt
FROM dept, emps
WHERE dept.deptno = emps.deptno
```

Exceptions to this rule are the *global hints*. With them it's possible to reference objects contained in other query blocks (provided they're named) by using the dot notation. For example, in the following SQL statement, the main query contains a hint intended for the subquery. Notice how the subquery name is used for the reference:

```
WITH
  emps AS (SELECT deptno, count(*) AS cnt
           FROM emp
           GROUP BY deptno)
SELECT /*+ full(dept) full(emps.emp) */ dept.dname, emps.cnt
FROM dept, emps
WHERE dept.deptno = emps.deptno
```

The syntax of global hints supports references for more than two levels (for example, for a view referenced in another view). The objects must simply be separated by a dot (for example, `view1.view2.view3.table`).

---

■ **Tip** Because global hints don't always cope with some query transformations, I advise you to use the syntax based on query block names (shown shortly) instead.

---

Because subqueries in the `WHERE` clause aren't named, their objects can't be referenced with global hints. To solve this problem, there is another way to achieve the same result. In fact, most hints accept a parameter specifying which query block they're valid for. In this way, the hints may be grouped at the beginning of a SQL statement and simply reference the query block to which they apply. To allow these references, not only does the query optimizer generate a *query block name* for each query block, but it also allows you to specify your own names through the `qb_name` hint. For instance, in the following query, the two query blocks are called `main` and `sq`, respectively. Then, in the `full` hint, the query block names are referenced by prefixing them with an `@` sign. Notice how the access path hint for the `emp` table of the subquery is specified in the main query:

```
WITH
  emps AS (SELECT /*+ qb_name(sq) */ deptno, count(*) AS cnt
           FROM emp
           GROUP BY deptno)
SELECT /*+ qb_name(main) full(@main dept) full(@sq emp) */ dept.dname, emps.cnt
FROM dept, emps
WHERE dept.deptno = emps.deptno
```

The previous example showed how to specify your own names. Now let's see how you can use the names generated by the query optimizer. First, you have to know what they are. For that, you can use the `EXPLAIN PLAN` statement and the `dbms_xplan` package, as shown in the following example. Note that the `alias` option is passed to the `display` function to make sure that the query block names and aliases are part of the output:

```
SQL> EXPLAIN PLAN FOR
  2  WITH emps AS (SELECT deptno, count(*) AS cnt
  3                FROM emp
  4                GROUP BY deptno)
  5  SELECT dept.dname, emps.cnt
  6  FROM dept, emps
  7  WHERE dept.deptno = emps.deptno;

SQL> SELECT * FROM table(dbms_xplan.display(NULL, NULL, 'basic +alias'));
```

```
-------------------------------------
| Id  | Operation             | Name |
-------------------------------------
|   0 | SELECT STATEMENT      |      |
|   1 |  HASH JOIN            |      |
|   2 |   VIEW                |      |
|   3 |    HASH GROUP BY      |      |
|   4 |     TABLE ACCESS FULL| EMP  |
|   5 |   TABLE ACCESS FULL   | DEPT |
-------------------------------------

Query Block Name / Object Alias (identified by operation id):
-------------------------------------------------------------

   1 - SEL$2
   2 - SEL$1 / EMPS@SEL$2
   3 - SEL$1
   4 - SEL$1 / EMP@SEL$1
   5 - SEL$2 / DEPT@SEL$2
```

The system-generated query block names are composed of a prefix followed by an alphanumeric string. The prefix is based on the operation contained in the query block. Table 11-2 summarizes them. The alphanumeric string is a numeration of the query blocks, based on their position (left to right) during the parse of the SQL statement. In the previous example, the main query block is named SEL$2, and the subquery query block is named SEL$1.

***Table 11-2.*** *Prefixes Used in Query Block Names*

| Prefix | Used For |
|---|---|
| CRI$ | CREATE INDEX statements |
| DEL$ | DELETE statements |
| INS$ | INSERT statements |
| MISC$ | Miscellaneous SQL statements like LOCK TABLE |
| MRC$ | MERGE statements |
| SEL$ | SELECT statements |
| SET$ | Set operators like UNION and MINUS |
| UPD$ | UPDATE statements |

As shown here, the utilization of system-generated query block names isn't different from the utilization of user-defined query block names:

```
WITH
  emps AS (SELECT deptno, count(*) AS cnt
           FROM emp
           GROUP BY deptno)
SELECT /*+ full(@sel$2 dept) full(@sel$1 emp) */ dept.dname, emps.cnt
FROM dept, emps
WHERE dept.deptno = emps.deptno
```

I need to make one last comment about the naming of the query blocks generated during the query transformation phase. Because they aren't part of the SQL statement during the parse, they can't be numbered like the others. In such cases, the query optimizer generates an eight-character hash value for them. The following example shows that situation. Here, the system-generated query block name is SEL$5DA710D3:

```
SQL> EXPLAIN PLAN FOR
  2  SELECT deptno
  3  FROM dept
  4  WHERE NOT EXISTS (SELECT 1 FROM emp WHERE emp.deptno = dept.deptno);

SQL> SELECT * FROM table(dbms_xplan.display(NULL,NULL,'basic +alias'));

-----------------------------------
| Id  | Operation          | Name |
-----------------------------------
|   0 | SELECT STATEMENT   |      |
|   1 |  HASH JOIN ANTI    |      |
|   2 |   TABLE ACCESS FULL| DEPT |
|   3 |   TABLE ACCESS FULL| EMP  |
-----------------------------------

Query Block Name / Object Alias (identified by operation id):
-------------------------------------------------------------

   1 - SEL$5DA710D3
   2 - SEL$5DA710D3 / DEPT@SEL$1
   3 - SEL$5DA710D3 / EMP@SEL$2
```

In the preceding output, it's interesting to notice that when such a query transformation takes place, for some lines in the execution plan (for example, line 2) there are two query block names. Both can be used in hints. However, the query block name after the query transformation (in this case SEL$5DA710D3) is available, from a query optimizer point of view, only when the very same query transformation takes place.

## When to Use It

The purpose of hints is twofold. First, they're convenient as workarounds when the query optimizer doesn't manage to automatically generate an efficient execution plan. In such cases, you would use them to get a better execution plan. The important thing to emphasize is that hints are workarounds and, therefore, should not be used as long-term solutions. In some situations, however, they're the only practicable way to solve a problem. Second, hints are useful for assessing the decisions of the query optimizer in that they lead to the generation of alternative execution plans. In such cases, you would use them to do a kind of what-if analysis.

## Pitfalls and Fallacies

Every time you want to lock up a specific execution plan through access path hints, join hints, or parallel processing hints, you must carefully specify enough hints to achieve stability. Here, stability means that even if the object statistics and, to some extent, the access structures change, the execution plan doesn't change. To lock up a specific execution plan, it's not unusual to have to add not only an access path hint for each table in the SQL statement but also several join hints to control the join methods and order. Note that other types of hints (for example, initialization parameter hints and query transformation hints) are usually not subject to this problem.

While processing a SQL statement, the parser checks the syntax of the hints. In spite of this, no error is raised when a hint is found with invalid syntax. This implies that the parser considers this particular pseudohint to be a comment. From one perspective, this is annoying if it's because of a typing error. On the other hand, this is beneficial because it avoids breaking an already deployed application because of changes to the access structures, which are often referenced in hints (for example, the `index` hint might reference an index name), or an upgrade to a newer database version. That said, I would welcome a way of validating the hints contained in a SQL statement. For instance, through the `EXPLAIN PLAN` statement, it should be simple enough to provide a warning (for example, a new note in the `dbms_xplan` output) in that regard. The only way I know to be able to do this partially is by setting the event 10132. In fact, at the end of the output generated by this event is a section dedicated to hints. You can check two things in this section. First, each hint should be listed. If a hint is missing, it means that it hasn't been recognized as such. Second, check whether a message informing that some hints have errors is present. (The `err` field is set to a value greater than 0 in such cases.) Note that to get the following output, two initialization parameter hints that conflict with each other were specified:

```
Dumping Hints
=============
  atom_hint=(@=0x6b796498 err=4 resol=0 used=0 token=454 org=1 lvl=1 txt=ALL_ROWS )
  atom_hint=(@=0x6b796578 err=4 resol=0 used=0 token=453 org=1 lvl=1 txt=FIRST_ROWS )
********** WARNING: SOME HINTS HAVE ERRORS *********
```

Be aware that with this method, hints having good syntax but referencing wrong objects aren't reported as having errors. So, this isn't a definitive check that everything is fine.

One of the most common mistakes made in the utilization of hints is related to table aliases. The rule is that when a table is referenced in a hint, the alias should be used instead of the table name, whenever the table has an alias. In the following example, you can see how a table alias (`e`) is defined for the `emp` table. In such a case, when the `full` hint referencing the table uses the table name, the hint has no effect. Notice how in the first example, an index scan is used instead of the wanted full table scan:

```
SQL> EXPLAIN PLAN FOR SELECT /*+ full(emp) */ * FROM emp e WHERE empno = 7788;

SQL> SELECT * FROM table(dbms_xplan.display(NULL,NULL,'basic'));

----------------------------------------------
| Id  | Operation                    | Name   |
----------------------------------------------
|   0 | SELECT STATEMENT             |        |
|   1 |  TABLE ACCESS BY INDEX ROWID| EMP    |
|   2 |   INDEX UNIQUE SCAN         | EMP_PK |
----------------------------------------------

SQL> EXPLAIN PLAN FOR SELECT /*+ full(e) */ * FROM emp e WHERE empno = 7788;

SQL> SELECT * FROM table(dbms_xplan.display(null,null,'basic'));

----------------------------------
| Id  | Operation          | Name |
----------------------------------
|   0 | SELECT STATEMENT  |      |
|   1 |  TABLE ACCESS FULL| EMP  |
----------------------------------
```

Something that should be checked but is frequently forgotten is the impact of hints during upgrades. Because hints are convenient workarounds in situations where the query optimizer doesn't manage to automatically provide efficient execution plans but their effect depends on the type of decision tree (see Figure 11-1) used by the query optimizer, whenever hinted SQL statements are executed from another database version (and, therefore, from another query optimizer version), they should be carefully checked. In other words, while validating an application against a new database version, the best practice is to reexamine and retest all SQL statements containing hints. For testing purposes, you might also want to disable all hints by setting the `_optimizer_ignore_hints` undocumented initialization parameter to `TRUE` at the session level. Be careful—you should avoid setting it at the system level because of the many hints used by the database engine itself.

Because views may be used in different contexts, specifying hints in views is usually not recommended. If you really have to add hints in views, make sure the hints make sense for all modules using them.

## Altering the Execution Environment

Chapter 9 describes how to configure the query optimizer. The configuration is the default execution environment used by all users connected to the database engine. Consequently, it must be suitable for most of them. When a database is used by multiple applications (for example, because of database server consolidation) or by a single application with different requirements that depend on the module in use (for example, OLTP during the day and batch during the night), it's not uncommon for a single environment to not be adequate in every situation. In such cases, altering the execution environment at the session level or even at the SQL statement level could be appropriate.

### How It Works

Altering the execution environment at the session level is completely different from doing it at the SQL statement level. Because of this, I describe the two situations in two distinct subsections. In addition, I describe several dynamic performance views displaying the environment related to a database instance, a single session, or a child cursor.

### Session Level

Most initialization parameters described in Chapter 9 can be changed at the session level with the `ALTER SESSION` statement. So, if you have users or modules requiring a particular configuration, you should simply change the defaults at the session level. For instance, to set up the execution environment, depending on the user connecting to the database, you could use a configuration table and a database trigger, as shown in the following example. You can find the SQL statements in the `exec_env_trigger.sql` script:

```
CREATE TABLE exec_env_conf (username  VARCHAR2(30),
                            parameter VARCHAR2(80),
                            value     VARCHAR2(512))

CREATE OR REPLACE TRIGGER execution_environment AFTER LOGON ON DATABASE
BEGIN
  FOR c IN (SELECT parameter, value
            FROM exec_env_conf
            WHERE username = sys_context('userenv','session_user'))
  LOOP
    EXECUTE IMMEDIATE 'ALTER SESSION SET ' || c.parameter || '=' || c.value;
  END LOOP;
END;
```

Then for each user requiring a particular configuration, you insert one row in the configuration table for each initialization parameter. For example, the following two INSERT statements change and define two parameters at the session level when the user named Alberto logs in:

```
INSERT INTO exec_env_conf VALUES ('ALBERTO', 'optimizer_mode', 'first_rows_10')

INSERT INTO exec_env_conf VALUES ('ALBERTO', 'optimizer_dynamic_sampling', '0')
```

Of course, you could also define the trigger for a single schema or perform other checks based, for example, on the userenv context.

## SQL Statement Level

The execution environment at the SQL statement level is changed through the initialization parameter hints. Because hints are used, the behavior and properties of hints previously described apply in this case as well.

Not all initialization parameters making up the query optimizer configuration can be changed at the SQL statement level. Table 11-3 summarizes which parameters and values have corresponding initialization parameter hints in order to achieve the same configuration at the SQL statement level. Note that for some initialization parameters (for example, cursor_sharing), not all values can be set with hints.

***Table 11-3.*** *Hints That Change the Query Optimizer Configuration at the SQL Statement Level*

| Initialization Parameter | Hint |
|---|---|
| cursor_sharing=exact | cursor_sharing_exact |
| optimizer_dynamic_sampling=x | dynamic_sampling(x) |
| optimizer_features_enable=x | optimizer_features_enable('x') |
| optimizer_features_enable not set | optimizer_features_enable(default) |
| optimizer_index_caching=x | opt_param('optimizer_index_caching' x) |
| optimizer_index_cost_adj=x | opt_param('optimizer_index_cost_adj' x) |
| optimizer_mode=all_rows | all_rows |
| optimizer_mode=first_rows | first_rows |
| optimizer_mode=first_rows_x | first_rows(x) |
| optimizer_secure_view_merging=x | opt_param('optimizer_secure_view_merging' 'x') |
| optimizer_use_pending_statistics=x | opt_param('optimizer_use_pending_statistics' 'x') |
| result_cache_mode=manual | no_result_cache |
| result_cache_mode=force | result_cache |
| star_transformation_enabled=x | opt_param('star_transformation_enabled' 'x') |

## Dynamic Performance Views

There are three dynamic performance views that provide information about the execution environment:

- v$sys_optimizer_env gives information about the execution environment at the instance level. For example, it's possible to find out which initialization parameters aren't set to the default value:

```
SQL> SELECT name, value, default_value
  2  FROM v$sys_optimizer_env
  3  WHERE isdefault = 'NO';

NAME                         VALUE DEFAULT_VALUE
---------------------------- ----- -------------
star_transformation_enabled true  false
```

- v$ses_optimizer_env gives information about the execution environment for each session. Because no column provides information about whether an initialization parameter has been modified at the system or session level, a query like the following can be used for that purpose:

```
SQL> SELECT name, value
  2  FROM v$ses_optimizer_env
  3  WHERE sid = 124 AND isdefault = 'NO'
  4  MINUS
  5  SELECT name, value
  6  FROM v$sys_optimizer_env;

NAME           VALUE
-------------- -------------
cursor_sharing force
optimizer_mode first_rows_10
```

- v$sql_optimizer_env gives information about the execution environment for each child cursor present in the library cache. For example, with a query like the following, it's possible to find out whether two child cursors belonging to the same parent cursor have a different execution environment:

```
SQL> SELECT e0.name, e0.value AS value_child_0, e1.value AS value_child_1
  2  FROM v$sql_optimizer_env e0, v$sql_optimizer_env e1
  3  WHERE e0.sql_id = e1.sql_id
  4  AND e0.sql_id = 'a5ks9fhw2v9s1'
  5  AND e0.child_number = 0
  6  AND e1.child_number = 1
  7  AND e0.name = e1.name
  8  AND e0.value <> e1.value;

NAME                 VALUE_CHILD_0 VALUE_CHILD_1
-------------------- ------------- -------------
hash_area_size       33554432      131072
optimizer_mode       first_rows_10 all_rows
cursor_sharing       force         exact
workarea_size_policy manual        auto
```

### When to Use It

Whenever the default configuration isn't suitable for part of the application or part of the users, changing it is a good thing. Although changing the initialization parameters at the session level should always be possible, hints can be used only when it's also possible to change the SQL statements.

### Pitfalls and Fallacies

Altering the execution environment at the session level is easy when the setting can be centralized either in the database or in the application. Take extra care if you're using a connection pool shared by applications or modules that need a different execution environment. In fact, session parameters are associated with the physical connection. Because the physical connection might have been used by another application or module, you must set the execution environment every time you get a connection from the pool (which is of course expensive, because additional round-trips to the database are needed). To avoid this overhead, if you have applications or modules needing different execution environments, you should use different connection pools using different users as well. In this way, you can have a single configuration for each connection pool, and by defining different users to connect to the database, you may possibly be able to centralize the configuration into a simple database trigger.

Altering the execution environment at the SQL statement level is also subject to the same pitfalls and fallacies previously described for hints.

## Stored Outlines

Stored outlines are designed to provide stable execution plans in case of changes in the execution environment or object statistics. For this reason, this feature is also called *plan stability*. Two important scenarios that can benefit from this feature are reported in the Oracle documentation. The first is the migration from the rule-based optimizer to the cost-based optimizer. The second is the upgrade of an Oracle Database release to a newer one. In both cases, the idea is to store information about the execution plans while the application is using the old configuration or version and then use that information to provide the same execution plans against the newer one. In practice, unfortunately, even with stored outlines in place, you may observe changes in execution plans. Probably for this reason, I've never seen a single database where stored outlines were used on a large scale. Consequently, in practice, stored outlines are used for specific SQL statements only.

---

■ **Note** From version 11.1 onward, stored outlines are deprecated in favor of *SQL plan management* (covered later in this chapter).

---

### How It Works

The following sections describe what stored outlines are and how to work with them.

#### What Are Stored Outlines?

A stored outline is an object associated to a SQL statement and is designed to influence the query optimizer while it generates an execution plan for the SQL statement. More concretely, a stored outline is a set of hints or, more precisely, all the hints that are necessary to force the query optimizer to consistently generate a specific execution plan for a given SQL statement.

■ **Caution** Not all hints can be stored in stored outlines. To know which hint can't be stored, you can run the following query:

```
SELECT name FROM v$sql_hint WHERE version_outline IS NULL
```

Even though most of the hints that can't be stored in stored outlines don't impact execution plans (for example, `gather_plan_statistics`), some of them do (for example, `materialize` and `inline`). As a result, there are some execution plans that can't be forced through a stored outline without specifying a hint in the SQL statement itself.

One of the advantages of a stored outline is that it applies to a specific SQL statement, but you don't need to modify the SQL statement in order to apply the stored outline. Stored outlines are stored in the data dictionary, and the query optimizer selects them automatically. Figure 11-2 shows the basic steps carried out during this selection. First, the SQL statement is normalized by removing blank spaces and converting nonliteral strings to uppercase. The signature of the resulting SQL statement (a hash value of the SQL statement's text) is computed. Then, based on that signature, a lookup in the data dictionary is performed. Whenever a stored outline with the same signature is found, a check is performed to make sure that the SQL statement to be optimized and the SQL statement tied to the stored outline are equivalent. This is necessary because the signature is a hash value, and consequently there could be conflicts. If the test is successful, the hints making up the stored outline are included in the generation of the execution plan.

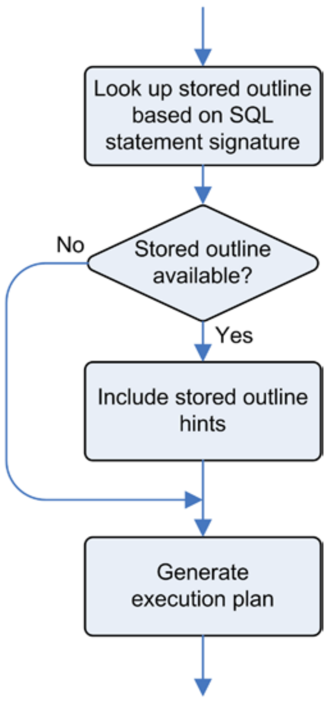

Look up stored outline based on SQL statement signature
Stored outline available?
No
Yes
Include stored outline hints
Generate execution plan

*Figure 11-2. Main steps carried out during the selection of a stored outline*

## Creating Stored Outlines

You can use two main methods to create stored outlines. Either you let the database automatically create them or you do it manually. The first method is useful if you want to create a stored outline for each SQL statement executed by a given session or even by the whole system. Nevertheless, as mentioned earlier, this is usually not desirable. Because of this, you'll usually create them manually.

To activate the automatic creation, you have to set the `create_stored_outlines` initialization parameter either to `TRUE` or to another value specifying a *category*. The purpose of the category is to group together several stored outlines for management purposes. The default category, which is used when the initialization parameter is set to `TRUE`, is named `DEFAULT`. The initialization parameter is dynamic and can be changed at the session and system levels. To disable automatic creation, you have to set the initialization parameter to `FALSE`.

To manually create a stored outline, you have to use the `CREATE OUTLINE` statement. The following SQL statement, an excerpt of the `outline_from_text.sql` script, shows the creation of a stored outline named `outline_from_text`, associated with the `test` category and based on the query specified in the `ON` clause:

```
CREATE OR REPLACE OUTLINE outline_from_text
FOR CATEGORY test
ON SELECT * FROM t WHERE n = 1970
```

Once created, you can display information about stored outlines and their properties through the `user_outlines` and `user_outline_hints` views (for both, the `all`, `dba`, and, in a 12.1 multitenant environment, `cdb` views exist as well). The `user_outlines` view displays all information except the hints. The following queries show the information provided for the stored outline created by the previous SQL statement:

```
SQL> SELECT category, sql_text, signature
  2  FROM user_outlines
  3  WHERE name = 'OUTLINE_FROM_TEXT';

CATEGORY SQL_TEXT                       SIGNATURE
-------- ------------------------------ --------------------------------
TEST     SELECT * FROM t WHERE n = 1970 73DC40455AF10A40D84EF59A2F8CBFFE

SQL> SELECT hint
  2  FROM user_outline_hints
  3  WHERE name = 'OUTLINE_FROM_TEXT';

HINT
-------------------------------------
FULL(@"SEL$1" "T"@"SEL$1")
OUTLINE_LEAF(@"SEL$1")
ALL_ROWS
DB_VERSION('11.2.0.3')
OPTIMIZER_FEATURES_ENABLE('11.2.0.3')
IGNORE_OPTIM_EMBEDDED_HINTS
```

You're also able to manually create a stored outline by referencing a cursor in the library cache. The following example, an excerpt of the output generated by the `outline_from_sqlarea.sql` script, shows how to select the cursor in the library cache and to create the stored outline through the `create_outline` procedure in the `dbms_outln` package:

```
SQL> SELECT hash_value, child_number
  2  FROM v$sql
  3  WHERE sql_text = 'SELECT * FROM t WHERE n = 1970';

HASH_VALUE CHILD_NUMBER
---------- ------------
 308120306            0

SQL> BEGIN
  2    dbms_outln.create_outline(hash_value   => '308120306',
  3                              child_number => 0,
  4                              category     => 'test');
  5  END;
  6  /
```

---

■ **Caution** The `create_outline` procedure doesn't create a stored outline based on the execution plan associated to the referenced cursor. Instead, it takes the text of the SQL statement associated to the cursor and reparses it. Therefore, the execution plan associated to the stored outline isn't necessarily identical to the one associated to the cursor. For example, a different execution environment can easily lead to another execution plan.

---

The `create_outline` procedure accepts only the three parameters shown in the example. This means that the name of the stored outline is automatically generated. To find out the system-generated name, you have to query a view like `user_outlines`. Here's an example of query that returns the name of the last-created stored outline:

```
SQL> SELECT name
  2  FROM user_outlines
  3  WHERE timestamp = (SELECT max(timestamp) FROM user_outlines);

NAME
-----------------------------
SYS_OUTLINE_13072411155434901
```

Changing the system-generated name to something more useful is advisable. The next section describes, among other things, how to do that.

## Altering Stored Outlines

To change the name of a stored outline, you have to execute the `ALTER OUTLINE` statement:

```
ALTER OUTLINE SYS_OUTLINE_13072411155434901 RENAME TO outline_from_sqlarea
```

With the ALTER OUTLINE statement or the update_by_cat procedure in the dbms_outln package, you're also able to change the category of stored outlines. Whereas the former changes the category of a single stored outline, the latter moves all stored outlines belonging to one category to another one. However, because of bug 5759631, it's not possible with ALTER OUTLINE to change the category of a stored outline to DEFAULT (for all other categories, it's not a problem). The following example shows not only what happens if you try to change it but also how to do it with the update_by_cat procedure:

```
SQL> ALTER OUTLINE outline_from_text CHANGE CATEGORY TO DEFAULT;
ALTER OUTLINE outline_from_text CHANGE CATEGORY TO DEFAULT
                                                   *
ERROR at line 1:
ORA-00931: missing identifier

SQL> execute dbms_outln.update_by_cat(oldcat => 'TEST', newcat => 'DEFAULT')

SQL> SELECT category
  2  FROM user_outlines
  3  WHERE name = 'OUTLINE_FROM_TEXT';

CATEGORY
--------
DEFAULT
```

Finally, with the ALTER OUTLINE statement, you can also regenerate a stored outline, which is like re-creating it. Usually, you'll use this possibility if you want the query optimizer to generate a new set of hints. This could be necessary if you have changed the access structures of the objects related to the stored outline:

```
ALTER OUTLINE outline_from_text REBUILD
```

## Activating Stored Outlines

The query optimizer considers only the stored outlines that are active. To be active, a stored outline must meet two conditions. The first is that the stored outlines must be enabled. This is the default when they're created. To enable and disable a stored outline, you use the ALTER OUTLINE statement:

```
ALTER OUTLINE outline_from_text DISABLE

ALTER OUTLINE outline_from_text ENABLE
```

The second condition is that the category must be activated through the use_stored_outlines initialization parameter at the session or system level. The initialization parameter takes as a value either TRUE, FALSE, or the name of a category. If TRUE is specified, the category defaults to the value DEFAULT. The following SQL statement activates the stored outlines belonging to the test category at the session level:

```
ALTER SESSION SET use_stored_outlines = test
```

Because the use_stored_outlines initialization parameter supports a single category, at a given time a session can activate only a single category.

To know whether the query optimizer is using a stored outline, you can take advantage of the functions available in the dbms_xplan package. In fact, as shown in the following example, the Note section of their output explicitly provides the needed information:

```
SQL> EXPLAIN PLAN FOR SELECT * FROM t WHERE n = 1970;

SQL> SELECT * FROM table(dbms_xplan.display);

---------------------------------
| Id  | Operation         | Name |
---------------------------------
|   0 | SELECT STATEMENT  |      |
|*  1 |  TABLE ACCESS FULL| T    |
---------------------------------

   1 - filter("N"=1970)

Note
-----
   - outline "OUTLINE_FROM_TEXT" used for this statement
```

For a cursor stored in the library cache, the outline_category column of the v$sql view informs whether a stored outline was used during the generation of the execution plan. Unfortunately, only the category is given. The name of the stored outline itself remains unknown. If no stored outline was used, the column is set to NULL.

A method that can be used to know whether a stored outline is used over a period of time is to reset its utilization flag with the clear_used procedure in the dbms_outln package. Then, by checking the flag after a while, you can determine whether the stored outline was used. However, no detailed information about the utilization (for example, the number of times or when) is given:

```
SQL> execute dbms_outln.clear_used(name => 'OUTLINE_FROM_TEXT')

SQL> SELECT used
  2  FROM user_outlines
  3  WHERE name = 'OUTLINE_FROM_TEXT';

USED
------
UNUSED

SQL> SELECT * FROM t WHERE n = 1970;

SQL> SELECT used
  2  FROM user_outlines
  3  WHERE name = 'OUTLINE_FROM_TEXT';

USED
------
USED
```

## Moving Stored Outlines

To move stored outlines, no particular feature is provided. Basically, you have to copy them yourself from one data dictionary to the other. This is easy because the data about the stored outlines is stored in three tables in the `outln` schema: `ol$`, `ol$hints`, and `ol$nodes`. You could use the following commands to export and import all available stored outlines:

```
exp tables=(outln.ol$,outln.ol$hints,outln.ol$nodes) file=outln.dmp

imp full=y ignore=y file=outln.dmp
```

To move a single stored outline (named `outline_from_text` in this case), you can add the following parameter to the export command:

```
query="WHERE ol_name='OUTLINE_FROM_TEXT'"
```

To move all stored outlines belonging to a category (named `test`, in this case), you can add the following parameter to the export command:

```
query="WHERE category='TEST'"
```

Be careful, because you may have to add some escape characters to successfully pass all parameters, depending on your operating system and shell. For example, on my Linux server, with bash I had to execute the following command:

```
exp tables=\(outln.ol\$,outln.ol\$hints,outln.ol\$nodes\) file=outln.dmp \
    query=\"WHERE ol_name=\'OUTLINE_FROM_TEXT\'\"
```

## Editing Stored Outlines

With stored outlines, it's possible to lock up execution plans. However, this is useful only if the query optimizer is able to generate an efficient execution plan that can later be captured and frozen by a stored outline. If that's not the case, the first thing you should investigate is the possibility of modifying the execution environment, the access structures, or the object statistics just for the creation of the stored outline storing an efficient execution plan. For instance, if the execution plan for a given SQL statement uses an index scan that you want to avoid, you could drop (or make invisible) the index on a test system, generate a stored outline there, and then move the stored outline in production.

When you find no way to force the query optimizer to automatically generate an efficient execution plan, the last resort is to manually edit the stored outline. Simply put, you have to modify the hints associated with the stored outline. However, in practice, you can't simply run a few SQL statements against the *public stored outlines* (which are the kind discussed so far) stored in the data dictionary tables. Instead, you have to carry out the editing as summarized in Figure 11-3. This process is based on the modification of *private stored outlines.* These are like public stored outlines, but instead of being stored in the data dictionary, they're stored in *working tables*. The aim of using these working tables is to avoid modifying the data dictionary tables directly. Therefore, to edit a stored outline, you have to create, modify, and test a private stored outline. Then, when the private stored outline is working correctly, you publish it as a public stored outline. The `dbms_outln_edit` package and a few extensions to the `CREATE OUTLINE` statement are available for editing stored outlines.

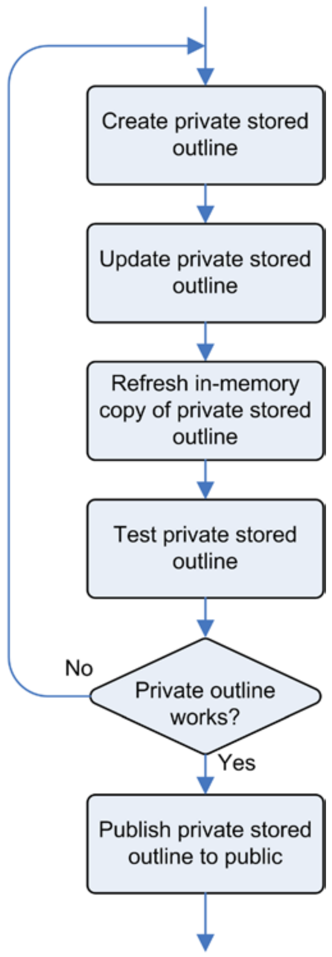

Create private stored outline
Update private stored outline
Refresh in-memory copy of private stored outline
Test private stored outline
Private outline works?
No
Yes
Publish private stored outline to public

***Figure 11-3.*** *Steps carried out during the editing of a stored outline*

Based on the example available in the `outline_editing.sql` script, I describe the whole process summarized by Figure 11-3. The purpose is to create and edit a stored outline to have a full table scan instead of an index scan for the following query:

```
SQL> EXPLAIN PLAN FOR SELECT * FROM t WHERE n = 1970;

SQL> SELECT * FROM table(dbms_xplan.display(NULL,NULL,'basic'));

--------------------------------------------
| Id  | Operation                   | Name |
--------------------------------------------
|   0 | SELECT STATEMENT            |      |
|   1 |  TABLE ACCESS BY INDEX ROWID| T    |
|   2 |   INDEX RANGE SCAN          | I    |
--------------------------------------------
```

First, you have to create a private stored outline. For that, you have two possibilities. The first is to create a private stored outline from scratch with a SQL statement like the following. The PRIVATE keyword specifies the kind of stored outline to be created:

```
SQL> CREATE OR REPLACE PRIVATE OUTLINE p_outline_editing
  2  ON SELECT * FROM t WHERE n = 1970;
```

The second possibility is to copy a public stored outline already present in the data dictionary by means of a SQL statement like the following. The PRIVATE and PUBLIC keywords specify the kind of stored outline to be created and copied respectively:

```
SQL> CREATE PRIVATE OUTLINE p_outline_editing FROM PUBLIC outline_editing;
```

Both methods create a private stored outline in the working tables. Here's the list of hints associated with that stored outline:

```
SQL> SELECT hint_text
  2  FROM ol$hints
  3  WHERE ol_name = 'P_OUTLINE_EDITING';

HINT_TEXT
--------------------------------------------
INDEX_RS_ASC(@"SEL$1" "T"@"SEL$1" ("T"."N"))
OUTLINE_LEAF(@"SEL$1")
ALL_ROWS
DB_VERSION('11.2.0.3')
OPTIMIZER_FEATURES_ENABLE('11.2.0.3')
IGNORE_OPTIM_EMBEDDED_HINTS
```

Once the private stored outline has been created, you can modify it with regular DML statements. However, changing everything as needed isn't an easy task. A much simpler approach is to create an additional private stored outline reproducing the expected execution plan and then swap the content of the two stored outlines. To create the additional stored outline, you issue a SQL statement like the following. Notice the hint used to instruct the query using a full table scan:

```
SQL> CREATE OR REPLACE PRIVATE OUTLINE p_outline_editing_hinted
  2  ON SELECT /*+ full(t) */ * FROM t WHERE n = 1970;
```

Then, you swap the content by issuing SQL statements like these:

```
SQL> UPDATE ol$
  2  SET hintcount = (SELECT hintcount
  3                   FROM ol$
  4                   WHERE ol_name = 'P_OUTLINE_EDITING_HINTED')
  5  WHERE ol_name = 'P_OUTLINE_EDITING';

SQL> DELETE ol$hints
  2  WHERE ol_name = 'P_OUTLINE_EDITING';
```

```
SQL> UPDATE ol$hints
  2  SET ol_name = 'P_OUTLINE_EDITING'
  3  WHERE ol_name = 'P_OUTLINE_EDITING_HINTED';
```

Here is the list of hints associated with the private stored outline after the swap. The only difference is that the index hint as been replaced by the full hint:

```
SQL> UPDATE ol$hints
  2  SET ol_name = 'P_OUTLINE_EDITING'
  3  WHERE ol_name = 'P_OUTLINE_EDITING_HINTED';

HINT_TEXT
-------------------------------------
FULL(@"SEL$1" "T"@"SEL$1")
OUTLINE_LEAF(@"SEL$1")
ALL_ROWS
DB_VERSION('11.2.0.3')
OPTIMIZER_FEATURES_ENABLE('11.2.0.3')
IGNORE_OPTIM_EMBEDDED_HINTS
```

To make sure that the in-memory copy of the stored outline is synchronized with the changes, you should execute the following PL/SQL call:

```
SQL> execute dbms_outln_edit.refresh_private_outline('P_OUTLINE_EDITING')
```

Then, to activate and test the private stored outline, set the use_private_outlines initialization parameter to TRUE, or to the name of the category to which the private stored outline belongs. Note how the full table scan in the execution plan and the message in the Note section both confirm the use of the private stored outline. For example:

```
SQL> ALTER SESSION SET use_private_outlines = TRUE;

SQL> EXPLAIN PLAN FOR SELECT * FROM t WHERE n = 1970;

SQL> SELECT * FROM table(dbms_xplan.display(NULL,NULL,'basic +note'));

----------------------------------
| Id  | Operation         | Name |
----------------------------------
|   0 | SELECT STATEMENT  |      |
|   1 |  TABLE ACCESS FULL| T    |
----------------------------------

Note
-----
   - outline "P_OUTLINE_EDITING" used for this statement
```

Once you're satisfied with the private stored outline, you can publish it as a public stored outline with the following SQL statement:

```
SQL> CREATE PUBLIC OUTLINE outline_editing FROM PRIVATE p_outline_editing;
```

### Dropping Stored Outlines

With the DROP OUTLINE statement or the drop_by_cat procedure in the dbms_outln package, you're able to drop stored outlines. While the former drops a single stored outline, the latter drops all stored outlines belonging to one category:

```
DROP OUTLINE outline_from_text

execute dbms_outln.drop_by_cat(cat => 'TEST')
```

To drop private stored outlines, you have to use the DROP PRIVATE OUTLINE statement.

### Privileges

The system privileges required to create, alter, and drop a stored outline are create any outline, drop any outline, and alter any outline, respectively. No object privileges exist for stored outlines.

By default, the dbms_outln package is available only to users who either have the dba or execute_catalog_role role. Instead, the dbms_outln_edit package is available to all users (the execute privilege is granted to public).

End users don't require specific privileges to use stored outlines.

■ **Tip** You should never need to log in with the outln account. Therefore, for security reasons, you should either lock it or change its default password. This is especially important because it owns a dangerous system privilege: execute any procedure.

## When to Use It

You should consider using stored outlines in two situations. First, consider using it whenever you're optimizing a specific SQL statement and you can't change it in the application (for example, when adding hints isn't an option). Second, you should consider using it when, for whatever reason, you're experiencing troublesome execution plans instability. Because the aim of stored outlines is to force the query optimizer to choose a specific execution plan for a given SQL statement, use this technique only when you want to explicitly restrict the choice of query optimizer to a single execution plan.

Stored outlines are deprecated in favor of SQL plan management beginning in version 11.1. Therefore, as of version 11.1, it's sensible to use stored outlines in Standard Edition only.

## Pitfalls and Fallacies

Oddly enough, the use_stored_outlines initialization parameter can't be specified in an initialization file (init.ora or spfile.ora). Consequently, the parameter must be set either at the system level every time an instance is bounced, or at the session level every time a session is created. In both cases, you can set the initialization parameter through a database trigger. For example, the following trigger sets the use_stored_outlines initialization parameter only for the user named Joze:

```
CREATE OR REPLACE TRIGGER enable_outlines AFTER LOGON ON joze.SCHEMA
BEGIN
  EXECUTE IMMEDIATE 'ALTER SESSION SET use_stored_outlines = TRUE';
END;
```

Even though a stored outline is applied during the generation of an execution plan, that application doesn't mean that the execution plan intended to be generated is actually chosen by the query optimizer. This caveat can be very confusing. It's all the more so because the output of the `dbms_xplan` package, and the `outline_category` column of the `v$sql` view, show that a stored outline has been used during the parse phase. The following example, which is an excerpt of the output generated by the `outline_unreproducible.sql` script, illustrates:

```
SQL> SELECT * FROM t WHERE n = 1970;

---------------------------------------------------
| Id  | Operation                           | Name |
---------------------------------------------------
|   0 | SELECT STATEMENT                    |      |
|   1 |  TABLE ACCESS BY INDEX ROWID BATCHED| T    |
|*  2 |   INDEX RANGE SCAN                  | I    |
---------------------------------------------------

   2 - access("N"=1970)

Note
-----
   - outline "OUTLINE_UNREPRODUCIBLE" used for this statement

SQL> DROP INDEX i;

SQL> SELECT * FROM t WHERE n = 1970;

----------------------------------
| Id  | Operation         | Name |
----------------------------------
|   0 | SELECT STATEMENT  |      |
|*  1 |  TABLE ACCESS FULL| T    |
----------------------------------

   1 - filter("N"=1970)

Note
-----
   - outline "OUTLINE_UNREPRODUCIBLE " used for this statement
```

One of the most important properties of stored outlines is that they're detached from the code. Nevertheless, that could lead to problems. In fact, because there is no direct reference between the stored outline and the SQL statement, it's possible that a developer will completely ignore the existence of the stored outline. As a result, if the developer modifies the SQL statement in a way that leads to a modification of its signature, the stored outline will no longer be used. Similarly, when you deploy an application that needs some stored outlines to perform correctly, during the database setup you must not forget to install them.

You must be aware that stored outlines aren't dropped when the objects they depend on are dropped. This isn't necessarily a problem, however. For example, if a table or an index needs to be re-created because it must be reorganized or moved, it's a good thing that the stored outlines aren't dropped; otherwise, it would be necessary to re-create them.

Two SQL statements with the same text have the same signature. That is also true even if they reference objects in different schemas. This means that a single stored outline could be used for two tables with the same name but located in a different schema! Once again, you should be very careful, especially if you have a database with multiple copies of the same objects.

Whenever a SQL statement has a stored outline and, at the same time, a SQL profile and/or a SQL plan baseline, the query optimizer uses only the stored outline. Of course, this is the case only when the usage of stored outlines is active.

## SQL Profiles

You can delegate SQL optimization to a component of the query optimizer called the *Automatic Tuning Optimizer*. It might seem strange to delegate this task to the same component that isn't able to find an efficient execution plan in the first place. In reality, the two situations are very different. In fact, in normal circumstances the query optimizer is constrained to generate a sub-optimal execution plan because it must operate very quickly, typically in the subsecond range. Instead, much more time can be given to the Automatic Tuning Optimizer to carry out an efficient execution plan. Further, it can use time-consuming techniques such as what-if analyses and make strong utilization of dynamic sampling techniques to verify its estimations.

The Automatic Tuning Optimizer is exposed through the *SQL Tuning Advisor*. Its aim is to analyze SQL statements and to advise how to enhance their performance by either gathering missing or stale object statistics, creating new indexes, altering the SQL statement, or accepting a SQL profile. The following sections are dedicated to advice related to SQL profiles.

It's essential to understand that SQL profiles, officially, can be generated only through the SQL Tuning Advisor. Nevertheless, as I describe later in this section, you can also create them manually.

### How It Works

The following sections describe what SQL profiles are and how to work with them. It also provides information about their internal workings. To manage them, you can use a graphical interface that is integrated into Enterprise Manager. We won't spend time here looking at that because, in my opinion, if you understand what is going on in the background, you'll have no problem using the graphical interface.

### What Are SQL Profiles?

A SQL profile is an object containing information that helps the query optimizer find an efficient execution plan for a specific SQL statement. It provides information about the execution environment, object statistics, and corrections related to the estimations performed by the query optimizer. One of its main advantages is the ability to influence the query optimizer without modifying the SQL statement or the execution environment of the session executing it. In other words, it's transparent to the application connected to the database engine. To understand how SQL profiles work, let's look at how they're generated and utilized.

Figure 11-4 illustrates the steps carried out during the generation of a SQL profile. Simply put, the user asks the SQL Tuning Advisor to optimize a SQL statement, and when a SQL profile is proposed, he accepts it.

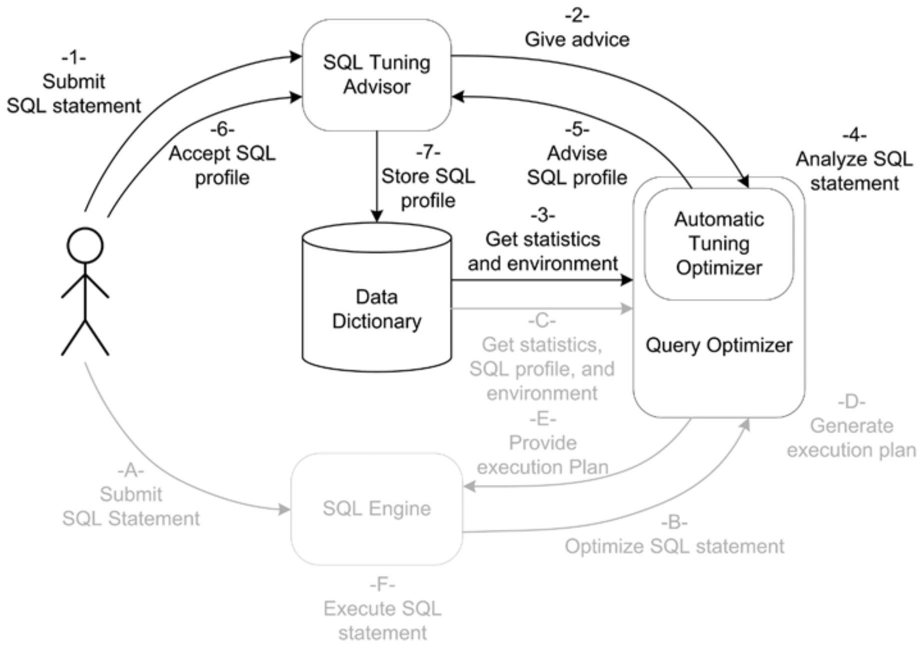


***Figure 11-4.*** *Main steps carried out during the generation of a SQL profile*

Here are the steps in detail:

1. The user passes the poorly performing SQL statement to the SQL Tuning Advisor.
2. The SQL Tuning Advisor asks the Automatic Tuning Optimizer to give advice aimed at optimizing the SQL statement.
3. The query optimizer gets the system statistics, the object statistics related to the objects referenced by the SQL statement, and the initialization parameters that set up the execution environment.
4. The SQL statement is analyzed. During this phase, the Automatic Tuning Optimizer performs its analysis and partially executes the SQL statement to confirm its guesses.
5. The Automatic Tuning Optimizer returns the SQL profile to the SQL Tuning Advisor.
6. The user accepts the SQL profile.
7. The SQL profile is stored in the data dictionary.

Figure 11-5 illustrates the steps carried out during the utilization of a SQL profile. The important thing is that the utilization is completely transparent to the user.

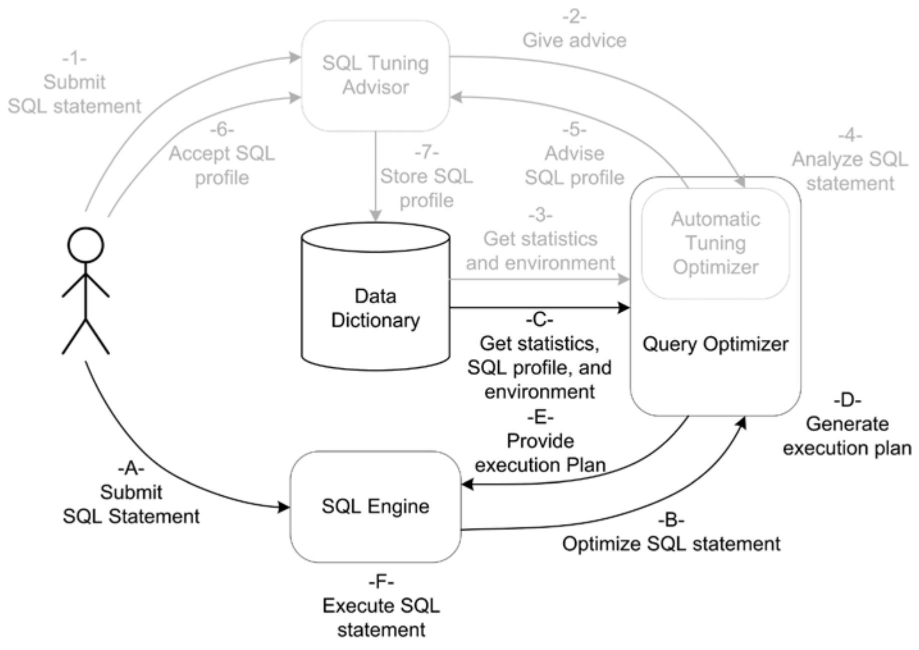


***Figure 11-5.*** *Main steps carried out during the execution of a SQL statement*

Here are the steps in detail:

A. The user sends a SQL statement to be executed to the SQL engine.

B. The SQL engine asks the query optimizer to provide an execution plan.

C. The query optimizer gets the system statistics, the object statistics related to the objects referenced by the SQL statement, the SQL profile, and the initialization parameters that set up the execution environment.

D. The query optimizer analyzes the SQL statement and generates the execution plan.

E. The execution plan is passed to the SQL engine.

F. The SQL engine executes the SQL statement.

The next sections describe in detail the central steps carried out during the generation and utilization of SQL profiles. Special focus is given to the steps that involve the user. Let's start by describing the SQL Tuning Advisor.

## SQL Tuning Advisor

The core interface of the SQL Tuning Advisor is available through the `dbms_sqltune` package. In addition, a graphical interface is integrated in Enterprise Manager. Both interfaces allow you to execute a *tuning task*. They also allow you to review the resulting advice and to accept it. I don't show you here how the graphical user interface works because it's more important to understand what goes on behind the scenes.

■ **Note** To use both the SQL Tuning Advisor and the `dbms_sqltune` package, the Tuning Pack and the Diagnostic Pack must be licensed. Remember, these packs are available only for Enterprise Edition.

To start a tuning task, you have to call the `create_tuning_task` function in the `dbms_sqltune` package and pass as a parameter one of the following (the function is overloaded four times to accept different kinds of parameters):

- The text of a SQL statement
- The reference (`sql_id`) to a SQL statement stored in the library cache
- The reference (`sql_id`) to a SQL statement stored in the Automatic Workload Repository
- The name of a SQL tuning set

## SQL TUNING SETS

Simply put, *SQL tuning sets* are objects that store a set of SQL statements along with their associated execution environments, execution statistics, and, optionally, execution plans. SQL tuning sets are managed with the `dbms_sqltune` package.

To use SQL tuning sets, either the Tuning Pack or Real Application Testing and, therefore, Enterprise Edition, are required.

You can find more information about SQL tuning sets in the *Oracle Database Performance Tuning Guide* manual (up to and including version 11.2), or in the *Oracle Database SQL Tuning Guide* manual (beginning with version 12.1).

To simplify the execution of the `create_tuning_task` function in the `dbms_sqltune` package by taking as a parameter a single SQL statement, I wrote the `tune_last_statement.sql` script. The idea is that you execute the SQL statement that you want to have analyzed in SQL*Plus and then call the script without parameters. The script gets the reference (`sql_id`) of the last SQL statement executed by the current session from the `v$session` view and then creates and executes a tuning task referencing it. The central part of the script is the following anonymous PL/SQL block:

```
DECLARE
  l_sql_id v$session.prev_sql_id%TYPE;
BEGIN
  SELECT prev_sql_id INTO l_sql_id
  FROM v$session
  WHERE audsid = sys_context('userenv','sessionid');

  :tuning_task := dbms_sqltune.create_tuning_task(sql_id => l_sql_id);
  dbms_sqltune.execute_tuning_task(:tuning_task);
END;
```

The tuning task externalizes the output of its analysis in several data dictionary views. Instead of querying the views directly, which is a bit bothersome, you can use the `report_tuning_task` function in the `dbms_sqltune` package to generate a detailed report about the analysis. The following query shows an example of its utilization. Note that to reference the tuning task, the name of the tuning task returned by the previous PL/SQL block is used:

```
SELECT dbms_sqltune.report_tuning_task(:tuning_task)
FROM dual
```

A report generated by the previous query that advises to use a SQL profile looks like the following. Note that this is an excerpt of the output generated by the `profile_opt_estimate.sql` script. The first section shows general information about the analysis and the SQL statement. The second section shows findings and recommendations. In this case, the advice is to accept a SQL profile. The last section shows the execution plans before and after applying the advice:

```
GENERAL INFORMATION SECTION
-------------------------------------------------------------------------------
Tuning Task Name   : TASK_3401
Tuning Task Owner  : CHRIS
Workload Type      : Single SQL Statement
Scope              : COMPREHENSIVE
Time Limit(seconds): 42
Completion Status  : COMPLETED
Started at         : 08/02/2013 15:31:44
Completed at       : 08/02/2013 15:31:45

-------------------------------------------------------------------------------
Schema Name: CHRIS
SQL ID     : bczb6dmm8gcfs
SQL Text   : SELECT * FROM t1, t2 WHERE t1.col1 = 666 AND t1.col2 > 42 AND
             t1.id = t2.id

-------------------------------------------------------------------------------
FINDINGS SECTION (1 finding)
-------------------------------------------------------------------------------

1- SQL Profile Finding (see explain plans section below)
--------------------------------------------------------
  A potentially better execution plan was found for this statement.

  Recommendation (estimated benefit: 65.35%)
  ------------------------------------------
  - Consider accepting the recommended SQL profile.
    execute dbms_sqltune.accept_sql_profile(task_name => 'TASK_3401',
            task_owner => 'CHRIS', replace => TRUE);

-------------------------------------------------------------------------------
EXPLAIN PLANS SECTION
-------------------------------------------------------------------------------

1- Original With Adjusted Cost
------------------------------
Plan hash value: 2452363886
```

```
-------------------------------------------------------------------------------------------------
| Id | Operation                     | Name            | Rows | Bytes | Cost (%CPU)| Time       |
-------------------------------------------------------------------------------------------------
|  0 | SELECT STATEMENT              |                 | 5000 |  9892K|  6210   (1)| 00:01:40 |
|  1 |  NESTED LOOPS                 |                 |      |       |             |          |
|  2 |   NESTED LOOPS                |                 | 5000 |  9892K|  6210   (1)| 00:01:40 |
|  3 |    TABLE ACCESS BY INDEX ROWID| T1              | 9646 |  9542K|  1385   (0)| 00:00:23 |
|* 4 |     INDEX RANGE SCAN          | T1_COL1_COL2_I  | 9500 |       |    27   (0)| 00:00:01 |
|* 5 |    INDEX UNIQUE SCAN          | T2_PK           |    1 |       |     0   (0)| 00:00:01 |
|  6 |   TABLE ACCESS BY INDEX ROWID | T2              |    1 |  1013 |     1   (0)| 00:00:01 |
-------------------------------------------------------------------------------------------------

   4 - access("T1"."COL1"=666 AND "T1"."COL2">42 AND "T1"."COL2" IS NOT NULL)
   5 - access("T1"."ID"="T2"."ID")

2- Using SQL Profile
--------------------
Plan hash value: 2959412835

-----------------------------------------------------------------------------------
| Id | Operation           | Name | Rows | Bytes |TempSpc| Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------
|  0 | SELECT STATEMENT    |      | 5000 |  9892K|       |  1081   (1)| 00:00:18 |
|* 1 |  HASH JOIN          |      | 5000 |  9892K|  5008K|  1081   (1)| 00:00:18 |
|  2 |   TABLE ACCESS FULL| T2   | 5000 |  4946K|       |   174   (0)| 00:00:03 |
|* 3 |   TABLE ACCESS FULL| T1   | 9646 |  9542K|       |   344   (0)| 00:00:06 |
-----------------------------------------------------------------------------------

   1 - access("T1"."ID"="T2"."ID")
   3 - filter("T1"."COL1"=666 AND "T1"."COL2">42)
```

To use the SQL profile recommended by the SQL Tuning Advisor, you have to accept it. The next section describes how you do it. Independently of whether the SQL profile is accepted, once you no longer need the tuning task, you can drop it by calling the `drop_tuning_task` procedure in the `dbms_sqltune` package:

```
dbms_sqltune.drop_tuning_task('TASK_3401');
```

## Accepting SQL Profiles

The `accept_sql_profile` procedure in the `dbms_sqltune` package is used to accept a SQL profile recommended by the SQL Tuning Advisor. It accepts the following parameters:

- The `task_name` and `task_owner` parameters reference the tuning task that advises the SQL profile.
- The `name` and `description` parameters specify a name and a description for the SQL profile itself. In the example, I use the name of the script generating it as the name.
- The `category` parameter is used to group together several SQL profiles for management purposes. It defaults to the value `DEFAULT`.

- The replace parameter specifies whether an already available SQL profile should be replaced. It defaults to FALSE.
- The force_match parameter specifies how text normalization is performed. It defaults to FALSE. The next section provides more information about text normalization.

The only parameter that is mandatory is task_name. For example, to accept the SQL profile recommended in the previous report, you could use the following PL/SQL call:

```
dbms_sqltune.accept_sql_profile(task_name   => 'TASK_3401',
                                task_owner  => user,
                                name        => 'opt_estimate',
                                description => NULL,
                                category    => 'TEST',
                                force_match => TRUE,
                                replace     => TRUE);
```

Once accepted, the SQL profile is stored in the data dictionary. The dba_sql_profiles view displays information about it. In addition, from version 12.1 onward, the cdb_sql_profiles view is also available. Because SQL profiles aren't tied to a specific user, the all_sql_profiles and user_sql_profiles views don't exist:

```
SQL> SELECT category, sql_text, force_matching
  2  FROM dba_sql_profiles
  3  WHERE name = 'opt_estimate';

CATEGORY SQL_TEXT                                      FORCE_MATCHING
-------- --------------------------------------------- --------------
TEST     SELECT * FROM t1, t2 WHERE t1.col1 = 666 AND  YES
         t1.col2 > 42 AND t1.id = t2.id
```

The accept_sql_profile function that has the same purpose as the accept_sql_profile procedure is available as well. The only difference is that the function returns the name of the SQL profile. This is useful if the name isn't specified as an input parameter and the system has to generate it as a result.

## Altering SQL Profiles

You can use the alter_sql_profile procedure in the dbms_sqltune package not only to modify some of the properties specified when the SQL profile was created, but also to change its status (either enabled or disabled). The procedure accepts the following parameters:

- The name parameter identifies the SQL profile to be modified.
- The attribute_name parameter specifies the property to be modified . It accepts the values name, description, category, and status.
- The value parameter specifies the new value of the property.

The three parameters are mandatory. For example, with the following PL/SQL call, the SQL profile created by the previous example is disabled:

```
dbms_sqltune.alter_sql_profile(name           => 'opt_estimate',
                               attribute_name => 'status',
                               value          => 'disabled');
```

## Text Normalization

One of the main advantages of a SQL profile is that although it applies to a specific SQL statement, no modification of the SQL statement itself is needed in order to use it. In fact, the SQL profiles are stored in the data dictionary, and the query optimizer selects them automatically. Figure 11-6 shows what the basic steps carried out during this selection are. First, the SQL statement is normalized to make it not only case insensitive but also independent of the used blank spaces. A signature is computed on the resulting SQL statement. Then, based on that signature, a lookup in the data dictionary is performed. Whenever a SQL profile with the same signature is found, a check is performed to make sure the SQL statement to be optimized and the SQL statement tied to the SQL profile are equivalent. This is necessary because the signature is a hash value, and, therefore, there could be collisions. If the test is successful, the hints associated to the SQL profile are included in the generation of the execution plan.

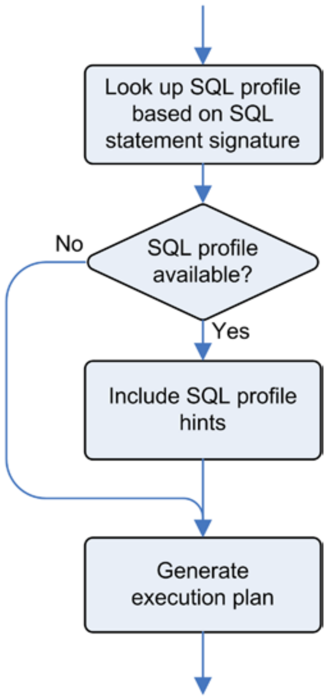

Look up SQL profile based on SQL statement signature
No
SQL profile available?
Yes
Include SQL profile hints
Generate execution plan

***Figure 11-6.*** *Main steps carried out during the selection of a SQL profile*

If the SQL statement contains literals that change, it's likely that the signature, which is a hash value, changes as well. Because of this, the SQL profile may be useless because it's tied to a very specific SQL statement that will possibly never be executed again. To avoid this problem, the database engine is able to remove literals during the normalization phase. This is done by setting the `force_match` parameter to `TRUE` while accepting the SQL profile.

To investigate how text normalization works, you can use the `sqltext_to_signature` function in the `dbms_sqltune` package. It takes two parameters, `sql_text` and `force_match`, as input. The former specifies the SQL statement, and the latter specifies the kind of text normalization to be used. The following excerpt of the output generated by the `profile_signature.sql` script shows the impact of the `force_match` parameter on the signature of different, but similar, SQL statements:

- `force_match` set to `FALSE`: Blank spaces and case-insensitive.

```
SQL_TEXT                                                   SIGNATURE
-------------------------------------------------- --------------------
SELECT * FROM dual WHERE dummy = 'X'                7181225531830258335
select  *  from  dual  where  dummy='X'             7181225531830258335
SELECT * FROM dual WHERE dummy = 'x'               18443846411346672783
SELECT * FROM dual WHERE dummy = 'Y'                 909903071561515954
SELECT * FROM dual WHERE dummy = 'X' OR dummy = :b1 14508885911807130242
SELECT * FROM dual WHERE dummy = 'Y' OR dummy = :b1   816238779370039768
```

- force_match set to TRUE: Blank spaces and case- and literal-insensitive. Nevertheless, the substitution of literals isn't performed if a bind variable is present in the SQL statement.

```
SQL_TEXT                                                   SIGNATURE
-------------------------------------------------- --------------------
SELECT * FROM dual WHERE dummy = 'X'               10668153635715970930
select  *  from  dual  where  dummy='X'            10668153635715970930
SELECT * FROM dual WHERE dummy = 'x'               10668153635715970930
SELECT * FROM dual WHERE dummy = 'Y'               10668153635715970930
SELECT * FROM dual WHERE dummy = 'X' OR dummy = :b1 14508885911807130242
SELECT * FROM dual WHERE dummy = 'Y' OR dummy = :b1   816238779370039768
```

Be aware that it's possible to have two SQL profiles for the same SQL statement: one with force_match set to FALSE, and the other with force_match set to TRUE. If two SQL profiles exist, the one with force_match set to FALSE takes precedence over the one with force_match set to TRUE. This is sound because the one with force_match set to FALSE is more specific than the other. That means that you might have one SQL profile to cover most of the literals, and another for literals requiring particular handling (for example, when a restriction based on a literal is applied to a column containing skewed data).

## Activating SQL Profiles

The activation of SQL profiles is controlled at the system and session levels by the sqltune_category initialization parameter. Its default value is DEFAULT. This is the same default value as for the category parameter of the accept_sql_profile procedure of the dbms_sqltune package. As a result, if no category is specified when accepting the SQL profile, by default the SQL profile is activated. It takes as a value the name of a category specified while accepting the SQL profile. For example, the following SQL statement activates the SQL profiles belonging to the test category at the session level:

```
ALTER SESSION SET sqltune_category = test
```

The initialization parameter supports a single category. Obviously, this implies that a session is able to activate only a single category at a given time.

In order to know whether a SQL profile is used by the query optimizer, you can take advantage of the functions available in the dbms_xplan package. As shown in the following example, the Note section of their output explicitly provides the needed information:

```
SQL> EXPLAIN PLAN FOR SELECT * FROM t ORDER BY id;

SQL> SELECT * FROM table(dbms_xplan.display);
```

```
--------------------------------------------
| Id  | Operation                   | Name |
--------------------------------------------
|   0 | SELECT STATEMENT            |      |
|   1 |  TABLE ACCESS BY INDEX ROWID| T    |
|   2 |   INDEX FULL SCAN           | T_PK |
--------------------------------------------

Note
-----
   - SQL profile "import_sql_profile" used for this statement
```

For a cursor stored in the library cache, the `sql_profile` column of the `v$sql` view shows the name of the SQL profile that was used during the generation of that cursor's execution plan. The column is set to `NULL` when no SQL profile was used.

## Moving SQL Profiles

The `dbms_sqltune` package provides several procedures to move SQL profiles between databases. As shown in Figure 11-7, the following features are provided:

- You can create a staging table through the `create_stgtab_sqlprof` procedure.
- You can copy a SQL profile from the data dictionary to the staging table through the `pack_stgtab_sqlprof` procedure.
- You can change the name and the category of a SQL profile stored in the staging table through the `remap_stgtab_sqlprof` procedure.
- You can copy a SQL profile from the staging table into the data dictionary through the `unpack_stgtab_sqlprof` procedure.

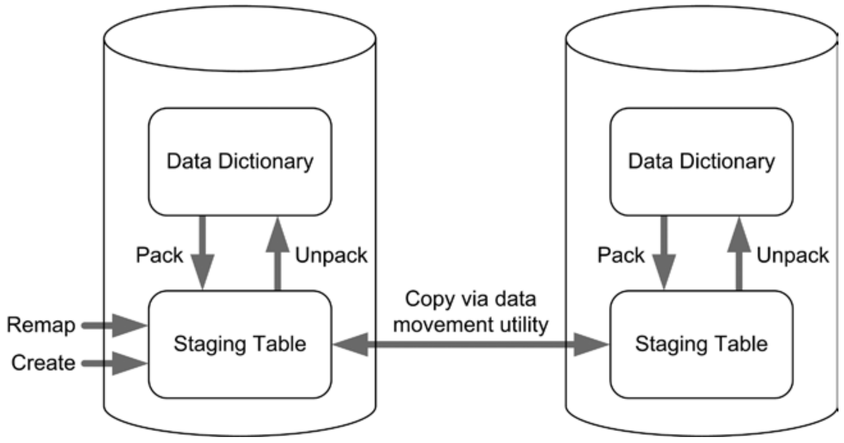


***Figure 11-7.*** *Moving SQL profiles with the `dbms_sqltune` package*

Note that moving the staging table between databases is performed by means of a data movement technique (for example, Data Pump or the legacy export and import utilities) and not with the `dbms_sqltune` package itself.

The following example, an excerpt of the profile_cloning.sql script, shows how to clone a SQL profile inside a single database. First, the mystgtab staging table is created in the current schema:

```
dbms_sqltune.create_stgtab_sqlprof(table_name      => 'MYSTGTAB',
                                   schema_name     => user,
                                   tablespace_name => 'USERS');
```

Then, a SQL profile named opt_estimate is copied from the data dictionary to the staging table:

```
dbms_sqltune.pack_stgtab_sqlprof(profile_name         => 'opt_estimate',
                                 profile_category     => 'TEST',
                                 staging_table_name   => 'MYSTGTAB',
                                 staging_schema_owner => user);
```

The name of the SQL profile must be changed before copying it back into the data dictionary. At the same time, its category is changed as well:

```
dbms_sqltune.remap_stgtab_sqlprof(old_profile_name     => 'opt_estimate',
                                  new_profile_name     => 'opt_estimate_clone',
                                  new_profile_category => 'TEST_CLONE',
                                  staging_table_name   => 'MYSTGTAB',
                                  staging_schema_owner => user);
```

Finally, the SQL profile is copied from the staging table into the data dictionary. Because the parameter replace is set to TRUE, a SQL profile with the same name would be overwritten:

```
dbms_sqltune.unpack_stgtab_sqlprof(profile_name         => 'opt_estimate_clone',
                                   profile_category     => 'TEST_CLONE',
                                   replace              => TRUE,
                                   staging_table_name   => 'MYSTGTAB',
                                   staging_schema_owner => user);
```

## Dropping SQL Profiles

You can use the drop_sql_profile procedure in the dbms_sqltune package to drop a SQL profile from the data dictionary. The name parameter specifies the name of the SQL profile. The ignore parameter specifies whether an error is raised in case the SQL profile doesn't exist. It defaults to FALSE:

```
dbms_sqltune.drop_sql_profile(name   => 'opt_estimate',
                              ignore => TRUE);
```

## Privileges

To create, alter, and drop a SQL profile, the create any sql profile, drop any sql profile, and alter any sql profile system privileges are required, respectively. However, as of version 11.1, these three system privileges are deprecated in favor of the administer sql management system privilege object. No object privileges for SQL profiles exist. To use the SQL Tuning Advisor, the advisor system privilege is required.

End users don't require specific privileges to use SQL profiles.

## Undocumented Features

How does a SQL profile influence the query optimizer? Oracle provides no real answer to this question in its documentation. It's my belief that the best way to use a feature efficiently is to know how it works. So, let's take a look under the hood. Simply put, a SQL profile stores a set of hints representing the adjustments to be performed by the query optimizer. Some of these hints are documented and used in other contexts. Others are undocumented and commonly used for SQL profiles only. In other words, they have probably been implemented for this purpose. All of them are regular hints and, therefore, can be directly added to a SQL statement as well.

Before discussing how to query the list of hints associated to a SQL profile, let's introduce an example based on the `profile_all_rows.sql` script. Its purpose is to show you that, with a SQL profile, it's possible to instruct the query optimizer to change the optimizer mode. In this specific case, changing the optimizer mode is required because a query contains the `rule` hint that forces the query optimizer to work in rule-based mode. The query and its execution plan are the following:

```
SQL> SELECT /*+ rule */ * FROM t ORDER BY id;

--------------------------------------------
| Id  | Operation                    | Name |
--------------------------------------------
|   0 | SELECT STATEMENT             |      |
|   1 |  TABLE ACCESS BY INDEX ROWID| T    |
|   2 |   INDEX FULL SCAN            | T_PK |
--------------------------------------------

Note
-----
   - rule based optimizer used (consider using cbo)
```

After letting the SQL Tuning Advisor work on the query and accepting the SQL profile it advises, the execution plan changes as follows. As pointed out in the `Note` section, a SQL profile is used during the generation of the execution plan:

```
SQL> SELECT /*+ rule */ * FROM t ORDER BY id;

-----------------------------------------------------------------------------------
| Id  | Operation          | Name | Rows  | Bytes |TempSpc| Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |      | 10000 |  1015K|       |   277   (1)| 00:00:04 |
|   1 |  SORT ORDER BY     |      | 10000 |  1015K|  1120K|   277   (1)| 00:00:04 |
|   2 |   TABLE ACCESS FULL| T    | 10000 |  1015K|       |    38   (0)| 00:00:01 |
-----------------------------------------------------------------------------------

Note
-----
   - SQL profile "all_rows" used for this statement
```

Unfortunately, the hints associated with a SQL profile can't be displayed through a data dictionary view. In fact, the only views providing information about SQL profiles, `dba_sql_profiles` and, in a 12.1 multitenant environment, `cdb_sql_profiles`, gives all information except for hints. If you want to know which hints are used for a SQL profile, you have two possibilities. The first is to directly query internal data dictionary tables. The following queries show how to do that for the SQL profile created by the `profile_all_rows.sql` script. Notice that two initialization parameter hints (`all_rows` and `optimizer_features_enable`) are used. In addition, to instruct the query optimizer to ignore the hints present in the SQL statement (in this case the `rule` hint), the `ignore_optim_embedded_hints` hint is used.

- This query works in version 10.2:

```
SQL> SELECT attr_val
  2  FROM sys.sqlprof$ p, sys.sqlprof$attr a
  3  WHERE p.sp_name = 'all_rows'
  4  AND p.signature = a.signature
  5  AND p.category = a.category;

ATTR_VAL
----------------------------------
ALL_ROWS
OPTIMIZER_FEATURES_ENABLE(default)
IGNORE_OPTIM_EMBEDDED_HINTS
```

- This query works as of version 11.1:

```
SQL> SELECT extractValue(value(h),'.') AS hint
  2  FROM sys.sqlobj$data od, sys.sqlobj$ so,
  3       table(xmlsequence(extract(xmltype(od.comp_data),'/outline_data/hint'))) h
  4  WHERE so.name = 'all_rows'
  5  AND so.signature = od.signature
  6  AND so.category = od.category
  7  AND so.obj_type = od.obj_type
  8  AND so.plan_id = od.plan_id;

HINT
----------------------------------
ALL_ROWS
OPTIMIZER_FEATURES_ENABLE(default)
IGNORE_OPTIM_EMBEDDED_HINTS
```

The second possibility is to move the SQL profile into a staging table, as described in the "Moving SQL Profiles" section. Then, with a query like the following, you can get the hints from the staging table. Note that the unnesting through the `table` function is performed because the hints are stored in a varray of `VARCHAR2`:

```
SQL> SELECT *
  2  FROM table(SELECT attributes
  3             FROM mystgtab
  4             WHERE profile_name = 'opt_estimate');
```

```
COLUMN_VALUE
----------------------------------
ALL_ROWS
OPTIMIZER_FEATURES_ENABLE(default)
IGNORE_OPTIM_EMBEDDED_HINTS
```

Switching the optimizer mode isn't the only thing SQL profiles can do. Actually, they were introduced for correcting wrong cardinality estimations performed by the query optimizer. The `profile_opt_estimate.sql` script shows such a case. By using the technique described in Chapter 10 for recognizing wrong estimations, you can see in the following example that the estimated cardinality (`E-Rows`) of several operations is completely different from the actual cardinality (`A-Rows`):

```
----------------------------------------------------------------------------------
| Id  | Operation                     | Name          | Starts | E-Rows | A-Rows |
----------------------------------------------------------------------------------
|   0 | SELECT STATEMENT              |               |      1 |        |   4750 |
|   1 |  NESTED LOOPS                 |               |      1 |        |   4750 |
|   2 |   NESTED LOOPS                |               |      1 |     20 |   4750 |
|   3 |    TABLE ACCESS BY INDEX ROWID| T1            |      1 |     20 |   9500 |
|*  4 |     INDEX RANGE SCAN          | T1_COL1_COL2_I |      1 |     20 |   9500 |
|*  5 |    INDEX UNIQUE SCAN          | T2_PK         |   9500 |      1 |   4750 |
|   6 |   TABLE ACCESS BY INDEX ROWID | T2            |   4750 |      1 |   4750 |
----------------------------------------------------------------------------------
```

If you let the SQL Tuning Advisor analyze such a case and you accept its advice, as the `profile_opt_estimate.sql` script does, a SQL profile containing the following hints is created:

```
OPT_ESTIMATE(@"SEL$1", INDEX_SCAN, "T1"@"SEL$1", "T1_COL1_COL2_I", SCALE_ROWS=477.9096254)
OPT_ESTIMATE(@"SEL$1", NLJ_INDEX_SCAN, "T2"@"SEL$1", ("T1"@"SEL$1"), "T2_PK",
             SCALE_ROWS=0.4814075109)
OPT_ESTIMATE(@"SEL$1", NLJ_INDEX_FILTER, "T2"@"SEL$1", ("T1"@"SEL$1"), "T2_PK",
             SCALE_ROWS=0.4814075109)
OPT_ESTIMATE(@"SEL$1", TABLE, "T1"@"SEL$1", SCALE_ROWS=486.2776343)
OPTIMIZER_FEATURES_ENABLE(default)
```

The important thing to note is the presence of the `opt_estimate` undocumented hint. With that particular hint, it's possible to inform the query optimizer that some of its estimations are wrong and by how much. For example, the first hint tells the query optimizer to scale up the estimation of the operation that accesses the `t1` table by about 478 times ("about" because the denominator in 9500/20 was rounded off in the `dbms_xplan` output).

With the SQL profile in place, the estimations are precise. Also note that the query optimizer chose another execution plan, which was the purpose of creating a SQL profile in the first place:

```
--------------------------------------------------------------
| Id  | Operation          | Name | Starts | E-Rows | A-Rows |
--------------------------------------------------------------
|   0 | SELECT STATEMENT   |      |      1 |        |   4750 |
|*  1 |  HASH JOIN         |      |      1 |   5000 |   4750 |
|   2 |   TABLE ACCESS FULL| T2   |      1 |   5000 |   5000 |
|*  3 |   TABLE ACCESS FULL| T1   |      1 |   9666 |   9500 |
--------------------------------------------------------------
```

Another possible utilization of a SQL profile is when there are objects that have inaccurate or missing object statistics. Of course, that shouldn't happen, but when it does and dynamic sampling can't be used to provide the query optimizer with the needed information, a SQL profile could be used. The `profile_object_stats.sql` script provides an example of this. The hints making up the SQL profile generated by that script are, among others, the following. As the name of the hints suggest, the aim of each hint is to provide object statistics either for a table, an index, or a column:

```
TABLE_STATS("CHRIS"."T2", scale, blocks=735 rows=5000)
INDEX_STATS("CHRIS"."T2", "T2_PK", scale, blocks=14 index_rows=5000)
COLUMN_STATS("CHRIS"."T2", "PAD", scale, length=1000)
COLUMN_STATS("CHRIS"."T2", "COL2", scale, length=3)
COLUMN_STATS("CHRIS"."T2", "COL1", scale, length=3)
COLUMN_STATS("CHRIS"."T2", "ID", scale, length=3 distinct=5000 nulls=0 min=2 max=10000)
```

The last area I describe in this section about undocumented features is the possibility to manually create a SQL profile. In other words, instead of asking the SQL Tuning Advisor to do an analysis and then, if advised, accept a SQL profile, you can build a SQL profile yourself. A manually produced SQL profile is created by calling the `import_sql_profile` procedure in the `dbms_sqltune` package. The following call is an example based on the `profile_import.sql` script. The `sql_text` parameter specifies the SQL statement that the SQL profile is tied to, and the `profile` parameter specifies the list of hints. All other parameters have the same meaning as the parameters of the `accept_sql_profile` procedure previously described:

```
dbms_sqltune.import_sql_profile(
  name        => 'import_sql_profile',
  description => 'SQL profile created manually',
  category    => 'TEST',
  sql_text    => 'SELECT * FROM t ORDER BY id',
  profile     => sqlprof_attr('first_rows(42)','optimizer_features_enable(default)'),
  replace     => FALSE,
  force_match => FALSE
);
```

---

■ **Note** Even though the `import_sql_profile` procedure in the `dbms_sqltune` package isn't officially documented, the method used to create the SQL profile is the same as the one used by the database engine when you accept advice provided by the SQL Tuning Advisor. Hence, I see no problem in using the `import_sql_profile` procedure. In addition, the `coe_xfr_sql_profile.sql` script distributed through the Oracle Support note *SQLT (SQLTXPLAIN) - Tool that helps to diagnose a SQL statement performing poorly or one that produces wrong results* (215187.1) uses this same procedure to create a SQL profile. By the way, you can execute the `coe_xfr_sql_profile.sql` script to create a SQL profile for a cursor cached in the library cache or stored in AWR.

---

## When to Use It

You should consider using SQL profiles whenever you're optimizing a specific SQL statement and you aren't able to change it in the application (for example, when adding hints isn't an option). Remember that the aim of SQL profiles is to provide the query optimizer with additional information about the data to be processed and about the execution environment. So, don't use this technique if you need to force a specific execution plan for a specific SQL statement.

For that purpose, you should use either stored outlines or SQL plan management. The only exception is when you want to take advantage of the text normalization feature related to the `force_match` parameter. In fact, neither stored outlines nor SQL plan management offer a similar feature.

When the `control_management_pack_access` initialization parameter is set to `none` or `diagnostic`, it's not possible to use the SQL Tuning Advisor. If you try to do so, the database engine raises an `ORA-13717: Tuning Package License is needed for using this feature` error. In addition, existing SQL profiles are ignored by the query optimizer.

## Pitfalls and Fallacies

One of the most important properties of SQL profiles is that they're detached from the code. Nevertheless, that could lead to problems. In fact, because there's no direct reference between the SQL profile and the SQL statement, it's possible that a developer will completely ignore the existence of the SQL profile. As a result, if the developer modifies the SQL statement in a way that leads to a modification of its signature, the SQL profile will no longer be used. Similarly, when you deploy an application that needs some SQL profiles to perform correctly, you must not forget to install them during the database setup.

If you have to generate a SQL profile, it's good practice to do it in the production environment (if available) and then to move it to another environment for the necessary tests. The problem is that before moving a SQL profile, you have to accept it. Because you don't want to enable it in production without having tested it, you should make sure to accept it by using a category that's different from the one activated through the `sqltune_category` initialization parameter. In that way, the SQL profile won't be used in the production database. In any case, it's always possible to change the category of a SQL profile later.

You must be aware that SQL profiles aren't dropped when the objects they depend on are dropped. This isn't necessarily a problem, though. For example, if a table or an index needs to be re-created because it must be reorganized or moved, it's a good thing that the SQL profiles aren't dropped; otherwise, it would be necessary to re-create them.

Two SQL statements with the same text have the same signature. This is also true even if they reference objects in different schemas. This means that a single SQL profile could be used for two tables that have the same name but are located in different schemas! You should be very careful, especially if you have a database with multiple copies of the same objects.

Because of the bug described in the Oracle Support note *SQL profile not used in the Active Physical Standby* (10050057.8), up to an including 11.2.0.2 the use of SQL profiles is restricted on Active Data Guard environments. You can use them on primary instances, but not always on standby instances.

Whenever a SQL statement has a SQL profile and a stored outline, the query optimizer uses only the stored outline. Of course, this is the case only when the usage of stored outlines is active.

Whenever a SQL statement has a SQL profile and a SQL plan baseline, the query optimizer tries to merge the hints associated to the SQL profile with those associated to the SQL plan baseline. However, merging a SQL profile with a SQL plan baseline is of limited use. In fact, as described in the next section, the aim of a SQL plan baseline is to force a specific execution plan to be used. As a result, the SQL profile might only be useful to generate a new nonaccepted execution plan before taking the SQL plan baseline into consideration.

## SQL Plan Management

From version 11.1 onward, SQL Plan Management (SPM) replaces stored outlines. Actually, it can be considered an enhanced version of stored outlines. In fact, not only does it share several characteristics with them, but SQL Plan Management shares the same design goal of providing stable execution plans in case of changes in the execution environment or object statistics. In addition, as with stored outlines, SQL Plan Management can be used to optimize an application without modifying it.

■ **Note** The only usage for SQL Plan Management mentioned in Oracle documentation is the stabilization of execution plans. For some reasons I ignore, the possibility of using SQL Plan Management to change the current execution plan (related to a given SQL statement) without modifying the application submitting that SQL statement, isn't mentioned.

Here are the key elements SQL Plan Management consists of:

*SQL plan baselines:* The actual objects that are used to make execution plans stable.

*Statement log:* A list of SQL statements that were executed in the past.

*SQL Management Base (SMB):* Where the SQL plan baselines and the statement log are stored. The required space is allocated in the `sysaux` tablespace.

## How It Works

The following sections describe how SQL plan management works. Specifically, they cover what SQL plan baselines are and how to manage them. To manage them, a graphical interface is integrated into Enterprise Manager. I don't spend time here looking at it, because in my opinion, if you understand what is going on in the background, you will have no problem at all in using the graphical interface.

### What Are SQL Plan Baselines?

A SQL plan baseline is an object associated with a SQL statement that's designed to influence the query optimizer while it generates execution plans. More concretely, a SQL baseline contains, among other things, one or more execution plans which in turn contain a set of hints. Basically a SQL plan baseline is used to force the query optimizer to consistently generate specific execution plans for a given SQL statement.

■ **Caution** Not all hints can be stored in SQL plan baselines. To know which hint can't be stored, you can run the following query:

```
SELECT name FROM v$sql_hint WHERE version_outline IS NULL
```

Even though most of the hints that can't be stored in SQL plan baselines don't impact execution plans (for example, `gather_plan_statistics`), some of them do (for example, `materialize` and `inline`). As a result, there are some execution plans that can't be forced through a SQL plan baseline without specifying a hint in the SQL statement itself.

One of the advantages of a SQL plan baseline is that it applies to a specific SQL statement, and yet no modification of the SQL statement itself is needed. In fact, the SQL plan baselines are stored in the SQL Management Base, and the query optimizer selects them automatically. Figure 11-8 shows the basic steps carried out during this selection. These are:

1. First, the SQL statement is parsed in the conventional way. In other words, the query optimizer generates an execution plan without the support of a SQL plan baseline.
2. Then, the query optimizer normalizes the SQL statement to make it both case-insensitive and independent of the blank spaces present in the text. The signature of the resulting SQL statement is computed, and a lookup into the SQL Management Base is performed. If a SQL plan baseline with the same signature is found, a check is performed to make sure that the SQL statement to be optimized and the SQL statement associated with the SQL plan baseline are equivalent. This check is necessary because the signature is a hash value, and consequently, there can be conflicts.

3. When the test is successful, the query optimizer verifies whether the SQL plan baseline contains the execution plan generated without the SQL plan baseline. If it's contained within it and accepted (trusted), it's executed.

4. If another accepted execution plan is stored in the SQL plan baseline, the hints associated to it are used for the generation of another execution plan. Note that if the SQL plan baseline contains several accepted execution plans, the query optimizer selects the one with the lowest cost.

5. Lastly, the query optimizer checks whether the execution plan generated with the information provided by the SQL plan baseline reproduced the expected execution plan. Only if this last check is fulfilled, the execution plan can be used. If it's not satisfied, the query optimizer tries the other accepted execution plans or, if all of them are unreproducible, it falls back to the execution plan generated without the SQL plan baseline.

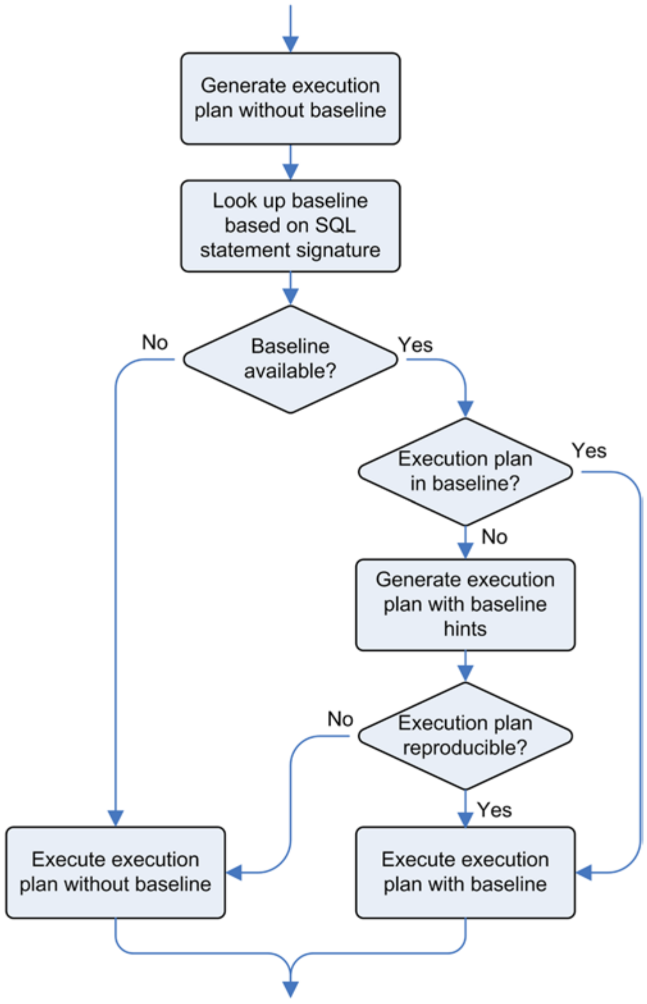

Generate execution plan without baseline
Look up baseline based on SQL statement signature
Baseline available?
No
Yes
Execution plan in baseline?
Yes
No
Generate execution plan with baseline hints
Execution plan reproducible?
No
Yes
Execute execution plan without baseline
Execute execution plan with baseline

***Figure 11-8.*** *Main steps carried out during the selection of a SQL plan baseline*

## Capturing SQL Plan Baselines

You can capture new SQL plan baselines in several ways. Basically, they're created automatically by the database engine or manually by database administrators or developers. The next three sections describe three of these methods.

### Automatic Capture

When the `optimizer_capture_sql_plan_baselines` initialization parameter is set to `TRUE`, the query optimizer automatically stores new SQL plan baselines. By default, the initialization parameter is set to `FALSE`. You can change it at the session and system levels.

When the automatic capture is enabled, the query optimizer stores a new SQL plan baseline for each SQL statement that is executed repeatedly (that is, executed at least twice). To that end, it manages a log in the SQL Management Base where it inserts the signature of each SQL statement it works on. This means that the first time a specific SQL statement is executed, its signature is inserted only into the log. Then, when the same SQL statement is executed for the second time, a SQL plan baseline containing only the current execution plan is created and marked as accepted. From the third execution on, because a SQL plan baseline is already associated with the SQL statement, the query optimizer also compares the current execution plan with the execution plan generated with the help of the SQL plan baseline. If they don't match, it means that according to the current query optimizer estimations, the optimal execution plan isn't the one stored in the SQL plan baseline. To save that information, the current execution plan is added to the SQL plan baseline and marked as nonaccepted. As you've seen before, however, the current execution plan can't be used. The query optimizer is forced to use the execution plan generated with the help of the SQL plan baseline. Figure 11-9 summarizes the whole process.

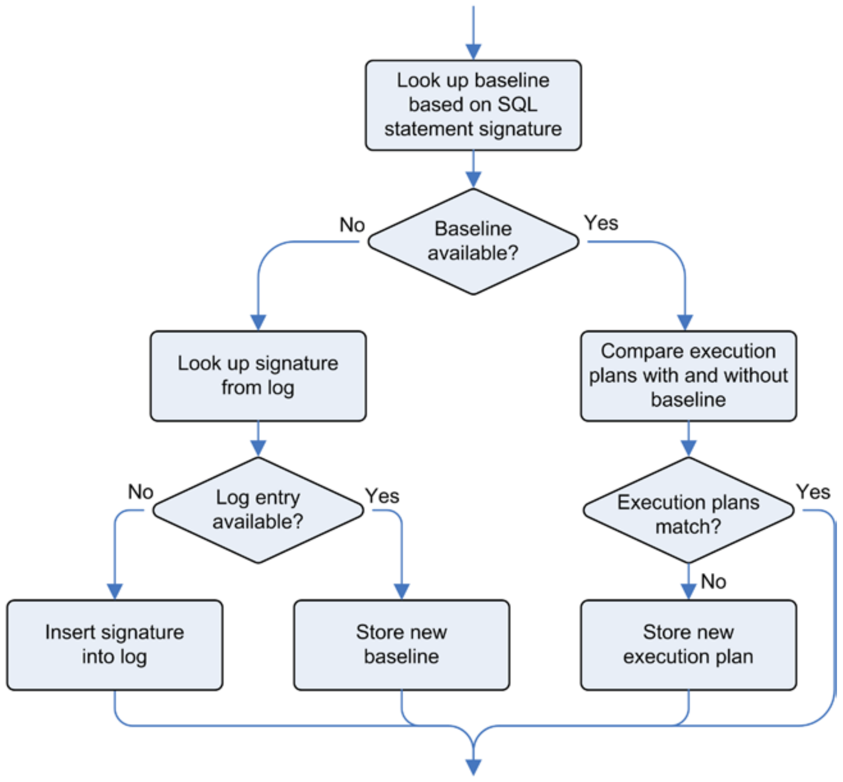

Look up baseline based on SQL statement signature
No
Baseline available?
Yes
Look up signature from log
Compare execution plans with and without baseline
No
Log entry available?
Yes
Execution plans match?
Yes
No
Insert signature into log
Store new baseline
Store new execution plan

***Figure 11-9.*** *Main steps carried out during the automatic capture of a SQL plan baseline*

When a new execution plan is stored in the SQL plan baseline, it's important to distinguish between two situations:

- If it's the first execution plan of the SQL plan baseline, the execution plan is stored as accepted, and consequently, the query optimizer will be able to use it.
- If it's not the first execution plan of the SQL plan baseline, it's stored as nonaccepted, and as a result, the query optimizer won't be able to use it. The "Evolving SQL Plan Baselines" section describes how to validate a SQL plan baseline to make it available to the query optimizer.

## Load from Library Cache

To manually load SQL plan baselines into the data dictionary based on cursors stored in the library cache, the `load_plans_from_cursor_cache` function in the `dbms_spm` package is available.

Actually, the function is overloaded several times to support different methods that identify which cursors have to be processed. There re two main possibilities. First, identify several SQL statements by specifying one of the following attributes:

- `sql_text`: Text of the SQL statement. Wildcards (for example, `%`) are supported with this attribute.
- `parsing_schema_name`: Schema name that was used to parse the cursor.
- `module`: Name of the module that executed the SQL statement.
- `action`: Name of the action that executed the SQL statement.

To illustrate, the following call, an excerpt of the `baseline_from_sqlarea1.sql` script, creates a SQL plan baseline for each SQL statement stored in the library cache that contains the `MySqlStm` string as a comment in its text:

```
ret := dbms_spm.load_plans_from_cursor_cache(attribute_name  => 'sql_text',
                                             attribute_value => '%/* MySqlStm */%');
```

Second, identify a single SQL statement by its SQL ID and, optionally, the hash value of the execution plan. If the hash value isn't specified or set to `NULL`, all execution plans available for the specified SQL statement are loaded. The following call, an excerpt of the `baseline_from_sqlarea2.sql` script, illustrates this:

```
ret := dbms_spm.load_plans_from_cursor_cache(sql_id          => '2y5r75r8y3sj0',
                                             plan_hash_value => NULL);
```

Execution plans loaded with these functions are stored as accepted, and so the query optimizer might immediately take advantage of them.

In the previous examples, the SQL plan baselines are based on the text of the SQL statement found in the library cache. This is relevant only if you want to ensure that the current execution plan will also be used in the future. Sometimes, the purpose of using a SQL plan baseline is to optimize a SQL statement without modifying the application. Let's look at an example of such a situation, based on the `baseline_from_sqlarea3.sql` script.

Let's say one of your applications executes the following SQL statement. The execution plan generated by the query optimizer is based on a full table scan. This is because the SQL statement contains a hint forcing the query optimizer toward this operation:

```
SQL> SELECT /*+ full(t) */ count(pad) FROM t WHERE n = 42;

SQL> SELECT * FROM table(dbms_xplan.display_cursor);
```

```
-----------------------------------
| Id  | Operation          | Name |
-----------------------------------
|   0 | SELECT STATEMENT   |      |
|   1 |  SORT AGGREGATE    |      |
|*  2 |   TABLE ACCESS FULL| T    |
-----------------------------------

   2 - filter("N"=42)
```

You notice that the column on which the restriction is applied (n) is indexed. You then wonder what the performance is when the index is used. So, as shown in the following example, you execute the SQL statement by specifying a hint to ensure that the index is used:

```
SQL> SELECT /*+ index(t) */ count(pad) FROM t WHERE n = 42;

SQL> SELECT * FROM table(dbms_xplan.display_cursor);

SQL_ID  dat4n4845zdxc, child number 0
-------------------------------------

Plan hash value: 3694077449

---------------------------------------------
| Id  | Operation                    | Name |
---------------------------------------------
|   0 | SELECT STATEMENT             |      |
|   1 |  SORT AGGREGATE              |      |
|   2 |   TABLE ACCESS BY INDEX ROWID| T    |
|*  3 |    INDEX RANGE SCAN          | I    |
---------------------------------------------

   3 - access("N"=42)
```

If the second execution plan is more efficient than the first one, your objective is to let the application use it. If you can't change the application in order to remove or modify the hint, you can take advantage of a SQL plan baseline to solve this problem. To do that, you could create a SQL plan baseline either automatically or manually, as described earlier. In this case, you decide to use the optimizer_capture_sql_plan_baselines initialization parameter:

```
SQL> ALTER SESSION SET optimizer_capture_sql_plan_baselines = TRUE;

SQL> SELECT /*+ full(t) */ count(pad) FROM t WHERE n = 42;

SQL> ALTER SESSION SET optimizer_capture_sql_plan_baselines = FALSE;
```

Once the SQL plan baseline is created, you check that it's really used. Notice how the dbms_xplan package clearly shows that a SQL plan baseline, identified through a *SQL plan name*, was used to generate the execution plan:

```
SQL> SELECT /*+ full(t) */ count(pad) FROM t WHERE n = 42;

SQL> SELECT * FROM table(dbms_xplan.display_cursor);
```

```
-----------------------------------
| Id  | Operation          | Name |
-----------------------------------
|   0 | SELECT STATEMENT   |      |
|   1 |  SORT AGGREGATE    |      |
|*  2 |   TABLE ACCESS FULL| T    |
-----------------------------------

   2 - filter("N"=42)

Note
-----
   - SQL plan baseline SQL_PLAN_3u6sbgq7v4u8z3fdbb376 used for this statement
```

Then, based on the SQL plan name provided by the previous output, you find the identifier of the SQL plan baseline, the *SQL handle*, through the dba_sql_plan_baselines view:

```
SQL> SELECT sql_handle
  2  FROM dba_sql_plan_baselines
  3  WHERE plan_name = 'SQL_PLAN_3u6sbgq7v4u8z3fdbb376';

SQL_HANDLE
--------------------
SQL_3d1b0b7d8fb2691f
```

Finally, you replace the execution plan used by the SQL plan baseline. To do so, you load the execution plan associated with the SQL statement leading to the index scan and remove the one associated with the full table scan. The former is referenced by the SQL identifier and the execution plan hash value, the latter by the SQL handle and SQL plan name:

```
ret := dbms_spm.load_plans_from_cursor_cache(sql_handle      => 'SQL_3d1b0b7d8fb2691f',
                                             sql_id          => 'dat4n4845zdxc',
                                             plan_hash_value => '3694077449');

ret := dbms_spm.drop_sql_plan_baseline(sql_handle => 'SQL_3d1b0b7d8fb2691f',
                                       plan_name  => 'SQL_PLAN_3u6sbgq7v4u8z3fdbb376');
```

To check whether the replacement has correctly taken place, you test the new SQL plan baseline. Notice that even if the SQL statement contains the full hint, the execution plan no longer uses a full table scan.

■ **Note** Inappropriate hints occur frequently in practice as the reason for inefficient execution plans. Being able to override them with the technique you've seen in this section is extremely useful.

```
SQL> SELECT /*+ full(t) */ count(pad) FROM t WHERE n = 42;

SQL> SELECT * FROM table(dbms_xplan.display_cursor);
```

```
--------------------------------------------
| Id  | Operation                    | Name |
--------------------------------------------
|   0 | SELECT STATEMENT             |      |
|   1 |  SORT AGGREGATE              |      |
|   2 |   TABLE ACCESS BY INDEX ROWID| T    |
|*  3 |    INDEX RANGE SCAN          | I    |
--------------------------------------------

   3 - access("N"=42)

Note
-----
   - SQL plan baseline SQL_PLAN_3u6sbgq7v4u8z59340d78 used for this statement
```

To know whether a SQL plan baseline was used for a specific SQL statement, it's also possible to check the sql_plan_baseline column in the v$sql view. Note that the column shows the SQL plan name, not the SQL handle, as the dbms_xplan package does.

### Load from SQL Tuning Set

To load SQL plan baselines from SQL tuning sets, the load_plans_from_sqlset function in the dbms_spm package is available. Loading is simply a matter of specifying the owner and the name of the SQL tuning set. The following call, an excerpt of the baseline_from_sqlset.sql script, illustrates this:

```
ret := dbms_spm.load_plans_from_sqlset(sqlset_name  => 'test_sqlset',
                                       sqlset_owner => user);
```

Execution plans loaded with this function are stored as accepted. Therefore, the query optimizer is immediately able to take advantage of them.

A possible use of this function is the upgrade to a newer release. In fact, it's also possible, for example, to load in a version 11.2 database a SQL tuning set created with a version 10.2 database. The baseline_upgrade_10g.sql and baseline_upgrade_11g.sql scripts illustrate this utilization.

## Displaying SQL Plan Baselines

General information about the available SQL plan baselines can be displayed through the dba_sql_plan_baselines view (from version 12.1 onward, the cdb_sql_plan_baselines view is also available). To display detailed information about them, the display_sql_plan_baseline function in the dbms_xplan package is available. Note that it works similarly to the other functions in the dbms_xplan package discussed in Chapter 10. The following example shows the kind of information that can be displayed with it:

```
SQL> SELECT *
  2  FROM table(dbms_xplan.display_sql_plan_baseline(sql_handle => 'SQL_971650b23f790eb7'));

--------------------------------------------------------------------------------
SQL handle: SQL_971650b23f790eb7
SQL text: SELECT /* MySqlStm */ count(pad) FROM t WHERE n = 28
--------------------------------------------------------------------------------
```

```
--------------------------------------------------------------------------------
Plan name: SQL_PLAN_9f5khq8zrk3pr3fdbb376         Plan id: 1071362934
Enabled: YES     Fixed: NO      Accepted: YES     Origin: MANUAL-LOAD
--------------------------------------------------------------------------------

Plan hash value: 2966233522

---------------------------------------------------------------------------
| Id  | Operation          | Name | Rows  | Bytes | Cost (%CPU)| Time     |
---------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |      |     1 |   505 |    20   (0)| 00:00:01 |
|   1 |  SORT AGGREGATE    |      |     1 |   505 |            |          |
|*  2 |   TABLE ACCESS FULL| T    |     1 |   505 |    20   (0)| 00:00:01 |
---------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   2 - filter("N"=28)
```

■ **Caution** To correctly display information about a SQL plan baseline in versions 11.1 and 11.2, the `display_sql_plan_baseline` function has to be able to reproduce the execution plans associated to it. If the function doesn't succeed, it might return wrong results or even an error message. To avoid such problems, from version 12.1 onward, the execution plan is stored for reporting purposes, and only for reporting purposes, in the SQL Management Base. You can execute the `baseline_unreproducible.sql` script to observe what the output looks like in the case of an unreproducible execution plan.

Unfortunately, data dictionary tables must be queried in version 11.1 to display the list of hints associated with a SQL plan baseline. The following SQL statement shows an example. Note that because hints are stored in XML format, a conversion is necessary to have a readable output:

```
SQL> SELECT extractValue(value(h),'.') AS hint
  2  FROM sys.sqlobj$data od, sys.sqlobj$ so,
  3       table(xmlsequence(extract(xmltype(od.comp_data),'/outline_data/hint'))) h
  4  WHERE so.name = 'SQL_PLAN_9f5khq8zrk3pr3fdbb376'
  5  AND so.signature = od.signature
  6  AND so.category = od.category
  7  AND so.obj_type = od.obj_type
  8  AND so.plan_id = od.plan_id;

HINT
-------------------------------------
IGNORE_OPTIM_EMBEDDED_HINTS
OPTIMIZER_FEATURES_ENABLE('11.2.0.3')
DB_VERSION('11.2.0.3')
ALL_ROWS
OUTLINE_LEAF(@"SEL$1")
FULL(@"SEL$1" "T"@"SEL$1")
```

However, from version 11.2 onward, it's also possible to display the list of hints with the `display_sql_plan_baseline` function. In fact, as for the other functions in the `dbms_xplan` package, the `format` parameter can be used to influence the output. The following is an excerpt of the output produced when the `format` parameter is set to `outline`:

```
SQL> SELECT *
  2  FROM table(dbms_xplan.display_sql_plan_baseline(sql_handle => 'SQL_971650b23f790eb7',
  3                                                  format     => 'outline'));

Outline Data from SMB:

  /*+
      BEGIN_OUTLINE_DATA
      IGNORE_OPTIM_EMBEDDED_HINTS
      OPTIMIZER_FEATURES_ENABLE('11.2.0.3')
      DB_VERSION('11.2.0.3')
      ALL_ROWS
      OUTLINE_LEAF(@"SEL$1")
      FULL(@"SEL$1" "T"@"SEL$1")
      END_OUTLINE_DATA
  */
```

## Evolving SQL Plan Baselines

When the query optimizer generates an execution plan different from one present in the SQL plan baseline associated to the SQL statement it's optimizing, a new nonaccepted execution plan is automatically added to the SQL plan baseline. This happens even if the query optimizer can't immediately use the nonaccepted execution plan. The idea is to keep the information that another and possibly better execution plan exists. To verify whether one of the nonaccepted execution plans will in fact perform better than the ones generated with the help of accepted SQL plan baselines, an *evolution* must be attempted. This is nothing other than asking the SQL engine to run the SQL statement with different execution plans and finding out whether a nonaccepted SQL plan baseline will lead to better performance than an accepted one. If this is in fact the case, the nonaccepted SQL plan baseline is set to accepted.

---

■ **Caution** The SQL engine processes SQL statements in a special way during an evolution. In fact, for `INSERT`/`UPDATE`/`MERGE`/`DELETE` statements, the data is accessed but not modified. Hence, SQL statements are only partially executed. However, I don't regard this fact as a problem. In fact, the operations that modify the data should always perform the same work independently of how the data to be modified is accessed.

---

To execute an evolution, the `evolve_sql_plan_baseline` function in the `dbms_spm` package is available. To call this function, in addition to identifying the SQL plan baseline with the `sql_handle` and/or `plan_name` parameters, the following parameters can be specified:

- `time_limit`: How long, in minutes, the evolution can last. This parameter accepts either a natural number or the `dbms_spm.auto_limit` and `dbms_spm.no_limit` constants.
- `verify`: If set to `yes` (default), the SQL statement is executed to verify the performance. If set to `no`, no verification is performed, and the SQL plan baselines are simply accepted.
- `commit`: If set to `yes` (default), the data dictionary is modified according to the result of the evolution. If set to `no`, provided the `verify` parameter is set to `yes`, the verification is performed without modifying the data dictionary.

The return value of the function is a report that provides details about the evolution. The following example, an excerpt of the output generated by the `baseline_automatic.sql` script, shows the SQL statement used to start the evolution and the resulting report that points out that a SQL plan baseline was evolved (including the statistics that led to that decision):

```
SQL> SELECT dbms_spm.evolve_sql_plan_baseline(sql_handle => 'SQL_492bdb47e8861a89',
  2                                           plan_name  => '',
  3                                           time_limit => 10,
  4                                           verify     => 'yes',
  5                                           commit     => 'yes')
  6  FROM dual;

-------------------------------------------------------------------------------
                        Evolve SQL Plan Baseline Report
-------------------------------------------------------------------------------

Inputs:
-------
  SQL_HANDLE = SQL_492bdb47e8861a89
  PLAN_NAME  =
  TIME_LIMIT = 10
  VERIFY     = yes
  COMMIT     = yes

Plan: SQL_PLAN_4kayv8zn8c6n959340d78
------------------------------------
  Plan was verified: Time used .05 seconds.
  Plan passed performance criterion: 24.59 times better than baseline plan.
  Plan was changed to an accepted plan.

                            Baseline Plan      Test Plan       Stats Ratio
                            -------------      ---------       -----------
  Execution Status:              COMPLETE       COMPLETE
  Rows Processed:                       1              1
  Elapsed Time(ms):                  .527           .054              9.76
  CPU Time(ms):                      .333           .111                 3
  Buffer Gets:                         74              3             24.67
  Physical Read Requests:               0              0
  Physical Write Requests:              0              0
  Physical Read Bytes:                  0              0
  Physical Write Bytes:                 0              0
  Executions:                           1              1

-------------------------------------------------------------------------------
                                 Report Summary
-------------------------------------------------------------------------------
Number of plans verified: 1
Number of plans accepted: 1
```

In addition to the manual evolution just explained, automatic evolution of SQL plan baselines is supported with the Tuning Pack. The idea is that, during the maintenance window, the SQL Tuning Advisor works on the SQL statements that had a significant impact on the system. When possible, the Tuning Advisor provides recommendations to improve their response time. If the Tuning Advisor notices that a nonaccepted SQL plan baseline leads to better performance than an accepted one, it recommends a SQL profile that says nothing else than to accept the SQL plan baseline. Obviously, if that SQL profile is accepted, then the SQL plan baseline is accepted as well. Therefore, the SQL plan baselines are automatically accepted only if the SQL profiles generated by the SQL Tuning Advisor are also automatically accepted.

It's essential to point out that the SQL profiles are automatically accepted only when the `accept_sql_profiles` parameter, which is specific to the SQL Tuning Advisor, is set to `TRUE`. By default it's set to `FALSE`. You can check its value through the `dba_advisor_parameters` view (note that also the `user` and, as of version 12.1, `cdb` related views exist) with a query like the following:

```
SQL> SELECT parameter_value
  2  FROM dba_advisor_parameters
  3  WHERE task_name = 'SYS_AUTO_SQL_TUNING_TASK'
  4  AND parameter_name = 'ACCEPT_SQL_PROFILES';

PARAMETER_VALUE
---------------
FALSE
```

The `dbms_auto_sqltune` package provides the `set_auto_tuning_task_parameter` procedure to change the value of the `accept_sql_profiles` parameter. The following example shows how to set that parameter to `TRUE` to activate the automatic acceptance of SQL profiles:

```
dbms_auto_sqltune.set_auto_tuning_task_parameter(parameter => 'ACCEPT_SQL_PROFILES',
                                                 value     => 'TRUE');
```

As of version 12.1, there's also a new advisor called *SPM Evolve Advisor*. Its purpose is to execute an evolution for the nonaccepted execution plans associated to SQL plan baselines. It runs during the maintenance window, just as other advisors do. To display what the SPM Evolve Advisor did, you can use the `report_auto_evolve_task` function in the `dbms_spm` package. If you call that function without parameters, it shows a report about the last execution. The following example shows how to find out, through the `dba_advisor_executions` view (note that also `user` and, as of version 12.1, `cdb` related views exist), when the last five executions took place, and how to display the report of one specific execution:

```
SQL> SELECT *
  2  FROM (
  3    SELECT execution_name, execution_start
  4    FROM dba_advisor_executions
  5    WHERE task_name = 'SYS_AUTO_SPM_EVOLVE_TASK'
  6    ORDER BY execution_start DESC
  7  )
  8  WHERE rownum <= 3;

EXECUTION_NAME EXECUTION_START
-------------- ----------------
EXEC_6294      23-APR-14
EXEC_6182      22-APR-14
EXEC_6082      21-APR-14

SQL> SELECT dbms_spm.report_auto_evolve_task(execution_name => 'EXEC_6294')
  2  FROM dual;
```

```
GENERAL INFORMATION SECTION
---------------------------------------------------

 Task Information:
 ---------------------------------------------
 Task Name            : SYS_AUTO_SPM_EVOLVE_TASK
 Task Owner           : SYS
 Description          : Automatic SPM Evolve Task
 Execution Name       : EXEC_6294
 Execution Type       : SPM EVOLVE
 Scope                : COMPREHENSIVE
 Status               : COMPLETED
 Started              : 04/23/2014 22:00:19
 Finished             : 04/23/2014 22:00:19
 Last Updated         : 04/23/2014 22:00:19
 Global Time Limit    : 3600
 Per-Plan Time Limit  : UNUSED
 Number of Errors     : 0
---------------------------------------------------

SUMMARY SECTION
---------------------------------------------------
  Number of plans processed  : 0
  Number of findings         : 0
  Number of recommendations  : 0
  Number of errors           : 0
---------------------------------------------------
```

## Altering SQL Plan Baselines

You can use the alter_sql_plan_baseline procedure in the dbms_spm package to modify some of the properties specified when the SQL plan baseline was created. The sql_handle and plan_name parameters identify the SQL plan baseline to be modified. At least one of the two must be specified. The attribute_name and attribute_value parameters specify the property to be modified and its new value. The attribute_name parameter accepts the following values:

- enabled: This attribute can be set to either yes or no, although a SQL plan baseline can be used by the query optimizer only when set to yes.
- fixed: With this attribute set to yes, no new execution plan is added to a SQL plan baseline and, as a result, it can't be evolved over time. In addition, if a SQL plan baseline contains several accepted execution plans, a fixed execution plan is preferred over a nonfixed one. It can be set to either yes or no.
- autopurge: A SQL plan baseline with this attribute set to yes is automatically removed if it's not used over a retention period (the configuration of the retention is discussed later in the "Dropping SQL Plan Baselines" section). It can be set to either yes or no.
- plan_name: This attribute is used to change the SQL plan name. It can be any string of up to 30 characters.
- description: This attribute is used to attach a description to the SQL plan baseline. It can be any string of up to 500 characters.

In the following call, an execution plan associated to a specific SQL plan baseline is disabled:

```
ret := dbms_spm.alter_sql_plan_baseline(sql_handle      => 'SQL_492bdb47e8861a89',
                                        plan_name       => 'SQL_PLAN_4kayv8zn8c6n93fdbb376',
                                        attribute_name  => 'enabled',
                                        attribute_value => 'no');
```

## Activating SQL Plan Baselines

The query optimizer uses the available SQL plan baselines only when the `optimizer_use_sql_plan_baselines` initialization parameter is set to `TRUE` (this is the default value). You can change it at the session and system levels.

## Moving SQL Plan Baselines

The `dbms_spm` package provides several procedures for moving SQL plan baselines between databases. This is needed when, for example, the SQL plan baselines have to be generated on a development or test database and moved to the production database. As shown in Figure 11-10, the following features are provided:

- You can create a staging table using the `create_stgtab_baseline` procedure.
- You can copy SQL plan baselines from the data dictionary to the staging table through the `pack_stgtab_baseline` function.
- You can copy SQL plan baselines from the staging table into the data dictionary through the `unpack_stgtab_baseline` function.

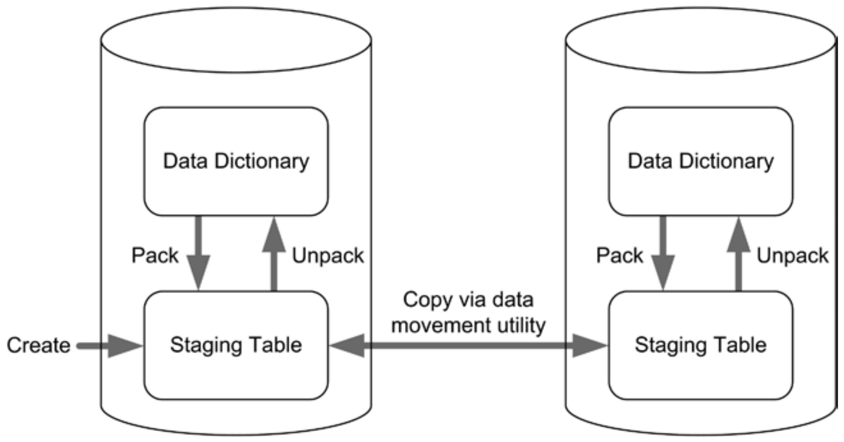


***Figure 11-10.*** *Moving SQL plan baselines with the dbms_spm package*

Notice that moving the staging table between databases is performed by means of any data movement technique (for example, Data Pump or the legacy export and import utilities) and not with the `dbms_spm` package itself (see Figure 11-10).

The following example, an excerpt of the `baseline_clone.sql` script, shows how to copy a SQL plan baseline from one database into another. First, the `mystgtab` staging table is created in the current schema:

```
dbms_spm.create_stgtab_baseline(table_name      => 'MYSTGTAB',
                                table_owner     => user,
                                tablespace_name => 'USERS');
```

Then a SQL plan baseline is copied from the data dictionary into the staging table. You can identify which SQL plan baselines are to be processed in four ways:

- Identify the SQL plan baseline exactly, through the `sql_handle` and, optionally, `plan_name` parameters.
- Select all SQL plan baselines that contain a specific string in the text of the SQL statement associated with them. For this purpose, the `sql_text` parameter, which also supports wildcards (for example, `%`), is available. Note that the parameter is case-sensitive.
- Select all SQL plan baselines matching one or several of the following parameters: `creator`, `origin`, `enabled`, `accepted`, `fixed`, `module`, and `action`. If several parameters are specified, all of them must be fulfilled.
- Processing all SQL plan baselines. For that, no parameters are specified.

The following call shows an example where the SQL plan baseline is identified exactly:

```
ret := dbms_spm.pack_stgtab_baseline(table_name  => 'MYSTGTAB',
                                     table_owner => user,
                                     sql_handle  => 'SQL_492bdb47e8861a89',
                                     plan_name   => 'SQL_PLAN_4kayv8zn8c6n93fdbb376');
```

At this point, the `mystgtab` staging table is copied by means of a data movement utility from one database into another.

Finally, the SQL plan baseline is copied from the staging table into the data dictionary in the target database. To identify which SQL plan baselines are processed, the same methods as for the `pack_stgtab_baseline` function are available. The following call shows an example where the SQL plan baselines are identified by the text of the SQL statement associated with them:

```
ret := dbms_spm.unpack_stgtab_baseline(table_name  => 'MYSTGTAB',
                                       table_owner => user,
                                       sql_text    => '%FROM t%');
```

## Dropping SQL Plan Baselines

You can use the `drop_sql_plan_baseline` procedure in the `dbms_spm` package to drop a SQL plan baseline from the data dictionary. The `sql_handle` and `plan_name` parameters identify the execution plan and/or the SQL plan baseline to be dropped. At least one of the two must be specified. The following call illustrates this:

```
ret := dbms_spm.drop_sql_plan_baseline(sql_handle => 'SQL_492bdb47e8861a89',
                                       plan_name  => 'SQL_PLAN_4kayv8zn8c6n93fdbb376');
```

Unused SQL plan baselines that don't have the fixed attribute set to yes are automatically removed after a retention period. The default retention period is 53 weeks. The current value can be displayed through the dba_sql_management_config view (from version 12.1 onward the cdb version of the view also exists) with a query like the following:

```
SQL> SELECT parameter_value
  2  FROM dba_sql_management_config
  3  WHERE parameter_name = 'PLAN_RETENTION_WEEKS';

PARAMETER_VALUE
---------------
             53
```

You can change the retention period by calling the configure procedure in the dbms_spm package. Values between 5 and 523 weeks are supported. The following example shows how to change it to 12 weeks. If the parameter_value parameter is set to NULL, the default value is restored:

```
dbms_spm.configure(parameter_name  => 'plan_retention_weeks',
                   parameter_value => 12);
```

### Privileges

When SQL plan baselines are automatically captured (that is, by setting the optimizer_capture_sql_plan_baselines initialization parameter to TRUE), no particular privilege is needed to create them.

The dbms_spm package can be executed only by users with the administer sql management object system privilege (the dba role includes it by default). No object privileges exist for SQL plan baselines.

End users don't require specific privileges to use SQL plan baselines.

## When to Use It

You should consider using SQL plan baselines in two situations. First, consider it whenever you're optimizing a specific SQL statement and you can't change it in the application (for example, when adding hints isn't an option). Second, you should consider using it when, for whatever reason, you're experiencing troublesome execution plans instability. Because the aim of SQL plan baselines is to force the query optimizer to choose an execution plan from a limited list of accepted execution plans, use this technique only when you want to explicitly restrict the query optimizer's choice to specific execution plans.

Unfortunately, SQL plan baselines are available only with Enterprise Edition. With Standard Edition, use stored outline instead.

## Pitfalls and Fallacies

One of the most important properties of SQL plan baselines is that they're detached from the code. Nevertheless, that could lead to problems. In fact, because there is no direct reference between the SQL plan baselines and the SQL statement, it's possible that a developer will completely ignore the existence of the SQL plan baseline. As a result, if the developer modifies the SQL statement in a way that leads to a modification of its signature, the SQL plan baseline will no longer be used. Similarly, when you deploy an application that needs some SQL plan baselines to perform correctly, you mustn't forget to install them during the database setup.

You must be aware that SQL plan baselines aren't immediately dropped when the objects they depend on are dropped. This isn't necessarily a problem, though. For example, if a table or an index needs to be re-created because it must be reorganized or moved, it's a good thing that the SQL plan baselines aren't dropped; otherwise, it would be necessary to re-create them. In any case, unused SQL plan baselines will be purged when the retention period ends.

Two SQL statements with the same text have the same signature. This is also true even if they reference objects in different schemas. That means that a single SQL plan baseline could be used for two tables that have the same name but are located in different schemas! You should be very careful, especially if you have a database with multiple copies of the same objects.

SQL plan baselines aren't supported for SQL statements referencing tables stored in remote databases.

Because of the bug described in the Oracle Support note *SQL profile not used in the Active Physical Standby* (10050057.8), up to an including 11.2.0.2, the use of SQL plan baselines is restricted on Active Data Guard environments. You can use them on primary instances, but not always on standby instances.

SQL plan baselines are stored in the SQL Management Base in the `sysaux` tablespace. By default, at most 10 percent of the tablespace can be used for them. The current value can be displayed through the `dba_sql_management_config` view:

```
SQL> SELECT parameter_value
  2  FROM dba_sql_management_config
  3  WHERE parameter_name = 'SPACE_BUDGET_PERCENT';

PARAMETER_VALUE
---------------
             10
```

When the threshold is exceeded, a warning message is written in the alert log. To change the default threshold, the `configure` procedure in the `dbms_spm` package is available. Values between 1 percent and 50 percent are supported. The following example shows how to change it to 5 percent. If the `parameter_value` parameter is set to `NULL`, the default value is restored:

```
dbms_spm.configure(parameter_name  => 'space_budget_percent',
                   parameter_value => 5);
```

Whenever a SQL statement has a SQL plan baseline and a stored outline, the query optimizer uses only the stored outline. Of course, this is the case only when the usage of stored outlines is active.

Whenever a SQL statement has a SQL profile and a SQL plan baseline, the query optimizer tries to merge the hints associated to the SQL profile with those associated to the SQL plan baseline. However, merging a SQL plan baseline with a SQL profile is of limited use. In fact, the aim of a SQL plan baseline is to force a specific execution plan to be used. As a result, the SQL profile might only be useful to generate a new nonaccepted execution plan before taking the SQL plan baseline into consideration.

## On to Chapter 12

This chapter describes several SQL optimization techniques. Selecting one of them isn't always easy. Nevertheless, if you understand not only how they work but also the pros and cons of using them, the choice is much easier. That said, in practice your choice is limited because you can't apply all techniques in all situations. This may be either because of technical limits or because there are licensing issues.

Chapter 12 is devoted to parsing, which is surely a central step in the execution of SQL statements. Parsing is so important because it's when the query optimizer generates the execution plans. To always have an efficient execution plan, you want to parse every SQL statement executed by the database engine. But conversely, parsing is inherently a very expensive operation. As a result, it must be minimized, and execution plans should be reused as much as possible—but not too much, though. This might mean that the execution plan isn't always an efficient one. Once again, in order to take advantage of the database engine in the best possible way, you have to understand how it works and what the pros and cons of the different features are.

CHAPTER 12

# Parsing

The impact of parsing on overall performance is extremely variable. In some cases, it's simply not noticeable. In other cases, it's one of the major causes of performance problems. If you have problems with parsing, it usually means the application doesn't handle it correctly. This is a major problem, because more often than not, to change the behavior of an application, you need to modify the code considerably. Developers need to be aware of the implications of parsing and how to write code in such a way as to avoid, as much as possible, the problems associated with it.

Chapter 2 describes the life cycle of a cursor and how parsing works. This chapter describes how to identify, solve, and work around parsing problems. I also discuss the overhead associated with parsing. Finally, I describe the features provided by common application programming interfaces that reduce parsing activities.

## Identifying Parsing Problems

While identifying parsing problems, it's easy to have an attack of compulsive tuning disorder. The problem is that several dynamic performance views contain counters that detail the number of soft parses, hard parses, and executions. These counters, as well as the ratios based on them, are useless because they provide no information about the time spent parsing. Note that with parses, this is a real problem because they have no typical duration. In fact, depending on the complexity of the SQL statement and the objects it references, the duration of parses commonly differs by several orders of magnitude. Simply put, such counters tell you only whether the database engine has done a few or many parses, without any information about whether this is a problem. Because of this, in practice they may be useful only for trending purposes.

If you are following the recommendations provided in Part 1 and 2, it should be clear that the only effective way of identifying parsing problems you should consider is to measure how much time the database engine spends parsing SQL statements. If you are looking for overall timing information for a single session, or for the whole system, you can query one of the dynamic performance views providing time model statistics. These views include `v$sess_time_model`, `v$sys_time_model`, and, in a 12.1 multitenant environment, `v$con_sys_time_model`.

For example, the output of the following query shows information about one session that spent a lot of time (almost 59%) parsing SQL statements:

```
SQL> WITH
       db_time AS (SELECT sid, value
                   FROM v$sess_time_model
                   WHERE sid = 137
                   AND stat_name = 'DB time')
     SELECT ses.stat_name AS statistic,
            round(ses.value / 1E6, 3) AS seconds,
            round(ses.value / nullif(tot.value, 0) * 1E2, 1) AS "%"
     FROM v$sess_time_model ses, db_time tot
     WHERE ses.sid = tot.sid
```

```
AND ses.stat_name <> 'DB time'
AND ses.value > 0
ORDER BY ses.value DESC;

STATISTIC                                SECONDS          %
---------------------------------------- ---------- ----------
DB CPU                                      18.204       99.3
parse time elapsed                          10.749       58.6
hard parse elapsed time                      8.048       43.9
sql execute elapsed time                     1.968       10.7
connection management call elapsed time       .021         .1
PL/SQL execution elapsed time                 .009         .1
repeated bind elapsed time                       0          0
```

Information like that provided by this query is useful in determining whether there is a problem with parsing. Unfortunately, the time model statistics provided by the dynamic performance views just mentioned aren't helpful in finding out just which SQL statements are causing the problem!

If you are looking for hard facts and not just clues, there are only two sources of information you can use: the output generated by SQL trace, and the active session history from either `v$active_session_history` or `dba_hist_active_sess_history`. In fact, these are the only sources that can provide timing information about parsing at the SQL statement level. That's why in this chapter I discuss the identification of parsing problems based only on SQL trace and active session history.

---

■ **Note** If you want to use active session history to analyze a parsing problem, you have to be aware of four limitations. First, using active session history requires not only Enterprise Edition, but also the Diagnostics Pack option. Second, active session history provides the necessary information to analyze a parsing problem (the flags `in_parse` and `in_hard_parse`) only from version 11.1 onward. Third, you cannot use Enterprise Manager for the analysis. Fourth, since the text of the SQL statement isn't directly available through active session history (you get only the SQL ID), the information provided isn't always sufficient to identify the SQL statements that are causing a parsing problem.

---

There are two main kinds of parsing problems. The first is associated with parses lasting a very short time. Let's call them *quick parses*. Of course, to be noticeable, a lot of them have to be executed. The second kind of parsing problem is associated with parses lasting a long time. Let's call this type *long parses*. This usually happens when the SQL statement is fairly complex and the query optimizer needs a long time to generate an efficient execution plan. In this case, the number of executions isn't relevant.

I describe the method used to identify the two kinds of parsing problems in the next two sections. Since there is no real difference in their identification, I describe only the first one in full detail.

## Quick Parses

The following sections describe how to identify performance problems caused by quick parses. The load used as an example is generated by executing the Java class stored in the `ParsingTest1.java` file against a version 11.2.0.3 database. Implementations of the same processing in PL/SQL, C (OCI), C# (ODP.NET), and PHP (PECL OCI8 extension) are available as well. Since Chapter 3 describes two profilers, TKPROF and TVD$XTAT, I discuss the same example for the output file of both profilers. The trace file and both output files are available in the `ParsingTest1.zip` file.

## Using TKPROF

As suggested in Chapter 3, TKPROF is executed with the following options:

```
tkprof <trace file> <output file> sys=no sort=prsela,exeela,fchela
```

To start the analysis of the output file, it's always good to take a look at the last few lines. In this specific case, it's important to notice that the processing lasted about 14 seconds, that the application executed 10,000 SQL statements, and that all SQL statements were different from each other (*user SQL statements* are equal to *unique SQL statements*).

```
     1  session in tracefile.
 10000  user  SQL statements in trace file.
     0  internal SQL statements in trace file.
 10000  SQL statements in trace file.
 10000  unique SQL statements in trace file.
120060  lines in trace file.
    14  elapsed seconds in trace file.
```

Next, it's time to check how long the execution of the first SQL statement listed in the output lasted. Remember, thanks to the `sort` option that you have specified, the SQL statements were sorted according to their response times. Interestingly enough, the response time (column `elapsed`) of the first cursor is less than a 100th of a second (0.00). In other words, all SQL statements were executed in less than a 100th of a second. Actually, on average, an execution lasted 1.4 milliseconds (14/10,000). This means the response time is certainly not due to a few long-running SQL statements but to the high number of SQL statements processed in a short time.

```
call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        1      0.00       0.00          0          0          0           0
Execute      1      0.00       0.00          0          0          0           0
Fetch        1      0.00       0.00          0          2          0           0
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total        3      0.00       0.00          0          2          0           0
```

In such a situation, to know whether parsing is a problem, you must examine the section providing the overall totals. According to the execution statistics, the parse time was responsible for about 95% (5.7/6) of the processing time. This clearly shows that the database engine did nothing else besides parsing.

```
call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse    10000      5.54       5.70          0          0          0           0
Execute  10000      0.17       0.15          0          0          0           0
Fetch    10000      0.13       0.14          0      23051          0        3048
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total    30000      5.86       6.00          0      23051          0        3048
```

The following line also shows that each of the 10,000 parses was a hard parse. Note that even if a high percentage of hard parses is usually not wanted, it's not necessarily a problem. It's just a clue that something may be suboptimal.

```
Misses in library cache during parse: 10000
```

The problem with the execution statistics is that about 57% (1 – 6.00/14) of the response time is missing from them. Actually, by looking at the table summarizing the wait events, it can be seen that 6.24 seconds were spent waiting for the client. That still leaves us with about 2 seconds (14 – 6.00 – 6.24) of time still unaccounted for, however.

```
Event waited on                             Times   Max. Wait  Total Waited
----------------------------------------   Waited  ----------  ------------
SQL*Net message to client                   10000        0.00          0.02
SQL*Net message from client                 10000        0.02          6.24
latch: shared pool                              5        0.00          0.00
log file sync                                   1        0.00          0.00
```

Since you know that parsing is a problem, it's wise to take a look at the SQL statements. In this case, by looking at a few of them (the following are the top five), it's evident that they are very similar. Only the literals used in the WHERE clauses are different. This is a typical case where bind variables aren't used.

```
SELECT pad FROM t WHERE val = 0
SELECT pad FROM t WHERE val = 2139
SELECT pad FROM t WHERE val = 9035
SELECT pad FROM t WHERE val = 8488
SELECT pad FROM t WHERE val = 1
```

The problem in such situations is that TKPROF doesn't recognize SQL statements that differ only in their literals. In fact, even when the aggregate option is set to yes, which is the default, only the SQL statements that have the same text are grouped together. This is a major flaw that in real cases makes analyzing quick parse problems with TKPROF difficult. To make it a bit easier, it's possible to specify the record option. In this way, a file containing only the SQL statements is generated.

```
tkprof <trace file> <output file> sys=no sort=prsela,exeela,fchela record=<sql file>
```

Then you can use command-line utilities such as grep and wc to find out how many similar SQL statements are available. For example, the following command returns the value 10,000:

```
grep "SELECT pad FROM t WHERE val =" <sql file> | wc -l
```

## Using TVD$XTAT

TVD$XTAT is executed without specifying particular options:

```
tvdxtat -i <trace file> -o <output file>
```

The analysis of the output file starts by looking at the overall resource usage profile. The processing here lasted about 14 seconds. Of this time, about 43% was spent waiting for the client, and 40% was spent running on the CPU. The figures are basically the same as the ones described in the previous section. Only the precision is different. The only additional information in the first section is that the unaccounted-for time is explicitly given.

```
                              Total          Number of Duration per
Component                  Duration       %     Events        Event
--------------------------- -------- ------- --------- ------------
SQL*Net message from client    6.243  43.075    10,000        0.001
CPU                            5.862  40.444       n/a          n/a
unaccounted-for                2.364  16.309       n/a          n/a
SQL*Net message to client      0.024   0.168    10,000        0.000
latch: shared pool             0.000   0.002         5        0.000
log file sync                  0.000   0.002         1        0.000
--------------------------- -------- -------
Total                         14.494 100.000
```

By just looking at the summary of nonrecursive SQL statements, you can see that a single SQL statement is responsible for the whole processing. This is a significant difference between TKPROF and TVD$XTAT. In fact, TVD$XTAT recognizes similar SQL statements and reports them together.

```
Statement          Total          Number of Duration per
ID        Type   Duration       % Executions    Execution
--------- ------ -------- ------- ---------- ------------
#1        SELECT   12.130  83.689     10,000        0.001
#2        COMMIT    0.000   0.002          1        0.000
--------- ------ -------- -------
Total              12.130  83.691
```

According to the execution statistics without recursive statements of the SQL statement number 1, the parse time is responsible for about 95% (5.705/6.009) of the processing time. This clearly shows that the database engine did nothing else besides parsing. The slight difference in the execution statistics between the output file of TKPROF and TVD$XTAT is that TVD$XTAT shows the number of misses (in other words, hard parses) next to the number of parse calls.

```
Call     Count Misses   CPU Elapsed PIO    LIO Consistent Current  Rows
------- ------ ------ ----- ------- --- ------ ---------- ------- -----
Parse   10,000 10,000 5.548   5.705   0      0          0       0     0
Execute 10,000      0 0.176   0.156   0      0          0       0     0
Fetch   10,000      0 0.138   0.148   0 23,051     23,051       0 3,048
------- ------ ------ ----- ------- --- ------ ---------- ------- -----
Total   30,000 10,000 5.862   6.009   0 23,051     23,051       0 3,048
```

The problem with these execution statistics is that about 51% (1 – 6.009/12.130) of the response time is missing from them. At any rate, you can see part of the missing time by looking at the resource usage profile at the SQL statement level shown here; specifically, 6.243 seconds were spent waiting for the client.

```
                              Total          Number of Duration per
Component                  Duration       %    Events        Event
-------------------------- -------- ------- --------- ------------
SQL*Net message from client   6.243  51.470    10,000        0.001
CPU                           5.862  48.327       n/a          n/a
SQL*Net message to client     0.024   0.201    10,000        0.000
latch: shared pool            0.000   0.003         5        0.000
-------------------------- -------- -------
Total                        12.130 100.000
```

## Using Active Session History

Active session history is based on sampling. You thus require plenty of samples for performing a sensible analysis. Given that test case 1 runs only for a dozen seconds, using active session history for the analysis isn't likely to result in accurate information. The aim of this section is to show you the kind of queries you might want to use for identifying which SQL statements were parsed and how much time was spent.

In active session history, the `in_parse` and `in_hard_parse` flags inform you whether, at the time a sample was taken, the session was parsing a SQL statement. Based on these flags, you can write a query like the following that estimates, for a specific session, not only the DB time, but also the time spent parsing SQL statements (note that both figures are in seconds):

```
SQL> SELECT count(*) AS db_time,
  2         count(nullif(in_parse, 'N')) AS parse_time,
  3         count(nullif(in_hard_parse, 'N')) AS hard_parse_time
  4  FROM v$active_session_history
  5  WHERE session_id = 68
  6  AND session_serial# = 23;

DB_TIME PARSE_TIME HARD_PARSE_TIME
------- ---------- ---------------
      5          4               4
```

If you recognize that parsing is a problem (for example, according to the previous output, 80% of the time is spent for parsing), you need to know not only which SQL statements were parsed, but also how much time was spent for each of them. Unfortunately, one of the limitations of active session history is that the text of a SQL statement itself isn't directly available. To get the statement text, you have to perform a lookup based on the SQL ID in another view. For example, you might join `v$active_session_history` to `v$sqlarea`.

Unfortunately, especially on a busy database instance that's suffering because of quick parses, it's quite common that cursors stay in the library cache for a short period of time. As a result, you might not manage to get as much information as required. To slightly increase the likelihood of getting more information, you might use `v$sqlstats` instead of `v$sqlarea`. In fact, the former has a greater retention. For example, the following query is able to retrieve the SQL statement of only two of the four samples related to parsing (remember, test case 1 executes 10,000 SQL statements):

```
SQL> SELECT a.sql_id, s.sql_text, count(*) AS parse_time
  2  FROM v$active_session_history a, v$sqlstats s
  3  WHERE a.sql_id = s.sql_id(+)
  4  AND a.session_id = 68
  5  AND a.session_serial# = 23
  6  AND a.in_parse = 'Y'
  7  GROUP BY a.sql_id, s.sql_text
  8  ORDER BY count(*) DESC;

SQL_ID        SQL_TEXT                          PARSE_TIME
------------- --------------------------------- ----------
a6z6qamdcwqdv                                            1
2hcrrthw3w4y8                                            1
aydf9rbd6mz1m SELECT pad FROM t WHERE val = 9580         1
50m9q01tmghmw SELECT pad FROM t WHERE val = 7574         1
```

## Summarizing the Problem

The analysis performed via active session history isn't very useful in this case. This is expected because sampling a single session over a dozen seconds can result in only a few samples. However, the analysis performed by TKPROF and TVD$XTAT clearly shows that the processing performed by the database engine is solely due to parsing. However, on the database side, parsing is responsible for only about 39% (5.705/14.494) of the overall response time.

This implies that eliminating it should approximately halve the overall response time. The analysis also shows that 10,000 SQL statements like the following one were parsed and executed once:

```
SELECT pad FROM t WHERE val = 0
```

Since a constantly changing literal is used, shared cursors in the library cache cannot be reused. In other words, every parse is a hard parse. Figure 12-1 shows a graphical representation of this processing.

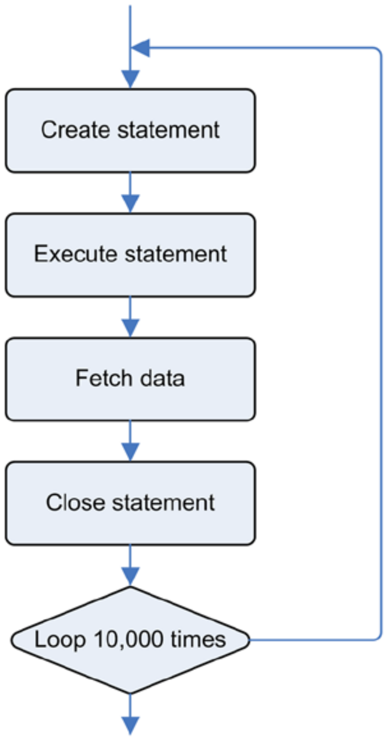


***Figure 12-1.*** *The processing performed by test case 1*

---

■ **Note** The processing shown in Figure 12-1 is later referred to as *test case 1*.

---

It goes without saying that such processing is inefficient. Refer to the section "Solving Parsing Problems" later in this chapter for possible solutions to such a problem.

## Long Parses

The following sections describe how to identify performance problems caused by long parses. However, active session history isn't covered. The reason is quite simple: it's rare to see parse calls that take more than few seconds. As a result, most of the time it isn't sensible to analyze such a problem through active session history. In any case, if you experience parse calls of several minutes, the analysis you would do with active session history is similar to the one described in the previous "Quick Parses" section. Since Chapter 3 describes two profilers, TKPROF and TVD$XTAT, I discuss the same example for the output file of both profilers. The trace file used as an example in this section was generated by executing the `long_parse.sql` script. The trace file and the output files are available in the `long_parse.zip` file.

## Using TKPROF

As with quick parses, the analysis starts at the end of the TKPROF output. In this specific case, it's significant to note that the processing lasted about 2 seconds and that the application executed only three SQL statements. All other SQL statements were recursively executed by the database engine.

```
   1  session in tracefile.
   3  user  SQL statements in trace file.
  13  internal SQL statements in trace file.
  16  SQL statements in trace file.
  16  unique SQL statements in trace file.
9644  lines in trace file.
   2  elapsed seconds in trace file.
```

By looking at the execution statistics of the first SQL statement in the output file, it's possible to see not only that it was responsible for the whole response time (more than 2 seconds), but also that all the time is spent on a single parse:

```
call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        1      2.65       2.65          0          0          0           0
Execute      1      0.00       0.00          0          0          0           0
Fetch        2      0.00       0.00         10         10          0           1
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total        4      2.65       2.66         10         10          0           1
```

## Using TVD$XTAT

As with quick parses, the analysis of the TVD$XTAT output starts by looking at the overall resource usage profile. The processing lasted about 2.8 seconds. Of this time, about 98% was spent running on the CPU. Also notice that in this case, the unaccounted-for time is very short and, therefore, completely negligible.

```
                              Total         Number of Duration per
Component                  Duration       %    Events        Event
-------------------------- -------- ------- --------- ------------
CPU                           2.769  98.383       n/a          n/a
db file sequential read       0.027   0.943       314        0.000
unaccounted-for               0.017   0.596       n/a          n/a
SQL*Net message from client   0.002   0.078         3        0.001
SQL*Net message to client     0.000   0.000         3        0.000
-------------------------- -------- -------
Total                         2.814 100.000
```

Just by looking at the summary of nonrecursive SQL statements, you can see that three SQL statements were executed. Of them, a `SELECT` statement is responsible for almost all the response time.

```
Statement           Total          Number of Duration per
ID        Type   Duration       % Executions    Execution
--------- ------ -------- ------- ---------- ------------
#1        SELECT    2.791  99.167          1        2.791
#9        PL/SQL    0.005   0.166          1        0.005
#12       PL/SQL    0.002   0.071          1        0.002
--------- ------ -------- -------
Total               2.797  99.404
```

According to the nonrecursive execution statistics of the SQL statement causing the problem, a single parse operation was responsible for about 100% (2.654/2.661) of the processing time. This clearly shows that the database engine did nothing else besides parsing.

```
Call    Count Misses   CPU Elapsed PIO LIO Consistent Current Rows
------- ----- ------ ----- ------- --- --- ---------- ------- ----
Parse       1      1 2.653   2.654   0   0          0       0    0
Execute     1      0 0.002   0.002   0   0          0       0    0
Fetch       2      0 0.004   0.006  10  10         10       0    1
------- ----- ------ ----- ------- --- --- ---------- ------- ----
Total       4      1 2.659   2.661  10  10         10       0    1
```

### Summarizing the Problem

The analysis shows that a single SQL statement is responsible for almost the whole response time. In addition, the whole response time is due to parsing for this particular SQL statement. Eliminating it would probably greatly reduce the response time.

## Solving Parsing Problems

The obvious way to solve parsing problems is to avoid the parse phase. Unfortunately, this isn't always that easy. In fact, depending on whether the parsing problem is related to quick parses or long parses, you have to implement different techniques to solve the problem. I discuss these separately in the following sections. In both cases, the examples described in the previous "Identifying Parsing Problems" section are used as the basis for explaining possible solutions.

■ **Note** The following sections describe the impact of parsing by showing the results of different performance tests. The performance figures are intended only to help compare different kinds of processing and to give you a feel for their impact. Remember, every system and every application has its own characteristics. Therefore, the relevance of using each technique might be very different depending on where it's applied.

### Quick Parses

This section describes how to take advantage of prepared statements to avoid unnecessary parse operations. Since implementation details depend on the development environment, they aren't covered here. Later in this chapter, specifically in the "Using Application Programming Interfaces" section, I provide details for PL/SQL, OCI, JDBC, ODP.NET, and PHP.

## Using Prepared Statements

The first thing to do when a SQL statement causing parsing problems uses literals that are constantly changing is to replace the literals with bind variables. For that, you have to use a *prepared statement*. The aim of using a prepared statement is to share a single cursor for all SQL statements and, consequently, to avoid unnecessary hard parses by turning them into soft parses. Figure 12-2 shows a graphical representation of the processing intended to improve on the performance in test case 1.

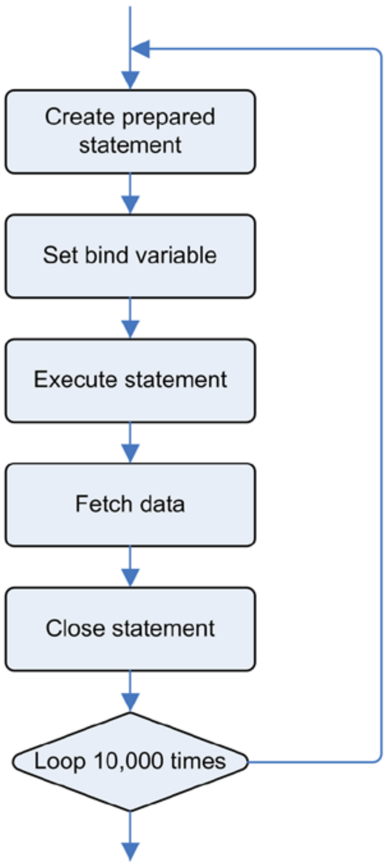


***Figure 12-2.*** *The processing performed by test case 2*

---

■ **Note** The processing shown in Figure 12-2 is later referenced as *test case 2*.

---

With that enhancement, as shown in Figure 12-3, the response time decreased by about 41% compared with test case 1. This was expected because the new code, thanks to the prepared statement, performed only a single hard parse. As a result, most of the processing carried out by the database engine in test case 1 was avoided. Note, however, that 10,000 soft parses were still performed.

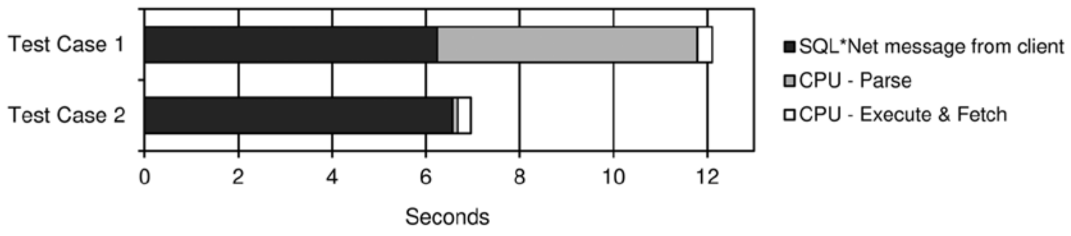


***Figure 12-3.*** *Comparison of the database-side resource usage profile in test case 1 and test case 2. (Components that account for less than 1% of the response time aren't displayed because they wouldn't be visible.)*

## Reusing Prepared Statements

I suggest in the previous section that using prepared statements is a very good thing. It's even better to reuse them to eliminate not only the hard parses but the soft parses as well. Since in test case 2 the elapsed time for the parses is almost not noticeable, you may ask yourself why. Before giving an answer, I show the performance figures related to processing that reuses a single prepared statement. Specifically, Figure 12-4 shows a graphical representation of the processing intended to improve on the performance in test case 2.

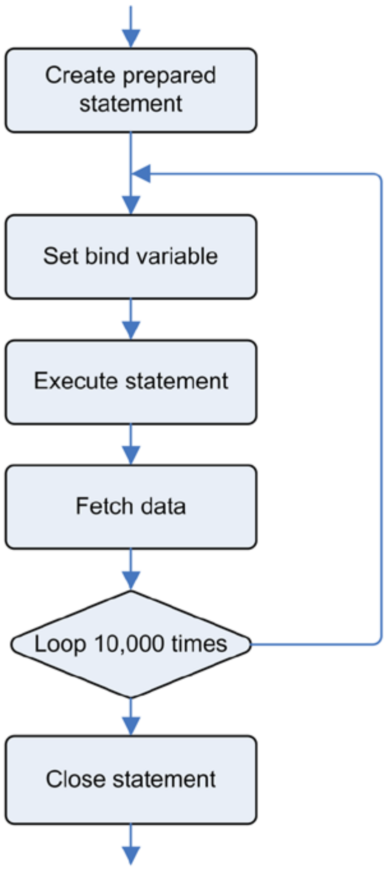


***Figure 12-4.*** *The processing performed by test case 3*

---

■ **Note** The processing shown in Figure 12-4 is later referred to as *test case 3*.

---

With this enhancement, as shown in Figure 12-5, the response time, compared with test case 1 and test case 2, decreased by about 61% and 33%, respectively. Significantly, the real difference was made not by the reduction of the CPU time spent to parse (which is already very low in test case 2) but by the reduction of the wait for `SQL*Net message from client`. This means you are saving resources either in the network or in the client, or possibly both.

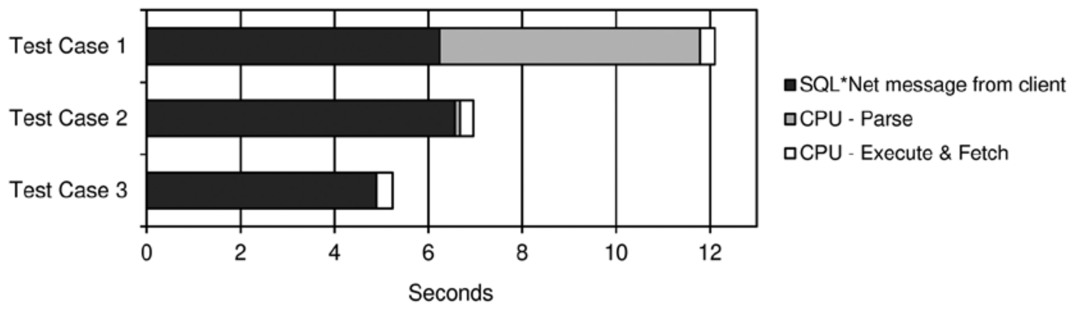


***Figure 12-5.** Comparison of the database-side resource usage profile for the three test cases. (Components that account for less than 1% of the response time aren't displayed because they wouldn't be visible.)*

In test case 2, the processing of soft parses at the database level lasted about one-tenth of a second. The question is, where does the improvement come from? It surely doesn't come from the reduction of resource utilization at the database level. You might intuitively think that the gain is because of fewer round-trips between the client and the server. However, by looking at the number of waits for `SQL*Net message from client` and `SQL*Net message to client`, you can see that there is no difference among the three test cases. In each case, there are 10,000 round-trips. This is significant because 10,000 executions are performed, and therefore, this implies that in this particular case, all necessary calls (among others, the parse, execute, and fetch calls) are packed into one single SQL*Net message by the client driver. There is, however, a difference in the network layer because of the size of the messages sent between the client and the server. You can use the following query to get information about them:

```
SELECT sn.name, ss.value
FROM v$statname sn, v$sesstat ss
WHERE sn.statistic# = ss.statistic#
AND sn.name LIKE 'bytes%client'
AND ss.sid = 42
```

---

■ **Note** At the SQL statement level, it's not possible to check the amount of data sent between the client and the server. For this reason, the previous query retrieves statistics at the session level. In this specific case, it's not a problem doing so because I can make sure that the session I'm looking at is executing only the SQL statements of my test case. In addition, I run the query against the dynamic performance views from another session.

---

Figure 12-6 shows the network traffic for the three test cases. It's important to notice how switching to prepared statements in test case 2 slightly increases the size of the messages received from the database engine. The most important difference, though, is the substantial reduction of the size of the messages both received and sent from the database engine, comparing test case 3 with the other two. This is caused by the data sent over the network in order to open and close a new cursor (in test case 3, the text of the SQL statement is sent only once over the network, together with the first open, and the cursor is closed only once, at the end).

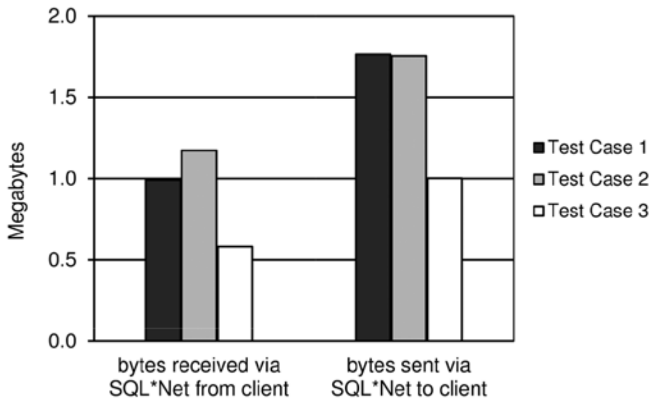


***Figure 12-6.*** *Network traffic for a single execution of the three test cases*

Since the size of the messages sent through the network differs, the response time is expected to depend on the network speed. If the network is fast, the impact of the communication between the client and the server is low or not even noticeable. If the network is slow, the impact may be significant. Figure 12-7 shows the response time for two network speeds. Notice how the time spent by the database engine processing the calls for a given test case doesn't depend, obviously, on the network speed.

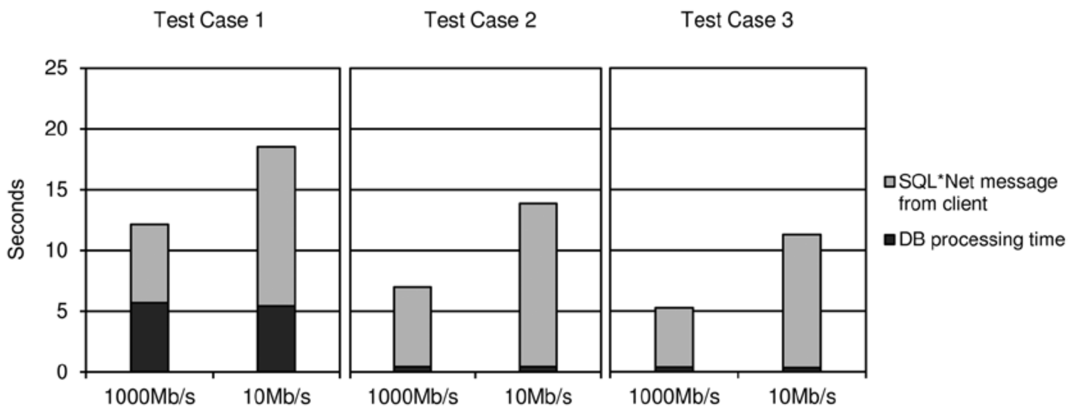


***Figure 12-7.*** *Response time with two network speeds for the three test cases*

Even if the network speed had the biggest impact on the overall response time, it's important to note that the three test cases have a different impact on the client-side resource utilization, specifically the CPU. Figure 12-8 shows the client-side CPU utilization for the three test cases. The comparison of the figures for test case 1 and test case 2 shows that the use of bind variables has an overhead for the client. The comparison of the figures for test case 2 and test case 3 shows that creating and closing SQL statements also cause an overhead for the client.

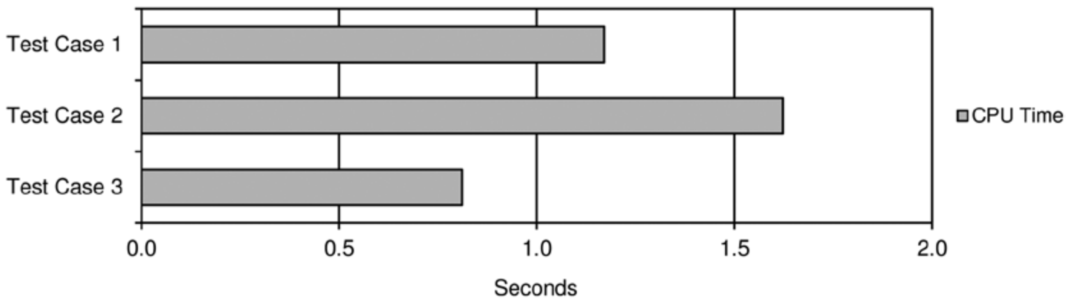


***Figure 12-8.*** *Client-side CPU utilization for the three test cases*

## Client-Side Statement Caching

This feature is designed to solve performance problems created by applications that cause too many soft parses because cursors are unnecessarily opened and closed. Earlier in this chapter, this problem was indicated by test case 2.

The concept of client-side statement caching is quite simple. Whenever the application closes a cursor, instead of really closing it, the client-side database layer (which is responsible for communicating with the database engine) keeps it open and adds it to a cache. Then, later, if a cursor based on the same SQL statement is opened and parsed again, instead of really opening and parsing it, the cached cursor is reused. Thus, the soft parse shouldn't take place. Basically, the aim is to have an application behaving like test case 3, even if it's written like test case 2.

To take advantage of this feature, it's usually only a matter of enabling it and defining the maximum number of cursors that can be cached by a session. Note that when the cache is full, the least recently used cursors are closed and replaced by newer ones. The activation takes place either by adding some initialization code into the application or by setting a variable in the environment. How this works exactly depends on the programming environment. Later in this chapter, specifically in the "Using Application Programming Interfaces" section, details are provided for PL/SQL, OCI, JDBC, ODP.NET, and PHP. To set the maximum number of cached cursors, you need to know the application that is being used. If you don't know it, you should analyze it to find out how many SQL statements are subject to a high number of soft parses. In both cases, this is just a first estimation. Afterward, you will have to perform some tests to verify whether the value is good. In any case, it makes no sense to exceed the value of the `open_cursors` initialization parameter.

As shown in Figure 12-9, test case 2 with client-side statement caching performs almost as well as test case 3. Actually, both executed a single hard parse and a single soft parse. As a result, thanks to statement caching, the client-side processing is greatly reduced.

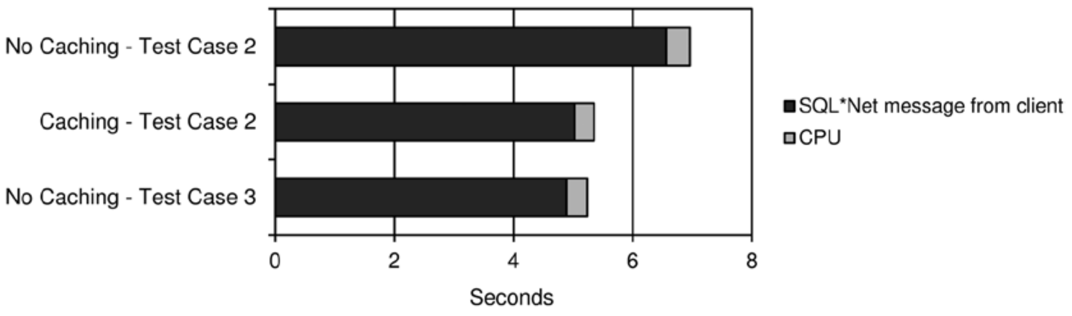


***Figure 12-9.*** *Comparison of the database-side resource usage profile with and without client-side statement caching. (Components that account for less than 1% of the response time aren't displayed because they wouldn't be visible.)*

## Summing Up

Utilizing prepared statements with bind variables is crucial in order to avoid unnecessary hard parses. However, when using them, you can expect a small overhead in the client-side CPU utilization and network traffic. You could argue that this overhead will lead to performance problems and, consequently, that prepared statements and bind variables should be used only when really necessary. Since the overhead is almost always negligible, the best practice is to use prepared statements and bind variables whenever possible, as long as they don't lead to inefficient execution plans (refer to Chapter 2 for detailed information about this topic). Whenever a prepared statement is frequently used, it's a good idea to reuse it. By doing so, not only do you avoid soft parses, but you reduce the client-side CPU utilization and the network traffic as well. The only problem related to keeping a prepared statement open has to do with memory utilization, on both the client side and the server side. This means that keeping thousands of cursors open per session must be done carefully and only when the necessary memory is available. Also note that the `open_cursors` initialization parameter limits the number of cursors that can be concurrently kept open by a single session. In case many prepared statements have to be cached, it's probably better to use client-side statement caching with a carefully sized cache instead of manually keeping them open. In this way, the memory pressure may be mitigated by allowing a limited number of prepared statements to be cached.

## Long Parses

In case of long parses that are executed only a few times (or as in the previous example, only once), it's usually not possible to avoid the parse phase. In fact, the SQL statement must be parsed at least once. In addition, if the SQL statement is rarely executed, a hard parse is probably inevitable because the cursor will be aged out of the library cache between executions. This is especially true if no bind variables are used. Therefore, the only possible solution is to reduce the parsing time itself.

What causes long parse times? Commonly, they are caused by the query optimizer evaluating too many different execution plans. In addition, it can happen because of recursive queries executed on behalf of dynamic sampling. Solving the latter should be obvious: you either reduce the level of dynamic sampling or completely avoid using it. However, solving the former is a bit more tricky. In fact, to shorten the parse times, you must reduce the number of evaluated execution plans. This is generally possible only by forcing a specific execution plan through hints or stored outlines. For example, after creating a stored outline for the SQL statement used as an example in the "Identifying Parsing Problems" section, the parse time is reduced by a factor of six (see Figure 12-10). A similar effect may be reached by directly specifying hints in the SQL statement, although this is possible only if you are able to modify the code.

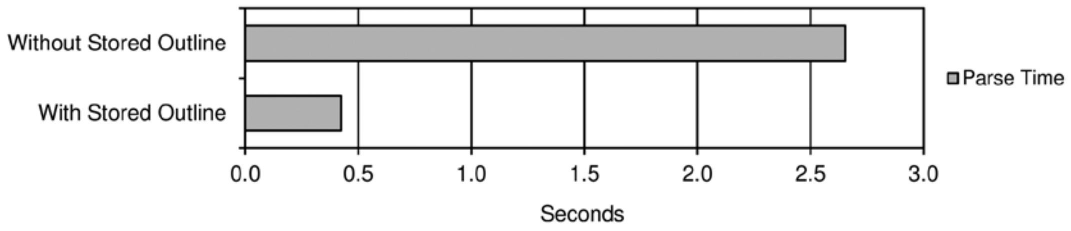


***Figure 12-10.*** *Comparison of the parse time with and without a stored outline*

## Working Around Parsing Problems

The previous sections describe three test cases related to quick parses. The first is simply a case of poor code writing. The second is much better than the first. The third is the best in most situations. The dilemma is that code similar to test case 1 has to be modified in order to be enhanced, and that, unfortunately, isn't always possible. This is because either the code isn't available, technical barriers prevent to enhance the code (for example, prepared statements aren't available in the programming environment), or it's too "expensive" to make all the necessary modifications.

The following sections explain how to work around such problems in order to achieve results similar to those reached by carrying out the right implementation. Even if the performance of such workarounds isn't as good as that possible with a correct implementation, in some situations the workaround is much better than doing nothing at all.

---

■ **Note** The following sections describe the impact of parsing by showing the results of different performance tests. The performance figures are intended only to help compare different kinds of processing and to give you a feel for their impact. Remember, every system and every application has its own characteristics. Therefore, the relevance of using each technique might be very different depending on where it's applied.

---

## Cursor Sharing

This feature is designed to work around performance problems caused by applications that improperly use literals instead of bind variables, which in turn leads to too many hard parses. Earlier in this chapter, I pointed this problem out in test case 1.

The concept of cursor sharing is simple. If an application executes SQL statements containing literals and if cursor sharing is enabled, the database engine automatically replaces the literals with bind variables. Thanks to these replacements, hard parses might be turned into soft parses for the SQL statements that differ only in the literals. Basically, the goal is to have an application behaving like test case 2, even if it's written like test case 1.

---

■ **Note** Cursor sharing doesn't replace literal values contained in static SQL statements executed through PL/SQL. For dynamic SQL statements, the replacement takes place only when literals aren't mixed with bind variables. This isn't a bug; it's a design decision. You can use the `cursor_sharing_mix.sql` script to reproduce this behavior.

---

Cursor sharing is controlled through the `cursor_sharing` initialization parameter. If it's set to `exact`, the feature is disabled. In other words, SQL statements share the same parent cursor only if their text is identical. If `cursor_sharing` is set to `force` or `similar`, the feature is enabled. The default value is `exact`. You can change it at the system and session levels. It's also possible to explicitly disable cursor sharing at the SQL statement level by specifying the `cursor_sharing_exact` hint.

Oracle Support note 1169017.1 (*Deprecating the cursor_sharing = 'SIMILAR' setting*) declares that, from version 11.1 onward, setting the `cursor_sharing` initialization parameter to `similar` is deprecated. In addition, as of version 11.2.0.3, when the parameter is set to `similar`, the database engine behaves as if it were set to `force`! There are two main reasons for deprecating the value `similar`. First, as you will read shortly, there are problems in its implementation. Second, the introduction of adaptive cursor sharing (refer to Chapter 2 for information about this feature) makes `similar` no longer necessary. In fact, adaptive cursor sharing can work with cursor sharing set to `force`.

---

■ **Caution** Cursor sharing has a reputation for not being very stable. This is because, over the years, plenty of bugs related to it have been found and fixed. Therefore, if you are considering using it, my advice is to carefully review Oracle Support note 94036.1 (*Init.ora Parameter "CURSOR_SHARING" Reference Note*), specifically the list of known bugs.

---

Since cursor sharing can be enabled with two values, `force` and `similar`, let's discuss the differences between them. For that purpose, test case 1 was executed against a 10.2.0.5 database once for each value of the `cursor_sharing` initialization parameter.

Let's take a look at the results with the value `force`. As shown in Figure 12-11, the database-side resource usage profile in test case 1 (with the value `force`) is similar to the one in test case 2 with `exact`. Actually, both executed a single hard parse and 9,999 soft parses. As a result, thanks to cursor sharing, the parse time was greatly reduced. With the value `force`, there is just a slight increase in the CPU utilization. Since the database engine has to perform more work in order to replace literals with bind variables, this is to be expected.

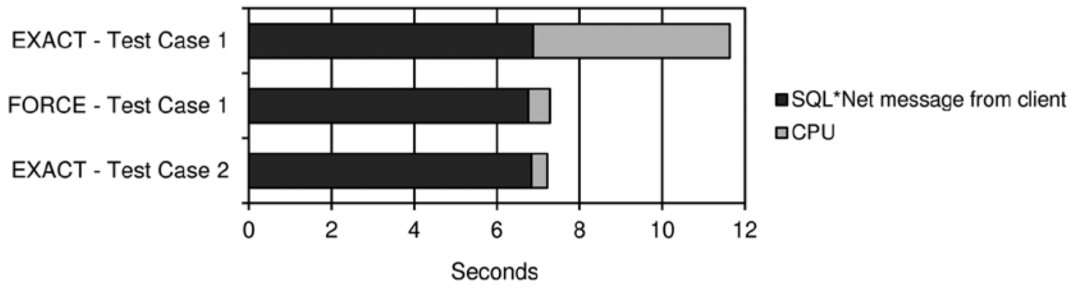


***Figure 12-11.** Comparison of the database-side resource usage profile with cursor sharing set to `force`. (Components that account for less than 1% of the response time aren't displayed because they wouldn't be visible.)*

If adaptive cursor sharing isn't considered, the problem related to the value `force` is that a single child cursor can be used for all SQL statements sharing the same text after the replacement of the literals. Consequently, the literals (that, among other things, are essential for taking advantage of histograms) are peeked only during the generation of the execution plan related to the first submitted SQL statement. Naturally, this could lead to suboptimal execution plans because literals used in subsequent SQL statements might require different execution plans. To avoid this problem, the value `similar` is available. In fact, with `similar`, before reusing a cursor that is already available, the SQL engine checks whether a histogram exists for one of the replaced literals. If it doesn't exist, any available child cursor that has a compatible execution environment can be used. If it does exist, only a child cursor that has been created with the very same literal value can be used. As a result, with `similar`, instead of having a single parent cursor for every literal value, you end up having a single child cursor for every literal value (which uses less memory).

As shown in Figure 12-12, the database-side resource usage profile in test case 1 with the value `similar` is even worse than in test case 1 with `exact`. The problem is not only that 10,000 hard parses were executed, but also that the CPU utilization of such parses is higher, because of cursor sharing. In fact, the parse time increases linearly with the number of child cursors per parent cursor. The parse time increases linearly because, during the parse, the SQL engine has to check whether an already available child cursor can be reused. Therefore, the list of child cursors must be scanned and every child cursor probed for compatibility. Simply put, many child cursors inhibit optimal performance. Note that after the replacement of literals, all SQL statements have the same text. As a result, the library cache contains a single parent cursor that has many or, in this case, thousands of child cursors.

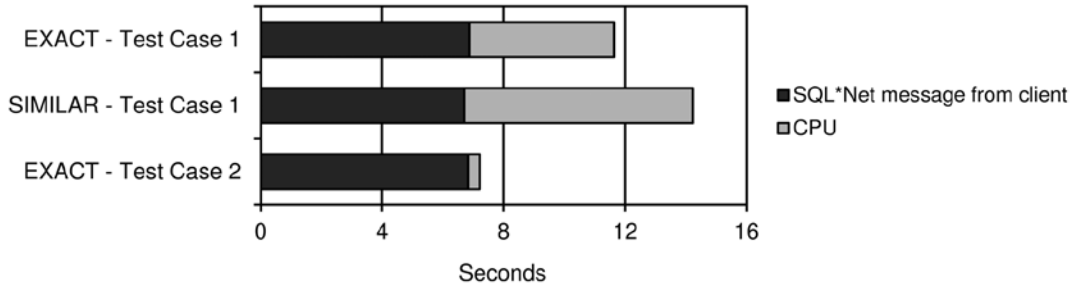


***Figure 12-12.** Comparison of the database-side resource usage profile with cursor sharing set to `similar`. (Components that account for less than 1% of the response time aren't displayed because they wouldn't be visible.)*

In summary, if an application uses literals and cursor sharing is set to `similar`, the behavior depends on the existence of relevant histograms. If they do exist, `similar` behaves like `exact`. If they don't exist, `similar` behaves like `force`. This means that if you are facing parsing problems, more often than not, it's pointless to use `similar`.

## Server-Side Statement Caching

This feature is similar to client-side statement caching because it's designed to reduce overhead when too many soft parses are taking place. From a conceptual point of view, the two types of statement caching are similar, except that one is implemented on the server side and the other on the client side. From a performance point of view, however, the differences are considerable. In fact, the server-side implementation is far less powerful than the client-side implementation. This is because the server-side implementation reduces the overhead of soft parses on the server side only, and in more than a few circumstances, the overhead of soft parses is much greater on the client than on the server. The only real advantage of the server-side implementation is the ability to cache SQL statements that are executed by PL/SQL or Java code deployed in the database engine.

If an application performs a lot of soft parses, the high pressure on library cache latches and mutexes may lead to a noticeable contention on the database engine as well. The following database-side resource usage profile shows such a situation. Note that to generate it, test case 2 was started while the database engine was processing more than 30,000 parses per second for the same SQL statement. Although this is certainly not a common workload, it helps demonstrate the impact of server-side cursor caching.

```
                              Total          Number of Duration per
Component                  Duration       %     Events        Event
--------------------------- -------- ------- ---------- ------------
SQL*Net message from client    4.166  54.569     10,000        0.000
library cache: mutex X         2.622  34.339        158        0.017
CPU                            0.557   7.294        n/a          n/a
latch free                     0.265   3.473          1        0.265
SQL*Net message to client      0.014   0.177     10,000        0.000
cursor: pin S                  0.011   0.148          1        0.011
--------------------------- -------- -------
Total                          7.635 100.000
```

Whenever the server-side overhead of soft parses is a problem and the application cannot be changed, server-side statement caching may be useful. In this specific case, after enabling it and reapplying the same load, the resulting resource usage profile is the following. Notice how most waits related to library cache latches and mutexes disappeared.

```
                              Total         Number of Duration per
Component                  Duration       %    Events        Event
--------------------------- -------- ------- --------- ------------
SQL*Net message from client    4.646  85.959     10000        0.000
CPU                            0.420   7.769       n/a          n/a
cursor: pin S                  0.328   6.070         2        0.164
SQL*Net message to client      0.011   0.202     10000        0.000
--------------------------- -------- -------
Total                          5.405 100.000
```

Server-side statement caching is configured through the `session_cached_cursors` initialization parameter. Its value specifies the maximum number of cursors each session is able to cache. So if it's set to 0, the feature is disabled, and if it's set to a value greater than 0, it's enabled. In version 10.2, the default value is 20; and from version 11.1 onward, it's 50. At the system level, it can be changed only by bouncing the instance. At the session level, it can be dynamically changed. As for the client-side statement caching, to decide which value to specify for the maximum number of cached cursors, you either need to know the application being used or you have to analyze it to find out how many SQL statements are subject to a high number of soft parses. Then, based on this first estimation, some tests will be necessary to verify whether the value is good. During such tests, it's possible to verify the effectiveness of the cache not only by verifying the impact on the response time, but also by looking at the statistics resulting from the following query. Note that the same statistics are available at the system level as well. In any case, you should focus on a single session that has experienced the problematic load in order to find meaningful clues.

```
SQL> SELECT sn.name, ss.value
  2  FROM v$statname sn, v$sesstat ss
  3  WHERE sn.statistic# = ss.statistic#
  4  AND sn.name IN ('session cursor cache hits',
  5                  'session cursor cache count',
  6                  'parse count (total)')
  7  AND ss.sid = 42;

NAME                           VALUE
-------------------------- ----------
session cursor cache hits       9997
session cursor cache count         9
parse count (total)            10008
```

First, compare the number of cached cursors (`session cursor cache count`) with the value of the `session_cached_cursors` initialization parameter. If the first is less than the second, it means that incrementing the value of the initialization parameter should have no impact on the number of cached cursors. Otherwise, if the two values are equal, increasing the value of the initialization parameter might be useful in order to cache more cursors. In any case, it makes no sense to exceed the value of the `open_cursors` initialization parameter. For example, according to the previous statistics, nine cursors are present in the cache. Since the `session_cached_cursors` initialization parameter was set to 50 during the test, increasing it serves no purpose.

Second, using the additional figures, it's possible to check how many parse calls were optimized with server-side statement caching (`session cursor cache hits`) relative to the total number of parse calls (`parse count (total)`). If the two values are close, it probably isn't worthwhile to increase the size of the cache. In the case of the previous statistics, more than 99% (9,997/10,008) of the parses are avoided thanks to the cache, so increasing it is probably pointless.

## CAUTION! BUGS

The values provided by the `parse count (total)` and `session cursor cache hits` statistics are subject to several bugs. The bugs you are most likely to notice are the following:

- As of version 11.1.0.6, the `session cursor cache hits` statistic is incremented for cursors taking advantage of the PL/SQL client-side statement caching. As a result, the `session cursor cache hits` statistic can be much higher than the `parse count (total)` statistic. Using client-side statement caching is the default for a PL/SQL program. Thus, the `session cursor cache hits` statistic is quite useless when PL/SQL is in use.
- As of version 11.2.0.1, the `session cursor cache hits` statistic at the session level is stored in an unsigned integer taking 16 bits. Therefore, sessions with more than 65,535 hits experience an overflow, and the value restarts from 0. And, even though the statistic at the system level doesn't have such a limitation, an overflow at the session level still causes the system level statistic to decrease by 65,535. As a result, the `session cursor cache hits` statistic is almost useless both at the session and at the system level.

- In version 11.2.0.3, the `parse count (total)` statistic isn't incremented for cursors taking advantage of server-side statement caching. As a result, the `session cursor cache hits` statistic can be much higher than the `parse count (total)` statistic. Since using server-side statement caching is the default, the `parse count (total)` statistic is practically useless in version 11.2.0.3. This bug was fixed in version 11.2.0.4.

The bottom line is to be careful when interpreting and relying upon the `session cursor cache hits` statistic. Review the known bugs to make sure none apply to your situation, and especially keep the three listed here in mind.

It's also important to notice that in the previous statistics there were "only" 9,997 hits in the cache. Since test case 2 executed the same SQL statement 10,000 times, why weren't there 9,999? The answer is that a cursor is put in the cursor cache only when it has been executed several times. The reason for this is to avoid caching cursors that are executed only once. Getting 9,999 could be possible only when a shareable cursor is already present in the library cache prior to the first parse call.

In summary, server-side statement caching is an important feature. In fact, when correctly sized, it might save some overhead server side. However, just because this feature is available, there is no excuse for the application to not manage cursors properly in the first place, especially because, as you have already seen, the parsing overhead is higher when the caching is performed server side instead of client side.

## Using Application Programming Interfaces

The goal of this section is to describe the features related to parsing for different application programming interfaces. As described in the previous sections, to avoid unnecessary hard and soft parses, three central features should be available: bind variables, the ability to reuse statements, and client-side statement caching. Table 12-1 summarizes which of these features are available with which application programming interface. The next sections provide some detailed information for PL/SQL, OCI, JDBC, ODP.NET, and PHP.

***Table 12-1.*** *Overview of the Features Provided by Different Application Programming Interfaces*

| Application Programming Interface | Bind Variables | Reusing Statements | Client-Side Statement Caching |
|---|---|---|---|
| Java Database Connectivity (JDBC) | | | |
| `java.sql.Statement` | | | |
| `java.sql.PreparedStatement` | ✓ | ✓ | ✓ |
| Oracle Call Interface (OCI) | ✓ | ✓ | ✓ |
| Oracle C++ Call Interface (OCCI) | ✓ | ✓ | ✓ |
| Oracle Data Provider for .NET (ODP.NET) | ✓ | | ✓ |
| Oracle Objects for OLE (OO4O) | ✓ | | |
| Oracle Provider for OLE DB | ✓ | | ✓ |
| PHP (PECL OCI8 extension) | ✓ | ✓ | ✓ |
| PL/SQL | | | |
| Static SQL | ✓ | | ✓ |
| Native dynamic SQL (`EXECUTE IMMEDIATE`) | ✓ | | ✓ |

(*continued*)

*Table 12-1.* (*continued*)

| Application Programming Interface | Bind Variables | Reusing Statements | Client-Side Statement Caching |
|---|---|---|---|
| Native dynamic SQL (OPEN/FETCH/CLOSE) | ✓ | | |
| Dynamic SQL with the dbms_sql package | ✓ | ✓ | |
| Precompilers | ✓ | ✓ | ✓ |
| SQLJ | ✓ | | ✓ |

## PL/SQL

PL/SQL offers different methods for executing SQL statements. The two main categories are static SQL and dynamic SQL. Dynamic SQL can be further divided into three subcategories: `EXECUTE IMMEDIATE`, `OPEN/FETCH/CLOSE`, and the `dbms_sql` package. The only feature related to parsing that is available for all of them is the possibility of using bind variables. In fact, the reutilization of statements and client-side statement caching are only partially available. They are simply not implemented for all categories of SQL statements. The next sections describe the particularities of each of these four categories.

■ **Note** Since PL/SQL runs in the database engine, it might seem strange to speak about client-side statement caching. Nevertheless, from the SQL engine's perspective, the PL/SQL engine is a client. In that client, the concept of client-side statement caching discussed earlier has been implemented.

The PL/SQL blocks provided as examples in this section are excerpts from the `ParsingTest1.sql`, `ParsingTest2.sql`, and `ParsingTest3.sql` scripts implementing test case 1, 2, and 3, respectively.

### Static SQL

Static SQL is integrated into the PL/SQL language. As its name indicates, it's static, and therefore, the SQL statement must be fully known during PL/SQL compilation. For this reason, the utilization of bind variables is unavoidable if a SQL statement references PL/SQL variables. For example, with static SQL, it's not possible to write a code snippet reproducing test case 1.

You can write static SQL in two ways. The first is based on implicit cursors, so it gives no possibility of controlling the life cycle of a cursor. The following PL/SQL block shows an example implementing test case 2:

```
DECLARE
  l_pad VARCHAR2(4000);
BEGIN
  FOR i IN 1..10000
  LOOP
    SELECT pad INTO l_pad
    FROM t
    WHERE val = i;
  END LOOP;
END;
```

The second way is based on explicit cursors. In this case, some control over the cursors is possible. Nevertheless, the open/parse/execute phases are merged in a single operation (OPEN). This means that only the fetch and close phase can be controlled. The following PL/SQL block shows an example that implements test case 2:

```
DECLARE
  CURSOR c (p_val NUMBER) IS SELECT pad FROM t WHERE val = p_val;
  l_pad VARCHAR2(4000);
BEGIN
  FOR i IN 1..10000
  LOOP
    OPEN c(i);
    FETCH c INTO l_pad;
    CLOSE c;
  END LOOP;
END;
```

From a performance perspective, the two methods are similar. Although they both prevent bad code from being written (test case 1), they don't allow very efficient code to be written (test case 3). This is because no full control over the cursors is available.

To solve this problem, client-side statement caching is available. The maximum number of cached cursors is determined by the `session_cached_cursors` initialization parameter. The default number of cached cursors is 20 in version 10.2 and 50 from version 11.1 onward. Be aware that an initialization parameter, which is not directly related to client-side statement caching, is "misused" in order to configure it! In fact, it's the same initialization parameter used to control server-side statement caching.

## Native Dynamic SQL: EXECUTE IMMEDIATE

From a cursor management perspective, native dynamic SQL based on `EXECUTE IMMEDIATE` is similar to static SQL with implicit cursors. In other words, it's not possible to control the life cycle of a cursor. The following PL/SQL block shows an example implementing test case 2:

```
DECLARE
  l_pad VARCHAR2(4000);
BEGIN
  FOR i IN 1..10000
  LOOP
    EXECUTE IMMEDIATE 'SELECT pad FROM t WHERE val = :1' INTO l_pad USING i;
  END LOOP;
END;
```

Without control over the cursors, it's not possible to write code implementing test case 3. For this reason, client-side cursor caching is used as with static SQL.

## Native Dynamic SQL: OPEN/FETCH/CLOSE

From a cursor management perspective, native dynamic SQL based on OPEN/FETCH/CLOSE is similar to static SQL with explicit cursors. In other words, it's possible to control only the fetch phase. The following PL/SQL block shows an example implementing test case 2:

```
DECLARE
  TYPE t_cursor IS REF CURSOR;
  l_cursor t_cursor;
  l_pad VARCHAR2(4000);
BEGIN
  FOR i IN 1..10000
  LOOP
    OPEN l_cursor FOR 'SELECT pad FROM t WHERE val = :1' USING i;
    FETCH l_cursor INTO l_pad;
    CLOSE l_cursor;
  END LOOP;
END;
```

Without full control over the cursors, it's not possible to write code implementing test case 3. In addition, the database engine isn't able to take advantage of client-side statement caching with native dynamic SQL based on OPEN/FETCH/CLOSE. This means that the only way to solve a parsing problem caused by code using this method is to rewrite it with EXECUTE IMMEDIATE or the dbms_sql package. As a workaround, you could also consider server-side statement caching.

## Dynamic SQL: dbms_sql Package

The dbms_sql package provides full control over the life cycle of cursors. In the following PL/SQL blocks (implementing test case 2), notice how each step is explicitly coded:

```
DECLARE
  l_cursor INTEGER;
  l_pad VARCHAR2(4000);
  l_retval INTEGER;
BEGIN
  FOR i IN 1..10000
  LOOP
    l_cursor := dbms_sql.open_cursor;
    dbms_sql.parse(l_cursor, 'SELECT pad FROM t WHERE val = :1', 1);
    dbms_sql.define_column(l_cursor, 1, l_pad, 10);
    dbms_sql.bind_variable(l_cursor, ':1', i);
    l_retval := dbms_sql.execute(l_cursor);
    IF dbms_sql.fetch_rows(l_cursor) > 0
    THEN
      NULL;
    END IF;
    dbms_sql.close_cursor(l_cursor);
  END LOOP;
END;
```

Since full control over the cursors is given, there is no problem in implementing test case 3. The following PL/SQL block shows an example. Notice how the procedures that prepare (`open_cursor`, `parse`, and `define_column`) and close (`close_cursor`) the cursor are placed outside the loop to avoid unnecessary soft parses.

```
DECLARE
  l_cursor INTEGER;
  l_pad VARCHAR2(4000);
  l_retval INTEGER;
BEGIN
  l_cursor := dbms_sql.open_cursor;
  dbms_sql.parse(l_cursor, 'SELECT pad FROM t WHERE val = :1', 1);
  dbms_sql.define_column(l_cursor, 1, l_pad, 10);
  FOR i IN 1..10000
  LOOP
    dbms_sql.bind_variable(l_cursor, ':1', i);
    l_retval := dbms_sql.execute(l_cursor);
    IF dbms_sql.fetch_rows(l_cursor) > 0
    THEN
      NULL;
    END IF;
  END LOOP;
  dbms_sql.close_cursor(l_cursor);
END;
```

The database engine isn't able to take advantage of client-side statement caching with the `dbms_sql` package. So, in order to optimize an application that is suffering because of too many soft parses (as test case 2 is), you must modify it to reuse the cursors (as test case 3 does). As a workaround, you could consider server-side statement caching.

## OCI

OCI is a low-level application programming interface. Consequently, it provides full control over the life cycle of cursors. For example, in the following code snippet, which implements test case 2, notice how every step is explicitly coded:

```
for (i=1 ; i<=10000 ; i++)
{
  OCIStmtPrepare2(svc, (OCIStmt **)&stm, err, sql, strlen(sql), NULL, 0, OCI_NTV_SYNTAX,
                  OCI_DEFAULT);
  OCIDefineByPos(stm, &def, err, 1, val, sizeof(val), SQLT_STR, 0, 0, 0,  OCI_DEFAULT);
  OCIBindByPos(stm, &bnd, err, 1, &i, sizeof(i), SQLT_INT, 0, 0, 0, 0, 0, OCI_DEFAULT);
  OCIStmtExecute(svc, stm, err, 0, 0, 0, 0, OCI_DEFAULT);
  if (r = OCIStmtFetch2(stm, err, 1, OCI_FETCH_NEXT, 0, OCI_DEFAULT) == OCI_SUCCESS)
  {
    // do something with data...
  }
  OCIStmtRelease(stm, err, NULL, 0, OCI_DEFAULT);
}
```

Since full control over the cursors is available, it's possible to implement test case 3 as well. The following code snippet is an example. Notice how the functions that prepare (`OCIStmtPrepare2` and `OCIDefineByPos`) and close (`OCIStmtRelease`) the cursor are placed outside the loop to avoid unnecessary soft parses.

```
OCIStmtPrepare2(svc, (OCIStmt **)&stm, err, sql, strlen(sql), NULL, 0, OCI_NTV_SYNTAX,
                OCI_DEFAULT);
OCIDefineByPos(stm, &def, err, 1, val, sizeof(val), SQLT_STR, 0, 0, 0, OCI_DEFAULT);
for (i=1 ; i<=10000 ; i++)
{
  OCIBindByPos(stm, &bnd, err, 1, &i, sizeof(i), SQLT_INT, 0, 0, 0, 0, 0, OCI_DEFAULT);
  OCIStmtExecute(svc, stm, err, 0, 0, 0, 0, OCI_DEFAULT);
  if (r = OCIStmtFetch2(stm, err, 1, OCI_FETCH_NEXT, 0, OCI_DEFAULT) == OCI_SUCCESS)
  {
    // do something with data...
  }
}
OCIStmtRelease(stm, err, NULL, 0, OCI_DEFAULT);
```

OCI not only enables full control of the cursors but also supports client-side statement caching. To use it, it's necessary only to enable statement caching and use the `OCIStmtPrepare2` and `OCIStmtRelease` functions (as the previous examples do). Cursors are added to the cache when the `OCIStmtRelease` function is called. Then, when a new cursor is created through the `OCIStmtPrepare2` function, the cache is consulted to find out whether a SQL statement with the same text is present in it.

Different methods exist to enable statement caching. Basically, though, it's only a matter of specifying it when the session is opened or retrieved from a pool. For example, if a nonpooled session is opened through the `OCILogon2` function, it's necessary to specify the `OCI_LOGON2_STMTCACHE` value as the mode.

```
OCILogon2(env, err, &svc, username, strlen(username), password, strlen(password),
          dbname, strlen(dbname), OCI_LOGON2_STMTCACHE)
```

By default, the size of the cache is 20. The following code snippet shows how to change it to 50 by setting the `OCI_ATTR_STMTCACHESIZE` attribute on the service context. Note that setting this attribute to 0 disables statement caching.

```
ub4 size = 50;
OCIAttrSet(svc, OCI_HTYPE_SVCCTX, &size, 0, OCI_ATTR_STMTCACHESIZE, err);
```

The C code examples provided in this section are excerpts from the `ParsingTest1.c`, `ParsingTest2.c`, and `ParsingTest3.c` files implementing test case 1, 2, and 3, respectively.

## JDBC

`java.sql.Statement` is the basic class provided by JDBC to execute SQL statements. As shown in Table 12-1, it's not unlikely that parsing problems will arise when using it. In fact, it doesn't support bind variables, reutilization of cursors, and client-side statement caching. Basically, it's possible to implement only test case 1 with it. The following code snippet demonstrates this:

```
sql = "SELECT pad FROM t WHERE val = ";
for (int i=0 ; i<10000; i++)
{
  statement = connection.createStatement();
  resultset = statement.executeQuery(sql + Integer.toString(i));
  if (resultset.next())
  {
    pad = resultset.getString("pad");
  }
  resultset.close();
  statement.close();
}
```

To avoid all the hard parses performed by the previous code snippet, you must use the `java.sql.PreparedStatement` class (or one of its subclasses), which is a subclass of `java.sql.Statement`. The following code snippet shows how to use it to implement test case 2. Notice how the value used for the lookup, instead of being concatenated to the `sql` variable (as in the previous example), is defined through a bind variable (defined with a question mark in Java and called *placeholder*).

```
sql = "SELECT pad FROM t WHERE val = ?";
for (int i=0 ; i<10000; i++)
{
  statement = connection.prepareStatement(sql);
  statement.setInt(1, i);
  resultset = statement.executeQuery();
  if (resultset.next())
  {
    pad = resultset.getString("pad");
  }
  resultset.close();
  statement.close();
}
```

The next improvement is to avoid the soft parses as well, that is to say, to implement test case 3. As the following code snippet shows, you can achieve this by moving the code for creating and closing the prepared statement outside the loop:

```
sql = "SELECT pad FROM t WHERE val = ?";
statement = connection.prepareStatement(sql);
for (int i=0 ; i<10000; i++)
{
  statement.setInt(1, i);
  resultset = statement.executeQuery();
  if (resultset.next())
  {
    pad = resultset.getString("pad");
  }
  resultset.close();
}
statement.close();
```

The Oracle JDBC drivers provide two extensions to support client-side statement caching: implicit and explicit statement caching. As the names suggest, while the former requires almost no code change, the latter must be explicitly implemented.

With explicit statement caching, statements are opened and closed by means of Oracle-defined methods. Since this has a huge impact on the code and, compared to implicit statement caching, it's more difficult to write faster code, it's not described here. For more information, refer to the *JDBC Developer's Guide* manual.

With implicit statement caching, prepared statements are added to the cache when the close method is called. Then, when a new prepared statement is instantiated through the prepareStatement method, the cache is checked to find out whether a cursor with the same text is already present in it.

---

■ **Note** Only classes implementing the java.sql.PreparedStatement and java.sql.CallableStatement interfaces support implicit statement caching. In other words, plain statements (based on java.sql.Statement) don't support implicit statement caching.

---

The following lines of code show how implicit statement caching is enabled at the connection level. Be careful: setting the size of the cache to a value greater than 0 is a requirement. The casts are necessary because both methods are Oracle extensions.

```
((oracle.jdbc.OracleConnection)connection).setImplicitCachingEnabled(true);
((oracle.jdbc.OracleConnection)connection).setStatementCacheSize(50);
```

Another way to enable implicit statement caching is through the setImplicitCachingEnabled and setMaxStatements methods of the OracleDataSource class. Note that the setMaxStatements method is deprecated, though.

By default, all prepared statements are cached with implicit statement caching. When the cache is full, the least recently used one is closed and replaced by a new one. If necessary, the caching of a specific statement can also be disabled. The following line of code shows how to do it:

```
((oracle.jdbc.OraclePreparedStatement)statement).setDisableStmtCaching(true);
```

The Java code used for the examples in this section is an excerpt from the ParsingTest1.java, ParsingTest2.java, and ParsingTest3.java files that implement test case 1, 2, and 3, respectively.

## ODP.NET

ODP.NET provides little control over the life cycle of a cursor. In the following code snippet that implements test case 1, the ExecuteReader method triggers parse, execute, and fetch calls at the same time:

```
sql = "SELECT pad FROM t WHERE val = ";
command = new OracleCommand(sql, connection);
for (int i = 0; i < 10000; i++)
{
  command.CommandText = sql + i;
  reader = command.ExecuteReader();
  if (reader.Read())
  {
    pad = reader[0].ToString();
  }
  reader.Close();
}
```

To avoid all the hard parses performed by the previous code snippet, the OracleParameter class has to be used for passing parameters (bind variables). The following code snippet shows how to use it to implement test case 2. Notice how the value used for the lookup, instead of being concatenated to the sql variable (as in the previous example), is defined through a parameter.

```
String sql = "SELECT pad FROM t WHERE val = :val";
OracleCommand command = new OracleCommand(sql, connection);
OracleParameter parameter = new OracleParameter("val", OracleDbType.Int32);
command.Parameters.Add(parameter);
OracleDataReader reader;
for (int i = 0; i < 10000; i++)
{
  parameter.Value = Convert.ToInt32(i);
  reader = command.ExecuteReader();
  if (reader.Read())
  {
    pad = reader[0].ToString();
  }
  reader.Close();
}
```

With ODP.NET, it's not possible to implement test case 3. However, to achieve the same result, you can use client-side statement caching. There are two methods for enabling it and setting the size of the cache. The first, which controls statement caching for all applications using a specific Oracle home, is by setting the following value in the registry. If it's set to 0, statement caching is disabled. Otherwise, statement caching is enabled, and the value specifies the size of the cache (`<Assembly_Version>` is the full version number of `Oracle.DataAccess.dll`).

```
HKEY_LOCAL_MACHINE\SOFTWARE\ORACLE\ODP.NET\<Assembly_Version>\StatementCacheSize
```

The second method controls statement caching directly in the code through the `Statement Cache Size` attribute provided by the `OracleConnection` class. Basically, it plays the same role as the registry value but for a single connection. The following code snippet shows how to enable statement caching and to set its size to 10:

```
String connectString = "User Id=" + user +
                       ";Password=" + password +
                       ";Data Source=" + dataSource +
                       ";Statement Cache Size=10";
OracleConnection connection = new OracleConnection(connectString);
```

Note that the setting at the connection level overrides the setting in the registry. In addition, when statement caching is enabled, it's possible to disable it at the command level by setting the `AddToStatementCache` property to `false`.

The C# code used for the examples in this section is an excerpt from the `ParsingTest1.cs` and `ParsingTest2.cs` files that implement test case 1 and 2, respectively.

## PHP

In PHP, the PECL OCI8 extension provides full control over the life cycle of cursors. For example, in the following code snippet, which implements test case 2, notice how every step is explicitly coded:

```
$sql = "SELECT pad FROM t WHERE val = :val";
for ($i = 1; $i <= 10000; $i++)
{
  $statement = oci_parse($connection, $sql);
  oci_bind_by_name($statement, ":val", $i, -1, SQLT_INT);
  oci_execute($statement, OCI_NO_AUTO_COMMIT);
  if ($row = oci_fetch_assoc($statement))
  {
        $pad = $row['PAD'];
  }
  oci_free_statement($statement);
}
```

Since full control over the cursors is available, it's possible to implement test case 3 as well. The following code snippet is an example. Notice how the functions that prepare (`oci_parse` and `oci_bind_by_name`) and close (`oci_free_statement`) the cursor are placed outside the loop to avoid unnecessary soft parses.

```
$sql = "SELECT pad FROM t WHERE val = :val";
$statement = oci_parse($connection, $sql);
oci_bind_by_name($statement, ":val", $i, -1, SQLT_INT);
for ($i = 1; $i <= 10000; $i++)
{
  oci_execute($statement, OCI_NO_AUTO_COMMIT);
  if ($row = oci_fetch_assoc($statement))
  {
    $pad = $row['PAD'];
  }
}
oci_free_statement($statement);
```

PHP not only enables full control of the cursors but, as of OCI8 1.1, also supports client-side statement caching. To control it, the `oci8.statement_cache_size` directive is available. If it's set to 0, client-side statement caching is disabled. Values higher than 0 enable client-side caching, and specify how many cursors are cached. The default value is 20, enabling client-side caching for up to 20 cursors. To change the value, add a line like the following to the `php.ini` configuration file:

```
oci8.statement_cache_size = 50
```

The PHP code used for the examples in this section comes from excerpts from the `ParsingTest1.php`, `ParsingTest2.php`, and `ParsingTest3.php` files. Those files implement test cases 1, 2, and 3, respectively.

## On to Chapter 13

This chapter describes how to identify, solve, and work around parsing problems. The key message is that by knowing how your application works and the possibilities given by the used application programming interface, you should be able to avoid parsing problems by writing efficient code during the development stage.

Since in the life cycle of a cursor the execution phase follows the parsing of the SQL statement and the binding of variables, it's necessary to know the different techniques used by the database engine to access data. The next chapter discusses this, and describes how to take advantage of the different types of indexes and partitioning methods, in order to help speed up the execution of SQL statements.

CHAPTER 13

# Optimizing Data Access

An execution plan, as described in Chapter 10, is composed of several operations. The most commonly used operations are those that access, filter, and transform data. This chapter specifically deals with data access operations, or, in other words, how the database engine is able to access data.

There are basically only two ways to locate data in a table. The first is to scan the whole table. The second is to do a lookup based on a redundant access structure (for example, an index) or on the structure containing the table itself (for example, a hash cluster). In addition, in the case of partitioning, access might be restricted to a subset of partitions. This is no different from looking up specific information in this book. Either you read the whole book, you read a single chapter, or you use the index or table of contents to find out where the information you're looking for is.

The first part of this chapter describes how to recognize inefficient access paths by looking at runtime statistics provided by either SQL trace or dynamic performance views. The second part describes available access methods and when you should take advantage of them. For each access path, the hint that you can use to reproduce it and the execution plan operation related to it are described as well.

---

**Note** In this chapter, several SQL statements contain hints. I do that not only to show you which hint leads to which access path, but also to show you examples of their use. In any case, neither real references nor full syntaxes are provided. You can find these in Chapter 2 of the *SQL Reference* manual.

---

## Identifying Suboptimal Access Paths

Chapter 10 describes how to judge the efficiency of an execution plan by checking both the estimations of the query optimizer and whether restrictions are correctly recognized. It's important to understand that even when the query optimizer correctly chooses the optimal execution plan, it doesn't necessarily mean that this specific execution plan will perform well. It might be that by altering the SQL statement or the access structures (for example, adding an index), an even better execution plan could be taken into consideration. The following sections describe additional checks that can be performed to help recognize an inefficient access path, what might be causing it, and what you can do to avoid the problem.

### Identification

The most efficient access path is able to process the data by consuming the least amount of resources. Therefore, to recognize whether an access path is efficient, you have to recognize whether the amount of resources used for its processing is acceptable. To do so, it's necessary to define both how to measure the utilization of resources and what "acceptable" means. In addition, you need to also consider the feasibility of the check. In other words, you also need to consider how much effort is needed to implement a check. It has to be as simple as possible. In fact, a perfect check that takes too much time to be implemented isn't acceptable in practice, especially if you need to work on tens, or even hundreds, of SQL statements that are candidates for optimization, or simply because you're working on a tight schedule.

As a side note, keep in mind that this section focuses on efficiency, not on speed alone. It's essential to understand that the most efficient access path isn't always the fastest one. As described in Chapter 15, with parallel processing, it's sometimes possible to achieve a better response time even though the amount of resources used is higher. Of course, when you consider the whole system, the fewer resources used by SQL statements (in other words, the higher their efficiency is), the more scalable, and faster, the system is. This is true because, by definition, resources are limited.

As a first approximation, the amount of resources used by an access path is acceptable when it's proportional to the amount of returned rows (that is, the number of rows that are returned to the parent operation in the execution plan). In other words, when few rows are returned, the expected utilization of resources is low, and when lots of rows are returned, the expected utilization of resources is high. Consequently, the check should be based on the amount of resources used to return a single row.

In an ideal world, you would like to measure the resource consumption by considering all four main types of resources used by the database engine: CPU, memory, the disk, and the network. Certainly, this can be done, but unfortunately getting and assessing all these figures takes a lot of time and effort and can usually be done only for a limited number of SQL statements in an optimization session. You should also consider that when processing a row, the CPU processing time depends on the speed of the processor, which obviously changes from system to system. Further, the amount of memory used is all but proportional to the number of returned rows, and the disk and network resources aren't always used. It's, in fact, not uncommon at all to see long-running SQL statements that use a modest amount of memory and are without disk or network access.

Fortunately, there's a single database metric, which is very easy to collect, that can tell you a lot about the amount of work done by the database engine: the number of logical reads—that is, the number of blocks that are accessed during the execution of a SQL statement. There are five good reasons for this. First, a logical read is a CPU-bound operation and, therefore, reflects CPU utilization very well. Second, a logical read might lead to a physical read, and therefore, if you reduce the number of logical reads, you likely reduce the disk I/O operations as well. Third, a logical read is an operation subject to serialization. Because you usually have to optimize for a multiuser load, minimizing the logical reads is good for avoiding scalability problems. Fourth, the number of logical reads is readily available at the SQL statement and execution plan operation levels, in both SQL trace files and dynamic performance views. And fifth, the number of logical reads is independent from the load to which the CPU and the disk I/O subsystem are subject.

Because logical reads are very good at approximating overall resource consumption, you can concentrate (at least for the first round of optimization) on access paths that have a high number of logical reads per returned rows. The following are generally considered good "rules of thumb":

- Access paths that lead to less than about 5 logical reads per returned row are probably good.
- Access paths that lead to up to about 10–15 logical reads per returned row are probably acceptable.
- Access paths that lead to more than about 20 logical reads per returned row are probably inefficient. In other words, there's probably room for improvement.

To check the number of logical reads per row, there are basically two methods. The first is to take advantage of the execution statistics provided by dynamic performance views and displayed using the `dbms_xplan` package (Chapter 10 fully describes this technique). The following execution plan was generated using that method. From it, for each operation, you can see how many rows were returned (A-Rows column) and how many logical reads were performed (Buffers column) in order to return them:

```
SELECT * FROM t WHERE n1 BETWEEN 6000 AND 7000 AND n2 = 19

-----------------------------------------------------------------
| Id  | Operation                   | Name   | A-Rows | Buffers |
-----------------------------------------------------------------
|   0 | SELECT STATEMENT            |        |      3 |      28 |
|*  1 |  TABLE ACCESS BY INDEX ROWID| T      |      3 |      28 |
|*  2 |   INDEX RANGE SCAN          | T_N2_I |     24 |       4 |
-----------------------------------------------------------------

   1 - filter(("N1">=6000 AND "N1"<=7000))
   2 - access("N2"=19)
```

The second method is to make use of the information provided by SQL trace (Chapter 3 fully describes this technique). The following is an excerpt of the output generated by TKPROF for the very same query as in the previous example. Note that the number of returned rows (Rows column) and logical reads (cr attribute) match the previous figures:

```
Rows     Row Source Operation
-------  ---------------------------------------------------
      3  TABLE ACCESS BY INDEX ROWID T (cr=28 pr=0 pw=0 time=80 us)
     24   INDEX RANGE SCAN T_N2_I (cr=4 pr=0 pw=0 time=25 us)(object id 39684)
```

Based on the rules of thumb mentioned earlier, the execution plan used as an example is acceptable. In fact, the number of logical reads per returned row for the access path is about 9 (28/3). Let's see what a bad execution plan looks like for the very same SQL statement. Note that it's bad because the number of logical reads per returned row for the access path is 130 (390/3), not because it contains a full table scan!

```
-----------------------------------------------------
| Id  | Operation         | Name | A-Rows | Buffers |
-----------------------------------------------------
|   0 | SELECT STATEMENT  |      |      3 |     390 |
|*  1 |  TABLE ACCESS FULL| T    |      3 |     390 |
-----------------------------------------------------

   1 - filter(("N2"=19 AND "N1">=6000 AND "N1"<=7000))
```

It's important to stress that this section is about access paths. So, you must consider the figures at the access-path level only, not for the whole SQL statement. In fact, the figures at the SQL statement level might be misleading. To understand what the problem might be, let's examine the following query. If only the figures at the SQL statement level (provided by operation 0) are erroneously taken into consideration, 387 logical reads are executed to return a single row. In other words, it would be erroneously classified as inefficient. However, if the figures of the access operation (operation 2) are correctly taken into consideration instead, the ratio between the number of logical reads (387) and the number of returned rows (160) classifies this access path as efficient. The problem in this case is that operation 1 is used to apply the sum function to the rows returned by operation 2. As a result, it always returns a single row and "hides" the access path performance figures:

```
SELECT sum(n1) FROM t WHERE n2 > 246
```

```
------------------------------------------------------
| Id  | Operation          | Name | A-Rows | Buffers |
------------------------------------------------------
|   0 | SELECT STATEMENT   |      |      1 |     387 |
|   1 |  SORT AGGREGATE    |      |      1 |     387 |
|*  2 |   TABLE ACCESS FULL| T    |    160 |     387 |
------------------------------------------------------

   2 - filter("N2">246)
```

If you really have no choice but to look at the figures at the SQL statement level (for example, because the SQL trace file doesn't contain the execution plan), you'll have a hard time using the rules of thumb provided earlier, simply because you don't have enough information. In this case, however, at least for simple SQL statements, you might try to guess the access path figures and adapt the rule of thumb. For example, you might carefully review the SQL statement to check whether there isn't an aggregation in it, find out how many tables are referenced in the SQL statement, and then increase the limits in the rules of thumb in proportion to the number of referenced tables.

## Pitfalls

While examining the number of logical reads, you must be aware of two pitfalls that might distort the figures. The first is related to read consistency, and the second is related to row prefetching.

### Read Consistency

For every SQL statement, the database engine has to guarantee the consistency of the processed data. For that purpose, based on current data blocks and undo blocks, consistent copies of data blocks might be created at runtime. To execute such an operation, several logical reads are performed. Therefore, the number of logical reads performed by an access path operation is strongly dependent on the number of blocks that have to be reconstructed. The following excerpt of the output generated by the `read_consistency.sql` script shows that behavior. Note that the query is the same as the one used in the previous section. According to the execution statistics, the same number of rows was returned (actually it returns the same data). However, many more logical reads were performed (in total 354, compared to 28). That effect is because of another session that modified the blocks needed to process this query. Because the changes weren't committed at the time the query was started, the database engine had to reconstruct the blocks. This led to a much higher number of logical reads:

```
SELECT * FROM t WHERE n1 BETWEEN 6000 AND 7000 AND n2 = 19

-----------------------------------------------------------------
| Id  | Operation                   | Name    | A-Rows | Buffers |
-----------------------------------------------------------------
|   0 | SELECT STATEMENT            |         |      3 |     354 |
|*  1 |  TABLE ACCESS BY INDEX ROWID| T       |      3 |     354 |
|*  2 |   INDEX RANGE SCAN          | T_N2_I  |     24 |     139 |
-----------------------------------------------------------------

   1 - filter(("N1">=6000 AND "N1"<=7000))
   2 - access("N2"=19)
```

## Row Prefetching

From a performance point of view, you should always avoid row-based processing. For example, when a client retrieves data from a database, it can do it row by row or, better yet, by retrieving several rows at the same time. This technique, known as *row prefetching*, is fully described in Chapter 15. For the moment, let's just look at its impact on the number of logical reads. Simply put, a logical read is counted each time the database engine accesses a block. With a full table scan, there are two extremes. If row prefetching is set to 1, approximately one logical read per returned row is performed. If row prefetching is set to a number greater than the number of rows stored in each table's block, the number of logical reads is close to the number of the table's blocks. The following excerpt of the output generated by the `row_prefetching.sql` script shows this behavior. In the first execution, with row prefetching set to 2 (the choice of this value is explained in the "Row Prefetching" section in Chapter 15), the number of logical reads (5,388) is about half of the number of rows (10,000). In the second execution, because the number of prefetched rows (100) is higher than the average number of rows per block (25), the number of logical reads (488) is about the same as the number of blocks (401):

```
SQL> SELECT num_rows, blocks, round(num_rows/blocks) AS rows_per_block
  2  FROM user_tables
  3  WHERE table_name = 'T';

NUM_ROWS BLOCKS ROWS_PER_BLOCK
-------- ------ --------------
   10000    401             25

SQL> set arraysize 2

SQL> SELECT * FROM t;

-----------------------------------------------------
| Id  | Operation          | Name | A-Rows | Buffers |
-----------------------------------------------------
|   0 | SELECT STATEMENT   |      |  10000 |    5388 |
|   1 |  TABLE ACCESS FULL| T    |  10000 |    5388 |
-----------------------------------------------------

SQL> set arraysize 100

-----------------------------------------------------
| Id  | Operation          | Name | A-Rows | Buffers |
-----------------------------------------------------
|   0 | SELECT STATEMENT   |      |  10000 |     488 |
|   1 |  TABLE ACCESS FULL| T    |  10000 |     488 |
-----------------------------------------------------
```

■ **Note** In SQL*Plus, you manage the number of prefetched rows through the `arraysize` system variable. The default value is 15.

Given the dependency of the number of logical reads on row prefetching, whenever you execute a SQL statement for testing purposes in a tool such as SQL*Plus, you should carefully set row prefetching like the application does. In other words, the tool you use for the tests should prefetch the same number of rows as the application. Failing to do so may cause severely misleading results.

When blocking operations (for example, aggregation operations) are executed, the SQL engine uses row prefetching internally. As a result, when aggregations are part of the execution plan, the number of logical reads of an access path is very close to the number of blocks. In other words, every time the SQL engine accesses a block, it extracts all rows contained in it, regardless of the row prefetching setting. The following example illustrates this:

```
SQL> set arraysize 2

SQL> SELECT sum(n1) FROM t;

------------------------------------------------------
| Id  | Operation          | Name | A-Rows | Buffers |
------------------------------------------------------
|   0 | SELECT STATEMENT   |      |      1 |     388 |
|   1 |  SORT AGGREGATE    |      |      1 |     388 |
|   2 |   TABLE ACCESS FULL| T    |  10000 |     388 |
------------------------------------------------------
```

## Causes

There are several main causes of inefficient access paths:

- No suitable access structures (for example, indexes) are available.
- A suitable access structure is available, but the syntax of the SQL statement doesn't allow the query optimizer to use it.
- The table or the index is partitioned, but no pruning is possible. As a result, all partitions are accessed.
- The table or the index, or both, aren't suitably partitioned.

In addition to the examples in the previous list, two additional situations lead to inefficient access paths:

- When the query optimizer makes wrong estimations because of a lack of object statistics, because of object statistics that aren't up-to-date, or because a wrong query optimizer configuration is in place. I don't cover that here, because I assume that the necessary object statistics are in place and that the query optimizer is correctly configured (Chapters 8 and 9 fully described these two topics).
- When the query optimizer is itself the problem, for example, when there are internal bugs or limitations in how it works. I don't deal with this either, because bugs or query optimizer limitations are responsible for a very limited number of problems.

## Solutions

As described in the previous sections, to efficiently execute a SQL statement, the objective is to minimize the number of logical reads or, in other words, to use the access path that accesses fewer blocks. To reach this objective, it may be necessary to add new access structures (for example, indexes) or change the physical layout (for example, partition some tables or their indexes). Given a SQL statement, there are many combinations of access structures and physical layouts. Luckily, to make the choice easier, it's possible to classify SQL statements (or better, data access operations) in two main categories with regard to selectivity:

- Operations with weak selectivity
- Operations with strong selectivity

The selectivity is important because the access structures and layouts that work well with operations with very weak selectivity work badly for operations with very strong selectivity, and vice versa. Be careful, however, that there's no fixed boundary between these two categories. Instead, it depends on the operation, on the data it processes, and on how this data is stored. For example, both data distribution and the number of rows per block strongly impact performance. In other words, it's absolutely wrong to say that selectivity up to 0.1 (or any other value you could think of) is necessarily strong, and above this value, it's necessarily weak. In spite of this, it may be said that, in practice, the limit commonly ranges between 0.05 and 0.25. As Figure 13-1 shows, only for values close to 0 or 1 can you be certain.

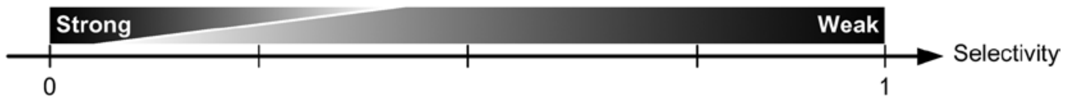


***Figure 13-1.** There's no fixed limit between strong and weak selectivity*

It's essential to understand that for determining the category of an operation, the absolute number of rows it returns isn't relevant. Only the selectivity is. For example, knowing that an operation returns 500,000 rows isn't relevant at all to choosing an access path. In contrast, knowing that the operation has a selectivity of 0.001 clearly puts it in the strong selectivity category.

The category is important in order to have a clue about the type of access path that should lead to an efficient execution plan. Figure 13-2 broadly correlates the selectivity with the access path that is usually optimal. Operations with strong selectivities are executed efficiently when a suitable index is in place. As you'll see later in this chapter, in some situations a rowid access or hash cluster might also be helpful. On the other hand, operations with very weak selectivities are processed efficiently by reading the whole table. In between these two possibilities, partitioned tables and hash clusters play an important role.

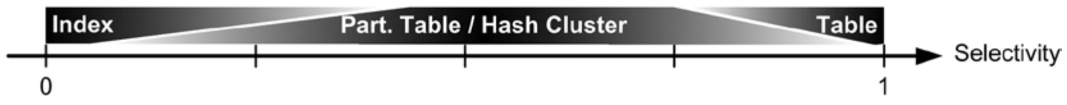


***Figure 13-2.** Specific access paths work efficiently only for a specific range of selectivity*

---

■ **Note** When a data file is stored on an Exadata storage server, operations using smart scans can take advantage of *storage indexes* to reduce the amount of data that has to be physically read from disk. As a result, some operations with an average selectivity, or even a quite strong selectivity, might be efficiently executed by reading the whole table. We don't have any control over storage indexes, as they're automatically maintained by the Exadata storage servers. Thus, storage indexes aren't covered in this chapter.

---

Let's take a look at two tests to demonstrate this. In the first test, a single row is retrieved, while in the second, thousands of rows are retrieved.

## Retrieving a Single Row

The aim of this test, which is based on the access_structures_1.sql script, is to compare the number of logical reads necessary to retrieve a single row, with the following access structures in place:

- A heap table with a primary key
- An index-organized table
- A single-table hash cluster that has the primary key as the cluster key

---

■ **Note** This chapter describes only how to take advantage of different types of segments (for example, tables, clusters, and indexes) to minimize the number of logical reads performed during the processing of SQL statements. You can find basic information about them in the *Oracle Database Concepts* manual, specifically in the "Schema Objects" chapter.

---

The queries used for the tests are the following. Note that the id column is the primary key of the table. A row with the value 6 exists, and the rid variable stores the rowid of that row:

```
SELECT * FROM sales WHERE id = 6

SELECT * FROM sales WHERE rowid = :rid
```

Because the number of logical reads depends on the height of the index, the test is performed while 10, 10,000, and 1,000,000 rows are being stored in the table. Figure 13-3 summarizes the results. They illustrate four main facts:

- For all access structures, a single logical read is performed through a rowid (obviously, to read the block where the row is stored, you can't do less work than this).
- For the heap table, at least two logical reads are necessary: one for the index and one for the table. As the number of rows increases, the height of the index increases, and the number of logical reads increases as well.
- An access through the index-organized table requires one less logical read than through the heap table.
- For the single-table hash cluster, not only is the number of logical reads independent of the number of rows, but in addition, it always leads to a single logical read.

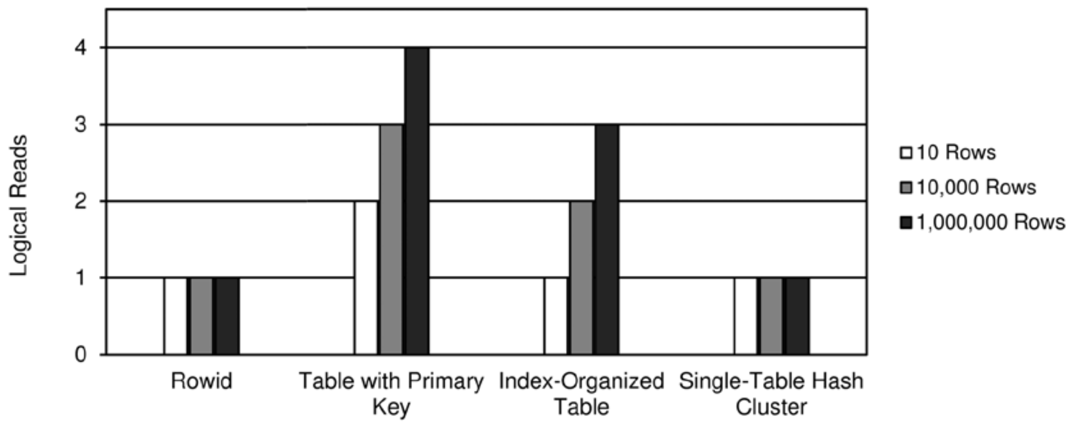


***Figure 13-3.*** *Different access structures lead to different numbers of logical reads*

In summary, for retrieving a single row, a "regular" table with an index is the least efficient access structure. However, as I describe later in this chapter, "regular" tables are the most commonly used because you can take advantage of the other access structures only in specific situations.

## Retrieving Thousands of Rows

The purpose of this test, which is based on the `access_structures_1000.sql` script, is to compare the number of logical reads necessary to retrieve thousands of rows with the following access structures in place:

- Nonpartitioned table without an index.
- List-partitioned table. The `prod_category` column is the partition key.
- Single-table hash cluster. The `prod_category` column is the cluster key.
- Nonpartitioned table with an index on the `prod_category` column. For this test, two different physical distributions of the rows in the table segment (and hence, different clustering factors) were tested.

The test data set consists of 918,843 rows. The following query shows the distribution of the values for the `prod_category` column:

```
SQL> SELECT prod_category, count(*), ratio_to_report(count(*)) over() AS selectivity
  2  FROM sales
  3  GROUP BY prod_category
  4  ORDER BY count(*);

PROD_CATEGORY    COUNT(*) SELECTIVITY
-------------- ---------- -----------
Hardware            15357        .017
Photo               95509        .104
Electronics        116267        .127
Peripherals        286369        .312
Software/Other     405341        .441
```

The queries used for the tests are the following:

```
SELECT sum(amount_sold) FROM sales WHERE prod_category = 'Hardware'

SELECT sum(amount_sold) FROM sales WHERE prod_category = 'Photo'

SELECT sum(amount_sold) FROM sales WHERE prod_category = 'Electronics'

SELECT sum(amount_sold) FROM sales WHERE prod_category = 'Peripherals'

SELECT sum(amount_sold) FROM sales WHERE prod_category = 'Software/Other'

SELECT sum(amount_sold) FROM sales
```

For each of them, the number of logical reads was measured. Figure 13-4 summarizes the results, which lead to four main facts:

- The number of logical reads needed to read the nonpartitioned table without an index is independent of the selectivity. Therefore, it's efficient only when the selectivity is weak.
- The number of logical reads needed to read a single partition of the list-partitioned table is proportional to the selectivity, because the table has been partitioned according to the `prod_category` column. Therefore, in all situations, a minimal number of logical reads is carried out.
- The number of logical reads needed to read the single-table hash cluster is proportional only to the selectivity for medium and high values. (As you'll see later, hash clusters might be very useful when the selectivity is very strong. In this test, however, they're at a disadvantage because of the nonuniform data distribution.)
- The number of logical reads needed to read the table through an index is highly dependent on the physical distribution of data. Therefore, knowing only the selectivity isn't enough to find out whether such an access path might process data efficiently.

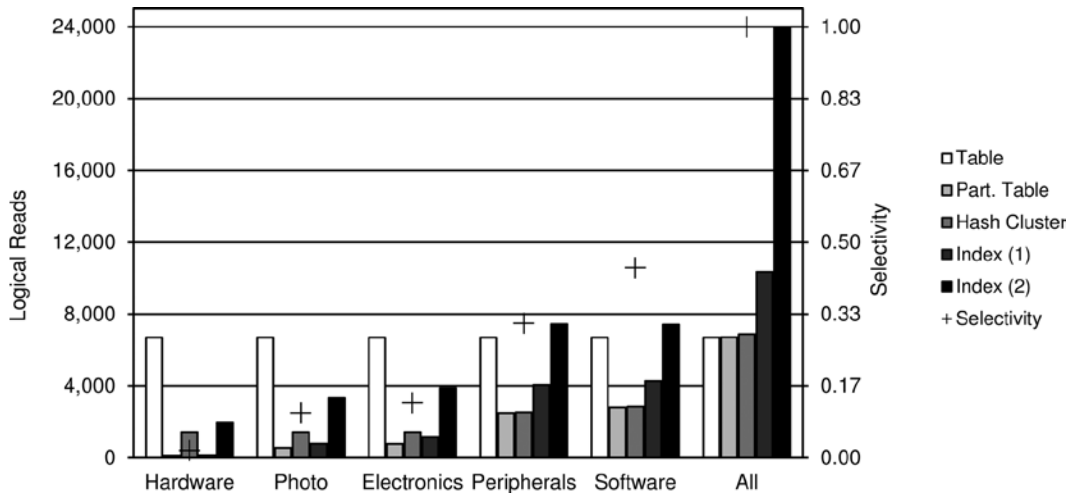


***Figure 13-4.** Specific access paths work efficiently only for a specific range of selectivity*

Now that you've seen what the main available options are that can access data efficiently in different situations, it's time to describe in detail the access paths used to process SQL statements with weak and strong selectivity.

## SQL Statements with Weak Selectivity

To process data efficiently, SQL statements with weak selectivity have to use either a full table scan or a full partition scan. But in plenty of situations, only full table scans come into play. There are three main reasons for this. First, partitioning is an Enterprise Edition option. So, you can't take advantage of it if you're using Standard Edition or, of course, if you don't have the Partitioning option license to use it. Second, even if you're allowed to use the Partitioning option, not all tables will be partitioned in practice. Third, a table might be partitioned by only a limited number of columns. As a result, even if a table is partitioned, not all SQL statements that reference it will be able to take advantage of partitioning, unless all of them reference the partitioning key(s), which is usually not the case in practice.

In particular situations, both full table scans and full partition scans might be avoided by replacing them with full index scans. In such cases, the idea is to take advantage of indexes not for the purpose of searching for particular values, but simply because they're smaller than tables.

### Full Table Scans

It's possible to perform a full table scan on all heap tables. Because there are no particular requirements for doing this type of scan, on occasion it may be the only access path possible. The following query is an example. Note that in the execution plan, the `TABLE ACCESS FULL` operation corresponds to the full table scan. The example also shows how to force a full table scan with the `full` hint:

```
SELECT /*+ full(t) */ * FROM t WHERE n2 = 19

-----------------------------------
| Id  | Operation         | Name |
-----------------------------------
|   0 | SELECT STATEMENT  |      |
|*  1 |  TABLE ACCESS FULL| T    |
-----------------------------------

   1 - filter("N2"=19)
```

During full table scans, server processes read all of the table's blocks below the high watermark sequentially. Server processes up to and including version 10.2 perform buffer cache reads. From version 11.1 onward, the type of disk I/O operation depends on the number of blocks to be read, the fraction of the target table's blocks that are already in the buffer cache, and whether the `BUFFER_POOL` storage parameter is set to `KEEP`. Simply put, when the number of blocks to be read from the disk is low or the `KEEP` buffer pool is used, server processes perform buffer cache reads. Otherwise, they perform direct reads. This option to perform direct reads is a major enhancement to make sure that large amounts of data aren't unnecessarily loaded through the buffer cache (from where they will be immediately discarded).

The minimum number of logical reads performed by a full table scan depends on the number of *blocks*, not on the number of *rows*. This can lead to suboptimal performance if the table contains a lot of empty or almost-empty blocks. Clearly, a block has to be read to know whether it contains data. One of the most common scenarios that can lead to a table with a lot of sparsely populated blocks is when tables are subject to more deletes than inserts. The following example, which is an excerpt of the output generated by the `full_scan_hwm.sql` script, illustrates this:

- At the beginning, a query leads to 468 logical reads in order to return 40 rows:

```
SQL> SELECT * FROM t WHERE n2 = 19;

SQL> SELECT last_output_rows, last_cr_buffer_gets, last_cu_buffer_gets
  2  FROM v$session s, v$sql_plan_statistics p
  3  WHERE s.prev_sql_id = p.sql_id
  4  AND s.prev_child_number = p.child_number
  5  AND s.sid = sys_context('userenv','sid')
  6  AND p.operation_id = 1;

LAST_OUTPUT_ROWS LAST_CR_BUFFER_GETS LAST_CU_BUFFER_GETS
---------------- ------------------- -------------------
              40                 468                   1
```

- Then, almost all rows (9,960 out of 10,000) are deleted. However, the number of logical reads needed to execute the query doesn't change. In other words, a lot of completely empty blocks were uselessly accessed:

```
SQL> DELETE t WHERE n2 <> 19;

9960 rows deleted.

SQL> SELECT * FROM t WHERE n2 = 19;

SQL> SELECT last_output_rows, last_cr_buffer_gets, last_cu_buffer_gets
  2  FROM v$session s, v$sql_plan_statistics p
  3  WHERE s.prev_sql_id = p.sql_id
  4  AND s.prev_child_number = p.child_number
  5  AND s.sid = sys_context('userenv','sid')
  6  AND p.operation_id = 1;

LAST_OUTPUT_ROWS LAST_CR_BUFFER_GETS LAST_CU_BUFFER_GETS
---------------- ------------------- -------------------
              40                 468                   1
```

- To lower the high watermark, a physical reorganization of the table is necessary. If the table is stored in a tablespace with automatic segment space management, you can do this with the following SQL statements. Note that row movement must be activated because rows might get a new rowid during the reorganization:

```
SQL> ALTER TABLE t ENABLE ROW MOVEMENT;

SQL> ALTER TABLE t SHRINK SPACE;
```

- After the reorganization, the query performs only 24 logical reads in order to return 40 rows:

```
SQL> SELECT * FROM t WHERE n2 = 19;
SQL> SELECT last_output_rows, last_cr_buffer_gets, last_cu_buffer_gets
  2  FROM v$session s, v$sql_plan_statistics p
  3  WHERE s.prev_sql_id = p.sql_id
  4  AND s.prev_child_number = p.child_number
  5  AND s.sid = sys_context('userenv','sid')
  6  AND p.operation_id = 1;

LAST_OUTPUT_ROWS LAST_CR_BUFFER_GETS LAST_CU_BUFFER_GETS
---------------- ------------------- -------------------
              40                  23                   0
```

Remember that the number of logical reads performed by a full table scan strongly depends on the setting for row prefetching. Refer to the “Row Prefetching” section earlier in this chapter for an example of this.

## Full Partition Scans

When the selectivity is very weak (that is, close to 1), full table scans are the most efficient way to access data. As soon as the selectivity decreases, many blocks are unnecessarily accessed by full table scans. Because the use of indexes isn’t beneficial with weak selectivity, partitioning is the most common option used to reduce the number of logical reads. The reason for using partitioning is to take advantage of the query optimizer’s ability to exclude the processing of partitions that contain processing-irrelevant data *a priori*. This feature is called *partition pruning*.

There are two basic prerequisite conditions to capitalize on partition pruning for a given SQL statement. First, and obviously, a table must be partitioned. Second, a restriction or a join condition on the partition key must be specified in the SQL statement. If these two requirements are met, the query optimizer can replace a full table scan by one or several full partition scans. In practice, though, things aren’t that easy. In fact, the query optimizer has to deal with several particular situations that might, or might not, lead to partition pruning. To understand these situations better, the following sections detail partition pruning basics, as well as more advanced pruning techniques such as OR, multicolumn, subquery, and join-filter pruning. These are followed by some practical advice on how to implement partitioning. Note that partitioned indexes are discussed in the “SQL Statements with Strong Selectivity” section later in this chapter.

## Range Partitioning

To illustrate how partition pruning works, let’s examine several examples based on the pruning_range.sql script. The test table is range partitioned and created with the following SQL statement. To be able to show all types of partition pruning, the partition key is composed of two columns: n1 and d1. The table is partitioned by four different values of the n1 column and by month based on d1 column. This means there are 48 partitions per year:

```
CREATE TABLE t (
  id NUMBER,
  d1 DATE,
  n1 NUMBER,
  n2 NUMBER,
  n3 NUMBER,
  pad VARCHAR2(4000),
  CONSTRAINT t_pk PRIMARY KEY (id)
)
PARTITION BY RANGE (n1, d1) (
  PARTITION t_1_jan_2014 VALUES LESS THAN (1, to_date('2014-02-01','yyyy-mm-dd')),
  PARTITION t_1_feb_2014 VALUES LESS THAN (1, to_date('2014-03-01','yyyy-mm-dd')),
  PARTITION t_1_mar_2014 VALUES LESS THAN (1, to_date('2014-04-01','yyyy-mm-dd')),
  ...
  PARTITION t_4_oct_2014 VALUES LESS THAN (4, to_date('2014-11-01','yyyy-mm-dd')),
  PARTITION t_4_nov_2014 VALUES LESS THAN (4, to_date('2014-12-01','yyyy-mm-dd')),
  PARTITION t_4_dec_2014 VALUES LESS THAN (4, to_date('2015-01-01','yyyy-mm-dd'))
)
```

■ **Caution** In a case like this one where two partition keys exist, the database engine uses the second key for inserting new rows only if the first key can't uniquely identify a single partition. For this reason, when specifying the `PARTITION BY RANGE` clause, the `n1` column is specified before the `d1` column.

Figure 13-5 is a graphical representation of the test table.

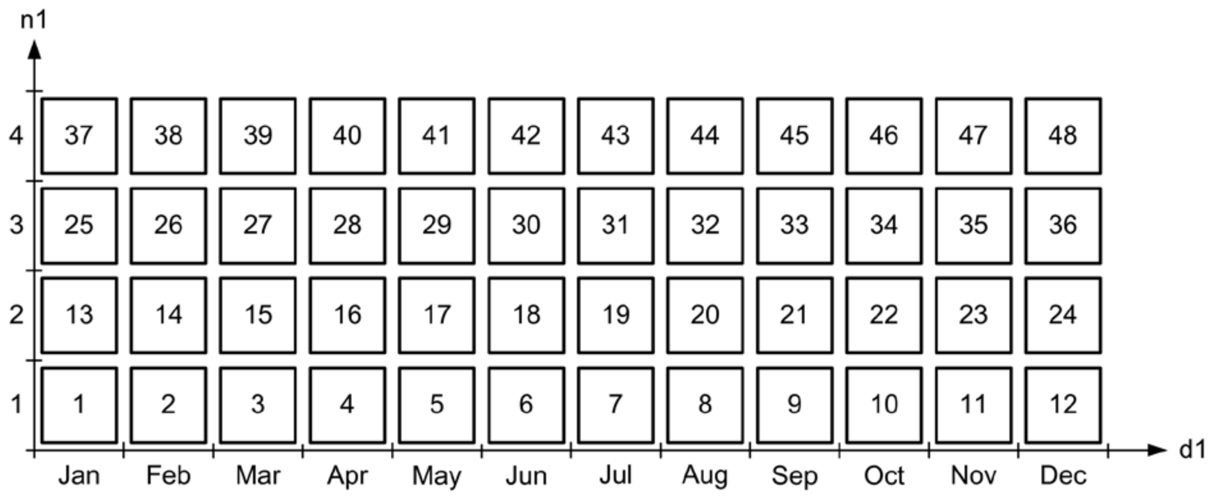


***Figure 13-5.*** *The test table is composed of 48 partitions per year*

Each partition can be identified by either its name or its "position" in the table (the latter is shown in Figure 13-5). Of course, the mapping between these two values is available in the data dictionary. The following query shows how to get it from the `user_tab_partitions` view:

```
SQL> SELECT partition_name, partition_position
  2  FROM user_tab_partitions
  3  WHERE table_name = 'T'
  4  ORDER BY partition_position;

PARTITION_NAME PARTITION_POSITION
-------------- ------------------
T_1_JAN_2014                    1
T_1_FEB_2014                    2
T_1_MAR_2014                    3
...
```

```
T_4_OCT_2014                    46
T_4_NOV_2014                    47
T_4_DEC_2014                    48
```

With such a table, if a restriction is applied to the partition key, the query optimizer recognizes it and whenever possible excludes partitions containing data that is processing-irrelevant. This is possible because the data dictionary contains the boundaries of the partitions, and therefore, the query optimizer can compare them to the restriction or join condition specified in the SQL statement. Because of limitations, however, this isn't always possible. The next subsections show different examples that point out how and when the query optimizer is able to use partition pruning.

---

■ **Note** Throughout this section, only queries are used in the examples. This doesn't mean that partition pruning works only with queries. Actually, it works in the same way for SQL statements such as `UPDATE` and `DELETE`. I'm using queries here for convenience only.

---

## PARTITION RANGE SINGLE

In the following SQL statement, the `WHERE` clause contains two restrictions: one for each column of the partition key. In this type of situation, the query optimizer recognizes that only a single partition contains relevant data. As a result, the `PARTITION RANGE SINGLE` operation appears in the execution plan. It's essential to understand that its child operation (`TABLE ACCESS FULL`) isn't a full table scan over the whole table. Instead, only a single partition is accessed. This is confirmed by the value of the `Starts` column as well. Which partition is accessed is specified by the `Pstart` and `Pstop` columns:

```
SELECT * FROM t WHERE n1 = 3 AND d1 = to_date('2014-07-19','yyyy-mm-dd')

---------------------------------------------------------------
| Id  | Operation              | Name | Starts | Pstart| Pstop |
---------------------------------------------------------------
|   0 | SELECT STATEMENT       |      |      1 |       |       |
|   1 |  PARTITION RANGE SINGLE|      |      1 |    31 |    31 |
|*  2 |   TABLE ACCESS FULL    | T    |      1 |    31 |    31 |
---------------------------------------------------------------

   2 - filter("D1"=TO_DATE(' 2014-07-19 00:00:00', 'syyyy-mm-dd hh24:mi:ss') AND "N1"=3)
```

Figure 13-6 is a graphical representation of this behavior.

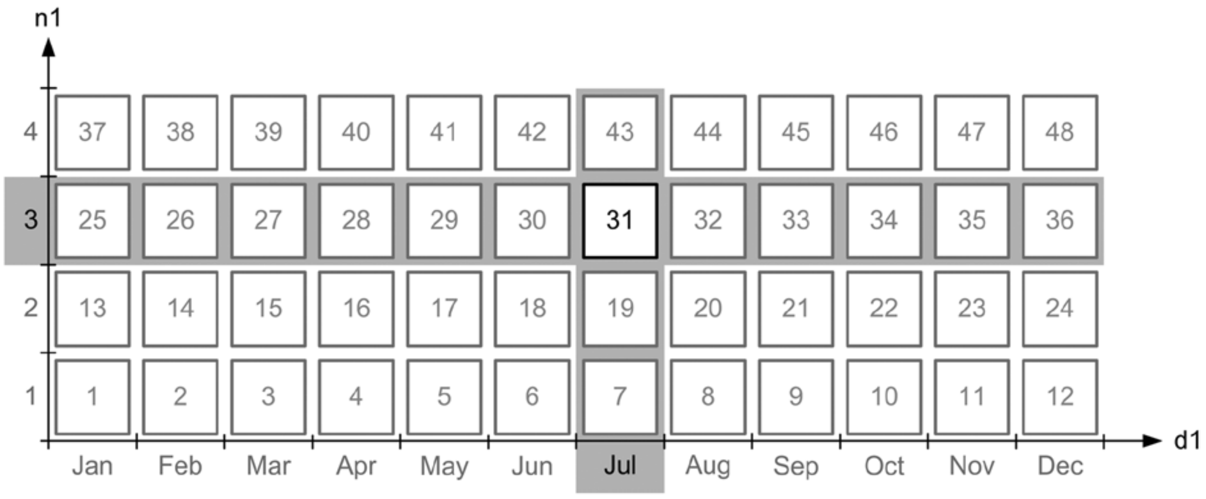


***Figure 13-6.*** *Representation of a* `PARTITION RANGE SINGLE` *operation*

As the output of the following query shows, the partition number in the `Pstart` and `Pstop` columns matches the value in the `partition_position` column in the `user_tab_partitions` view:

```
SQL> SELECT partition_name
  2  FROM user_tab_partitions
  3  WHERE table_name = 'T'
  4  AND partition_position = 31;

PARTITION_NAME
--------------
T_3_JUL_2014
```

Whenever a bind variable is used in a restriction, the query optimizer is no longer able to determine which partitions need to be accessed at parse time. In such cases, partition pruning is performed at runtime. The execution plan doesn't change, but the values of the `Pstart` and `Pstop` columns are set to `KEY`. This indicates that partition pruning occurs, but that at parse time, the query optimizer doesn't know which partition contains relevant data:

```
SELECT * FROM t WHERE n1 = :n1 AND d1 = to_date(:d1,'YYYY-MM-DD')

----------------------------------------------------------------
| Id  | Operation             | Name | Starts | Pstart| Pstop |
----------------------------------------------------------------
|   0 | SELECT STATEMENT      |      |      1 |       |       |
|   1 |  PARTITION RANGE SINGLE|      |      1 |   KEY |   KEY |
|*  2 |   TABLE ACCESS FULL   | T    |      1 |   KEY |   KEY |
----------------------------------------------------------------

   2 - filter(("D1"=TO_DATE(:D1,'YYYY-MM-DD') AND "N1"=:N1))
```

## PARTITION RANGE ITERATOR

The execution plans described in the previous section contain the `PARTITION RANGE SINGLE` operation. This was because the query optimizer recognizes that only a single partition contains processing-relevant data. Obviously, there are situations where several partitions have to be accessed. For example, in the following query, the restriction uses a less-than condition (<) instead of being based on an equality condition (=) Thus, the operation becomes a `PARTITION RANGE ITERATOR`, and the `Pstart` and `Pstop` columns show the range of partitions that are accessed (see Figure 13-7). In addition, the `Starts` column shows that operation 1 is executed only once, but operation 2 is executed once per partition. In other words, seven full partition scans are executed:

```
SELECT * FROM t WHERE n1 = 3 AND d1 < to_date('2014-07-19','YYYY-MM-DD')

-------------------------------------------------------------------
| Id  | Operation                 | Name | Starts | Pstart| Pstop |
-------------------------------------------------------------------
|   0 | SELECT STATEMENT          |      |      1 |       |       |
|   1 |  PARTITION RANGE ITERATOR|      |      1 |    25 |    31 |
|*  2 |   TABLE ACCESS FULL       | T    |      7 |    25 |    31 |
-------------------------------------------------------------------

   2 - filter(("N1"=3 AND "D1"<TO_DATE(' 2014-07-19 00:00:00', 'syyyy-mm-dd hh24:mi:ss')))
```

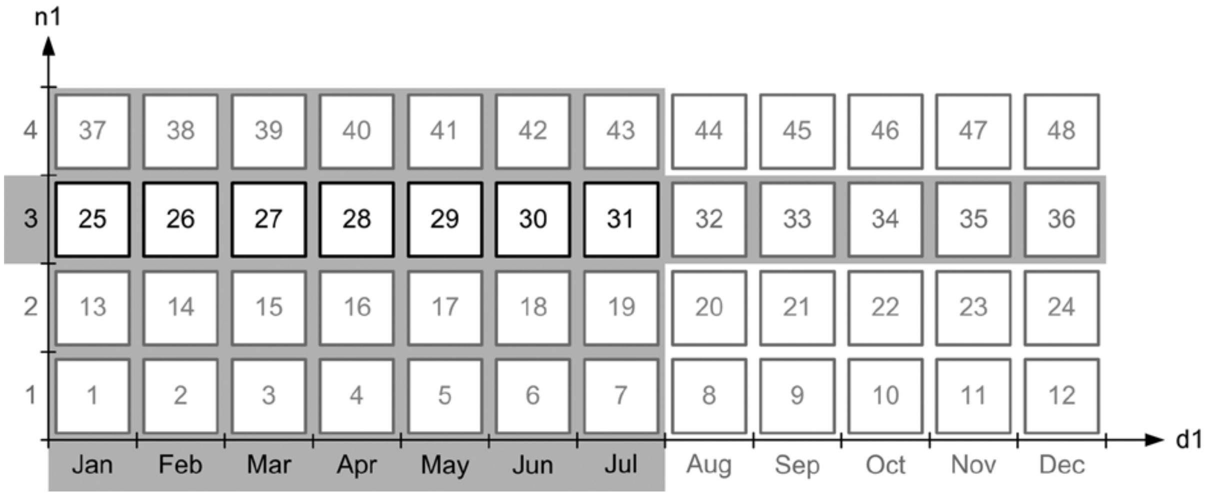


***Figure 13-7.*** *Representation of a `PARTITION RANGE ITERATOR` operation*

The PARTITION RANGE ITERATOR operation is also used when a restriction is based on the leading part of the partition key only. The following query illustrates this, where the restriction is applied to the first column of the partition key. Note that partition 37 is accessed as well. This is because rows that have the value of the n1 column equal to 3 would be stored in that partition if the d1 column had a value later than the 31st of December, 2014:

```
SELECT * FROM t WHERE n1 = 3

-----------------------------------------------------------------
| Id  | Operation                | Name | Starts | Pstart| Pstop |
-----------------------------------------------------------------
|   0 | SELECT STATEMENT         |      |      1 |       |       |
|   1 |  PARTITION RANGE ITERATOR|      |      1 |    25 |    37 |
|*  2 |   TABLE ACCESS FULL      | T    |     13 |    25 |    37 |
-----------------------------------------------------------------

   2 - filter("N1"=3)
```

As the name of this operation implies, it works only with a continuous range of partitions. When a noncontinuous range is used, the operation presented in the next section comes into play.

## PARTITION RANGE INLIST

If a restriction is based on one or several IN conditions that are composed of more than one element, a specific operation, PARTITION RANGE INLIST, appears in the execution plan. With this operation, the Pstart and Pstop columns don't give precise information about which partitions are accessed. Instead, they show the value KEY(I). This indicates that partition pruning occurs separately for every value in the IN condition. In addition, the Starts column shows how many partitions are accessed (in this case, two):

```
SELECT * FROM t WHERE n1 IN (1,3) AND d1 = to_date('2014-07-19','YYYY-MM-DD')

---------------------------------------------------------------
| Id  | Operation              | Name | Starts | Pstart| Pstop |
---------------------------------------------------------------
|   0 | SELECT STATEMENT       |      |      1 |       |       |
|   1 |  PARTITION RANGE INLIST|      |      1 |KEY(I) |KEY(I) |
|*  2 |   TABLE ACCESS FULL    | T    |      2 |KEY(I) |KEY(I) |
---------------------------------------------------------------

   2 - filter(("D1"=TO_DATE(' 2014-07-19 00:00:00', 'syyyy-mm-dd hh24:mi:ss') AND
              INTERNAL_FUNCTION("N1")))
```

In this specific case, based on the WHERE clause, you can infer that only partitions 7 and 31 are accessed. Figure 13-8 illustrates this.

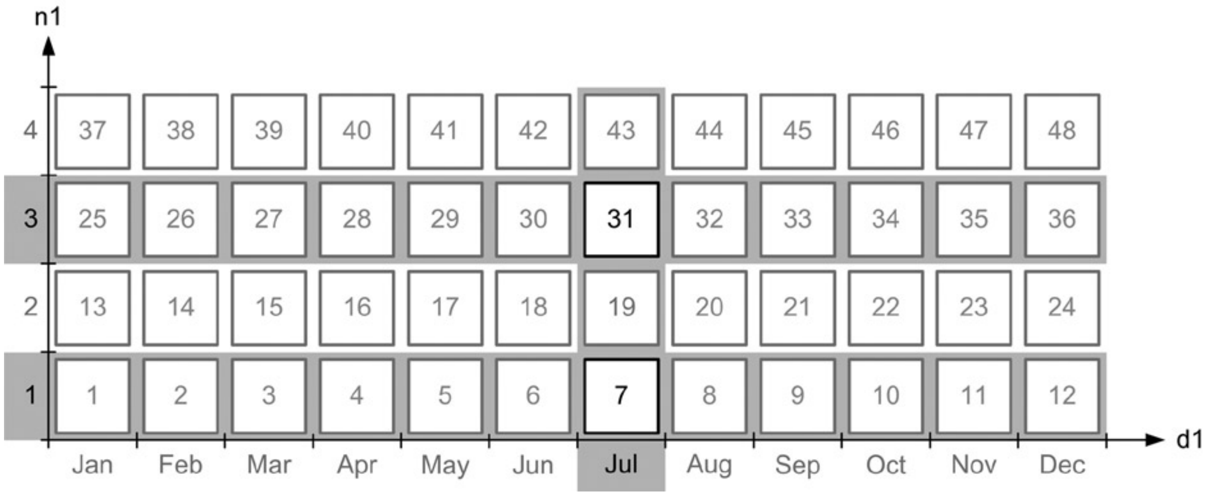


***Figure 13-8.*** *Representation of a `PARTITION RANGE INLIST` operation*

Of course, if the values in the `IN` condition are sufficiently spread, it's possible that most of the partitions are accessed. In cases where the query optimizer recognizes that all partitions are accessed, the operation in the next section applies.

## PARTITION RANGE ALL

If no restriction is applied to the partition key, all partitions must be accessed. In such a case, the execution plan contains the `PARTITION RANGE ALL` operation, and the `Starts`, `Pstart`, and `Pstop` columns clearly show that all partitions are accessed:

```
SELECT * FROM t WHERE n3 BETWEEN 6000 AND 7000

-------------------------------------------------------------
| Id  | Operation           | Name | Starts | Pstart| Pstop |
-------------------------------------------------------------
|   0 | SELECT STATEMENT    |      |      1 |       |       |
|   1 |  PARTITION RANGE ALL|      |      1 |     1 |    48 |
|*  2 |   TABLE ACCESS FULL | T    |     48 |     1 |    48 |
-------------------------------------------------------------

   2 - filter(("N3">=6000 AND "N3"<=7000))
```

This very same execution plan is also used when inequalities are used as restrictions on the partition key. The following query is an example:

```
SELECT * FROM t WHERE n1 != 3 AND d1 != to_date('2014-07-19','YYYY-MM-DD')
```

Another case where the very same execution plan is used is when a restriction on the partition key is based on an expression or function. For example, in the following query, one is added to the n1 column, and the d1 column is modified through the to_char function:

```
SELECT * FROM t WHERE n1 + 1 = 4 AND to_char(d1,'YYYY-MM-DD') = '2014-07-19'
```

This means that to take advantage of partition pruning, not only should you have a restriction based on the partition key, but you should also not apply an expression or function to it. If applying an expression is a must, as of version 11.1 it's possible to choose a virtual column as partition key.

## PARTITION RANGE EMPTY

A particular operation, PARTITION RANGE EMPTY, appears in execution plans when the query optimizer recognizes that no partition is able to store processing-relevant data. For example, the following query is looking for data that has no partition where it could be stored (for the n1 column, the value 5 is out of range). It's also important to note that not only are the Pstart and Pstop columns set to the value INVALID, but in addition, only operation 1 is executed (consuming no resources at all because, basically, it's a no-op operation):

```
SELECT * FROM t WHERE n1 = 5 AND d1 = to_date('2014-07-19','YYYY-MM-DD')

---------------------------------------------------------------
| Id  | Operation             | Name | Starts | Pstart| Pstop |
---------------------------------------------------------------
|   0 | SELECT STATEMENT      |      |      1 |       |       |
|   1 |  PARTITION RANGE EMPTY|      |      1 |INVALID|INVALID|
|*  2 |   TABLE ACCESS FULL   | T    |      0 |INVALID|INVALID|
---------------------------------------------------------------

   2 - filter(("D1"=TO_DATE(' 2014-07-19 00:00:00', 'syyyy-mm-dd hh24:mi:ss') AND "N1"=5))
```

## PARTITION RANGE OR

The type of pruning described in this section, the so-called *OR pruning*, is used for WHERE clauses that contain disjunctive predicates (predicates combined by OR conditions) on the partition key. The following query is an example of such a situation. When this type of pruning is used, the PARTITION RANGE OR operation appears in the execution plan. Also note that the Pstart and Pstop columns are set to the value KEY(OR). In the following example, according to the Starts column, 18 partitions are accessed. There are 18 because although the restriction applied to the n1 column causes partitions 25 to 37 to be accessed, the restriction applied to the d1 column causes partitions 1, 3, 15, 27, and 39 to be accessed (partition 1 is necessary in order to find out whether there are rows with the n1 column containing values less than 1):

```
SELECT * FROM t WHERE n1 = 3 OR d1 = to_date('2014-03-06','YYYY-MM-DD')

------------------------------------------------------------
| Id  | Operation          | Name | Starts | Pstart| Pstop |
------------------------------------------------------------
|   0 | SELECT STATEMENT   |      |      1 |       |       |
|   1 |  PARTITION RANGE OR|      |      1 |KEY(OR)|KEY(OR)|
|*  2 |   TABLE ACCESS FULL| T    |     18 |KEY(OR)|KEY(OR)|
------------------------------------------------------------

   2 - filter(("N1"=3 OR "D1"=TO_DATE(' 2014-03-06 00:00:00', 'syyyy-mm-dd hh24:mi:ss')))
```

## PARTITION RANGE SUBQUERY

In the previous sections, all restrictions used for partition pruning are based on literals or bind variables. However, it's not uncommon for restrictions to actually be join conditions. Whenever a join is based on a partition key, not only is the query optimizer not always able to take advantage of partition pruning, but in some situations it's also not sensible to do so. To choose the execution plan with the lowest cost, the query optimizer has to choose between three strategies.

---

■ **Note** Chapter 14 covers the join methods in detail.

---

The first strategy is to elude partition pruning. The following query (note that the `tx` table is a copy of the `t` table; the only difference is that the `tx` table isn't partitioned) illustrates this, where no partition pruning on the `t` table is performed. In fact, because operation 4 is a `PARTITION RANGE ALL`, operation 5 is processed for all partitions. In this particular example, this execution plan is highly inefficient. This is especially the case given that the selectivity of the query is strong:

```
SELECT * FROM tx, t WHERE tx.d1 = t.d1 AND tx.n1 = t.n1 AND tx.id = 19

----------------------------------------------------------------------
| Id  | Operation                    | Name  | Starts | Pstart| Pstop |
----------------------------------------------------------------------
|   0 | SELECT STATEMENT             |       |      1 |       |       |
|*  1 |  HASH JOIN                   |       |      1 |       |       |
|   2 |   TABLE ACCESS BY INDEX ROWID| TX    |      1 |       |       |
|*  3 |    INDEX UNIQUE SCAN         | TX_PK |      1 |       |       |
|   4 |   PARTITION RANGE ALL        |       |      1 |     1 |    48 |
|   5 |    TABLE ACCESS FULL         | T     |     48 |     1 |    48 |
----------------------------------------------------------------------

   1 - access("TX"."D1"="T"."D1" AND "TX"."N1"="T"."N1")
   3 - access("TX"."ID"=19)
```

This strategy is always available. Nevertheless, it could lead to poor performance if the selectivity of the join condition isn't close to 1—or, in other words, in situations where partition pruning should be used.

The second strategy is to execute the join with the operation `NESTED LOOPS` and access the table, which the partition pruning should occur on, as the second child. In fact, as discussed in Chapter 10, the `NESTED LOOPS` operation is a related-combine operation, and therefore, its first child controls the execution of the second child. The following example shows such a situation. Note that the `PARTITION RANGE ITERATOR` operation and the values of the `Pstart` and `Pstop` columns confirm that partition pruning takes place. According to the `Starts` column, a single partition is accessed. In this specific case, the following execution plan is, therefore, far more efficient than the one used by the first strategy:

```
SELECT * FROM tx, t WHERE tx.d1 = t.d1 AND tx.n1 = t.n1 AND tx.id = 19

----------------------------------------------------------------------
| Id  | Operation                      | Name  | Starts | Pstart| Pstop |
----------------------------------------------------------------------
|   0 | SELECT STATEMENT               |       |      1 |       |       |
|   1 |  NESTED LOOPS                  |       |      1 |       |       |
|   2 |   TABLE ACCESS BY INDEX ROWID| TX    |      1 |       |       |
|*  3 |    INDEX UNIQUE SCAN           | TX_PK |      1 |       |       |
|   4 |   PARTITION RANGE ITERATOR     |       |      1 |   KEY |   KEY |
|*  5 |    TABLE ACCESS FULL           | T     |      1 |   KEY |   KEY |
----------------------------------------------------------------------

   3 - access("TX"."ID"=19)
   5 - filter(("TX"."D1"="T"."D1" AND "TX"."N1"="T"."N1"))
```

This strategy performs well only if the number of rows returned by the first child of the NESTED LOOP operation (in this case operation 2) is low. Otherwise, it's even possible that the same partition is accessed several times by the second child (in this case operation 4).

The third strategy is to execute the join with the HASH JOIN or MERGE JOIN operation. No regular partition pruning based on the join condition is possible, however, with these join methods. In fact, as discussed in Chapter 10, they're unrelated-combine operations, and consequently, the two children are executed separately. In such cases, the query optimizer can take advantage of another type of partition pruning, the *subquery pruning*. The idea here is to find out with a recursive query which partitions of the second child should be accessed. For that purpose, the SQL engine executes a recursive query (on the table accessed by the first child) to retrieve the columns used in the join condition that maps to the partition keys of the second child. Then, by consulting the partition definitions of the second child stored in the data dictionary, the partitions to be accessed by the second child are identified, and so it's possible to scan only them. The following query shows an example of this. Note that the operation PARTITION RANGE SUBQUERY and the value of the Pstart and Pstop columns (KEY(SQ)) confirm that partition pruning takes place. According to the Starts column, a single partition is accessed:

```
SELECT * FROM tx, t WHERE tx.d1 = t.d1 AND tx.n1 = t.n1 AND tx.id = 19

----------------------------------------------------------------------
| Id  | Operation                      | Name  | Starts | Pstart| Pstop |
----------------------------------------------------------------------
|   0 | SELECT STATEMENT               |       |      1 |       |       |
|*  1 |  HASH JOIN                     |       |      1 |       |       |
|   2 |   TABLE ACCESS BY INDEX ROWID| TX    |      1 |       |       |
|*  3 |    INDEX UNIQUE SCAN           | TX_PK |      1 |       |       |
|   4 |   PARTITION RANGE SUBQUERY     |       |      1 |KEY(SQ)|KEY(SQ)|
|   5 |    TABLE ACCESS FULL           | T     |      1 |KEY(SQ)|KEY(SQ)|
----------------------------------------------------------------------

   1 - access("TX"."D1"="T"."D1" AND "TX"."N1"="T"."N1")
   3 - access("TX"."ID"=19)
```

What actually happens here is that, to find out which partitions have to be accessed, the SQL engine executes the following query recursively. This recursive query retrieves the number of the partitions that contain relevant data with the `tbl$or$idx$part$num` function. With this information, operation 5 can take advantage of partition pruning. For example, in this case, only partition 37 needs to be scanned:

```
SQL> SELECT DISTINCT TBL$OR$IDX$PART$NUM("T", 0, 1, 0, "N1", "D1") AS PART_NUM
  2  FROM (SELECT "TX"."N1" "N1", "TX"."D1" "D1"
  3        FROM "TX" "TX"
  4        WHERE "TX"."ID"=19)
  5  ORDER BY 1;

  PART_NUM
----------
        37
```

Clearly, it makes sense to use this third technique only when the overhead caused by the execution of the recursive query is less than the gain because of partition pruning. For the query used in this example, the efficiency of the execution plans resulting from the second and third strategies are very similar. Nevertheless, if the selectivity is weaker, the execution plan resulting from the third strategy would be more efficient.

## PARTITION RANGE JOIN-FILTER

Subquery pruning is a useful optimization technique. However, as discussed in the previous section, part of the SQL statement is executed twice. To avoid this double execution, as of version 11.1 the database engine provides another type of partition pruning: the *join-filter pruning* (also known as *bloom-filter pruning*). To understand how it works, let's look at the execution plan generated by the same query as the one used to describe subquery pruning. Note that several new things appear: the `PART JOIN FILTER CREATE` operation; the `PARTITION RANGE JOIN-FILTER` operation; and, in the `Name`, `Pstart`, and `Pstop` columns, the string `:BF0000`:

```
SELECT * FROM tx, t WHERE tx.d1 = t.d1 AND tx.n1 = t.n1 AND tx.id = 19

-----------------------------------------------------------------------------------
| Id  | Operation                      | Name    | Starts | E-Rows | Pstart| Pstop |
-----------------------------------------------------------------------------------
|   0 | SELECT STATEMENT               |         |      1 |        |       |       |
|*  1 |  HASH JOIN                     |         |      1 |      7 |       |       |
|   2 |   PART JOIN FILTER CREATE      | :BF0000 |      1 |      1 |       |       |
|   3 |    TABLE ACCESS BY INDEX ROWID| TX      |      1 |      1 |       |       |
|*  4 |     INDEX UNIQUE SCAN          | TX_PK   |      1 |      1 |       |       |
|   5 |   PARTITION RANGE JOIN-FILTER  |         |      1 |  10000 |:BF0000|:BF0000|
|   6 |    TABLE ACCESS FULL           | T       |      1 |  10000 |:BF0000|:BF0000|
-----------------------------------------------------------------------------------

   1 - access("TX"."N1"="T"."N1" AND "TX"."D1"="T"."D1")
   4 - access("TX"."ID"=19)
```

The execution plan is executed as follows:

- Operations 3 and 4 access the `tx` table through the `tx_pk` index.
- Based on the data returned by operation 3, operation 2 creates a memory structure (a bloom filter) based on values from the columns used in the join condition (`tx.d1` and `tx.n1`).
- Based on the memory structure created by operation 2, operation 5 is able to take advantage of partition pruning and, therefore, is able to access only the partitions that contain relevant data. In this case, a single partition is accessed (see the `Starts` column).

## PARTITION RANGE MULTI-COLUMN

If the partition key is composed of several columns, it's important to observe what happens when a restriction isn't defined for every column. The main question is, does the query optimizer take advantage of partition pruning? The answer is, thanks to *multicolumn pruning*, yes. The goal of multicolumn pruning is quite simple: independently of which columns a restriction is defined on, partition pruning always occurs.

Let's see this feature in action on the same test table we've used before. Because the partition key of the test table is composed of two columns, there are two cases to consider: the restriction is applied to either the first column or the second column. The following query is an example of the former:

```
SELECT * FROM t WHERE n1 = 3

------------------------------------------------------------------
| Id  | Operation                | Name | Starts | Pstart| Pstop |
------------------------------------------------------------------
|   0 | SELECT STATEMENT         |      |      1 |       |       |
|   1 |  PARTITION RANGE ITERATOR|      |      1 |    25 |    37 |
|*  2 |   TABLE ACCESS FULL      | T    |     13 |    25 |    37 |
------------------------------------------------------------------

   2 - filter("N1"=3)
```

The following query is an example of the latter. Note that the `PARTITION RANGE MULTI-COLUMN` operation and the value of the `Pstart` and `Pstop` columns confirm that partition pruning takes place; however, no information is provided about which partitions are accessed:

```
SELECT * FROM t WHERE d1 = to_date('2014-07-19','YYYY-MM-DD')

----------------------------------------------------------------------
| Id  | Operation                    | Name | Starts | Pstart| Pstop |
----------------------------------------------------------------------
|   0 | SELECT STATEMENT             |      |      1 |       |       |
|   1 |  PARTITION RANGE MULTI-COLUMN|      |      1 |KEY(MC)|KEY(MC)|
|*  2 |   TABLE ACCESS FULL          | T    |      8 |KEY(MC)|KEY(MC)|
----------------------------------------------------------------------

   2 - filter("D1"=TO_DATE(' 2014-07-19 00:00:00', 'syyyy-mm-dd hh24:mi:ss'))
```

## PARTITION RANGE AND

In some situations the query optimizer can take advantage of several of the pruning techniques described in the previous sections. As an example, have a look at the following query:

```
SELECT * FROM tx, t WHERE tx.d1 = t.d1 AND tx.n1 = t.n1 AND t.n1 = 3 AND tx.n2 = 42
```

With such a SQL statement, the query optimizer should consider at least two pruning techniques:

- Partition pruning based on the `t.n1 = 3` restriction:

```
-----------------------------------------------------------------------------
| Id  | Operation                  | Name | Starts | Pstart| Pstop | Buffers |
-----------------------------------------------------------------------------
|   0 | SELECT STATEMENT           |      |      1 |       |       |     889 |
|*  1 |  HASH JOIN                 |      |      1 |       |       |     889 |
|*  2 |   TABLE ACCESS FULL        | TX   |      1 |       |       |     403 |
|   3 |   PARTITION RANGE ITERATOR|      |      1 |    25 |    37 |     486 |
|*  4 |    TABLE ACCESS FULL       | T    |     13 |    25 |    37 |     486 |
-----------------------------------------------------------------------------

   1 - access("TX"."N1"="T"."N1" AND "TX"."D1"="T"."D1")
   2 - filter(("TX"."N2"=42 AND "TX"."N1"=3))
   4 - filter("T"."N1"=3)
```

- Partition pruning based on the `tx.d1 = t.d1 AND tx.n1 = t.n1` join condition (to take advantage of the `tx.n2 = 42` restriction):

```
-----------------------------------------------------------------------------------
| Id  | Operation                    | Name    | Starts | Pstart| Pstop | Buffers |
-----------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |         |      1 |       |       |     963 |
|*  1 |  HASH JOIN                   |         |      1 |       |       |     963 |
|   2 |   PART JOIN FILTER CREATE    | :BF0000 |      1 |       |       |     403 |
|*  3 |    TABLE ACCESS FULL         | TX      |      1 |       |       |     403 |
|   4 |   PARTITION RANGE JOIN-FILTER|         |      1 |:BF0000|:BF0000|     560 |
|*  5 |    TABLE ACCESS FULL         | T       |     15 |:BF0000|:BF0000|     560 |
-----------------------------------------------------------------------------------

   1 - access("TX"."N1"="T"."N1" AND "TX"."D1"="T"."D1")
   3 - filter("TX"."N2"=42)
   5 - filter("T"."N1"=3)
```

From version 11.2 onward, the query optimizer can even take advantage of several pruning techniques at once. This guarantees that the least number of partitions is accessed (compare the `Starts` column of the three cases). The following example shows that when this type of pruning, called *AND pruning*, is used, the `PARTITION RANGE AND` operation appears in the execution plan. Also note that the `Pstart` and `Pstop` columns are set to the value `KEY(AP)`. In this specific case, the partition pruning is based on both the restriction (`t.n1 = 3`) and the join condition (`tx.d1 = t.d1 AND tx.n1 = t.n1`). For the join condition, notice that a bloom filter is created:

```
SELECT * FROM tx, t WHERE tx.d1 = t.d1 AND tx.n1 = t.n1 AND t.n1 = 3
```

```
------------------------------------------------------------------------------
| Id  | Operation                  | Name    | Starts | Pstart| Pstop | Buffers |
------------------------------------------------------------------------------
|   0 | SELECT STATEMENT           |         |      1 |       |       |     630 |
|*  1 |  HASH JOIN                 |         |      1 |       |       |     630 |
|   2 |   PART JOIN FILTER CREATE| :BF0000 |      1 |       |       |     403 |
|*  3 |    TABLE ACCESS FULL       | TX      |      1 |       |       |     403 |
|   4 |   PARTITION RANGE AND      |         |      1 |KEY(AP)|KEY(AP)|     227 |
|*  5 |    TABLE ACCESS FULL       | T       |      6 |KEY(AP)|KEY(AP)|     227 |
------------------------------------------------------------------------------

   1 - access("TX"."N1"="T"."N1" AND "TX"."D1"="T"."D1")
   3 - filter(("TX"."N2"=42 AND "TX"."N1"=3))
   5 - filter("T"."N1"=3)
```

## Hash and List Partitioning

The previous section covered range partitioning only. Anyway, with hash and list partitioning, most of the techniques described for range partitioning are available as well.

The following are the operations available with hash partitioning. The `pruning_hash.sql` script provides examples of execution plans that contain these operations:

- `PARTITION HASH SINGLE`
- `PARTITION HASH ITERATOR`
- `PARTITION HASH INLIST`
- `PARTITION HASH ALL`
- `PARTITION HASH SUBQUERY`
- `PARTITION HASH JOIN-FILTER`
- `PARTITION HASH AND`

The following are the operations available with list partitioning. The `pruning_list.sql` script provides examples of execution plans containing these operations:

- `PARTITION LIST SINGLE`
- `PARTITION LIST ITERATOR`
- `PARTITION LIST INLIST`
- `PARTITION LIST ALL`
- `PARTITION LIST EMPTY`
- `PARTITION LIST OR`
- `PARTITION LIST SUBQUERY`
- `PARTITION LIST JOIN-FILTER`
- `PARTITION LIST AND`

## Composite Partitioning

There's little that can be said about composite partitioning. Basically, everything that applies at the partition level applies at the subpartition level as well. Nevertheless, it makes sense to illustrate at least one example. The following test table is partitioned by range (based on the `d1` column) and subpartitioned by list (based on the `n1` column). The following SQL statement, which is an excerpt of the `pruning_composite.sql` script, was used to create the table. Note that also in this case, there are 48 partitions per year:

```
CREATE TABLE t (
  id NUMBER,
  d1 DATE,
  n1 NUMBER,
  n2 NUMBER,
  n3 NUMBER,
  pad VARCHAR2(4000),
  CONSTRAINT t_pk PRIMARY KEY (id)
)
PARTITION BY RANGE (d1)
SUBPARTITION BY LIST (n1)
SUBPARTITION TEMPLATE (
  SUBPARTITION sp_1 VALUES (1),
  SUBPARTITION sp_2 VALUES (2),
  SUBPARTITION sp_3 VALUES (3),
  SUBPARTITION sp_4 VALUES (4)
)(
  PARTITION t_jan_2014 VALUES LESS THAN (to_date('2014-02-01','YYYY-MM-DD')),
  PARTITION t_feb_2014 VALUES LESS THAN (to_date('2014-03-01','YYYY-MM-DD')),
  PARTITION t_mar_2014 VALUES LESS THAN (to_date('2014-04-01','YYYY-MM-DD')),
  PARTITION t_apr_2014 VALUES LESS THAN (to_date('2014-05-01','YYYY-MM-DD')),
  PARTITION t_may_2014 VALUES LESS THAN (to_date('2014-06-01','YYYY-MM-DD')),
  PARTITION t_jun_2014 VALUES LESS THAN (to_date('2014-07-01','YYYY-MM-DD')),
  PARTITION t_jul_2014 VALUES LESS THAN (to_date('2014-08-01','YYYY-MM-DD')),
  PARTITION t_aug_2014 VALUES LESS THAN (to_date('2014-09-01','YYYY-MM-DD')),
  PARTITION t_sep_2014 VALUES LESS THAN (to_date('2014-10-01','YYYY-MM-DD')),
  PARTITION t_oct_2014 VALUES LESS THAN (to_date('2014-11-01','YYYY-MM-DD')),
  PARTITION t_nov_2014 VALUES LESS THAN (to_date('2014-12-01','YYYY-MM-DD')),
  PARTITION t_dec_2014 VALUES LESS THAN (to_date('2015-01-01','YYYY-MM-DD'))
)
```

Figure 13-9 is a graphical representation of this test table. If you compare it with the previous one (see Figure 13-5), the only difference is that no single value identifies the position of subpartitions throughout the table. In fact, the position of a subpartition is based on its “parent” partition.

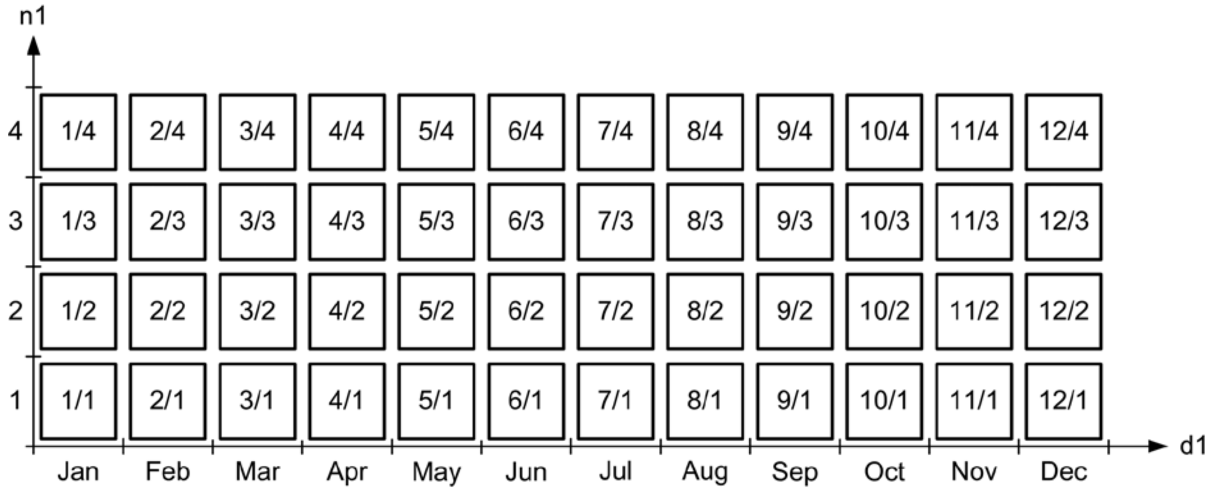


***Figure 13-9.*** *The test table is composed of 48 partitions per year*

Of course, the mapping between name and position is available in the data dictionary also in this case. The following query shows how to get it from the `user_tab_partitions` and `user_tab_subpartitions` views:

```
SQL> SELECT subpartition_name, partition_position, subpartition_position
  2  FROM user_tab_partitions p, user_tab_subpartitions s
  3  WHERE p.table_name = 'T'
  4  AND s.table_name = p.table_name
  5  AND s.partition_name = p.partition_name
  6  ORDER BY p.partition_position, s.subpartition_position;

SUBPARTITION_NAME PARTITION_POSITION SUBPARTITION_POSITION
----------------- ------------------ ---------------------
T_JAN_2014_SP_1                    1                     1
T_JAN_2014_SP_2                    1                     2
T_JAN_2014_SP_3                    1                     3
T_JAN_2014_SP_4                    1                     4
T_FEB_2014_SP_1                    2                     1
...
T_NOV_2014_SP_4                   11                     4
T_DEC_2014_SP_1                   12                     1
T_DEC_2014_SP_2                   12                     2
T_DEC_2014_SP_3                   12                     3
T_DEC_2014_SP_4                   12                     4
```

The following query is an example of a restriction at both the partition and subpartition levels. The operations are those described in the previous sections. Operation 1 applies at the partition level, and operation 2 applies at the subpartition level. At the partition level, partitions 1 to 7 are accessed. For each of them, only subpartition 3 is accessed. Figure 13-10 shows this behavior. Notice how the values in the Pstart and Pstop columns match the result of the previous query when executed against the data dictionary:

```
SELECT * FROM t WHERE n1 = 3 AND d1 < to_date('2014-07-19','YYYY-MM-DD')

------------------------------------------------------------------
| Id  | Operation                  | Name | Starts | Pstart| Pstop |
------------------------------------------------------------------
|   0 | SELECT STATEMENT           |      |      1 |       |       |
|   1 |  PARTITION RANGE ITERATOR|      |      1 |     1 |     7 |
|   2 |   PARTITION LIST SINGLE   |      |      7 |     3 |     3 |
|*  3 |    TABLE ACCESS FULL       | T    |      7 |   KEY |   KEY |
------------------------------------------------------------------

   3 - filter("D1"<TO_DATE(' 2014-07-19 00:00:00', 'syyyy-mm-dd hh24:mi:ss'))
```

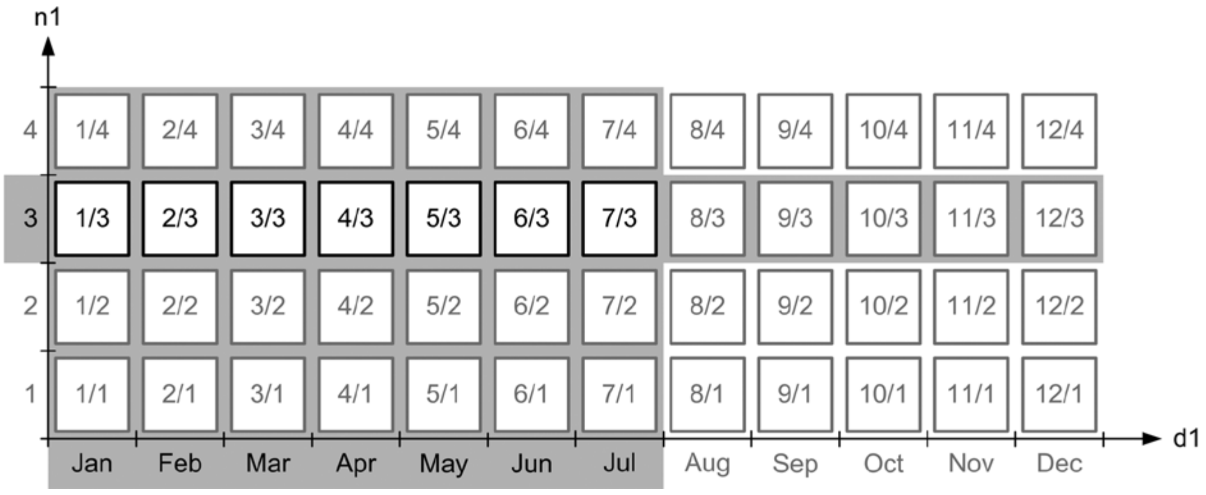


***Figure 13-10.*** *Representation of composite partition pruning*

## Design Considerations

As described in the previous sections, the query optimizer can take advantage of partition pruning in a wide range of situations. Table 13-1 summarizes when partition pruning occurs for each type of partitioning method and for the most common SQL conditions.

***Table 13-1.*** *Conditions That Lead to Partition Pruning**

| Condition | Range | List | Hash |
|---|---|---|---|
| Equality (=) | ✓ | ✓ | ✓ |
| `IN` | ✓ | ✓ | ✓ |
| `BETWEEN`, >, >=, <, or <= | ✓ | ✓ | |
| `IS NULL` | ✓ | ✓ | |

**Inequality (!= or <>), NOT IN, IS NOT NULL conditions, and restrictions based on expressions and functions don't lead to partition pruning.*

Choosing the partition key and the partitioning method is probably the most important decision you have to make while working on the design of a partitioned table. The objective is to take advantage of partition pruning to efficiently process as many SQL statements as possible. In addition, one partition should ideally contain only the data you expect to be processed by a single SQL statement. For example, if there are frequent SQL statements processing data by the day, you should partition by the day. Or, if there are frequent SQL statements that process data by country, you should partition the data by country. If you do this incorrectly, you'll never be able to take advantage of partition pruning. The following four characteristics of your application have to be considered very carefully, because they're the ones that impact the partitioning strategy the most:

1. What columns the restrictions are expected to be applied on and with what frequency
2. What kind of data will be stored in those columns
3. What the SQL conditions used in those restrictions will be
4. Whether data must be regularly compressed or purged and what criterion to base the processing on

The first and the fourth are essential for choosing the partition key. The second and the third are for choosing the partitioning method. Let's discuss them in detail.

Knowing which columns the restrictions are applied on is essential because no partition pruning is possible if no restriction is applied to the partition key. In other words, based on this criterion, you restrict the choice between a limited number of columns. In practice, it's quite common for several, possibly very different, restrictions to be applied by different SQL statements. Therefore, you must know the frequency of utilization of the different SQL statements as well. In this way, you can decide which restrictions are the most important to optimize. In any case, you should consider only those restrictions that have weak selectivity. In fact, restrictions with strong selectivity can be optimized with other access structures (for example, indexes).

Once the columns to be potentially used in the partition key are known, it's time to take a look at the data they store. The idea is to find out which partitioning method would be applicable to them. For that, it's essential to recognize two things. First, only range and list partitioning allow the grouping together of "related" data (for example, all of July's sales or all European countries) or an exact mapping of specific values with specific partitions. Second, each partitioning method is suitable only for a very specific kind of data:

- *Range* is suitable for values that are sequential by nature. Typical examples are timestamps and numbers generated by a sequence.
- *List* is suitable when the number of distinct values is both well known and limited in number. Typical examples are all kinds of status information (for example, enabled, disabled, processed) and attributes that describe people (for example, sex, marital status) or things (for example, country and postal code, currency, category, format).
- *Hash* is suitable for all kinds of data for which the number of distinct values is much higher than the number of partitions (for example, customer number).

The partitioning method has to be both suitable for the kind of data and suitable for the restrictions applied to the columns making up the partition key. Here, the restrictions related to hash partitioning described in Table 13-1 should be taken into consideration. In addition, note that even though it's technically possible to use range conditions on list-partitioned tables, it's not uncommon for such conditions to *not* lead to partition pruning. In fact, data in a list-partitioned table is by nature not sequential.

If regular compression or purging activities take place, you should also take into consideration whether they can be implemented by taking advantage of partitioning. For example, dropping or truncating a partition is much faster than deleting the data it contains. Such a strategy is usually possible only when range or list partitioning is used.

Once the partition key has been chosen, it's necessary to decide whether the partition key can be modified. Such a modification typically means moving the row into another partition (an operation very similar to deleting the old row and inserting it again in another partition) and, consequently, changing its rowid. Usually, a row never changes its rowid. So, if it's a possibility, not only must row movement be enabled at the table level to allow the database engine to make such a critical modification, but the application also needs to use rowids with special care.

One final—but in my honest opinion very important—remark, is to prevent the most common mistake I have experienced in projects that implement partitioning. The mistake is designing and implementing the database and the application without implementing partitioning and then, afterward, partitioning it. More often than not, such an approach is doomed to fail. I strongly recommend planning the use of partitioning from the beginning of a project. If you think you'll be able to easily introduce it later, if necessary, you may be in for a big surprise.

## Full Index Scans

The database engine can use indexes not only to extract lists of rowids and use them as pointers to read the corresponding rows from the table, but it can also directly read the column values that are part of the index keys, thus avoiding following the rowid and accessing the table altogether. Thanks to this very important optimization, when an index contains all the data needed to process a query, a full table scan or a full partition scan might be replaced by a *full index scan*. And because an index segment is usually much smaller than a table segment, this is useful for reducing the number of logical reads.

Full index scans are useful in three main situations. The first is when an index stores all the columns used by a query. For example, because the n1 column is indexed, the following query can take advantage of a full index scan. The following execution plan confirms that no table access is performed (note that all examples in this section are based on the index_full_scan.sql script):

```
SELECT /*+ index(t t_n1_i) */ n1 FROM t WHERE n1 IS NOT NULL

-----------------------------------
| Id  | Operation        | Name   |
-----------------------------------
|   0 | SELECT STATEMENT |        |
|*  1 |  INDEX FULL SCAN | T_N1_I |
-----------------------------------

   1 - filter("N1" IS NOT NULL)
```

It's essential to understand that this execution plan is possible because the condition available in the WHERE clause (n1 IS NOT NULL) makes sure that the index stores all the data required for the processing (a NOT NULL constraint on the n1 column would have the same effect because it makes sure that no NULL values are inserted). Otherwise, because single-column B-tree indexes don't store NULL values, a full table scan must be executed.

The INDEX FULL SCAN operation, which can be forced by the index hint as shown in the previous example, scans an index according to its structure. The advantage of this is that the retrieved data is sorted according to the index key. The disadvantage is that if the index blocks aren't already in the buffer cache, they're read from the data files with single-block reads (following one of the pointers in the leaf block to the next/previous leaf block). Because a full index scan may read a lot of data, this is usually inefficient. To improve the performance in such cases, you can use *index fast full scans*. The following execution plan shows an example:

```
SELECT /*+ index_ffs(t t_n1_i) */ n1 FROM t WHERE n1 IS NOT NULL

---------------------------------------
| Id  | Operation             | Name   |
---------------------------------------
|   0 | SELECT STATEMENT      |        |
|*  1 |  INDEX FAST FULL SCAN| T_N1_I |
---------------------------------------

   1 - filter("N1" IS NOT NULL)
```

The peculiarity of the INDEX FAST FULL SCAN operation, which can be forced by the index_ffs hint, is that index blocks are read from the data files with multiblock reads, as full table scans do for tables. During such a scan, the root and the branch blocks can simply be discarded because all data is stored in the leaf blocks (usually root and branch blocks are only a tiny fraction of the index segment, so even if they're read uselessly, the overhead is usually negligible). As a consequence, the index structure isn't considered for the access, and therefore, the retrieved data isn't sorted according to the index key.

The second case is similar to the previous one. The only difference is that the data has to be delivered in the same order as it's stored in the index. For example, as shown in the following query, this is the case because an ORDER BY clause is specified. Because the order matters, only the INDEX FULL SCAN operation can be used to avoid the sort operation:

```
SELECT /*+ index(t t_n1_i) */ n1 FROM t WHERE n1 IS NOT NULL ORDER BY n1

-----------------------------------
| Id  | Operation        | Name   |
-----------------------------------
|   0 | SELECT STATEMENT |        |
|*  1 |  INDEX FULL SCAN | T_N1_I |
-----------------------------------

   1 - filter("N1" IS NOT NULL)
```

Because the disk I/O operations performed by the INDEX FULL SCAN operation are less efficient than those performed by the INDEX FAST FULL SCAN operation, the former is used only when the order matters.

---

■ **Caution** The nls_sort parameter affects ORDER BY operations. If it's set to a value different than binary, an index full scan can be used for optimizing an ORDER BY only when either the data type of the indexed columns isn't impacted by the NLS settings (for example, for NUMBER and DATE) or a linguistic index is used (the section "Linguistic Indexes" later in this chapter provides information about this type of index).

---

Per default, an index scan is performed in ascending order. Consequently, the index hint instructs the query optimizer to behave in this way also. To explicitly specify the scan order, you can use the index_asc and index_desc hints. The following query shows how. The scan in descending order is shown in the execution plan:

```
SELECT /*+ index_desc(t t_n1_i) */ n1 FROM t WHERE n1 IS NOT NULL ORDER BY n1 DESC

---------------------------------------------
| Id  | Operation                  | Name   |
---------------------------------------------
|   0 | SELECT STATEMENT           |        |
|*  1 |  INDEX FULL SCAN DESCENDING| T_N1_I |
---------------------------------------------

   1 - filter("N1" IS NOT NULL)
```

The third case is related to the count function. If a query contains it, the query optimizer tries to take advantage of an index in order to avoid a full table scan. The following query is an example. Notice that the SORT AGGREGATE operation is used to execute the count function:

```
SELECT /*+ index_ffs(t t_n1_i) */ count(n1) FROM t

----------------------------------------
| Id  | Operation             | Name   |
----------------------------------------
|   0 | SELECT STATEMENT      |        |
|   1 |  SORT AGGREGATE       |        |
|   2 |   INDEX FAST FULL SCAN| T_N1_I |
----------------------------------------
```

When a count against a *nullable* column is processed, the query optimizer can pick out any index containing that column (this is because NULL values aren't counted). When either a count(*) or a count against a *not-nullable* column is processed, the query optimizer is able to pick out either any B-tree index that contains at least a not-nullable column (this is necessary because only in this case the number of index entries is guaranteed to be the same as the number of rows), or any bitmap index. Therefore, it picks out the smaller index that can be considered.

Even though the examples in this section are based on B-tree indexes, most techniques apply to bitmap indexes as well. There are only two differences. First, bitmap indexes can't be scanned in descending order (this is a limitation due to the implementation). Second, bitmap indexes always store NULL values. Because of this, they can be used in more situations than B-tree indexes. The following queries show examples that are analogous to the previous ones:

```
SELECT /*+ index(t t_n2_i) */ n2 FROM t WHERE n2 IS NOT NULL

----------------------------------------------
| Id  | Operation                   | Name   |
----------------------------------------------
|   0 | SELECT STATEMENT            |        |
|   1 |  BITMAP CONVERSION TO ROWIDS|        |
|*  2 |   BITMAP INDEX FULL SCAN    | T_N2_I |
----------------------------------------------

   2 - filter("N2" IS NOT NULL)
```

```
SELECT /*+ index_ffs(t t_n2_i) */ n2 FROM t WHERE n2 IS NOT NULL

-----------------------------------------------
| Id  | Operation                      | Name   |
-----------------------------------------------
|   0 | SELECT STATEMENT               |        |
|   1 |  BITMAP CONVERSION TO ROWIDS   |        |
|*  2 |   BITMAP INDEX FAST FULL SCAN| T_N2_I |
-----------------------------------------------

   2 - filter("N2" IS NOT NULL)

SELECT /*+ index(t t_n2_i) */ n2 FROM t WHERE n2 IS NOT NULL ORDER BY n2

----------------------------------------------
| Id  | Operation                     | Name   |
----------------------------------------------
|   0 | SELECT STATEMENT              |        |
|   1 |  BITMAP CONVERSION TO ROWIDS|        |
|*  2 |   BITMAP INDEX FULL SCAN      | T_N2_I |
----------------------------------------------

   2 - filter("N2" IS NOT NULL)

SELECT /*+ index_ffs(t t_n2_i) */ count(n2) FROM t

------------------------------------------------
| Id  | Operation                       | Name   |
------------------------------------------------
|   0 | SELECT STATEMENT                |        |
|   1 |  SORT AGGREGATE                 |        |
|   2 |   BITMAP CONVERSION TO ROWIDS   |        |
|   3 |    BITMAP INDEX FAST FULL SCAN| T_N2_I |
------------------------------------------------
```

## SQL Statements with Strong Selectivity

To efficiently process SQL statements with strong selectivity, the data should be accessed through a rowid, an index, or a single-table hash cluster.[1] These three possibilities are described in the next sections.

### Rowid Access

The most efficient way to access a row is to directly specify its rowid in the WHERE clause. However, to take advantage of that access path, you have to get the rowid first, store it, and then reuse it for further accesses. In other words, it's a method that can be considered only if a row is accessed at least two times. In practice, this is something that happens quite frequently when SQL statements have strong selectivity. For example, applications used to manually maintain data (in other words, not batch jobs) commonly access the same rows at least two times—at least once to show the current data and at least a second time to store the modifications. In such cases, it makes sense to carefully take advantage of the efficiency of rowid accesses.

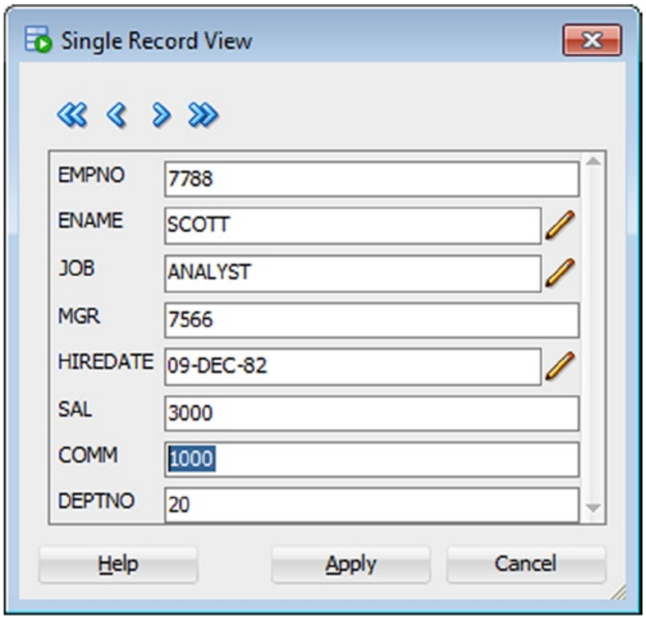
[1]Actually, there are also multitable hash clusters and index clusters. They're not described here because they're rarely used in practice.

A good example demonstrating such an implementation is offered by one of Oracle's tools: SQL Developer. For instance, when SQL Developer displays data, it gets both the data itself as well as the rowids. For the `emp` table of the `scott` schema, the tool executes the following query. Notice that the first column in the `SELECT` clause is the rowid:

```
SELECT ROWID,"EMPNO","ENAME","JOB","MGR","HIREDATE","SAL","COMM","DEPTNO" FROM "SCOTT"."EMP"
```

Later, the rowid can be used to access each specific row directly. For example, if you open the Single Record View dialog box (see Figure 13-11), edit the `comm` column, and commit the modification, the following SQL statement is executed. As you can see, the tool uses the rowid to reference the modified row instead of the primary key (thus saving the overhead of reading several blocks from the primary key index):

```
UPDATE "SCOTT"."EMP" SET COMM=:sqldevvalue WHERE ROWID = :sqldevgridrowid
```



| | |
|---|---|
| EMPNO | 7788 |
| ENAME | SCOTT |
| JOB | ANALYST |
| MGR | 7566 |
| HIREDATE | 09-DEC-82 |
| SAL | 3000 |
| COMM | 1000 |
| DEPTNO | 20 |

***Figure 13-11.*** *The SQL Developer's dialog box is used to display, browse, and edit data*

Using the rowid is very good from a performance point of view because the row can be accessed directly, without the help of a secondary access structure such as an index. The following is an execution plan that is related to such a SQL statement:

```
--------------------------------------------
| Id  | Operation                   | Name |
--------------------------------------------
|   0 | UPDATE STATEMENT            |      |
|   1 |  UPDATE                     | EMP  |
|   2 |   TABLE ACCESS BY USER ROWID| EMP  |
--------------------------------------------
```

Note that the TABLE ACCESS BY USER ROWID operation is exclusively used when the rowid is directly passed as a parameter or literal. In the next section, you'll see that when a rowid is extracted from an index, the TABLE ACCESS BY INDEX ROWID operation is used instead. The effectiveness of these table accesses is the same; the two operations are used only to distinguish the source of the rowid.

When a SQL statement specifies several rowids through an IN condition, an additional INLIST ITERATOR operation appears in the execution plan. The following query is an example. Note that operation 1 (the parent) simply indicates that operation 2 (the child) is processed several times. According to the value of the Starts column of operation 2, it's processed twice. In other words, the emp table is accessed twice through a rowid:

```
SELECT * FROM emp WHERE rowid IN ('AAADGZAAEAAAAAoAAH', 'AAADGZAAEAAAAAoAAI')

-----------------------------------------------------
| Id  | Operation                     | Name | Starts |
-----------------------------------------------------
|   0 | SELECT STATEMENT              |      |      1 |
|   1 |  INLIST ITERATOR              |      |      1 |
|   2 |   TABLE ACCESS BY USER ROWID| EMP  |      2 |
-----------------------------------------------------
```

In summary, whenever specific rows are accessed at least two times, you should consider getting the rowid during the first access and then take advantage of it for subsequent accesses.

## Index Access

Index accesses are by far the most often used access paths for SQL statements with strong selectivity. To take advantage of them, you have to to apply at least one of the restrictions present in the WHERE clause or a join condition through an index. To do that, it's essential to not only index the columns that provide strong selectivity but also to understand which type of conditions might be applied efficiently through an index. The database engine supports different types of indexes. Before describing the properties and access paths supported by B-tree and bitmap indexes in detail, it's important to discuss the clustering factor or, in other words, why the distribution of data impacts the performance of an index scan (see Figure 13-4).

---

■ **Note** Although the database engine supports domain indexes for complex data such as PDF documents or images, this isn't covered in this book. Refer to *Oracle Database Documentation* for further information.

---

### Clustering Factor

As described in Chapter 8, the clustering factor indicates how many adjacent index keys don't refer to the same data block in the table (bitmap indexes are an exception because their clustering factor is always set to the number of keys in the index). Here's a mental picture that might help: if the whole table is accessed through an index and in the buffer cache there's a single buffer to store the data blocks, the clustering factor is the number of physical reads performed against the table. For example, the clustering factor of the index shown in Figure 13-12 is 10 (notice that there are 12 rows and only two adjacent index keys, which are highlighted, refer to the same data block).

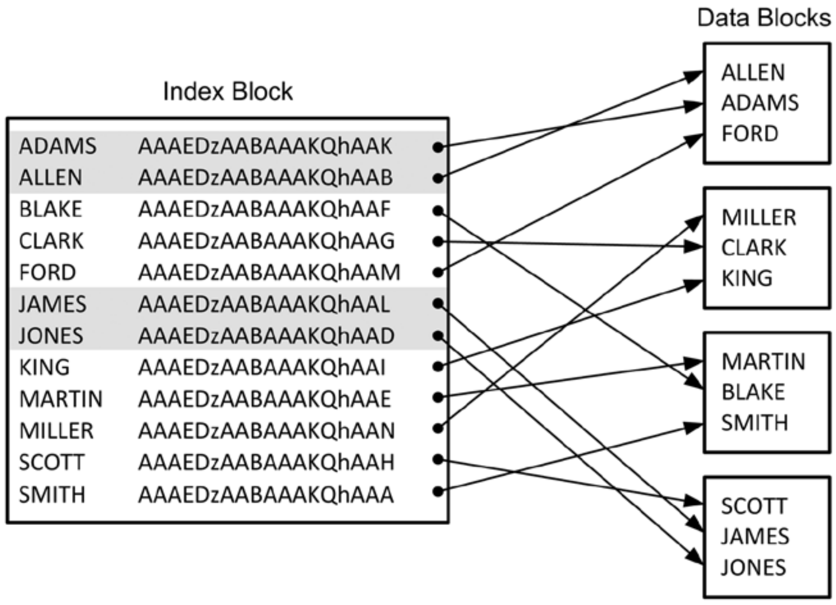


***Figure 13-12.*** *Relationship between index blocks and data blocks*

The following PL/SQL function, which is available in the `clustering_factor.sql` script, illustrates how it's computed. Note that this function works only for single-column B-tree indexes:

```
CREATE OR REPLACE FUNCTION clustering_factor (
  p_owner IN VARCHAR2,
  p_table_name IN VARCHAR2,
  p_column_name IN VARCHAR2
) RETURN NUMBER IS
  l_cursor             SYS_REFCURSOR;
  l_clustering_factor  BINARY_INTEGER := 0;
  l_block_nr           BINARY_INTEGER := 0;
  l_previous_block_nr  BINARY_INTEGER := 0;
  l_file_nr            BINARY_INTEGER := 0;
  l_previous_file_nr   BINARY_INTEGER := 0;
BEGIN
  OPEN l_cursor FOR
    'SELECT dbms_rowid.rowid_block_number(rowid) block_nr, '||
    '       dbms_rowid.rowid_to_absolute_fno(rowid, '''||
                                        p_owner||''','''||
                                        p_table_name||''') file_nr '||
    'FROM '||p_owner||'.'||p_table_name||' '||
    'WHERE '||p_column_name||' IS NOT NULL '||
    'ORDER BY ' || p_column_name ||', rowid';
  LOOP
    FETCH l_cursor INTO l_block_nr, l_file_nr;
    EXIT WHEN l_cursor%NOTFOUND;
    IF (l_previous_block_nr <> l_block_nr OR l_previous_file_nr <> l_file_nr)
    THEN
      l_clustering_factor := l_clustering_factor + 1;
    END IF;
    l_previous_block_nr := l_block_nr;
    l_previous_file_nr := l_file_nr;
  END LOOP;
  CLOSE l_cursor;
  RETURN l_clustering_factor;
END;
```

Notice how the values it generates match the statistics stored in the data dictionary:

```
SQL> SELECT i.index_name, i.clustering_factor,
  2         clustering_factor(user, i.table_name, ic.column_name) AS my_clus_fact
  3  FROM user_indexes i, user_ind_columns ic
  4  WHERE i.table_name = 'T'
  5  AND i.index_name = ic.index_name
  6  ORDER BY i.index_name;

INDEX_NAME CLUSTERING_FACTOR MY_CLUS_FACT
---------- ----------------- ------------
T_PK                     990          990
T_VAL_I                   77           77
```

## THE CLUSTERING FACTOR COMPUTATION IS PESSIMISTIC

The algorithm used by the `dbms_stats` package for computing the clustering factor is rather pessimistic. In fact, the algorithm considers that during an index range scan, only the block referenced by the previous index key stays in the cache. In practice, several of the previous blocks likely remain in the cache. As a result, it's not uncommon that the clustering factor doesn't accurately describe the real data distribution.

To prevent or solve problems related to this pessimistic computation, as of version 11.2.0.4 (or when a patch implementing the enhancement associated to bug 13262857 is installed) the `table_cached_blocks` preference can be set through the `dbms_stats` package. Values between 1 and 255 specify how many blocks are expected to be cached. The default value is 1. Even though advising a value is always very difficult, it's likely that all values larger than 1 lead to more reliable clustering factors. In fact, the default value is way too pessimistic. In addition, the value `dbms_stats.auto_table_cached_blocks` specifies to use either 255, 1% of the table blocks or 0.1% of the buffer cache, whichever is less.

From a performance point of view, you should avoid row-based processing. As discussed previously in this chapter, the database engine, thanks to row prefetching, also tries to avoid it as much as possible. In fact, when several rows have to be extracted from the same block, instead of accessing the block (that is, doing a logical read) once per row, all rows are extracted in a single access. To emphasize this point, let's look at an example based on the `clustering_factor.sql` script. The following is the test table created by the script:

```
SQL> CREATE TABLE t (
  2    id NUMBER,
  3    val NUMBER,
  4    pad VARCHAR2(4000),
  5    CONSTRAINT t_pk PRIMARY KEY (id)
  6  );

SQL> INSERT INTO t
  2  SELECT rownum, dbms_random.value, dbms_random.string('p',500)
  3  FROM dual
  4  CONNECT BY level <= 1000;
```

Because the values of the id column are inserted in increasing order, the clustering factor of the index that supports the primary key is close to the number of blocks in the table. In other words, it's very good:

```
SQL> SELECT blocks, num_rows
  2  FROM user_tables
  3  WHERE table_name = 'T';

    BLOCKS   NUM_ROWS
---------- ----------
        80       1000

SQL> SELECT blevel, leaf_blocks, clustering_factor
  2  FROM user_indexes
  3  WHERE index_name = 'T_PK';

    BLEVEL LEAF_BLOCKS CLUSTERING_FACTOR
---------- ----------- -----------------
         1           2                77
```

It's now useful to look at the number of logical reads performed when the whole table is accessed through the primary key (a hint is used to force such an execution plan). The first test is performed with row prefetching set to 2. Hence, at least 500 calls to the database engine must be performed to retrieve the 1,000 rows. This behavior can be confirmed by looking at the number of logical reads (Buffers column): 503 on the index and 539 (1,042 – 503) on the table. Basically, for each call, two rowids were extracted from the index block, and their data was found in the same data block almost every time, thanks to the good clustering factor:

```
SQL> set arraysize 2

SQL> SELECT /*+ index(t t_pk) */ * FROM t;

-------------------------------------------------------------------------
| Id  | Operation                   | Name | Starts | A-Rows | Buffers |
-------------------------------------------------------------------------
|   0 | SELECT STATEMENT            |      |      1 |   1000 |    1042 |
|   1 |  TABLE ACCESS BY INDEX ROWID | T    |      1 |   1000 |    1042 |
|   2 |   INDEX FULL SCAN           | T_PK |      1 |   1000 |     503 |
-------------------------------------------------------------------------
```

The second test is performed with row prefetching set to 100. Hence, ten calls to the database engine are enough to retrieve all rows. Also in this case, this behavior is confirmed by looking at the number of logical reads: 13 on the index and 87 (100 – 13) on the table. Basically, for each call, 100 rowids were extracted from the index block, and their corresponding table rows were often found in the same data block, that was accessed only once:

```
SQL> set arraysize 100

SQL> SELECT /*+ index(t t_pk) */ * FROM t;

-----------------------------------------------------------------------
| Id  | Operation                   | Name | Starts | A-Rows | Buffers |
-----------------------------------------------------------------------
|   0 | SELECT STATEMENT            |      |      1 |   1000 |     100 |
|   1 |  TABLE ACCESS BY INDEX ROWID | T    |      1 |   1000 |     100 |
|   2 |   INDEX FULL SCAN           | T_PK |      1 |   1000 |      13 |
-----------------------------------------------------------------------
```

Now let's perform the same test with a much higher clustering factor. To achieve this, an ORDER BY clause based on a random value is added to the INSERT statement used to load data into the table. The only statistic that changes is the clustering factor. Notice that it's close to the number of rows. In other words, it's very bad:

```
SQL> TRUNCATE TABLE t;

SQL> INSERT INTO t
  2  SELECT rownum, dbms_random.value, dbms_random.string('p',500)
  3  FROM dual
  4  CONNECT BY level <= 1000
  5  ORDER BY dbms_random.value;

SQL> SELECT blocks, num_rows
  2  FROM user_tables
  3  WHERE table_name = 'T';

    BLOCKS   NUM_ROWS
---------- ----------
        80       1000

SQL> SELECT blevel, leaf_blocks, clustering_factor
  2  FROM user_indexes
  3  WHERE index_name = 'T_PK';

    BLEVEL LEAF_BLOCKS CLUSTERING_FACTOR
---------- ----------- -----------------
         1           2               990
```

The following are the numbers of logical reads for the two tests. On one hand, for both tests the number of logical reads on the index has not changed. This makes sense because, in both cases, the index stores the same keys in exactly the same order (only the corresponding rowids are different). On the other hand, for both tests the number of logical reads on the table is close to the number of rows. This is reasonable because adjacent rowids in the index almost never refer to the same block:

```
SQL> set arraysize 2

SQL> SELECT /*+ index(t t_pk) */ * FROM t;

-----------------------------------------------------------------------
| Id  | Operation                   | Name | Starts | A-Rows | Buffers |
-----------------------------------------------------------------------
|   0 | SELECT STATEMENT            |      |      1 |   1000 |    1499 |
|   1 |  TABLE ACCESS BY INDEX ROWID| T    |      1 |   1000 |    1499 |
|   2 |   INDEX FULL SCAN           | T_PK |      1 |   1000 |     502 |
-----------------------------------------------------------------------

SQL> set arraysize 100

-----------------------------------------------------------------------
| Id  | Operation                   | Name | Starts | A-Rows | Buffers |
-----------------------------------------------------------------------
|   0 | SELECT STATEMENT            |      |      1 |   1000 |    1003 |
|   1 |  TABLE ACCESS BY INDEX ROWID| T    |      1 |   1000 |    1003 |
|   2 |   INDEX FULL SCAN           | T_PK |      1 |   1000 |      13 |
-----------------------------------------------------------------------
```

In summary, row prefetching is less effective with a high clustering factor, and therefore, a higher number of logical reads is performed. The impact of the clustering factor on the resource consumption is so high that the query optimizer (as described in Chapter 9, specifically, Formula 9-4) does use the clustering factor to compute the cost related to index accesses, which is very often the main factor in the calculation.

## B-tree Indexes vs. Bitmap Indexes

Simply put, there are some situations where only B-tree indexes can be considered. If you aren't in one of those situations, bitmap indexes should be taken into consideration most of the time. Table 13-2 summarizes the features that you need to take into account when deciding between B-tree and bitmap indexes.

***Table 13-2.*** *Essential Features Supported only by B-tree and Bitmap Indexes*

| Feature | B-tree | Bitmap |
|---|---|---|
| Primary and unique key | ✓ | |
| Row-level locking | ✓ | |
| Efficient combination of several indexes | | ✓ |
| Global indexes and non-partitioned indexes on partitioned tables | ✓ | |

■ **Note** Bitmap indexes are available only in Enterprise Edition.

The use of bitmap indexes is mainly limited by two situations. First, only B-tree indexes can be used for primary and unique keys. There's simply no choice here. Second, only B-tree indexes support row-level locking because locks in (B-tree and bitmap) indexes are internally set for an index entry. Because a single bitmap index entry might index thousands of rows, a modification of a bitmap-indexed column may prevent the concurrent modification of thousands of other rows (those referenced by the same index entry), which may greatly inhibit scalability. Another downside of bitmap indexes is that the database engine generates more redo when they have to be modified. This is because, in most situations, the keys of a bitmap index are larger than those of a B-tree index.

Note that the selectivity or the number of distinct keys of the index is irrelevant to choose between B-tree and bitmap indexes. This is true despite the fact that many books and papers about bitmap indexes contain advice like the following:

> *Bitmap indexes are suitable for low cardinality data that is infrequently modified. Data has low cardinality when the number of distinct values in a column is low in relation to the total number of rows.*
>
> *Through compression techniques, these indexes can generate many rowids with minimal I/O. Bitmap indexes provide especially useful access paths in queries that contain the following:*
>
> - *Multiple conditions in the WHERE clause*
> - *AND and OR operations on low cardinality columns*
> - *The COUNT function*
> - *Predicates that select for null values*
>
> —*Oracle Database SQL Tuning Guide* 12c Release 1

Honestly, in my opinion, such information is at the very least misleading. The fact is, a SQL statement with weak selectivity can never be efficiently executed by getting a list of rowids from an index. That's because the time needed to build the list of rowids is much smaller, both for B-tree and for bitmap indexes, than the time needed to access the table with them in such situations—hence, most of the time is going to be spent reading the table, regardless of the index being a B-tree or a bitmap. That said, it's true that bitmap indexes behave better than B-tree indexes with a low number of distinct keys (note that in the excerpt, the term *cardinality* has a different meaning than the one used in this book!). But be careful, *better* doesn't necessarily mean *efficient*. For example, a *bad* product isn't *good* simply because it's better than a *very bad* product. It may be that combining bitmap indexes is very efficient, but, once again, building a list of rowids is just the beginning. The rows still have to be accessed. I should also mention `OR`. If you think about it, you'll realize that combining several nonselective conditions with `OR` can lead only to even weaker selectivity, while the goal is to increase it if you want to efficiently use indexes.

---

■ **Tip** Forget about selectivity and cardinality when you have to choose between a B-tree index and a bitmap index.

---

In addition to the differences summarized in Table 13-2, B-tree indexes and bitmap indexes don't show the same efficiency when dealing with the same SQL conditions. Actually, bitmap indexes are usually more powerful. Table 13-3 summarizes, for both types of indexes, their ability to cope with different types of conditions. The aim of the following sections is to provide some examples regarding this. I also describe different properties and limitations.

***Table 13-3.*** *Conditions That Can Lead to an Index Unique/Range Scan*

| Condition | B-tree | Bitmap |
|---|---|---|
| Equality (=) | ✓ | ✓ |
| `IS NULL` | ✓* | ✓ |
| Range (`BETWEEN`, >, >=, < and <=) | ✓ | ✓ |
| `IN` | ✓ | ✓ |
| `LIKE` | ✓ | ✓ |
| Inequality (!=, <>) and `IS NOT NULL` | | ✓† |

**Never applicable to single-column indexes; applicable to composite indexes only if either another condition leading to an index range scan is specified or the NULL values are guaranteed to be stored in the index.*
*†Applicable only when several bitmaps are combined.*

Many examples in the following sections are based on the `conditions.sql` script. The test table and its indexes are created with the following SQL statements:

```
CREATE TABLE t (
  id NUMBER,
  d1 DATE,
  n1 NUMBER,
  n2 NUMBER,
  n3 NUMBER,
  n4 NUMBER,
  n5 NUMBER,
  n6 NUMBER,
  c1 VARCHAR2(20),
  c2 VARCHAR2(20),
  pad VARCHAR2(4000),
  CONSTRAINT t_pk PRIMARY KEY (id)
);

CREATE INDEX i_n1 ON t (n1);

CREATE INDEX i_n2 ON t (n2);

CREATE INDEX i_n3 ON t (n3);

CREATE INDEX i_n123 ON t (n1, n2, n3);

CREATE BITMAP INDEX i_n4 ON t (n4);

CREATE BITMAP INDEX i_n5 ON t (n5);

CREATE BITMAP INDEX i_n6 ON t (n6);

CREATE INDEX i_c1 ON t (c1);

CREATE BITMAP INDEX i_c2 ON t (c2);
```

## Equality Conditions and B-tree Indexes

With B-tree indexes, equality conditions are carried out with one of two operations. The first, INDEX UNIQUE SCAN, is used exclusively with unique indexes. As the name suggests, at most one rowid is returned with it. The following query is an example. The execution plan confirms, through the access predicate of operation 2, that the condition on the id column is applied with the t_pk index. Then, operation 1 uses the rowid extracted from the index to access the table. This is carried out with the TABLE ACCESS BY INDEX ROWID operation. Notice that both operations are executed only once:

```
SELECT /*+ index(t) */ * FROM t WHERE id = 6

--------------------------------------------------------------
| Id  | Operation                   | Name | Starts | A-Rows |
--------------------------------------------------------------
|   0 | SELECT STATEMENT            |      |      1 |      1 |
|   1 |  TABLE ACCESS BY INDEX ROWID| T    |      1 |      1 |
|*  2 |   INDEX UNIQUE SCAN         | T_PK |      1 |      1 |
--------------------------------------------------------------

   2 - access("ID"=6)
```

■ **Note** In this section, to force index scans, the index hint is used by specifying the name of the table only. When it's used in this way, the query optimizer is free to choose one of the indexes from those available. In all the examples, a single predicate is present in the WHERE clause. For this reason, the query optimizer always chooses the index based on the column referenced in that predicate.

The second operation, INDEX RANGE SCAN, is used with nonunique indexes. The only difference between this operation and the previous one is that it can extract many rowids (527 in the example), not just one:

```
SELECT /*+ index_asc(t) */ * FROM t WHERE n1 = 6

--------------------------------------------------------------
| Id  | Operation                   | Name | Starts | A-Rows |
--------------------------------------------------------------
|   0 | SELECT STATEMENT            |      |      1 |    527 |
|   1 |  TABLE ACCESS BY INDEX ROWID| T    |      1 |    527 |
|*  2 |   INDEX RANGE SCAN          | I_N1 |      1 |    527 |
--------------------------------------------------------------

   2 - access("N1"=6)
```

From version 12.1 onward, the operation used to access a table based on rowids returned from an index range scan is TABLE ACCESS BY INDEX ROWID BATCHED. The aim of this operation is to optimize the table accesses by taking advantage of the fact that several rowids are accessed. The following example shows the execution plan for the same query as the previous one:

```
SELECT /*+ index_asc(t) */ * FROM t WHERE n1 = 6
```

```
-------------------------------------------------------------------
| Id  | Operation                           | Name | Starts | A-Rows |
-------------------------------------------------------------------
|   0 | SELECT STATEMENT                    |      |      1 |    527 |
|   1 |  TABLE ACCESS BY INDEX ROWID BATCHED| T    |      1 |    527 |
|*  2 |   INDEX RANGE SCAN                  | I_N1 |      1 |    527 |
-------------------------------------------------------------------

   2 - access("N1"=6)
```

Per default, an index scan is performed in ascending order. Therefore, the `index` hint instructs the query optimizer to behave in that way also. To explicitly specify the scan order, it's possible to use the `index_asc` and `index_desc` hints. The following query shows how. In the execution plan, the scan in descending order is shown through the `INDEX RANGE SCAN DESCENDING` operation:

```
SELECT /*+ index_desc(t) */ * FROM t WHERE n1 = 6

------------------------------------------------------------
| Id  | Operation                    | Name | Starts | A-Rows |
------------------------------------------------------------
|   0 | SELECT STATEMENT             |      |      1 |    527 |
|   1 |  TABLE ACCESS BY INDEX ROWID | T    |      1 |    527 |
|*  2 |   INDEX RANGE SCAN DESCENDING| I_N1 |      1 |    527 |
------------------------------------------------------------

   2 - access("N1"=6)
       filter("N1"=6)
```

Note that `INDEX RANGE SCAN` and `INDEX RANGE SCAN DESCENDING` return the same data. Only the order is different. Later on, the section on range conditions describes when such an access path is useful.

## Equality Conditions and Bitmap Indexes

With bitmap indexes, equality conditions are carried out in three operations. In order of execution, the first operation is `BITMAP INDEX SINGLE VALUE`, which scans the index and applies the restriction. As the name suggests, this operation looks for a single value. The second operation, `BITMAP CONVERSION TO ROWIDS`, converts the bitmaps it gets from the first operation into a list of rowids. The third operation accesses the table with the list of rowids built by the second operation. Notice that all three operations are executed only once:

```
SELECT /*+ index(t i_n4) */ * FROM t WHERE n4 = 6

------------------------------------------------------------
| Id  | Operation                    | Name | Starts | A-Rows |
------------------------------------------------------------
|   0 | SELECT STATEMENT             |      |      1 |    527 |
|   1 |  TABLE ACCESS BY INDEX ROWID | T    |      1 |    527 |
|   2 |   BITMAP CONVERSION TO ROWIDS|      |      1 |    527 |
|*  3 |    BITMAP INDEX SINGLE VALUE | I_N4 |      1 |      1 |
------------------------------------------------------------

   3 - access("N4"=6)
```

■ **Caution** In this section, to force index scans, the `index` hint is used by specifying both the name of the table and the name of the index. In other words, the hint specifies which index should be used. When it's used in this way, the hint is valid only if an index with that name exists. Because the name of an index can be easily changed (for example, with the `ALTER INDEX RENAME` statement), in practice it's also very easy for hints with that syntax to be invalidated by mistake.

As for B-tree indexes, from version 12.1 onward the operation used to access a table with several rowids returned from an index scan is `TABLE ACCESS BY INDEX ROWID BATCHED`. The following example shows the execution plan for the same query as the previous one:

```
SELECT /*+ index(t i_n4) */ * FROM t WHERE n4 = 6

----------------------------------------------------------------------
| Id  | Operation                            | Name | Starts | A-Rows |
----------------------------------------------------------------------
|   0 | SELECT STATEMENT                     |      |      1 |    527 |
|   1 |  TABLE ACCESS BY INDEX ROWID BATCHED| T    |      1 |    527 |
|   2 |   BITMAP CONVERSION TO ROWIDS        |      |      1 |    527 |
|*  3 |    BITMAP INDEX SINGLE VALUE         | I_N4 |      1 |      1 |
----------------------------------------------------------------------

   3 - access("N4"=6)
```

## IS NULL Conditions and B-tree Indexes

With B-tree indexes, `IS NULL` conditions can be applied only through composite B-tree indexes if either another condition leading to an index range scan is specified or the `NULL` values are guaranteed to be stored in the index. At least one of these two conditions must be met, because index entries that only have `NULL` values are neither stored in a single-column index nor in a composite index.

The following query demonstrates the case where two conditions are specified. The execution plan confirms, through the access predicate of operation 2, that the condition on the `n2` column is applied using the `i_n123` index. Also notice that operation 2 returns only 5 rows, whereas for the example in the "Equality Conditions and B-tree Indexes" section, which doesn't have the `n2 IS NULL` restriction, the range scan returned 527 rows:

```
SELECT /*+ index(t) */ * FROM t WHERE n1 = 6 AND n2 IS NULL

----------------------------------------------------------------
| Id  | Operation                    | Name   | Starts | A-Rows |
----------------------------------------------------------------
|   0 | SELECT STATEMENT             |        |      1 |      5 |
|   1 |  TABLE ACCESS BY INDEX ROWID| T      |      1 |      5 |
|*  2 |   INDEX RANGE SCAN           | I_N123 |      1 |      5 |
----------------------------------------------------------------

   2 - access("N1"=6 AND "N2" IS NULL)
```

As the following example shows, the very same execution plan is also used when the IS NULL condition is specified for the leading column of the index:

```
SELECT /*+ index(t) */ * FROM t WHERE n1 IS NULL AND n2 = 8

----------------------------------------------------------------
| Id  | Operation                    | Name   | Starts | A-Rows |
----------------------------------------------------------------
|   0 | SELECT STATEMENT             |        |      1 |      4 |
|   1 |  TABLE ACCESS BY INDEX ROWID| T      |      1 |      4 |
|*  2 |   INDEX RANGE SCAN           | I_N123 |      1 |      4 |
----------------------------------------------------------------

   2 - access("N1" IS NULL AND "N2"=8)
       filter("N2"=8)
```

The other case where IS NULL conditions can be applied through composite B-tree indexes is when the NULL values are guaranteed to be stored in the index. The following is a query taking advantage of this case. Notice that, because of the n2 IS NOT NULL condition, it's guaranteed that all rows fulfilling the WHERE clause have a corresponding index entry in the i_n123 index. As a result, an index range scan can be used to find them:

```
SELECT /*+ index(t) */ * FROM t WHERE n1 IS NULL AND n2 IS NOT NULL

----------------------------------------------------------------
| Id  | Operation                    | Name   | Starts | A-Rows |
----------------------------------------------------------------
|   0 | SELECT STATEMENT             |        |      1 |    521 |
|   1 |  TABLE ACCESS BY INDEX ROWID| T      |      1 |    521 |
|*  2 |   INDEX RANGE SCAN           | I_N123 |      1 |    521 |
----------------------------------------------------------------

   2 - access("N1" IS NULL)
       filter("N2" IS NOT NULL)
```

The NULL values are also guaranteed to be stored in a composite index when at least one of the columns isn't nullable. A special case satisfying this condition is an index created on a nullable column and a constant value. It goes without saying that this is a trick. But, in some situations, it's a useful one. The following index shows an example (notice that instead of the "0", another value could have been used):

```
CREATE INDEX i_n1_nn ON t (n1, 0)
```

With such an index in place, a query like the following is able to carry out an index range scan:

```
SELECT /*+ index(t) */ * FROM t WHERE n1 IS NULL

--------------------------------------------------------------------------
| Id  | Operation                    | Name    | Starts | E-Rows | A-Rows |
--------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |         |      1 |        |    526 |
|   1 |  TABLE ACCESS BY INDEX ROWID| T       |      1 |    526 |    526 |
|*  2 |   INDEX RANGE SCAN           | I_N1_NN |      1 |    526 |    526 |
--------------------------------------------------------------------------

   2 - access("N1" IS NULL)
```

Single-column indexes, however, can't be used to apply IS NULL conditions. This is because the NULL values aren't stored in the index. Therefore, the query optimizer is simply unable to take advantage of an index range scan in such a case. Even if you try to force its utilization with the index hint, a full table scan or a full index scan is performed:

```
SELECT /*+ index(t i_n1) */ * FROM t WHERE n1 IS NULL

----------------------------------------------------
| Id  | Operation         | Name | Starts | A-Rows |
----------------------------------------------------
|   0 | SELECT STATEMENT  |      |      1 |    526 |
|*  1 |  TABLE ACCESS FULL| T    |      1 |    526 |
----------------------------------------------------

   1 - filter("N1" IS NULL)
```

## IS NULL Conditions and Bitmap Indexes

With bitmap indexes, IS NULL conditions are carried out in the same way as equality conditions. This is possible because the bitmap index stores NULL values in the same way as any other value:

```
SELECT /*+ index(t i_n4) */ * FROM t WHERE n4 IS NULL

---------------------------------------------------------------
| Id  | Operation                    | Name | Starts | A-Rows |
---------------------------------------------------------------
|   0 | SELECT STATEMENT             |      |      1 |    526 |
|   1 |  TABLE ACCESS BY INDEX ROWID | T    |      1 |    526 |
|   2 |   BITMAP CONVERSION TO ROWIDS|      |      1 |    526 |
|*  3 |    BITMAP INDEX SINGLE VALUE | I_N4 |      1 |      1 |
---------------------------------------------------------------

   3 - access("N4" IS NULL)
```

## Range Conditions and B-tree Indexes

With B-tree indexes, range conditions are carried out in the same way as equality conditions on nonunique indexes, or, in other words, with the INDEX RANGE SCAN operation. For range conditions, the index type (that is, its uniqueness) isn't relevant. Because it's a range scan, several rowids could always be returned. For example, the following query shows a range condition that is applied to the column that the primary key consists of:

```
SELECT /*+ index(t (t.id)) */ * FROM t WHERE id BETWEEN 6 AND 19

--------------------------------------------------------------
| Id  | Operation                   | Name | Starts | A-Rows |
--------------------------------------------------------------
|   0 | SELECT STATEMENT            |      |      1 |     14 |
|   1 |  TABLE ACCESS BY INDEX ROWID| T    |      1 |     14 |
|*  2 |   INDEX RANGE SCAN          | T_PK |      1 |     14 |
--------------------------------------------------------------

   2 - access("ID">=6 AND "ID"<=19)
```

---

■ **Note** In this section, several hints are used to force index scans by specifying the name of the table and on which columns the index has to be created. The advantage of this syntax compared to the method where the index name is specified is that the hint doesn't depend on the index name. This gives greater robustness to the hint. Its disadvantage is that the hint doesn't guarantee that the query optimizer always selects the same index.

---

As mentioned in the previous section, the index scan is performed in ascending order per default. This means that when an ORDER BY using binary comparisons (later on, the "Linguistic Indexes" section provides information about the different types of comparison and how to deal with them) is applied to the same column as the range condition, the result set is already sorted. As a result, no explicit sort is carried out. However, when the ORDER BY is required in descending order, an explicit sort needs to be executed, as the following query illustrates. The sort is carried out by operation 1, SORT ORDER BY. Notice how the index scan in ascending order is forced by the index_asc hint:

```
SELECT /*+ index_asc(t (t.id)) */ * FROM t WHERE id BETWEEN 6 AND 19 ORDER BY id DESC

--------------------------------------------------------------
| Id  | Operation                    | Name | Starts | A-Rows |
--------------------------------------------------------------
|   0 | SELECT STATEMENT             |      |      1 |     14 |
|   1 |  SORT ORDER BY               |      |      1 |     14 |
|   2 |   TABLE ACCESS BY INDEX ROWID| T    |      1 |     14 |
|*  3 |    INDEX RANGE SCAN          | T_PK |      1 |     14 |
--------------------------------------------------------------

   3 - access("ID">=6 AND "ID"<=19)
```

The same query can naturally take advantage of a descending index scan to avoid the explicit sort. Here's an example, where the SORT ORDER BY operation is no longer present in the execution plan:

```
SELECT /*+ index_desc(t (t.id)) */ * FROM t WHERE id BETWEEN 6 AND 19 ORDER BY id DESC

-----------------------------------------------------------------------
| Id  | Operation                    | Name | Starts | E-Rows | A-Rows |
-----------------------------------------------------------------------
|   0 | SELECT STATEMENT             |      |      1 |        |     14 |
|   1 |  TABLE ACCESS BY INDEX ROWID | T    |      1 |     15 |     14 |
|*  2 |   INDEX RANGE SCAN DESCENDING| T_PK |      1 |     15 |     14 |
-----------------------------------------------------------------------

   2 - access("ID"<=19 AND "ID">=6)
```

## Range Conditions and Bitmap Indexes

With bitmap indexes, range conditions are carried out in a similar way as equality conditions. The only difference is that the `BITMAP INDEX RANGE SCAN` operation is used instead of the `BITMAP INDEX SINGLE VALUE` operation:

```
SELECT /*+ index(t (t.n4)) */ * FROM t WHERE n4 BETWEEN 6 AND 19

---------------------------------------------------------------
| Id  | Operation                      | Name | Starts | A-Rows |
---------------------------------------------------------------
|   0 | SELECT STATEMENT               |      |      1 |   6840 |
|   1 |  TABLE ACCESS BY INDEX ROWID   | T    |      1 |   6840 |
|   2 |   BITMAP CONVERSION TO ROWIDS  |      |      1 |   6840 |
|*  3 |    BITMAP INDEX RANGE SCAN     | I_N4 |      1 |     13 |
---------------------------------------------------------------

   3 - access("N4">=6 AND "N4"<=19)
```

With bitmap indexes, because there's no concept of ascending and descending scans, it's not possible to avoid, and thereby optimize, `ORDER BY` operations.

## IN Conditions

`IN` conditions don't have a specific access path. Instead, in the execution plan, the `INLIST ITERATOR` operation points out that part of the execution plan is executed several times because of an `IN` condition. The following three queries show how the operation used for the index scan itself depends on the index type. The first is a unique index, the second is a nonunique B-tree index, and the third is a bitmap index. Basically, an `IN` condition is just a series of equality conditions. Note that operations related to the index and table access are executed once for each value in the `IN` list (see the `Starts` column):

```
SELECT /*+ index(t t_pk) */ * FROM t WHERE id IN (6, 8, 19, 28)

---------------------------------------------------------------
| Id  | Operation                      | Name | Starts | A-Rows |
---------------------------------------------------------------
|   0 | SELECT STATEMENT               |      |      1 |      4 |
|   1 |  INLIST ITERATOR               |      |      1 |      4 |
|   2 |   TABLE ACCESS BY INDEX ROWID  | T    |      4 |      4 |
|*  3 |    INDEX UNIQUE SCAN           | T_PK |      4 |      4 |
---------------------------------------------------------------

   3 - access(("ID"=6 OR "ID"=8 OR "ID"=19 OR "ID"=28))
```

```
SELECT /*+ index(t i_n1) */ * FROM t WHERE n1 IN (6, 8, 19, 28)

------------------------------------------------------------------------
| Id  | Operation                    | Name | Starts | E-Rows | A-Rows |
------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |      |      1 |        |   1579 |
|   1 |  INLIST ITERATOR             |      |      1 |        |   1579 |
|   2 |   TABLE ACCESS BY INDEX ROWID| T    |      4 |   1710 |   1579 |
|*  3 |    INDEX RANGE SCAN          | I_N1 |      4 |   1710 |   1579 |
------------------------------------------------------------------------

   3 - access(("N1"=6 OR "N1"=8 OR "N1"=19 OR "N1"=28))

SELECT /*+ index(t i_n4) */ * FROM t WHERE n4 IN (6, 8, 19, 28)

----------------------------------------------------------------
| Id  | Operation                     | Name | Starts | A-Rows |
----------------------------------------------------------------
|   0 | SELECT STATEMENT              |      |      1 |   1579 |
|   1 |  INLIST ITERATOR              |      |      1 |   1579 |
|   2 |   TABLE ACCESS BY INDEX ROWID | T    |      4 |   1579 |
|   3 |    BITMAP CONVERSION TO ROWIDS|      |      4 |   1579 |
|*  4 |     BITMAP INDEX SINGLE VALUE | I_N4 |      4 |      3 |
----------------------------------------------------------------

   4 - access(("N4"=6 OR "N4"=8 OR "N4"=19 OR "N4"=28))
```

## DYNAMIC IN CONDITIONS WITH MANY EXPRESSIONS

Oracle Database doesn't support IN conditions with more than 1,000 expressions. Even though you could use several disjunctive predicates as a workaround, from a performance point of view, there's a good reason for setting a limit. Hence, you should neither use IN conditions with many expressions nor disjunctive predicates, to avoid the 1,000 expressions limit. In fact, long lists of expressions frequently lead to performance problems. You should avoid them as much as possible. Instead, you should implement one of the following techniques:

- Use an IN condition based on a subquery reading a (temporary) table.
- Use an IN condition based on a subquery with a pipelined table function that takes as input a nested table based on an object type and returns one row for each element.
- Use a MEMBER condition that tests whether an element is member of a nested table based on an object type.

The dynamic_in_conditions.sql script provides an example for each one of these techniques.

## LIKE Conditions

The database engine is able to apply LIKE conditions as access predicates based on the character string preceeding the first wild card only. As a result, provided that patterns don't begin with a wildcard (the underscore and the percent characters), LIKE conditions are carried out in the same way as range conditions. Otherwise, a full table scan or a full index scan can't be avoided. The following examples show this behavior. The first two queries retrieve all rows that begin with the letter *A*, for the c1 and c2 columns, respectively. Hence, a range scan is possible. The third and forth queries retrieve all rows that contain the letter *A* in any position for the c1 and c2 columns, respectively. Hence, a full index scan is performed:

```
SELECT /*+ index(t i_c1) */ * FROM t WHERE c1 LIKE 'A%'

-------------------------------------------------------------
| Id  | Operation                    | Name | Starts | A-Rows |
-------------------------------------------------------------
|   0 | SELECT STATEMENT             |      |      1 |    119 |
|   1 |  TABLE ACCESS BY INDEX ROWID| T    |      1 |    119 |
|*  2 |   INDEX RANGE SCAN           | I_C1 |      1 |    119 |
-------------------------------------------------------------

   2 - access("C1" LIKE 'A%')
       filter("C1" LIKE 'A%')

SELECT /*+ index(t i_c2) */ * FROM t WHERE c2 LIKE 'A%'

--------------------------------------------------------------
| Id  | Operation                     | Name | Starts | A-Rows |
--------------------------------------------------------------
|   0 | SELECT STATEMENT              |      |      1 |    108 |
|   1 |  TABLE ACCESS BY INDEX ROWID  | T    |      1 |    108 |
|   2 |   BITMAP CONVERSION TO ROWIDS|      |      1 |    108 |
|*  3 |    BITMAP INDEX RANGE SCAN    | I_C2 |      1 |    108 |
--------------------------------------------------------------

   3 - access("C2" LIKE 'A%')
       filter(("C2" LIKE 'A%' AND "C2" LIKE 'A%'))

SELECT /*+ index(t i_c1) */ * FROM t WHERE c1 LIKE '%A%'

-------------------------------------------------------------
| Id  | Operation                    | Name | Starts | A-Rows |
-------------------------------------------------------------
|   0 | SELECT STATEMENT             |      |      1 |   1921 |
|   1 |  TABLE ACCESS BY INDEX ROWID| T    |      1 |   1921 |
|*  2 |   INDEX FULL SCAN            | I_C1 |      1 |   1921 |
-------------------------------------------------------------

   2 - filter(("C1" LIKE '%A%' AND "C1" IS NOT NULL))

SELECT /*+ index(t i_c2) */ * FROM t WHERE c2 LIKE '%A%'

--------------------------------------------------------------
| Id  | Operation                     | Name | Starts | A-Rows |
--------------------------------------------------------------
|   0 | SELECT STATEMENT              |      |      1 |   1846 |
|   1 |  TABLE ACCESS BY INDEX ROWID  | T    |      1 |   1846 |
|   2 |   BITMAP CONVERSION TO ROWIDS|      |      1 |   1846 |
|*  3 |    BITMAP INDEX FULL SCAN     | I_C2 |      1 |   1846 |
--------------------------------------------------------------

   3 - filter(("C2" LIKE '%A%' AND "C2" IS NOT NULL))
```

## Inequality and IS NOT NULL Conditions

As pointed out in Table 13-3, conditions based on either inequalities (`!=`, `<>`) or `IS NOT NULL` can't lead to index range scans. To illustrate this limitation and show you how to optimize SQL statements hitting it, let's have a look at an example based on the `inequalities.sql` script. The test table has a column named `status` that is characterized by a very non-uniform distribution. In fact, most of the rows have their status set to processed (P). Here's the example:

```
SQL> SELECT status, count(*)
  2  FROM t
  3  GROUP BY status;

S   COUNT(*)
- ----------
A          7
P     159981
R          4
X          8
```

An application has to select all rows with a status different than processed. For that purpose, it executes the following query:

```
SELECT * FROM t WHERE status != 'P'
```

Even though the query has a very strong selectivity and the `status` column is indexed, the query optimizer chooses a full table scan that leads to way too many logical reads (23,063) for reading 19 rows:

```
-------------------------------------------------------------
| Id  | Operation          | Name | Starts | A-Rows | Buffers |
-------------------------------------------------------------
|   0 | SELECT STATEMENT   |      |      1 |     19 |   23063 |
|*  1 |  TABLE ACCESS FULL| T    |      1 |     19 |   23063 |
-------------------------------------------------------------

   1 - filter("STATUS"<>'P')
```

In a case like this, where the inequality condition has a strong selectivity, it's nevertheless possible to take advantage of an index. There are three techniques you can apply.

First, if the inequality condition can be rewritten into an `IN` condition, it's possible to use one index range scan to apply the `IN` condition. This is an option only when the number of values to be selected is known and the number is limited. The following query is an example:

```
SELECT * FROM t WHERE status IN ('A','R','X')
```

```
----------------------------------------------------------------------------
| Id  | Operation                    | Name     | Starts | A-Rows | Buffers |
----------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |          |      1 |     19 |      13 |
|   1 |  INLIST ITERATOR             |          |      1 |     19 |      13 |
|   2 |   TABLE ACCESS BY INDEX ROWID| T        |      3 |     19 |      13 |
|*  3 |    INDEX RANGE SCAN          | I_STATUS |      3 |     19 |       7 |
----------------------------------------------------------------------------

   3 - access(("STATUS"='A' OR "STATUS"='R' OR "STATUS"='X'))
```

Second, if the previous technique can't be applied because the values are unknown or the number of values to be specified is too high, it's always possible to rewrite an inequality with two disjunct range predicates and, as a result, execute one index range scan for each of them. The idea is to take advantage of the or expansion query transformation (refer to Chapter 6 for information about it). The query would be rewritten like this:

```
SELECT * FROM t WHERE status < 'P' OR status > 'P'

--------------------------------------------------------------------------
| Id  | Operation                    | Name     | Starts | A-Rows | Buffers |
--------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |          |      1 |     19 |      12 |
|   1 |  CONCATENATION               |          |      1 |     19 |      12 |
|   2 |   TABLE ACCESS BY INDEX ROWID| T        |      1 |      7 |       5 |
|*  3 |    INDEX RANGE SCAN          | I_STATUS |      1 |      7 |       3 |
|   4 |   TABLE ACCESS BY INDEX ROWID| T        |      1 |     12 |       7 |
|*  5 |    INDEX RANGE SCAN          | I_STATUS |      1 |     12 |       3 |
--------------------------------------------------------------------------

   3 - access("STATUS"<'P')
   5 - access("STATUS">'P')
       filter(LNNVL("STATUS"<'P'))
```

In case or expansion doesn't kick in automatically, you can manually rewrite the query to make sure that both component queries can take advantage of an index range scan and, as a result, reduce the number of logical reads to a minimum:

```
SELECT * FROM t WHERE status < 'P'
UNION ALL
SELECT * FROM t WHERE status > 'P'

--------------------------------------------------------------------------
| Id  | Operation                    | Name     | Starts | A-Rows | Buffers |
--------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |          |      1 |     19 |      12 |
|   1 |  UNION-ALL                   |          |      1 |     19 |      12 |
|   2 |   TABLE ACCESS BY INDEX ROWID| T        |      1 |      7 |       5 |
|*  3 |    INDEX RANGE SCAN          | I_STATUS |      1 |      7 |       3 |
|   4 |   TABLE ACCESS BY INDEX ROWID| T        |      1 |     12 |       7 |
|*  5 |    INDEX RANGE SCAN          | I_STATUS |      1 |     12 |       3 |
--------------------------------------------------------------------------

   3 - access("STATUS"<'P')
   5 - access("STATUS">'P')
```

The third technique is based on an index full scan. To get it, you can simply force an index full scan with, for example, the `index` hint. From a performance point of view, as shown in the following example, it's not optimal, though. For a query with very strong selectivity, the ratio between the number of logical reads (299) and the number of returned rows (19) is too high:

```
SELECT /*+ index(t) */ * FROM t WHERE status != 'P'
```

```
----------------------------------------------------------------------------
| Id  | Operation                    | Name     | Starts | A-Rows | Buffers |
----------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |          |      1 |     19 |     299 |
|   1 |  TABLE ACCESS BY INDEX ROWID| T        |      1 |     19 |     299 |
|*  2 |   INDEX FULL SCAN            | I_STATUS |      1 |     19 |     293 |
----------------------------------------------------------------------------

   2 - filter("STATUS"<>'P')
```

To optimally execute an index full scan, the size of the index should be as small as possible. To achieve that, you can implement one of two tricks. The idea of the first one is to avoid indexing the popular value by defining a function-based index (later on, the "Function-based Indexes" section provides additional information about these indexes) that excludes it. To implement this trick, both the index and the SQL statement using the index have to be altered:

- Create the function-based index excluding the popular value (for more complex conditions you can also use the CASE expression or the decode function):

  ```
  CREATE INDEX i_status ON t (nullif(status, 'P'))
  ```

- Alter the predicate of the query to take advantage of the index:

  ```
  SELECT * FROM t WHERE nullif(status, 'P') IS NOT NULL
  ```

```
----------------------------------------------------------------------------
| Id  | Operation                    | Name     | Starts | A-Rows | Buffers |
----------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |          |      1 |     19 |       9 |
|   1 |  TABLE ACCESS BY INDEX ROWID| T        |      1 |     19 |       9 |
|*  2 |   INDEX FULL SCAN            | I_STATUS |      1 |     19 |       3 |
----------------------------------------------------------------------------

   2 - filter("T"."SYS_NC00004$" IS NOT NULL)
```

The idea of the second trick is to replace the most popular value with NULL and, thereby preventing most rows from being referenced in the index. But be careful—this technique works for B-tree indexes only. To implement it, carry out the following steps:

- Replace the most popular value with NULL:

  ```
  UPDATE t SET status = NULL WHERE status = 'P'
  ```

- Rebuild the index to shrink its size to the minimum:

  ```
  ALTER INDEX i_status REBUILD
  ```

- Replace the inequality with a IS NOT NULL:

  ```
  SELECT * FROM t WHERE status IS NOT NULL
  ```

```
-------------------------------------------------------------------------
| Id  | Operation                    | Name     | Starts | A-Rows | Buffers |
-------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |          |      1 |     19 |      10 |
|   1 |  TABLE ACCESS BY INDEX ROWID| T        |      1 |     19 |      10 |
|*  2 |   INDEX FULL SCAN            | I_STATUS |      1 |     19 |       3 |
-------------------------------------------------------------------------

   2 - filter("STATUS" IS NOT NULL)
```

In summary, as the previous examples show, it's possible to efficiently execute a SQL statement with inequalities or IS NOT NULL conditions. However, special care is required.

## Min/Max Functions

To execute queries containing the min or max functions efficiently, two specific operations are available with B-tree indexes. The first, INDEX FULL SCAN (MIN/MAX), is used when a query doesn't specify a range condition. In spite of its name, however, it performs no full index scan. It simply gets either the rightmost or the leftmost index key:

```
SELECT /*+ index(t t_pk) */ min(id) FROM t

-----------------------------------------------------------
| Id  | Operation                   | Name | Starts | A-Rows |
-----------------------------------------------------------
|   0 | SELECT STATEMENT            |      |      1 |      1 |
|   1 |  SORT AGGREGATE             |      |      1 |      1 |
|   2 |   INDEX FULL SCAN (MIN/MAX)| T_PK |      1 |      1 |
-----------------------------------------------------------
```

The second, INDEX RANGE SCAN (MIN/MAX), is used when the query specifies a condition on the same column used in the function:

```
SELECT /*+ index(t t_pk) */ min(id) FROM t WHERE id > 42

-------------------------------------------------------------
| Id  | Operation                     | Name | Starts | A-Rows |
-------------------------------------------------------------
|   0 | SELECT STATEMENT              |      |      1 |      1 |
|   1 |  SORT AGGREGATE               |      |      1 |      1 |
|   2 |   FIRST ROW                   |      |      1 |      1 |
|*  3 |    INDEX RANGE SCAN (MIN/MAX)| T_PK |      1 |      1 |
-------------------------------------------------------------

   3 - access("ID">42)
```

Unfortunately, this optimization technique can't be applied when both functions (min and max) are used in the same query. In this type of situation, an index full scan is performed. The following query is an example:

```
SELECT /*+ index(t t_pk) */ min(id), max(id) FROM t

Plan hash value: 56794325
```

```
-------------------------------------------------
| Id  | Operation        | Name | Starts | A-Rows |
-------------------------------------------------
|   0 | SELECT STATEMENT |      |      1 |      1 |
|   1 |  SORT AGGREGATE  |      |      1 |      1 |
|   2 |   INDEX FULL SCAN| T_PK |      1 |  10000 |
-------------------------------------------------
```

For bitmap indexes, no specific operation is available to execute the `min` and `max` functions. The same operations used for equality conditions and range conditions are used.

## Function-based Indexes

Every time an indexed column is passed as an argument to a function, or is involved in an expression, the SQL engine can't use the index built on that column for an index range scan. As a result, one of the fundamental rules to follow is to never modify the values returned by an indexed column in the `WHERE` clause. For example, if an index exists on the `c1` column, a restriction like `upper(c1) = 'SELDON'` can't be applied efficiently through the index built on the `c1` column. This should be pretty obvious, because you can search only for a value that is stored in an index, rather than something else. The following example, as the others in this section, is based on the `fbi.sql` script:

```
SQL> CREATE INDEX i_c1 ON t (c1);

SQL> SELECT * FROM t WHERE upper(c1) = 'SELDON';

--------------------------------------------------
| Id  | Operation         | Name | E-Rows | A-Rows |
--------------------------------------------------
|   0 | SELECT STATEMENT  |      |        |      4 |
|*  1 |  TABLE ACCESS FULL| T    |    100 |      4 |
--------------------------------------------------

   1 - filter(UPPER("C1")='SELDON')
```

An exception to the fundamental rule is when constraints ensure that an index contains the necessary information. In the case just illustrated, two constraints on the `c1` column provide that information to the query optimizer:

```
SQL> ALTER TABLE t MODIFY (c1 NOT NULL);

SQL> ALTER TABLE t ADD CONSTRAINT t_c1_upper CHECK (c1 = upper(c1));

SQL> SELECT * FROM t WHERE upper(c1) = 'SELDON';

------------------------------------------------------------
| Id  | Operation                   | Name | E-Rows | A-Rows |
------------------------------------------------------------
|   0 | SELECT STATEMENT            |      |        |      4 |
|   1 |  TABLE ACCESS BY INDEX ROWID| T    |    100 |      4 |
|*  2 |   INDEX RANGE SCAN          | I_C1 |      4 |      4 |
------------------------------------------------------------

   2 - access("C1"='SELDON')
       filter(UPPER("C1")='SELDON')
```

Obviously, you want to take advantage of an index if a restriction leads to strong selectivity. For that purpose, if it isn't possible to modify the `WHERE` clause or specify constraints, you can create a *function-based index*. Simply put, this is an index created on the return value of a function or the result of an expression. Here's an example:

```
SQL> CREATE INDEX i_c1_upper ON t (upper(c1));

SQL> SELECT * FROM t WHERE upper(c1) = 'SELDON';

-------------------------------------------------------------------
| Id  | Operation                   | Name       | E-Rows | A-Rows |
-------------------------------------------------------------------
|   0 | SELECT STATEMENT            |            |        |      4 |
|   1 |  TABLE ACCESS BY INDEX ROWID| T          |      4 |      4 |
|*  2 |   INDEX RANGE SCAN          | I_C1_UPPER |      4 |      4 |
-------------------------------------------------------------------

   2 - access(UPPER("C1")='SELDON')
```

■ **Caution** Function-based indexes based on functions that take literals as parameter aren't selected by the query optimizer when the `cursor_sharing` initialization parameter is set to either `force` or `similar`. That's because the literals are replaced by bind variables. The `fbi_cs.sql` script demostrates such a case.

Another problem related to the utilization of functions and expressions in `WHERE` clauses is that the query optimizer, as described in the "Extended Statistics" section in Chapter 8, misestimates the cardinality of the result set produced by the row source operations applying them. The examples in this section (notice the `E-Rows` and `A-Rows` columns in the execution plans) illustrate that with a function-based index in place, the query optimizer can also improve the estimations it makes. And that improvement can occur independently of whether a function-based index is used to access data. More accurate estimations are possible because each function-based index adds a hidden column to the table it's created on. Because column statistics and histograms are gathered for a hidden column as for any other column, the query optimizer gets additional information that wouldn't be available without a function-based index. It's also essential to point out that the object statistics at the table level for the new hidden column aren't gathered while the function-based index is built. Only index statistics are automatically gathered. Therefore, you shouldn't forget to gather the object statistics at the table level after creating a new function-based index.

Function-based indexes can also be created on user-defined functions written in PL/SQL. The only requirement is that the function must be defined as `DETERMINISTIC`.

■ **Caution** Function-based indexes based on a user-defined function aren't invalidated or made unusable when the PL/SQL code they depend on is changed. That, of course, may lead to wrong results. If you change the code of such a function, you should immediately rebuild the dependent index. An example of this behavior is available in the `fbi_udf.sql` script.

As of version 11.1, to avoid repeating a function or expression in the index as well as in several or possibly many SQL statements, it's possible to create a virtual column based on the function or expression. In this way, the index can be created directly on the virtual column, and the code can be made transparent to the definition. The following example shows how to add, index, and use a virtual column that applies the `upper` function to the `c1` column:

```
SQL> ALTER TABLE t ADD (c1_upper AS (upper(c1)));

SQL> CREATE INDEX i_c1_upper ON t (c1_upper);

SQL> SELECT * FROM t WHERE c1_upper = 'SELDON';

--------------------------------------------------------------------
| Id  | Operation                   | Name       | E-Rows | A-Rows |
--------------------------------------------------------------------
|   0 | SELECT STATEMENT            |            |        |      4 |
|   1 |  TABLE ACCESS BY INDEX ROWID| T          |      4 |      4 |
|*  2 |   INDEX RANGE SCAN          | I_C1_UPPER |      4 |      4 |
--------------------------------------------------------------------

   2 - access("C1_UPPER"='SELDON')
```

Even though the examples in this section are based on B-tree indexes, function-based bitmap indexes are supported as well.

## Linguistic Indexes

Per default, the database engine performs *binary comparisons* for the purpose of comparing character strings. With them, characters are compared according to their binary value. Consequently, two character strings are considered equal only when the numeric code of each corresponding character is identical.

The database engine is also able to perform *linguistic comparisons*. With these comparisons, the numeric code of each character doesn't have to be identical in order to match. For example, it's possible to instruct the database engine to consider equal lowercase and uppercase characters, or characters with and without accents. To manage this behavior of the SQL operators, the `nls_comp` initialization parameter is available. It can be set to one of the following values:

- `binary`: Binary comparisons are used. This is the default.
- `linguistic`: Linguistic comparisons are used. The `nls_sort` initialization parameter specifies the linguistic sort sequence (and therefore the rules) that applies to the comparisons. The accepted values for a specific version can be displayed with the following query:

  ```
  SELECT value FROM v$nls_valid_values WHERE parameter = 'SORT'
  ```
- `ansi`: This value is available for backward compatibility only. `linguistic` should be used instead.

The dynamic `nls_comp` and `nls_sort` initialization parameters can be set at the instance and session levels. At the session level, they can be set with the `ALTER SESSION` statement, as well as be defined on the client side at the operating system level (for example, in Microsoft Windows with an entry in the registry). Be aware that having client-side settings is the rule, not the exception. Therefore it's quite common to see server-side settings that are overridden by client-side settings.

As an example, consider a table storing the following data (the table and the test queries are available in the `linguistic_index.sql` script):

```
SQL> SELECT c1 FROM t;
```

```
C1
----------
Leon
Léon
LEON
LÉON
```

Per default, binary comparisons are performed. To use linguistic comparisons, it's necessary to set the `nls_comp` initialization parameter to `linguistic`, and the linguistic sort sequence (and therefore the rules) used for the comparisons must be specified through the `nls_sort` initialization parameter. The following example uses `generic_m`, an ISO standard for Latin-based characters:

```
SQL> ALTER SESSION SET nls_comp = linguistic;

SQL> ALTER SESSION SET nls_sort = generic_m;

SQL> SELECT c1 FROM t WHERE c1 = 'LEON';

C1
----------
LEON
```

As expected, nothing particular happens with the previous setting. The useful feature is provided by two extensions of `generic_m`. The first is `generic_m_ci`. With it, as shown in the following query, the comparisons are case insensitive:

```
SQL> ALTER SESSION SET nls_sort = generic_m_ci;

SQL> SELECT c1 FROM t WHERE c1 = 'LEON';

C1
----------
Leon
LEON
```

The second is `generic_m_ai`. With it, as shown in the following query, the comparisons are case and accent insensitive:

```
SQL> ALTER SESSION SET nls_sort = generic_m_ai;

SQL> SELECT c1 FROM t WHERE c1 = 'LEON';

C1
----------
Leon
Léon
LEON
LÉON
```

From a functional point of view, this is excellent. By setting two initialization parameters, you're able to control the behavior of the SQL operators. Let's check whether the execution plan changes when the `nls_comp` initialization parameter is set to `linguistic`:

```
SQL> CREATE INDEX i_c1 ON t (c1);

SQL> ALTER SESSION SET nls_sort = generic_m_ai;

SQL> ALTER SESSION SET nls_comp = binary;

SQL> SELECT /*+ index(t) */ * FROM t WHERE c1 = 'LEON';

--------------------------------------------
| Id  | Operation                    | Name |
--------------------------------------------
|   0 | SELECT STATEMENT             |      |
|   1 |  TABLE ACCESS BY INDEX ROWID| T    |
|*  2 |   INDEX RANGE SCAN           | I_C1 |
--------------------------------------------

   2 - access("C1"='LEON')

SQL> ALTER SESSION SET nls_comp = linguistic;

SQL> SELECT /*+ index(t) */ * FROM t WHERE c1 = 'LEON';

----------------------------------
| Id  | Operation          | Name |
----------------------------------
|   0 | SELECT STATEMENT   |      |
|*  1 |  TABLE ACCESS FULL| T    |
----------------------------------

   1 - filter(NLSSORT("C1",'nls_sort=''GENERIC_M_AI''')=HEXTORAW('022601FE02380232') )
```

Obviously, when the `nls_comp` initialization parameter is set to `linguistic`, the index is no longer used. The reason is indicated by the last line of the output. No lookup in the index is possible because a function, `nlssort`, is silently applied to the indexed column `c1`. Hence, for that purpose, a function-based index is necessary to avoid a full table scan. It's essential to recognize that the definition of the index must contain the same value as the `nls_sort` initialization parameter. Therefore, if several languages are in use, several indexes ought to be created:

```
SQL> CREATE INDEX i_c1_linguistic ON t (nlssort(c1,'nls_sort=generic_m_ai'));

SQL> SELECT /*+ index(t) */ * FROM t WHERE c1 = 'LEON';

-------------------------------------------------------
| Id  | Operation                    | Name            |
-------------------------------------------------------
|   0 | SELECT STATEMENT             |                 |
|   1 |  TABLE ACCESS BY INDEX ROWID| T               |
|*  2 |   INDEX RANGE SCAN           | I_C1_LINGUISTIC |
-------------------------------------------------------

   2 - access(NLSSORT("C1",'nls_sort=''GENERIC_M_AI''')=HEXTORAW('022601FE02380232') )
```

In version 10.2, another limitation is that in order to apply a LIKE operator, the database engine can't take advantage of linguistic indexes. In other words, a full index scan or full table scan can't be avoided. This limitation is no longer present from version 11.1 onward.

Even though the examples in this section are based on B-tree indexes, linguistic bitmap indexes are supported as well.

The following example shows that linguistic indexes can also be used to avoid ORDER BY operations:

```
SQL> SELECT /*+ index(t) */ * FROM t WHERE c1 BETWEEN 'L' AND 'M' ORDER BY c1;

-------------------------------------------------------
| Id  | Operation                   | Name            |
-------------------------------------------------------
|   0 | SELECT STATEMENT            |                 |
|   1 |  TABLE ACCESS BY INDEX ROWID| T               |
|*  2 |   INDEX RANGE SCAN          | I_C1_LINGUISTIC |
-------------------------------------------------------

   2 - access(NLSSORT("C1",'nls_sort=''GENERIC_M_AI''')>=HEXTORAW('0226')  AND NLSSORT(
              "C1",'nls_sort=''GENERIC_M_AI''')<=HEXTORAW('0230') )
```

In summary, linguistic comparison is a powerful feature that's transparent for SQL statements. However, the database engine can apply them efficiently only when a set of adapted indexes is available. Because the setting at the client level might impact the utilization of the indexes, it's essential to plan its use carefully.

## Composite Indexes

So far, with one exception, I've discussed only indexes that have an index key consisting of a single column. Index keys, however, can consist of many columns (the limit is 32 for B-tree indexes and 30 for bitmap indexes). Indexes with multiple columns are called *composite indexes* (sometimes the terms *concatenated indexes* or *multicolumn indexes* are used). In this regard, B-tree indexes and bitmap indexes have completely different behaviors, so I discuss them separately. Note that all examples in this section are based on the composite_index.sql script.

### B-tree Indexes

The purpose of composite indexes is twofold. First, they can be used to implement a primary key or unique key constraint composed of several columns. Second, they can be used to apply a predicate composed of several SQL conditions combined with AND. Be careful, because when several SQL conditions are combined with OR, composite indexes can't be used efficiently!

Naturally, it's important to discuss how to use composite indexes for applying restrictions. The following query is used for this:

```
SELECT * FROM t WHERE n1 = 6 AND n2 = 42 AND n3 = 11
```

Let's begin by looking at what happens when a single column index is used. With an index built on the n1 column, 527 rowids are returned from the index scan. Because the index stores only the data related to the n1 column, only the n1 = 6 predicate can be applied through the index by operation 2. The other two predicates are applied as a filter by operation 1. Because of the many rowids returned by operation 2, the execution generates 327 logical reads in total. This is unacceptable when retrieving a single row:

```
-------------------------------------------------------------------------
| Id  | Operation                    | Name | Starts | A-Rows | Buffers |
-------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |      |      1 |      1 |     327 |
|*  1 |  TABLE ACCESS BY INDEX ROWID| T    |      1 |      1 |     327 |
|*  2 |   INDEX RANGE SCAN          | I_N1 |      1 |    527 |       4 |
-------------------------------------------------------------------------

   1 - filter(("N2"=42 AND "N3"=11))
   2 - access("N1"=6)
```

With an index built on the n2 column, the situation is basically the same as in the previous example. The only improvement is that fewer rowids (89) are returned by the index scan. Therefore, far fewer logical reads (85) are performed in total:

```
-------------------------------------------------------------------------
| Id  | Operation                    | Name | Starts | A-Rows | Buffers |
-------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |      |      1 |      1 |      85 |
|*  1 |  TABLE ACCESS BY INDEX ROWID| T    |      1 |      1 |      85 |
|*  2 |   INDEX RANGE SCAN          | I_N2 |      1 |     89 |       4 |
-------------------------------------------------------------------------

   1 - filter(("N3"=11 AND "N1"=6))
   2 - access("N2"=42)
```

With an index built on the n3 column, the situation is still similar to the previous ones. In fact, the index scan returns plenty of rowids (164). The total number of logical reads (141) is still too high:

```
-------------------------------------------------------------------------
| Id  | Operation                    | Name | Starts | A-Rows | Buffers |
-------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |      |      1 |      1 |     141 |
|*  1 |  TABLE ACCESS BY INDEX ROWID| T    |      1 |      1 |     141 |
|*  2 |   INDEX RANGE SCAN          | I_N3 |      1 |    164 |       4 |
-------------------------------------------------------------------------

   1 - filter(("N2"=42 AND "N1"=6))
   2 - access("N3"=11)
```

In summary, none of the three indexes can apply the predicates efficiently. The selectivities of the three restrictions taken one by one are too high. This observation is in line with the object statistics stored in the data dictionary. In fact, the number of distinct values for each column is low, as demonstrated by the following query:

```
SQL> SELECT column_name, num_distinct
  2  FROM user_tab_columns
  3  WHERE table_name = 'T' AND column_name IN ('ID', 'N1', 'N2', 'N3');
```

```
COLUMN_NAME NUM_DISTINCT
----------- ------------
ID                 10000
N1                    18
N2                   112
N3                    60
```

In such situations, it's more efficient to apply the various conditions with a single index built on several columns. For example, the following execution plan shows what happens if a composite index is created on the three columns. It's essential to understand that with this index, the number of logical reads (4) is lower because the index scan returns only rows that fulfill the whole WHERE clause (in this case, one row):

```
-------------------------------------------------------------------------
| Id  | Operation                    | Name   | Starts | A-Rows | Buffers |
-------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |        |      1 |      1 |       4 |
|   1 |  TABLE ACCESS BY INDEX ROWID| T      |      1 |      1 |       4 |
|*  2 |   INDEX RANGE SCAN          | I_N123 |      1 |      1 |       3 |
-------------------------------------------------------------------------

   2 - access("N1"=6 AND "N2"=42 AND "N3"=11)
```

At this point, it's crucial to recognize that the database engine is able to carry out an index range scan even when not all columns on which the index is built are referenced in the WHERE clause. The basic requirement is that a condition should be applied to the leading column of the index key. For example, with the i_n123 index used in the previous example, the conditions on the columns n2 and n3 are optional. The following query shows an example where no condition on the n2 column is present:

```
SELECT * FROM t WHERE n1 = 6 AND n3 = 11

-------------------------------------------------------------------------
| Id  | Operation                    | Name   | Starts | A-Rows | Buffers |
-------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |        |      1 |      8 |      12 |
|   1 |  TABLE ACCESS BY INDEX ROWID| T      |      1 |      8 |      12 |
|*  2 |   INDEX RANGE SCAN          | I_N123 |      1 |      8 |       4 |
-------------------------------------------------------------------------

   2 - access("N1"=6 AND "N3"=11)
       filter("N3"=11)
```

Be aware that in the previous execution plan, all index keys fulfilling the n1 = 6 predicate must be accessed, even though the n3 = 11 predicate is mentioned in the access predicate. On the one hand, that approach is suboptimal because unnecessary parts of the index are accessed. On the other hand, it's much better to apply the n3 = 11 predicate as a filter during the index scan than when the table is accessed, as in the example provided before. In any case, for best performance, the predicates should be applied to the leading columns of the index.

There are cases where the index can even be used (efficiently) when there's no condition on the leading column of the index key. Such an operation is called an *index skip scan*. However, using it makes sense only when the leading column has a very low number of distinct values, because an independent index range scan is performed for each value of the leading column. The following query shows such an example. Notice the `index_ss` hint and the `INDEX SKIP SCAN` operation:

```
SELECT /*+ index_ss(t i_n123) */ * FROM t WHERE n2 = 42 AND n3 = 11

-------------------------------------------------------------------------
| Id  | Operation                    | Name   | Starts | A-Rows | Buffers |
-------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |        |      1 |      2 |      33 |
|   1 |  TABLE ACCESS BY INDEX ROWID| T      |      1 |      2 |      33 |
|*  2 |   INDEX SKIP SCAN            | I_N123 |      1 |      2 |      31 |
-------------------------------------------------------------------------

   2 - access("N2"=42 AND "N3"=11)
       filter(("N2"=42 AND "N3"=11))
```

Because descending index skip scans are supported for “regular” index scans (using the `INDEX SKIP SCAN DESCENDING` operation), you can use the two hints `index_ss_asc` and `index_ss_desc` to control the order of the scan.

Speaking of composite indexes, I believe it’s necessary to mention the most common mistake I come across when dealing with them, as well as the most frequently asked question. The mistake is related to overindexation. The misconception is that the database engine is able to take advantage of an index only when all columns comprising the index key are used in the `WHERE` clause. As you’ve just seen in several examples, that’s not the case. This misconception usually leads to the creation of several indexes on the same table, with the same leading column—for instance, one index with the `n1`, `n2`, and `n3` columns and another with the `n1` and `n3` columns. The second is generally superfluous. Note that superfluous indexes are a problem because not only do they slow down SQL statements that modify the indexed data, but also because they waste space unnecessarily.

The most frequently asked question is: how do I choose the order of the columns? For example, if an index key is composed of the `n1`, `n2`, and `n3` columns, what is the best order? When all indexed columns are present in a `WHERE` clause, the efficiency of the index is independent of the order of the columns in the index. Therefore, the best order is the one that maximizes the chances of using the index as frequently as possible when not all indexed columns are present in `WHERE` clauses. In other words, it should be possible to use an index for the greatest number of SQL statements. To make sure this is the case, the columns should be ordered according to their frequency of utilization. Especially the leading column should be the one that’s more frequently (ideally speaking, of course) specified in `WHERE` clauses. Whenever several columns are used with equal frequency, there are two opposing approaches you can follow:

- The leading column should be the one that’s expected to provide the best selectivity. If only equalities are used, this is the column with the highest number of distinct values. If range conditions are used, the number of distinct values itself might not be relevant. For example, think about the case of a timestamp: it’s likely that the number of distinct values is high. But because a timestamp is frequently used in range predicates, what matters is the actual selectivity and not the number of distinct values. Having a leading column that can provide a strong selectivity is useful if a restriction is applied only on that particular column in future SQL statements. In other words, you maximize the chances that the index could be selected by the query optimizer.
- The leading column should be the one with the lowest number of distinct values. This could be useful to achieve a better compression ratio for the index.

## INDEX COMPRESSION

One important difference between B-tree and bitmap indexes is the compression used to store the keys in the index leaf blocks. Whereas bitmap indexes are always compressed, B-tree indexes are compressed only when requested.

In a noncompressed B-tree index, every key is fully stored. In other words, if several keys have the same value, the value is repeatedly stored for each key. Consequently, in nonunique indexes, it's common to have the same value stored several times in the same leaf block. To eliminate these repeated occurrences, you can compress the index keys by (re)building the index with the `COMPRESS` parameter and, optionally, the number of columns that need to be compressed. For example, the `i_n123` index is composed of three columns: `n1`, `n2`, and `n3`. With `COMPRESS 1`, you specify to compress only the `n1` column; with `COMPRESS 2`, you specify to compress the `n1` and `n2` columns; and with `COMPRESS 3`, you specify to compress all three columns. When the number of columns to be compressed isn't specified, for nonunique indexes all columns are compressed, and for unique indexes the number of columns minus one are compressed.

Because columns are compressed from left to right, columns should be ordered by decreasing selectivity to achieve the best compression. However, you should reorder the columns of an index only when this doesn't prevent the query optimizer from using the index.

B-tree index compression isn't activated per default because it doesn't always reduce the size of indexes. Actually, in some situations, the index might become larger with compression! Because of this, you should enable compression only if there's a real advantage to doing so. You have two options for checking the expected compression ratio for a given index. First, you can build the index once without compression and then again with compression, and then compare the size. Second, you can let the `ANALYZE INDEX` statement perform an analysis to find out the optimal number of columns to compress and how much space can be saved with the optimal compression. The following example shows such an analysis for the `i_n123` index. Note that the output of the analysis is written in the `index_stats` table. In this case, you're informed that by compressing two columns you can save 17 percent of the space currently occupied by the index:

```
SQL> ANALYZE INDEX i_n123 VALIDATE STRUCTURE;

SQL> SELECT opt_cmpr_count, opt_cmpr_pctsave FROM index_stats;

OPT_CMPR_COUNT OPT_CMPR_PCTSAVE
-------------- ----------------
             2               17
```

The following SQL statements show not only how to implement the compression of the `i_n123` index but also how to check the result of the compression:

```
SQL> SELECT blocks FROM index_stats;

    BLOCKS
----------
        40

SQL> ALTER INDEX i_n123 REBUILD COMPRESS 2;

SQL> ANALYZE INDEX i_n123 VALIDATE STRUCTURE;
SQL> SELECT blocks FROM index_stats;

    BLOCKS
----------
        32
```

From a performance point of view, the key advantage of compressed indexes is that because of their smaller size, not only are fewer logical reads needed to perform index range scans and index full scans, but, in addition, their blocks are more likely to be found in the buffer cache. The disadvantage, however, is the increasing likelihood of suffering from block contention (this topic is covered in Chapter 16).

## Bitmap Indexes

Composite bitmap indexes are rarely created. This is because several indexes can be combined efficiently in order to apply a restriction. To see how powerful bitmap indexes are, let's look at several queries.

The first query takes advantage of three bitmap indexes that are combined with AND in order to apply three equality conditions. Note that the index_combine hint forces this type of execution plan. First, operation 4 scans the index based on the n5 column by looking for the rows that fulfill the restriction on that column. The resulting bitmaps are passed to operation 3. Then operations 5 and 6 perform the same scan on the indexes created on the n6 and n4 columns, respectively. Once the three index scans are completed, operation 3 computes the AND of the three sets of bitmaps. Finally, operation 2 converts the resulting bitmap into a list of rowids, and then they're used by operation 1 to access the table:

```
SELECT /*+ index_combine(t i_n4 i_n5 i_n6) */ *
FROM t
WHERE n4 = 6 AND n5 = 42 AND n6 = 11

-------------------------------------------------------------------------
| Id  | Operation                     | Name | Starts | A-Rows | Buffers |
-------------------------------------------------------------------------
|   0 | SELECT STATEMENT              |      |      1 |      1 |       7 |
|   1 |  TABLE ACCESS BY INDEX ROWID  | T    |      1 |      1 |       7 |
|   2 |   BITMAP CONVERSION TO ROWIDS|      |      1 |      1 |       6 |
|   3 |    BITMAP AND                 |      |      1 |      1 |       6 |
|*  4 |     BITMAP INDEX SINGLE VALUE| I_N5 |      1 |      1 |       2 |
|*  5 |     BITMAP INDEX SINGLE VALUE| I_N6 |      1 |      1 |       2 |
|*  6 |     BITMAP INDEX SINGLE VALUE| I_N4 |      1 |      1 |       2 |
-------------------------------------------------------------------------

   4 - access("N5"=42)
   5 - access("N6"=11)
   6 - access("N4"=6)
```

For this first query, it's worthwhile to also show you the execution plan with a composite bitmap index. As you can see, the number of logical reads isn't much lower (4 instead of 7). It's better, but such a composite index is far less flexible that the three single-column indexes. This is the reason why in practice composite bitmap indexes are rarely created:

```
----------------------------------------------------------------------------
| Id  | Operation                     | Name   | Starts | A-Rows | Buffers |
----------------------------------------------------------------------------
|   0 | SELECT STATEMENT              |        |      1 |      1 |       4 |
|   1 |  TABLE ACCESS BY INDEX ROWID  | T      |      1 |      1 |       4 |
|   2 |   BITMAP CONVERSION TO ROWIDS|        |      1 |      1 |       3 |
|*  3 |    BITMAP INDEX SINGLE VALUE  | I_N456 |      1 |      1 |       3 |
----------------------------------------------------------------------------

   3 - access("N4"=6 AND "N5"=42 AND "N6"=11)
```

The second query is very similar to the first one. The only difference is because of the `OR` instead of the `AND`. Notice how only operation 3 has changed in the execution plan:

```
SELECT /*+ index_combine(t i_n4 i_n5 i_n6) */ *
FROM t
WHERE n4 = 6 OR n5 = 42 OR n6 = 11
```

```
--------------------------------------------------------------------------
| Id  | Operation                     | Name | Starts | A-Rows | Buffers |
--------------------------------------------------------------------------
|   0 | SELECT STATEMENT              |      |      1 |    767 |     420 |
|   1 |  TABLE ACCESS BY INDEX ROWID  | T    |      1 |    767 |     420 |
|   2 |   BITMAP CONVERSION TO ROWIDS|      |      1 |    767 |       7 |
|   3 |    BITMAP OR                  |      |      1 |      1 |       7 |
|*  4 |     BITMAP INDEX SINGLE VALUE| I_N4 |      1 |      1 |       3 |
|*  5 |     BITMAP INDEX SINGLE VALUE| I_N6 |      1 |      1 |       2 |
|*  6 |     BITMAP INDEX SINGLE VALUE| I_N5 |      1 |      1 |       2 |
--------------------------------------------------------------------------

   4 - access("N4"=6)
   5 - access("N6"=11)
   6 - access("N5"=42)
```

The third query is similar to the first one. This time, the only difference is the `n4 != 6` condition (instead of `n4 = 6`). Because the execution plan is quite different, let's look at it in detail. Initially, operation 6 scans the index based on the `n5` column by looking for the rows fulfilling the `n5 = 42` condition on that column. The resulting bitmaps are passed to operation 5. Then, operation 7 performs the same scan on the index created on the `n6` column for the `n6 = 11` condition. Once the two index scans are completed, operation 5 computes the `AND` of the two sets of bitmaps and passes the resulting bitmaps to operation 4. Next, operation 8 scans the index based on the `n4` column by looking for rows fulfilling the `n4 = 6` condition (which is the opposite of what is specified in the `WHERE` clause). The resulting bitmaps are passed to operation 4, which subtracts them from the bitmaps delivered by operation 5. Then, operations 9 and 3 perform the same scan for the `n4 IS NULL` condition. This is necessary because `NULL` values don't fulfill the `n4 != 6` condition. Finally, operation 2 converts the resulting bitmap into a list of rowids, which are then used by operation 1 to access the table:

```
SELECT /*+ index_combine(t i_n4 i_n5 i_n6) */ *
FROM t
WHERE n4 != 6 AND n5 = 42 AND n6 = 11

---------------------------------------------------------------------------
| Id  | Operation                         | Name | Starts | A-Rows | Buffers |
---------------------------------------------------------------------------
|   0 | SELECT STATEMENT                  |      |      1 |      1 |       9 |
|   1 |  TABLE ACCESS BY INDEX ROWID      | T    |      1 |      1 |       9 |
|   2 |   BITMAP CONVERSION TO ROWIDS     |      |      1 |      1 |       8 |
|   3 |    BITMAP MINUS                   |      |      1 |      1 |       8 |
|   4 |     BITMAP MINUS                  |      |      1 |      1 |       6 |
|   5 |      BITMAP AND                   |      |      1 |      1 |       4 |
|*  6 |       BITMAP INDEX SINGLE VALUE| I_N5 |      1 |      1 |       2 |
|*  7 |       BITMAP INDEX SINGLE VALUE| I_N6 |      1 |      1 |       2 |
|*  8 |      BITMAP INDEX SINGLE VALUE | I_N4 |      1 |      1 |       2 |
|*  9 |     BITMAP INDEX SINGLE VALUE  | I_N4 |      1 |      1 |       2 |
---------------------------------------------------------------------------

   6 - access("N5"=42)
   7 - access("N6"=11)
   8 - access("N4"=6)
   9 - access("N4" IS NULL)
```

In summary, bitmap indexes can be combined efficiently and have several SQL conditions applied during combinations. In a few words, they're very flexible. Because of these characteristics, they're essential for reporting systems where the queries aren't known (fixed) in advance.

## Bitmap Plans for B-tree Indexes

The bitmap plans described in the previous section perform so well, they can also be applied to B-tree indexes. The idea is that the database engine is able to build a kind of in-memory bitmap index based on the data returned by B-tree index scans. The following query, which is the same as the one used in the part about composite B-tree indexes, is an example. Note that the `BITMAP CONVERSION FROM ROWIDS` operations are responsible for the conversion in the execution plan:

```
SELECT /*+ index_combine(t i_n1 i_n2 i_n3) */ *
FROM t
WHERE n1 = 6 AND n2 = 42 AND n3 = 11

-----------------------------------------------------------------------------
| Id  | Operation                           | Name | Starts | A-Rows | Buffers |
-----------------------------------------------------------------------------
|   0 | SELECT STATEMENT                    |      |      1 |      1 |      10 |
|   1 |  TABLE ACCESS BY INDEX ROWID        | T    |      1 |      1 |      10 |
|   2 |   BITMAP CONVERSION TO ROWIDS       |      |      1 |      1 |       9 |
|   3 |    BITMAP AND                       |      |      1 |      1 |       9 |
|   4 |     BITMAP CONVERSION FROM ROWIDS|      |      1 |      1 |       3 |
|*  5 |      INDEX RANGE SCAN            | I_N2 |      1 |     89 |       3 |
|   6 |     BITMAP CONVERSION FROM ROWIDS|      |      1 |      1 |       3 |
|*  7 |      INDEX RANGE SCAN            | I_N3 |      1 |    164 |       3 |
|   8 |     BITMAP CONVERSION FROM ROWIDS|      |      1 |      1 |       3 |
|*  9 |      INDEX RANGE SCAN            | I_N1 |      1 |    527 |       3 |
-----------------------------------------------------------------------------

   5 - access("N2"=42)
   7 - access("N3"=11)
   9 - access("N1"=6)
```

■ **Note** Bitmap plans for B-tree indexes as well as bitmap indexes are available only in Enterprise Edition.

## Index-only Scans

A useful optimization technique related to indexes is that the database engine can not only extract lists of rowids from indexes to access tables, it can get column data stored in the indexes. Therefore, when an index contains all the data needed to process a query, an *index-only scan* can be executed. This is useful for reducing the number of logical reads. In fact, an index-only scan doesn't access the table. This could be especially useful for index range scans if the clustering factor of the index is high. The following query illustrates this. Notice that no table access is performed:

```
SELECT c1 FROM t WHERE c1 LIKE 'A%'

-------------------------------------------------------------
| Id  | Operation        | Name | Starts | A-Rows | Buffers |
-------------------------------------------------------------
|   0 | SELECT STATEMENT |      |      1 |    119 |      11 |
|*  1 |  INDEX RANGE SCAN| I_C1 |      1 |    119 |      11 |
-------------------------------------------------------------

   1 - access("C1" LIKE 'A%')
       filter("C1" LIKE 'A%')
```

If the SELECT clause references the n1 column instead of c1, the query optimizer isn't able to take advantage of the index-only scan. Notice, in the following example, how the query performed 130 logical reads (11 against the index and 119 against the table—in other words, one for each rowid get from the index) to retrieve 119 rows:

```
SELECT n1 FROM t WHERE c1 LIKE 'A%'

-------------------------------------------------------------------------
| Id  | Operation                    | Name | Starts | A-Rows | Buffers |
-------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |      |      1 |    119 |     130 |
|   1 |  TABLE ACCESS BY INDEX ROWID| T    |      1 |    119 |     130 |
|*  2 |   INDEX RANGE SCAN           | I_C1 |      1 |    119 |      11 |
-------------------------------------------------------------------------

   2 - access("C1" LIKE 'A%')
       filter("C1" LIKE 'A%')
```

In this type of situation, in order to take advantage of index-only scans, you might add columns to an index even if they aren't used to apply a restriction. The idea is to create a composite index with an index key that's composed of all columns that are referenced in the SQL statement (also known as *covering index*), not only those in the WHERE clause. In other words, you "misuse" the index to store redundant data and, therefore, minimize the number of logical reads. Note, however, that the leading column of the index must be one of the columns referenced in the WHERE clause. In this specific case, this means that a composite index on the c1 and n1 columns is created. With that index in place, the very same query retrieves the same rows with only 10 logical reads instead of 130:

```
SELECT n1 FROM t WHERE c1 LIKE 'A%'

---------------------------------------------------------------
| Id  | Operation        | Name   | Starts | A-Rows | Buffers |
---------------------------------------------------------------
|   0 | SELECT STATEMENT |        |      1 |    119 |      10 |
|*  1 |  INDEX RANGE SCAN| I_C1N1 |      1 |    119 |      10 |
---------------------------------------------------------------

   1 - access("C1" LIKE 'A%')
       filter("C1" LIKE 'A%')
```

---

■ **Caution** For list partitioned tables, the query optimizer generates an execution plan based on an index-only scan to resolve an IN condition only when the partition key is part of the index. The index_only_scan_list_part.sql script provides an example. For range and hash partitioned tables, this restriction doesn't exist.

---

Even though the examples in this section are based on B-tree indexes, index-only scans are available for bitmap indexes as well.

## Index-organized Tables

One particular way to achieve an index-only scan is to create an index-organized table. The central idea of this kind of table is, in fact, to avoid having a table segment at all. Instead, all data is stored in an index segment based on the primary key. It's also possible to store part of the data in an overflow segment. By doing so, however, the benefit of using an index-organized table vanishes (unless the overflow segment is rarely accessed). The same happens when a *secondary index* (that is, another index in addition to the primary key) is created: two segments need to be accessed. Hence, there's no benefit in using it. For these reasons, you should consider using index-organized tables only when two requirements are met. First, the table is normally accessed through the primary key. Second, all data can be stored in the index structure (a row can take at most 50 percent of a block). In all other cases, it makes little sense to use them.

A row in an index-organized table isn't referenced by a *physical rowid*. Instead, it's referenced by a *logical rowid*. This kind of rowid is composed of two parts: first, a guess referencing the block that contains the row (key) at the time it was inserted, and second, the value of the primary key. A visit to the index-organized table by logical rowid at first follows the guess, hoping to find the row still in the insert-time block, but because the guess isn't updated when block splits occurs, it might become stale when INSERT and UPDATE statements are executed. If the guess is correct, with a logical rowid, it's possible to access one row with a single logical read. In case the guess is wrong, the number of logical reads would be equal to or greater than two (one for the useless access through the guess, plus the regular access with the primary key). Naturally, to have the best performance, it's capital to have correct guesses. To assess the correctness of such guesses, the pct_direct_access column, which is updated by the dbms_stats package, is available in the user_indexes view (the dba, all and, in a 12.1 multitenant environment, cdb versions of this view also have the pct_direct_access column). The value provides the percentage of correct guess for a specific index. The following example, which is an excerpt of the iot_guess.sql script, shows not only the impact of stale guesses on the number of logical reads but also how to rectify this type of suboptimal situation (note that the index used in the example is a secondary index):

```
SQL> SELECT pct_direct_access
  2  FROM user_indexes
  3  WHERE table_name = 'T' AND index_name = 'I';

PCT_DIRECT_ACCESS
-----------------
               76

SQL> SELECT count(pad) FROM t WHERE n > 0;

---------------------------------------------------------------
| Id  | Operation          | Name | Starts | A-Rows | Buffers |
---------------------------------------------------------------
|   0 | SELECT STATEMENT   |      |      1 |      1 |    1496 |
|   1 |  SORT AGGREGATE    |      |      1 |      1 |    1496 |
|*  2 |   INDEX UNIQUE SCAN| T_PK |      1 |   1000 |    1496 |
|*  3 |    INDEX RANGE SCAN| I    |      1 |   1000 |       6 |
---------------------------------------------------------------

   2 - access("N">0)
   3 - access("N">0)

SQL> ALTER INDEX i UPDATE BLOCK REFERENCES;

SQL> execute dbms_stats.gather_index_stats(ownname => user, indname => 'i')

SQL> SELECT pct_direct_access
  2  FROM user_indexes
  3  WHERE table_name = 'T' AND index_name = 'I';

PCT_DIRECT_ACCESS
-----------------
              100

SQL> SELECT count(pad) FROM t WHERE n > 0;

---------------------------------------------------------------
| Id  | Operation          | Name | Starts | A-Rows | Buffers |
---------------------------------------------------------------
|   0 | SELECT STATEMENT   |      |      1 |      1 |    1006 |
|   1 |  SORT AGGREGATE    |      |      1 |      1 |    1006 |
|*  2 |   INDEX UNIQUE SCAN| T_PK |      1 |   1000 |    1006 |
|*  3 |    INDEX RANGE SCAN| I    |      1 |   1000 |       6 |
---------------------------------------------------------------

   2 - access("N">0)
   3 - access("N">0)
```

An interesting side-effect of logical rowids is that secondary indexes always contain the primary key, also if it's not explicitly indexed. The following example illustrates how the database engine is able to extract the primary key (`id`) from a secondary index created only on another column (`n`) by performing an index-only scan:

```
SQL> SELECT id FROM t WHERE n = 42;

---------------------------------
| Id  | Operation        | Name |
---------------------------------
|   0 | SELECT STATEMENT |      |
|*  1 |  INDEX RANGE SCAN| I    |
---------------------------------

   1 - access("N"=42)
```

In addition to avoiding accessing a table segment, index-organized tables provide two more advantages that shouldn't be underestimated. The first is that data is always clustered, and therefore, range scans based on the primary key can always be performed efficiently, and not only when the clustering factor is low like with heap tables. The second advantage is that range scans based on the primary key always return the data in the order in which the data is stored in the primary key index. This could be useful for optimizing `ORDER BY` operations.

## Global, Local, or Nonpartitioned Indexes?

With partitioned tables, it's common to create local partitioned indexes. The main advantage of doing so is to reduce the dependencies between indexes and table partitions. For example, it makes things much easier when partitions are added, dropped, truncated, or exchanged. Simply put, creating local indexes is generally good. Nevertheless, there are situations where it's not possible or not advisable to do so.

### PREFIXED VS. NONPREFIXED INDEXES

An index is prefixed if the partition key is the left prefix of the index columns and, for subpartitioned indexes, the subpartioning key is included in the index key. While local indexes can be prefixed or nonprefixed, only global prefixed indexes can be created.

According to the *Oracle Database VLDB and Partitioning Guide* manual, nonprefixed indexes don't perform as well as prefixed indexes. In practice, I've never seen a performance problem related to the fact that an index was nonprefixed. My advice is therefore to create the most sensible index without being concerned about whether it's prefixed or nonprefixed.

The first problem is related to primary keys and unique indexes. In fact, to be based on local indexes, their keys must contain the partition key. Although this is sometimes possible, more often than not there's no such possibility without distorting the logical database design. This is especially true when range partitioning is used. So, in my opinion, this should be considered only as a last resort. You should never mess up the logical design. Because the logical design can't be changed, only two other possibilities remain. The first is to create a non-partitioned index. The second is to create a global partitioned index. The latter should be implemented only if there's a real advantage in doing so. Because such indexes are commonly hash partitioned, however, it's advantageous to do it only for very large indexes or for indexes experiencing a very high load. In summary, it's not uncommon at all to create nonpartitioned indexes in order to support primary keys and unique keys.

The second problem with local partitioned indexes is that they can make the performance worse for SQL statements that are unable to take advantage of partition pruning. The causes of such situations are described in the “Range Partitioning” section earlier in this chapter. The impact on index scans might be very high. The following example, based on the range-partitioned table in Figure 13-5, shows what the problem might be. At first, a nonpartitioned index is created. With it, a query retrieves one row by performing four logical reads. This is good. Notice that the TABLE ACCESS BY GLOBAL INDEX ROWID operation indicates that the rowid comes from a global or nonpartitioned index:

```
SQL> CREATE INDEX i ON t (n3);

SQL> SELECT * FROM t WHERE n3 = 3885;

------------------------------------------------------------------------------
| Id | Operation                            | Name | Starts | A-Rows | Buffers |
------------------------------------------------------------------------------
|  0 | SELECT STATEMENT                     |      |      1 |      1 |       4 |
|  1 |  TABLE ACCESS BY GLOBAL INDEX ROWID| T    |      1 |      1 |       4 |
|* 2 |   INDEX RANGE SCAN                   | I    |      1 |      1 |       3 |
------------------------------------------------------------------------------

   2 - access("N3"=3885)
```

For the second part of this test, the index is re-created. This time it’s a local index. Because the table has 48 partitions, the index will have 48 partitions as well. Because the test query doesn’t contain a restriction based on the partition key, no partition pruning can be performed. This is confirmed not only by the PARTITION RANGE ALL operation but also by the Pstart and Pstop columns. Also notice that the TABLE ACCESS BY LOCAL INDEX ROWID operation indicates that the rowid comes from a local partitioned index. The problem with this execution plan is that instead of executing a single index scan like in the previous case, this time an index scan is performed for each partition (notice the Starts column for operations 2 and 3). Therefore, even if only a single row is retrieved, 50 logical reads are necessary:

```
SQL> CREATE INDEX i ON t (n3) LOCAL;

SQL> SELECT * FROM t WHERE n3 = 3885;

----------------------------------------------------------------------------------------------
| Id | Operation                            | Name | Starts | Pstart| Pstop | A-Rows | Buffers |
----------------------------------------------------------------------------------------------
|  0 | SELECT STATEMENT                     |      |      1 |       |       |      1 |      50 |
|  1 |  PARTITION RANGE ALL                 |      |      1 |     1 |    48 |      1 |      50 |
|  2 |   TABLE ACCESS BY LOCAL INDEX ROWID| T    |     48 |     1 |    48 |      1 |      50 |
|* 3 |    INDEX RANGE SCAN                  | I    |     48 |     1 |    48 |      1 |      49 |
----------------------------------------------------------------------------------------------

   3 - access("N3"=3885)
```

In summary, without partition pruning, the number of logical reads increases proportionally to the number of partitions. Therefore, as pointed out previously, sometimes it can be better to use a nonpartitioned index than a partitioned one. Or, as a compromise, it could be good to have a limited number of partitions. Note, though, that sometimes you have no choice. For example, bitmap indexes can be created only as local indexes.

## Invisible Indexes

From version 11.1 onward, there's an optional index attribute that specifies whether an index is visible to the query optimizer. By default, indexes are visible. In case an index is invisible, it's regularly maintained when the data of the table it's based on is modified, but the query optimizer can't take advantage of it during the generation of execution plans. Because invisible indexes are regularly maintained, constraints based on a unique index are still regularly enforced even if the index they're based on is invisible.

---

■ **Caution** In version 11.1 the invisibility of an index isn't total. In fact, the query optimizer takes advantage of invisible indexes in two unexpected situations. First, even though an invisible index isn't included in the execution plan, the query optimizer can use the statistics associated to it to improve the estimations it makes. The `invisible_index_stats.sql` script demonstrates such a case. Second, the database engine can take advantage of an invisible index to avoid false contention due to unindexed foreign keys.

---

The following example, based on the `invisible_index.sql` script, shows how to make an index invisible and the impact of such an operation for a specific query:

```
SQL> SELECT * FROM t WHERE id = 42;

--------------------------------------------
| Id  | Operation                    | Name |
--------------------------------------------
|   0 | SELECT STATEMENT             |      |
|   1 |  TABLE ACCESS BY INDEX ROWID| T    |
|*  2 |   INDEX UNIQUE SCAN         | T_PK |
--------------------------------------------

   2 - access("ID"=42)

SQL> SELECT visibility FROM user_indexes WHERE index_name = 'T_PK';

VISIBILITY
----------
VISIBLE

SQL> ALTER INDEX t_pk INVISIBLE;

SQL> SELECT visibility FROM user_indexes WHERE index_name = 'T_PK';

VISIBILITY
----------
INVISIBLE

SQL> SELECT * FROM t WHERE id = 42;
```

```
----------------------------------
| Id  | Operation         | Name |
----------------------------------
|   0 | SELECT STATEMENT  |      |
|*  1 |  TABLE ACCESS FULL| T    |
----------------------------------

   1 - filter("ID"=42)
```

Making or creating an index invisible is helpful in two situations:

- To assess whether it's possible to drop an existing index without jeopardizing performance. This is useful when the index to be dropped is large. In fact, tentatively dropping a large index isn't an option, because it takes too long and too many resources to rebuild the index if later you decide that dropping it was a mistake.
- To create an index without making it immediately available to the query optimizer.

By default, the query optimizer honors the visibility of an index. This is because, by default, the `optimizer_use_invisible_indexes` initialization parameter is set to `FALSE`. If this parameter is set to `TRUE`, either at the system or at the session level, the query optimizer is allowed to treat invisible indexes as visible ones. As of version 11.1.0.7, it's also possible to control whether the query optimizer honors the visibility of indexes by adding the `(no_)use_invisible_indexes` hint to SQL statements.

---

■ **Caution** According to the *High Availability Overview* manual: "An invisible index is maintained for any DML operation but isn't used by the optimizer unless you explicitly specify the index with a hint." Unfortunately, this sentence contains a mistake. The issue is that an `index` hint can't be used to change the visibility of invisible indexes. Only the `(no_)use_invisible_indexes` hint impacts the visibility of invisible indexes.

---

Up to and including version 11.2, it's not possible to create several indexes on the same set of columns (if you try, the database engine raises an ORA-01408). From version 12.1 onward this restriction has been removed. In fact, it's possible to create several indexes on the same set of columns provided that only one of those indexes is visible at a given time. This possibility, for applications that must be made highly available, is useful for changing the uniqueness, type (B-tree or bitmap), and partitioning of an index without having to plan downtime. Without this feature, it might be necessary to stop an otherwise highly available application while dropping and re-creating an index. The following example, which is an excerpt of the output generated by the `multiple_indexes.sql` script, illustrates the feature:

1. Setup the initial objects:

```
SQL> CREATE TABLE t (n1 NUMBER, n2 NUMBER, n3 NUMBER);

SQL> CREATE INDEX i_i ON t (n1);
```

2. Creating another *visible* index on the same column (`n1`) as the previous one isn't supported (notice that the new index is unique):

```
SQL> CREATE UNIQUE INDEX i_ui ON t (n1);
CREATE UNIQUE INDEX i_ui ON t (n1)
                               *
ERROR at line 1:
ORA-01408: such column list already indexed
```

3. Creating several *invisible* indexes is supported (notice the differences in every index):

```
SQL> CREATE UNIQUE INDEX i_ui ON t (n1) INVISIBLE;

SQL> CREATE BITMAP INDEX i_bi ON t (n1) INVISIBLE;

SQL> CREATE INDEX i_hpi ON t (n1) INVISIBLE
  2  GLOBAL PARTITION BY HASH (n1) PARTITIONS 4;

SQL> CREATE INDEX i_rpi ON t (n1) INVISIBLE
  2  GLOBAL PARTITION BY RANGE (n1) (
  3    PARTITION VALUES LESS THAN (10),
  4    PARTITION VALUES LESS THAN (MAXVALUE)
  5  );
```

4. Switch between two indexes by making the older one invisible and the new one visible:

```
SQL> ALTER INDEX i_i INVISIBLE;

SQL> ALTER INDEX i_ui VISIBLE;
```

## Partial Indexes

For performance purposes, it's sometimes *not* necessary to index all data stored in a table. That's especially true for huge, range partitioned tables containing a long history of specific events like orders or phone calls. For example, it might only be necessary to index the data from the last day, or from the most recent week, and to leave any older data unindexed. Such indexes are called *partial indexes*. Using them in the right situations can save a lot of disk space that would be unnecessarily allocated.

Even though in versions up to and including 11.2, some kind of partial indexes are supported by implementing particular tricks, it's only from version 12.1 onward that Oracle Database provides a formal syntax to support partial indexes. The basic idea of the syntax introduced in version 12.1 is that an indexing property specifying whether data has to be indexed can be set at the table and at the partition level.

The following example, based on the `partial_index.sql` script, shows how to specify that indexing be disabled for all partitions except for the one that explicitly sets the `INDEXING ON` property (obviously you can also set `INDEXING ON` at the table level and `INDEXING OFF` for specific partitions only):

```
CREATE TABLE t (
  id NUMBER NOT NULL,
  d DATE NOT NULL,
  n NUMBER NOT NULL,
  pad VARCHAR2(4000) NOT NULL
)
INDEXING OFF
PARTITION BY RANGE (d) (
  PARTITION t_jan_2014 VALUES LESS THAN (to_date('2014-02-01','yyyy-mm-dd')),
  PARTITION t_feb_2014 VALUES LESS THAN (to_date('2014-03-01','yyyy-mm-dd')),
  PARTITION t_mar_2014 VALUES LESS THAN (to_date('2014-04-01','yyyy-mm-dd')),
  PARTITION t_apr_2014 VALUES LESS THAN (to_date('2014-05-01','yyyy-mm-dd')),
  PARTITION t_may_2014 VALUES LESS THAN (to_date('2014-06-01','yyyy-mm-dd')),
  PARTITION t_jun_2014 VALUES LESS THAN (to_date('2014-07-01','yyyy-mm-dd')),
  PARTITION t_jul_2014 VALUES LESS THAN (to_date('2014-08-01','yyyy-mm-dd')),
  PARTITION t_aug_2014 VALUES LESS THAN (to_date('2014-09-01','yyyy-mm-dd')),
  PARTITION t_sep_2014 VALUES LESS THAN (to_date('2014-10-01','yyyy-mm-dd')),
  PARTITION t_oct_2014 VALUES LESS THAN (to_date('2014-11-01','yyyy-mm-dd')),
  PARTITION t_nov_2014 VALUES LESS THAN (to_date('2014-12-01','yyyy-mm-dd')),
  PARTITION t_dec_2014 VALUES LESS THAN (to_date('2015-01-01','yyyy-mm-dd')) INDEXING ON
)
```

When an index is created, it's possible to specify whether the indexing property has to be observed (INDEXING PARTIAL) or not (INDEXING FULL, this is the default value). The following SQL statement shows how to create a partial index:

```
CREATE INDEX i ON t (d) INDEXING PARTIAL
```

The key requirement for the use of partial indexes is that the data has to be stored in a partitioned table. Whether an index is nonpartitioned, local or global isn't relevant. Independently of that, only rows stored in a partition with INDEXING ON are indexed. With a table and an index like the ones created in the previous example, the query optimizer isn't restricted to either access all data through a table scan or an index scan. Instead, it can take advantage of the table expansion query transformation (refer to Chapter 6) and, as a result, generate different access paths depending on whether the data is indexed or not. The following example illustrates such a case:

```
SQL> SELECT *
  2  FROM t
  3  WHERE d BETWEEN to_date('2014-11-30 23:00:00','yyyy-mm-dd hh24:mi:ss')
  4              AND to_date('2014-12-01 01:00:00','yyyy-mm-dd hh24:mi:ss');

--------------------------------------------------------------------------------
| Id  | Operation                                  | Name    | Pstart| Pstop |
--------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                           |         |       |       |
|   1 |  VIEW                                      | VW_TE_2 |       |       |
|   2 |   UNION-ALL                                |         |       |       |
|   3 |    TABLE ACCESS BY GLOBAL INDEX ROWID BATCHED| T     |    12 |    12 |
|*  4 |     INDEX RANGE SCAN                       | I       |       |       |
|   5 |    PARTITION RANGE SINGLE                  |         |    11 |    11 |
|*  6 |     TABLE ACCESS FULL                      | T       |    11 |    11 |
--------------------------------------------------------------------------------

   4 - access("T"."D">=TO_DATE(' 2014-12-01 00:00:00', 'syyyy-mm-dd
              hh24:mi:ss') AND "D"<=TO_DATE(' 2014-12-01 01:00:00', 'syyyy-mm-dd
              hh24:mi:ss'))
   6 - filter("D">=TO_DATE(' 2014-11-30 23:00:00', 'syyyy-mm-dd
              hh24:mi:ss'))
```

## Single-table Hash Cluster Access

In practice, too few databases take advantage of single-table hash clusters. As a matter of fact, when they're correctly sized and accessed through an equality condition on the cluster key, they provide excellent performance. There are two reasons for this. First, they need no separate access structure (for example, an index) to locate data—in fact, the cluster key is enough to locate it. Second, all data related to a cluster key is clustered together. These two advantages were also demonstrated by the tests summarized in Figures 13-3 and 13-4 earlier in this chapter.

Single-table hash clusters are dedicated to the implementation of lookup tables that are frequently (ideally, always) accessed through a specific key. Basically, this is the same utilization you can get from index-organized tables. However, there are some major differences between the two. Table 13-4 lists the main advantages and disadvantages of single-table hash clusters compared to index-organized tables. The crucial disadvantage is that single-table hash clusters need to be accurately sized to take advantage of them.

***Table 13-4.*** *Single-table Hash Clusters Compared to Index-organized Tables*

| Advantages | Disadvantages |
|---|---|
| Better performance (if accessed through cluster key and sizing is done correctly) | Careful sizing needed to avoid hash collisions and waste of space |
| Cluster key might be different from primary key | Partitioning not supported |
| | LOB columns not supported |

When a single-table hash cluster is accessed through the cluster key, the TABLE ACCESS HASH operation appears in the execution plan. What it does is access the block(s) containing the required data directly through the cluster key. The following excerpt of the output generated by the hash_cluster.sql script illustrates this:

```
SELECT * FROM t WHERE id = 6

-------------------------------------------------------------
| Id  | Operation         | Name | Starts | A-Rows | Buffers |
-------------------------------------------------------------
|   0 | SELECT STATEMENT  |      |      1 |      1 |       1 |
|*  1 |  TABLE ACCESS HASH| T    |      1 |      1 |       1 |
-------------------------------------------------------------

   1 - access("ID"=6)
```

In addition to the equality condition, the only other condition that enables access to data through the cluster key is the IN condition. When it's specified, the operation that appears in the execution plan depends on the database version. In fact, while up to and including version 11.1 the CONCATENATION operation is used, from version 11.2 onward INLIST ITERATOR is used instead. Each child of both operations is executed once to get a particular cluster key. The following execution plan was generated on version 11.1.0.7:

```
SELECT * FROM t WHERE id IN (6, 8, 19, 28)

--------------------------------------------------------------
| Id  | Operation          | Name | Starts | A-Rows | Buffers |
--------------------------------------------------------------
|   0 | SELECT STATEMENT   |      |      1 |      4 |       4 |
|   1 |  CONCATENATION     |      |      1 |      4 |       4 |
|*  2 |   TABLE ACCESS HASH| T    |      1 |      1 |       1 |
|*  3 |   TABLE ACCESS HASH| T    |      1 |      1 |       1 |
|*  4 |   TABLE ACCESS HASH| T    |      1 |      1 |       1 |
|*  5 |   TABLE ACCESS HASH| T    |      1 |      1 |       1 |
--------------------------------------------------------------
```

```
   2 - access("ID"=28)
   3 - access("ID"=19)
   4 - access("ID"=8)
   5 - access("ID"=6)
```

The following execution plan was generated on version 11.2.0.1:

```
SELECT * FROM t WHERE id IN (6, 8, 19, 28)

---------------------------------------------------------------
| Id  | Operation          | Name | Starts | A-Rows | Buffers |
---------------------------------------------------------------
|   0 | SELECT STATEMENT   |      |      1 |      4 |       4 |
|   1 |  INLIST ITERATOR   |      |      1 |      4 |       4 |
|*  2 |   TABLE ACCESS HASH| T    |      4 |      4 |       4 |
---------------------------------------------------------------

   2 - access(("ID"=6 OR "ID"=8 OR "ID"=19 OR "ID"=28))
```

It's important to stress that all other conditions would lead to a full table scan if no index is available. For example, the following query, that contains a range condition in the `WHERE` clause, uses an index:

```
SELECT * FROM t WHERE id < 6

------------------------------------------------------------------------
| Id  | Operation                   | Name | Starts | A-Rows | Buffers |
------------------------------------------------------------------------
|   0 | SELECT STATEMENT            |      |      1 |      5 |       5 |
|   1 |  TABLE ACCESS BY INDEX ROWID| T    |      1 |      5 |       5 |
|*  2 |   INDEX RANGE SCAN          | T_PK |      1 |      5 |       3 |
------------------------------------------------------------------------

   2 - access("ID"<6)
```

Be aware that clusters have a specific object statistic that provides the average number of blocks per key. You can display it through the `avg_blocks_per_key` column of the `user_clusters` view (of course, there are `dba`, `all` and, in a 12.1 multitenant environment, `cdb` versions of that view as well). Unfortunately, this statistic isn't collected by the `dbms_stats` package. Instead, you have to execute the `ANALYZE CLUSTER` statement. For accurate estimations, you shouldn't forget to gather it.

## On to Chapter 14

This chapter describes not only the importance of selectivity in choosing an efficient access path, but also the different methods that are available for accessing data stored in a single table. For that purpose, SQL statements with weak selectivity should use either full table scans, full partition scans, or full index scans. It also discusses that in order to process SQL statements with strong selectivity efficiently, the access paths of choice are based on rowids, indexes, and single-table hash clusters.

This chapter covers only those SQL statements that process a single table. In practice, it's quite common for several tables to be joined together. To address that area, the next chapter describes the three basic join methods as well as their pros and cons. In other words, it describes when it makes sense to use each join method.

CHAPTER 14

# Optimizing Joins

When a SQL statement references several tables, the query optimizer has to determine, in addition to the access path for each table, which order the tables are joined in and which join methods are used. The goal of the query optimizer is to minimize the amount of processing by filtering out unneeded data as soon as possible.

This chapter starts by defining key terms and explains how the three basic join methods (nested loops join, merge join, and hash join) work. Some advice follows on how to choose the join methods. Finally, the chapter describes optimization techniques such as partition-wise joins and star transformation.

---

**Note** In this chapter, several SQL statements contain hints. This is done not only to show you which hint leads to which execution plan but also to show you examples of their utilization. In any case, neither real references nor full syntaxes are provided. You can find these in Chapter 2 of the *Oracle Database SQL Reference* manual.

---

## Definitions

To avoid misunderstandings, the following sections define some terms and concepts used through this chapter. Specifically, I cover the different types of join trees, the difference between restrictions and join conditions, and the different types of joins.

### Join Trees

All join methods supported by the database engine process only two sets of data at a time. These are called *left input* and *right input*. They're named in this way because when a graphical representation (see Figure 14-1) is used, one of the inputs is placed on the left of the join (T1) and the other on the right (T2). Note that in this graphical representation, the node on the left is executed before the node on the right.

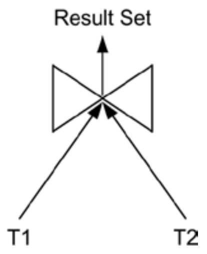


***Figure 14-1.*** *Graphical representation of a join between two sets of data*

When more than two sets of data must be joined, the query optimizer evaluates *join trees*. The types of join trees employed by the query optimizer are described in the next four sections.

## Left-Deep Trees

A *left-deep tree,* as shown in Figure 14-2, is a join tree where every join has a table (that is, not a result set generated by a previous join) as its right input. This is the join tree most commonly chosen by the query optimizer.

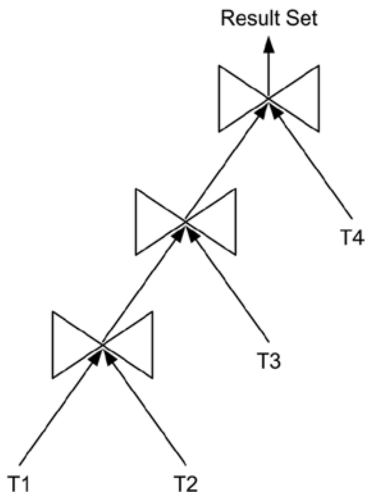


***Figure 14-2.*** *In a left-deep tree, the right input is always a table*

The following execution plan illustrates the join tree depicted in Figure 14-2. Note that the second child (that is, the right input) of each join operation (that is, lines 5, 6, and 7) is always a table:

```
-------------------------------------
| Id  | Operation            | Name |
-------------------------------------
|   0 | SELECT STATEMENT     |      |
|   1 |  HASH JOIN           |      |
|   2 |   HASH JOIN          |      |
|   3 |    HASH JOIN         |      |
|   4 |     TABLE ACCESS FULL| T1   |
|   5 |     TABLE ACCESS FULL| T2   |
|   6 |    TABLE ACCESS FULL | T3   |
|   7 |   TABLE ACCESS FULL  | T4   |
-------------------------------------
```

## Right-Deep Trees

A *right-deep tree*, as shown in Figure 14-3, is a join tree where every join has a table in its left input. This join tree is rarely chosen by the query optimizer.

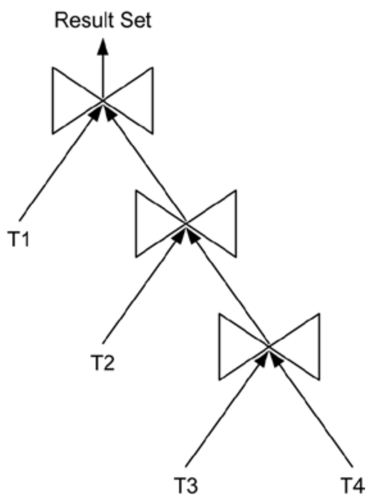


***Figure 14-3.*** *In a right-deep tree, the left input is always a table*

The following execution plan illustrates the join tree depicted in Figure 14-3. Note that the first child (that is, the left input) of each join operation (that is, lines 2, 4, and 6) is a table:

```
-------------------------------------
| Id  | Operation            | Name |
-------------------------------------
|   0 | SELECT STATEMENT     |      |
|   1 |  HASH JOIN           |      |
|   2 |   TABLE ACCESS FULL  | T1   |
|   3 |   HASH JOIN          |      |
|   4 |    TABLE ACCESS FULL | T2   |
|   5 |    HASH JOIN         |      |
|   6 |     TABLE ACCESS FULL| T3   |
|   7 |     TABLE ACCESS FULL| T4   |
-------------------------------------
```

## Zig-zag Trees

A *zig-zag tree*, as shown in Figure 14-4, is a join tree where every join has at least one table as input, but the input based on the table is sometimes on the left and sometimes on the right. This type of join tree isn't commonly used by the query optimizer.

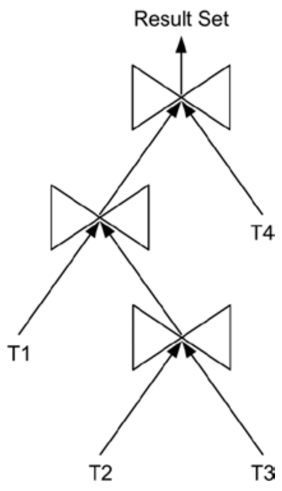


***Figure 14-4.*** *In a zig-zag tree, at least one of the two inputs is a table*

The following execution plan illustrates the join tree depicted in Figure 14-4:

```
-------------------------------------
| Id  | Operation            | Name |
-------------------------------------
|   0 | SELECT STATEMENT     |      |
|   1 |  HASH JOIN           |      |
|   2 |   HASH JOIN          |      |
|   3 |    TABLE ACCESS FULL | T1   |
|   4 |    HASH JOIN         |      |
|   5 |     TABLE ACCESS FULL| T2   |
|   6 |     TABLE ACCESS FULL| T3   |
|   7 |   TABLE ACCESS FULL  | T4   |
-------------------------------------
```

## Bushy Trees

A *bushy tree*, as shown in Figure 14-5, is a join tree that might have a join with two inputs that aren't tables. In other words, the structure of the tree is completely free. The query optimizer chooses this type of join tree only if it has no other possibility. This usually happens when unmergeable views or subqueries are present.

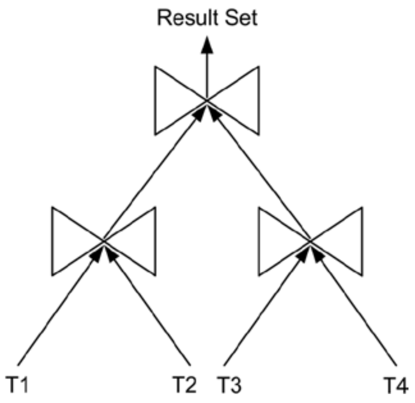


***Figure 14-5.*** *The structure of a bushy tree is completely free*

The following execution plan illustrates the join tree depicted in Figure 14-5. Notice that the children of join operation 1 are result sets of two other join operations:

```
-------------------------------------
| Id  | Operation            | Name |
-------------------------------------
|   0 | SELECT STATEMENT     |      |
|   1 |  HASH JOIN           |      |
|   2 |   VIEW               |      |
|   3 |    HASH JOIN         |      |
|   4 |     TABLE ACCESS FULL| T1   |
|   5 |     TABLE ACCESS FULL| T2   |
|   6 |   VIEW               |      |
|   7 |    HASH JOIN         |      |
|   8 |     TABLE ACCESS FULL| T3   |
|   9 |     TABLE ACCESS FULL| T4   |
-------------------------------------
```

## Types of Joins

There are two syntax types for specifying joins. The legacy syntax, which was specified by the very first SQL standard (SQL-86), uses both the FROM clause and the WHERE clause to specify joins. The newer syntax, available for the first time in SQL-92, uses only the FROM clause to specify a join. The newer syntax is sometimes called *ANSI join syntax*. However, both syntax types are valid from a SQL standard point of view. With Oracle Database, for historical reasons, the most commonly used syntax is the legacy one. In fact, not only are many developers and DBAs used to it, but many applications were developed using it as well. Nevertheless, the newer syntax offers possibilities that the legacy syntax doesn't support. The following sections provide examples based on both syntaxes. All queries used here as examples are provided in the `join_types.sql` script.

---

■ **Note** The join types described in this section aren't mutually exclusive. A given join may fall into more than one category. For example, it's perfectly plausible to conceive a theta join that is also a self-join.

---

### Cross Joins

A *cross join*, also called *Cartesian product*, is the operation that combines every row of one table with every row of another table. This type of operation is carried out in the two situations illustrated with the following queries. The first uses the legacy join syntax (no join condition is specified):

```
SELECT emp.ename, dept.dname FROM emp, dept
```

The second uses the new join syntax (the CROSS JOIN is used):

```
SELECT emp.ename, dept.dname FROM emp CROSS JOIN dept
```

In reality, cross joins are rarely needed. Nevertheless, the latter syntax is better to document the developer's intention. In fact, it has the advantage of being explicitly specified. With the former, it's not clear whether the person who wrote the SQL statement forgot a WHERE clause.

### Theta Joins

A *theta join* is equivalent to performing a selection over the result set of a cross join. In other words, instead of returning a combination of every row from one table with every row from another table, only the rows satisfying a join condition are returned. The following two queries are examples of this type of join:

```
SELECT emp.ename, salgrade.grade
FROM emp, salgrade
WHERE emp.sal BETWEEN salgrade.losal AND salgrade.hisal

SELECT emp.ename, salgrade.grade
FROM emp JOIN salgrade ON emp.sal BETWEEN salgrade.losal AND salgrade.hisal
```

Theta joins are also called *inner joins*. In the previous query using the new join syntax, the keyword INNER was assumed, but it can be explicitly coded, as in the following example:

```
SELECT emp.ename, salgrade.grade
FROM emp INNER JOIN salgrade ON emp.sal BETWEEN salgrade.losal AND salgrade.hisal
```

## Equi-joins

An *equi-join* is a special type of theta join where only equality operators are used in the join condition. The following two queries are examples:

```
SELECT emp.ename, dept.dname
FROM emp, dept
WHERE emp.deptno = dept.deptno

SELECT emp.ename, dept.dname
FROM emp JOIN dept ON emp.deptno = dept.deptno
```

## Self-joins

A *self-join* is a special type of theta join where a table is joined to itself. The following two queries are examples of this. Notice that the emp table is referenced twice in the FROM clause:

```
SELECT emp.ename, mgr.ename
FROM emp, emp mgr
WHERE emp.mgr = mgr.empno

SELECT emp.ename, mgr.ename
FROM emp JOIN emp mgr ON emp.mgr = mgr.empno
```

## Outer Joins

An *outer join* extends the result set of a theta join. In fact, with an outer join, all rows of one table (the *preserved table*) are returned even if no matching value is found in the other table. The value NULL is associated with the returned columns of the table that don't contain matching rows. For instance, the queries in the previous section (self-joins) don't return all rows of the emp table because the employee KING, who is the president, has no manager. To specify an outer join with the legacy syntax, an Oracle extension (based on the operator (+)) must be used. The following query is an example:

```
SELECT emp.ename, mgr.ename
FROM emp, emp mgr
WHERE emp.mgr = mgr.empno(+)
```

To specify an outer join with the new syntax, several possibilities exist. For example, the following two queries are equivalent to the previous one:

```
SELECT emp.ename, mgr.ename
FROM emp LEFT JOIN emp mgr ON emp.mgr = mgr.empno

SELECT emp.ename, mgr.ename
FROM emp mgr RIGHT JOIN emp ON emp.mgr = mgr.empno
```

The following query shows that, as for the theta join, the keyword OUTER might be added to explicitly specify that it's an outer join:

```
SELECT emp.ename, mgr.ename
FROM emp LEFT OUTER JOIN emp mgr ON emp.mgr = mgr.empno
```

In addition, with the new join syntax, it's possible to specify that all rows of both tables be returned by means of a *full outer join*. In other words, rows of both tables that have no matching row in the other table are preserved. The following query is an example:

```
SELECT mgr.ename AS manager, emp.ename AS subordinate
FROM emp FULL OUTER JOIN emp mgr ON emp.mgr = mgr.empno
```

Another possibility is to specify a *partitioned outer join*. Be careful: the word *partitioned* isn't related to the physical partitioning of objects discussed in Chapter 13. Instead, its meaning is that data is divided at runtime into several subsets. The idea is to perform an outer join not between two tables but between one table and subsets of another table. For example, in the following query, the emp table is divided into subsets based on the job column. Then each subset is outer joined with the dept table:

```
SELECT dept.dname, count(emp.empno)
FROM dept LEFT JOIN emp PARTITION BY (emp.job) ON emp.deptno = dept.deptno
WHERE emp.job = 'MANAGER'
GROUP BY dept.dname
```

## Semi-joins

A *semi-join* between two tables returns rows from one table when matching rows are available in the other table. Contrary to a theta join, rows from the left input are returned once at most. In addition, data from the right input isn't returned at all. The join condition is written with IN, EXISTS, ANY, or SOME. The following queries are examples:

```
SELECT deptno, dname, loc
FROM dept
WHERE deptno IN (SELECT deptno FROM emp)

SELECT deptno, dname, loc
FROM dept
WHERE EXISTS (SELECT deptno FROM emp WHERE emp.deptno = dept.deptno)

SELECT deptno, dname, loc
FROM dept
WHERE deptno = ANY (SELECT deptno FROM emp)

SELECT deptno, dname, loc
FROM dept
WHERE deptno = SOME (SELECT deptno FROM emp)
```

## Anti-joins

An *anti-join* is a special type of semi-join, where only rows from one table without matching rows in the other table are returned. The join condition is usually written with `NOT IN` or `NOT EXISTS`. The following two queries are examples:

```
SELECT deptno, dname, loc
FROM dept
WHERE deptno NOT IN (SELECT deptno FROM emp)

SELECT deptno, dname, loc
FROM dept
WHERE NOT EXISTS (SELECT deptno FROM emp WHERE emp.deptno = dept.deptno)
```

## Lateral Inline Views

A *lateral inline view* is an inline view (a query specified in the `FROM` clause of another query) that contains a correlation referring to other tables that precede it in the `FROM` clause. From version 12.1 onward, lateral inline views are supported through the `LATERAL` keyword. The following query shows an example:

```
SELECT dname, ename
FROM dept, LATERAL(SELECT * FROM emp WHERE dept.deptno = emp.deptno)
```

Note that if the `LATERAL` keyword is missing, the following error is raised:

```
SQL> SELECT dname, empno
  2  FROM dept, (SELECT * FROM emp WHERE dept.deptno = emp.deptno);
FROM dept, (SELECT * FROM emp WHERE dept.deptno = emp.deptno)
                                    *
ERROR at line 2:
ORA-00904: "DEPT"."DEPTNO": invalid identifier
```

For outer joins and cross joins, a similar functionality is provided through the `OUTER APPLY` and `CROSS APPLY` keywords.

## Restrictions vs. Join Conditions

To choose a join method, it's essential to understand the difference between *restrictions* (also known as *filtering conditions*) and *join conditions*. From a syntactical point of view, the two might be confused only when the legacy join syntax is used. In fact, with the legacy join syntax, the `WHERE` clause is used to specify both the restrictions and the join conditions. With the new join syntax, the restrictions are specified in the `WHERE` clause, and the join conditions are specified in the `FROM` clause. The following pseudo SQL statement illustrates this:

```
SELECT <columns>
FROM <table1> [OUTER] JOIN <table2> ON ( <join conditions> )
WHERE <restrictions>
```

From a conceptual point of view, a SQL statement containing join conditions and restrictions is executed in the following way:

- The two sets of data are joined based on the join conditions.
- The restrictions are applied to the result set returned by the join.

In other words, a join condition is specified to avoid a cross join while joining two sets of data. It isn't intended to filter out the result set. Instead, a restriction is specified to filter the result set returned by a previous operation (for example, a join). For instance, in the following query, the join condition is `emp.deptno = dept.deptno`, and the restriction is `dept.loc = 'DALLAS'`:

```
SELECT emp.ename
FROM emp, dept
WHERE emp.deptno = dept.deptno
AND dept.loc = 'DALLAS'
```

From an implementation point of view, it's not unusual that the query optimizer takes advantage of both restrictions and join conditions. On the one hand, join conditions might be used to filter out data. On the other hand, restrictions might be evaluated before join conditions to minimize the amount of data to be joined. For example, the previous query might be executed with the following execution plan. Notice how the `dept.loc = 'DALLAS'` restriction (operation 2) is applied before the `emp.deptno = dept.deptno` join condition (operation 1):

```
-----------------------------------
| Id  | Operation          | Name |
-----------------------------------
|   0 | SELECT STATEMENT   |      |
|*  1 |  HASH JOIN         |      |
|*  2 |   TABLE ACCESS FULL| DEPT |
|   3 |   TABLE ACCESS FULL| EMP  |
-----------------------------------

   1 - access("EMP"."DEPTNO"="DEPT"."DEPTNO")
   2 - filter("DEPT"."LOC"='DALLAS')
```

## Nested Loops Joins

The following sections describe how *nested loops joins* work. I describe their general behavior and then give some examples of two-table and four-table joins. Finally, I describe some optimization techniques. All examples are based on the `nested_loops_join.sql` script.

### Concept

The two sets of data processed by a nested loops join are called *outer loop* (also known as *driving row source*) and *inner loop*. The outer loop is the left input, and the inner loop is the right input. As illustrated in Figure 14-6, whereas the outer loop is executed once, the inner loop is executed once for each row returned by the outer loop.

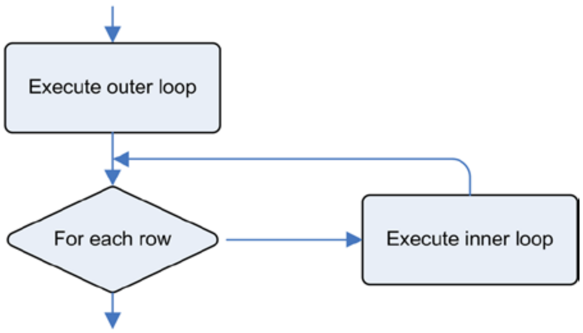


***Figure 14-6.*** *Overview of the processing performed by a nested loops join*

Nested loops joins have the following specific characteristics:

- The left input (outer loop) is executed only once. The right input (inner loop) is potentially executed many times.
- They're able to return the first row of the result set before completely processing all rows.
- They can take advantage of indexes to apply both restrictions and join conditions.
- They support all types of joins.

## Two-Table Join

The following is a simple execution plan processing a nested loops join between two tables. The example also shows how to force a nested loops join by using the `leading` and `use_nl` hints. The former indicates the order in which the tables are accessed. In other words, it specifies which table is accessed in the outer loop (`t1`) and which table is accessed in the inner loop (`t2`). The latter specifies which join method is used to join the data returned by the inner loop (that is, table `t2`) to table `t1`. It's essential to note that the `use_nl` hint contains no reference to table `t1`:

```
SELECT /*+ leading(t1 t2) use_nl(t2) full(t1) full(t2) */ *
FROM t1, t2
WHERE t1.id = t2.t1_id
AND t1.n = 19

-----------------------------------
| Id  | Operation          | Name |
-----------------------------------
|   0 | SELECT STATEMENT   |      |
|   1 |  NESTED LOOPS      |      |
|*  2 |   TABLE ACCESS FULL| T1   |
|*  3 |   TABLE ACCESS FULL| T2   |
-----------------------------------

   2 - filter("T1"."N"=19)
   3 - filter("T1"."ID"="T2"."T1_ID")
```

As described in Chapter 10, the NESTED LOOPS operation is of type *related combine*. This means the first child (the outer loop) controls the execution of the second child (the inner loop). In this case, the processing of the execution plan can be summarized as follows:

- All rows in table t1 are read through a full scan, and then the n = 19 restriction is applied.
- The full scan of table t2 is executed as many times as the number of rows returned by the previous step.

Clearly, when operation 2 (TABLE ACCESS FULL) returns more than one row, the previous execution plan is inefficient and, therefore, almost never chosen by the query optimizer. For this reason, to produce this specific example, specifying two access hints (full) is necessary to force the query optimizer to use this execution plan. On the other hand, if the outer loop returns a single row and the selectivity of the inner loop is weak, the full scan of table t2 might be good. To illustrate, let's create the following unique index on column n for table t1:

```
CREATE UNIQUE INDEX t1_n ON t1 (n)
```

With this index in place, the previous query can be executed with the following execution plan. Note that this time only one hint to control the access path for table t2 is specified. The other hints aren't necessary, because the query optimizer is aware that a nested loops join is the most efficient way in such a case to execute the query. In fact, because of operation 3 (INDEX UNIQUE SCAN), the inner loop is guaranteed to be executed no more than once:

```
SELECT /*+ full(t2) */ *
FROM t1, t2
WHERE t1.id = t2.t1_id
AND t1.n = 19

---------------------------------------------
| Id  | Operation                    | Name |
---------------------------------------------
|   0 | SELECT STATEMENT             |      |
|   1 |  NESTED LOOPS                |      |
|   2 |   TABLE ACCESS BY INDEX ROWID| T1   |
|*  3 |    INDEX UNIQUE SCAN         | T1_N |
|*  4 |   TABLE ACCESS FULL          | T2   |
---------------------------------------------

   3 - access("T1"."N"=19)
   4 - filter("T1"."ID"="T2"."T1_ID")
```

As discussed in the the previous section, if the selectivity of the inner loop is strong, using an index scan for the inner loop makes sense. Because the nested loops join is a related-combine operation, for the inner loop it's even possible to take advantage of the join condition for that purpose. For example, in the following execution plan, operation 5 does a lookup using the value of column t1.id that is returned by operation 3:

```
SELECT /*+ ordered use_nl(t2) index(t1) index(t2) */ *
FROM t1, t2
WHERE t1.id = t2.t1_id
AND t1.n = 19
```

```
-------------------------------------------------
| Id  | Operation                    | Name     |
-------------------------------------------------
|   0 | SELECT STATEMENT             |          |
|   1 |  NESTED LOOPS                |          |
|   2 |   TABLE ACCESS BY INDEX ROWID| T1       |
|*  3 |    INDEX UNIQUE SCAN         | T1_N     |
|   4 |   TABLE ACCESS BY INDEX ROWID| T2       |
|*  5 |    INDEX RANGE SCAN          | T2_T1_ID |
-------------------------------------------------

   3 - access("T1"."N"=19)
   5 - access("T1"."ID"="T2"."T1_ID")
```

In summary, if the inner loop is executed several (or many) times, only access paths that are sensible in case of strong selectivity and that lead to very few logical reads make sense.

## Four-Table Join

The following execution plan is an example of a typical left-deep tree, implemented with nested loops joins (refer to Figure 14-2 for a graphical representation). Notice how each table is accessed through indexes. The example also shows how to force nested loops joins by using the `ordered` and `use_nl` hints. The former specifies to access the tables in the same order as they appear in the `FROM` clause. The latter specifies which join method is used to join the tables referenced by the hint to the first table or to the result sets of the previous join operations:

```
SELECT /*+ ordered use_nl(t2 t3 t4) */ t1.*, t2.*, t3.*, t4.*
FROM t1, t2, t3, t4
WHERE t1.id = t2.t1_id
AND t2.id = t3.t2_id
AND t3.id = t4.t3_id
AND t1.n = 19
```

```
---------------------------------------------------
| Id  | Operation                      | Name     |
---------------------------------------------------
|   0 | SELECT STATEMENT               |          |
|   1 |  NESTED LOOPS                  |          |
|   2 |   NESTED LOOPS                 |          |
|   3 |    NESTED LOOPS                |          |
|   4 |     TABLE ACCESS BY INDEX ROWID| T1       |
|*  5 |      INDEX RANGE SCAN          | T1_N     |
|   6 |     TABLE ACCESS BY INDEX ROWID| T2       |
|*  7 |      INDEX RANGE SCAN          | T2_T1_ID |
|   8 |    TABLE ACCESS BY INDEX ROWID | T3       |
|*  9 |     INDEX RANGE SCAN           | T3_T2_ID |
|  10 |   TABLE ACCESS BY INDEX ROWID  | T4       |
|* 11 |    INDEX RANGE SCAN            | T4_T3_ID |
---------------------------------------------------
```

```
   5 - access("T1"."N"=19)
   7 - access("T1"."ID"="T2"."T1_ID")
   9 - access("T2"."ID"="T3"."T2_ID")
  11 - access("T3"."ID"="T4"."T3_ID")
```

The processing of this type of execution plan can be summarized as follows (this description assumes that no row prefetching is used):

1. When the first row is fetched (in other words, not when the query is parsed or executed), the processing starts by getting the first row that fulfills the `t1.n = 19` restriction from table `t1`.
2. Based on the data found in table `t1`, table `t2` is looked up. Note that the database engine takes advantage of the `t1.id = t2.t1_id` join condition to access table `t2`. In fact, no restriction is applied to that table. Only the first row that fulfills the join condition is returned to the parent operation.
3. Based on the data found in table `t2`, table `t3` is looked up. Also in this case, the database engine takes advantage of a join condition, `t2.id = t3.t2_id`, to access table `t3`. Only the first row that fulfills the join condition is returned to the parent operation.
4. Based on the data found in table `t3`, table `t4` is looked up. Here, too, the database engine takes advantage of a join condition, `t3.id = t4.t3_id`, to access table `t4`. The first row that fulfills the join condition is immediately returned to the client.
5. When the subsequent rows are fetched, the same actions are performed that were performed for the first fetch. Obviously, the processing is restarted from the position following the last match (that could be the second row that matches in table `t4`, if any). It's essential to stress that data is returned as soon as a row that fulfills the request is found. It isn't necessary to fully execute the join before returning the first row.

## Buffer Cache Prefetches

Basically, each access path, except for full scans, leads to single-block physical reads in the event of a cache miss. For nested loops joins, especially when many rows are processed, these single-block reads can be very inefficient. In fact, it's not unusual for nested loops joins to access blocks with many single-block physical reads.

To improve the efficiency of nested loops joins, the database engine is able to take advantage of optimization techniques that substitute single-block physical reads with multiblock physical reads. Three features use such an approach: *table prefetching*, *batching*, and *buffer cache prewarm*. The first two are associated with the execution plans presented in this section; the latter happens shortly after an instance bounce for any nested loops join. Note that these optimization techniques take place while accessing both indexes and tables.

The "Two-Table Join" section shows an execution plan based on a nested loops join that has the following shape:

```
-------------------------------------------------
| Id  | Operation                    | Name     |
-------------------------------------------------
|   0 | SELECT STATEMENT             |          |
|   1 |  NESTED LOOPS                |          |
|   2 |   TABLE ACCESS BY INDEX ROWID| T1       |
|*  3 |    INDEX UNIQUE SCAN         | T1_N     |
|   4 |   TABLE ACCESS BY INDEX ROWID| T2       |
|*  5 |    INDEX RANGE SCAN          | T2_T1_ID |
-------------------------------------------------

   3 - access("T1"."N"=19)
   5 - access("T1"."ID"="T2"."T1_ID")
```

In practice, for the Oracle Database versions covered in this book, this type of execution plan is used only when either the outer loop or the inner loop is based on an index unique scan (here, the `t1_n` index is unique). Let's see what happens if the `t1_n` index on column `n` is defined as follows (nonunique):

```
CREATE INDEX t1_n ON t1 (n)
```

With this index in place, the following execution plan is used. Notice the different position of the rowid access on table `t2`. In the previous plan, it's operation 4, whereas in the following, it's operation 1. It's peculiar that the child of the rowid access (operation 1) is the nested loops join (operation 2). Even though from a functional point of view the two execution plans are equivalent, the database engine uses execution plans with the following shape in order to take advantage of table prefetching:

```
-------------------------------------------------
| Id  | Operation                     | Name     |
-------------------------------------------------
|   0 | SELECT STATEMENT              |          |
|   1 |  TABLE ACCESS BY INDEX ROWID  | T2       |
|   2 |   NESTED LOOPS                |          |
|   3 |    TABLE ACCESS BY INDEX ROWID| T1       |
|*  4 |     INDEX RANGE SCAN          | T1_N     |
|*  5 |    INDEX RANGE SCAN           | T2_T1_ID |
-------------------------------------------------

   4 - access("T1"."N"=19)
   5 - access("T1"."ID"="T2"."T1_ID")
```

From version 11.1 onward, you can control the usage of the previous execution plan through the `nlj_prefetch` and `no_nlj_prefetch` hints.

From version 11.1 onward, for further optimizing nested loops joins, table prefetching is superseded by batching. As a result, the following execution plan should be observed instead of one from the previous section:

```
-------------------------------------------------
| Id  | Operation                     | Name     |
-------------------------------------------------
|   0 | SELECT STATEMENT              |          |
|   1 |  NESTED LOOPS                 |          |
|   2 |   NESTED LOOPS                |          |
|   3 |    TABLE ACCESS BY INDEX ROWID| T1       |
|*  4 |     INDEX RANGE SCAN          | T1_N     |
|*  5 |    INDEX RANGE SCAN           | T2_T1_ID |
|   6 |   TABLE ACCESS BY INDEX ROWID | T2       |
-------------------------------------------------

   4 - access("T1"."N"=19)
   5 - access("T1"."ID"="T2"."T1_ID")
```

Note that even though the query is always the same (that is, a two-table join), the execution plan contains two nested loops joins! To control batching, the `nlj_batching` and `no_nlj_batching` hints are available.

Looking at an execution plan can't tell you whether the database engine actually uses table prefetching or batching. The fact is, even though the query optimizer produces an execution plan that can take advantage of either table prefetch or batching, it's the execution engine that decides whether using that plan is sensible. The only way to know whether an optimization technique is used is to look at the physical reads performed by the server process—specifically, the wait events associated with them:

- The `db file sequential read` event is associated with single-block physical reads. Therefore, if it occurs, either no optimization technique took place or one was not needed (for example, because the required blocks are already in the buffer cache).
- The `db file scattered read` and `db file parallel read` events are associated with multiblock physical reads. The difference between the two is that the former is used for physical reads of adjacent blocks, and the latter is used for physical reads of nonadjacent blocks. Therefore, if one of them occurs for rowid accesses or index range scans, it means that an optimization technique took place.

## Merge Joins

The following sections describe how *merge joins* (also known as *sort-merge joins*) work. I begin by describing their general behavior and give some examples of two-table and four-table joins. Finally, I describe the work areas used during processing. All examples are based on the `merge_join.sql` script.

### Concept

The database engine, depending on the SQL statement and the physical database design, has the choice between several ways to carry out merge joins. In the general case, both sets of data are read and sorted according to the columns of the join condition. Once these operations are completed, the data contained in the two work areas is merged, as illustrated in Figure 14-7.

Merge joins are characterized by the following properties:

- The left input is executed only once.
- The right input is executed at most once. In the event the left input doesn't return any row, the right input isn't executed at all.
- Except when a Cartesian product is executed, the data returned by both inputs must be sorted according to the columns of the join condition.
- When data is sorted, both inputs must be fully read and sorted before returning the first row of the result set.
- All types of joins are supported.

Merge joins aren't used very often. The reason is that in most situations either nested loops joins or hash joins perform better than merge joins.

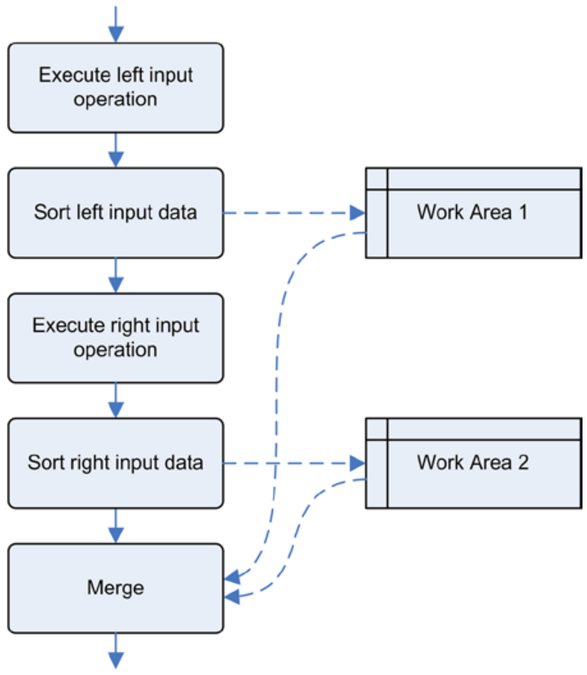


***Figure 14-7.*** *Overview of the processing performed by a merge join*

## Two-Table Join

The following is a simple execution plan processing a merge join between two tables. The example also shows how to force a merge join by using the `ordered` and `use_merge` hints:

```
SELECT /*+ ordered use_merge(t2) */ *
FROM t1, t2
WHERE t1.id = t2.t1_id
AND t1.n = 19

------------------------------------
| Id  | Operation              | Name |
------------------------------------
|   0 | SELECT STATEMENT       |      |
|   1 |  MERGE JOIN            |      |
|   2 |   SORT JOIN            |      |
|*  3 |    TABLE ACCESS FULL| T1   |
|*  4 |   SORT JOIN            |      |
|   5 |    TABLE ACCESS FULL| T2   |
------------------------------------

   3 - filter("T1"."N"=19)
   4 - access("T1"."ID"="T2"."T1_ID")
       filter("T1"."ID"="T2"."T1_ID")
```

As described in Chapter 10, the `MERGE JOIN` operation is of type unrelated combine. This means the two children are processed at most once and independently of each other. Provided that both inputs return data, the processing of the previous execution plan can be summarized as follows:

- All rows in table `t1` are read through a full scan, the `n = 19` restriction is applied, and the resulting rows are sorted according to the columns used as the join condition (`id`).
- All rows in table `t2` are read through a full scan and sorted according to the columns used as the join condition (`t1_id`).
- The two sets of data are joined together, and the resulting rows are returned. Note that the join itself is straightforward because the two sets of data are sorted according to the same value (the columns used in the join condition).

It's interesting to notice in the previous execution plan that the join condition is applied by the `SORT JOIN` operation of the right input, not by the `MERGE JOIN` operation as you might expect. This is because for each row returned by the left input, the `MERGE JOIN` operation accesses the memory structure associated to the right input to check whether rows fulfilling the join condition exist.

---

■ **Note** The key difference between the `SORT JOIN` operation and the `SORT ORDER BY` operation is that the former always performs a binary sort, but the latter, depending on the NLS configuration, can perform either a binary or a linguistic sort.

---

The most important limitation of the `MERGE JOIN` operation (like for the other unrelated-combine operations) is its inability to take advantage of indexes to apply join conditions. In other words, indexes can be used only as an access path to evaluate restrictions (if specified) before sorting the inputs. Therefore, in order to choose the access path, you have to apply the methods discussed in Chapter 13 to both tables. For instance, if the `n = 19` restriction provides strong selectivity, it could be useful to create an index to apply it:

```
CREATE INDEX t1_n ON t1 (n)
```

In fact, with this index in place, the following execution plan might be used. You should notice that table `t1` is no longer accessed through a full table scan:

```
----------------------------------------------
| Id  | Operation                     | Name |
----------------------------------------------
|   0 | SELECT STATEMENT              |      |
|   1 |  MERGE JOIN                   |      |
|   2 |   SORT JOIN                   |      |
|   3 |    TABLE ACCESS BY INDEX ROWID| T1   |
|*  4 |     INDEX RANGE SCAN          | T1_N |
|*  5 |   SORT JOIN                   |      |
|   6 |    TABLE ACCESS FULL          | T2   |
----------------------------------------------

   4 - access("T1"."N"=19)
   5 - access("T1"."ID"="T2"."T1_ID")
       filter("T1"."ID"="T2"."T1_ID")
```

To execute merge joins, a non-negligible amount of resources may be spent on sort operations. To improve performance, the query optimizer avoids performing sort operations whenever it saves resources. But of course, this is possible only when the data is already sorted according to the columns used as the join condition. That happens in two situations. The first is when an index range scan taking advantage of an index built on the columns used as the join condition is used. The second is when a step preceding the merge join (for example, another merge join) already sorted the data in the right order. For example, in the following execution plan, notice how table `t1` is accessed through the `t1_pk` index (which is built on the `id` column used as the join condition). As a result, for the left input, the sort operation (`SORT JOIN`) is avoided:

```
----------------------------------------------
| Id  | Operation                    | Name  |
----------------------------------------------
|   0 | SELECT STATEMENT             |       |
|   1 |  MERGE JOIN                  |       |
|*  2 |   TABLE ACCESS BY INDEX ROWID| T1    |
|   3 |    INDEX FULL SCAN           | T1_PK |
|*  4 |   SORT JOIN                  |       |
|   5 |    TABLE ACCESS FULL         | T2    |
----------------------------------------------

   2 - filter("T1"."N"=19)
   4 - access("T1"."ID"="T2"."T1_ID")
       filter("T1"."ID"="T2"."T1_ID")
```

An important caveat about execution plans such as the previous one is that, because no sort operation takes place for the left input, no work area is associated to it. As a result, there's no place to store data resulting from the left input while the right input is executed. The processing of the previous execution plan can be summarized as follows:

1. A first batch of rows is extracted from the `t1` table through the `t1_pk` index. It's essential to understand that this first batch contains all rows only when the result set is very small. Remember, there's no work area to temporarily store many rows.
2. Provided the previous step returns some data, all rows in the `t2` table are read through a full scan, sorted according to the columns used as the join condition, and stored in the work area, possibly spilling temporary data to the disk.
3. The two sets of data are joined together, and the resulting rows are returned. When the first batch of rows extracted from the left input has been completely processed, more rows are extracted from the `t1` table, if necessary, and joined to the data of the right input already present in a work area.

As shown in the previous execution plan, it's possible to avoid the sort associated to the left input. The same doesn't apply to the right input, though. To explain why, the following example shows the execution plan chosen by the query optimizer if you execute the same query as before, but this time by specifying, through the `leading` hint, that table `t2` must be accessed in the left input. Notice that even though the access path of the right input returns data in the correct order, the data has to go through a `SORT JOIN` operation (4):

```
SELECT /*+ leading(t2 t1) use_merge(t1) index(t1 t1_pk) */ *
FROM t1, t2
WHERE t1.id = t2.t1_id
AND t1.n = 19
```

```
----------------------------------------------
| Id  | Operation                    | Name  |
----------------------------------------------
|   0 | SELECT STATEMENT             |       |
|   1 |  MERGE JOIN                  |       |
|   2 |   SORT JOIN                  |       |
|   3 |    TABLE ACCESS FULL         | T2    |
|*  4 |   SORT JOIN                  |       |
|*  5 |    TABLE ACCESS BY INDEX ROWID| T1    |
|   6 |     INDEX FULL SCAN          | T1_PK |
----------------------------------------------

   4 - access("T1"."ID"="T2"."T1_ID")
       filter("T1"."ID"="T2"."T1_ID")
   5 - filter("T1"."N"=19)
```

The SORT JOIN operation is required because the MERGE JOIN operation needs to access the memory structure associated to the right input to check whether rows fulfilling the join condition exist. And that access must be performed in a memory structure that not only contains the data in a specific order, but also allows performing efficient lookups based on the join condition.

Cartesian products based on a merge join are executed differently. Compared to all other cases described in this section, the main optimization applied to them is that data doesn't need to be sorted. The reason is simple: no join condition exists, and therefore it's not possible to sort the data according to the columns referenced in that nonexistent condition. As a result, idependently of the access paths used to get data, no SORT JOIN operations are required. For the right input, it's nevertheless necessary to store the data in a memory structure. For that purpose, a BUFFER SORT operation (which, despite its name, doesn't sort the data) is used. The following example illustrates such a case:

```
SELECT /*+ ordered use_merge(t2) */ *
FROM t1, t2

-------------------------------------
| Id  | Operation            | Name |
-------------------------------------
|   0 | SELECT STATEMENT     |      |
|   1 |  MERGE JOIN CARTESIAN|      |
|   2 |   TABLE ACCESS FULL  | T1   |
|   3 |   BUFFER SORT        |      |
|   4 |    TABLE ACCESS FULL | T2   |
-------------------------------------
```

## Four-Table Join

The following execution plan is an example of a typical left-deep tree implemented with merge joins (refer to Figure 14-2 for a graphical representation). The example also shows how to force a merge join by means of the leading and use_merge hints. Note that the leading hint supports several tables:

```
SELECT /*+ leading(t1 t2 t3 t4) use_merge(t2 t3 t4) */ t1.*, t2.*, t3.*, t4.*
FROM t1, t2, t3, t4
WHERE t1.id = t2.t1_id
AND t2.id = t3.t2_id
AND t3.id = t4.t3_id
AND t1.n = 19

----------------------------------------
| Id  | Operation                | Name |
----------------------------------------
|   0 | SELECT STATEMENT         |      |
|   1 |  MERGE JOIN              |      |
|   2 |   SORT JOIN              |      |
|   3 |    MERGE JOIN            |      |
|   4 |     SORT JOIN            |      |
|   5 |      MERGE JOIN          |      |
|   6 |       SORT JOIN          |      |
|*  7 |        TABLE ACCESS FULL| T1   |
|*  8 |       SORT JOIN          |      |
|   9 |        TABLE ACCESS FULL| T2   |
|* 10 |     SORT JOIN            |      |
|  11 |      TABLE ACCESS FULL   | T3   |
|* 12 |   SORT JOIN              |      |
|  13 |    TABLE ACCESS FULL     | T4   |
----------------------------------------

   7 - filter("T1"."N"=19)
   8 - access("T1"."ID"="T2"."T1_ID")
       filter("T1"."ID"="T2"."T1_ID")
  10 - access("T2"."ID"="T3"."T2_ID")
       filter("T2"."ID"="T3"."T2_ID")
  12 - access("T3"."ID"="T4"."T3_ID")
       filter("T3"."ID"="T4"."T3_ID")
```

The processing isn't really different from the two-table join discussed in the previous section. However, it's important to emphasize that data is sorted several times because each join condition is based on different columns. For example, the data resulting from the join between table `t1` and table `t2`, which is sorted according to the `id` column of table `t1`, is sorted again by operation 4 according to the `id` column of table `t2`. The same happens with the data returned by operation 3. In fact, it has to be sorted according to the `id` column of table `t3`. In summary, to process this type of execution plan, six sorts have to be performed, and all of them have to be performed before being able to return a single row.

## Work Areas

To process a merge join, up to two work areas in memory are used to sort data. If a sort is completely processed in memory, it's called an *in-memory sort*. If a sort needs to spill temporary data to the disk, it's called an *on-disk sort*. From a performance point of view, it should be obvious that in-memory sorts should be faster than on-disk sorts. The next sections discuss how these two types of sorts work. I also discuss how to recognize which one is used to process a SQL statement, based on the output of the `dbms_xplan` package.

I discuss work area configuration (sizing) in Chapter 9. As you might recall from that chapter, there are two sizing methods. Which one is used depends on the value of the `workarea_size_policy` initialization parameter. The two methods are as follows:

> `auto`: The database engine automatically does the sizing of the work areas. The total amount of PGA dedicated to one instance is controlled by the `pga_aggregate_target` initialization parameter or, as of version 11.1, by the `memory_target` initialization parameter.
>
> `manual`: The `sort_area_size` initialization parameter limits the maximum size of a single work area. In addition, the `sort_area_retained_size` initialization parameter controls how the PGA is released when the sort is over.

## In-Memory Sorts

The processing of an in-memory sort is straightforward. The data is loaded into a work area, and the sorting takes place. It's important to stress that all data must be loaded into the work area, not only the columns referenced as the join condition. Therefore, to avoid wasting a lot of memory, only the columns that are really necessary should be referenced in the `SELECT` clause. To illustrate this point, let's look at two examples based on the four-table join discussed in the previous section.

In the following example, all columns in all tables are referenced in the `SELECT` clause. In the execution plan, the `OMem` and `Used-Mem` columns provide information about the work areas. The former is the estimated amount of memory needed for an in-memory sort. The latter is the actual amount of memory used by the operation during execution. The value between brackets (that is, the zero) means that the sorts were fully processed in memory:

```
SELECT t1.*, t2.*, t3.*, t4.*
FROM t1, t2, t3, t4
WHERE t1.id = t2.t1_id
AND t2.id = t3.t2_id
AND t3.id = t4.t3_id
AND t1.n = 19

-----------------------------------------------------------
| Id  | Operation               | Name |  OMem | Used-Mem |
-----------------------------------------------------------
|   0 | SELECT STATEMENT        |      |       |          |
|   1 |  MERGE JOIN             |      |       |          |
|   2 |   SORT JOIN             |      | 34816 |30720  (0)|
|   3 |    MERGE JOIN           |      |       |          |
|   4 |     SORT JOIN           |      |  5120 | 4096  (0)|
|   5 |      MERGE JOIN         |      |       |          |
|   6 |       SORT JOIN         |      |  3072 | 2048  (0)|
|*  7 |        TABLE ACCESS FULL| T1   |       |          |
|*  8 |       SORT JOIN         |      | 21504 |18432  (0)|
|   9 |        TABLE ACCESS FULL| T2   |       |          |
|* 10 |     SORT JOIN           |      |   160K|  142K (0)|
|  11 |      TABLE ACCESS FULL  | T3   |       |          |
|* 12 |   SORT JOIN             |      |  1045K|  928K (0)|
|  13 |    TABLE ACCESS FULL    | T4   |       |          |
-----------------------------------------------------------
```

```
   7 - filter("T1"."N"=19)
   8 - access("T1"."ID"="T2"."T1_ID")
       filter("T1"."ID"="T2"."T1_ID")
  10 - access("T2"."ID"="T3"."T2_ID")
       filter("T2"."ID"="T3"."T2_ID")
  12 - access("T3"."ID"="T4"."T3_ID")
       filter("T3"."ID"="T4"."T3_ID")
```

In the following example, only one of the columns referenced in the SELECT clause isn't referenced in the WHERE clause (t4.id). It's important to note that except for operation 6, smaller work areas were used for all other sorts and that this was true even though the execution plan was the same in both cases. Also notice that the query optimizer's estimations (column Omem) take this difference into consideration:

```
SELECT t1.id, t2.id, t3.id, t4.id
FROM t1, t2, t3, t4
WHERE t1.id = t2.t1_id
AND t2.id = t3.t2_id
AND t3.id = t4.t3_id
AND t1.n = 19
```

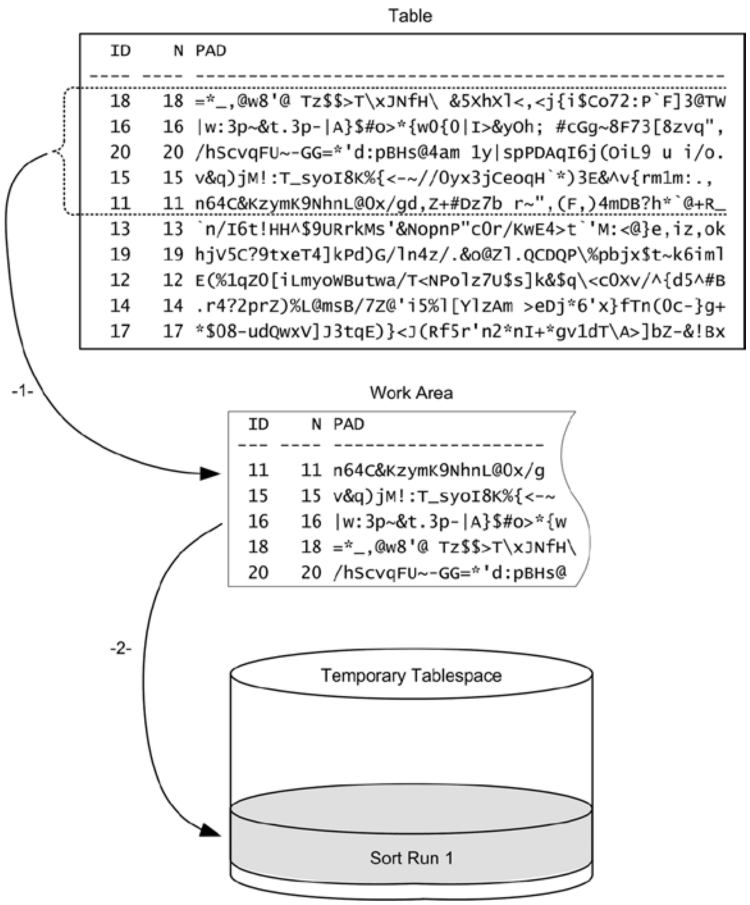
```
-----------------------------------------------------------
| Id  | Operation                | Name |  OMem | Used-Mem |
-----------------------------------------------------------
|   0 | SELECT STATEMENT         |      |       |          |
|   1 |  MERGE JOIN              |      |       |          |
|   2 |   SORT JOIN              |      | 14336 |12288  (0)|
|   3 |    MERGE JOIN            |      |       |          |
|   4 |     SORT JOIN            |      |  3072 | 2048  (0)|
|   5 |      MERGE JOIN          |      |       |          |
|   6 |       SORT JOIN          |      |  3072 | 2048  (0)|
|*  7 |        TABLE ACCESS FULL| T1   |       |          |
|*  8 |       SORT JOIN          |      | 16384 |14336  (0)|
|   9 |        TABLE ACCESS FULL| T2   |       |          |
|* 10 |     SORT JOIN            |      |   106K|96256  (0)|
|  11 |      TABLE ACCESS FULL   | T3   |       |          |
|* 12 |   SORT JOIN              |      |   407K|  361K (0)|
|  13 |    TABLE ACCESS FULL     | T4   |       |          |
-----------------------------------------------------------

   7 - filter("T1"."N"=19)
   8 - access("T1"."ID"="T2"."T1_ID")
       filter("T1"."ID"="T2"."T1_ID")
  10 - access("T2"."ID"="T3"."T2_ID")
       filter("T2"."ID"="T3"."T2_ID")
  12 - access("T3"."ID"="T4"."T3_ID")
       filter("T3"."ID"="T4"."T3_ID")
```

## On-disk Sorts

When a work area is too small to contain all data, the database engine processes the sort in several steps. These steps are detailed in the following list. The actual number of steps depends, obviously, not only on the amount of data but also on the size of the work areas.

1. The data is read from the table and stored in the work area. While storing it, a structure is built that organizes the data according to the sort criteria. In this example, the data is sorted according to the `id` column. This is step 1 in Figure 14-8.



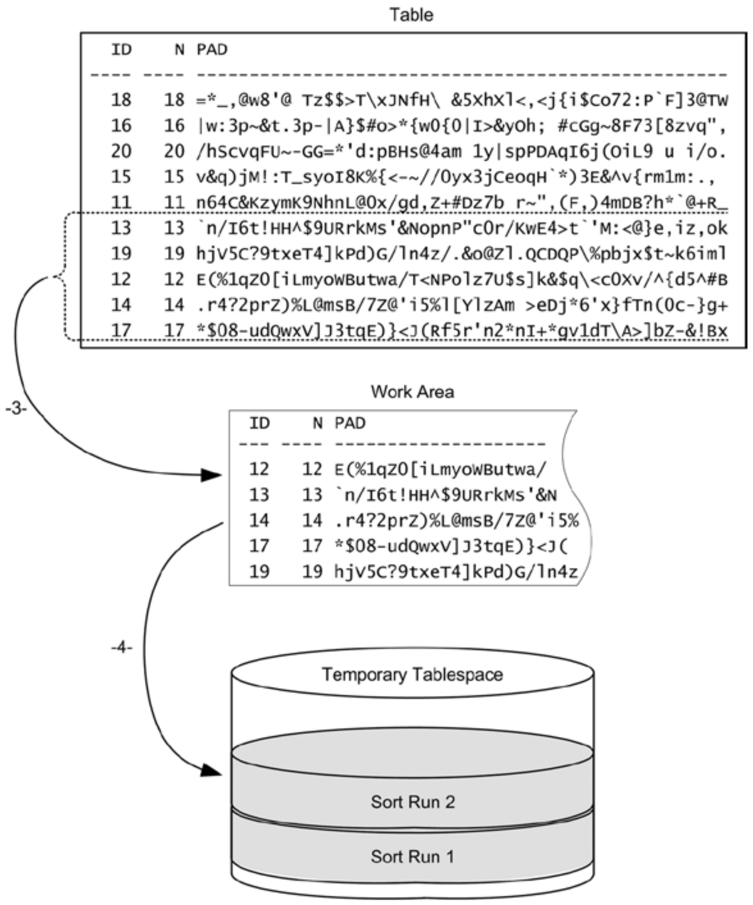
```
  ID     N PAD
---- ---- --------------------------------------------------
  18    18 =*_,@w8'@ Tz$$>T\xJNfH\ &5XhXl<,<j{i$Co72:P`F]3@TW
  16    16 |w:3p~&t.3p-|A}$#o>*{w0{0|I>&yOh; #cGg~8F73[8zvq",
  20    20 /hScvqFU~-GG=*'d:pBHs@4am 1y|spPDAqI6j(OiL9 u i/o.
  15    15 v&q)jM!:T_syoI8K%{<-~//Oyx3jCeoqH`*)3E&^v{rm1m:.,
  11    11 n64C&KzymK9NhnL@0x/gd,Z+#Dz7b r~",(F,)4mDB?h*`@+R_
  13    13 `n/I6t!HH^$9URrkMs'&NopnP"c0r/KwE4>t`'M:<@}e,iz,ok
  19    19 hjV5C?9txeT4]kPd)G/ln4z/.&o@Zl.QCDQP\%pbjx$t~k6iml
  12    12 E(%1qZO[iLmyoWButwa/T<NPolz7U$s]k&$q\<c0Xv/^{d5^#B
  14    14 .r4?2prZ)%L@msB/7Z@'i5%l[YlzAm >eDj*6'x}fTn(0c-}g+
  17    17 *$08-udQwxV]J3tqE)}<J(Rf5r'n2*nI+*gv1dT\A>]bZ-&!Bx
```

```
 ID     N PAD
--- ---- --------------------
 11    11 n64C&KzymK9NhnL@0x/g
 15    15 v&q)jM!:T_syoI8K%{<-~
 16    16 |w:3p~&t.3p-|A}$#o>*{w
 18    18 =*_,@w8'@ Tz$$>T\xJNfH\
 20    20 /hScvqFU~-GG=*'d:pBHs@
```

***Figure 14-8.*** *On-disk sort, first sort run*

2. When the work area is full, part of its content is spilled into a temporary segment in the user's temporary tablespace. This type of data batch is called a *sort run*. Note that all data is stored not only in the work area but also in the temporary segment. This is step 2 in Figure 14-8.

3. Since data has been spilled into the temporary segment, some free space is available in the work area. Therefore, it's possible to continue reading and storing the input data in the work area. This is step 3 in Figure 14-9.

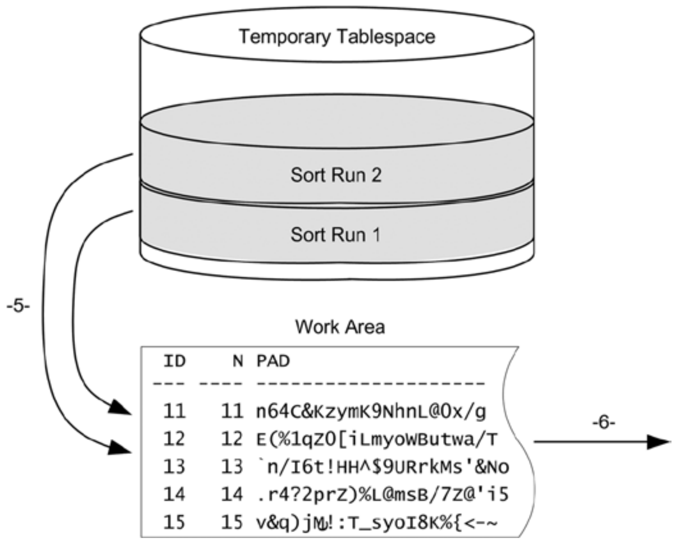


```
  ID    N PAD
---- ---- ---------------------------------------------------
  18   18 =*_,@w8'@ Tz$$>T\xJNfH\ &5XhXl<,<j{i$Co72:P`F]3@TW
  16   16 |w:3p~&t.3p-|A}$#o>*{w0{0|I>&yOh; #cGg~8F73[8zvq",
  20   20 /hScvqFU~-GG=*'d:pBHs@4am 1y|spPDAqI6j(OiL9 u i/o.
  15   15 v&q)jM!:T_syoI8K%{<-~//0yx3jCeoqH`*)3E&^v{rm1m:.,
  11   11 n64C&KzymK9NhnL@0x/gd,Z+#Dz7b r~",(F,)4mDB?h*`@+R_
  13   13 `n/I6t!HH^$9URrkMs'&NopnP"c0r/KwE4>t`'M:<@}e,iz,ok
  19   19 hjV5C?9txeT4]kPd)G/ln4z/.&o@Zl.QCDQP\%pbjx$t~k6iml
  12   12 E(%1qZO[iLmyoWButwa/T<NPolz7U$s]k&$q\<c0Xv/^{d5^#B
  14   14 .r4?2prZ)%L@msB/7Z@'i5%l[YlzAm >eDj*6'x}fTn(0c-}g+
  17   17 *$08-udQwxV]J3tqE)}<J(Rf5r'n2*nI+*gv1dT\A>]bZ-&!Bx
```

```
 ID    N PAD
--- ---- --------------------
 12   12 E(%1qZO[iLmyoWButwa/
 13   13 `n/I6t!HH^$9URrkMs'&N
 14   14 .r4?2prZ)%L@msB/7Z@'i5%
 17   17 *$08-udQwxV]J3tqE)}<J(
 19   19 hjV5C?9txeT4]kPd)G/ln4z
```

***Figure 14-9.*** *On-disk sort, second sort run*

4. When the work area is full again, another sort run is stored in the temporary segment. This is step 4 in Figure 14-9.
5. When all data has been sorted and stored in the temporary segment, it's time to merge it. The merge phase is necessary because each sort run is sorted independently of each other. To perform the merge, some data from each sort run is read back in the work area, starting from the head of each sort run. For example, while the row with `id` equal to 11 is stored in sort run 1, the row with `id` equal to 12 is stored in sort run 2. In other words, the merge takes advantage of the fact that data was sorted before spilling it into the temporary segment, in order to read each sort run sequentially. This is step 5 in Figure 14-10.



```
ID     N  PAD
---  ----  --------------------
11    11  n64C&KzymK9NhnL@0x/g
12    12  E(%1qZ0[iLmyowButwa/T
13    13  `n/I6t!HH^$9URrkMs'&No
14    14  .r4?2prZ)%L@msB/7Z@'i5
15    15  v&q)jM!:T_syoI8K%{<-~
```

*Figure 14-10. On-disk sort, merge phase*

6. As soon as some data sorted in the right way is available, it can be returned to the parent operation. This is step 6 in Figure 14-10.

In the example just described, the data has been written and read into/from the temporary segment only once. This type of sort is called a *one-pass sort*. When the size of the work area is much smaller than the amount of data to be sorted, several merge phases are necessary. In such a situation, the data is written and read into/from the temporary segment several times. This kind of sort is called a *multipass sort*. Obviously, from a performance point of view, a one-pass sort should be faster than a multipass sort.

To recognize the two types of sorts, you can use the output generated by the `dbms_xplan` package. Let's take a look at an output based on the four-table join already used in the previous section. In this output, two additional columns are displayed: `1Mem` and `Used-Tmp`. The former is the estimated amount of memory needed for a one-pass sort. The latter is the actual size of the temporary segment used by the operation during the execution. If no value is available, it means that an in-memory sort has been performed. Also, note how the values between brackets are no longer 0 for the operations using temporary space. Their value is set to the number of passes executed for the sort. In other words, while operation 10 was a one-pass sort, operation 12 was a multipass (nine-pass) sort:

```
SELECT t1.*, t2.*, t3.*, t4.*
FROM t1, t2, t3, t4
WHERE t1.id = t2.t1_id
AND t2.id = t3.t2_id
AND t3.id = t4.t3_id
AND t1.n = 19
```

```
-----------------------------------------------------------------------------
| Id  | Operation               | Name |  OMem  |  1Mem  | Used-Mem | Used-Tmp|
-----------------------------------------------------------------------------
|   0 | SELECT STATEMENT        |      |        |        |          |         |
|   1 |  MERGE JOIN             |      |        |        |          |         |
|   2 |   SORT JOIN             |      | 34816  | 34816  |30720  (0)|    1024 |
|   3 |    MERGE JOIN           |      |        |        |          |         |
|   4 |     SORT JOIN           |      |  5120  |  5120  | 4096  (0)|         |
|   5 |      MERGE JOIN         |      |        |        |          |         |
|   6 |       SORT JOIN         |      |  3072  |  3072  | 2048  (0)|         |
|*  7 |        TABLE ACCESS FULL| T1   |        |        |          |         |
|*  8 |       SORT JOIN         |      |  9216  |  9216  |18432  (0)|    1024 |
|   9 |        TABLE ACCESS FULL| T2   |        |        |          |         |
|* 10 |     SORT JOIN           |      |    99K|     99K|32768  (1)|    1024 |
|  11 |      TABLE ACCESS FULL  | T3   |        |        |          |         |
|* 12 |   SORT JOIN             |      |   954K|    532K|41984  (9)|    2048 |
|  13 |    TABLE ACCESS FULL    | T4   |        |        |          |         |
-----------------------------------------------------------------------------

   7 - filter("T1"."N"=19)
   8 - access("T1"."ID"="T2"."T1_ID")
       filter("T1"."ID"="T2"."T1_ID")
  10 - access("T2"."ID"="T3"."T2_ID")
       filter("T2"."ID"="T3"."T2_ID")
  12 - access("T3"."ID"="T4"."T3_ID")
       filter("T3"."ID"="T4"."T3_ID")
```

■ **Caution** Usually, the output of the `dbms_xplan` package displays values about the size of memory in bytes. Unfortunately, as pointed out in Chapter 10, the values in the `Used-Tmp` column must be multiplied by 1,024 to be converted to bytes. For example, in the previous output operations 10 and 12 used 1MB and 2MB of temporary space, respectively.

## Hash Joins

This section describes how hash joins work. A description of their general behavior and some examples of two-table and four-table joins are given first, followed by a description of the work areas used during processing. Finally, a particular optimization technique, index joins, is described. All examples are based on the `hash_join.sql` script.

### Concept

The two sets of data processed by a hash join are called *build input* and *probe input*. The build input is the left input, and the probe input is the right input. As illustrated in Figure 14-11, using every row of the build input, a hash table in memory (also using temporary space, if not enough memory is available) is built. Note that the hash key used for that purpose is computed based on the columns used as the join condition. Once the hash table contains all data from the build input, the processing of the probe input begins. Every row is probed against the hash table to find out whether it fulfills the join condition. Obviously, only matching rows are returned.

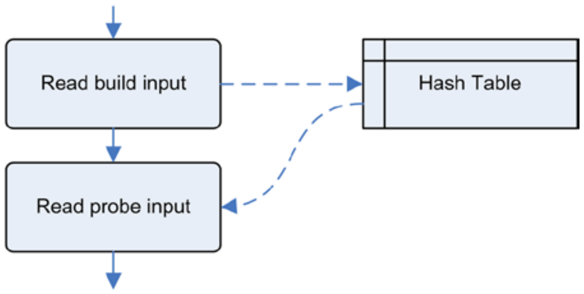


***Figure 14-11.*** *Overview of the processing performed by a hash join*

Hash joins are characterized by the following properties:

- The build input is executed only once.
- The probe input is executed at most once. In the event the build input doesn't return any row, the probe input isn't executed at all.
- The hash table is built on the build input only. Consequently, it's usually built on the smallest input.
- Before returning the first row, only the build input must be fully processed.
- Cross joins, theta joins, and partitioned outer joins aren't supported.

## Two-table Joins

The following is a simple execution plan processing a hash join between two tables. The example also shows how to force a hash join by means of the `leading` and `use_hash` hints:

```
SELECT /*+ leading(t1 t2) use_hash(t2) */ *
FROM t1, t2
WHERE t1.id = t2.t1_id
AND t1.n = 19

-----------------------------------
| Id  | Operation           | Name |
-----------------------------------
|   0 | SELECT STATEMENT    |      |
|*  1 |  HASH JOIN          |      |
|*  2 |   TABLE ACCESS FULL| T1   |
|   3 |   TABLE ACCESS FULL| T2   |
-----------------------------------

   1 - access("T1"."ID"="T2"."T1_ID")
   2 - filter("T1"."N"=19)
```

As described in Chapter 10, the HASH JOIN operation is of type unrelated combine. This means that the two children are processed at most once and independently of each other. In this case, the processing of the execution plan can be summarized as follows:

- All rows of table t1 are read through a full scan, the n = 19 restriction is applied, and a hash table is built with the resulting rows. To build the hash table, a hash function is applied to the columns used as the join condition (id).
- All rows of table t2 are read through a full scan, the hash function is applied to the columns used as the join condition (t1_id), and the hash table is probed. If a match is found, the resulting row is returned.

The most important limitation of the HASH JOIN operation (as for other unrelated-combine operations) is the inability to take advantage of indexes to apply join conditions. This means that indexes can be used as the access path only if restrictions are specified. Consequently, in order to choose the access path, it's necessary to apply the methods discussed in Chapter 10 to both tables. For instance, if the n = 19 restriction provides strong selectivity, it could be useful to create an index like the following one to apply it:

```
CREATE INDEX t1_n ON t1 (n)
```

In fact, with this index in place, the following execution plan might be used. Note that table t1 is no longer accessed through a full table scan:

```
---------------------------------------------
| Id  | Operation                    | Name |
---------------------------------------------
|   0 | SELECT STATEMENT             |      |
|*  1 |  HASH JOIN                   |      |
|   2 |   TABLE ACCESS BY INDEX ROWID| T1   |
|*  3 |    INDEX RANGE SCAN          | T1_N |
|   4 |   TABLE ACCESS FULL          | T2   |
---------------------------------------------

   1 - access("T1"."ID"="T2"."T1_ID")
   3 - access("T1"."N"=19)
```

## Four-Table Joins

The following execution plan is an example of a typical left-deep tree implemented with hash joins (refer to Figure 14-2 for a graphical representation). The example also shows how to force a hash join by using the leading and use_hash hints:

```
SELECT /*+ leading(t1 t2 t3 t4) use_hash(t2 t3 t4) */ t1.*, t2.*, t3.*, t4.*
FROM t1, t2, t3, t4
WHERE t1.id = t2.t1_id
AND t2.id = t3.t2_id
AND t3.id = t4.t3_id
AND t1.n = 19
```

```
-------------------------------------
| Id  | Operation            | Name |
-------------------------------------
|   0 | SELECT STATEMENT     |      |
|*  1 |  HASH JOIN           |      |
|*  2 |   HASH JOIN          |      |
|*  3 |    HASH JOIN         |      |
|*  4 |     TABLE ACCESS FULL| T1   |
|   5 |     TABLE ACCESS FULL| T2   |
|   6 |    TABLE ACCESS FULL | T3   |
|   7 |   TABLE ACCESS FULL  | T4   |
-------------------------------------

   1 - access("T3"."ID"="T4"."T3_ID")
   2 - access("T2"."ID"="T3"."T2_ID")
   3 - access("T1"."ID"="T2"."T1_ID")
   4 - filter("T1"."N"=19)
```

The processing of this type of execution plan is summarized here:

- Table `t1` is read through a full scan, the `n = 19` restriction is applied, and a hash table containing the resulting rows is created.
- Table `t2` is read through a full scan, and the hash table created in the previous step is probed. Then a hash table containing the resulting rows is created.
- Table `t3` is read through a full scan, and the hash table created in the previous step is probed. Then a hash table containing the resulting rows is created.
- Table `t4` is read through a full scan, and the hash table created in the previous step is probed. The resulting rows are returned. The first row can be returned only when the `t1`, `t2`, and `t3` tables have been fully processed. Instead, it isn't necessary to fully process table `t4` in order to return the first row.

One peculiar property of hash joins is that they also support right-deep and zig-zag trees. The following execution plan is an example of the former (refer to Figure 14-3 for a graphical representation). Compared to the previous example, only the hints specified in the SQL statement are different. Notice that in this case, the `leading` hint doesn't directly specify the order in which the tables are accessed (which is `t1` ➤ `t2` ➤ `t3` ➤ `t4`). Instead, it specifies the order before applying the `swap_join_inputs` hints that request to swap the left and right inputs:

```
SELECT /*+ leading(t3 t4 t2 t1) use_hash(t1 t2 t4) swap_join_inputs(t1)
           swap_join_inputs(t2) */ t1.*, t2.*, t3.*, t4.*
FROM t1, t2, t3, t4
WHERE t1.id = t2.t1_id
AND t2.id = t3.t2_id
AND t3.id = t4.t3_id
AND t1.n = 19
```

```
------------------------------------
| Id  | Operation             | Name |
------------------------------------
|   0 | SELECT STATEMENT      |      |
|*  1 |  HASH JOIN            |      |
|*  2 |   TABLE ACCESS FULL   | T1   |
|*  3 |   HASH JOIN           |      |
|   4 |    TABLE ACCESS FULL  | T2   |
|*  5 |    HASH JOIN          |      |
|   6 |     TABLE ACCESS FULL| T3   |
|   7 |     TABLE ACCESS FULL| T4   |
------------------------------------

   1 - access("T1"."ID"="T2"."T1_ID")
   2 - filter("T1"."N"=19)
   3 - access("T2"."ID"="T3"."T2_ID")
   5 - access("T3"."ID"="T4"."T3_ID")
```

One of the differences between the two execution plans (the left-deep tree and the right-deep tree) is the number of active work areas (hash tables) that are being used at a given time. With a left-deep tree, at most two work areas are used at the same time. In addition, when the last table is processed, only a single work area is needed. On the other hand, in a right-deep tree, during almost the entire execution a number of work areas (that is equal to the number of joins) are allocated and probed. Another difference between the two execution plans is the size of their work areas. Whereas the right-deep tree work areas contain data from a single table, the left-deep tree work areas can contain data resulting from the join of several tables. Therefore, the size of the left-deep tree work areas varies depending on whether the joins restrict the amount of data that's returned.

The v$sql_workarea_active dynamic performance view provides information about the active work areas. The following query shows the work areas used by one session that is currently executing the previous execution plan. Although the operation_id column is used to relate the work areas to an operation in the execution plan, the actual_mem_used column shows the size (in bytes), and the tempseg_size columns and tablespace give information about the utilization of temporary space:

```
SQL> SELECT operation_id, operation_type, actual_mem_used, tempseg_size, tablespace
  2  FROM v$session s, v$sql_workarea_active w
  3  WHERE s.sid = w.sid
  4  AND s.sid = 24
  5  ORDER BY operation_id;

OPERATION_ID OPERATION_TYPE ACTUAL_MEM_USED TEMPSEG_SIZE TABLESPACE
------------ -------------- --------------- ------------ ----------
           1 HASH-JOIN                79872
           3 HASH-JOIN               161792
           5 HASH-JOIN               185344      1048576 TEMP
```

## Work Areas

To process a hash join, a work area in memory is used to store the hash table. If the work area is large enough to store the whole hash table, the hash join is fully processed in memory. If the work area isn't large enough, data is spilled into a temporary segment. (I explain how to recognize whether a join is fully executed in memory earlier in the chapter.)

I discuss work area configuration (sizing) in Chapter 9. As you may recall, there are two methods to perform the sizing. Which one you use depends on the value of the workarea_size_policy initialization parameter:

> auto: The database engine automatically does the sizing of the work areas. The total amount of the PGA dedicated to one instance is controlled by the pga_aggregate_target initialization parameter or, as of version 11.1, by the memory_target initialization parameter.
>
> manual: The hash_area_size initialization parameter limits the maximum size of a single work area.

## Index Joins

Index joins can be executed only with hash joins. Because of this, they can be considered a special case of hash joins. Their purpose is to avoid expensive table scans by joining two or more indexes belonging to the same table. This may be very useful when a table has many indexed columns and few of them are referenced by a SQL statement. The following query is an example. Note how the query references a single table, but in spite of what you might expect, a join is executed instead of a single table access. It's also important to notice that the join condition between the two data sets is based on the rowids. The example also shows how to force an index join through the index_join hint:

```
SELECT /*+ index_join(t4 t4_n t4_pk) */ id, n
FROM t4
WHERE id BETWEEN 10 AND 20
AND n < 100

-----------------------------------------------
| Id  | Operation          | Name             |
-----------------------------------------------
|   0 | SELECT STATEMENT   |                  |
|*  1 |  VIEW              | index$_join$_001 |
|*  2 |   HASH JOIN        |                  |
|*  3 |    INDEX RANGE SCAN| T4_N             |
|*  4 |    INDEX RANGE SCAN| T4_PK            |
-----------------------------------------------

   1 - filter("ID"<=20 AND "N"<100 AND "ID">=10)
   2 - access(ROWID=ROWID)
   3 - access("N"<100)
   4 - access("ID">=10 AND "ID"<=20)
```

An interesting restriction of index joins is that the query optimizer doesn't choose them if the rowid of the table is referenced in the SELECT clause. Because the rowid is always part of an index, this restriction is due to the current implementation, and not to the lack of needed information in the index itself.

## Outer Joins

The three basic join methods described in the previous sections support outer joins. When an outer join is executed, the only difference visible in the execution plan is the OUTER keyword that is appended to the join operation. To illustrate, the following SQL statement is executed, because of hints, with an outer hash join. Notice that even if the SQL statement is written with the new join syntax, the predicate uses the Oracle proprietary syntax based on the (+) operator:

```
SELECT /*+ leading(t1) use_hash(t2) */ *
FROM t1 LEFT JOIN t2 ON t1.id = t2.t1_id

-----------------------------------
| Id  | Operation         | Name |
-----------------------------------
|   0 | SELECT STATEMENT   |      |
|*  1 |  HASH JOIN OUTER   |      |
|   2 |   TABLE ACCESS FULL| T1   |
|   3 |   TABLE ACCESS FULL| T2   |
-----------------------------------

   1 - access("T1"."ID"="T2"."T1_ID"(+))
```

Except for hash joins that are right outer joins, the preserved table (for example, the t1 table in the previous SQL statement) must be the left input of the join operation. The following execution plan, based on the same SQL statement as the previous one, illustrates this. In practice, the query optimizer chooses to build the hash table on the smallest result set. Of course, this is useful to limit the size of the work area. In this case, because table t1 is smaller than table t2, for illustration purposes it's necessary to force the query optimizer to swap the two join inputs with the swap_join_inputs hint:

```
SELECT /*+ leading(t1) use_hash(t2) swap_join_inputs(t2) */ *
FROM t1 LEFT JOIN t2 ON t1.id = t2.t1_id

--------------------------------------
| Id  | Operation            | Name |
--------------------------------------
|   0 | SELECT STATEMENT      |      |
|*  1 |  HASH JOIN RIGHT OUTER|      |
|   2 |   TABLE ACCESS FULL   | T2   |
|   3 |   TABLE ACCESS FULL   | T1   |
--------------------------------------

   1 - access("T1"."ID"="T2"."T1_ID"(+))
```

Consequently, whenever the query optimizer has to generate an execution plan for a SQL statement containing an outer join, except with hash joins, its options are limited.

## Choosing the Join Method

To choose a join method, you must consider the following issues:

- The optimizer goal—that is, first-rows and all-rows optimization
- The type of join to be optimized and the selectivities of the predicates
- Whether to execute the join in parallel

The next sections discuss, based on these three criteria, how to choose a join method—or more specifically, how to choose among a nested loops join, a merge join, and a hash join.

### First-Rows Optimization

With first-rows optimization, the overall response time is a secondary goal of the query optimizer. The response time to return the first rows is far and away its most important goal. Therefore, for a successful first-rows optimization, joins should return rows as soon as the first matches are found and not after all rows have been processed. For this purpose, nested loops joins are often the best choice. Hash joins, which support partial execution to only some extent, are useful now and then. In contrast, merge joins are rarely suitable for a first-rows optimization.

### All-Rows Optimization

With an all-rows optimization, the response time to return the last row is the most important goal of the query optimizer. Therefore, for a successful all-rows optimization, joins should be completely executed as fast as possible. To choose the best join method, it must be considered whether, to reduce the number of logical reads to the minimum, it's necessary to take advantage of an index to apply the join condition. If applying the join condition through an index is necessary, a nested loops join has to be used. Otherwise, a hash join is often the best choice. Generally speaking, merge joins are considered only when either the result sets are already sorted or hash joins can't be used because of technical limitations (see the next section). Another case where a merge join could be interesting is when a sort operation can be avoided. This happens when the merge join returns the rows in the order specified by the `ORDER BY` clause.

### Supported Join Methods

To choose a join method, you have to know which type of join has to be executed. In fact, not all join methods support all types of joins. Table 14-1 summarizes which methods are available in which situation.

***Table 14-1.*** *Types of Joins Supported by Each Join Method*

| Join | Nested Loops Join | Hash Join | Merge Join |
|---|---|---|---|
| Cross join | ✓ | | ✓ |
| Theta join | ✓ | | ✓ |
| Equi-join | ✓ | ✓ | ✓ |
| Semi/anti-join | ✓ | ✓ | ✓ |
| Outer join | ✓ | ✓ | ✓ |
| Partitioned outer join | ✓ | | ✓ |

## Parallel Joins

All join methods can be executed in parallel. However, as shown in Figure 14-12, they scale differently. Depending on the degree of parallelism, one method may be faster than the other. So, to pick out the best join method, it's essential to know both whether parallel processing is used and what the degree of parallelism is. (Chapter 15 covers parallel processing.)

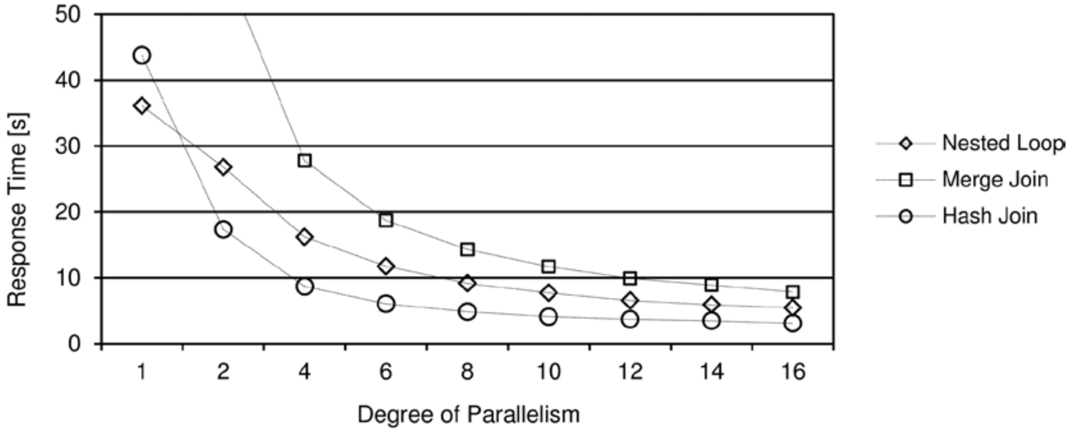


***Figure 14-12.*** *Comparison of the performance with different degrees of parallelism. This figure shows a two-table join (50K and 94M rows) executed on a system with 8 cores*

## Partition-wise Joins

A *partition-wise join* (which shouldn't be confused with a partitioned outer join) is an optimization technique that the query optimizer applies to merge and hash joins only. Partition-wise joins are used to reduce the amount of CPU, memory, and, in case of RAC, network resources used to process joins. The basic idea is to divide a large join into several smaller joins. Partition-wise joins can be full or partial. The following sections describe these two alternatives. All queries used as examples are provided in the `pwj.sql` script.

---

■ **Note** Partition-wise joins require partitioned tables. Consequently, they're available only when the Partitioning option in Enterprise Edition is used.

---

### Full Partition-wise Joins

To illustrate the operation of a full partition-wise join, let's begin by describing how a join without this optimization is performed. Figure 14-13 shows a join between two partitioned tables. A single join of all rows of the two tables is executed by a single server process.

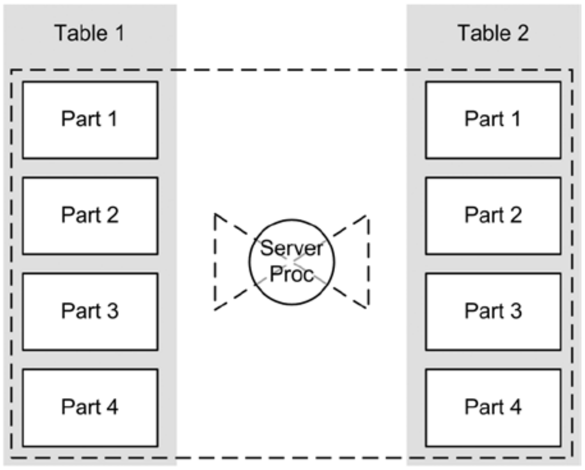


***Figure 14-13.*** *Joining two partitioned tables without a partition-wise join*

When the two tables are equi-partitioned on their join keys (for example, a child table that is reference-partitioned based on the foreign key to its parent table), the database engine is able to take advantage of a full partition-wise join. Instead of executing a single large join, it performs, as shown in Figure 14-14, several smaller joins (in this case, four). Note that this is possible because the tables are partitioned in the same way. Because of this, every row that is stored in, for example, partition 1 of table 1 can have matching rows only in partition 1 of table 2.

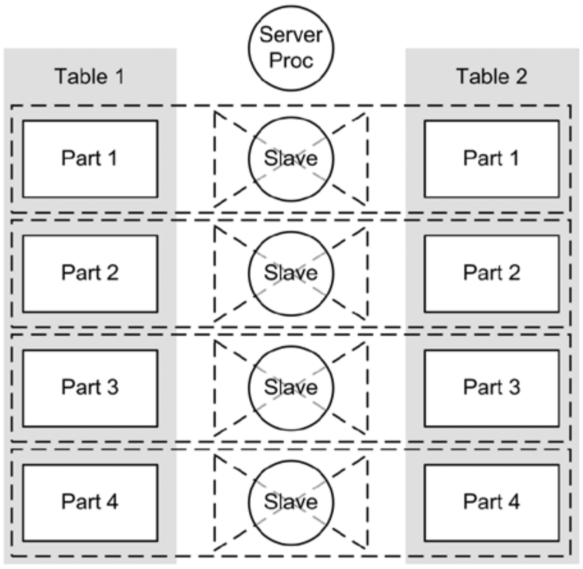


***Figure 14-14.*** *Joining two tables with a full partition-wise join*

One of the most useful things about decomposing a large join into several smaller joins is the possibility of parallelizing the execution. In fact, the database engine can start a separate slave process for each join. For example, in Figure 14-14 the server process coordinates four slave processes to execute the full partition-wise join. (Chapter 15 provides detailed information about parallel processing.)

Figure 14-15 shows the results of a performance test based on the `pwj_performance.sql` script. The purpose of the test was to reproduce an execution like the one illustrated in Figure 14-14 or, specifically, a join of two tables with four partitions. In this particular case, the tables contained 10,000,000 and 100,000,000 rows, respectively. Note that the degree of the parallel executions was equal to the number of partitions—that is, four.

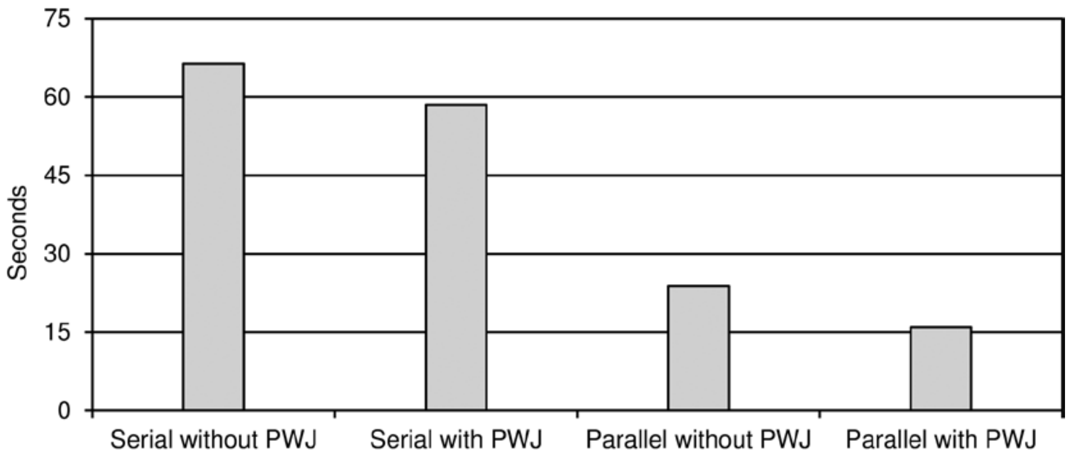


***Figure 14-15.*** *Response time of a two-table join with and without full partition-wise join*

To recognize whether a full partition-wise join is used, it's necessary to look at the execution plan. If the partition operation appears *before* the join operation, it means that a full partition-wise join is being used. In the following execution plan, the `PARTITION HASH ALL` operation appears before the `HASH JOIN` operation:

```
SELECT *
FROM t1p, t2p
WHERE t1p.id = t2p.id

------------------------------------
| Id  | Operation            | Name |
------------------------------------
|   0 | SELECT STATEMENT     |      |
|   1 |  PARTITION HASH ALL  |      |
|*  2 |   HASH JOIN          |      |
|   3 |    TABLE ACCESS FULL| T1P   |
|   4 |    TABLE ACCESS FULL| T2P   |
------------------------------------

   2 - access("T1P"."ID"="T2P"."ID")
```

The previous execution plan shows a serial full partition-wise join. The following shows the execution plan used with parallel processing for the very same SQL statement. Also in this case, the `PX PARTITION HASH ALL` operation appears before the `HASH JOIN` operation. Notice how you can use the `pq_distribute` hint to instruct the query optimizer to use a full partition-wise join (Chapter 15 provides more information about the operations used for parallel processing):

```
SELECT /*+ ordered use_hash(t2p) pq_distribute(t2p none none) */ *
FROM t1p, t2p
WHERE t1p.id = t2p.id

--------------------------------------------
| Id  | Operation                 | Name     |
--------------------------------------------
|   0 | SELECT STATEMENT          |          |
|   1 |  PX COORDINATOR           |          |
|   2 |   PX SEND QC (RANDOM)     | :TQ10000 |
|   3 |    PX PARTITION HASH ALL|          |
|*  4 |     HASH JOIN             |          |
|   5 |      TABLE ACCESS FULL    | T1P      |
|   6 |      TABLE ACCESS FULL    | T2P      |
--------------------------------------------

   4 - access("T1P"."ID"="T2P"."ID")
```

Because full partition-wise joins require two equi-partitioned tables, special care is necessary to use this optimization technique during physical database design. In other words, it's common to equi-partition tables that are expected to be frequently subject to massive joins. If you don't equi-partition them, you won't be able to benefit from full partition-wise joins.

---

■ **Caution** Two list-partitioned tables are considered equi-partitioned not only when both tables have the same number of partitions defined on the same list of values, but also when the partitions are defined in the same order. The `pwj_list.sql` script illustrates this limitation.

---

It's also important to note that all partitioning methods are supported and that a full partition-wise join can join partitions to subpartitions. To illustrate, let's say you have two tables: `sales` and `customers`. The join key is `customer_id`. If both tables are hash partitioned, have the same number of partitions, and both use the join key as the partition key, it's possible to take advantage of full partition-wise joins. Keep in mind that it's often a requirement to partition a table like `sales` (in other words, a table containing historical data) with range partitioning. In such a situation, it's possible to use composite partitioning for the `sales` table. This composite partitioning is done at the partition level with a range, and at the subpartition level with a hash in order to meet both requirements. Thus, the full partition-wise join is performed between the hash partitions of the `customers` table with the subpartitions of the `sales` table.

## Partial Partition-wise Joins

In contrast to full partition-wise joins, partial partition-wise joins don't require two equi-partitioned tables. In addition, only one table must be partitioned according to the join key; the other table (which can be partitioned or not) is dynamically partitioned based on the join key. Another characteristic of partial partition-wise joins is that they can be executed only in parallel. Figure 14-16 illustrates a partial partition-wise join. In this case, one of the tables isn't partitioned at all. During execution, the database engine starts two sets of parallel slaves. The first reads the nonpartitioned table (Table 2) and distributes the data according to the join key. The second receives the data from the first set, and each slave reads one partition of the partitioned table and then performs its part of the join.

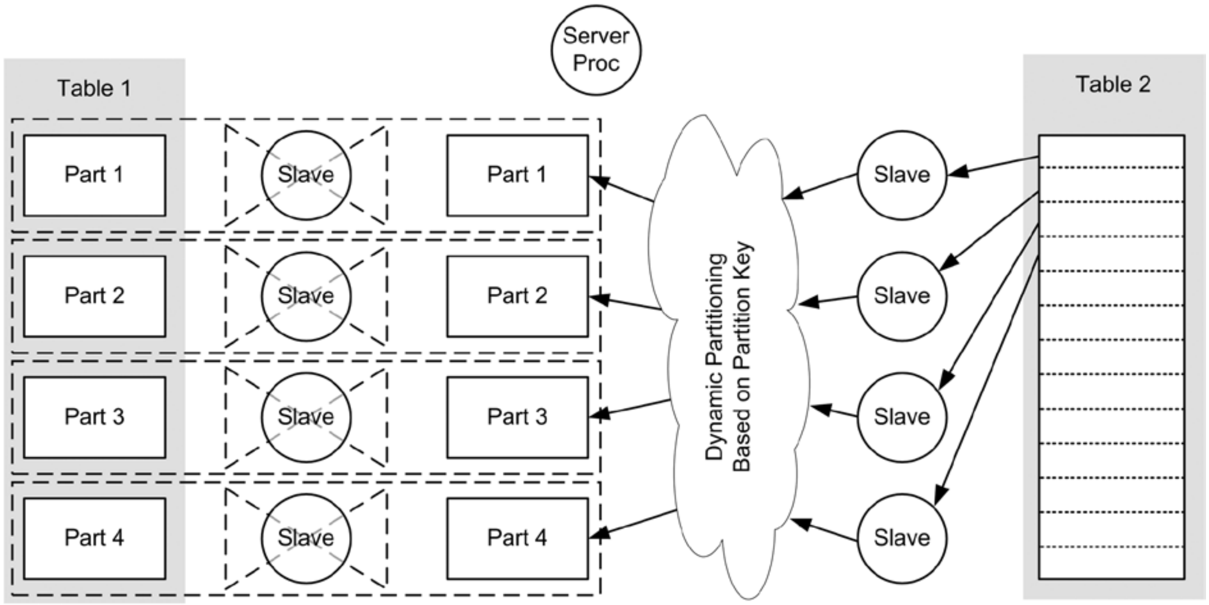


*Figure 14-16.* *Joining two tables with a partial partition-wise join*

To recognize whether a partial partition-wise join is used, it's necessary to look at the execution plan. If the PX SEND operation is of type PARTITION (KEY), it means that a partial partition-wise join is being used. In the following example, operation 7 provides that information:

```
SELECT /*+ ordered use_hash(t2p) pq_distribute(t2p none partition) */ *
FROM t1p, t2p
WHERE t1p.id = t2p.id

------------------------------------------------
| Id  | Operation                     | Name     |
------------------------------------------------
|   0 | SELECT STATEMENT              |          |
|   1 |  PX COORDINATOR               |          |
|   2 |   PX SEND QC (RANDOM)         | :TQ10001 |
|*  3 |    HASH JOIN BUFFERED         |          |
|   4 |     PX PARTITION HASH ALL     |          |
|   5 |      TABLE ACCESS FULL        | T1P      |
|   6 |     PX RECEIVE                |          |
|   7 |      PX SEND PARTITION (KEY)| :TQ10000 |
|   8 |       PX BLOCK ITERATOR       |          |
|   9 |        TABLE ACCESS FULL      | T2P      |
------------------------------------------------

   3 - access("T1P"."ID"="T2P"."ID")
```

In practice, partial partition-wise joins don't necessarily lead to improved performance. In fact, regular joins might be faster than partial partition-wise joins. Because using partial partition-wise joins can be detrimental to performance, you'll seldom see the query optimizer using this optimization technique.

## Star Transformation

The *star transformation* is an optimization technique used with *star schemas* (also known as dimensional models). This type of schema is composed of one large central table, the *fact table*, and of several other tables, the *dimension tables*. Its main characteristic is that the fact table references the dimension tables. Figure 14-17 is an example based on the sample schema SH (the *Sample Schemas* manual describes this fully).

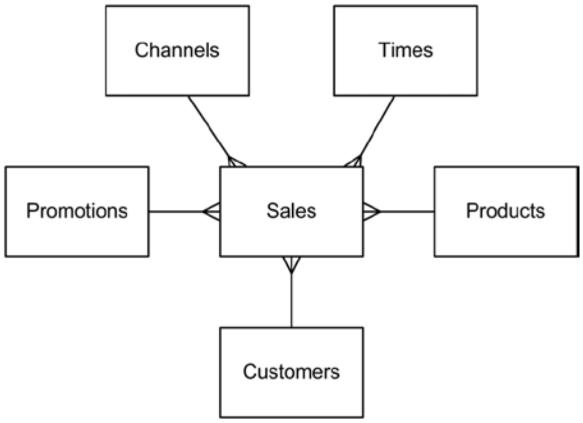


***Figure 14-17.*** *A typical star schema*

---

■ **Note** Star transformation is only available in Enterprise Edition. To take advantage of a similar optimization technique in Standard Edition, you have to rewrite the query yourself. At the end of this section I provide an example of such a rewrite.

---

The following is a typical query executed against a star schema (notice that no restrictions are applied to the fact table, but only on the dimension tables):

```
SELECT c.cust_state_province, t.fiscal_month_name, sum(s.amount_sold) AS amount_sold
FROM sales s, customers c, times t, products p
WHERE s.cust_id = c.cust_id
AND s.time_id = t.time_id
AND s.prod_id = p.prod_id
AND c.cust_year_of_birth BETWEEN 1970 AND 1979
AND p.prod_subcategory = 'Cameras'
GROUP BY c.cust_state_province, t.fiscal_month_name
ORDER BY c.cust_state_province, sum(s.amount_sold) DESC
```

To optimize this query against the star schema, the query optimizer should do the following:

1. Start evaluating each dimension table that has restrictions on it.
2. Assemble a list with the resulting dimension keys.
3. Use this list to extract the matching rows from the fact table.

Unfortunately, this approach can't be implemented with regular joins. On the one hand, the query optimizer can join only two data sets at one time. On the other hand, joining two dimension tables leads to a Cartesian product. To solve this problem, Oracle Database implements the *star transformation*.

---

■ **Caution** Although the star transformation was initially introduced with version 8.0 in 1997 and was strongly enhanced in version 8.1 two years later—that is, a long time ago—its stability has always been a problem. Probably every patchset released since its introduction has fixed bugs related to it. Whenever something goes wrong, errors like ORA-07445, ORA-00600, ORA-00942, or incorrect results are generated. That said, I have successfully used this feature since version 8.1.6. The Query Optimizer group at Oracle, aware of this issue, completely rewrote the code for version 11.2. Hence, in recent versions the stability is improved. My advice is simply to test it carefully. If it works, the improvement in performance will be considerable. If not, at least you would know it before going into production. I also advise you to check Oracle Support note 47358.1 (Init.ora Parameter STAR_TRANSFORMATION_ENABLED Reference Note). It gives you a list of the bugs affecting each specific version.

---

You need to meet two basic requirements to take advantage of the star transformation. First, the feature must be enabled. You use the `star_transformation_enabled` initialization parameter to control it. Note that, per default, the feature is disabled because the parameter is set to `FALSE`. To enable it, you should set it to either `temp_disable` or `TRUE`. Second, on the fact table, there must be an index for each join condition referencing a dimension table. The join conditions don't have to be based on foreign keys, but if foreign keys do exist, they assist the query optimizer in finding an optimal execution plan. Even though the query optimizer can convert B-tree indexes into bitmap indexes on the fly, execution plans using bitmap indexes are more efficient. In summary, for best performance I advise creating foreign keys and bitmap indexes on every join condition.

---

■ **Tip** A join condition between a fact table and a dimension table based on several columns isn't always supported by the star transformation. Therefore, I strongly recommend that you use use join conditions and, consequently, foreign keys and bitmap indexes based on a single column.

---

When the `star_transformation_enabled` initialization parameter is set to `temp_disable`, the following execution plan is used for the sample SQL statement shown previously. This example, like the following ones, is based on the `star_transformation.sql` script:

```
-------------------------------------------------------------------------
| Id  | Operation                               | Name                    |
-------------------------------------------------------------------------
|   0 | SELECT STATEMENT                        |                         |
|   1 |  SORT ORDER BY                          |                         |
|   2 |   HASH GROUP BY                         |                         |
|*  3 |    HASH JOIN                            |                         |
|   4 |     TABLE ACCESS FULL                   | TIMES                   |
|*  5 |     HASH JOIN                           |                         |
|*  6 |      TABLE ACCESS FULL                  | CUSTOMERS               |
|   7 |      VIEW                               | VW_ST_FE4FBDB9          |
|   8 |       NESTED LOOPS                      |                         |
|   9 |        BITMAP CONVERSION TO ROWIDS      |                         |
|  10 |         BITMAP AND                      |                         |
|  11 |          BITMAP MERGE                   |                         |
|  12 |           BITMAP KEY ITERATION          |                         |
|  13 |            TABLE ACCESS BY INDEX ROWID| PRODUCTS                |
|* 14 |             INDEX RANGE SCAN            | PRODUCTS_PROD_SUBCAT_IX |
|* 15 |            BITMAP INDEX RANGE SCAN      | SALES_PROD_BIX          |
|  16 |          BITMAP MERGE                   |                         |
|  17 |           BITMAP KEY ITERATION          |                         |
|* 18 |            TABLE ACCESS FULL            | CUSTOMERS               |
|* 19 |            BITMAP INDEX RANGE SCAN      | SALES_CUST_BIX          |
|  20 |        TABLE ACCESS BY USER ROWID       | SALES                   |
-------------------------------------------------------------------------

   3 - access("ITEM_1"="T"."TIME_ID")
   5 - access("ITEM_2"="C"."CUST_ID")
   6 - filter("C"."CUST_YEAR_OF_BIRTH">=1970 AND "C"."CUST_YEAR_OF_BIRTH"<=1979)
  14 - access("P"."PROD_SUBCATEGORY"='Cameras')
  15 - access("S"."PROD_ID"="P"."PROD_ID")
  18 - filter("C"."CUST_YEAR_OF_BIRTH">=1970 AND "C"."CUST_YEAR_OF_BIRTH"<=1979)
  19 - access("S"."CUST_ID"="C"."CUST_ID")
```

Because this execution plan contains some peculiar operations, let's take a detailed look at its operation:

1. The execution starts with operation 4, the full scan of the `times` dimension table. With the data returned from it, a hash table is built by operation 3, the hash join.
2. Operation 6 does a full scan of the `customers` dimension table and applies the `c.cust_year_of_birth BETWEEN 1970 AND 1979` restriction. With the data returned from it, a hash table is built by operation 5, the hash join.
3. Operations 13 and 14 access the `products` dimension table and apply the `p.prod_subcategory='Cameras'` restriction.
4. Operation 12, `BITMAP KEY ITERATION`, is a related-combine operation. For each row returned by its first child (operation 13), the second child (operation 15) is executed once. In this case, a lookup based on the bitmap index defined on the fact table is performed.

5. Operation 11, BITMAP MERGE, merges the bitmaps passed to it by its child operation. This operation is necessary because one index key from a bitmap index might cover only part of the indexed table.
6. Operations 16 to 19 process the customers dimension table in the same way as the products dimension table is processed by operations 11 to 15. In fact, every dimension that a restriction is applied to is processed in the same way.
7. Operation 10, BITMAP AND, combines the bitmaps passed from its two child operations (11 and 16) and keeps only the matching entries.
8. Operation 9, BITMAP CONVERSION TO ROWIDS, converts the bitmaps passed from its child operation (10) in rowids of the sales fact table.
9. Operation 20, because of the nested loops join (operation 8) accesses the fact table with the rowids generated by operation 9.
10. Operation 7 is merely informative, telling you that the query block contains an unmergeable view resulting from a star transformation (notice the VW_ST prefix as a view name). This operation is available only from version 11.2.0.2 onward.
11. With the rows returned by operation 8, the hash tables of the two hash joins (operations 3 and 5) are probed. If matching rows are found, they're passed to operation 2.
12. Operation 2, HASH GROUP BY, processes the GROUP BY clause and passes the resulting rows to operation 1.
13. Finally, operation 1, SORT ORDER BY, processes the ORDER BY clause.

In summary, the following steps are performed to execute a star transformation:

1. The dimension tables are "joined" to the corresponding bitmap index on the fact table. This operation is necessary only for the dimension tables that have restrictions applied to them—in this case, the products and customers tables.
2. The resulting bitmaps are merged and converted to rowids. Then the fact table is accessed through the rowids.
3. The dimension tables are joined to the data selected from the fact table. This operation is necessary only for the dimensions that have columns referenced outside the WHERE clause—in this case, for the times and customers tables. This is why the customers table appears twice in the execution plan.

You can apply two additional optimization techniques to this basic behavior: temporary tables and bitmap-join indexes.

The purpose of temporary tables is to avoid the double processing of dimension tables. For example, in the previous execution plan, not only is the customers dimension table accessed twice with a full scan (operations 6 and 18), but the predicate applied to it is also executed twice. The idea is to access each dimension only once, apply the predicates, and store the resulting row in a temporary table. This optimization technique is enabled when the star_transformation_enabled initialization parameter is set to TRUE. The following execution plan, which is based on the same SQL statement as before, is an example. Notice the creation of the sys_temp_0fd9d6647_1cb85c temporary table (operations from 1 to 3) and its utilization (operations 7 and 21):

```
---------------------------------------------------------------------------
| Id  | Operation                         | Name                        |
---------------------------------------------------------------------------
|   0 | SELECT STATEMENT                  |                             |
|   1 |  TEMP TABLE TRANSFORMATION        |                             |
|   2 |   LOAD AS SELECT                  | SYS_TEMP_0FD9D6647_1CB85C |
|*  3 |    TABLE ACCESS FULL              | CUSTOMERS                   |
|   4 |   SORT ORDER BY                   |                             |
|   5 |    HASH GROUP BY                  |                             |
|*  6 |     HASH JOIN                     |                             |
|   7 |      TABLE ACCESS FULL            | SYS_TEMP_0FD9D6647_1CB85C |
|*  8 |      HASH JOIN                    |                             |
|   9 |       TABLE ACCESS FULL           | TIMES                       |
|  10 |       VIEW                        | VW_ST_16AF99B7              |
|  11 |        NESTED LOOPS               |                             |
|  12 |         BITMAP CONVERSION TO ROWIDS |                           |
|  13 |          BITMAP AND               |                             |
|  14 |           BITMAP MERGE            |                             |
|  15 |            BITMAP KEY ITERATION   |                             |
|  16 |             TABLE ACCESS BY INDEX ROWID| PRODUCTS               |
|* 17 |              INDEX RANGE SCAN     | PRODUCTS_PROD_SUBCAT_IX     |
|* 18 |             BITMAP INDEX RANGE SCAN | SALES_PROD_BIX            |
|  19 |           BITMAP MERGE            |                             |
|  20 |            BITMAP KEY ITERATION   |                             |
|  21 |             TABLE ACCESS FULL     | SYS_TEMP_0FD9D6647_1CB85C |
|* 22 |             BITMAP INDEX RANGE SCAN | SALES_CUST_BIX            |
|  23 |         TABLE ACCESS BY USER ROWID | SALES                      |
---------------------------------------------------------------------------

   3 - filter("C"."CUST_YEAR_OF_BIRTH">=1970 AND "C"."CUST_YEAR_OF_BIRTH"<=1979)
   6 - access("ITEM_2"="C0")
   8 - access("ITEM_1"="T"."TIME_ID")
  17 - access("P"."PROD_SUBCATEGORY"='Cameras')
  18 - access("S"."PROD_ID"="P"."PROD_ID")
  22 - access("S"."CUST_ID"="C0")
```

The second optimization technique is based on bitmap-join indexes. The idea is to avoid the “join” between the dimension tables and the corresponding bitmap index on the fact tables. For this purpose, the bitmap-join indexes must be created on the fact table and index one or several columns of the dimension tables. For example, the following indexes are necessary to apply the restrictions `c.cust_year_of_birth BETWEEN 1970 AND 1979` and `p.prod_ subcategory = 'Cameras'`, respectively:

```
CREATE BITMAP INDEX sales_cust_year_of_birth_bix ON sales (c.cust_year_of_birth)
FROM sales s, customers c
WHERE s.cust_id = c.cust_id

CREATE BITMAP INDEX sales_prod_subcategory_bix ON sales (p.prod_subcategory)
FROM sales s, products p
WHERE s.prod_id = p.prod_id
```

With these two indexes in place, the following execution plan results. Note that the method used to produce the rowids (lines 8 to 12) is much more straightforward than the one used in the previous examples. Actually, instead of accessing the dimension tables and joining them to the bitmap indexes on the fact table, it's enough to access the bitmap-join indexes. This is possible because the value of the associated dimension row is already present in the bitmap-join index of the fact table:

```
-------------------------------------------------------------------------
| Id  | Operation                          | Name                        |
-------------------------------------------------------------------------
|   0 | SELECT STATEMENT                   |                             |
|   1 |  SORT ORDER BY                     |                             |
|   2 |   HASH GROUP BY                    |                             |
|*  3 |    HASH JOIN                       |                             |
|   4 |     TABLE ACCESS FULL              | TIMES                       |
|*  5 |     HASH JOIN                      |                             |
|*  6 |      TABLE ACCESS FULL             | CUSTOMERS                   |
|   7 |      TABLE ACCESS BY INDEX ROWID   | SALES                       |
|   8 |       BITMAP CONVERSION TO ROWIDS|                             |
|   9 |        BITMAP AND                  |                             |
|* 10 |         BITMAP INDEX SINGLE VALUE| SALES_PROD_SUBCATEGORY_BIX  |
|  11 |         BITMAP MERGE               |                             |
|* 12 |          BITMAP INDEX RANGE SCAN | SALES_CUST_YEAR_OF_BIRTH_BIX |
-------------------------------------------------------------------------

   3 - access("S"."TIME_ID"="T"."TIME_ID")
   5 - access("S"."CUST_ID"="C"."CUST_ID")
   6 - filter("C"."CUST_YEAR_OF_BIRTH">=1970 AND "C"."CUST_YEAR_OF_BIRTH"<=1979)
  10 - access("S"."SYS_NC00009$"='Cameras')
  12 - access("S"."SYS_NC00008$">=1970 AND "S"."SYS_NC00008$"<=1979)
```

The star transformation is a cost-based query transformation. Therefore, when enabled, the query optimizer decides not only whether it makes sense to use a star transformation, but also whether temporary tables and/or bitmap-join indexes are useful for efficient SQL statement execution. The utilization of this feature can also be controlled with the `star_transformation` and `no_star_transformation` hints.

If you're working in Standard Edition, neither the star transformation nor bitmap indexes are available. In that case, to have decent performance you might want to rewrite the query yourself. Even though, as the following example shows, the rewritten query is much less readable, the execution plan is quite similar:

```
SELECT c.cust_state_province, t.fiscal_month_name, sum(s.amount_sold) AS amount_sold
FROM (SELECT *
      FROM sales
      WHERE rowid IN (SELECT c.rid
                      FROM (SELECT s.rowid AS rid
                            FROM customers c, sales s
                            WHERE c.cust_id = s.cust_id
                            AND c.cust_year_of_birth BETWEEN 1970 AND 1979) c,
                            (SELECT s.rowid AS rid
                            FROM products p, sales s
                            WHERE p.prod_id = s.prod_id
                            AND p.prod_subcategory = 'Cameras') p
                      WHERE c.rid = p.rid)) s,
     customers c, times t
```

```
WHERE s.cust_id = c.cust_id
AND s.time_id = t.time_id
GROUP BY c.cust_state_province, t.fiscal_month_name
ORDER BY c.cust_state_province, sum(s.amount_sold) DESC

-------------------------------------------------------------------------
| Id  | Operation                             | Name                    |
-------------------------------------------------------------------------
|   0 | SELECT STATEMENT                      |                         |
|   1 |  SORT ORDER BY                        |                         |
|   2 |   HASH GROUP BY                       |                         |
|*  3 |    HASH JOIN                          |                         |
|   4 |     TABLE ACCESS FULL                 | TIMES                   |
|*  5 |     HASH JOIN                         |                         |
|   6 |      TABLE ACCESS FULL                | CUSTOMERS               |
|   7 |      VIEW                             |                         |
|   8 |       NESTED LOOPS                    |                         |
|   9 |        VIEW                           | VW_NSO_1                |
|  10 |         HASH UNIQUE                   |                         |
|* 11 |          HASH JOIN                    |                         |
|  12 |           VIEW                        |                         |
|  13 |            NESTED LOOPS               |                         |
|  14 |             TABLE ACCESS BY INDEX ROWID| PRODUCTS               |
|* 15 |              INDEX RANGE SCAN         | PRODUCTS_PROD_SUBCAT_IX |
|* 16 |             INDEX RANGE SCAN          | SALES_PROD_BIX          |
|  17 |           VIEW                        |                         |
|  18 |            NESTED LOOPS               |                         |
|* 19 |             TABLE ACCESS FULL         | CUSTOMERS               |
|* 20 |             INDEX RANGE SCAN          | SALES_CUST_BIX          |
|  21 |        TABLE ACCESS BY USER ROWID     | SALES                   |
-------------------------------------------------------------------------

   3 - access("S"."TIME_ID"="T"."TIME_ID")
   5 - access("S"."CUST_ID"="C"."CUST_ID")
  11 - access("C"."RID"="P"."RID")
  15 - access("P"."PROD_SUBCATEGORY"='Cameras')
  16 - access("P"."PROD_ID"="S"."PROD_ID")
  19 - filter("C"."CUST_YEAR_OF_BIRTH">=1970 AND
              "C"."CUST_YEAR_OF_BIRTH"<=1979)
  20 - access("C"."CUST_ID"="S"."CUST_ID")
```

## On to Chapter 15

This chapter describes two main subjects related to joins. First, it covers the methods used by the database engine to perform joins (nested loops joins, merge joins, and hash joins) and talks about when it makes sense to use each of them. Second, it covers some optimization techniques that the query optimizer applies to improve the performance.

Now that I've discussed the basic access path and join methods, it's time to look at advanced optimization techniques. In the next chapter, I discuss materialized views, result caching, parallel processing, and direct-path inserts. All of these features aren't used that often, but when correctly applied, they can greatly improve performance. It's time to go beyond accesses and join optimization.

CHAPTER 15

# Beyond Data Access and Join Optimization

The optimization of data accesses and joins must be performed before considering the advanced optimization techniques presented in this chapter. In fact, the optimization techniques described here are intended only to further improve performance when it isn't possible to achieve it otherwise. In other words, you should fix the basics first, and then, if the performance is still not acceptable, you can consider special means.

This chapter describes how materialized views, result caches, parallel processing, direct-path inserts, row prefetching, and the array interface work and how they can be used to improve performance. Each section that describes an optimization technique is organized in the same way. A short introduction is followed by a description of how the technique works and when you should use it. All sections end with a discussion of some common pitfalls and fallacies.

**Note** In this chapter, several SQL statements contain hints in order to show you examples of their utilization. In any case, neither real references nor full syntaxes are provided. You can find these in Chapter 2 of the *Oracle Database SQL Language Reference* manual.

This chapter shows the results of different performance tests. The performance figures are intended only to help you compare different kinds of processing and to give you a feel for their impact. Remember, every system and every application has its own characteristics. Therefore, the relevance of using each technique might be very different, depending on where it's applied.

## Materialized View

A *view* is a virtual table based on the result set returned by the query specified at view creation time. Every time a view is accessed, the query is executed. To avoid executing the query for every access, the result set of the query can be stored in a *materialized view*. In other words, materialized views simply transform and duplicate data that is already stored elsewhere.

**Note** Materialized views can also be used in distributed environments in order to replicate data between databases. This usage isn't covered in this book.

## How It Works

The following sections describe what a materialized view is and how it works. After describing the concepts that materialized views are based on, query rewrite and refreshes are covered in detail.

### Concepts

Let's say you have to improve the performance of the following query (available in the `mv.sql` script), which is based on the sample schema `sh` (the *Oracle Database Sample Schemas* manual describes this fully):

```
SELECT p.prod_category, c.country_id,
       sum(quantity_sold) AS quantity_sold,
       sum(amount_sold) AS amount_sold
FROM sales s, customers c, products p
WHERE s.cust_id = c.cust_id
AND s.prod_id = p.prod_id
GROUP BY p.prod_category, c.country_id
ORDER BY p.prod_category, c.country_id
```

If you judge the efficiency of the execution plan by applying the methods and rules described in Chapters 10 and 13, you'll find that everything is fine. The estimations are excellent, and the number of logical reads per returned row of the different access paths is very low:

```
-----------------------------------------------------------------------------
| Id  | Operation                 | Name      | Starts | E-Rows | A-Rows | Buffers |
-----------------------------------------------------------------------------
|   0 | SELECT STATEMENT          |           |      1 |        |     81 |    3094 |
|   1 |  SORT GROUP BY            |           |      1 |     68 |     81 |    3094 |
|*  2 |   HASH JOIN               |           |      1 |    968 |    956 |    3094 |
|   3 |    TABLE ACCESS FULL      | PRODUCTS  |      1 |     72 |     72 |       3 |
|   4 |    VIEW                   | VW_GBC_9  |      1 |    968 |    956 |    3091 |
|   5 |     HASH GROUP BY         |           |      1 |    968 |    956 |    3091 |
|*  6 |      HASH JOIN            |           |      1 |   918K|   918K|    3091 |
|   7 |       TABLE ACCESS FULL   | CUSTOMERS |      1 |  55500 |  55500 |    1456 |
|   8 |       PARTITION RANGE ALL|           |      1 |   918K|   918K|    1635 |
|   9 |        TABLE ACCESS FULL | SALES     |     28 |   918K|   918K|    1635 |
-----------------------------------------------------------------------------

   2 - access("ITEM_1"="P"."PROD_ID")
   6 - access("S"."CUST_ID"="C"."CUST_ID")
```

The "problem" is that lots of data is processed before the aggregation takes place. The performance can't be improved by just changing an access path or a join method, because they're already as optimal as they can be; in other words, their full potential is already exploited. It's time then to apply an advanced optimization technique. Let's create a materialized view based on the query to be optimized.

A materialized view is created with the `CREATE MATERIALIZED VIEW` statement. In the simplest case, you have to specify a name and the query on which the materialized view is based. Note that the tables on which the materialized view is based are called *base tables* (they're also known as *master tables*). The following SQL statement and Figure 15-1 illustrate this (notice that the ORDER BY clause used in the original query is omitted):

```
CREATE MATERIALIZED VIEW sales_mv
AS
SELECT p.prod_category, c.country_id,
       sum(quantity_sold) AS quantity_sold,
       sum(amount_sold) AS amount_sold
FROM sales s, customers c, products p
WHERE s.cust_id = c.cust_id
AND s.prod_id = p.prod_id
GROUP BY p.prod_category, c.country_id
```

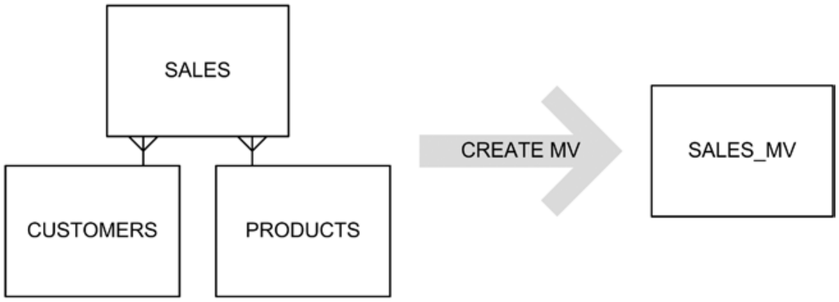


***Figure 15-1.** Creation of a materialized view*

■ **Note** When you create a materialized view based on a query containing the ORDER BY clause, the rows are sorted according to the ORDER BY clause only during the creation of the materialized view. Later, during refreshes, this sorting criterion isn't maintained. This is also because the ORDER BY clause isn't included in the materialized view's definition that is stored in the data dictionary.

When you execute the previous SQL statement, the database engine creates a materialized view (which is only an object in the data dictionary—in other words, it's only metadata) and a *container table*. The container table is a regular heap table that has the same name as the materialized view. It's used to store the result set returned by the query.

You can query the container table as you would with any other table. The following SQL statement shows an example of this:

```
SELECT *
FROM sales_mv
ORDER BY prod_category, country_id

-------------------------------------------------------------------------------
| Id  | Operation              | Name     | Starts | E-Rows | A-Rows | Buffers |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT       |          |      1 |        |     81 |       3 |
|   1 |  SORT ORDER BY         |          |      1 |     81 |     81 |       3 |
|   2 |   MAT_VIEW ACCESS FULL| SALES_MV |      1 |     81 |     81 |       3 |
-------------------------------------------------------------------------------
```

Notice that the number of logical reads, compared to the original query, has dropped from 3,094 to 3. Also notice that the MAT_VIEW ACCESS FULL access path clearly states that a materialized view is accessed. This access path operates as a TABLE ACCESS FULL. It's merely a naming convention used to conveniently point out that a materialized view is being used. For all practical purposes, the two access paths are absolutely the same.

Directly referencing the container table is always an option. But, if you want to improve the performance of an application without modifying the SQL statements it executes, there's a second powerful possibility: use *query rewrite*.

---

■ **Note** Query rewrite is available only in Enterprise Edition.

---

The concept of query rewrite is straightforward. When the query optimizer receives a query to be optimized, it can decide to use it as is (in other words, to *not* use query rewrite), or it can choose to rewrite it so as to use a materialized view that contains all, or part of, the data required to execute the query. Figure 15-2 illustrates this. The decision, of course, is based on the cost estimated by the query optimizer for the execution plans, with and without query rewrite. The execution plan with the lower cost is used to execute the query. The rewrite and no_rewrite hints are available to control the query optimizer's decisions.

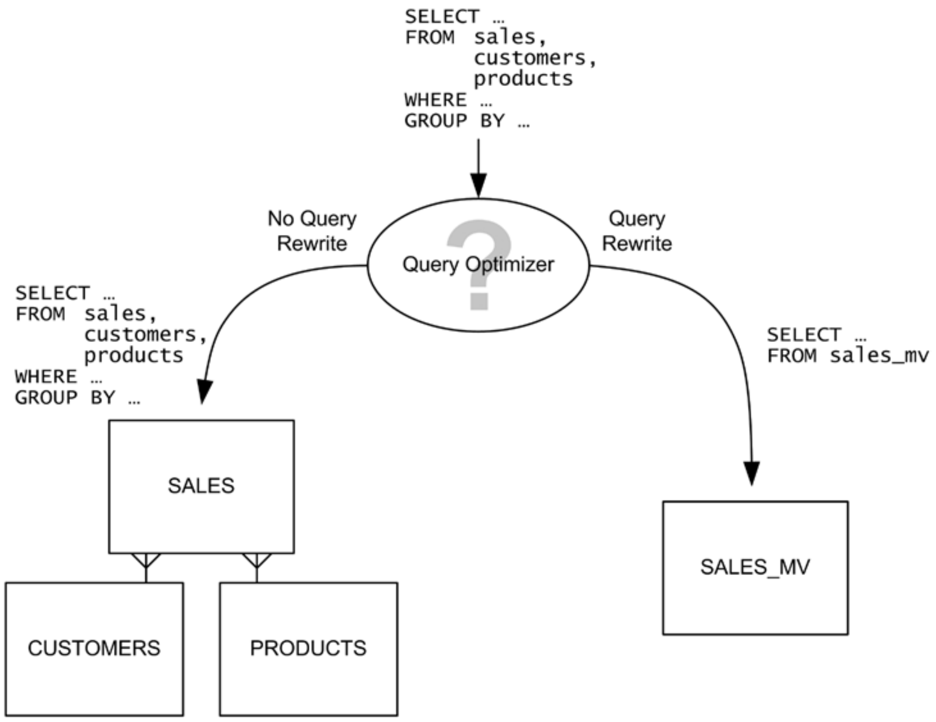


***Figure 15-2.*** *The query optimizer can use query rewrite to automatically use a materialized view*

To take advantage of query rewrite, it must be enabled at two levels. First, you have to set the query_rewrite_enabled initialization parameter to TRUE. Second, you have to enable it for the materialized view. The following SQL statement shows how to enable query rewrite for a materialized view that already exists:

```
ALTER MATERIALIZED VIEW sales_mv ENABLE QUERY REWRITE
```

Once query rewrite is enabled, if you submit the original query, the query optimizer considers the materialized view as a candidate for query rewrite. In this case, the query optimizer does, in fact, rewrite the query to use the materialized view. Notice that the MAT_VIEW REWRITE ACCESS FULL access path clearly states that query rewrite takes place:

```
SELECT p.prod_category, c.country_id,
       sum(quantity_sold) AS quantity_sold,
       sum(amount_sold) AS amount_sold
FROM sales s, customers c, products p
WHERE s.cust_id = c.cust_id
AND s.prod_id = p.prod_id
GROUP BY p.prod_category, c.country_id
ORDER BY p.prod_category, c.country_id

---------------------------------------------------------------------------------------
| Id  | Operation                       | Name     | Starts | E-Rows | A-Rows | Buffers |
---------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                |          |      1 |        |     81 |       3 |
|   1 |  SORT ORDER BY                  |          |      1 |     81 |     81 |       3 |
|   2 |   MAT_VIEW REWRITE ACCESS FULL| SALES_MV |      1 |     81 |     81 |       3 |
---------------------------------------------------------------------------------------
```

In summary, with query rewrite, the query optimizer can automatically use a materialized view that contains the data required to execute a query. As an analogy, it's similar to what happens when you add an index to a table. You (usually) don't have to modify the SQL statements to take advantage of it. Thanks to the data dictionary, the query optimizer knows that such an index exists, and if it's useful for executing a SQL statement more efficiently, the query optimizer will use it. The same goes for materialized views.

When the base tables are modified through DML or DDL statements, the materialized view (actually, the container table) may contain *stale* data ("stale" means "old"—that is, data that's no longer equal to the result set of the materialized view query, if executed on the new content of the base tables now). As a result, the database engine stops using the materialized view for query rewrite. For this reason, as shown in Figure 15-3, after modifying the base tables, a *refresh* of the materialized view has to be performed. You can choose how and when the refresh of a materialized view is performed.

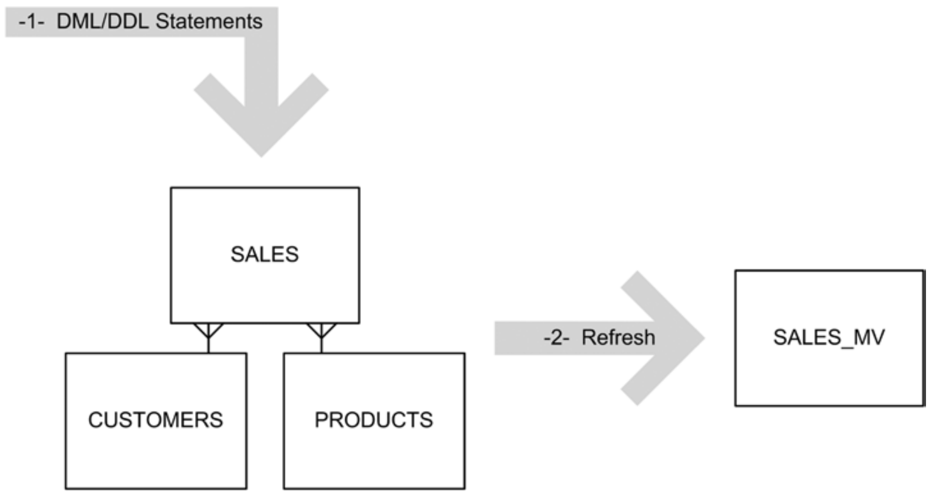


***Figure 15-3.*** *After modifying the base tables, the materialized view has to be refreshed*

Now that I've introduced the basic concepts, let me describe, in more detail, which parameters can be specified during the creation of materialized views, and how query rewrite and refreshes work.

## Parameters

As you saw in the previous section, you can create a materialized view without specifying parameters. However, you can fully customize its creation:

- You can specify physical properties such as partitioning, compression, tablespace, and storage parameters for the container table. In this regard, the container table is handled like any other heap table. Owing to this, you can apply the techniques discussed in Chapter 13 to further optimize data access.
- When the materialized view is created, the query is executed, and the result set is inserted into the container table. This is because the `build immediate` parameter is used by default. Two additional possibilities exist: first, to defer the insertion of the rows to the first refresh by specifying the `build deferred` parameter, and second, to reuse an already existing table as the container table by specifying the `on prebuilt table` parameter.
- By default, query rewrite is disabled. To enable it, you must specify the `enable query rewrite` parameter. Enabling query rewrite is supported in Enterprise Edition only.
- To improve the performance of fast refreshes (described later in this chapter), an index is created on the container table by default. To suppress the creation of this index, you can specify the `using no index` parameter. This is useful, for example, for avoiding index maintenance overhead, which might be far from negligible, if you never want to perform fast refreshes.

The following SQL statement shows an example that's based on the same query as earlier, but where several of the parameters that have just been described are specified:

```
CREATE MATERIALIZED VIEW sales_mv
PARTITION BY HASH (country_id) PARTITIONS 8
TABLESPACE users
BUILD IMMEDIATE
USING NO INDEX
ENABLE QUERY REWRITE
AS
SELECT p.prod_category, c.country_id,
       sum(quantity_sold) AS quantity_sold,
       sum(amount_sold) AS amount_sold
FROM sales s, customers c, products p
WHERE s.cust_id = c.cust_id
AND s.prod_id = p.prod_id
GROUP BY p.prod_category, c.country_id
```

In addition, you can also specify how the materialized view is refreshed. The "Materialized View Refreshes" section provides detailed information on this topic.

## Query Rewrite

The query optimizer is able to take advantage of query rewrite whenever a `SELECT` clause is present in a SQL statement, or, to be more specific, in the following cases:

- `SELECT ... FROM ...`
- `CREATE TABLE ... AS SELECT ... FROM ...`
- `INSERT INTO ... SELECT ... FROM ...`
- Subqueries

In addition, as already described, query rewrite is used only when two requirements are fulfilled. First, the `query_rewrite_enabled` initialization parameter must be set to `TRUE` (the default value). Second, the materialized view must be created with the `enable query rewrite` parameter.

Once these requirements are met, every time the query optimizer generates an execution plan, it has to find out whether a materialized view that contains the required data can be used to rewrite a SQL statement. For that purpose, it uses one of three methods:

- *Full-text-match query rewrite*: The text of the query passed to the query optimizer is compared to the text of the query of each available materialized view. If they match, the materialized view obviously contains the required data. Note that the comparison is less strict than the one commonly used by the database engine: it's case insensitive (except for literals) and ignores blank spaces (for example, new lines and tabs) and the `ORDER BY` clause.
- *Partial-text-match query rewrite*: The comparison is similar to the one used for full-text-match query rewrite. With this one, however, differences in the `SELECT` clause are permitted. For example, if the materialized view stores three columns and only two of them are referenced by the query to be optimized, the materialized view contains all the required data, and therefore a query rewrite is possible.

- *General query rewrite*: To find a matching materialized view, general query rewrite does a semantic analysis. For that purpose, it makes extensive use of constraints and dimensions to infer the semantic relations between data in the base tables. The purpose is to apply query rewrite even if the query passed to the query optimizer is quite different from the one associated with the matching materialized view. In fact, it's quite common for a well-designed materialized view to be used to rewrite many (and possibly quite different) SQL statements.

## DIMENSIONS

The query optimizer uses constraints stored in the data dictionary to infer data relations that enable general query rewrite to be used to the fullest extent possible. Sometimes, other very useful relationships, not covered by constraints, exist between columns in the same table or even in different tables. This is especially true for denormalized tables (such as the `times` table in the `sh` schema). To provide such information to the query optimizer, it's possible to use a *dimension*. Because of it, it's possible to specify 1:n relations with *hierarchies* and 1:1 relations with *attributes*. Both hierarchies and attributes are based on *levels*, which are, simply put, columns in a table. The following SQL statement illustrates this:

```
CREATE DIMENSION times_dim
LEVEL day IS times.time_id
LEVEL month IS times.calendar_month_desc
LEVEL quarter IS times.calendar_quarter_desc
LEVEL year IS times.calendar_year
HIERARCHY cal_rollup (day CHILD OF month CHILD OF quarter CHILD OF year)
ATTRIBUTE day DETERMINES (day_name, day_number_in_month)
ATTRIBUTE month DETERMINES (calendar_month_number, calendar_month_name)
ATTRIBUTE quarter DETERMINES (calendar_quarter_number)
ATTRIBUTE year DETERMINES (days_in_cal_year)
```

Detailed information about dimensions is available in the *Oracle Database Data Warehousing Guide* manual.

Full-text-match and partial-text-match query rewrites can be applied very quickly. But because their decisions are based on a simple text match, they aren't very flexible. As a result, they can only rewrite a limited number of queries. In contrast, general query rewrite is much more powerful. The downside is that the overhead of applying it is much higher. For this reason, the query optimizer applies the methods in an increasing order of complexity (and thereby parse overhead) until a matching materialized view is found. This process is illustrated in Figure 15-4.

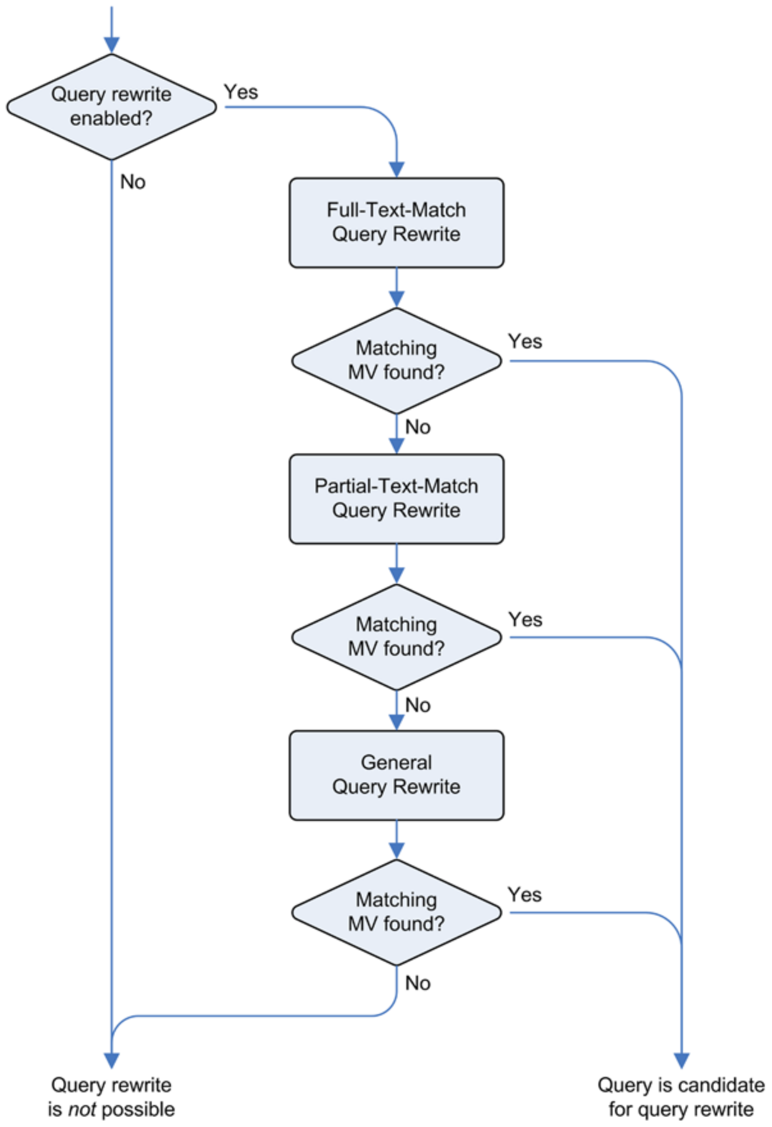

Query rewrite enabled?
Yes
No
Full-Text-Match Query Rewrite
Matching MV found?
Yes
No
Partial-Text-Match Query Rewrite
Matching MV found?
Yes
No
General Query Rewrite
Matching MV found?
Yes
No
Query rewrite is *not* possible
Query is candidate for query rewrite

***Figure 15-4.*** *The query rewrite process*

The following example, based on the `mv_rewrite.sql` script, shows general query rewrite in action. Notice that the query is similar to the one used in the previous section to define the `sales_mv` materialized view. There are five differences:

- The `SELECT` clause is different. Note, however, that the materialized view contains all the necessary data.
- The `FROM` clause is written using the newer join syntax (notice the `JOIN` and `ON` keywords).
- The `GROUP BY` clause is different. Data is aggregated on fewer columns (`country_id` is missing compared to the materialized view definition).

- An ORDER BY clause is specified.
- The customers table isn't referenced. However, thanks to a validated foreign key constraint on the sales table that references the customers table, the query optimizer can determine that there is no loss of data by omitting that join (since the join can't eliminate any row).

Regardless of these differences, with general query rewrite, the query optimizer can take advantage of the sales_mv materialized view:

```
SQL> SELECT upper(p.prod_category) AS prod_category,
  2         sum(s.amount_sold) AS amount_sold
  3  FROM sales s JOIN products p ON s.prod_id = p.prod_id
  4  GROUP BY p.prod_category
  5  ORDER BY p.prod_category;

--------------------------------------------------
| Id  | Operation                      | Name     |
--------------------------------------------------
|   0 | SELECT STATEMENT               |          |
|   1 |  SORT GROUP BY                 |          |
|   2 |   MAT_VIEW REWRITE ACCESS FULL| SALES_MV |
--------------------------------------------------
```

It's important to note that per default, the query optimizer doesn't use constraints that aren't validated. As a result, if nonvalidated constraints exist, the query optimizer can't use general query rewrite. Because, with this specific query, full-text-match query rewrite and partial-text-match query rewrite can't be used, no query rewrite occurs. The following example illustrates this. Notice that, except for the FROM clause (but, as seen before, this detail is irrelevant), this is the same query used in the previous example. However, the status of the sales_customer_fk constraint is changed:

```
SQL> ALTER TABLE sales MODIFY CONSTRAINT sales_customer_fk NOVALIDATE;

SQL> SELECT upper(p.prod_category) AS prod_category,
  2         sum(s.amount_sold) AS amount_sold
  3  FROM sales s, products p
  4  WHERE s.prod_id = p.prod_id
  5  GROUP BY p.prod_category
  6  ORDER BY p.prod_category;

--------------------------------------------
| Id  | Operation                | Name     |
--------------------------------------------
|   0 | SELECT STATEMENT         |          |
|   1 |  SORT GROUP BY           |          |
|*  2 |   HASH JOIN              |          |
|   3 |    VIEW                  | VW_GBC_5 |
|   4 |     HASH GROUP BY        |          |
|   5 |      PARTITION RANGE ALL|          |
|   6 |       TABLE ACCESS FULL | SALES    |
|   7 |    TABLE ACCESS FULL     | PRODUCTS |
--------------------------------------------

   2 - access("ITEM_1"="P"."PROD_ID")
```

Especially for data marts, it isn't uncommon to use constraints that, although not validated by the database engine, are known to be fulfilled by the data, thanks to the way the tables' data is (carefully) maintained. At the same time, it's also not uncommon to have materialized views that, although considered stale by the database engine, are known to be safe for rewriting queries.

To take advantage of general query rewrites in such situations, you can use the `query_rewrite_integrity` initialization parameter. With it, you can specify whether only enforced constraints (and therefore, validated by the database engine) are to be used and whether a materialized view containing stale data is to be used. The parameter can be set to the following three values:

- `enforced`: Only materialized views containing fresh data are considered for query rewrite. Note that materialized views based on external tables are always considered stale. In addition, only validated constraints are used for general query rewrite. This is the default value.
- `trusted`: Only materialized views containing fresh data are considered for query rewrite. In addition, dimensions and constraints that aren't validated and marked with `rely` are trusted for general query rewrite.
- `stale_tolerated`: All existing materialized views, including those with stale data, are considered for query rewrite. In addition, dimensions and constraints that aren't validated and marked with `rely` are trusted for general query rewrite.

The following example shows how to use general query rewrite without validating the constraint. As shown, the constraint is marked with `rely`, and the integrity level is set to `trusted`:

```
SQL> ALTER TABLE sales MODIFY CONSTRAINT sales_customer_fk RELY;

SQL> ALTER SESSION SET query_rewrite_integrity = trusted;

SQL> SELECT upper(p.prod_category) AS prod_category,
  2         sum(s.amount_sold) AS amount_sold
  3  FROM sales s, products p
  4  WHERE s.prod_id = p.prod_id
  5  GROUP BY p.prod_category
  6  ORDER BY p.prod_category;

--------------------------------------------------
| Id  | Operation                     | Name     |
--------------------------------------------------
|   0 | SELECT STATEMENT              |          |
|   1 |  SORT GROUP BY                |          |
|   2 |   MAT_VIEW REWRITE ACCESS FULL| SALES_MV |
--------------------------------------------------
```

If you're in trouble because one of your SQL statements isn't using query rewrite and you don't understand why, you can use the `explain_rewrite` procedure in the `dbms_mview` package to find out what the problem is. The following PL/SQL block is an example of how it's used. Notice that the `query` parameter specifies the query that should be rewritten, the `mv` parameter specifies the materialized view that should be used for the rewrite, and the `statement_id` parameter specifies an arbitrary character string that's used to identify the information stored in the output table `rewrite_table`:

```
SQL> ALTER SESSION SET query_rewrite_integrity = enforced;

SQL> DECLARE
```

```
  2    l_query CLOB := 'SELECT upper(p.prod_category) AS prod_category,
  3                            sum(s.amount_sold) AS amount_sold
  4                     FROM sh.sales s, sh.products p
  5                     WHERE s.prod_id = p.prod_id
  6                     GROUP BY p.prod_category
  7                     ORDER BY p.prod_category';
  8  BEGIN
  9    DELETE rewrite_table WHERE statement_id = '42';
 10    dbms_mview.explain_rewrite(
 11      query        => l_query,
 12      mv           => 'sh.sales_mv',
 13      statement_id => '42'
 14    );
 15  END;
 16  /
```

---

■ **Note** The `rewrite_table` table doesn't exist per default. You can create it in the schema used for the analysis by executing the `utlxrw.sql` script stored under `$ORACLE_HOME/rdbms/admin`.

---

The output of the procedure, provided in the `rewrite_table` table, gives the reasons why the query rewrite doesn't happen. The output is composed of messages that are documented in the *Oracle Database Error Messages* manual. Here's the output of the previous analysis:

```
SQL> SELECT message
  2  FROM rewrite_table
  3  WHERE statement_id = '42';

MESSAGE
---------------------------------------------------------------------------------------------
QSM-01150: query did not rewrite
QSM-01284: materialized view SALES_MV has an anchor table CUSTOMERS not found in query
QSM-01052: referential integrity constraint on table, SALES, not VALID in ENFORCED integrity
mode
```

It's also essential to understand that not all query rewrite methods can be applied to all materialized views. Certain materialized views support full-text-match query rewrites only. Others support only full-text-match and partial-text-match query rewrites. In general, as the complexity (given, for example, by the utilization of constructs such as set operators and hierarchical queries) of materialized views increases, less often are advanced query rewrite methods supported. The restrictions also depend on the Oracle Database version. So, instead of providing a list of what is supported, I show you how to find out, given a specific case, which query rewrite methods are supported. To illustrate, let's re-create the materialized view with the following SQL statement. Notice that, compared to the previous examples, I have added only `p.prod_status` to the `GROUP BY` clause (in practice, executing such a SQL statement is usually pointless, but, as you'll see shortly, it's an easy way to partially deactivate query rewrite):

```
CREATE MATERIALIZED VIEW sales_mv
ENABLE QUERY REWRITE
AS
SELECT p.prod_category, c.country_id,
       sum(s.quantity_sold) AS quantity_sold,
       sum(s.amount_sold) AS amount_sold
```

```
FROM sales s, customers c, products p
WHERE s.cust_id = c.cust_id
AND s.prod_id = p.prod_id
GROUP BY p.prod_category, c.country_id, p.prod_status
```

To display the query rewrite methods supported by a materialized view, you can query the `user_mviews` view (the `dba`, `all`, and, in a 12.1 multitenant environment, `cdb` related views can also be used) as shown in the following example. In this case, according to the `rewrite_enabled` column, query rewrite is enabled at the materialized view level, and according to the `rewrite_capability` column, only text-match query rewrite is supported (in other words, general query rewrite isn't):

```
SQL> SELECT rewrite_enabled, rewrite_capability
  2  FROM user_mviews
  3  WHERE mview_name = 'SALES_MV';

REWRITE_ENABLED REWRITE_CAPABILITY
--------------- ------------------
Y               TEXTMATCH
```

Note that the `rewrite_capability` column can have only one of the following values: `none`, `textmatch`, or `general`. If `general` query rewrite is supported (and, consequently, the other two methods as well), the information provided by the `user_mviews` view is enough. However, as in this case, if the value `textmatch` is shown, it would be useful to know at least two more things. First, which of the two types of text-match query rewrite is supported? Only full-text-match query rewrite or also partial-text-match query rewrite? Second, why isn't general query rewrite supported?

To answer these questions, you can use the `explain_mview` procedure in the `dbms_mview` package, as shown in the following example. Notice that the `mv` parameter specifies the name of the materialized view, and the `stmt_id` parameter specifies an arbitrary string used to identify the information stored in the output table `mv_capabilities_table`:

```
SQL> execute dbms_mview.explain_mview(mv => 'sales_mv', stmt_id => '42')
```

---

■ **Note** The `mv_capabilities_table` table isn't available per default. You can create it in the schema used for the analysis by executing the `utlxmv.sql` script stored under `$ORACLE_HOME/rdbms/admin`.

---

The output of the procedure, found in the `mv_capabilities_table` table, shows whether the `sales_mv` materialized view supports the three query rewrite modes. If it doesn't, the `msgtxt` column indicates the reason why a specific query rewrite mode isn't supported. In this case, notice that the problem is caused by at least one column that's referenced in the `GROUP BY` clause only (upon inspecting the SQL statement, you can immediately identify the column causing the problem: `p.prod_status`):

```
SQL> SELECT capability_name, possible, msgtxt
  2  FROM mv_capabilities_table
  3  WHERE statement_id = '42'
  4  AND capability_name IN ('REWRITE_FULL_TEXT_MATCH',
  5                          'REWRITE_PARTIAL_TEXT_MATCH',
  6                          'REWRITE_GENERAL');
```

```
CAPABILITY_NAME            POSSIBLE MSGTXT
-------------------------  -------- ----------------------------------------
REWRITE_FULL_TEXT_MATCH    Y
REWRITE_PARTIAL_TEXT_MATCH  N        grouping column omitted from SELECT list
REWRITE_GENERAL            N        grouping column omitted from SELECT list
```

## Refreshes

When a table is modified, all dependent materialized views become stale. Therefore, for each stale materialized view, a refresh is necessary. When you create a materialized view, you can specify how and when refreshes will take place.

To specify how the database engine performs refreshes, you can choose from these methods:

- `REFRESH COMPLETE`: The whole content of the container table is deleted, and all data is reloaded from the base tables. Obviously, this method is always supported. You should use it only when a sizable part of a base table has been modified or fast refresh isn't available due to the complexity of the materialized view.
- `REFRESH FAST`: The content of the container table is reused, and only the modifications are propagated to the container table. If little data has been modified in the base tables, this is the method you should use. This method is available only if several requirements are fulfilled. If one of them isn't fulfilled, either the `REFRESH FAST` is refused as a valid parameter of the materialized view or an error is raised. Fast refreshes, as well as PCT refreshes (a special kind of fast refreshes), are covered in detail in the next sections.
- `REFRESH FORCE`: At first, a fast refresh is attempted. If it doesn't work, a complete refresh is performed. This is the default method.
- `NEVER REFRESH`: The materialized view is never refreshed. If a refresh is attempted, it terminates with the `ORA-23538: cannot explicitly refresh a NEVER REFRESH materialized view` error. You can use this method to make sure that a refresh will never be performed.

You can choose the point in time when the refresh of a materialized view occurs in two different ways:

- `ON DEMAND`: The materialized view is refreshed when explicitly requested (either manually or by running a job at a regular interval). This means that the materialized view may contain stale data during the period of time from the modification of the base tables to the refresh of the materialized view.
- `ON COMMIT`: The materialized view is automatically refreshed at the end of the transaction that modifies the tables it's based on. In other words, the materialized view always contains fresh data as far as the other sessions are concerned.

You can combine the options to specify how and when a materialized view is refreshed and use them with both the `CREATE MATERIALIZED VIEW` and `ALTER MATERIALIZED VIEW` statements. Here's an example of this:

```
ALTER MATERIALIZED VIEW sales_mv REFRESH FORCE ON DEMAND
```

It's even possible to create a materialized view with the `REFRESH COMPLETE ON COMMIT` options. However, it's very unlikely that such a configuration would be useful in practice.

To display the parameters associated with a materialized view, whether it's fresh, and how and when it was last refreshed, you can query the `user_mviews` view (`all`, `dba`, and, in a 12.1 multitenant environment, `cdb` variants of this view are also available):

```
SQL> SELECT refresh_method, refresh_mode, staleness, last_refresh_type, last_refresh_date
  2  FROM user_mviews
  3  WHERE mview_name = 'SALES_MV';

REFRESH_METHOD REFRESH_MODE STALENESS LAST_REFRESH_TYPE LAST_REFRESH_DATE
-------------- ------------ --------- ----------------- -----------------
FORCE          DEMAND       FRESH     COMPLETE          2013-12-10 15:51
```

When you choose to manually refresh a materialized view, you call one of the following procedures in the `dbms_mview` package:

- `refresh`: This procedure refreshes the materialized views specified as a comma-separated list through the `list` parameter. For example, the following call refreshes the `sales_mv` and `cal_month_sales_mv` materialized views that are owned by the `sh` user:

  ```
  dbms_mview.refresh(list => 'sh.sales_mv,sh.cal_month_sales_mv')
  ```

- `refresh_all_mviews`: This procedure refreshes all materialized views stored in the database except those that are marked to never be refreshed. The output parameter `number_of_failures` returns the number of failures that occurred during processing:

  ```
  dbms_mview.refresh_all_mviews(number_of_failures => :r)
  ```

- `refresh_dependent`: This procedure refreshes the materialized views that depend on the base tables that are specified as a comma-separated list through the parameter `list`. The output parameter `number_of_failures` returns the number of failures that occurred during processing. For example, the following call refreshes all materialized views, depending on the `sales` table owned by the `sh` user:

  ```
  dbms_mview.refresh_dependent(number_of_failures => :r, list => 'sh.sales')
  ```

All these procedures also support the parameters `method` and `atomic_refresh`. The former specifies how the refresh is done (`'c'` for complete, `'f'` for fast, `'p'` for PCT refresh, and `'?'` for force), and the latter specifies whether the refresh is performed in a single transaction. If the `atomic_refresh` parameter is set to `FALSE` (the default value is `TRUE`), no single transaction is used. As a result, for complete refreshes, the materialized views are truncated instead of being deleted. On the one hand, the refresh is faster. On the other hand, if another session queries the materialized view while a refresh is running, the query might return a wrong result (no rows selected).

In addition, from version 12.1 onward, a new parameter called `out_of_place` is available. If the `out_of_place` parameter is set to `FALSE` (the default value), refreshes are directly performed on the container table associated to the materialized view. Such refreshes might lead to some issues for concurrent queries accessing the materialized view.

If `out_of_place` is set to `TRUE`, refreshes are performed with the help of another table. What happens is that a new container table is created, the up-to-date data is inserted into it through a direct-path insert, the new container table is switched with the old container table, and finally the old container table is dropped. This method, called *out-of-place refresh*, makes sure that the impact on concurrent queries accessing the materialized is minimized. The drawback is that during the refresh, twice as much space is needed.

In case you want to automate a refresh on demand, with both CREATE MATERIALIZED VIEW and ALTER MATERIALIZED VIEW, you can also specify the time of the first refresh (START WITH clause) and an expression that evaluates to the time of subsequent ones (NEXT clause). For example, with the following SQL statement, a refresh is scheduled every ten minutes starting from the time the SQL statement is executed:

```
ALTER MATERIALIZED VIEW sales_mv
REFRESH COMPLETE ON DEMAND
START WITH sysdate
NEXT sysdate+to_dsinterval('0 00:10:00')
```

To schedule the refreshes, a job based on the dbms_job package is automatically submitted. Notice that the dbms_refresh package is used instead of the dbms_mview package:

```
SQL> SELECT what, interval
  2  FROM user_jobs;

WHAT                                       INTERVAL
------------------------------------------ -----------------------------------
dbms_refresh.refresh('"CHRIS"."SALES_MV"'); sysdate+to_dsinterval('0 00:10:00')
```

## REFRESH GROUPS

The dbms_refresh package is used to manage *refresh groups*. A refresh group is simply a collection of one or more materialized views. A refresh performed with the refresh procedure in the dbms_refresh package is performed in a single transaction (atomic_refresh is set to TRUE). This behavior is necessary if the consistency between several materialized views is to be guaranteed. This also means that either all materialized views contained in the group are successfully refreshed or the whole refresh is rolled back.

## Fast Refreshes with Materialized View Logs

During a fast refresh, the content of the container table is reused, and only the modifications are propagated from the base tables to the container table. Obviously, the database engine is able to propagate the modifications only if it knowns them. For that purpose, you have to create a *materialized view log* on each base table in order to enable fast refreshes (partition change tracking fast refreshes, discussed in the next section, are an exception to this). For example, in order to be fast-refreshed, the sales_mv materialized view requires materialized view logs on the sales, customers, and products tables.

Simply put, a materialized view log is a table that is automatically maintained by the database engine that tracks the modifications that occur on a base table. In addition to materialized view logs, an internal log table is used for direct-path inserts. You don't need to create it, because it's automatically installed when the database is created. To display its content, you can query the all_sumdelta view.

In the simplest case, you create the materialized view logs with SQL statements like the following (this example is based on the mv_refresh_log.sql script):

```
SQL> CREATE MATERIALIZED VIEW LOG ON sales WITH ROWID;

SQL> CREATE MATERIALIZED VIEW LOG ON customers WITH ROWID;

SQL> CREATE MATERIALIZED VIEW LOG ON products WITH ROWID;
```

The WITH ROWID clause is added to specify how rows are identified in the materialized view log—that is, how to identify the base table row whose modification is tracked by each materialized view log row. It's also possible to create materialized view logs that identify rows with the primary key or the object ID. However, for the purposes of this chapter, the rows have to be identified by their rowid (the others are useful when materialized views are used in distributed environments).

Just as materialized views have an associated container table, every materialized view log also has an associated table where the modifications to the base table are logged. The following query shows how to display its name:

```
SQL> SELECT master, log_table
  2  FROM dba_mview_logs
  3  WHERE master IN ('SALES', 'CUSTOMERS', 'PRODUCTS')
  4  AND log_owner = 'SH';

MASTER    LOG_TABLE
--------- ---------------
CUSTOMERS MLOG$_CUSTOMERS
PRODUCTS  MLOG$_PRODUCTS
SALES     MLOG$_SALES
```

For some materialized views, such a basic materialized view log isn't enough to support fast refreshes. There are additional requirements to be fulfilled. Because these requirements are strongly dependent on the query associated with the materialized view and the Oracle Database version, instead of providing a list, I show you how to find out, given a specific case, what these requirements are. To do this, you can use the same method used to find out what the supported query rewrite modes are (see the earlier section "Query Rewrite"). In other words, you can use the explain_mview procedure in the dbms_mview package, as shown in the following example:

```
SQL> execute dbms_mview.explain_mview(mv => 'sales_mv', stmt_id => '42')
```

The output of the procedure is provided in the mv_capabilities_table table. To see whether the materialized view can be fast-refreshed, you can use a query like the following. Notice that in the output, the possible column is always set to N. This means that no fast refresh is possible. In addition, the msgtxt and related_text columns indicate the cause of the problem:

```
SQL> SELECT capability_name, possible, msgtxt, related_text
  2  FROM mv_capabilities_table
  3  WHERE statement_id = '42'
  4  AND capability_name LIKE 'REFRESH_FAST_AFTER%';

CAPABILITY_NAME              POSSIBLE MSGTXT                         RELATED_TEXT
---------------------------- -------- ------------------------------ -------------
REFRESH_FAST_AFTER_INSERT    N        mv log must have new values    SH.PRODUCTS
REFRESH_FAST_AFTER_INSERT    N        mv log does not have all necess SH.PRODUCTS
                                      ary columns
REFRESH_FAST_AFTER_INSERT    N        mv log must have new values    SH.CUSTOMERS
REFRESH_FAST_AFTER_INSERT    N        mv log does not have all necess SH.CUSTOMERS
                                      ary columns
REFRESH_FAST_AFTER_INSERT    N        mv log must have new values    SH.SALES
REFRESH_FAST_AFTER_INSERT    N        mv log does not have all necess SH.SALES
                                      ary columns
REFRESH_FAST_AFTER_ONETAB_DML N       SUM(expr) without COUNT(expr)  AMOUNT_SOLD
REFRESH_FAST_AFTER_ONETAB_DML N       SUM(expr) without COUNT(expr)  QUANTITY_SOLD
REFRESH_FAST_AFTER_ONETAB_DML N       see the reason why REFRESH_FAST
                                      _AFTER_INSERT is disabled
REFRESH_FAST_AFTER_ONETAB_DML N       COUNT(*) is not present in the
                                      select list
REFRESH_FAST_AFTER_ONETAB_DML N       SUM(expr) without COUNT(expr)
REFRESH_FAST_AFTER_ANY_DML    N       mv log does not have sequence # SH.PRODUCTS
REFRESH_FAST_AFTER_ANY_DML    N       mv log does not have sequence # SH.CUSTOMERS
REFRESH_FAST_AFTER_ANY_DML    N       mv log does not have sequence # SH.SALES
REFRESH_FAST_AFTER_ANY_DML    N       see the reason why REFRESH_FAST
                                      _AFTER_ONETAB_DML is disabled
```

Some of the problems are related to the materialized view logs, others to the materialized view. Simply put, the database engine needs much more information to perform a fast refresh.

To solve the problems related to the materialized view logs, you must add some options to the `CREATE MATERIALIZED VIEW LOG` statements:

- For the "mv log does not have all necessary columns" problem, you have to specify that every column referenced in the materialized view be stored in the materialized view log.
- For the "mv log must have new values" problem, you have to add the `INCLUDING NEW VALUES` clause. With this option, materialized view logs store both old and new values (by default only old ones are stored) when an update is performed (in other words, two rows are written in the materialized view log).
- For the "mv log does not have sequence" problem, it's necessary to add the `SEQUENCE` clause. With this option, a sequential number is associated to each row stored in the materialized view log.

The following are the redefined materialized view logs:

```
SQL> CREATE MATERIALIZED VIEW LOG ON sales WITH ROWID, SEQUENCE
  2  (cust_id, prod_id, quantity_sold, amount_sold) INCLUDING NEW VALUES;

SQL> CREATE MATERIALIZED VIEW LOG ON customers WITH ROWID, SEQUENCE
  2  (cust_id, country_id) INCLUDING NEW VALUES;

SQL> CREATE MATERIALIZED VIEW LOG ON products WITH ROWID, SEQUENCE
  2  (prod_id, prod_category) INCLUDING NEW VALUES;
```

## TIMESTAMP VS. COMMIT SCN–BASED MATERIALIZED VIEW LOGS

From version 11.2 onward, there are two types of materialized view logs: those based on timestamps, and those based on commit SCN numbers. Because timestamp-based materialized view logs are the only ones that exist through version 11.1.0.7, they're used by default in later versions as well. To use the new type, the `COMMIT SCN` clause must be specified when creating a new materialized view log. The following SQL statement shows an example:

```
CREATE MATERIALIZED VIEW LOG ON sales WITH ROWID, COMMIT SCN, SEQUENCE (cust_id, prod_id,
quantity_sold, amount_sold) INCLUDING NEW VALUES
```

Although, from a user perspective, there's no difference between the two, the algorithm used for fast refreshes by materialized view logs based on commit SCN numbers can lead to better performance.

Note that materialized view logs based on commit SCN numbers aren't enabled by default because they're subject to some limitations. For example, tables with LOB columns aren't supported.

---

To solve the problems related to the materialized view, some new columns, based on the `count` function, have to be added to the query associated to the materialized view. The following SQL statement shows the definition that includes the new columns:

```
CREATE MATERIALIZED VIEW sales_mv
REFRESH FORCE ON DEMAND
AS
SELECT p.prod_category, c.country_id,
       sum(s.quantity_sold) AS quantity_sold,
       sum(s.amount_sold) AS amount_sold,
       count(*) AS count_star,
       count(s.quantity_sold) AS count_quantity_sold,
       count(s.amount_sold) AS count_amount_sold
FROM sales s, customers c, products p
WHERE s.cust_id = c.cust_id
AND s.prod_id = p.prod_id
GROUP BY p.prod_category, c.country_id
```

After redefining the materialized view logs and the materialized view, a further analysis using the `explain_mview` procedure shows that fast refreshes are possible in all situations (the `possible` column is always set to `Y`). So, let's test how fast the refresh is by inserting data into two tables and then executing a fast refresh:

```
SQL> INSERT INTO products
  2  SELECT 619, prod_name, prod_desc, prod_subcategory,  prod_subcategory_id,
  3         prod_subcategory_desc, prod_category, prod_category_id,
  4         prod_category_desc, prod_weight_class, prod_unit_of_measure,
  5         prod_pack_size, supplier_id, prod_status, prod_list_price,
  6         prod_min_price, prod_total, prod_total_id, prod_src_id,
  7         prod_eff_from, prod_eff_to, prod_valid
  8  FROM products
  9  WHERE prod_id = 136;

SQL> INSERT INTO sales
  2  SELECT 619, cust_id, time_id, channel_id, promo_id, quantity_sold, amount_sold
  3  FROM sales
  4  WHERE prod_id = 136;

SQL> COMMIT;

SQL> execute dbms_mview.refresh(list => 'sh.sales_mv', method => 'f')

Elapsed: 00:00:00.12
```

In this case, the fast refresh lasted 0.12 seconds. If you aren't satisfied with the performance of a fast refresh, you should use SQL trace to investigate why it's taking too long. Then, by applying the techniques described in Chapter 13, you might be able to speed it up by adding indexes (on the master tables, on the materialized view, or even sometimes on the materialized view log) or by partitioning a segment.

---

■ **Tip** There are several undocumented parameters that, depending on the version you're using and the type of materialized view you have troubles with, might be relevant for performance. Discussing these parameters goes beyond the scope of this book. If you have performance issues with fast refreshes, I advise you to take a look at the Oracle Support note *Master Note for Materialized View* (1353040.1), specifically to the references in the "Performance Issues with MVIEW" section.

---

## Fast Refreshes with Partition Change Tracking

Tables that store historical data are frequently range partitioned by day, week, or month. In other words, partitioning is based on a column that stores timing information. Therefore, it happens regularly that new partitions are added, data is loaded into them, and older ones are dropped (it's common to keep online only a specific number of partitions). After performing these operations, all dependent materialized views are stale and thus should be refreshed.

The problem is that fast refreshes with materialized view logs (those described in the previous section) can't be executed after a partition management operation such as `ADD PARTITION` or `DROP PARTITION`. If such a refresh is attempted, the database engine raises an `ORA-32313: REFRESH FAST of <mview> unsupported after PMOPs` error. Of course, it's always possible to execute a complete refresh. However, if there are many partitions and only one or two of them have been modified, the refresh time might be unacceptable.

To solve this problem, fast refreshes with partition change tracking (PCT) are available. This is possible because the database engine is able to track the staleness at partition level, and not only at table level. In other words, it's able to skip the refresh for all the partitions that have not been altered. To use this refresh method, the materialized view must fulfill some requirements. Basically, the database engine must be able to map the rows that are stored in the materialized view to the base table partitions. This is possible if the materialized view contains one of the following:

- Partition key
- Rowid
- Partition marker
- Join-dependent expression

It should be obvious what the first two are; let's look at examples of the third and fourth. A partition marker is nothing other than a partition identifier (actually, it's the data object ID associated to the segment of the partition) generated by the `pmarker` function in the `dbms_mview` package. To generate the partition marker, the function uses the rowid passed as a parameter. The following example, based on the `mv_refresh_pct.sql` script, shows how to create a materialized view containing a partition marker (note that `sales` table is partitioned):

```
CREATE MATERIALIZED VIEW sales_mv
REFRESH FORCE ON DEMAND
AS
SELECT p.prod_category, c.country_id,
       sum(quantity_sold) AS quantity_sold,
       sum(amount_sold) AS amount_sold,
       count(*) AS count_star,
       count(quantity_sold) AS count_quantity_sold,
       count(amount_sold) AS count_amount_sold,
       dbms_mview.pmarker(s.rowid) AS pmarker
FROM sales s, customers c, products p
WHERE s.cust_id = c.cust_id
AND s.prod_id = p.prod_id
GROUP BY p.prod_category, c.country_id, dbms_mview.pmarker(s.rowid)
```

■ **Note** Because the pmarker function is called for each row, don't underestimate the time needed to call it. On my system, creating the materialized view takes 2.5 times longer with the partition marker than without it.

The materialized view contains a join-dependent expression when one of the columns referenced in the SELECT clause belongs to a table that is joined through an equality predicate based on the partition key. In the example used in this section, it means that not only does an equi-join with the times table have to be added to the materialized view (s.time_id = t.time_id), but one of the columns of the times table is also added to the SELECT and GROUP BY clauses (t.fiscal_year). Here's an example:

```
CREATE MATERIALIZED VIEW sales_mv
REFRESH FORCE ON DEMAND
AS
SELECT p.prod_category, c.country_id, t.fiscal_year,
       sum(quantity_sold) AS quantity_sold,
       sum(amount_sold) AS amount_sold,
       count(*) AS count_star,
       count(quantity_sold) AS count_quantity_sold,
       count(amount_sold) AS count_amount_sold
FROM sales s, customers c, products p, times t
WHERE s.cust_id = c.cust_id
AND s.prod_id = p.prod_id
AND s.time_id = t.time_id
GROUP BY p.prod_category, c.country_id, t.fiscal_year
```

With either the partition marker or the join-dependent expression in place, an analysis using the explain_mview function in the dbms_mview package shows that a fast refresh based on partition change tracking is possible. However, it's possible only for modifications performed on the sales table:

```
SQL> SELECT capability_name, possible, msgtxt, related_text
  2  FROM mv_capabilities_table
  3  WHERE statement_id = '43'
  4  AND capability_name IN ('PCT_TABLE','REFRESH_FAST_PCT');

CAPABILITY_NAME  POSSIBLE MSGTXT                              RELATED_TEXT
---------------- -------- ----------------------------------- ------------
PCT_TABLE        Y                                            SALES
PCT_TABLE        N        relation is not a partitioned table CUSTOMERS
PCT_TABLE        N        relation is not a partitioned table PRODUCTS
REFRESH_FAST_PCT Y
```

## When to Use It

Materialized views are redundant access structures. Like all redundant access structures, they're useful for accessing data efficiently, but they impose an overhead to keep them up-to-date. If you compare materialized views to indexes, both the improvement and the overhead of materialized views may be much higher than those of indexes. Clearly, the two concepts are aimed at solving different problems. Simply put, you should use materialized views only if the pros of improving data access exceed the cons of managing redundant copies of the data (such as indexes, of course).

In general, I see two uses of materialized views:

- To improve the performance of large aggregations and/or joins for which the ratio between the number of logical reads and the number of returned rows is very high.
- To improve the performance of single-table accesses that are neither performed efficiently with a full table scan nor performed efficiently with an index range scan. Basically, these are accesses with an average selectivity that would require partitioning, but if it isn't possible to take advantage of partitioning (Chapter 13 discusses when this isn't possible), materialized views might be helpful.

Materialized views are commonly used in data warehouses to build stored aggregates. There are two reasons for this. First, data is mostly read-only; therefore, the overhead of refreshing materialized views can be minimized and segregated in time windows while the database is dedicated to modifying the tables only. Second, in such environments, the improvement may be huge. In fact, without materialized views, it's common to see queries based on large aggregates or joins that require an unacceptable amount of resources to be processed.

Even if data warehouses are the primary environment where materialized views are used, I have been successful in implementing them in OLTP systems as well. This may be beneficial for tables that are frequently queried and undergo, in comparison, relatively few modifications. In such environments, to refresh materialized views, it's common to use fast refreshes on commit in order to guarantee subsecond refresh times and always-fresh materialized views.

## Pitfalls and Fallacies

Because fast refreshes aren't always faster than complete refreshes, you shouldn't use them in all situations. One specific case is when lots of data is modified in the base tables. In addition, you shouldn't underestimate the overhead of maintaining the materialized view log while modifying the base tables. Hence, you should assess the pros and cons of using fast refreshes carefully.

When creating a materialized view log, you must be very careful with the use of commas. Can you identify what's wrong with the following SQL statement?

```
SQL> CREATE MATERIALIZED VIEW LOG ON products WITH ROWID, SEQUENCE,
  2  (prod_id, prod_category) INCLUDING NEW VALUES;
CREATE MATERIALIZED VIEW LOG ON products WITH ROWID, SEQUENCE,
*
ERROR at line 1:
ORA-12026: invalid filter column detected
```

The problem is the comma between the keyword `SEQUENCE` and the filter list (in other words, the list of columns between brackets). If the comma is present, the option `PRIMARY KEY` is implied, and that option can't be specified in this case because the primary key (the `prod_id` column) is already in the filter list. The following is the correct SQL statement. Notice that only the comma has been removed:

```
SQL> CREATE MATERIALIZED VIEW LOG ON products WITH ROWID, SEQUENCE
  2  (prod_id, prod_category) INCLUDING NEW VALUES;

Materialized view log created.
```

## Result Caching

Caching is one of the most common techniques used in computer systems to improve performance. Both hardware and software make extensive use of it. Oracle Database is no exception. For example, it caches data file blocks in the buffer cache, data dictionary information in the dictionary cache, and cursors in the library cache. As of version 11.1, *result caches* are also available.

---

■ **Note** Result caches are available in Enterprise Edition only.

---

### How It Works

Oracle Database provides three result caches:

- The *server result cache* (also known as *query result cache*) is a server-side cache that stores query result sets.
- The *PL/SQL function result cache* is a server-side cache that stores the return value of PL/SQL functions.
- The *client result cache* is a client-side cache that stores query result sets.

The next sections describe how these caches work and what you have to do to take advantage of them. Note that, by default, result caches aren't used.

### Server Result Cache

The server result cache can be used to avoid the reexecution of queries and some subqueries (those defined in the `WITH` clause and inline view defined in the `FROM` clause). Simply put, the first time a query is executed, its result set is stored in the shared pool. Then, for subsequent executions of the same query, the result set is served directly from the result cache instead of being recalculated. Note that two queries are considered equal and, therefore, can use the same cached result, only if they have the same text (differences in blank spaces and capitalization are allowed, though). In addition, if bind variables are present, their values must all be the same. This is necessary because, quite obviously, bind variables are input parameters that are passed to the query, and hence, the result set is usually different for different bind variable values. Also note that the result cache is stored in the shared pool, and all sessions connected to a given database instance share the same cache entries.

To provide you with an example (based on the `rc_query_hint.sql` script), let's execute the query already used in the section on materialized views twice (notice that the `result_cache` hint is specified in the query to enable the result cache):

```
SQL> SELECT /*+ result_cache */
  2         p.prod_category, c.country_id,
  3         sum(s.quantity_sold) AS quantity_sold,
  4         sum(s.amount_sold) AS amount_sold
  5  FROM sales s, customers c, products p
  6  WHERE s.cust_id = c.cust_id
  7  AND s.prod_id = p.prod_id
  8  GROUP BY p.prod_category, c.country_id
  9  ORDER BY p.prod_category, c.country_id;
```

```
Elapsed: 00:00:01.25

---------------------------------------------------------------------------------
| Id  | Operation                  | Name                         | Starts | A-Rows |
---------------------------------------------------------------------------------
|   0 | SELECT STATEMENT           |                              |      1 |     81 |
|   1 |  RESULT CACHE              | 089x05gkvfuxq7wqg06u9z0zkb   |      1 |     81 |
|   2 |   SORT GROUP BY            |                              |      1 |     81 |
|*  3 |    HASH JOIN               |                              |      1 |    956 |
|   4 |     TABLE ACCESS FULL      | PRODUCTS                     |      1 |     72 |
|   5 |     VIEW                   | VW_GBC_9                     |      1 |    956 |
|   6 |      HASH GROUP BY         |                              |      1 |    956 |
|*  7 |       HASH JOIN            |                              |      1 |    918K|
|   8 |        TABLE ACCESS FULL   | CUSTOMERS                    |      1 |  55500 |
|   9 |        PARTITION RANGE ALL|                              |      1 |    918K|
|  10 |         TABLE ACCESS FULL | SALES                        |     28 |    918K|
---------------------------------------------------------------------------------

   3 - access("ITEM_1"="P"."PROD_ID")
   7 - access("S"."CUST_ID"="C"."CUST_ID")
```

The first execution took 1.25 seconds. Notice that in the execution plan, the RESULT CACHE operation confirms that the result cache was enabled for the query. However, the Starts column in the execution plan clearly shows that all operations were executed at least once. The execution of all operations was necessary because this was the first execution of the query, and consequently, the result cache didn't contain the result set yet.

The second execution is faster (0.16 seconds):

```
SQL> SELECT /*+ result_cache */
  2         p.prod_category, c.country_id,
  3         sum(s.quantity_sold) AS quantity_sold,
  4         sum(s.amount_sold) AS amount_sold
  5  FROM sales s, customers c, products p
  6  WHERE s.cust_id = c.cust_id
  7  AND s.prod_id = p.prod_id
  8  GROUP BY p.prod_category, c.country_id
  9  ORDER BY p.prod_category, c.country_id;

Elapsed: 00:00:00.16

---------------------------------------------------------------------------------
| Id  | Operation                  | Name                         | Starts | A-Rows |
---------------------------------------------------------------------------------
|   0 | SELECT STATEMENT           |                              |      1 |     81 |
|   1 |  RESULT CACHE              | 089x05gkvfuxq7wqg06u9z0zkb   |      1 |     81 |
|   2 |   SORT GROUP BY            |                              |      0 |      0 |
|*  3 |    HASH JOIN               |                              |      0 |      0 |
|   4 |     TABLE ACCESS FULL      | PRODUCTS                     |      0 |      0 |
|   5 |     VIEW                   | VW_GBC_9                     |      0 |      0 |
|   6 |      HASH GROUP BY         |                              |      0 |      0 |
|*  7 |       HASH JOIN            |                              |      0 |      0 |
|   8 |        TABLE ACCESS FULL  | CUSTOMERS                         |      0 |      0 |
|   9 |        PARTITION RANGE ALL|                                   |      0 |      0 |
|  10 |         TABLE ACCESS FULL | SALES                             |      0 |      0 |
-----------------------------------------------------------------------------------

   3 - access("ITEM_1"="P"."PROD_ID")
   7 - access("S"."CUST_ID"="C"."CUST_ID")
```

This time, the Starts column in the execution plan shows that none of the operations were executed, except for RESULT CACHE. Note that this was one of those cases mentioned in Chapter 10 where a row source operation can completely avoid calling its child operation because it doesn't require it to fulfill its work. In other words, the result set for the query is served directly from the result cache.

In the execution plan, it's also interesting to note that a name, the *cache ID*, is associated with the RESULT CACHE operation. If you know the cache ID, you can query the v$result_cache_objects view to display information about the cached data. The following query shows that the cached result set has been published (in other words, available for use), when the result cache was created, how much time (in hundredths of seconds) it took to build it, how many rows are stored in it, and how many times it has been referenced:

```
SQL> SELECT status, creation_timestamp, build_time, row_count, scan_count
  2  FROM v$result_cache_objects
  3  WHERE cache_id = '089x05gkvfuxq7wqg06u9z0zkb';

STATUS    CREATION_TIMESTAMP BUILD_TIME ROW_COUNT SCAN_COUNT
--------- ------------------ ---------- --------- ----------
Published 2013-12-11 10:27           95        81          2
```

Other views that provide information about the result cache are v$result_cache_dependency, v$result_cache_memory, and v$result_cache_statistics.

From version 11.2 onward, specifying the result_cache hint isn't the only available way to enable the result cache. Another technique is to specify at the table level, with the result_cache clause set to force, that the result set of all queries referencing the table have to be cached (unless the no_result_cache hint is specified). Note that the default mode of the result_cache clause is default. This technique is especially useful for tables containing read-mostly data. In fact, thanks to this technique, it's not necessary to change the application to take advantage of the result cache. All you need to do is to specify at the table level that the result cache should be used.

Note that when several tables are referenced in a single query, all tables must have the result_cache clause set to force to enable the result cache. The following SQL statements, which are an excerpt of the rc_query_table.sql script, show how to enable the result cache for the three tables used in the queries used as examples in this section:

```
SQL> ALTER TABLE sales RESULT_CACHE (MODE FORCE);

SQL> ALTER TABLE customers RESULT_CACHE (MODE FORCE);

SQL> ALTER TABLE products RESULT_CACHE (MODE FORCE);
```

To guarantee the consistency of the result sets (that is, that the result set is the same whether it's served from the result cache or calculated from the database content), every time something changes in the tables referenced by a query, the cache entries dependent on it are invalidated (an exception with remote objects is discussed shortly). This is the case even if no real changes occur. For example, even a SELECT FOR UPDATE, immediately followed by a COMMIT, leads to the invalidation of the cache entries that depend on the selected table. It means that invalidations take place when a table with dependent cache entries is involved in a transaction, not when the data which the cache entries are based on changes. In other words, dependencies aren't fine-tracked: it doesn't matter whether the modified rows influence the cached result or not.

The following are the dynamic initialization parameters that control the server result cache:

- `result_cache_mode` specifies in which situation the result cache is used. You can set it either to `manual`, which is the default, or to `force`. With `manual`, the result cache is used only if it's enabled either through the `result_cache` hint or the `result_cache` clause. With `force`, the result cache is used for all queries unless the `no_result_cache` hint is specified. Because in most situations you want to use the result cache for a limited number of queries, I advise you to leave this initialization parameter to its default value.
- `result_cache_max_size` specifies (in bytes) the amount of shared pool memory that can be used for the result cache. If it's set to 0, the feature is completely disabled. The default value, which is greater than 0, is derived from the size of the shared pool. Memory allocation is dynamic, and therefore, this initialization parameter specifies only the upper limit. You can display the currently allocated memory by using a query like the following:

```
SQL> SELECT name, sum(bytes)
  2  FROM v$sgastat
  3  WHERE name LIKE 'Result Cache%'
  4  GROUP BY rollup(name);

NAME                      SUM(BYTES)
------------------------- ----------
Result Cache                  145928
Result Cache: Bloom Fltr        2048
Result Cache: Cache Mgr          208
Result Cache: Memory Mgr         200
Result Cache: State Objs        2896
                              151280
```

- `result_cache_max_result` specifies (in percent) the amount of `result_cache_max_size` that any single entry in the result cache can use. The default value is 5. Values from 0 to 100 are allowed.
- `result_cache_remote_expiration` specifies (in minutes) the temporal validity of an entry in the result cache based on remote objects. This is necessary because the invalidation of entries based on remote objects isn't performed when the remote objects have been changed. Instead, the entries are invalidated when the temporal validity defined by this initialization parameter is elapsed. The default value is 0, which means that the caching of queries based on remote objects is disabled.

The `result_cache_max_size` and `result_cache_max_result` initialization parameters can be changed at the system level only. In addition, in a 12.1 multitenant environment, they can only be set at the CDB level. The other two, `result_cache_mode` and `result_cache_remote_expiration`, can also be changed at the session level.

---

■ **Caution** Setting the `result_cache_remote_expiration` initialization parameter to a value greater than 0 can lead to stale results. Therefore, you should use values greater than 0 only if you fully understand and accept the implications of doing so.

---

There are a few, albeit obvious, limitations with the utilization of the result cache:

- Queries that reference nondeterministic SQL functions, sequences, and temporary tables aren't cached.
- Queries that violate read consistency aren't cached. For example, the result set created by a session with outstanding transactions on the referenced tables can't be cached.
- Queries that reference data dictionary views aren't cached.

### DBMS_RESULT_CACHE

You can use the `dbms_result_cache` package to manage the result cache. To do this, it provides the following subprograms:

- `bypass` temporarily disables (or enables) the result cache at the session or system level.
- `flush` removes all the objects from the result cache.
- `invalidate` invalidates all result sets that are dependent on a given database object.
- `invalidate_object` invalidates a single cache entry.
- `memory_report` produces a report on memory utilization.
- `status` shows the status of the result cache.

## PL/SQL Function Result Cache

The PL/SQL function result cache is similar to the server result cache, but it supports PL/SQL functions. It also shares the same memory structures as the server result cache. Its purpose is to store the return value of PL/SQL functions (and only the return value of functions—the result cache can't be used for `OUT` parameters) in the result cache. Obviously, functions with different input values are cached in separate cache entries. The following example, an excerpt of the output generated by the `rc_plsql.sql` script, shows a function for which the result cache is enabled. To enable it, the `RESULT_CACHE` clause is specified:

```
SQL> CREATE OR REPLACE FUNCTION f(p IN NUMBER)
  2    RETURN NUMBER
  3    RESULT_CACHE
  4  IS
  5    l_ret NUMBER;
  6  BEGIN
  7    SELECT count(*) INTO l_ret
  8    FROM t
  9    WHERE id = p;
 10    RETURN l_ret;
 11  END;
 12  /
```

In the following example, the function is called 10,000 times without taking advantage of the result cache (with the procedure bypass, the cache is temporarily disabled). The execution takes about 4 seconds:

```
SQL> execute dbms_result_cache.bypass(bypass_mode => TRUE, session => TRUE)

SQL> SELECT count(f(1)) FROM t;

COUNT(F(1))
-----------
      10000

Elapsed: 00:00:04.02
```

Now, let's call the function 10,000 times again, but this time with the result cache enabled. The execution takes only about three hundredths of a second:

```
SQL> execute dbms_result_cache.bypass(bypass_mode => FALSE, session => TRUE)

SQL> SELECT count(f(1)) FROM t;

COUNT(F(1))
-----------
      10000

Elapsed: 00:00:00.03
```

From version 11.2 onward, the database engine automatically detects which tables a function relies on. Based on that information, the result cache entries can be automatically invalidated whenever a transaction modifies any rows of a table a result cache entry relies on.

---

■ **Caution** In version 11.1, the result cache entries are invalidated only if the `RELIES_ON` clause is specified at the function level. The purpose of the `RELIES_ON` clause is to specify which tables the return value of the function depend on. This information is critical to the invalidation of the cache entries. If it's not specified, or contains wrong information, no invalidation will occur because of modifications that take place in the objects on which the function depends. Consequently, stale results can occur.

---

There are a few limitations to the utilization of the PL/SQL function result cache. The result cache can't be used for the following functions:

- Functions with `OUT` and/or `IN OUT` parameters
- Functions that are defined with invoker's rights (this limitation no longer exists from version 12.1 onward)
- Pipelined table functions
- Functions defined in anonymous blocks
- Functions with `IN` parameters or return values of the following types: LOBs, `REF CURSOR`, objects, and records

In addition, note that unhandled exceptions aren't stored in the result cache. In other words, if a function raises an exception and the exception is propagated to the caller, the next call of the same function will be executed again.

## Client Result Cache

The client result cache is a client-side cache that stores the result sets of queries executed by applications using Oracle Database drivers built on top of OCI libraries (for example JDBC OCI, ODP.NET, OCCI, and ODBC). Its purpose and workings are similar to the server result cache. In particular, the available techniques to enable the cache (the `result_cache` hint, the `RESULT_CACHE` clause, and the `result_cache_mode` initialization parameter) are the same. The only additional requirement is that exclusively SQL statements that take advantage of client-side statement caching (refer to Chapter 12) can use the client result cache. Compared to server-side implementation, there are two important differences. First, it avoids the client/server round-trips needed to execute SQL statements. This is a big advantage. Second, the invalidations are based on a polling mechanism, and therefore, consistency can't be guaranteed. This is a big disadvantage.

---

■ **Note** In a 12.1 multitenant environment, the client result cache isn't supported.

---

To implement polling, a client has to regularly execute database calls to check with the database engine whether one of its cached result sets has to be invalidated. To minimize the overhead associated with polling, every time a client executes a database call for another reason, it checks the validity of the cached result sets as well. In this way, database calls that are used exclusively for the invalidation of the cached result sets are avoided for clients that are steadily executing "regular" database calls.

Even if this is a client-side cache, you have to enable it on the server side. Note that the client-side cache also works if the server-side cache is disabled (in other words, if the `result_cache_max_size` initialization parameter is set to `0`). The following are the initialization parameters that control the client result cache:

- `client_result_cache_size` specifies (in bytes) the maximum amount of memory that every client process can use for the result cache. If it's set to 0, which is the default, the feature is disabled. This initialization parameter is static and can be set only at the system level. A database instance bounce is therefore necessary to change it.
- `client_result_cache_lag` specifies (in milliseconds) the maximum time lag between two database calls. In other words, it specifies how long stale result sets can remain in the client-side cache. The default value is 3,000. This initialization parameter is static and can be set only at the system level. A database instance bounce is therefore necessary to change it.

In addition to the server-side configuration, the following parameters can be specified in the client's `sqlnet.ora` file:

- `oci_result_cache_max_size` overrides the server-side setting that is specified with the `client_result_cache_size` initialization parameter. Note, however, that if the client result cache is disabled on the server side, this parameter can't enable it.
- `oci_result_cache_max_rset_size` specifies (in bytes) the maximum amount of memory any single result set can use.
- `oci_result_cache_max_rset_rows` specifies the maximum number of rows any single result set can store.

## When to Use It

If you're dealing with a performance problem caused by an application that executes the same operation over and over again, you have to reduce either the frequency of execution or the response time of the operation. Ideally, you should do both. However, sometimes (for example, when the application's code can't be modified) you can implement only the latter. To reduce response time, you should initially employ the techniques presented in Chapters 13 and 14. If this isn't enough, only then should advanced optimization techniques, such as result caches, be considered. Basically, result caches are effective given two conditions. First, the same data is queried more often than it's modified. Second, there is enough memory to cache the result sets.

In most situations, you shouldn't enable the result cache for all queries. In fact, most of the time, only specific queries can benefit from the result cache. For other queries than those specific ones, result cache management is simply pure overhead that might also overstress the cache. Also keep in mind that server-side caches are shared by all sessions, so their access is synchronized (they can become a point of serialization like any shared resource). Therefore, you should enable result caches only for the queries, subqueries, and tables requiring them. In other words, the result cache should be enabled selectively and only when it's really necessary to improve performance.

The server result cache doesn't completely avoid the overhead of executing a query. This means that if a query already performs relatively few logical reads (and no physical reads) without using the result cache, it won't be much faster when using it. Remember, both the buffer cache and the result cache are stored in the same shared memory.

The PL/SQL function result cache is especially useful for functions that are frequently called from SQL statements. In fact, it isn't uncommon for such functions to be called for every row that is either processed or returned, whereas the input parameters are different on only a few rows. However, functions that are frequently called from PL/SQL can also take advantage of the result cache.

Because of the problem with consistency, the client result cache should be used only for read-only or read-mostly tables.

Finally, note that you can take advantage of server and client result caches at the same time. However, for the queries executed by the client, you can't choose to bypass the client result cache and use the server result cache only. In other words, both result caches are used.

## Pitfalls and Fallacies

As pointed out in the previous sections, the consistency of the results isn't guaranteed in the following cases:

- When the `result_cache_remote_expiration` initialization parameter is set to a value greater than 0 and queries via database link are executed
- When, in version 11.1, PL/SQL functions that don't specify (or wrongly specify) the `RELIES_ON` clause are defined
- When the client result cache is used

In such cases, therefore, it's best to avoid result caches, unless you fully understand and accept the implications of each of these situations.

Caching the result of queries that reference nondeterministic PL/SQL functions, or functions that are sensitive to session-specific settings like NLS parameters and contexts, might not work as you expect. The issue is that by default, the database engine considers those functions as deterministic, and consequently, wrong results might be generated. Several examples are provided in the `rc_query_nondet.sql` script. The following example is one of them (notice how the second query returns a wrong result):

```
SQL> CREATE OR REPLACE FUNCTION f RETURN VARCHAR2
  2  IS
  3    l_ret VARCHAR2(64);
  4  BEGIN
  5    SELECT /*+ no_result_cache */ to_char(sysdate) INTO l_ret FROM dual;
  6    RETURN l_ret;
  7  END f;
  8  /

SQL> ALTER SESSION SET nls_date_format = 'YYYY-MM-DD HH24:MI:SS';

SQL> SELECT /*+ result_cache */ f() FROM dual;

F()
-------------------
2014-01-06 18:08:05

SQL> ALTER SESSION SET nls_date_format = 'YYYY-MM-DD';

SQL> SELECT /*+ result_cache */ f() FROM dual;

F()
-------------------
2014-01-06 18:08:05
```

To avoid this issue, from version 11.2.0.4 onward, you can set the `_result_cache_deterministic_plsql` undocumented initialization parameter to TRUE. Refer to the Oracle Support note *Bug 14320218 Wrong results with query results cache using PL/SQL function* (14320218.8) for more information.

## Parallel Processing

When you submit a SQL statement to a database engine, by default it's executed serially by a single server process. Therefore, even if the server running the database engine has several CPU cores, your SQL statement runs on a single CPU core. The purpose of parallel processing is to distribute the execution of a single SQL statement over several CPU cores.

---

■ **Note** Parallel processing is available in Enterprise Edition only.

---

### How It Works

Before describing the specifics of how queries, DML statements, and DDL statements are executed in parallel, it's important to understand the basics of parallel processing, how to configure a database instance to take advantage of parallel processing, and how to control the degree of parallelism.

#### Basics

Without parallel processing, a SQL statement is executed serially by a single server process that, in turn, runs on a single CPU core. This means that the amount of resources used for the execution of a SQL statement is restricted by the amount of processing a single CPU core can do. For example, as illustrated in Figure 15-5, if a SQL statement executes a data access operation that scans a whole segment (which is probably a disk I/O bound operation if most of the data is read from the disk) independently of the total throughput the disk I/O subsystem can deliver, the response time is limited by the bandwidth that a single CPU core can use. This bandwidth is of course limited because of hardware limitations of the data access path between the CPU core and the disk, and can't be used completely when the execution is serial: when the server process is on the CPU, by definition it isn't accessing the disk (asynchronous I/O operations are an exception to this) and thus, isn't taking advantage of the total throughput the disk I/O subsystem can deliver.

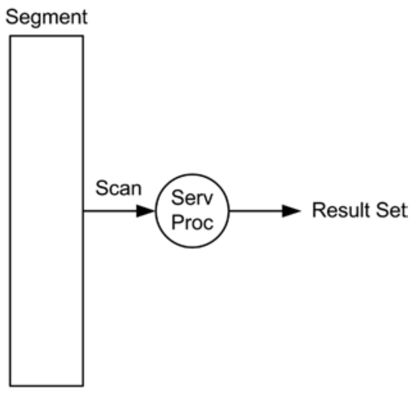


***Figure 15-5.*** *Serially executed SQL statements are processed by a single server process*

The following SQL statement with the related execution plan shows an example of the processing illustrated in Figure 15-5:

```
SELECT * FROM t

----------------------------------
| Id  | Operation          | Name |
----------------------------------
|   0 | SELECT STATEMENT   |      |
|   1 |  TABLE ACCESS FULL| T    |
----------------------------------
```

The aim of parallel processing is to split one large task into several smaller subtasks. If there is a parallel processed SQL statement, this basically means there are several *parallel query slave processes* (for simplicity, called *slave processes* throughout this book) that cooperate to execute a single SQL statement. The coordination of the slave processes is under the control of the server process associated to the session that submits the SQL statement. Because of this role, it's commonly called the *query coordinator*. The query coordinator is responsible for acquiring the slave processes, assigning a subtask to each of them, collecting and combining the partial result sets they deliver, and returning the final result set to the client. For example, in the case of a SQL statement that carries out a scan of a whole segment, the query coordinator can instruct each of the slave processes to scan part of the segment and to deliver the necessary data to it. Figure 15-6 illustrates this. Because each of the four slave processes is able to run on a different CPU core, in this case the response time is no longer limited by the bandwidth that a single CPU core can use.

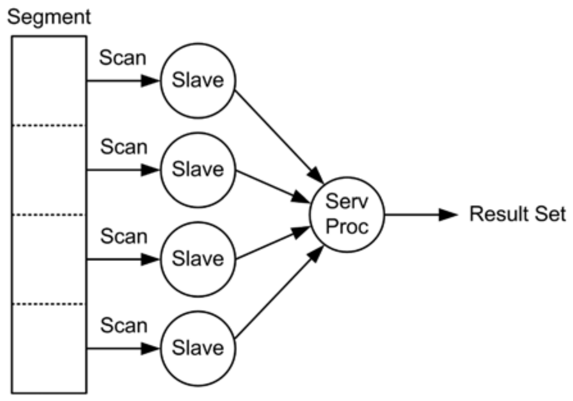


***Figure 15-6.*** *Parallel executed SQL statements are processed by several slave processes coordinated by a server process called query coordinator*

In a parallel scan like the one illustrated in Figure 15-6, the work is distributed among the slave processes in units of work called *granules*. Each slave process, at a given time, works on a single granule. If there are more granules than slave processes, when a slave process has finished working on a granule, it will receive another one to work on until all granules have been processed. The database engine can use two types of granules:

- A *partition granule* is a whole partition or subpartition. Obviously, this type of granule can be used only with partitioned segments.
- A *block range granule* is a range of blocks from a segment dynamically defined at runtime (not at parse time).

---

■ **Note** For a parallel scan of an external table, granules are defined as pieces of an external file (the default size is 10 megabytes). So it's not necessary to have several external files to allow parallel access.

---

Because the definition of partition granules is static (only the number, because of partition pruning, can change), block range granules tend to be used most of the time. Their main advantage is that they allow, in most situations, an even distribution of the work to the slave processes. In fact, with partition granules, the distribution of the work is highly dependent not only on the ratio of the number of partitions to the number of slave processes but also on the amount of data stored in each partition. Assume that every partition contains approximately the same amount of data. In that case, for a good distribution of the work, the number of partitions should be a multiple of the number of slave processes. If work isn't evenly distributed, some of the slave processes could work much more than others, and therefore, this could lead to a longer response time. As a result, the overall efficiency of the parallel execution might be jeopardized.

The following execution plan shows an example of the processing illustrated in Figure 15-6:

```
SELECT * FROM t

----------------------------------------------------------------------
| Id  | Operation             | Name     |    TQ  |IN-OUT| PQ Distrib |
----------------------------------------------------------------------
|   0 | SELECT STATEMENT      |          |        |      |            |
|   1 |  PX COORDINATOR       |          |        |      |            |
|   2 |   PX SEND QC (RANDOM)| :TQ10000 |  Q1,00 | P->S | QC (RAND)  |
|   3 |    PX BLOCK ITERATOR |          |  Q1,00 | PCWC |            |
|   4 |     TABLE ACCESS FULL| T        |  Q1,00 | PCWP |            |
----------------------------------------------------------------------
```

The execution plan is executed in the following way:

1. Through operation 4 (`TABLE ACCESS FULL`), each slave process scans part of the table. Which part it scans depends on its parent operation, 3 (`PX BLOCK ITERATOR`). This is the operation related to block range granules.
2. Operation 2 (`PX SEND QC`) sends the retrieved data to the query coordinator.
3. The query coordinator, through operation 1 (`PX COORDINATOR`), receives data from the slave processes and sends it back to the client.

When communication between processes take place, the processes that send data are called *producers*, and the processes that receive data are called *consumers*. To send data, a producer writes into a queue known as *table queue*. To receive data, a consumer reads from a table queue. Depending on the operation executed by the producers and the consumers, rows are distributed using one of the following methods (in the `dbms_xplan` package output, the `PQ Distrib` column provides this information):

- *Broadcast*: Each producer sends all rows to each consumer.
- *Broadcast Local*: This is a variation of the Broadcast distribution. It's used to send all rows to a subset of the slave processes only. It's most useful in a RAC environment to minimize cross-instance communication.
- *Round-robin*: Producers send each row to a single consumer one at a time, like dealing cards. As a result, rows are evenly distributed among consumers.
- *Range*: Producers send specific ranges of rows to different consumers. Dynamic range partitioning is performed to determine which row has to be sent to which consumer. For example, for a sort, this method range partitions the rows based on the columns used in the `ORDER BY` clause, so that each consumer can order only its subset of rows.
- *Hash*: Producers send rows to consumers as determined by a hash function. Dynamic hash partitioning is performed to determine which row is to be sent to which consumer. For example, for an aggregation, this method may hash partition the rows based on the columns used in the `GROUP BY` clause.
- *Partition Key*: Producers send rows to consumers based on the partition key. For addition information, refer to the "Partial Partition-wise Joins" section in Chapter 14.
- *Hybrid Hash*: Producers send rows to consumers using either the Broadcast or the Hash distribution method. Which one of the two is used is determined at runtime. Available from version 12.1 onward only.

- *One Slave*: Producers send all rows to a single consumer. Available from version 12.1 onward only.
- *QC Random*: Each producer sends all rows to the query coordinator. The order isn't important (hence, random). This is the most commonly used distribution to communicate with the query coordinator.
- *QC Order*: Each producer sends all rows to the query coordinator. The order is important. For example, this is used by a sort executed in parallel to send data to the query coordinator.

## RELATIONSHIP BETWEEN PARALLEL OPERATIONS

The following relationships between parallel operations are used in execution plans executed in parallel:

- Parallel to serial (`P->S`): A parallel operation sends data to a serial operation. For example, this is used in every execution plan to send data to the query coordinator.
- Parallel to parallel (`P->P`): A parallel operation sends data to another parallel operation. This is used when there are two sets of slave processes.
- Parallel combined with parent (`PCWP`): An operation is executed in parallel by the same slave processes that also execute the parent operation in the execution plan. Therefore, no communication takes place.
- Parallel combined with child (`PCWC`): An operation is executed in parallel by the same slave processes that also execute the child operation in the execution plan. Therefore, no communication takes place.
- Serial to parallel (`S->P`): A serial operation sends data to a parallel operation. Because most of the time this is inefficient, it should be avoided. There is one main reason for inefficiency: a single process might not be able to produce data as fast as several processes can consume it. If that's the case, the consumers spend much of their time waiting for data instead of doing real work.
- Serial combined with parent (SCWP): An operation is executed serially by the same slave process that also executes the parent operation in the execution plan. Therefore, no communication takes place. Available from version 12.1 onward.
- Serial combine with child (SCWC): An operation is executed serially by the same slave process that also executes the child operation in the execution plan. Therefore, no communication takes place. Available from version 12.1 onward.

In the output generated by the `dbms_xplan` package, the relationship between parallel operations is provided in the `IN-OUT` column.

Several parallel operations carried out as a sequence are collectively called *data flow operation* (DFO). In many situations, execution plans have a single data flow operation. However, there are cases where several data flow operations are required. To know the number of data flow operations, in the output generated by the `dbms_xplan` package, you have to check the TQ column. With it, you can identify not only how many data flow operations are used in an execution plan, but also which operations are executed by which set of slave processes. In fact, the content of the TQ column provides the following information:

- A NULL value corresponds to the operations carried out by the query coordinator.
- The value prefixed by the letter Q corresponds to the ID of the data flow operation.
- The value following the comma corresponds to the ID of the table queue where a set of slave processes *write*. What you can't know is how many slave processes belong to the set.

In the previous execution plan, because of the value Q1,00, you know that operations 2 to 4 belong to a single data flow operation (whose ID is 1) and they are all executed by a single set of slave processes (whose ID is 0).

Data access operations aren't the only operations that can be executed in parallel. In fact, among other things, the database engine is able to parallelize inserts, joins, aggregations, and sorts. When a SQL statement executes two or more independent operations (for example, a scan and a sort), it's common for the database engine to use two sets of slave processes. For example, as illustrated in Figure 15-7, if a SQL statement executes a scan and then a sort, one set is used for the scan and another set is used for the sort.

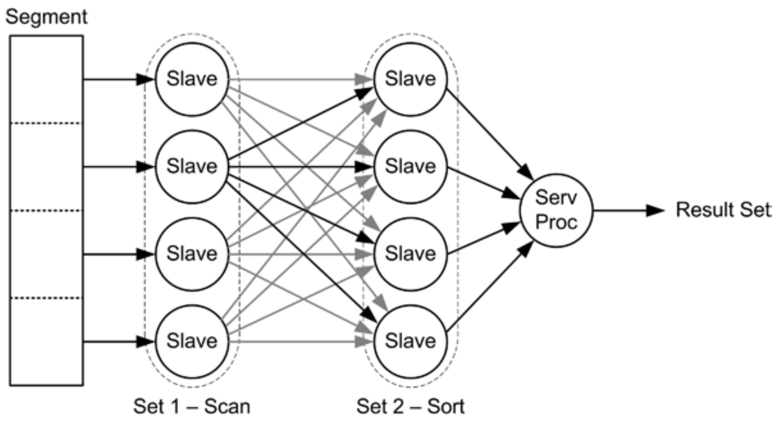


***Figure 15-7.*** *Two sets of slave processes can be used to execute a SQL statement*

The parallelization of a single operation is referred to as *intra-operation parallelism*. For example, in Figure 15-7, intra-operation parallelism (with four slave processes) is used twice: once for the scan and once for the sort. When two sets of slave processes are used to execute a data flow operation, the parallelization is referred to as *inter-operation parallelism*. For example, in Figure 15-7, inter-operation parallelism is used between set 1 (scan) and set 2 (sort).

The following execution plan is an example of the processing illustrated in Figure 15-7:

```
SELECT * FROM t ORDER BY id

-------------------------------------------------------------------------
| Id  | Operation               | Name     |    TQ  |IN-OUT| PQ Distrib |
-------------------------------------------------------------------------
|   0 | SELECT STATEMENT        |          |        |      |            |
|   1 |  PX COORDINATOR         |          |        |      |            |
|   2 |   PX SEND QC (ORDER)    | :TQ10001 |  Q1,01 | P->S | QC (ORDER) |
|   3 |    SORT ORDER BY        |          |  Q1,01 | PCWP |            |
|   4 |     PX RECEIVE          |          |  Q1,01 | PCWP |            |
|   5 |      PX SEND RANGE      | :TQ10000 |  Q1,00 | P->P | RANGE      |
|   6 |       PX BLOCK ITERATOR |          |  Q1,00 | PCWC |            |
|   7 |        TABLE ACCESS FULL| T        |  Q1,00 | PCWP |            |
-------------------------------------------------------------------------
```

The preceding execution plan is composed of a single data flow operation (whose ID is 1). Operations 5 to 7 have the same value for the column TQ (Q1,00), which means they're executed by one set of slave processes (set 1 in Figure 15-7). On the other hand, operations 2 to 4 have another value (Q1,01), and so are executed by another set of slave processes (set 2 in Figure 15-7). Set 1, the producer, scans the table based on block range granules (operation 6) and sends the retrieved data to set 2. In turn, set 2, the consumer, receives the data, sorts it, and sends the sorted result set to the query coordinator. Set 1 and set 2 do their processing concurrently. Because the two sets are communicating with each other, the set that processes data faster waits for the other one.

## Basic Configuration

This section describes the basic initialization parameters that you have to know to successfully set up a database instance for parallel processing. The initialization parameters involve the slave process pool and memory usage.

### Slave Processes Pool

The maximum number of slave processes per database instance is limited and maintained by a database instance as a pool of slave processes. A query coordinator requests slave processes from the pool, uses them to execute one SQL statement, and finally, when the execution is complete, returns them to the pool. The following initialization parameters are set to configure the pool:

- `parallel_min_servers` specifies the number of slave processes that are started at database instance startup. These slave processes are always available and don't need to be started when a server process requires them. The slave processes exceeding this minimum are dynamically started when required and, once returned to the pool, stay idle for five minutes. If they aren't reused in that period, they're shut down. By default, this initialization parameter is set to 0. This means that no slave processes are created at startup. I advise changing this value only if some SQL statements are waiting too long for the startup of the slave processes. The wait event related to this operation is `os thread startup`.
- `parallel_max_servers` specifies the maximum number of slave processes available in the pool. It's difficult to give advice on how to set this parameter. Nevertheless, a value of 10–20 times the number of CPU cores is a good starting point. The default value depends on several other initialization parameters, the version, and the platform. The maximum value that `parallel_max_servers` can be set to is 15 less than the value of the `processes` initialization parameter. If you try to set `parallel_max_servers` to a higher value, then at database instance startup, the value of `parallel_max_servers` is automatically adjusted, and a message is written into the alert log.

To display the status of the pool, you can use the following query:

```
SQL> SELECT *
  2  FROM v$px_process_sysstat
  3  WHERE statistic LIKE 'Servers%';
```

```
STATISTIC          VALUE
------------------ -----
Servers In Use         4
Servers Available      8
Servers Started       46
Servers Shutdown      34
Servers Highwater     12
Servers Cleaned Up     0
```

In a RAC environment, every database instance that is part of the cluster has its own slave processes pool. When a SQL statement is executed in parallel, slave processes can be allocated either locally, remotely from a single database instance, or from several database instances. The following methods are available to control the database instances which the slave processes are allocated from:

- The `parallel_instance_group` and `instance_groups` initialization parameters can be used to restrict the allocation of slave processes to specific database instances. With the `instance_groups` initialization parameter, you specify which groups each database instance lies in. With the `parallel_instance_group` initialization parameter, you specify which group the slave processes are allocated from. Since version 11.1, the `instance_groups` initialization parameter is deprecated. So the method just described should be used in version 10.2 only.
- From version 11.1 onward, the allocation of slave processes is service-aware. Slave processes are allocated only from database instances running the same service that is being run by the session executing the SQL statement in parallel. As of version 11.1 this method replaces the previous one.
- From version 11.2 onward, only local slave processes are allocated when the `parallel_force_local` initialization parameter is set to `TRUE` (the default is `FALSE`).

Because the configuration of the slave processes pool is database instance specific, in a 12.1 multitenant environment, it's not possible to set the `parallel_min_servers` and `parallel_max_servers` initialization parameters at the PDB level. However, it is possible to control the allocation of slave processes through services or the `parallel_force_local` initialization parameter at the PDB level.

## Memory Utilization

The table queues used for communication between processes are memory structures that can be allocated from either the shared pool or the large pool. However, using the shared pool for them is *not* recommended. The large pool, which is specialized for nonreusable memory structures, is a much better choice. There are two configurations that lead to a utilization of the large pool for table queues:

- Automatic SGA management is enabled through the `sga_target` or `memory_target` initialization parameters (the latter is available as of version 11.1 only).
- The `parallel_automatic_tuning` initialization parameter is set to `TRUE`. Note that this initialization parameter is deprecated. However, if you don't want to use automatic SGA management, setting it is the only way to use the large pool for parallel processing.

■ **Note** Despite its name, the `parallel_automatic_tuning` initialization parameter does only two simple things. First, it changes the default value of several initialization parameters related to parallel processing. Second, it instructs the database engine to use the large pool for the table queues.

Each table queue is composed of three (up to five with RAC) buffers for each pair of processes that communicate through it. The size of each buffer (in bytes) is set through the `parallel_execution_message_size` initialization parameter. The default size depends on the database engine version. Through version 11.1, it's either 2,152 bytes or, if the `parallel_automatic_tuning` initialization parameter is set to `TRUE`, 4,096 bytes. From version 11.2 onward, the default size is 16KB. For the best performance, you should set it to the highest supported value. Depending on the platform you're using, this could be either 16KB, 32KB, or 64KB. Therefore, especially prior to version 11.2, I advise you to change the default value.

When increasing the `parallel_execution_message_size` initialization parameter, you should make sure that the necessary memory is available. You can use Formula 15-1 to estimate the maximum amount of the large pool that should be available for a non-RAC database instance. For that purpose, the formula computes how many buffers are necessary for an execution that requires two sets of slave processes and uses the maximum possible degree of parallelism (half the value of the `parallel_max_servers` initialization parameter). Note that in a RAC environment, not only can the number of buffers used for each pair of processes that communicate be higher (up to five instead of three), but the maximum degree of parallelism also depends on the number of instances.

***Formula 15-1.*** The amount of the large pool used by non-RAC database instances for table queues

$$large\_pool\_size > 3 \cdot \left( parallel\_max\_servers + \frac{parallel\_max\_servers^2}{4} \right) \cdot parallel\_execution\_message\_size$$

To display how much of the large pool is currently in use by a database instance, you can run the following query:

```
SQL> SELECT *
  2  FROM v$sgastat
  3  WHERE name = 'PX msg pool';

POOL       NAME            BYTES
---------- ----------- ---------
large pool PX msg pool 823296000
```

You can run the following query to display the number of queue table buffers that are currently allocated (the `Buffers Current` statistic) and also the maximum number that have been allocated at one time since database instance startup (the `Buffers HWM` statistic):

```
SQL> SELECT *
  2  FROM v$px_process_sysstat
  3  WHERE statistic IN ('Buffers Current              ',
  4                      'Buffers HWM                  ');

STATISTIC                     VALUE
----------------------------- -----
Buffers Current               45076
Buffers HWM                   49924
```

Because the memory configuration is database instance specific, it's not possible, in a 12.1 multitenant environment, to set the `parallel_automatic_tuning` and `parallel_execution_message_size` initialization parameters at the PDB level.

In a RAC environment every database instance can have its own memory settings. The only exception is the `parallel_execution_message_size` initialization parameter. In fact, database instances with different values for this initialization parameter can't communicate and, therefore, raise an error when a SQL statement involving several database instances is executed. The following is an example of such an error:

```
SQL> SELECT * FROM gv$instance;
SELECT * FROM gv$instance
       *
ERROR at line 1:
ORA-12850: Could not allocate slaves on all specified instances: 2 needed, 1 allocated
ORA-12801: error signaled in parallel query server P001, instance 32766
```

## Degree of Parallelism

The number of slave processes used for intra-operation parallelism is called *degree of parallelism* (DOP). Because the degree of parallelism defines the number of slave processes for intra-operation parallelism, when inter-operation parallelism is used, the number of slave processes used to execute a SQL statement is higher than the degree of parallelism. In any case, note that a single data flow operation can't use more slave processes than two times the degree of parallelism. For example, Figure 15-7 shows a case where the number of slave processes is twice the degree of parallelism.

When a SQL statement (or part of it) is processed in parallel, the database engine has to select the degree of parallelism that is used for that purpose. Even though there are several initialization parameters and other factors that determine the actual degree of parallelism, there are really just two main modes in which to setup a database instance to control the degree of parallelism:

- *Manual degree of parallelism*: In this mode, you can control the degree of parallelism either at the session, object, or SQL statement level.
- *Automatic degree of parallelism*: In this mode, the database engine automatically selects the optimal degree of parallelism for every SQL statement.

Manual degree of parallelism is the only mode available through version 11.1.

---

■ **Caution** I advise you to use automatic degree of parallelism only from version 11.2.0.3 onward. The reason is that in prior versions, several bugs make it difficult to successfully implement. Refer to the Oracle Support note entitled *Init.ora Parameter "PARALLEL_DEGREE_POLICY" Reference Note* (1216277.1) for additional information. Also note that some changes in the functionality of automatic degree of parallelism were introduced in version 11.2.0.2. I don't think it makes sense to use that approach prior to version 11.2.0.3, so the functionality of version 11.2.0.1 isn't covered in this book.

---

From version 11.2 onward, the `parallel_degree_policy` initialization parameter is used not only to choose between manual and automatic degree of parallelism, but also to enable other features related to parallel processing. It accepts to one of the following values:

- `manual`: Manual degree of parallelism is enabled. This is the default value.
- `limited`: Automatic degree of parallelism is enabled only for SQL statements that reference objects whose `PARALLEL` is set to `DEFAULT` (see below for detail). For the others, manual degree of parallelism is used.

- `auto`: Automatic degree of parallelism is enabled for all SQL statements. In addition, two other features are enabled: *parallel statement queuing* and *in-memory parallel execution*.
- `adaptive`: This mode is similar to `auto`. The only difference is that *performance feedback* is also enabled. This value is available as of version 12.1 only.

---

■ **Note** Parallel statement queuing, in-memory parallel execution, and performance feedback are unrelated to automatic degree of parallelism. It's therefore unfortunate that they're all activated with a single initialization parameter. It would be much better to be able to activate them selectively.

---

The `parallel_degree_policy` initialization parameter can be set at the session level and, as of version 12.1, at the PDB level. It's also possible to override its value at the SQL statement level by specifying the `parallel` hint with the statement-level syntax. The `parallel` hint supports the following values:

- `parallel(manual)` activates manual degree of parallelism.
- `parallel(auto)` activates automatic degree of parallelism (but it doesn't activate parallel statement queuing and in-memory parallel execution).
- `parallel(`*n*`)` sets the degree of parallelism to the integer value specified as parameter (*n*).

---

■ **Caution** From version 11.2 onward, the `parallel` hint supports two syntaxes: statement-level and object-level. The statement-level syntax, as just described, overrides the `parallel_degree_policy` initialization parameter at the SQL statement level. The object-level syntax, covered in the upcoming "Manual Degree of Parallelism" section, overrides the degree of parallelism associated to each table and index.

---

The aim of the following sections is to describe how the database engine determines the degree of parallelism without considering that the number of slave processes might be capped by the load on the system or other factors. Later on, the "Limiting the Degree of Parallelism" section describes situations in which the degree of parallelism could be reduced.

## Default Degree of Parallelism

What is commonly called the *default degree of parallelism* is actually the maximum degree of parallelism you might want to use for any parallel SQL statement. In fact, the default value is only good if you want to run at most one SQL statement in parallel at any given time. How and when this default value is used depends on several factors, like the database instance configuration. The next two sections, while discussing how manual and automatic degree of parallelism work, provide more information about the default degree of parallelism.

To compute the default degree of parallelism, as shown by Formula 15-2, the database engine multiplies the value of the `cpu_count` initialization parameters by the number of slave processes that a CPU core is expected to handle (the `parallel_threads_per_cpu` initialization parameter). In the case of a RAC, the resulting value is further multiplied by the number of database instances in the cluster.

***Formula 15-2.*** The default degree of parallelism is the maximum degree of parallelism you might want to use for any parallel SQL statement

$$default\_dop = cpu\_count \cdot parallel\_threads\_per\_cpu \cdot number\_of\_instances$$

---

■ **Tip** On most platforms, the default value of the `parallel_threads_per_cpu` initialization parameter is 2. In case multithreading is enabled at the CPU level and, as a result, the value of the `cpu_count` initialization parameters is artificially inflated, I advise you to set the `parallel_threads_per_cpu` initialization parameter to 1.

---

## Manual Degree of Parallelism

A degree of parallelism is associated to each table and index. It's used by default for the operations referencing it. Its default value is 1, which means that no parallel processing is used. As shown in the following SQL statements, the degree of parallelism is set with the `PARALLEL` clause, either when an object is created or later:

```
CREATE TABLE t (id NUMBER, pad VARCHAR2(1000)) PARALLEL 4

ALTER TABLE t PARALLEL 2

CREATE INDEX i ON t (id) PARALLEL 4

ALTER INDEX i PARALLEL 2
```

---

■ **Caution** It's quite common to use parallel processing to improve the performance of maintenance tasks or batch jobs that create tables or indexes. For that purpose, it's common to specify the `PARALLEL` clause. Be aware, though, that when this clause is used, the degree of parallelism is used not only during the creation of the table or index but also later for the operations executed on it. Therefore, if you want to use parallel processing only during the creation of a table or index, it's essential that you alter the degree of parallelism once created.

---

To disable parallel processing, either the degree of parallelism is set to 1 or the `NOPARALLEL` clause is specified:

```
ALTER TABLE t PARALLEL 1

ALTER INDEX i NOPARALLEL
```

When the `PARALLEL` clause is used without specifying a degree of parallelism (for example, `ALTER TABLE t PARALLEL`), the default degree of parallelism is used. Because the default value is only good if you want to run at most one SQL statement in parallel at any given time, I usually recommend specifying a value.

To override the degree of parallelism defined at the table and index levels, it's possible to use the `parallel`, `no_parallel`, `parallel_index`, and `no_parallel_index` hints. In fact, when these hints are used with the object-level syntax, the first two override the setting at the table level, and the third and fourth override it at the index level. Let me stress that the object-level syntax, the one that specifies the name or the alias of the object, must be used. The following are examples of queries using that syntax to specify not only the object name (`t` for the table and `i` for the index) but also the degree of parallelism (16):

```
SELECT /*+ parallel(t 16) */ * FROM t

SELECT /*+ parallel_index(t i 16) */ * FROM t
```

With the `parallel` hint, it's also possible to explicitly call for the default degree of parallelism:

```
SELECT /*+ parallel(t default) */ * FROM t
```

When a different degree of parallelism is specified for different tables or indexes used in a single data flow operation, the database engine calculates a single degree of parallelism for the whole data flow operation. In general, the degree of parallelism chosen is simply the maximum of the ones specified at the table or index level.

## Automatic Degree of Parallelism

The idea behind automatic degree of parallelism is quite simple: for each SQL statement, the query optimizer chooses the optimal degree of parallelism. Hence, the query optimizer adapts the degree of parallelism according to the execution plan and the amount of processing that is expected to be done. As an example, Figure 15-8 shows the degree of parallelism chosen by the query optimizer on my test server when I execute a full scan on tables of different sizes. Note that to perform this test, I executed the `px_dop_auto.sql` script.

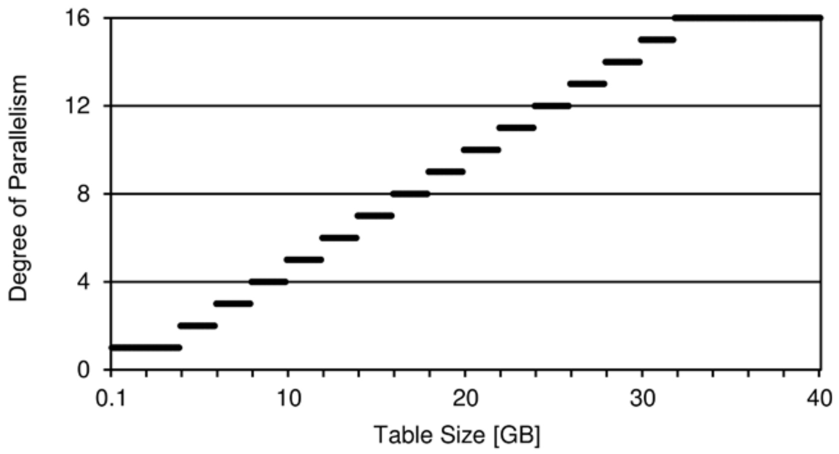


***Figure 15-8.*** *Between the minimum (1) and the maximum (16), the degree of parallelism increases proportionally to the amount of processing (the segment size in case of a full scan)*

Figure 15-8 shows that, on the one hand, there is a threshold under which the query optimizer decides to run the SQL statement serially, and on the other hand, there is a maximum degree of parallelism that isn't crossed.

The threshold is defined as the minimum amount of time that a SQL statement run serially should last (according to the query optimizer estimations) to be considered for parallel processing. It's configured through the `parallel_min_time_threshold` initialization parameter. The default value is `auto`, which is presently equivalent to 10 seconds. If you want to change that threshold, set the `parallel_min_time_threshold` initialization parameter to the number of seconds that fulfills your expectations.

The maximum degree of parallelism depends on the parallel_degree_limit initialization parameter. It can be set to one of the following values:

- CPU: The maximum degree of parallelism is equal to the default degree of parallelism. This is the default value.
- IO: The maximum degree of parallelism is defined by disk I/O cap. Refer to the "Disk I/O Cap" section later in this chapter for additional information about it.
- An integer value explicitly specifies the maximum degree of parallelism.

To use automatic degree of parallelism, two conditions have to be fulfilled. First, statistics gathered through I/O calibration have to be available. What these statistics are and how to gather them is covered in the next section, specifically in the "Disk I/O Cap" subsection. Second, the feature has to be enabled either through the parallel_degree_policy initialization parameter or the parallel(auto) hint. When the parallel_degree_policy initialization parameter is set to limited, there is an additional requirement: only tables and indexes having an associated default degree of parallelism are considered for parallel processing. The following example, based on the output of the px_dop_limited.sql script, illustrates this:

```
SQL> ALTER SESSION SET parallel_degree_policy = limited;

SQL> ALTER TABLE t NOPARALLEL;

SQL> SELECT * FROM t;

----------------------------------
| Id  | Operation         | Name |
----------------------------------
|   0 | SELECT STATEMENT  |      |
|   1 |  TABLE ACCESS FULL| T    |
----------------------------------

SQL> ALTER TABLE t PARALLEL;

SQL> SELECT * FROM t;

----------------------------------------------------------------------
| Id  | Operation            | Name     |    TQ  |IN-OUT| PQ Distrib |
----------------------------------------------------------------------
|   0 | SELECT STATEMENT     |          |        |      |            |
|   1 |  PX COORDINATOR      |          |        |      |            |
|   2 |   PX SEND QC (RANDOM)| :TQ10000 |  Q1,00 | P->S | QC (RAND)  |
|   3 |    PX BLOCK ITERATOR |          |  Q1,00 | PCWC |            |
|   4 |     TABLE ACCESS FULL| T        |  Q1,00 | PCWP |            |
----------------------------------------------------------------------

Note
-----
   - automatic DOP: Computed Degree of Parallelism is 4 because of degree limit
```

The last line of the preceding example shows that the note section generated by the `dbms_xplan` package clearly states whether automatic degree of parallelism is used. And, if it's used, the chosen degree of parallelism is mentioned. Other messages related to automatic degree of parallelism that can show up in the note section are the following:

```
automatic DOP: Computed Degree of Parallelism is 2

automatic DOP: Computed Degree of Parallelism is 1 because of parallel threshold

automatic DOP: skipped because of IO calibrate statistics are missing
```

If the degree of parallelism selected by the query optimizer is suboptimal, from version 12.1 onward, you can adjust the computation through the `parallel_degree_level` initialization parameter. Its default value is 100. If you specify values lower than 100, the degree of parallelism decreases proportionally. For example, with the value 50, the degree of parallelism reduces by 50%. If you specify values higher than 100, the degree of parallelism increases proportionally. For example, with the value 200 the degree of parallelism, provided the maximum isn't crossed, doubles.

## Limiting the Degree of Parallelism

The previous section describes how the database engine determines the degree of parallelism; this section describes those situations in which the degree of parallelism determined by the database engine can be reduced. Specifically, a reduction takes place when:

- adaptive parallelism is enabled,
- disk I/O cap is enabled,
- the Database Resource Manager caps the degree of parallelism, and
- a user profile limits the number of concurrent sessions a specific user can have.

Note that several of these features can also be concurrently enabled.

---

■ **Caution** The query optimizer is aware of the limitations imposed by the techniques described in this section only when two conditions are fulfilled: the cap is imposed by the Database Resource Manager, and automatic degree of parallelism is used. In all other situations, the query optimizer is unaware of the fact that a limitation to the degree of parallelism is in place. Note that being aware of this is essential because the costs estimated by the query optimizer do depend on the degree of parallelism. As a result, when the limitation isn't known, the query optimizer might choose a suboptimal execution plan.

---

### Adaptive Parallelism

Adaptive parallelism is controlled by the `parallel_adaptive_multi_user` initialization parameter. Its purpose is to influence the number of slave processes assigned to a server process. It accepts two values:

- `FALSE`: If the pool isn't exhausted, the requested number of slave processes is assigned to the server process.
- `TRUE`: As the number of already assigned slave processes increases, the requested degree of parallelism is automatically reduced, even if there are still enough slave processes in the pool to satisfy the required degree of parallelism. This is the default value.

■ **Note** The parallel_adaptive_multi_user initialization parameter is relevant only in two cases. First, when using manual degree of parallelism. Second, when using automatic degree of parallelism which is enabled by setting the parallel_degree_policy initialization parameter to limited.

To illustrate the impact of the parallel_adaptive_multi_user initialization parameter, let's take a look at the number of slave processes allocated when an increasing number of concurrent parallel operations are executed at short intervals. For this purpose, the following shell script was used. Its purpose is to start 20 concurrent parallel queries (each query runs for more than a dozen minutes) with a degree of parallelism of 16 (this is the default at the table level) at intervals of 5 seconds:

```
sql="select * from t;"
for i in 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
do
sqlplus -s $user/$password <<<$sql &
sleep 5
done
```

Figure 15-9 summarizes the results measured on version 11.2. With the parallel_adaptive_multi_user initialization parameter set to FALSE, the number of allocated slave processes is proportional to the number of executed parallel operations (in other words, each one runs with the same degree of parallelism) up to the limit imposed by the default value of the parallel_max_servers initialization parameter (160 in this case). With the parallel_adaptive_multi_user initialization parameter set to TRUE, starting from nine concurrent parallel operations, the degree of parallelism decreases, and therefore, fewer slave processes than requested are allocated.

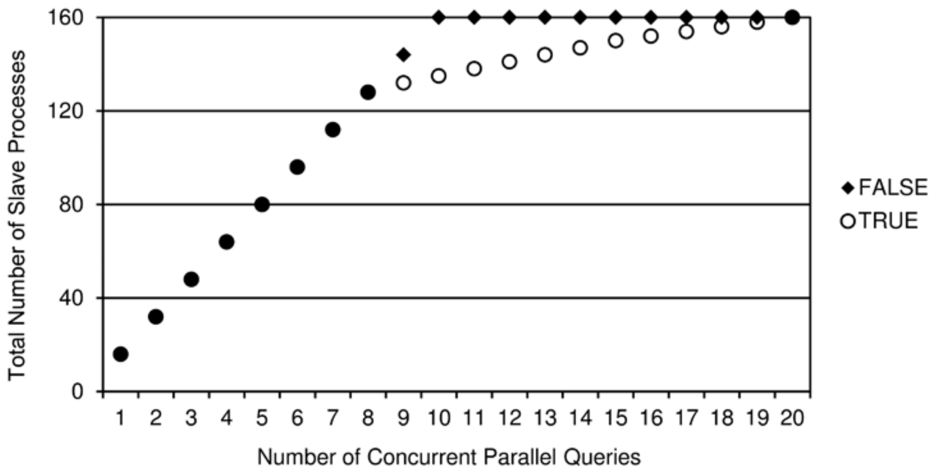


*Figure 15-9.* *Impact of the parallel_adaptive_multi_user initialization parameter*

## Disk I/O Cap

Disk I/O cap is a feature available from version 11.1 onward. Its purpose is to cap the default degree of parallelism according to the maximum throughput that the disk I/O subsystem can sustain. Let me stress, it only caps the default degree of parallelism. As a result, if the default degree of parallelism isn't relevant (for example, when using the manual degree of parallelism by specifying a particular value), disk I/O cap has no impact at all.

Disk I/O cap is especially useful for those systems that are I/O bound because of an unbalanced configuration. In the case of parallel processing, an unbalanced configuration often means too high a number of CPU cores compared to the throughput that the disk I/O subsystem can sustain.

---

■ **Tip** A good rule of thumb for dimensioning a database server that's intended to support a large number of parallel SQL statements (for example, a typical database server used for a data warehouse), is that the throughput the disk I/O subsystem should be able to sustain is as high as the number of CPU cores multiplied by 200 MB/s. For example, if a database server has 16 CPU cores, its disk I/O subsystem should sustain 3,200 MB/s.

---

To use disk I/O cap, two conditions have to be fulfilled. First, the feature has to be enabled through an initialization parameter. Which one depends on the version you're using:

- In version 11.1, the `parallel_io_cap_enabled` initialization parameter has to be set to `TRUE` (the default value is `FALSE`).
- From version 11.2 onward, the `parallel_degree_limit` initialization parameter has to be set to `IO` (the default value is `CPU`). Note that, as of version 11.2, the `parallel_io_cap_enabled` initialization parameter should be avoided because it's deprecated.

Second, statistics gathered through I/O calibration have to be available. These statistics are required because they provide the database engine with information about the maximum throughput the disk I/O subsystem can sustain. To gather them, the `calibrate_io` procedure of the `dbms_resource_manager` package has to be executed. The following PL/SQL block, an excerpt of the `px_calibrate_io.sql` script, shows how to do it. Notice that the `num_physical_disks` parameter has to be set to the number of physical disks on which the database is stored (in my test system, I have ten disks provisioned through ASM).

```
DECLARE
  l_max_iops PLS_INTEGER;
  l_max_mbps PLS_INTEGER;
  l_actual_latency PLS_INTEGER;
BEGIN
  dbms_resource_manager.calibrate_io(
    num_physical_disks => 10,
    max_iops           => l_max_iops,
    max_mbps           => l_max_mbps,
    actual_latency     => l_actual_latency
  );
END;
```

The gathering of statistics takes a few minutes. Once that work is over, the resulting statistics are externalized through the `dba_rsrc_io_calibrate` view. There are two values that are relevant for disk I/O cap: the maximum throughput that the disk I/O subsystem can sustain (`max_mbps`) and the maximum throughput that a single server process can sustain (`max_pmbps`). On my test systems they're as follows:

```
SQL> SELECT max_mbps, max_pmbps
  2  FROM dba_rsrc_io_calibrate;

MAX_MBPS MAX_PMBPS
-------- ---------
     664       297
```

Based on the two values returned by the preceding query, the database engine computes the maximum degree of parallelism, as shown by Formula 15-3. If the resulting value is lower than the default degree of parallelism, it becomes the new default. If the resulting value is higher than the default degree of parallelism, it's ignored.

***Formula 15-3.*** The default degree of parallelism is capped by the ratio between the maximum throughput the disk I/O subsystem can sustain and the maximum throughput a single server process can sustain

$$max\_default\_dop = \frac{max\_mbps}{max\_pmbps}$$

## Database Resource Manager

The Resource Manager provides control of database resources allocated to server processes. Among other things, it can be used to cap the degree of parallelism to a specific value. Because describing the Resource Manager in detail goes beyond the scope of this chapter (refer to the *Oracle Database Administrator's Guide* manual for more information), I'll just provide one example based on the `px_rm_cap_dop.sql` script. This example shows how to configure the Resource Manager to cap the degree of parallelism of the SQL statements executed by a specific user to eight. The configuration steps are the following:

1. Create a resource plan called `control_dop` that, through the `cap_dop` consumer group, limits the degree of parallelism to eight:

```
BEGIN
  dbms_resource_manager.create_pending_area();
  dbms_resource_manager.create_plan(
    plan    => 'CONTROL_DOP',
    comment => 'Control the degree of parallelism'
  );
  dbms_resource_manager.create_consumer_group (
    consumer_group => 'CAP_DOP',
    comment        => 'Users with a restricted degree of parallelism'
  );
  dbms_resource_manager.create_plan_directive(
    plan                     => 'CONTROL_DOP',
    group_or_subplan         => 'CAP_DOP',
    comment                  => 'Cap degree of parallelism',
    parallel_degree_limit_p1 => 8
  );
  dbms_resource_manager.create_plan_directive(
    plan             => 'CONTROL_DOP',
    group_or_subplan => 'OTHER_GROUPS',
    comment          => 'Unrestricted degree of parallelism'
  );
  dbms_resource_manager.validate_pending_area();
  dbms_resource_manager.submit_pending_area();
END;
```

2. Provide a specific user with the privilege to switch to the cap_dop consumer group:

```
BEGIN
  dbms_resource_manager_privs.grant_switch_consumer_group(
    grantee_name    => 'CHRIS',
    consumer_group  => 'CAP_DOP',
    grant_option    => FALSE
  );

END;
```

3. Map the sessions of a specific user to the cap_dop consumer group:

```
BEGIN
  dbms_resource_manager.create_pending_area();
  dbms_resource_manager.set_consumer_group_mapping(
    attribute      => 'ORACLE_USER',
    value          => 'CHRIS',
    consumer_group => 'CAP_DOP'
  );
  dbms_resource_manager.submit_pending_area();
END;
```

4. Enable the control_dop resource plan at the system level:

```
ALTER SYSTEM SET resource_manager_plan = control_dop
```

## User Profile

Through user profiles, specifically the sessions_per_user parameter, it's possible to limit the number of concurrent sessions a specific user can have. For example, the following SQL statements create a new user profile (limit_dop) that limits the number of sessions to 16, associates it to a user, and enables it by setting the resource_limit initialization parameter to TRUE (the default value is FALSE):

```
CREATE PROFILE limit_dop LIMIT sessions_per_user 16

ALTER USER chris PROFILE limit_dop
```

Despite the fact that the limit imposed by a user profile was originally introduced to prevent an end-user from concurrently logging into the same database instance more than a specific number of times, the limit is also useful in managing the number of parallel sessions. Such sessions are created automatically by the database engine when a SQL statement is executed in parallel. The limit applies to them as well.

The reason extra sessions are created is because one SQL statement executed in parallel requires not only one session for the query coordinator, but also one session for each slave process. As a result, even though an end-user logs in only once, he might require several sessions. Therefore, depending on how the `sessions_per_user` parameter is set, the degree of parallelism might be limited.

---

■ **Tip** I don't advise limiting resources through user profiles. To do that, you should ideally use the Resource Manager. I present the user profile approach just because you must be aware that user profiles can limit the degree of parallelism.

---

## Downgrades

Downgrades take place when the number of slave processes that the query coordinator requests is higher than the number of slave processes it actually gets. This happens in two situations:

- When the degree of parallelism is limited by the techniques described in the preceding section. In other words, when the degree of parallelism is limited by either adaptive parallelism, the Resource Manager (for manual degree of parallelism only), or a user profile.
- When the query coordinator requests from the pool a number of slave processes that's higher than the number of slave processes actually available.

In fact, the database engine, depending on how many slave processes are already running at the time the query coordinator requests some of them, might not be able to fulfill the request. For example, if the maximum number of slave processes is set to 40, for the execution plan illustrated in Figure 15-7 (that requires 8 slave processes) only 5 concurrent SQL statements (40/8) can be executed with the required degree of parallelism. When the limit is reached, there are three possibilities:

- Either the degree of parallelism is downgraded (in other words, reduced),
- An `ORA-12827: insufficient parallel query slaves available` error is returned to the query coordinator, or
- The execution of the SQL statement is put on hold until the necessary number of slave processes is available.

The latter approach is used from 11.2 onward, and only if statement queuing is enabled (the next section covers this). If statement queuing isn't enabled, one of the other two possibilities has to be used. To configure which one is used, you have to set the `parallel_min_percent` initialization parameter. It can be set to an integer value ranging from 0 to 100. There are three main situations:

- *0*: This value (which is the default) specifies that the degree of parallelism can be silently downgraded. In other words, the database engine can provide as many slave processes as possible. If less than two slave processes are available, the execution is serialized. This means that the SQL statements are always executed, and the ORA-12827 error is never raised.
- *1–99*: The values ranging from 1 to 99 define a limit for the downgrade. At least the specified percentage of the slave processes must be provided; otherwise, the ORA-12827 error is raised. For example, if it's set to 25 and 16 slave processes are requested, at least 4 (16*25/100) must be provided to avoid the error.
- *100*: With this value, either all the requested slave processes are provided or the ORA-12827 error is raised.

The following example (based on the `px_min_percent.sql` script), executed while no other parallel execution was running, illustrates this (note that 40 is 80% of 50):

```
SQL> ALTER SYSTEM SET parallel_max_servers = 40;

SQL> ALTER TABLE t PARALLEL 50;

SQL> ALTER SESSION SET parallel_min_percent = 80;

SQL> SELECT count(pad) FROM t;

COUNT(PAD)
----------
    100000

SQL> SELECT * FROM table(dbms_xplan.display_cursor(NULL, NULL, 'basic +parallel'));

------------------------------------------------------------------------
| Id  | Operation              | Name     |    TQ  |IN-OUT| PQ Distrib |
------------------------------------------------------------------------
|   0 | SELECT STATEMENT       |          |        |      |            |
|   1 |  SORT AGGREGATE        |          |        |      |            |
|   2 |   PX COORDINATOR       |          |        |      |            |
|   3 |    PX SEND QC (RANDOM) | :TQ10000 |  Q1,00 | P->S | QC (RAND)  |
|   4 |     SORT AGGREGATE     |          |  Q1,00 | PCWP |            |
|   5 |      PX BLOCK ITERATOR |          |  Q1,00 | PCWC |            |
|   6 |       TABLE ACCESS FULL| T        |  Q1,00 | PCWP |            |
------------------------------------------------------------------------

SQL> ALTER SESSION SET parallel_min_percent = 81;

SQL> SELECT count(pad) FROM t;
SELECT count(pad) FROM t
*
ERROR at line 1:
ORA-12827: insufficient parallel query slaves (requested 50, available 40, parallel_min_percent 81)
```

If you want to know how many operations were downgraded on a running database instance and by how much, you can execute the following query. Obviously, when you see too many downgrades, especially when many operations are serialized, you should question the configuration:

```
SQL> SELECT name, value
  2  FROM v$sysstat
  3  WHERE name like 'Parallel operations%';

NAME                                           VALUE
-------------------------------------------- -----
Parallel operations not downgraded                14
Parallel operations downgraded to serial          10
Parallel operations downgraded 75 to 99 pct       14
Parallel operations downgraded 50 to 75 pct        2
Parallel operations downgraded 25 to 50 pct        0
Parallel operations downgraded 1 to 25 pct         0
```

## Statement Queuing

The aim of statement queuing, a feature available as of version 11.2, is to avoid downgrades. To do this, the Resource Manager recognizes when not enough slave processes are available to execute a specific SQL statement. When it recognizes such a situation, it suspends the execution by enqueuing the session in a waiting list. Then, as soon as the required number of slave processes is available, it dequeues the session and resumes the execution. The waiting list is managed with a first in, first out queue where, by default, sessions have to wait until the slave processes are availble (in other words, when there is no timeout).

Contrary to what you might expect, the Resource Manager doesn't check whether a sufficient number of slave processes is available by comparing the current number of active slave processes with the maximum number that the database instance can handle (the value defined by the parallel_max_servers initialization parameter). Instead, the Resource Manager uses another threshold configured through the parallel_servers_target initialization parameter. The reason behind this additional initialization parameter is quite simple: statement queuing isn't necessarily enabled for all SQL statements that are executed in parallel. As a result, it might be worthwhile to manage only part of the slave processes pool through statement queuing.

Even though the parallel_servers_target initialization parameter is dynamic, it can only be set at the system level. In addition, in a 12.1 multitenant environment, it's not possible to set it at the PDB level. In other words, as for the slave processes pool, the configuration can be performed at the database instance level only.

Statement queuing is only enabled in two situations. First, for SQL statements executed by sessions that have the parallel_degree_policy initialization parameter set to auto (provided they don't contain a hint disabling statement queuing). Second, for SQL statements that contain the statement_queuing hint. This second possibility is devoted to applications that use manual degree of parallelism.

For sessions with the parallel_degree_policy initialization parameter set to auto, statement queuing can be explicitly disabled at the SQL statement level by adding the no_statement_queuing hint.

When a session is waiting in the queue, it's waiting on the resmgr:pq queued event. For example, the following query shows which sessions are suspended because of statement queuing:

```
SQL> SELECT sid, seconds_in_wait
  2  FROM v$session
  3  WHERE event = 'resmgr:pq queued';

SID SECONDS_IN_WAIT
--- ---------------
113             121
 37              60
143              34
```

The suspended sessions can also be monitored through the v$rsrc_session_info dynamic performance view. With it, you can see not only how long (in milliseconds) they're waiting for in the queue (the current_pq_queued_time column), but also which session will be dequeued next (the pq_status column is set to Queue head) and how many slave processes every suspended session requires (the pq_servers column). Note that the last two columns are available only from version 12.1 onward. The following example illustrates this for the three sessions already listed by the previous query:

```
SQL> SELECT sid, pq_status, current_pq_queued_time, pq_servers
  2  FROM v$rsrc_session_info
  3  WHERE state = 'PQ QUEUED';

SID PQ_STATUS  CURRENT_PQ_QUEUED_TIME PQ_SERVERS
--- ---------- ---------------------- ----------
113 Queue head                 121068         16
 37 Queued                      60686         16
143 Queued                      34843          4
```

In addition to the default behavior just described, the Resource Manager provides several directives that allow you to set up resource plans that take advantage of the following features:

- Managing the order in which the sessions in the waiting list are dequeued
- Limiting the slave processes usage at the consumer group level
- Specifying a timeout
- Defining critical SQL statements that bypass statement queuing (version 12.1 only)
- Managing statement queuing in multitenant environments (version 12.1 only)

Detailed information about these features is provided in the *Oracle Database VLDB and Partitioning Guide* manual.

## Parallel Queries

The following operations, in both queries and subqueries, can be executed in parallel:

- Full table scans, full partition scans, and fast full index scans
- Index full and range scans, but only if the index is partitioned (at a given time, a partition can be accessed by a single slave process only—as a side effect, the degree of parallelism is limited by the number of accessed partitions)
- Joins (Chapter 14 also provides some examples)
- Set operators
- Sorts
- Aggregations

---

■ **Note** Full table scans, full partition scans, and fast full index scans executed in parallel generally use direct reads and, therefore, bypass the buffer cache. An exception is when, from version 11.2 onward, in-memory parallel execution is active. In fact, the aim of in-memory execution is precisely to avoid direct reads and to cache as much data as possible. Note that the database engine, for index full and range scans, always performs regular physical reads.

---

Queries can't be executed in parallel when they reference a user-defined function that doesn't support parallel processing. Basically, to support parallel processing, a user-defined function must neither write to the database nor read or modify package variables. When writing PL/SQL code, you should mark user-defined functions that support parallel processing with the `PARALLEL_ENABLE` clause. Note that in some situations the capability of executing a user-defined function in parallel depends not only on the user-defined function itself, but also on the query (and, in the end, on the execution plan itself). The script `px_query_udf.sql` provides an example where one function prevents one query from running in parallel while it doesn't place such a restriction on another query.

Parallel queries are *enabled* by default. At the session level, you can enable and disable them with the following SQL statements:

```
ALTER SESSION ENABLE PARALLEL QUERY

ALTER SESSION DISABLE PARALLEL QUERY
```

In addition, it's also possible to enable parallel queries and, at the same time, override the degree of parallelism defined either by manual degree of parallelism at the segment level or by automatic degree of parallelism with the following SQL statement:

```
ALTER SESSION FORCE PARALLEL QUERY PARALLEL 4
```

Be aware, however, that hints have precedence over the setting at the session level. On one hand, even if parallel queries are disabled at the session level, hints can enable a parallel execution. The only way to really turn off parallel queries is to either set the `parallel_max_servers` initialization parameter to 0 or configure the Resource Manager to do so. On the other hand, even if a parallel degree is forced at the session level, hints can lead to another degree of parallelism. To check whether parallel queries are enabled or disabled at the session level, you can execute a query like the following one (the `pq_status` column is set to either `ENABLED`, `DISABLED`, or `FORCED`):

```
SELECT pq_status
FROM v$session
WHERE sid = sys_context('userenv','sid')
```

The following execution plan shows an example with a parallel index range scan, a parallel full table scan, and a parallel hash join. It's based on the `px_query.sql` script. Notice the hints: the `parallel_index` hint is used for the index access, and the `parallel` hint is used for the table scan. Both hints use the object-level syntax to specify a degree of parallelism of 2. In addition, the `pq_distribute` hint is used to specify the distribution method. The column `TQ` contains three values, which means that three sets of slave processes are used to carry out this execution plan. Operation 8 scans index `i1` in parallel (this is possible because the index is partitioned). Then, operation 7, with the rowid extracted from index `i1`, accesses table t1. As shown in operation 6, partition granules are used for these two operations. Then, the data is sent with a hash distribution to the consumers (the slave processes of set `Q1,02`). When the consumers receive the data (operation 4), they pass it to operation 3 to build the hash table in memory for the hash join. As soon as all the data of table `t1` is fully processed, the parallel full scan of table `t2` can start. This is performed in operation 12. As shown in operation 11, block range granules are used for this operation. Then the data is sent with a hash distribution to the consumers (the slave processes of the set `Q1,02`). When the consumers receive data (operation 9), they pass it to operation 3 to probe the hash table. Finally, operation 2 sends the rows fulfilling the join condition to the query coordinator (Figure 15-10 illustrates this execution plan):

```
SELECT /*+ leading(t1) use_hash(t2)
           index(t1) parallel_index(t1 2)
           full(t2) parallel(t2 2)
           pq_distribute(t2 hash,hash) */ *
FROM t1, t2
WHERE t1.id > 9000
AND t1.id = t2.id+1

----------------------------------------------------------------------------------------------
| Id | Operation                         | Name       | Pstart| Pstop |   TQ   |IN-OUT| PQ Distri |
----------------------------------------------------------------------------------------------
|  0 | SELECT STATEMENT                  |            |       |       |        |      |           |
|  1 |  PX COORDINATOR                   |            |       |       |        |      |           |
|  2 |   PX SEND QC (RANDOM)             | :TQ10002   |       |       | Q1,02  | P->S | QC (RAND) |
|* 3 |    HASH JOIN BUFFERED             |            |       |       | Q1,02  | PCWP |           |
|  4 |     PX RECEIVE                    |            |       |       | Q1,02  | PCWP |           |
|  5 |      PX SEND HASH                 | :TQ10000   |       |       | Q1,00  | P->P | HASH      |
|  6 |       PX PARTITION HASH ALL       |            |     1 |     4 | Q1,00  | PCWC |           |
|  7 |        TABLE ACCESS BY INDEX ROW| T1         |       |       | Q1,00  | PCWP |           |
|* 8 |         INDEX RANGE SCAN          | I1         |     1 |     4 | Q1,00  | PCWP |           |
|  9 |     PX RECEIVE                 |          |        |        | Q1,02 | PCWP |            |
| 10 |      PX SEND HASH              | :TQ10001 |        |        | Q1,01 | P->P | HASH       |
| 11 |       PX BLOCK ITERATOR        |          |        |        | Q1,01 | PCWC |            |
|*12 |        TABLE ACCESS FULL       | T2       |        |        | Q1,01 | PCWP |            |
-------------------------------------------------------------------------------------------

   3 - access("T1"."ID"="T2"."ID"+1)
   8 - access("T1"."ID">9000)
  12 - filter("T2"."ID"+1>9000)
```

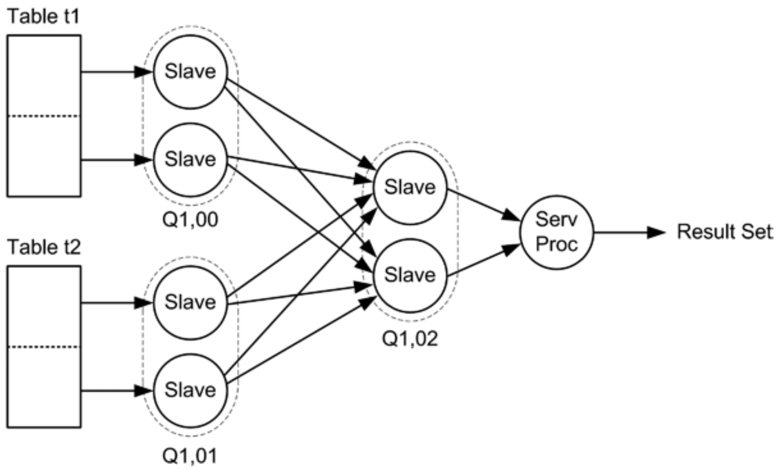


***Figure 15-10.*** *Even though three sets of slave processes are shown, a single data flow operation can't be executed with more than two sets of slave processes*

According to the execution plan and Figure 15-10, one data flow operation and three table queues are used. Note that even though Figure 15-10 shows three sets of slave processes (since the requested degree of parallelism is 2, six slave processes in total), during the execution, only two sets are allocated from the pool (in other words, four slave processes). This is because a single data flow operation can't use more than two sets of slave processes. What happens in this particular case is that the set used for scanning table t1 (`Q1,00`) never works concurrently with the set used for scanning table t2 (`Q1,01`). Therefore, the query coordinator simply (re)uses the same slave processes for these two sets.

---

■ **Caution** The `HASH JOIN BUFFERED` operation (operation 3, in the previous execution plan) not only creates a hash table containing the data returned by the build input (operations 5 through 8), but also buffers the data returned by the probe input (operations 10 through 12) that fulfills the join condition. Hence, the suffix `BUFFERED`. This is a special behavior that the database engine has to implement because of an internal limitation (two distribution operations can't be active at the same time). From a performance point of view, the buffering might be a major issue.

---

## Parallel DML Statements

The following DML statements can be executed in parallel:

- `DELETE`
- `INSERT` with a subquery (`INSERT` statements with the `VALUES` clause can't be parallelized)
- `MERGE`
- `UPDATE`

---

■ **Note** `INSERT` statements and `MERGE` statements (for the part inserting data) executed in parallel use direct-path inserts. Therefore, they're subject not only to the pros and cons of direct-path inserts, but also to their restrictions. I describe them in the "Direct-Path Insert" section later in this chapter.

---

DML statements can't be executed in parallel when:

- a table has a trigger;
- a table has either a foreign key constraint referencing itself, a foreign key constraint with delete cascade, or a deferred constraint;
- they're executed in a distributed transaction;
- they reference a remote object;
- they reference a user-defined function that can't be executed in parallel (in PL/SQL, use the `PARALLEL_ENABLE` clause to mark functions that support parallel processing);
- an object column is modified; or
- a clustered or temporary table is modified.

Parallel DML statements are *disabled* by default (be careful, this is the opposite of parallel queries). At the session level, you can enable and disable them with the following SQL statements:

```
ALTER SESSION ENABLE PARALLEL DML

ALTER SESSION DISABLE PARALLEL DML
```

In addition, it's also possible to force the parallel execution to a specific degree of parallelism with the following SQL statement:

```
ALTER SESSION FORCE PARALLEL DML PARALLEL 4
```

In contrast to what happens with parallel queries, hints alone can't enable parallel DML statements. In other words, parallel processing of DML statements must be absolutely enabled at the session level to take advantage of it. To check whether parallel DML statements are enabled or disabled at the session level, you can execute a query like the following (the `pdml_status` column is set to either `ENABLED`, `DISABLED`, or `FORCED`):

```
SELECT pdml_status
FROM v$session
WHERE sid = sys_context('userenv','sid')
```

Except for INSERT statements, parallel queries must also be enabled to execute DML statements in parallel. In fact, DML statements are basically composed of two operations: the first finds the rows to be modified, and the second modifies them. The problem is that if the part that finds the rows isn't executed in parallel, the part that modifies the rows can't be parallelized. To illustrate this behavior, let's look at several examples based on the px_dml.sql script.

- When only parallel DML statements are enabled, no operation is parallelized:

```
SQL> ALTER SESSION DISABLE PARALLEL QUERY;

SQL> ALTER SESSION ENABLE PARALLEL DML;

SQL> ALTER TABLE t PARALLEL 2;

SQL> UPDATE t SET id = id + 1;

-----------------------------------
| Id  | Operation          | Name |
-----------------------------------
|   0 | UPDATE STATEMENT   |      |
|   1 |  UPDATE            | T    |
|   2 |   TABLE ACCESS FULL| T    |
-----------------------------------
```

- When only parallel queries are enabled, the update part of the DML statement isn't executed in parallel. In fact, only operations 3 to 5 are executed by slave processes. Therefore, the update part (operation 1) is executed serially by the query coordinator:

```
SQL> ALTER SESSION ENABLE PARALLEL QUERY;

SQL> ALTER SESSION DISABLE PARALLEL DML;

SQL> ALTER TABLE t PARALLEL 2;

SQL> UPDATE t SET id = id + 1;

-----------------------------------------------------------------------
| Id  | Operation             | Name     |    TQ  |IN-OUT| PQ Distrib |
-----------------------------------------------------------------------
|   0 | UPDATE STATEMENT      |          |        |      |            |
|   1 |  UPDATE               | T        |        |      |            |
|   2 |   PX COORDINATOR      |          |        |      |            |
|   3 |    PX SEND QC (RANDOM)| :TQ10000 |  Q1,00 | P->S | QC (RAND)  |
|   4 |     PX BLOCK ITERATOR |          |  Q1,00 | PCWC |            |
|   5 |      TABLE ACCESS FULL| T        |  Q1,00 | PCWP |            |
-----------------------------------------------------------------------
```

- When both parallel queries and DML statements are enabled, the update part (operation 3) is executed in parallel. In this case, only one set of slave processes is used (operations 2 through 5 have the same value in the TQ column). This implies that each slave process scans the granules of the table and modifies the rows as it finds them:

```
SQL> ALTER SESSION ENABLE PARALLEL QUERY;

SQL> ALTER SESSION ENABLE PARALLEL DML;

SQL> ALTER TABLE t PARALLEL 2;

SQL> UPDATE t SET id = id + 1;

-----------------------------------------------------------------------
| Id  | Operation             | Name     |    TQ  |IN-OUT| PQ Distrib |
-----------------------------------------------------------------------
|   0 | UPDATE STATEMENT      |          |        |      |            |
|   1 |  PX COORDINATOR       |          |        |      |            |
|   2 |   PX SEND QC (RANDOM) | :TQ10000 |  Q1,00 | P->S | QC (RAND)  |
|   3 |    UPDATE             | T        |  Q1,00 | PCWP |            |
|   4 |     PX BLOCK ITERATOR |          |  Q1,00 | PCWC |            |
|   5 |      TABLE ACCESS FULL| T        |  Q1,00 | PCWP |            |
-----------------------------------------------------------------------
```

## Parallel DDL Statements

Parallel DDL statements are supported for tables and indexes. The following are operations that are typically parallelized:

- `CREATE TABLE ... AS SELECT ...` (CTAS) statements
- Creation and rebuild of indexes
- Creation and validation of constraints

In addition, partition-management operations such as `COALESCE`, `MOVE`, and `SPLIT` can be parallelized for partitioned tables and indexes. Usually, DDL statements that can take advantage of parallel processing provide the `PARALLEL` clause (as you'll see shortly, constraints are an exception) to specify whether parallel processing should be used and, if it is used, the degree of parallelism.

Parallel DDL statements are *enabled* by default. At the session level, you can enable and disable them with the following SQL statements:

```
ALTER SESSION ENABLE PARALLEL DDL

ALTER SESSION DISABLE PARALLEL DDL
```

You can also force parallel executions with a specific degree of parallelism (for DDL statements that support it) using the following SQL statement:

```
ALTER SESSION FORCE PARALLEL DDL PARALLEL 4
```

To check whether parallel DDL statements are enabled or disabled at the session level, you can execute a query like the following (the `pddl_status` column is set to either `ENABLED`, `DISABLED`, or `FORCED`):

```
SELECT pddl_status
FROM v$session
WHERE sid = sys_context('userenv','sid')
```

The following sections show, for the three main types of DDL statements that can be executed in parallel, several examples based on the `px_ddl.sql` script.

## CTAS Statements

A CTAS statement is composed of two operations that process data: the query used to retrieve data from the source tables and the insert into the target table. Each part can be executed either serially or in parallel independently of the other. However, if parallel processing is used, it's common to parallelize both operations. The following execution plans illustrate this:

- *Parallelization of insert*: Only operations 2 through 4 are executed in parallel. The query coordinator scans table `t1` and distributes its content to the slave processes using the round-robin method. Since the query coordinator communicates with several slave processes, the relationship between the operations is serial to parallel (`S->P`). A set of slave processes receives the data and performs the insert (operation `LOAD AS SELECT`) in parallel:

```
CREATE TABLE t2 PARALLEL 2 AS SELECT /*+ no_parallel(t1) */ * FROM t1

-------------------------------------------------------------------------
| Id  | Operation                | Name     |    TQ  |IN-OUT| PQ Distrib |
-------------------------------------------------------------------------
|   0 | CREATE TABLE STATEMENT   |          |        |      |            |
|   1 |  PX COORDINATOR          |          |        |      |            |
|   2 |   PX SEND QC (RANDOM)    | :TQ10001 |  Q1,01 | P->S | QC (RAND)  |
|   3 |    LOAD AS SELECT        | T2       |  Q1,01 | PCWP |            |
|   4 |     PX RECEIVE           |          |  Q1,01 | PCWP |            |
|   5 |      PX SEND ROUND-ROBIN| :TQ10000 |        | S->P | RND-ROBIN  |
|   6 |       TABLE ACCESS FULL | T1       |        |      |            |
-------------------------------------------------------------------------
```

- *Parallelization of query*: Only operations 3 through 5 are executed in parallel. The slave processes scan table `t1` in parallel based on block range granules and send its content to the query coordinator, which is the reason for the parallel to serial (`P->S`) relationship. The query coordinator executes the insert (operation `LOAD AS SELECT`):

```
CREATE TABLE t2 NOPARALLEL AS SELECT /*+ parallel(t1 2) */ * FROM t1

------------------------------------------------------------------------
| Id  | Operation               | Name     |    TQ  |IN-OUT| PQ Distrib |
------------------------------------------------------------------------
|   0 | CREATE TABLE STATEMENT  |          |        |      |            |
|   1 |  LOAD AS SELECT         | T2       |        |      |            |
|   2 |   PX COORDINATOR        |          |        |      |            |
|   3 |    PX SEND QC (RANDOM)  | :TQ10000 |  Q1,00 | P->S | QC (RAND)  |
|   4 |     PX BLOCK ITERATOR   |          |  Q1,00 | PCWC |            |
|   5 |      TABLE ACCESS FULL  | T1       |  Q1,00 | PCWP |            |
------------------------------------------------------------------------
```

- *Parallelization of both operations*: The slave processes scan table t1 in parallel based on block range granules and insert the data they get in the target table directly, without sending the rows to another parallel slave set. Two important things should be highlighted. First, the query coordinator isn't directly involved in the processing of data. Second, the data isn't sent through a table queue (except for a negligible amount of information sent to the query coordinator by operation 2, no communication takes place):

  ```
  CREATE TABLE t2 PARALLEL 2 AS SELECT /*+ parallel(t1 2) */ * FROM t1

  ----------------------------------------------------------------------
  | Id  | Operation              | Name     |    TQ  |IN-OUT| PQ Distrib |
  ----------------------------------------------------------------------
  |   0 | CREATE TABLE STATEMENT |          |        |      |            |
  |   1 |  PX COORDINATOR        |          |        |      |            |
  |   2 |   PX SEND QC (RANDOM)  | :TQ10000 |  Q1,00 | P->S | QC (RAND)  |
  |   3 |    LOAD AS SELECT      | T2       |  Q1,00 | PCWP |            |
  |   4 |     PX BLOCK ITERATOR  |          |  Q1,00 | PCWC |            |
  |   5 |      TABLE ACCESS FULL | T1       |  Q1,00 | PCWP |            |
  ----------------------------------------------------------------------
  ```

The preceding examples show hints using the object-level syntax. To enable the parallelization of both operations (the last example), from version 11.2 onward, for CTAS statements it's possible to use the statement-level syntax, as shown in the following example:

```
CREATE /*+ parallel(2) */ TABLE t2 AS SELECT * FROM t1
```

An important difference of using this syntax, compared to the previous one, is that because the PARALLEL clause isn't specified, the degree of parallelism is used only during the creation of the table. In other words, in the data dictionary, a degree of parallelism of 1 is associated to the table.

## Creation and Rebuild of Indexes

You can create and rebuild indexes in parallel. To do this, two sets of slave processes work together. The first set reads the data to be indexed. The second set sorts the data it receives from the first set and builds the index. The following SQL statement is an example. Notice how the first set executes operations 6 through 8 (Q1,00) and the second executes operations 2 through 5 (Q1,01). Data is distributed between the two sets using the range method (so that each parallel slave of the second set works on its slice of the index) and a parallel to parallel (P->P) relationship:

```
CREATE INDEX i1 ON t1 (id) PARALLEL 4


------------------------------------------------------------------------
| Id  | Operation                | Name     |    TQ  |IN-OUT| PQ Distrib |
------------------------------------------------------------------------
|   0 | CREATE INDEX STATEMENT   |          |        |      |            |
|   1 |  PX COORDINATOR          |          |        |      |            |
|   2 |   PX SEND QC (ORDER)     | :TQ10001 |  Q1,01 | P->S | QC (ORDER) |
|   3 |    INDEX BUILD NON UNIQUE| I1       |  Q1,01 | PCWP |            |
|   4 |     SORT CREATE INDEX    |          |  Q1,01 | PCWP |            |
|   5 |      PX RECEIVE          |          |  Q1,01 | PCWP |            |
|   6 |       PX SEND RANGE      | :TQ10000 |  Q1,00 | P->P | RANGE      |
|   7 |        PX BLOCK ITERATOR |          |  Q1,00 | PCWC |            |
|   8 |         TABLE ACCESS FULL| T1       |  Q1,00 | PCWP |            |
------------------------------------------------------------------------
```

A rebuild of an index leads to a very similar execution plan (notice that according to operation 8, the data is extracted from the index, not from the table):

```
ALTER INDEX i1 REBUILD PARALLEL 4

-----------------------------------------------------------------------------
| Id  | Operation                    | Name     |    TQ  |IN-OUT| PQ Distrib |
-----------------------------------------------------------------------------
|   0 | ALTER INDEX STATEMENT        |          |        |      |            |
|   1 |  PX COORDINATOR              |          |        |      |            |
|   2 |   PX SEND QC (ORDER)         | :TQ10001 |  Q1,01 | P->S | QC (ORDER) |
|   3 |    INDEX BUILD NON UNIQUE    | I1       |  Q1,01 | PCWP |            |
|   4 |     SORT CREATE INDEX        |          |  Q1,01 | PCWP |            |
|   5 |      PX RECEIVE              |          |  Q1,01 | PCWP |            |
|   6 |       PX SEND RANGE          | :TQ10000 |  Q1,00 | P->P | RANGE      |
|   7 |        PX BLOCK ITERATOR     |          |  Q1,00 | PCWC |            |
|   8 |         INDEX FAST FULL SCAN| I1       |  Q1,00 | PCWP |            |
-----------------------------------------------------------------------------
```

## Creation and Validation of Constraints

When constraints (such as foreign keys and check constraints) are created or validated, the data already stored in the table must be validated. For that purpose, the database engine executes a recursive query. For example, let's say that you execute the following SQL statement:

```
ALTER TABLE t ADD CONSTRAINT t_id_nn CHECK (id IS NOT NULL)
```

Recursively, the database engine executes a query like the following one to validate the data stored in the table (note that if the query returns no row, the data is valid):

```
SELECT rowid
FROM t
WHERE NOT (id IS NOT NULL)
```

As a result, if the table which the constraint is created on has a degree of parallelism of 2 or higher, the database engine executes the query in parallel.

■ **Note** The degree of parallelism defined at the table level is used for the recursive query, regardless of whether at the session level the parallel queries and DDL statements are enabled, forced, or disabled. In other words, the `ALTER SESSION ... PARALLEL` statement has no influence on the recursive query.

When you define a primary key constraint, the database engine can't create the index in parallel. To avoid this limitation, you have to create the (unique) index before defining the constraint. The following SQL statements illustrate:

```
CREATE UNIQUE index t_pk ON t (id) PARALLEL 2

ALTER TABLE t ADD CONSTRAINT t_pk PRIMARY KEY (id)
```

## When to Use It

Parallel processing should be used only when two conditions are met. First, you can use it when plenty of free resources (CPU, memory, and disk I/O bandwidth) are available. Remember, the aim of parallel processing is to reduce the response time by distributing the work usually done by a single process (and hence a single CPU core) to several processes (and hence several CPU cores). Second, you can use it for SQL statements that take more than a dozen seconds to execute serially; otherwise, the time and resources needed to initialize, coordinate and terminate the parallel environment (mainly, the slave processes and the table queues) might be higher than the time gained by the parallelization itself. The actual limit depends on the amount of resources that are available. Therefore, in some situations, only SQL statements that take more than a few minutes, or even longer, are good candidates for being executed in parallel. It's important to stress that if these two conditions aren't met, performance could decrease instead of increase.

If parallel processing is commonly used for many SQL statements, either automatic degree of parallelism is enabled at the system level or manual degree of parallelism is enabled at the segment levels. Otherwise, if it's used only for specific batches or reports, it's usually better to enable it at the session level or through hints.

## Pitfalls and Fallacies

It's very important to understand that the `parallel` and `parallel_index` hints using the object-level syntax don't force the query optimizer to use parallel processing. Instead, they override the degree of parallelism defined at the table or index level. This change, in turn, allows the query optimizer to consider parallel processing with the specified degree of parallelism. This means that the query optimizer considers execution plans with and without parallel processing and, as usual, picks out the one with the lower cost. Let me stress this point by showing you an example based on the `px_dop_manual.sql` script. As shown in the following SQL statements, the cost associated with a full table scan decreases proportionally to the degree of parallelism (refer to Chapter 7 for addition information about the cost of parallel operations):

```
SQL> EXPLAIN PLAN SET STATEMENT_ID 'dop1' FOR
  2  SELECT /*+ full(t) parallel(t 1) */ * FROM t WHERE id > 93000;

SQL> EXPLAIN PLAN SET STATEMENT_ID 'dop2' FOR
  2  SELECT /*+ full(t) parallel(t 2) */ * FROM t WHERE id > 93000;

SQL> EXPLAIN PLAN SET STATEMENT_ID 'dop3' FOR
  2  SELECT /*+ full(t) parallel(t 3) */ * FROM t WHERE id > 93000;

SQL> EXPLAIN PLAN SET STATEMENT_ID 'dop4' FOR
  2  SELECT /*+ full(t) parallel(t 4) */ * FROM t WHERE id > 93000;

SQL> SELECT statement_id, cost
  2  FROM plan_table
  3  WHERE id = 0;

STATEMENT_ID COST
------------ ----
dop1          296
dop2          164
dop3          110
dop4           82
```

If the SQL statement is executed without hints and the degree of parallelism is set to 1, the query optimizer chooses an index range scan:

```
SQL> SELECT * FROM t WHERE id > 93000;

---------------------------------------------------------
| Id  | Operation                   | Name | Cost (%CPU)|
---------------------------------------------------------
|   0 | SELECT STATEMENT            |      |   125   (0)|
|   1 |  TABLE ACCESS BY INDEX ROWID| T    |   125   (0)|
|*  2 |   INDEX RANGE SCAN          | I    |    17   (0)|
---------------------------------------------------------

   2 - access("ID">93000)
```

Notice that the cost associated with the preceding execution plan (125) is lower than the cost of the full table scan with a degree of parallelism up to 2. In contrast, with a degree of parallelism equal to or higher than 3, the full table scan is cheaper.

Now, let's see what happens when only the `parallel` hint is added to the SQL statement—in other words, when no access path hints are used. What happens is that the query optimizer picks out a serial index range scan when the degree of parallelism is set to 2 but chooses a parallel full table scan when the degree of pallelism is set to 3:

```
SQL> SELECT /*+ parallel(t 2) */ * FROM t WHERE id > 93000;

---------------------------------------------------------
| Id  | Operation                   | Name | Cost (%CPU)|
---------------------------------------------------------
|   0 | SELECT STATEMENT            |      |   125   (0)|
|   1 |  TABLE ACCESS BY INDEX ROWID| T    |   125   (0)|
|*  2 |   INDEX RANGE SCAN          | I    |    17   (0)|
---------------------------------------------------------

   2 - filter("ID">93000)

SQL> SELECT /*+ parallel(t 3) */ * FROM t WHERE id > 93000;

------------------------------------------------------
| Id  | Operation             | Name     | Cost (%CPU)|
------------------------------------------------------
|   0 | SELECT STATEMENT      |          |   110   (1)|
|   1 |  PX COORDINATOR       |          |            |
|   2 |   PX SEND QC (RANDOM)| :TQ10000 |   110   (1)|
|   3 |    PX BLOCK ITERATOR |          |   110   (1)|
|*  4 |     TABLE ACCESS FULL| T        |   110   (1)|
------------------------------------------------------

   4 - filter("ID">93000)
```

In summary, the `parallel` and `parallel_index` hints simply allow the query optimizer to consider parallel processing; they don't force it.

To achieve an efficient parallelization, it's critical that the amount of work is evenly distributed among all slave processes. In fact, all slave processes belonging to a set have to wait until all slave processes of the same set have finished. Simply put, a parallel operation is as fast as the slowest slave process. If you want to check the actual distribution for a SQL statement, you can use either Real-time Monitoring (refer to Chapter 4) or the `v$pq_tqstat` dynamic performance view. Basically, the view provides one row for each slave process and for each `PX SEND` and `PX RECEIVE` operation in the execution plan. Just be careful that information is provided only for the current session and only for the last SQL statement successfully executed in parallel. Let's take a look at an example based on the output generated by the `px_tqstat.sql` script. The mapping between the two outputs is performed with the `TQ` column of the execution plan and the `dfo_number` and `tq_id` columns of the `v$pq_tqstat` view. Just remember, as explained before, that the execution plan shows the information about the producers. For example, `Q1,00` maps to `dfo_number` equals 1 and `tq_id` equals 0. In addition, `PX SEND` operations map to producers, and `PX RECEIVE` operations map to consumers:

```
SQL> SELECT * FROM t t1, t t2 WHERE t1.id = t2.id;

-----------------------------------------------------------------------------------------------
| Id  | Operation                   | Name     | Pstart| Pstop |    TQ  |IN-OUT| PQ Distrib |
-----------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT            |          |       |       |        |      |            |
|   1 |  PX COORDINATOR             |          |       |       |        |      |            |
|   2 |   PX SEND QC (RANDOM)       | :TQ10001 |       |       |  Q1,01 | P->S | QC (RAND)  |
|*  3 |    HASH JOIN                |          |       |       |  Q1,01 | PCWP |            |
|   4 |     PX RECEIVE              |          |       |       |  Q1,01 | PCWP |            |
|   5 |      PX SEND PARTITION (KEY)| :TQ10000 |       |       |  Q1,00 | P->P | PART (KEY) |
|   6 |       PX BLOCK ITERATOR     |          |     1 |     2 |  Q1,00 | PCWC |            |
|   7 |        TABLE ACCESS FULL    | T        |     1 |     2 |  Q1,00 | PCWP |            |
|   8 |     PX PARTITION HASH ALL   |          |     1 |     2 |  Q1,01 | PCWC |            |
|   9 |      TABLE ACCESS FULL      | T        |     1 |     2 |  Q1,01 | PCWP |            |
-----------------------------------------------------------------------------------------------

   3 - access("T1"."ID"="T2"."ID")

SQL> SELECT dfo_number, tq_id, server_type, process, num_rows, bytes
  2  FROM v$pq_tqstat
  3  ORDER BY dfo_number, tq_id, server_type DESC, process;

DFO_NUMBER      TQ_ID SERVER_TYP PROCES   NUM_ROWS      BYTES
---------- ---------- ---------- ------ ---------- ----------
         1          0 Producer   P002        29042    3136278
         1          0 Producer   P003        70958    7673358
         1          0 Consumer   P000        20238    2188357
         1          0 Consumer   P001        79762    8621279
         1          1 Producer   P000        20238    4376714
         1          1 Producer   P001        79762   17242534
         1          1 Consumer   QC         100000   21619248
```

That output gives you the following information:

- Operation 5 has sent 29,042 rows with the slave process P002 and 70,958 rows with the slave process P003.
- Operation 4 has received the data sent by operation 5: 20,238 rows with the slave process P000 and 79,762 rows with the slave process P001. This shows that the distribution based on the partition key doesn't work very well in this specific case.

- Operation 2 has sent to the query coordinator 20,238 rows with the slave process P000 and 79,762 rows with the slave process P001. As a result of the previous distribution, this one is also suboptimal.
- Operation 1, which is executed by the query coordinator, has received 100,000 rows.

Each slave process opens its own session to the database instance. This means that if you want to monitor or trace the processing performed to execute a single SQL statement, you can't focus on a single session. Therefore, either you're using a tool like Real-time Monitoring that aggregates the execution statistics coming from several sessions for you, or you have to do it yourself. For example, with SQL trace, every slave process generates its own trace file (the TRCSESS command-line tool might be useful in such a situation). One of the main problems related to this is that the query coordinator, because of a limitation in the current implementation, ignores the execution statistics of the slave processes working for it. The following execution plan generated by the `dbms_xplan` package illustrates this. Notice how the values of the `Starts`, `A-Rows`, and `Buffers` columns are set to 0 except for the operation executed by the query coordinator (`PX COORDINATOR`):

```
SQL> SELECT * FROM table(dbms_xplan.display_cursor('6j5z013saaz9r',0,'iostats last'));

----------------------------------------------------------------------------
| Id  | Operation                    | Name     | Starts | A-Rows | Buffers |
----------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |          |      1 |    100K|      16 |
|   1 |  PX COORDINATOR              |          |      1 |    100K|      16 |
|   2 |   PX SEND QC (RANDOM)        | :TQ10001 |      0 |      0 |       0 |
|*  3 |    HASH JOIN                 |          |      0 |      0 |       0 |
|   4 |     PX RECEIVE               |          |      0 |      0 |       0 |
|   5 |      PX SEND PARTITION (KEY)| :TQ10000 |      0 |      0 |       0 |
|   6 |       PX BLOCK ITERATOR      |          |      0 |      0 |       0 |
|*  7 |        TABLE ACCESS FULL     | T        |      0 |      0 |       0 |
|   8 |     PX PARTITION HASH ALL    |          |      0 |      0 |       0 |
|   9 |      TABLE ACCESS FULL       | T        |      0 |      0 |       0 |
----------------------------------------------------------------------------
```

When dealing with cursors in the library cache, a possible workaround is to *not* specify `last` in the `format` parameter. In fact, if the SQL statement was executed only once (you can check it by looking at the `Starts` column for the `PX COORDINATOR` operation), you will see the sum of the work performed by all slave processes. Unfortunately, that workaround doesn't work when either the execution plan uses several data flow operations or, when, in a RAC environment, part of the slave processes are allocated from one or several remote database instances. The following example, which shows the same child cursor as the previous one, illustrates a case where it works:

```
SQL> SELECT * FROM table(dbms_xplan.display_cursor('6j5z013saaz9r',0,'iostats'));

--------------------------------------------------------------------------------------
| Id  | Operation                    | Name     | Starts | A-Rows | Buffers | Reads  |
--------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |          |      1 |    100K|      16 |      0 |
|   1 |  PX COORDINATOR              |          |      1 |    100K|      16 |      0 |
|   2 |   PX SEND QC (RANDOM)        | :TQ10001 |      0 |      0 |       0 |      0 |
|*  3 |    HASH JOIN                 |          |      2 |    100K|    2719 |   2464 |
|   4 |     PX RECEIVE               |          |      2 |    100K|       0 |      0 |
|   5 |      PX SEND PARTITION (KEY)| :TQ10000 |      0 |      0 |       0 |      0 |
|   6 |       PX BLOCK ITERATOR     |          |      2 |    100K|    2767 |   2464 |
|*  7 |        TABLE ACCESS FULL    | T        |     26 |    100K|    2767 |   2464 |
|   8 |      PX PARTITION HASH ALL  |          |      2 |    100K|    2719 |   2464 |
|   9 |       TABLE ACCESS FULL     | T        |      2 |    100K|    2719 |   2464 |
-------------------------------------------------------------------------------------
```

The session executing a parallel DML statement (and only that session—for other sessions, the uncommitted data isn't even visible) can't access the modified table without committing (or rolling back) the transaction. SQL statements executed before committing (or rolling back) terminate with an `ORA-12838: cannot read/modify an object after modifying it in parallel` error. Here's an example (note that the `UPDATE` statement is parallelized):

```
SQL> UPDATE t SET id = id + 1;

SQL> SELECT count(*) FROM t;
SELECT count(*) FROM t
                     *
ERROR at line 1:
ORA-12838: cannot read/modify an object after modifying it in parallel

SQL> COMMIT;

SQL> SELECT count(*) FROM t;

  COUNT(*)
----------
    100000
```

A restriction similar to the preceding one, is when a parallel DML statement, which is attempting to modify an object that was previously modified with a serial DML statement, raises an `ORA-12839: cannot modify an object in parallel after modifying it` error. Here's an example (note that a `SELECT FOR UPDATE`, to set the row lock, has to modify the table):

```
SQL> SELECT id FROM t WHERE rownum = 1 FOR UPDATE;

        ID
----------
      2343

SQL> UPDATE t SET id = id + 1;
UPDATE t SET id = id + 1
       *
ERROR at line 1:
ORA-12839: cannot modify an object in parallel after modifying it

SQL> COMMIT;

SQL> UPDATE t SET id = id + 1;

100000 rows updated.
```

## Direct-Path Insert

Oracle Database provides two ways for loading data into a table (provided it's not stored in a cluster): conventional inserts and direct-path inserts. *Conventional inserts*, as the name suggests, are the ones that are generally used. *Direct-path inserts* are used only when the database engine is explicitly instructed to do so. The aim of direct-path inserts is to efficiently load large amounts of data (they can have poorer performance than conventional inserts for small amounts of data). They're able to achieve this goal because their implementation is optimized for performance at the expense of functionality. For this reason, they're subject to more requirements and restrictions than conventional inserts. In this section, I discuss how direct-path inserts work, when they should be used, and some common pitfalls and fallacies related to them.

---

■ **Note** To load data CTAS statements, use a direct-path insert.

---

### How It Works

You can enable direct-path inserts either by specifying a hint or by using a specific feature. The following possibilities exist:

- Specify the append hint in `INSERT INTO ... SELECT ...` statements (including multitable inserts) and `MERGE` statements (for the part inserting data):

```
INSERT /*+ append */ INTO ... SELECT ...
```

- Specify the append hint in "regular" `INSERT` statements that use the `VALUES` clause (works only in version 11.1):

```
INSERT /*+ append */ INTO ... VALUES (...)
```

- Specify the `append_values` hint in "regular" `INSERT` statements that use the `VALUES` clause (works only from version 11.2 onward):

```
INSERT /*+ append_values */ INTO ... VALUES (...)
```

- Execute `INSERT INTO ... SELECT ...` statements in parallel. Note that in this case, both the `INSERT` and the `SELECT` can be parallelized independently. To take advantage of direct-path inserts, at least the `INSERT` part must be parallelized.
- Use the OCI direct-path interface either directly or via an application that uses it (for example, the SQL*Loader utility).

If you need to disable a direct-path insert for a SQL statement that automatically enables it (for example, an `INSERT INTO ... SELECT ...` statements executed in parallel), you can specify the noappend hint.

To improve efficiency and, thereby, performance, a direct-path insert uses direct writes to load data directly above the high watermark of the modified segment. This fact has important implications:

- The buffer cache, because of direct writes, is bypassed.
- Concurrent `DELETE`, `INSERT`, `MERGE`, and `UPDATE` statements, as well as the (re)build of indexes on the modified segment, aren't permitted. Naturally, segment locks are obtained to guarantee this.
- The blocks containing free space below the high watermark aren't taken into consideration. This means that even if `DELETE` statements are regularly executed in order to purge data, the size of the segment would increase constantly.

One of the reasons that direct-path inserts lead to better performance is that only minimal undo is generated for the table segment. In fact, undo is generated only for space management operations (for example, to increase the high watermark and add a new extent to the segment), and not for the rows contained in the blocks that are inserted via direct-path. If the table is indexed, however, undo is normally generated for the index segments. If you want to avoid the undo related to index segments as well, you can make the indexes unusable before the load and rebuild them when the load is finished. Especially in ETL jobs, this is common practice; also, it's popular because it may be faster to rebuild the index than to let the database engine do the maintenance at the end of the load.

To further improve performance, you can also use *minimal logging*. The aim of minimal logging is to minimize redo generation. This is optional, but it's often essential to greatly reduce response time. You can instruct minimal logging to be used by setting the `nologging` parameter at the table or partition level. The essential thing to understand is that minimal logging is supported only for direct-path inserts and some DDL statements. In fact, redo is always generated for all other operations. Be aware that minimal logging can't be used for tables stored in clusters.

---

■ **Note** You should specify `nologging` and, thereby, minimize redo generation only if you fully understand the implications of doing so. In fact, media recovery can't be performed for blocks modified with minimal logging. This means that if media recovery is performed, the database engine can only mark those blocks as logically corrupted, because media recovery needs to access the redo information in order to reconstruct the block's contents, and that isn't possible since, as mentioned previously, redo information isn't stored when using minimal logging. As a result, SQL statements that access objects containing such blocks terminate with an `ORA-26040: Data block was loaded using the NOLOGGING option` error. Therefore, you should use minimal logging only if either you can manually reload data, you're willing to make a backup after the load, or you can afford to lose data.

---

Figure 15-11 shows an example of the improvement you can achieve with direct-path inserts. These figures were measured by starting the `dpi_performance.sql` script on my test system.

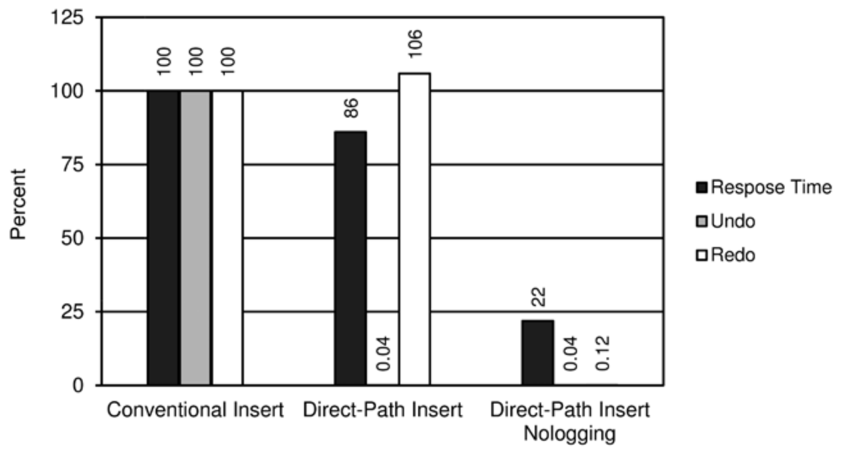


***Figure 15-11.** Comparison of loading data with and without direct-path inserts (table without indexes)*

Notice that in Figure 15-11, the undo generation for both direct-path inserts is negligible. This is because the modified table isn't indexed. Figure 15-12 shows the figures of the same test but with a primary key in place. As expected, undo for the index segment is generated.

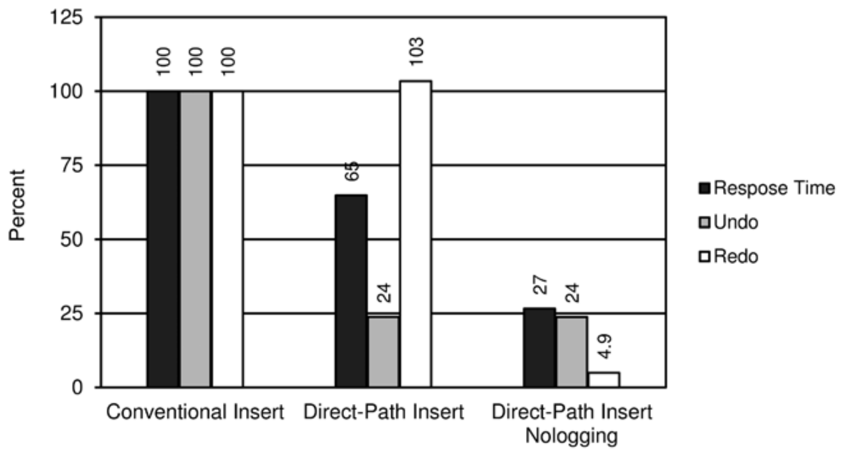


***Figure 15-12.** Comparison of loading data with and without direct-path inserts (table with primary key)*

Direct-path inserts don't support all objects that conventional inserts do. Their functionality is restricted. If the database engine can't execute a direct-path insert, the operation is silently converted into a conventional insert. This happens when one of the following conditions is met:

- An enabled `INSERT` trigger is present on the modified table. (Note that `DELETE` and `UPDATE` triggers have no impact on direct-path inserts.)
- An enabled foreign key is present on the modified table (foreign keys of other tables that point to the modified table aren't a problem).

- The modified table is index organized.
- The modified table is stored in a cluster.
- The modified table contains object type columns.
- The modified table has a primary (or unique) key that is policed via a nonunique index. From version 11.1, this limitation no longer exists.

## When to Use It

You should use direct-path inserts whenever you have to load a large amount of data and the restrictions that apply to direct-path inserts aren't a concern for you.

If performance is your primary goal, you might also consider using minimal logging (`nologging`). As previously explained, however, you should use this possibility only if you fully understand and accept the implications of doing so and if you take necessary measures to not lose data in the process.

## Pitfalls and Fallacies

Even if minimal logging is *not* used, a database running in noarchivelog mode doesn't generate redo for direct-path inserts.

It isn't possible to use minimal logging for segments stored in a database or a tablespace in *force logging* mode. In fact, force logging overrides the `nologging` parameter. Note that force logging is particularly useful when replication features like standby databases and Streams are used. To successfully use them, redo logs need to contain information about all modifications.

During a direct-path insert, the high watermark isn't increased. This operation is performed only when the transaction is committed. Therefore, the session executing a direct-path insert (and only that session—for other sessions, the uncommitted data above the high watermark isn't even visible) can't access the modified table after the load without committing (or rolling back) the transaction. SQL statements executed before committing (or rolling back) terminate with an `ORA-12838: cannot read/modify an object after modifying it in parallel` error. Here's an example:

```
SQL> INSERT /*+ append */ INTO t SELECT * FROM t;

SQL> SELECT count(*) FROM t;
SELECT count(*) FROM t
                     *
ERROR at line 1:
ORA-12838: cannot read/modify an object after modifying it in parallel

SQL> COMMIT;

SQL> SELECT count(*) FROM t;

  COUNT(*)
----------
     10000
```

The text associated with the 0RA-12938 error may be confusing also because it's generated even if no parallel processing is used.

## Row Prefetching

When an application fetches data from a database, it can do so row by row or, better yet, by fetching numerous rows at the same time. Fetching numerous rows at a time is called *row prefetching.*

### How It Works

The concept of row prefetching is straightforward. Every time an application asks the driver to retrieve a row from the database, several rows are prefetched with it and stored in client-side memory. In this way, several subsequent requests don't have to execute database calls to fetch data. They can be served from the client-side memory. As a result, the number of round-trips to the database decreases proportionally to the number of prefetched rows. Hence, the overhead of retrieving result sets with numerous rows may be strongly reduced. As an example, Figure 15-13 shows you the response time of retrieving 100,000 rows by increasing the number of prefetched rows up to 50. The Java class in the `RowPrefetchingPerf.java` file was used for this test.

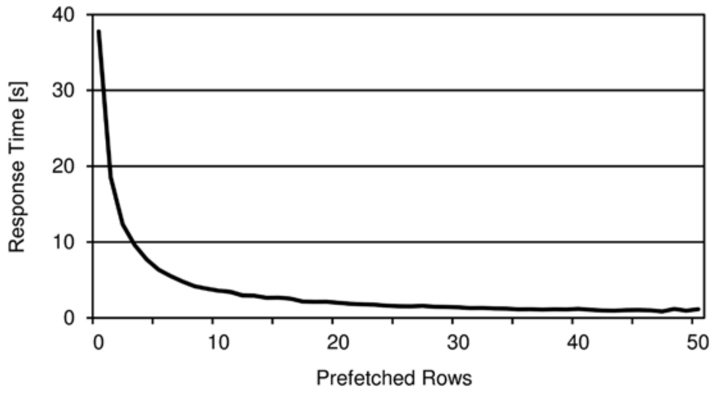


***Figure 15-13.*** *The time needed to retrieve a result set containing numerous rows is strongly dependent on the number of prefetched rows*

It's essential to understand that the poor performance of the retrieval without row prefetching (in other words, row by row processing) is *not* caused by the database engine. Instead, it's the application that causes it and in turn suffers from it. This becomes obvious when looking at the execution statistics generated with SQL trace for the nonprefetching case. The following execution statistics show that even if the client-side elapsed time lasted about 37 seconds (see Figure 15-13), only 2.3 seconds were spent processing the query on the database side!

```
call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        1      0.00       0.00          0          0          0           0
Execute      1      0.00       0.00          0          0          0           0
Fetch   100001      2.14       2.30        213     100004          0      100000
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total   100003      2.14       2.30        213     100004          0      100000
```

Even if row prefetching is much more important for the client, the database also profits from it. In fact, row prefetching greatly reduces the number of logical reads (from 100,004 to 3,542). The following execution statistics show the reduction when 50 rows are prefetched:

```
call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        1      0.00       0.08          0          0          0           0
Execute      1      0.00       0.00          0          0          0           0
Fetch     2001      0.11       0.13        665       3542          0      100000
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total     2003      0.11       0.21        665       3542          0      100000
```

The next sections provide some basic information on how to take advantage of row prefetching with PL/SQL, OCI, JDBC, ODP.NET, and PHP. In addition to the functionalities provided by every API, from version 12.1 onward, the value set by an application can be overridden by Oracle's client `$TNS_ADMIN/oraaccess.xml` configuration file. Note that because it's a client configuration file, the PL/SQL engine isn't impacted by it. However, all applications that are connected through the OCI libraries are. The following example shows how to set row prefetching to 100 for all connections:

```
<?xml version="1.0" encoding="ASCII" ?>
  <oraaccess xmlns="http://xmlns.oracle.com/oci/oraaccess"
             xmlns:oci="http://xmlns.oracle.com/oci/oraaccess"
             schemaLocation="http://xmlns.oracle.com/oci/oraaccess
                             http://xmlns.oracle.com/oci/oraaccess.xsd">
  <default_parameters>
    <prefetch>
      <rows>100</rows>
    </prefetch>
  </default_parameters>
</oraaccess>
```

■ **Note** To take advantage of the `$TNS_ADMIN/oraaccess.xml` configuration file, only the client binaries must be those of version 12.1. In other words, the database version doesn't matter.

Detailed information about the `$TNS_ADMIN/oraaccess.xml` configuration file is provided by the *Oracle Call Interface Programmer's Guide* manual.

## PL/SQL

If, at compile time, the `plsql_optimize_level` initialization parameter is set to 2 (the default value) or higher, row prefetching is used for cursor `FOR` loops. For example, the query in the following PL/SQL block prefetches 100 rows at a time:

```
BEGIN
  FOR c IN (SELECT * FROM t)
  LOOP
    -- process data
    NULL;
  END LOOP;
END;
```

■ **Note** The number of prefetched rows can't be changed.

It's essential to understand that row prefetching is automatically used for cursor FOR loops only. To use row prefetching with other types of cursors, the BULK COLLECT clause must be used. Its utilization with an implicit cursor is shown here:

```
DECLARE
  TYPE t_t IS TABLE OF t%ROWTYPE;
  l_t t_t;
BEGIN
  SELECT * BULK COLLECT INTO l_t
  FROM t;
  FOR i IN l_t.FIRST..l_t.LAST
  LOOP
    -- process data
    NULL;
  END LOOP;
END;
```

With the preceding PL/SQL block, all rows of the result set are returned in a single fetch. If the number of rows is high, a lot of memory is required. Therefore, in practice, either you know that the number of rows that will be returned is limited or you set a limit for a single fetch with the LIMIT clause. The following PL/SQL block shows how to fetch 100 rows at a time:

```
DECLARE
  CURSOR c IS SELECT * FROM t;
  TYPE t_t IS TABLE OF t%ROWTYPE;
  l_t t_t;
BEGIN
  OPEN c;
  LOOP
    FETCH c BULK COLLECT INTO l_t LIMIT 100;
    EXIT WHEN l_t.COUNT = 0;
    FOR i IN l_t.FIRST..l_t.LAST
    LOOP
      -- process data
      NULL;
    END LOOP;
  END LOOP;
  CLOSE c;
END;
```

Row prefetching is supported by the dbms_sql package, native dynamic SQL, and the RETURNING clause. However, as in the previous two examples, it must be explicitly enabled (for example, with BULK COLLECT).

## OCI

With OCI, row prefetching is controlled by two statement attributes: OCI_ATTR_PREFETCH_ROWS and OCI_ATTR_PREFETCH_MEMORY. The former limits the number of fetched rows. The latter limits the amount of memory (in bytes) used to fetch the rows. The following code snippet shows how to call the OCIAttrSet function to set these attributes. The C program in the row_prefetching.c file provides a complete example:

```
ub4 rows = 100;
OCIAttrSet(stm,                    // statement handle
           OCI_HTYPE_STMT,         // type of handle being modified
           &rows,                  // attribute.s value
           sizeof(rows),           // size of the attribute.s value
           OCI_ATTR_PREFETCH_ROWS, // attribute being set
           err);                   // error handle

ub4 memory = 10240;
OCIAttrSet(stm,                      // statement handle
           OCI_HTYPE_STMT,           // type of handle being modified
           &memory,                  // attribute.s value
           sizeof(memory),           // size of the attribute.s value
           OCI_ATTR_PREFETCH_MEMORY, // attribute being set
           err);                     // error handle
```

When both attributes are set, the limit that is reached first is honored. To switch off row prefetching, you must set both attributes to zero.

## JDBC

Row prefetching is enabled with the Oracle JDBC driver by default. You can change the default number of fetched rows (10) in two ways. The first is to specify a property when opening a connection to the database engine with either the OracleDataSource or the OracleDriver class. The following code snippet shows an example where the user, the password, and the number of prefetched rows are set for an OracleDataSource object. Note that in this case, because it's set to 1, row prefetching is disabled:

```
connectionProperties = new Properties();
connectionProperties.put(OracleConnection.CONNECTION_PROPERTY_USER_NAME, user);
connectionProperties.put(OracleConnection.CONNECTION_PROPERTY_PASSWORD, password);
connectionProperties.put(OracleConnection.CONNECTION_PROPERTY_DEFAULT_ROW_PREFETCH, "1");
dataSource.setConnectionProperties(connectionProperties);
```

The second way is to override the default value at the connection level by using the setFetchSize method of either the java.sql.Statement or java.sql.ResultSet interface (and hence, their subinterfaces). The following code snippet shows an example where the setFetchSize method is used to set the number of fetched rows to 100. The Java program in the RowPrefetching.java file provides a complete example:

```
sql = "SELECT id, pad FROM t";
statement = connection.prepareStatement(sql);
statement.setFetchSize(100);
resultset = statement.executeQuery();
```

```
while (resultset.next())
{
  id = resultset.getLong("id");
  pad = resultset.getString("pad");
  // process data
}
resultset.close();
statement.close();
```

## ODP.NET

The default fetch size of ODP.NET (65,536) is defined in bytes, *not* rows. You can change this value through the `FetchSize` property provided by the `OracleCommand` and `OracleDataReader` classes. The following code snippet is an example of how to set the value for fetching 100 rows. Notice how the `RowSize` property of the `OracleCommand` class is used to compute the amount of memory needed to store the 100 rows. The C# program in the `RowPrefetching.cs` file provides a complete example:

```
sql = "SELECT id, pad FROM t";
command = new OracleCommand(sql, connection);
reader = command.ExecuteReader();
reader.FetchSize = command.RowSize * 100;
while (reader.Read())
{
  id = reader.GetDecimal(0);
  pad = reader.GetString(1);
  // process data
}
reader.Close();
```

As of ODP.NET version 10.2.0.3, you can also change the default fetch size through the following registry entry (`<Assembly_Version>` is the full version number of `Oracle.DataAccess.dll`):

```
HKEY_LOCAL_MACHINE\SOFTWARE\ORACLE\ODP.NET\<Assembly_Version>\FetchSize
```

## PHP

Row prefetching is enabled with the PECL OCI8 extension by default. You can change the default number of fetched rows (100—through version 1.3.3 of the extension it's 10) in two ways. The first is to change the default value by setting the `oci8.default_prefetch` option in the `php.ini` configuration file. The second is to override the default at the statement level by calling the `oci_set_prefetch` function between the parse and the execution phase. The following code snippet is an example of how to set the value to fetch 100 rows. The `RowPrefetching.php` script provides a complete example:

```
$sql = "SELECT id, pad FROM t";
$statement = oci_parse($connection, $sql);
oci_set_prefetch($statement, 100);
oci_execute($statement, OCI_NO_AUTO_COMMIT);
```

```
while ($row = oci_fetch_assoc($statement))
{
  $id = $row['ID'];
  $pad = $row['PAD'];
  // process data
}
oci_free_statement($statement);
```

### When to Use It

Whenever more than one row has to be fetched, using row prefetching makes sense.

### Pitfalls and Fallacies

When OCI libraries are used, it isn't always possible to fully disable row prefetching. For example, with the JDBC OCI driver or with SQL*Plus, the minimum number of fetched rows is two. In practice, this isn't a problem, but there may be some confusion as to why, for example, in spite of setting the `arraysize` system variable to 1 in SQL*Plus, you see that two rows are fetched.

If an application displays, let's say, 10 rows at time, it's usually pointless to prefetch 100 rows from the database. The number of prefetched rows should ideally be the same as the number of rows needed by an application at a given time.

## Array Interface

The preceding section shows that when an application fetches data from a database, it can do it row by row or, even better, by using row prefetching. The same concept applies to the situations where the application passes data to the database engine or, in other words, during the binding of input variables. For this purpose, the *array interface* is available.

### How It Works

The array interface allows you to bind arrays instead of scalar values. This is very useful when a specific DML statement needs to insert or modify numerous rows. Instead of executing the DML statement separately for each row, you can bind all necessary values as arrays and execute it only once, or if the number of rows is high, you can split the execution into smaller batches. As a result, the number of round-trips to the database decreases proportionally to the array size. Figure 15-14 shows the response time of inserting 100,000 rows by increasing the size of the arrays up to 50. The Java class in the `ArrayInterfacePerf.java` file was used for this test.

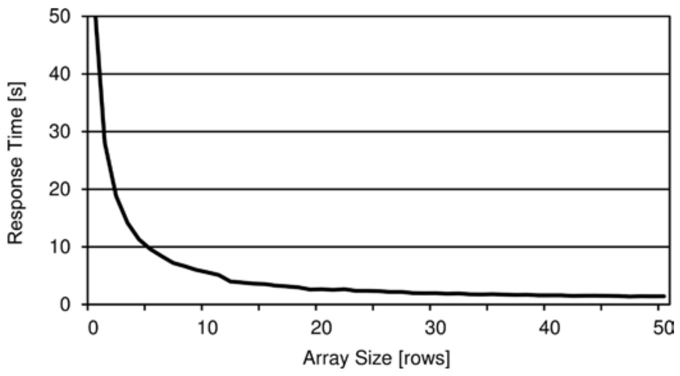


***Figure 15-14.*** *The time needed to load data into the database is strongly dependent on the number of rows processed by each execution*

It's essential to understand that poor performance of the load without array processing (in other words, row by row processing) is *not* due to the database engine. Instead, it's the application that causes and suffers from it. You can clearly see this by looking at the execution statistics generated with SQL trace. The following execution statistics show that even if the client-side elapsed time lasted more than 50 seconds (see Figure 15-14), only 3.1 seconds were spent processing the database side inserts:

```
call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        1      0.00       0.00          0          0          0           0
Execute 100000      3.06       3.10          2       2075     114173      100000
Fetch        0      0.00       0.00          0          0          0           0
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total   100001      3.06       3.10          2       2075     114173      100000
```

Even if the array interface is much more effective for the client, the database engine also profits from it. In fact, the array interface reduces the number of logical reads (from 116,248 to 18,143). The following execution statistics show the reduction when the rows are inserted in batches of 50:

```
call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        1      0.00       0.00          0          0          0           0
Execute   2000      0.26       0.38          0       3132      15011      100000
Fetch        0      0.00       0.00          0          0          0           0
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total     2001      0.26       0.38          0       3132      15011      100000
```

The next sections provide some basic information on how to take advantage of the array interface with PL/SQL, OCI, JDBC, and ODP.NET. Note that PHP, through the PECL OCI8 extension, doesn't support the array interface. Given the kind of applications that are developed with PHP, I don't consider this a major issue.

## PL/SQL

To use the array interface in PL/SQL, the FORALL statement is available. With it, you can execute a DML statement that binds arrays to pass data to the database engine. The following PL/SQL block shows how to insert 100,000 rows with a single execution. Notice that the first part of the code is used only to prepare the arrays. The FORALL statement itself with the INSERT statement takes only the last two lines of the PL/SQL block:

```
DECLARE
  TYPE t_id IS TABLE OF t.id%TYPE;
  TYPE t_pad IS TABLE OF t.pad%TYPE;
  l_id t_id := t_id();
  l_pad t_pad := t_pad();
BEGIN
  -- prepare data
  l_id.extend(100000);
  l_pad.extend(100000);
  FOR i IN 1..100000
  LOOP
    l_id(i) := i;
    l_pad(i) := rpad('*',100,'*');
  END LOOP;
  -- insert data
  FORALL i IN l_id.FIRST..l_id.LAST
    INSERT INTO t VALUES (l_id(i), l_pad(i));
END;
```

It's important to note that even if the syntax is based on the keyword FORALL, this isn't a loop. All rows are sent in a single database call.

The array interface is supported not only in this case, but the dbms_sql package and native dynamic SQL also support it.

## OCI

To take advantage of the array interface with OCI, no specific function is needed. In fact, the functions used to bind the variables, OCIBindByPos and OCIBindByName, and the function used to execute the SQL statement, OCIStmtExecute, can work with arrays as parameters. The C program in the array_interface.c file provides an example.

## JDBC

To use the array interface with JDBC, *batch updates* are available. As shown in the following code snippet, which inserts 100,000 rows in a single execution, you can add an "execution" to a batch by executing the addBatch method. When the whole batch containing several executions is then ready, it can be submitted to the database engine by executing the executeBatch method. Both methods are available in the java.sql.Statement interface and, consequently, in the subinterfaces java.sql.PreparedStatement and java.sql.CallableStatement as well. The Java program in the ArrayInterface.java file provides a complete example:

```
sql = "INSERT INTO t VALUES (?, ?)";
statement = connection.prepareStatement(sql);
for (int i=1 ; i<=100000 ; i++)
```

```
{
  statement.setInt(1, i);
  statement.setString(2, "... some text ...");
  statement.addBatch();
}
counts = statement.executeBatch();
statement.close();
```

---

■ **Caution** According to the JDBC standard, batch updates are supported by the `java.sql.Statement` interface and its subinterfaces `java.sql.PreparedStatement` and `java.sql.CallableStatement`. Even though Oracle's implementation supports the standard API, you should expect performance improvements only when using the `java.sql.PreparedStatement` interface to repeatedly execute the same SQL statement with different bind variables.

---

## ODP.NET

To use the array interface with ODP.NET, it's enough to define parameters based on arrays and to set the `ArrayBindCount` property to the number of values stored in the arrays. The following code snippet, which inserts 100,000 rows in a single execution, illustrates this. You can find a complete example in the C# program in the `ArrayInterface.cs` file:

```
Decimal[] idValues = new Decimal[100000];
String[] padValues = new String[100000];

for (int i=0 ; i<100000 ; i++)
{
  idValues[i] = i;
  padValues[i] = "... some text ...";
}

id = new OracleParameter();
id.OracleDbType = OracleDbType.Decimal;
id.Value = idValues;

pad = new OracleParameter();
pad.OracleDbType = OracleDbType.Varchar2;
pad.Value = padValues;

sql = "INSERT INTO t VALUES (:id, :pad)";
command = new OracleCommand(sql, connection);
command.ArrayBindCount = idValues.Length;
command.Parameters.Add(id);
command.Parameters.Add(pad);
command.ExecuteNonQuery();
```

### When to Use It

Whenever more than one row has to be inserted or modified, using the array interface makes sense. You just have to take into consideration that more memory is used on the client to store the array. Usually, this isn't an issue unless an exaggerated array size is used.

### Pitfalls and Fallacies

In the execution statistics generated through SQL trace, there is no explicit information about the utilization of the array processing. However, if you know the SQL statement, by looking at the ratio between the number of modified rows and the number of executions, you should be able to identify whether array processing was used. For example, in the following execution statistics, a plain `INSERT` statement, which was executed only once, inserted 2,342 rows. Something like that is possible only when the array interface is used:

```
INSERT INTO T VALUES (:B1 , :B2 )

call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        1      0.00       0.00          0          0          0           0
Execute      1      0.00       0.00          0         78        522        2342
Fetch        0      0.00       0.00          0          0          0           0
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total        2      0.00       0.00          0         78        522        2342
```

## On to Chapter 16

This chapter describes some advanced optimization techniques aimed at improving performance. Some of them (materialized views, result caches, parallel processing, and direct-path inserts) should be used only if with "regular" optimization techniques it isn't possible to achieve the required performance. In contrast, others (row prefetching and array processing) should always be used if possible.

Although this chapter mainly describes optimization techniques that aren't commonly used, the next (and last) chapter covers techniques that basically apply to every table stored in a database. In fact, when you perform the translation from logical design to physical design, it's necessary to decide how data is physically stored for every table.

CHAPTER 16

# Optimizing the Physical Design

During the translation from the logical design to the physical design, you must make four kinds of decisions. First, for each table, you have to decide not only whether you should use a heap table, a cluster, or an index-organized table, but also whether it has to be partitioned. Second, you must consider whether you should utilize redundant access structures such as indexes and materialized views. Third, you have to decide how to implement the constraints (not *whether* you have to implement them). Fourth, you have to decide how data will be stored in blocks, including the order of the columns, what datatypes are to be used, how many rows per block should be stored, or whether compression should be activated. This chapter focuses on the fourth topic only. For information about the others, especially the first two, refer to Chapters 13, 14, and 15.

The aim of this chapter is to explain why the optimization of the physical design shouldn't be seen as a fine-tuning activity but as a basic optimization technique. The chapter starts by discussing why choosing the correct column order and the correct datatype is essential. It continues by explaining what row migration and row chaining are, how to identify problems related to them, and how to avoid row migration and row chaining in the first place. Then, it describes a common performance issue experienced by systems with a high workload: block contention. Finally, it describes how to take advantage of data compression to improve performance.

## Optimal Column Order

Little care is generally taken to find the optimal column order for a table. Depending on the situation, this might have no impact at all or may cause a significant overhead. To understand what situations this might cause significant overhead in, it's essential to describe how the database engine stores rows into blocks.

A row stored into a block has a very simple format (see Figure 16-1). First, there's a header (H) recording some properties about the row itself, such as whether it's locked or how many columns it contains. Then, there are the columns. Because every column might have a different size, each of them consists of two parts. The first is the length (L*n*) of the data. The second is the data (D*n*) itself.

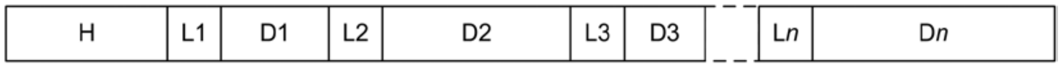


***Figure 16-1.*** *Format of a row stored in a database block (H = row header, Ln = length of column n, Dn = data of column n)*

The essential thing to understand in this format is that the database engine doesn't know the offset of the columns in a row. For example, if it has to locate column 3, it has to start by locating column 1 (that's simple, since the length of the header is known). Then, based on the length of column 1, it locates column 2. Finally, based on the length of column 2, it locates column 3. So whenever a row has more than a few columns, a column near the beginning of the row might be located much faster than a column near the end of the row. To better understand this, you can perform the following test, based on the `column_order.sql` script, to measure the overhead associated with the search of a column:

1. Create a table with 250 columns:

   ```
   CREATE TABLE t (n1 NUMBER, n2 NUMBER, ..., n249 NUMBER, n250 NUMBER)
   ```

2. Insert 10,000 rows. Every column of every row stores the same value.
3. Measure the response time for the following query, executed 1,000 times in a loop, for each column:

   ```
   SELECT count(<col>) FROM t
   ```

Figure 16-2 summarizes the results of this test run on my test server. It's important to note that the query referencing the first column (position 1) performs about five times faster than the query referencing the 250th column (position 250). This is because the database engine optimizes every access and thus avoids locating and reading columns that aren't necessary for the processing. For example, the query `SELECT count(n3) FROM t` stops walking the row when the third column is located. Figure 16-2 also reports, at position 0, the figure for `count(*)`, which doesn't need to access any column at all.

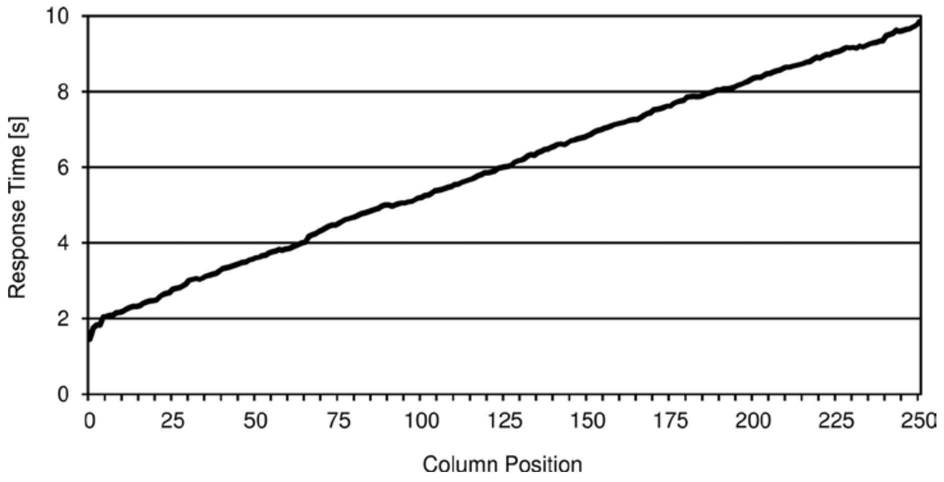


***Figure 16-2.*** *The position of a column in a row vs. the amount of processing needed to access it*

Because of this, the general rule is to place intensively accessed columns first. However, in order to take advantage of this, you should be careful to access (reference) only the columns that are really needed. In any case, from a performance point of view, selecting not-needed columns (or worse, as it's sadly very often done, referencing all columns using a `SELECT *` even if only some of them are actually needed by the application) is bad, not only because there's an overhead when reading them from blocks, as you have just seen, but also because more memory is needed to temporarily store them on the server and on the client and because more time and resources are needed to send data over the network. Simply put, every time data is processed, there's an overhead.

In practice, the overhead related to the position of the columns is (more) noticeable in one of the following situations:

- When tables have many columns, and SQL statements frequently reference very few of the ones located near the end of the row.
- When many rows are read from a single block, such as during a full table scan. This is because more often than not, when accessing few rows per block, the overhead for locating and accessing a block is by far more significant than the one for locating and accessing the columns if few rows are read. For example, if I run the `column_order.sql` script by setting `PCTFREE` to 90 (hence I decrease the number of rows per block), the query referencing the first column performs less than two times faster than the query referencing the 250th column (as shown in Figure 16-2, it's about five times faster with `PCTFREE` set to 10).

Since trailing `NULL` values aren't stored, it makes sense to place columns expected to contain `NULL` values at the end of the table. In this way, the number of physically stored columns and consequently the average size of the rows might decrease.

## Optimal Datatype

As briefly described in Chapter 1, specifically in the "Designing for Performance" section, in recent years I have witnessed a worrying trend in physical design that I call *wrong datatype selection*. At first glance, choosing the datatype for a column seems like a very straightforward decision to make. Nevertheless, in a world where software architects spend a lot of time discussing high-level things such as agile software development, SOA, or persistence frameworks, most people seem to forget about low-level ones. I'm convinced it's essential to get back to the basics and discuss why datatype selection is important.

### Pitfalls in Datatype Selection

To illustrate wrong datatype selection, I present five examples of typical problems that I have encountered over and over again.

The first problem caused by wrong datatype selection is wrong or lacking validation of data when it's inserted or modified in the database. For example, if a column is supposed to store numeric values, choosing a character string datatype for it calls for an external validation. In other words, the database engine isn't able to validate the data. It leaves it to the application to do. Even if such a validation is easy to implement, bear in mind that every time the same piece of code is spread to several locations, instead of being centralized in the database, sooner or later there will be a mismatch in functionality (typically, in some locations the validation may be forgotten, or maybe the validation changes later and its implementation is updated only in some locations). The example I'm presenting is related to the `nls_numeric_characters` initialization parameter. Remember that this initialization parameter specifies the characters used as decimals and group separators. For example, in Switzerland it's usually set to ".,", and therefore the value pi is formatted as follows: 3.14159. Instead, in Germany it's commonly set to ",.", and therefore the same value is formatted as follows: 3,14159. Sooner or later, running an application with different client-side settings of this initialization parameter will cause an `ORA-01722: invalid number error` if conversions from `VARCHAR2` to `NUMBER` take place because of using a wrong datatype in the database. And by the time you notice this, your database will be filled with `VARCHAR2` columns containing both formats, and therefore a painful data reconciliation will have to be performed.

The second problem caused by wrong datatype selection is loss of information. In other words, during the conversion of the original (correct) datatype to the database datatype, information gets lost. For example, imagine what happens when the date and time of an event is stored with a `DATE` datatype instead of a `TIMESTAMP WITH TIME ZONE` datatype. Fractional seconds and time zone information get lost. Although the former leads to what could be considered a small error (less than 1 second), the latter might be a bigger problem. In one case that I witnessed, a customer's data was always generated using local standard time (without daylight saving time adjustments) and stored directly in the database. The problems arose when, for reporting purposes, a correction for daylight saving time had to be applied. A function designed to make a conversion between two time zones was implemented. Its signature was the following:

```
new_time_dst(in_date DATE, tz1 VARCHAR2, tz2 VARCHAR2) RETURN DATE
```

Calling such a function once was very fast. The problem was calling it thousands of times for each report. The response time increased by a factor of 25 as a result. Clearly, with the correct datatype, everything would be not only faster, but also easier (the conversion would be performed automatically).

The third problem caused by wrong datatype selection is that things don't work as expected. Let's say you have to range-partition a table, based on a `DATE` or `TIMESTAMP` column storing date and time information. This is usually no big deal. The problem arises if the column used as the partition key contains the numeric representation of the datetime value based on some format mask, or an equivalent string representation, instead of plain `DATE` or `TIMESTAMP` values. If the conversion from the datetime values to the numeric values is performed with a format mask like `YYYYMMDDHH24MISS`, the definition of the range partitions is still possible. However, if the conversion is based on a format mask like `DDMMYYYYHH24MISS`, you have no chance of solving the problem without changing the datatype or format of the column since the numeric (or string) order doesn't preserve the natural datetime value order (as of version 11.1, in some cases it's possible to work around the problem by implementing virtual column based partitioning).

The fourth problem caused by wrong datatype selection is related to the query optimizer. This is probably the least obvious of this short list and also the one leading to the subtlest problems. The reason for this is that with the wrong datatypes, the query optimizer will perform wrong estimates and, consequently, will choose suboptimal access paths. Frequently, when something like that happens, most people blame the query optimizer that "once again" isn't doing its job correctly. In reality, the problem is that information is hidden from it, so the query optimizer can't do its job correctly. To better understand this problem, take a look at the following example, which is based on the `wrong_datatype.sql` script. Here, you're checking the estimated cardinality of similar restrictions based on three columns that store the same set of data (the date of each day in 2014) but that are based on different datatypes. As you can see, the query optimizer is able to make a sensible estimation (the correct cardinality is 28) only for the column that's correctly defined:

```
SQL> CREATE TABLE t (d DATE, n NUMBER(8), c VARCHAR2(8));

SQL> INSERT INTO t (d)
  2  SELECT to_date('20140101','YYYYMMDD')+level-1
  3  FROM dual
  4  CONNECT BY level <= 365;

SQL> UPDATE t SET n = to_number(to_char(d,'YYYYMMDD')), c = to_char(d,'YYYYMMDD');

SQL> execute dbms_stats.gather_table_stats(ownname=>user, tabname=>'t')

SQL> SELECT * FROM t ORDER BY d;

D                  N C
--------- ---------- --------
01-JAN-14   20140101 20140101
02-JAN-14   20140102 20140102
...
```

```
30-DEC-14   20141230 20141230
31-DEC-14   20141231 20141231

SQL> EXPLAIN PLAN SET STATEMENT_ID = 'd' FOR
  2  SELECT *
  3  FROM t
  4  WHERE d BETWEEN to_date('20140201','YYYYMMDD') AND to_date('20140228','YYYYMMDD');

SQL> EXPLAIN PLAN SET STATEMENT_ID = 'n' FOR
  2  SELECT *
  3  FROM t
  4  WHERE n BETWEEN 20140201 AND 20140228;

SQL> EXPLAIN PLAN SET STATEMENT_ID = 'c' FOR
  2  SELECT *
  3  FROM t
  4  WHERE c BETWEEN '20140201' AND '20140228';

SQL> SELECT statement_id, cardinality FROM plan_table WHERE id = 0;

STATEMENT_ID CARDINALITY
------------ -----------
d                     29
n                     11
c                     11
```

The fifth problem is also related to the query optimizer. This one, however, is because of implicit conversion (as a general rule, you should always avoid implicit conversion). What might happen is that an implicit conversion prevents the query optimizer from choosing an index. To illustrate this problem, the same table as in the previous example is used. For this table, an index based on the column of datatype VARCHAR2 is created. If a WHERE clause contains a restriction on that column that uses a character string, the query optimizer picks out the index. However, if the restriction uses a number (the developer "knows" that only numeric values are stored in it. . .), a full table scan is used (notice, in the second SQL statement, that the implicit conversion based on the `to_number` function prevents the index from being used) and as a result the query optimizer correctly ignores the index.

```
SQL> CREATE INDEX i ON t (c);

SQL> SELECT /*+ index(t) */ *
  2  FROM t
  3  WHERE c = '20140228';

--------------------------------------------
| Id  | Operation                   | Name |
--------------------------------------------
|   0 | SELECT STATEMENT            |      |
|   1 |  TABLE ACCESS BY INDEX ROWID| T    |
|*  2 |   INDEX RANGE SCAN          | I    |
--------------------------------------------

   2 - access("C"='20140228')
```

```
SQL> SELECT /*+ index(t) */ *
  2  FROM t
  3  WHERE c = 20140228;

----------------------------------
| Id  | Operation         | Name |
----------------------------------
|   0 | SELECT STATEMENT  |      |
|*  1 |  TABLE ACCESS FULL| T    |
----------------------------------

   1 - filter(TO_NUMBER("C")=20140228)
```

In summary, there are plenty of good reasons for selecting your datatypes correctly. Doing so may just save you a lot of problems.

## Best Practices in Datatype Selection

As discussed in the previous section, the key principle is that every datatype should store only values for which it has been designed. For instance, a number must be stored in a numeric datatype and not in a character string datatype. In addition, whenever several datatypes exist (for example, several datatypes that may store character strings), it's important to choose the one that is able to fully store the data in the most efficient way. In other words, you should avoid losing information or performance.

The following sections provide some information (mainly related to performance) that you should consider when selecting a datatype. They cover the four main categories of built-in datatypes: numbers, character strings, bit strings, and datetimes.

### Numbers

The main datatype used to store floating-point numbers and integers is NUMBER. It's a variable-length datatype. This means it's possible to specify, through the *precision* and *scale*, the accuracy used to store data. Whenever this datatype is used to store integers or whenever full accuracy isn't needed, it's important to specify the scale in order to save space. The following example shows how the same input value is rounded to either 21 bytes or 2 bytes, depending on the scale:

```
SQL> CREATE TABLE t (n1 NUMBER, n2 NUMBER(*,2));

SQL> INSERT INTO t VALUES (1/3, 1/3);

SQL> SELECT * FROM t;

                                        N1  N2
---------------------------------------- ---
.3333333333333333333333333333333333333333 .33

SQL> SELECT vsize(n1), vsize(n2) FROM t;

VSIZE(N1) VSIZE(N2)
--------- ---------
       21         2
```

Since the internal format is proprietary, a CPU can't directly process values stored as `NUMBER` using the hardware floating-point unit. Instead, they're processed by internal Oracle library routines. For this reason, the `NUMBER` datatype isn't efficient when supporting number-crunching loads. To solve this problem, `BINARY_FLOAT` and `BINARY_DOUBLE` are available. Their key advantage over the `NUMBER` datatype is that they implement the IEEE 754 standard, so a CPU can directly process them. Their key disadvantage is that they are based on binary floating-point. Therefore, these types cannot accurately represent decimal fractions that are so frequently used in commercial and financial applications. Table 16-1 summarizes the key differences between these three datatypes.

***Table 16-1.*** *Comparing Number Datatypes*

| Property | NUMBER(precision, scale) | BINARY_FLOAT | BINARY_DOUBLE |
|---|---|---|---|
| Range of values | ±1.0E126 | ±3.40E38 | ±1.79E308 |
| Size | 1–22 bytes | 4 bytes | 8 bytes |
| Support ±infinity | Yes | Yes | Yes |
| Support NAN | No | Yes | Yes |
| Advantages | Accuracy | Speed | Speed |
| | Precision and scale can be specified | Fixed length | Fixed length |

## Character Strings

There are three basic datatypes used for storing character strings: `VARCHAR2`, `CHAR`, and `CLOB`. The first two support up to 4,000 and 2,000 bytes, respectively (note that the maximum length is specified in bytes, not characters). The third supports up to several terabytes of data (the actual value depends on the default block size). The main difference between `VARCHAR2` and `CHAR` is that the former is of variable length, while the latter is of fixed length. This means `CHAR` is usually used when the length of the character string is known. However, my advice is to always use `VARCHAR2` because it provides better performance than `CHAR`. `CLOB` should be used only when character strings are expected to be larger than the maximum size of `VARCHAR2`. From version 11.1 onward, there are two storage methods for `CLOB`: `basicfile` and `securefile`. For better performance, you should use `securefile`.

When using `VARCHAR2` and `CHAR` datatypes, the maximum length shouldn't be set too high unnecessarily. This is because even though the full space might not be used, in some situations the database engine has to allocate enough memory to store whatever maximum length you specify. Hence, large amounts of memory might be allocated for nothing.

The three basic datatypes store character strings according to the database character set. In addition, three other datatypes, `NVARCHAR2`, `NCHAR`, and `NCLOB`, are available to store character strings according to the *national character set* (a secondary Unicode character set defined at the database level). These three datatypes have the same characteristics as the basic ones with the same name. Only their character set is different.

`LONG` is another character string datatype that has been deprecated in favor of `CLOB`. You should no longer use it; it's provided for backward compatibility only.

---

■ **Caution** From version 12.1 onward, you can increase the maximum size of `VARCHAR2`, `NVARCHAR2`, and `RAW` to 32,767 bytes by setting the `max_string_size` initialization parameter to `extended`. The drawback of doing so is that the database engine silently uses LOB datatypes to support that larger maximum size. My advice is to leave the `max_string_size` initialization parameter to the default (`standard`) and, if more space is required, to explicitly use LOB datatypes.

---

### Bit Strings

Two datatypes are used for storing bit strings: `RAW` and `BLOB`. The first supports up to 2,000 bytes. The second should be used only when bit strings are expected to be larger than 2,000 bytes. From version 11.1 onward, there are two storage methods for `BLOB`: `basicfile` and `securefile`. For better performance, you should use `securefile`.

Another bit string datatype is `LONG RAW`, but it has been deprecated in favor of `BLOB`. You should no longer use it; it's provided for backward compatibility only.

### Datetimes

The datatypes used to store datetime values are `DATE`, `TIMESTAMP`, `TIMESTAMP WITH TIME ZONE`, and `TIMESTAMP WITH LOCAL TIME ZONE`. All of them store the following information: year, month, day, hours, minutes, and seconds. The length of this part is fixed at 7 bytes. The three datatypes based on `TIMESTAMP` might also store the fractional part of seconds (0–9 digits, per default 6). This part is of variable length: 0–4 bytes. Lastly, `TIMESTAMP WITH TIME ZONE` stores the time zone in two additional bytes. Since all of them store different information, the best-suited datatype is the one that takes the minimum amount of space for storing the required data.

## Row Migration and Row Chaining

Migrated and chained rows are often confused. In my opinion, this is for two main reasons. First, they share some characteristics, so it's easy to confuse them. Second, Oracle, in its documentation and in the implementation of its software, has never been very consistent in distinguishing them. So, before describing how to detect and avoid them, it's essential to briefly describe the differences between the two.

### Migration vs. Chaining

When rows are inserted into a block, the database engine reserves some free space for future updates. You define the amount of free space reserved for updates by using the `PCTFREE` parameter. To illustrate, I inserted six rows in the block depicted in Figure 16-3. Since the limit set through `PCTFREE` has been reached, this block is no longer available for inserts.

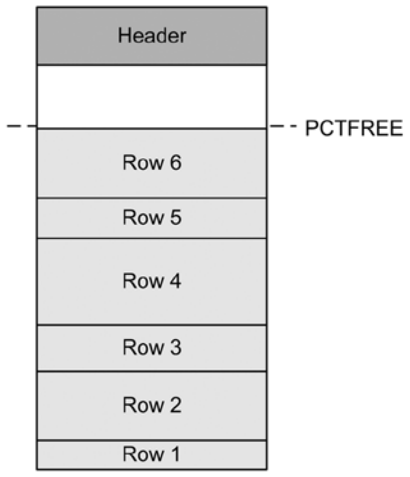


***Figure 16-3.*** *Inserts leave some free space for future updates*

When a row is updated and its size increases, the database engine tries to find enough free space in the block where it's stored. When not enough free space is available, the row is split into two pieces. The first piece (containing only control information, such as a rowid pointing to the second piece) remains in the original block. This is necessary to avoid changing the rowid. Note that this is crucial because rowids might not only be permanently stored in indexes by the database engine, but also be temporarily stored in memory by client applications. The second piece, containing all the data, goes into another block. This kind of row is called a *migrated row*. For example, in Figure 16-4, row 4 has been migrated.

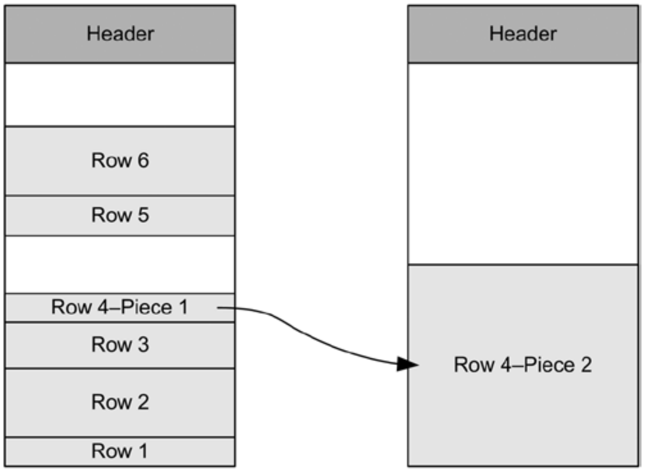


***Figure 16-4.*** *Updated rows that can no longer be stored in the original block are migrated to another one*

When a row is too big to fit into a single block, it's split into two or more pieces. Then, each piece is stored in a different block, and a chain between the pieces is built. This type of row is called a *chained row*. To illustrate, Figure 16-5 shows a row that is chained over three blocks.

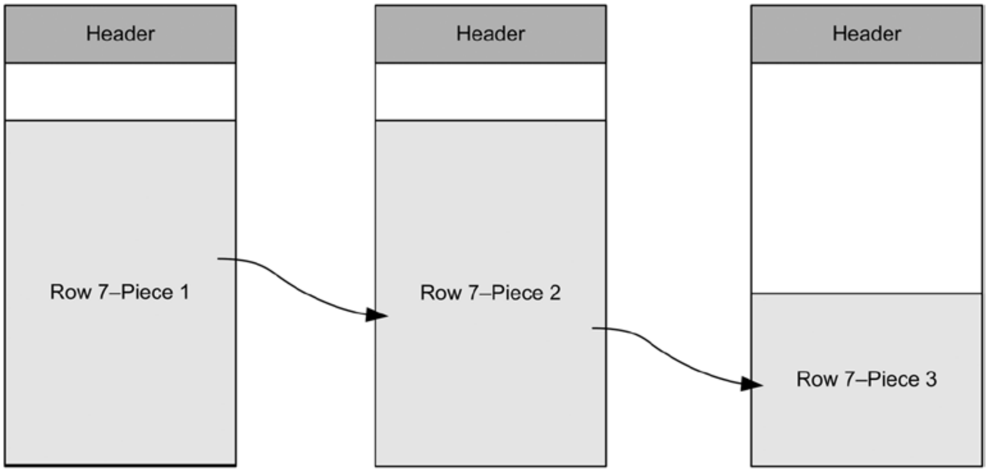


***Figure 16-5.*** *A chained row is split into two or more pieces*

There's a second situation that causes row chaining: tables with more than 255 columns. In fact, the database engine isn't able to store more than 255 columns in a single row piece. Consequently, whenever more than 255 columns have to be stored, the row is split into several pieces. This situation is particular, in that several pieces belonging to the same row can also be stored in a single block. This is called *intra-block row chaining*. Figure 16-6 shows a row with three pieces (since it has 654 columns).

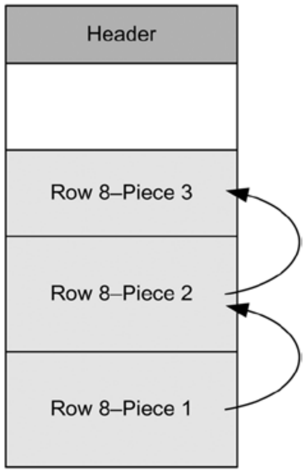


***Figure 16-6.*** *Tables with more than 255 columns might cause intra-block row chaining*

Note that migrated rows are caused by updates, while chained rows are caused by either inserts or updates. When chained rows are caused by updates, the rows might be migrated and chained at the same time.

## Problem Description

The impact on performance caused by row migration depends on the access path used to read the rows. If they're accessed by rowid, the cost doubles. In fact, both row pieces have to be accessed separately. Instead, there's no overhead if they're accessed through full scans. This is because the first row piece, which contains no data, is simply skipped.

The impact on performance caused by chaining is independent of the access path. In fact, every time the first piece is found, it's necessary to read all the other pieces through the rowid as well. There's one exception, however. As discussed previously in the "Optimal Column Order" section, when only part of a row is needed, not all pieces may need to be accessed. For example, if only columns stored in the first piece are needed, there's no need to access all other pieces.

An overhead that applies to both migration and chaining is related to row-level locking. Every row piece has to be locked. This means that the overhead due to the lock increases proportionally with the number of pieces.

## Problem Identification

There are two main techniques for detecting migrated and chained rows. Unfortunately, neither is based on response time. This means no information about the real impact of the problem is available. The first, which is based on the `v$sysstat` and `v$sesstat` views, merely gives a clue that somewhere in the database there are either migrated or chained rows. The idea is to check the statistic that gives the number of fetches that read more than one row piece (including intra-block chained rows), namely, `table fetch continued row`. To assess the relative impact of row chaining and migration, this statistic could also be compared with `table scan rows gotten` and `table fetch by rowid`.

In contrast, the second technique gives precise information about the migrated and chained rows. Unfortunately, it requires the execution of the `ANALYZE TABLE LIST CHAINED ROWS` statement for each table potentially containing chained or migrated rows. If chained or migrated rows are found, their rowid is inserted into the `chained_rows` table. Then, based on the rowids, as shown in the following query, it's possible to estimate the size of the rows and, from that, by comparing their size with the block size, recognize whether they're migrated or chained.

```
SELECT vsize(<col1>) + vsize(<col2>) + ... + vsize(<coln>)
FROM <table>
WHERE rowid = '<rowid>'
```

Alternatively, as a rough estimation, it's also possible to look at the `avg_row_len` column in a view like `dba_tables`. If the average row length is close or even larger than the block size, it's likely that there are chained rows.

■ **Caution** As discussed in Chapter 8, the `chain_cnt` column of the data dictionary views like `dba_tables` should provide the number of chained and migrated rows. Unfortunately, this statistic isn't gathered by the `dbms_stats` package. If `chain_cnt` isn't set, the package sets it to 0. Otherwise, the package doesn't modify `chain_cnt` at all. The `chain_cnt.sql` script demonstrates this behavior. Although the only way to populate `chain_cnt` with correct values is to execute the `ANALYZE TABLE COMPUTE STATISTICS` statement, this will cause all object statistics for the analyzed table to be overwritten. This isn't recommended practice.

### Solutions

The measures applied to avoid migration are different from those applied for chaining. Because of this, let me stress that you have to determine whether the problem is caused by migration or chaining before taking measures.

Preventing row migration is possible. It's only a matter of correctly setting `PCTFREE` or, in other words, of reserving enough free space to fully store the modified rows in the original blocks. This way, if you have determined that you're are experiencing row migration, you should increase the current values of `PCTFREE`. To choose a good value, you should estimate the average row growth. To do that, you should know their average size at the time when they're inserted and their average size once they're no longer being updated.

To remove migrated rows from a table, there are two possibilities. First, you can completely reorganize the table with export/import or `ALTER TABLE MOVE`. Second, you can copy only the migrated rows into a temporary table and then delete and reinsert them into the original table. The second approach is especially useful when a small percentage of the rows are migrated and there isn't enough time or resources to fully reorganize the table.

Avoiding row chaining is much more difficult. The obvious measure to apply is to use a larger block size. Sometimes, however, even the largest block size isn't large enough. In addition, if chaining is due to the number of columns being greater than 255, only a redesign can help. Therefore, in some situations, the only possible workaround to this problem is to place infrequently accessed columns at the end of the table and thereby avoid accessing all pieces every time.

## Block Contention

Block contention, which occurs when multiple processes vie for access to the same blocks at the same time, can lead to poor application performance. Block contention can sometimes be alleviated by manipulating the physical storage parameters for a table or for an index. This section describes those situations in which an application can experience block contention, and shows how to identify and prevent it.

## Problem Description

The buffer cache is shared among all processes belonging to a database instance. As a result, several processes might concurrently need to read or modify the very same block stored in the buffer cache. To avoid conflicting accesses, each process, before being able to access a block in the buffer cache, has to hold a pin on it (there are exceptions to this rule, but it's not important to discuss them for the purpose of this section). A *pin*, which is a short-term lock, is taken in either shared or exclusive mode. On a given block, several processes might hold a pin in shared mode (for example, if they all need to read the block only), while only a single process can hold it in exclusive mode (needed to modify the block). Whenever a process needs a pin in a mode conflicting with other pins held by other processes, it has to wait. It's confronted with *block contention*.

---

■ **Note** Before being able to pin or unpin a block, a process has to get the cache buffers chains latch protecting the block. Because of that, it might happen that block contention is masked by and/or comes with contention for a latch.

---

## Problem Identification

If you're following the recommendations provided in Part 2, the only effective way to identify block contention problems is to measure how much time is lost because of them. For that purpose, you should check whether the application experiences the wait event related to block contention, namely, `buffer busy waits`. In fact, processes that experience block contention wait for that event. As a result, if this event shows up as a relevant component in the resource usage profile, the application is suffering because of block contention. To troubleshoot such a situation, you need the following information:

- The SQL statement that experiences the waits
- The segment on which the waits occur
- The class of the blocks on which the waits occur

As described in Part 2, the best way to get the required information depends on the kind of problem you're facing. Is the problem reproducible or irreproducible? For irreproducible problems, is the analysis performed in real time or postmortem? In addition, licensing requirements should be considered. For example, do you have the Diagnostic Pack license? It's also essential to recognize that not all techniques described in Part 2 are suitable to accurately troubleshoot a block contention problem. In fact, although you can unequivocally identify block contention problems by using techniques based on dynamic performance views (for example, Snapper or Active Session History) and on SQL Trace, you end up in situations involving some uncertainty when using techniques based on AWR and Statspack reports. There are two main reasons for this:

- A system-wide analysis can pinpoint only problems that are so relevant that they impact the whole system. As a result, you might miss a problem that impacts only specific sessions.
- AWR and Statspack reports are based on a number of dynamic performance views that contain data that can't always be correlated. To illustrate this limitation, let's have a look at the `v$waitstat` view (the following query shows an example of the information it provides). Even though the content of this view provides required information about the class of the blocks on which the waits occurred, there's no way to know for sure which SQL statements waited on those blocks (notice that all columns of the view are shown).

```
SQL> SELECT * FROM v$waitstat;

CLASS                   COUNT       TIME
------------------ ---------- ----------
data block             102011       5162
sort block                  0          0
save undo block             0          0
segment header          76053        719
save undo header            0          0
free list                3265         12
extent map                  0          0
1st level bmb            6318        352
2nd level bmb             185          3
3rd level bmb               0          0
bitmap block                0          0
bitmap index block          0          0
file header block         389       2069
unused                      0          0
system undo header          1          1
system undo block           0          0
undo header              3244         70
undo block                 38          2
```

■ **Note** The essential thing to understand about the `v$waitstat` view is that the `class` column is referring to the block type, not to the type of data or structure on which the wait occurred. For example, if there's contention due to freelists contained in a segment header block, the waits are reported under the `segment header` class and not under the `free list` class. In fact, the `free list` class is used for blocks storing only freelist information (such blocks are created when `FREELIST GROUPS` is set to a value greater than 1). Another example is given by indexes. If there's contention for a block that stores an index, the waits are reported under the `data block` class.

For the reasons just explained, the following two sections provide examples of identification based on SQL Trace and the `v$session` view (based on Snapper; in this case, Active Session History isn't very suitable because the test I'm using only runs for a few seconds). The block contention used in these examples were generated with the `buffer_busy_waits.sql` script.

## Using SQL Trace

I advise using TVD$XTAT to process the trace file generated by the `buffer_busy_waits.sql` script. I recommend this even though you can use either TKPROF or TVD$XTAT. This is because TKPROF doesn't provide all information that you require to troubleshoot a block contention issue. Specifically, it doesn't provide information about the blocks experiencing `buffer busy waits`.

The output file generated by TVD$XTAT for the current example, which is available, with the trace file it's based on, in the `buffer_busy_waits.zip` file, shows that one SQL statement, an `UPDATE` statement, is responsible for almost the whole response time. The following excerpt shows the `UPDATE` statement's execution statistics. For 10,000 executions, an elapsed time of 6.187 seconds and a CPU time of 2.411 seconds were measured.

```
UPDATE /*+ index(t) */ T SET D = SYSDATE WHERE ID = :B1 AND N10 = ID

Call     Count Misses   CPU Elapsed PIO    LIO Consistent Current   Rows
------- ------ ------ ----- ------- --- ------ ---------- ------- ------
Parse        1      1 0.001   0.000   0      0          0       0      0
Execute 10,000      1 2.410   6.186   0 73,084     43,797  29,287 10,000
Fetch        0      0 0.000   0.000   0      0          0       0      0
------- ------ ------ ----- ------- --- ------ ---------- ------- ------
Total   10,001      2 2.411   6.187   0 73,084     43,797  29,287 10,000
```

Since the CPU time is only 39% of the response time, it's necessary to look at the waits that occurred during processing to find out how the time was spent. The following excerpt shows precisely that information. You can see that the greatest consumer is, with 3.322 seconds, `buffer busy waits`. Also notice that, in this specific case, there's minimal contention for the cache buffers chains latch.

```
                              Total           Number of Duration per
Component                  Duration       %    Events          Event
-------------------------- -------- ------- --------- ------------
buffer busy waits             3.322  53.843    10,953        0.000
CPU                           2.410  39.056       n/a          n/a
latch: In memory undo latch   0.278   4.509     6,389        0.000
latch: cache buffers chains   0.158   2.559     6,238        0.000
recursive statements          0.001   0.016       n/a          n/a
enq: HW - contention          0.000   0.008         1        0.000
Disk file operations I/O      0.000   0.007         1        0.000
latch free                    0.000   0.001         1        0.000
-------------------------- -------- -------
Total                         6.170 100.000
```

The additional information provided by TVD$XTAT and missing in the TKPROF output is a list containing the blocks on which the waits occurred. The following excerpt shows what such a list looks like in the case being analyzed here. You can see that more than 99% of the `buffer busy waits` occurred on block 836,775 in file 4. Also notice that the block experiencing contention is a data block.

```
File   Block      Total          Number of         Duration per
Number Number  Duration       %    Events       %         Event Class
------ ------- -------- ------- --------- ------- ------------ ---------------
4      836,775    3.290  99.045     9,779  89.281          336 data block (1)
3      272        0.006   0.173       196   1.789           29 undo header (35)
3      192        0.003   0.094       108   0.986           29 undo header (25)
3      128        0.003   0.094       117   1.068           27 undo header (17)
3      256        0.003   0.090       122   1.114           24 undo header (33)
3      144        0.003   0.088        88   0.803           33 undo header (19)
3      208        0.003   0.083        99   0.904           28 undo header (27)
3      240        0.003   0.083       112   1.023           24 undo header (31)
3      176        0.003   0.082       105   0.959           26 undo header (23)
3      224        0.003   0.081       107   0.977           25 undo header (29)
...
------ ------- -------- ------- --------- ------- ------------
Total             3.322 100.000    10,953 100.000          303
```

Based on this information, you can find out the name of the segment on which the waits occurred with a query like the following (be careful, as executing this query might be resource-intensive):

```
SQL> SELECT owner, segment_name, segment_type
  2  FROM dba_extents
  3  WHERE file_id = 4
  4  AND 836775 BETWEEN block_id AND block_id+blocks-1;

OWNER SEGMENT_NAME SEGMENT_TYPE
----- ------------ ------------
CHRIS T            TABLE
```

In summary, the whole analysis provided the following information:

- The SQL statement that experienced the waits is an UPDATE statement.
- The waits occurred, most of the time, on a single data block.
- The segment on which the waits occurred is the table on which the UPDATE statement was executed.

## Using Snapper

With Snapper, you can aggregate the data according to several criteria. But, provided you already identified the session(s) you have to troubleshoot, you probably start by executing a command like the one shown in the following example. In this case, I specified to target all the sessions executing the `buffer_busy_waits.sql` script. As the output shows, not only is one SQL statement (`9bjs886z43g7k`) consuming most of the database time (during the sampling period, it has seven active sessions in total), but in doing so, it experiences a lot of `buffer busy waits`.

```
SQL> @snapper.sql ash=sql_id+wait_class+event 1 1 user=chris

-------------------------------------------------------------------
Active% | SQL_ID        | WAIT_CLASS  | EVENT
-------------------------------------------------------------------
   560% | 9bjs886z43g7k | Concurrency | buffer busy waits
   100% | 9bjs886z43g7k | ON CPU      | ON CPU
    40% | 091f2847g34rm | ON CPU      | ON CPU
    20% | 48gc5511n38a1 | ON CPU      | ON CPU
    20% | 9bjs886z43g7k | Concurrency | latch: cache buffers chains
    20% | 9bjs886z43g7k | Concurrency | latch: In memory undo latch
    10% | 93053g60rwz0x | ON CPU      | ON CPU
    10% |               | ON CPU      | ON CPU
    10% | cktrdz5u39r04 | ON CPU      | ON CPU
    10% | 93053g60rwz0x | Concurrency | latch: In memory undo latch
```

Then, to get the text and the execution plan of the problematic SQL statement, you can use the `dbms_xplan` package. As expected, it's the same UPDATE statement as the one identified in the previous section.

```
SQL> SELECT * FROM table(dbms_xplan.display_cursor('9bjs886z43g7k', 0, 'basic'));

UPDATE /*+ index(t) */ T SET D = SYSDATE WHERE ID = :B1 AND N10 = ID
```

```
--------------------------------------------
| Id  | Operation                     | Name |
--------------------------------------------
|   0 | UPDATE STATEMENT              |      |
|   1 |  UPDATE                       | T    |
|   2 |   TABLE ACCESS BY INDEX ROWID| T    |
|   3 |    INDEX UNIQUE SCAN          | T_PK |
--------------------------------------------
```

To further investigate the `buffer busy waits`, you can execute Snapper once again, as in the following example, but this time execute it with a set of parameters to show detailed information about the `buffer busy waits`. Note that for `buffer busy waits`, the parameters associated to the event have the following meaning: `p1` is the file number (4), `p2` is the block number (836,775), and `p3` is the class of the block (1 = data block) that experienced the waits.

```
SQL> @snapper.sql ash=event+p1+p2+p3 1 1 user=chris

-----------------------------------------------------------------------------
Active% | EVENT                      | P1         | P2          | P3
-----------------------------------------------------------------------------
   560% | buffer busy waits          | 4          | 836775      | 1
   190% | ON CPU                     |            |             |
    20% | latch: cache buffers chains | 1992028296 | 155         | 0
    20% | latch: In memory undo latch | 1966009168 | 251         | 0
    10% | latch: In memory undo latch | 1966009648 | 251         | 0
```

In summary, the analysis with Snapper pinpoints exactly the same problem as was identified with SQL Trace.

## Solutions

By identifying the SQL statement, the block class, and the segment that experienced the waits, it should be possible to identify the root cause of the problem. Let's discuss some typical cases for common block classes.

### Contention for Data Blocks

All the blocks that make up a table or index segments that are *not* used for storing metadata (for example, segment headers) are called *data blocks*. Contention for them has two main causes. The first is the very high frequency of data block accesses on a given segment. The second is the very high frequency of executions. At first glance, both are the same thing. Why they're, in fact, different requires some explanation. In the first case, the problem is inefficient execution plans causing frequent data block accesses of the same blocks. Usually, it's because of inefficient related-combine operations (for example, nested loops joins). In this case, even two or three SQL statements executed concurrently might be enough to cause contention. Instead, in the second case, the problem is the very high frequency of the execution of several SQL statements accessing the same block at the same time. In other words, it's the number of SQL statements executed concurrently against (few) blocks that's the problem. It might be that both happen at the same time. If this is the case, take care of solving the first problem before facing the second one. In fact, the second problem might disappear when the first is gone.

To solve the first problem, SQL optimization is necessary. An efficient execution plan must be executed in place of the inefficient one. Of course, in some situations, that is easier said than done. Nevertheless, this is really what you have to achieve.

To solve the second problem, several approaches are available. Which one you have to use depends on the type of the SQL statement (that is, DELETE, INSERT, SELECT,[1] and UPDATE) and on the type of the segment (that is, table or index). However, before starting, you should always ask one question when the frequency of execution is high: is it really necessary to execute those SQL statements against the same data so often? Actually, it's not unusual to see applications (that implement some kind of polling, for example) that unnecessarily execute the same SQL statement too often. If the frequency of execution can't be reduced, there are the following possibilities. Note that in most situations, the goal is to spread the activities over a greater number of blocks in order to solve the problem. The only exception is when several sessions wait from the same row.

- If there's contention for a table's blocks because of DELETE, SELECT, and UPDATE statements, you should reduce the number of rows per block. Note that this is the opposite of the common best practice to fit the maximum number of rows per block. To store fewer rows per block, either a higher PCTFREE or a smaller block size can be used.
- If there's contention for a table's blocks because of INSERT statements and freelist segment space management is in use, the number of freelists can be increased. In fact, the goal of having several freelists is precisely to spread concurrent INSERT statements over several blocks. Another possibility is to move the segment into a tablespace with automatic segment space management.
- If there's contention for an index's blocks, there are two possible solutions. First, the index can be created with the option REVERSE. Note, however, that this method doesn't help if the contention is on the root block of the index. Second, the index can be hash partitioned (or subpartitioned), based on the leading column of the index key (this creates multiple root blocks and so helps with root block contention if a single partition is accessed).

The important thing to note about reverse indexes is that range scans on them can't apply restrictions based on range conditions (for example, BETWEEN, >, or <=). Of course, equality predicates are supported. The following example, based on the reserve_index.sql script, shows that the query optimizer no longer uses the index after rebuilding it with the REVERSE option:

```
SQL> SELECT * FROM t WHERE n < 10;

--------------------------------------------
| Id  | Operation                   | Name |
--------------------------------------------
|   0 | SELECT STATEMENT            |      |
|   1 |  TABLE ACCESS BY INDEX ROWID| T    |
|*  2 |   INDEX RANGE SCAN          | T_I  |
--------------------------------------------

   2 - access("N"<10)

SQL> ALTER INDEX t_i REBUILD REVERSE;

SQL> SELECT * FROM t WHERE n < 10;
```

[1]SELECT statements modify blocks in two situations: first, when the FOR UPDATE option is specified, and second, when deferred block cleanout occurs.

```
----------------------------------
| Id  | Operation         | Name |
----------------------------------
|   0 | SELECT STATEMENT  |      |
|*  1 |  TABLE ACCESS FULL| T    |
----------------------------------

   1 - filter("N"<10)
```

Note that hints don't help in such a situation either. The database engine is simply not able to apply a range condition through an index range scan with a reverse index. Therefore, as the following example illustrates, if you try to force the query optimizer to use an index access, an index full scan will be used:

```
SQL> SELECT /*+ index(t) */ * FROM t WHERE n < 10;

------------------------------------------
| Id  | Operation                   | Name |
------------------------------------------
|   0 | SELECT STATEMENT            |      |
|   1 |  TABLE ACCESS BY INDEX ROWID| T    |
|*  2 |   INDEX FULL SCAN           | T_I  |
------------------------------------------

   2 - filter("N"<10)
```

## Contention for Segment Header Blocks

Every table and index segment has a header block. This block contains the following metadata: information about the high watermark of the segment, a list of the extents making up the segment, and information about the free space. To manage the free space, the header block contains (depending on the type of segment space management that is in use) either freelists or a list of blocks containing automatic segment space management information. Typically, contention for a segment header block is experienced when its content is modified by several processes concurrently. Note that the header block is modified in the following situations:

- If INSERT statements make it necessary to increase the high watermark
- If INSERT statements make it necessary to allocate new extents
- If DELETE, INSERT, and UPDATE statements make it necessary to modify a freelist

A possible solution for these situations is to partition the segment in order to spread the load over several segment header blocks. Most of the time, this might be achieved with hash partitioning, although, depending on the load and the partition key, other partitioning methods might work as well. However, if the problem is because of the second or third situation, other solutions exist. For the second, you should simply use bigger extents. In this way, new extents would seldom be allocated. For the third, which doesn't apply to tablespaces using automatic segment space management, freelists can be moved into other blocks by means of freelist groups. In fact, when several freelist groups are used, the freelists are no longer located in the segment header block (they're spread on a number of blocks equal to the value specified with the parameter FREELIST GROUPS, so you will have less contention on them—you're not simply moving the contention to another place!). Another possibility is to use a tablespace with automatic segment space management instead of freelist segment space management.

■ **Note** One of the long-lasting myths about Oracle Database is that freelist groups are useful only when Real Application Clusters is in use. This is *wrong*. Freelist groups are useful in every database. I stress this point because I have read and heard wrong information about this too many times.

### Contention for Undo Header and Undo Blocks

Contention for these types of blocks occurs in two situations. The first, and only for undo header blocks, is when few undo segments are available and lots of transactions are concurrently committed (or initialized or rolled back). This should be a problem only if you're using manual undo management. In other words, it usually happens if the database administrator has manually created the rollback segments. To solve this problem, you should use automatic undo management. The second situation is when several sessions modify and query the same blocks at the same time. As a result, lots of consistent read blocks have to be created, and this requires you to access both the block and its associated undo blocks. There's little that can be done about this situation, other than reducing the concurrency for the data blocks, thereby reducing the ones for the undo blocks at the same time.

### Contention for Extent Map Blocks

As discussed previously in the "Contention for Segment Header Blocks" section, the segment header blocks contain a list of the extents that make up the segment. If the list doesn't fit in the segment header, it's distributed over several blocks: the segment header block and one or more extent map blocks. Contention for an extent map block is experienced when concurrent `INSERT` statements have to constantly allocate new extents. To solve this problem, you should use bigger extents.

### Contention for Freelist Blocks

As discussed in the "Contention for Segment Header Blocks" section, freelists can be moved into other blocks, called *freelist blocks*, by means of freelist groups. Contention for a freelist block is experienced when concurrent `DELETE`, `INSERT`, or `UPDATE` statements have to modify the freelists. To solve this problem, you should increase the number of freelist groups. Another possibility is to use a tablespace with automatic segment space management instead of freelist segment space management.

## Data Compression

The common goal of compressing data is to save disk space. Since we are dealing with performance, this section describes another advantage of data compression that is frequently forgotten: improving response time.

### Concept

The idea of taking advantage of data compression to achieve better performance is based on a simple concept. If a SQL statement has to process a lot of data through a full table (or partition) scan, it's likely that the main contributor to its resource usage profile is related to disk I/O operations. In such a situation, decreasing the amount of data to be read from the disks will increase performance. Actually, the performance should increase almost proportionally to the compression factor. The following example, based on the `data_compression.sql` script, illustrates this:

```
SQL> CREATE TABLE t NOCOMPRESS AS
  2  WITH
  3    t AS (SELECT /*+ materialize */ rownum AS n
  4          FROM dual
  5          CONNECT BY level <= 1000)
  6  SELECT rownum AS n, rpad(' ',500,mod(rownum,15)) AS pad
  7  FROM t, t, t
  8  WHERE rownum <= 1E7;

SQL> execute dbms_stats.gather_table_stats(ownname=>user, tabname=>'t')

SQL> SELECT table_name, blocks FROM user_tables WHERE table_name = 'T';

TABLE_NAME BLOCKS
---------- ------
T          715474

SQL> SELECT count(n) FROM t;

  COUNT(N)
----------
  10000000

Elapsed: 00:00:27.91

SQL> ALTER TABLE t MOVE COMPRESS;

SQL> execute dbms_stats.gather_table_stats(ownname=>user, tabname=>'t')

SQL> SELECT table_name, blocks FROM user_tables WHERE table_name = 'T';

TABLE_NAME BLOCKS
---------- ------
T          140367

SQL> SELECT count(n) FROM t;

  COUNT(N)
----------
  10000000

Elapsed: 00:00:05.38

SQL> SELECT 715474/140367, 27.91/05.38 FROM dual;

715474/140367 27.91/05.38
------------- -----------
   5.09716671  5.18773234
```

To take advantage of data compression for full scan operations, as shown in the previous example (in other words, to make full scan operations faster), it might be necessary to have spare CPU resources. This isn't because of the CPU overhead of "uncompressing" the blocks (which is very small because the default compression is based on a fairly simple algorithm that only deduplicates repeated column values) but simply because the operations performed by the SQL engine (in the previous example, accessing the blocks and doing the count) are executed in a shorter period of time. Also note that reducing the number of physical I/O operations also reduces CPU consumption. For example, on my test system, the average CPU utilization at the session level during the execution of the test queries was about 18% without compression and 27% with compression.

## Requirements

Oracle Database Enterprise Edition (and not any other edition) provides several data compression methods. In addition, some of them are available only in specific releases; others are limited by licensing issues. Table 16-2 summarizes which release provides which method, and the licensing requirements to use them.

***Table 16-2.*** *Compression Methods Provided by Oracle Database*

| Method | Versions | Licensing Requirements |
|---|---|---|
| Basic table compression | From 9.2 onward | None. |
| Advanced row compression (a.k.a. OLTP table compression) | From 11.1 onward | Advanced Compression option. |
| Hybrid columnar compression | From 11.2 onward | The tablespace containing the data must reside in either Exadata storage, ZFS storage, or Pillar Axiom 600 storage. |

Whether a data compression method can be applied also depends on the implementation of the table to be compressed. Specifically, the following limitations apply:

- Only heap tables can be compressed (index-organized tables, external tables, and tables that are part of a cluster aren't supported).
- Except for hybrid columnar compression, the compressed table can't have more than 255 columns.
- The compressed table can't have `LONG` or `LONG RAW` columns.
- The compressed table can't have row-level dependency tracking enabled.
- The compressed table can't be owned by the `sys` user or be stored in the `system` tablespace.

## Methods

To give an overview of the differences between the three methods mentioned in Table 16-2, let's quickly look at their key characteristics and when they should be implemented.

Basic table compression was the first compression method to be introduced by Oracle. To use the method, it's necessary to perform the loads through the direct-path interface. In other words, basic table compression compresses data blocks only when one of the following operations is used:

- `CREATE TABLE ... COMPRESS ... AS SELECT ...`
- `ALTER TABLE ... MOVE COMPRESS`
- `INSERT /*+ append */ INTO ... SELECT ...`
- `INSERT /*+ parallel(...) */ INTO ... SELECT ...`
- Loads performed by applications using the OCI direct-path interface (for example, the SQL*Loader utility)

To make sure that as much data as possible is stored in every block, with basic table compression, the database engine sets `PCTFREE` to `0` by default. In case data is inserted through regular `INSERT` statements, it's stored in uncompressed blocks. Another disadvantage of basic table compression is that not only do `UPDATE` statements commonly lead to migrated rows stored in uncompressed blocks, but also the free space in compressed blocks caused by `DELETE` statements is usually not reused. For these reasons, I suggest using basic table compression only on segments that are (mostly) read-only. For example, in a partitioned table storing a long history in which only the last few partitions are modified, it could be useful to compress the partitions that are (mostly) read-only. Data marts and completely refreshed materialized views are also good candidates for basic table compression.

Advanced row compression was introduced to support tables experiencing regular `INSERT` statements and also modifications (such as from `UPDATE` and `DELETE` statements) by providing a compression ratio similar to basic table compression (the internal storage is basically the same). Since the way this compression method works is dynamic (data compression doesn't take place for every `INSERT` statement or modification; instead, it takes place when a given block contains enough uncompressed data), it's difficult to give advice about its utilization. There are still several situations in which advanced row compression isn't better than basic table compression. In addition, with advanced row compression, modifications can generate much more undo and redo than for a uncompressed table. As a result, to figure out whether advanced row compression is able to correctly handle data that is *not* (mostly) read-only, I strongly advise you to carefully test first with the expected load.

Hybrid columnar compression is based on completely different technology. The key difference is that the columns of a specific row are no longer stored sequentially, as shown in Figure 16-1. Instead, data is stored column by column and, as a result, columns of the same rows can be stored in different blocks. In addition, to cluster together as much data of the same type as possible, the basic storage structure, called a *logical compression unit*, is composed of several blocks. The aim of storing data column by column and of using larger storage structures is to achieve much higher compression ratios. However, when processing compressed data, higher compression ratios are usually related to higher CPU consumption. For that reason, you're able to choose between four compression levels (here sorted according to the expected compression ratio and CPU consumption): `QUERY LOW`, `QUERY HIGH`, `ARCHIVE LOW`, and `ARCHIVE HIGH`. Note that on an Exadata system, while the decompression can be offloaded (a smart scan is required, though), the compression is always performed by a database instance. Other drawbacks of hybrid columnar compression:

- Data is compressed only when loaded through the direct-path interface (same requirement as for basic table compression); regular `INSERT` statements store data in blocks using advanced row compression.
- Row-level locking isn't supported; only whole logical compression units can be locked.
- Even though row movement isn't explicitly enabled at the table level, `UPDATE` statements lead to row movement, and therefore, rowids can change.

In summary, I advise using hybrid columnar compression only in the same circumstances as basic table compression. The only additional requirement is that you need one of the storage subsystems listed in Table 16-2.

It's important to note that, as shown in Table 16-3, every version between 11.1 and 12.1 changed the keywords used to activate a specific data compression method. And, at the same time, each newer release deprecated the keywords used by the prior release. The only keywords that work in exactly the same way in all releases are:

- `NOCOMPRESS` disables table compression.
- `COMPRESS` enables basic table compression.

***Table 16-3.*** *Different Releases Support Different Table Compression Clauses*

| Version | Basic Table Compression | Advanced Row Compression | Hybrid Table Compression |
|---|---|---|---|
| 11.1 | COMPRESS FOR DIRECT_LOAD OPERATIONS | COMPRESS FOR ALL OPERATIONS | n/a |
| 11.2 | COMPRESS BASIC | COMPRESS FOR OLTP | COMPRESS FOR [QUERY\|ARCHIVE] [LOW\|HIGH] |
| 12.1 | ROW STORE COMPRESS BASIC | ROW STORE COMPRESS ADVANCED | COLUMN STORE COMPRESS FOR [QUERY\|ARCHIVE] [LOW\|HIGH] |

PART V

# Appendix

# Bibliography

Adams, Steve, “Oracle Internals and Advanced Performance Tuning.” Miracle Master Class, 2003.

Ahmed, Rafi et al, “Cost-Based Transformation in Oracle.” VLDB Endowment, 2006.

Ahmed, Rafi, “Query processing in Oracle DBMS.” ACM, 2010.

Lee, Allison and Mohamed Zait, “Closing the query processing loop in Oracle 11g.” VLDB Endowment, 2008.

Alomari, Ahmed, *Oracle8i & Unix Performance Tuning*. Prentice Hall PTR, 2001.

Andersen, Lance, *JDBC 4.1 Specification*. Oracle Corporation, 2011.

Antognini, Christian, “Tracing Bind Variables and Waits.” SOUG Newsletter, 2000.

Antognini, Christian, “When should an index be used?” SOUG Newsletter, 2001.

Antognini, Christian, Dominique Duay, Arturo Guadagnin, and Peter Welker, “Oracle Optimization Solutions.” Trivadis TechnoCircle, 2004.

Antognini, Christian, “CBO: A Configuration Roadmap.” Hotsos Symposium, 2005.

Antognini, Christian, “SQL Profiles.” Trivadis CBO Days, 2006.

Antognini, Christian, “Oracle Data Storage Internals.” Trivadis Traning, 2007.

Bellamkonda, Srikanth et al, “Enhanced subquery optimizations in Oracle.” VLDB Endowment, 2009.

Booch, Grady, *Object-Oriented Analysis and Design with Applications*. Addison-Wesley, 1994.

Brady, James, “A Theory of Productivity in the Creative Process.” IEEE Computer Graphics and Applications, 1986.

Breitling, Wolfgang, “A Look Under the Hood of CBO: the 10053 Event.” Hotsos Symposium, 2003.

Breitling, Wolfgang, “Histograms—Myths and Facts.” Trivadis CBO Days, 2006.

Breitling, Wolfgang, “Joins, Skew and Histograms.” Hotsos Symposium, 2007.

Brown, Thomas, “Scaling Applications through Proper Cursor Management.” Hotsos Symposium, 2004.

Burns, Doug, “Statistics on Partitioned Objects.” Hotsos Symposium, 2011.

Caffrey, Melanie et al, *Expert Oracle Practices*. Apress, 2010.

Chakkappen, Sunil et al, “Efficient and Scalable Statistics Gathering for Large Databases in Oracle 11g.” ACM, 2008.

Chaudhuri, Surajit, “An Overview of Query Optimization in Relational Systems.” ACM Symposium on Principles of Database Systems, 1998.

Dageville, Benoît and Mohamed Zait, "SQL Memory Management in Oracle9*i*." VLDB Endowment, 2002.

Dageville, Benoît et al, "Automatic SQL Tuning in Oracle 10*g*." VLDB Endowment, 2004.

Database Language – SQL. ANSI, 1992.

Database Language – SQL – Part 2: Foundation. ISO/IEC, 2003.

Date, Chris, *Database In Depth*. O'Reilly, 2005.

Dell'Era, Alberto, "Join Over Histograms." 2007.

Dell'Era, Alberto, Alberto Dell'Era's Blog (`http://www.adellera.it`).

Dyke, Julian, "Library Cache Internals." 2006.

Engsig, Bjørn, "Efficient use of bind variables, cursor_sharing and related cursor parameters." Miracle White Paper, 2002.

Flatz, Lothar, "How to Avoid a Salted Banana." DOAG Conference, 2013.

Foote, Richard, Richard Foote's Oracle Blog (`http://richardfoote.wordpress.com`).

Foote, Richard, "Indexing New Features: Oracle 11g Release 1 and Release 2", 2010.

Geist, Randolf, "Dynamic Sampling." All Things Oracle, 2012.

Geist, Randolf, "Everything You Wanted To Know About FIRST_ROWS_n But Were Afraid To Ask." UKOUG Conference, 2009.

Geist, Randolf, Oracle Related Stuff Blog (`http://oracle-randolf.blogspot.com`).

Grebe, Thorsten, "Glücksspiel Systemstatistiken – das Märchen von typischen Workload." DOAG Conference, 2012.

Green, Connie and John Beresniewicz, "Understanding Shared Pool Memory Structures." UKOUG Conference, 2006.

Goldratt, Eliyahu, *Theory of Constraints*. North River Press, 1990.

Gongloor, Prabhaker, Sameer Patkar, "Hash Joins, Implementation and Tuning." Oracle Technical Report, 1997.

Gülcü Ceki, *The complete log4j manual*. QOS.ch, 2003.

Hall, Tim, ORACLE-BASE (`http://www.oracle-base.com`).

Held, Andrea et al, *Der Oracle DBA*. Hanser, 2011.

Hoogland, Frits, "About Multiblock Reads." Hotsos Symposium, 2013.

Jain, Raj, *The Art of Computer Systems Performance Analysis*. Wiley, 1991.

Kolk, Anjo, "The Life of an Oracle Cursor and its Impact on the Shared Pool." AUSOUG Conference, 2006.

Knuth, Donald, "Structured Programming with go to Statements." Computing Surveys, 1974.

Knuth, Donald, *The Art of Computer Programming, Volume 3 – Sorting and Searching*. Addison-Wesley, 1998.

Kyte, Thomas, *Effective Oracle by Design*. McGraw-Hill/Osborne, 2003.

Lahdenmäki, Tapio and Michael Leach, *Relational Database Index Design and the Optimizers*. Wiley, 2005.

Lee, Allison and Mohamed Zait, "Closing The Query Processing Loop in Oracle 11g." VLDB Endowment, 2008.

Lewis, Jonathan, "Compression in Oracle." All Things Oracle, 2013.

Lewis, Jonathan, *Cost-Based Oracle Fundamentals*. Apress, 2006.

Lewis, Jonathan, “Hints and how to use them.” Trivadis CBO Days, 2006.

Lewis, Jonathan, Oracle Scratchpad Blog (http://jonathanlewis.wordpress.com).

Lilja, David, *Measuring Computer Performance*. Cambridge University Press, 2000.

Machiavelli Niccoló, *Il Principe*. Einaudi, 1995.

Mahapatra, Tushar and Sanjay Mishra, *Oracle Parallel Processing*. O’Reilly, 2000.

Menon, R.M., *Expert Oracle JDBC Programming*. Apress, 2005.

Mensah, Kuassi, *Oracle Database Programming using Java and Web Services*. Digital Press, 2006.

Merriam-Webster online dictionary (http://www.merriam-webster.com).

Millsap, Cary, “Why You Should Focus on LIOs Instead of PIOs.” 2002.

Millsap, Cary with Jeff Holt, *Optimizing Oracle Performance*. O’Reilly, 2003.

Millsap, Cary, *The Method R Guide to Mastering Oracle Trace Data*. CreateSpace, 2013.

Moerkotte, Guido, *Building Query Compilers*. 2009.

Morton, Karen et al, *Pro Oracle SQL*. Apress, 2010.

Nørgaard, Mogens et al, *Oracle Insights*: *Tales of the Oak Table*. Apress, 2004.

Oracle Corporation, “Bug 10050057 - SQL profile not used in the Active Physical Standby (ADG)).” Oracle Support note 10050057.8, 2013.

Oracle Corporation, “Bug 13262857  Enh: provide some control over DBMS_STATS index clustering factor computation.” Oracle Support note 13262857.8, 2013.

Oracle Corporation, “Bug 14320218 Wrong results with query results cache using PL/SQL function.” Oracle Support note 14320218.8, 2013.

Oracle Corporation, “Bug 8328200 - Misleading or excessive STAT# lines for SQL_TRACE / 10046.” Oracle Support note 8328200.8, 2012.

Oracle Corporation, “CASE STUDY: Analyzing 10053 Trace Files.” Oracle Support note 338137.1, 2012.

Oracle Corporation, “Delete or Update running slow—db file scattered read waits on index range scan.” Oracle Support note 296727.1, 2005.

Oracle Corporation, “Deprecating the cursor_sharing = 'SIMILAR' setting.” Oracle Support note 1169017.1, 2013.

Oracle Corporation, “EVENT: 10046 ‘enable SQL statement tracing (including binds/waits).’” Oracle Support note 21154.1, 2012.

Oracle Corporation, “Extra NESTED LOOPS Step In Explain Plan on 11g and Above.” Oracle Support note 978496.1, 2013.

Oracle Corporation, “Global statistics - An Explanation.” Oracle Support note 236935.1, 2012.

Oracle Corporation, “Handling and resolving unshared cursors/large version_counts.” Oracle Support note 296377.1, 2007.

Oracle Corporation, “How to Edit a Stored Outline to Use the Plan from Another Stored Outline.” Oracle Support note 730062.1, 2012.

Oracle Corporation, “How To Collect Statistics On Partitioned Table in 10g and 11g.” Oracle Support note 1417133.1, 2013.

Oracle Corporation, “How to Monitor SQL Statements with Large Plans Using Real-Time SQL Monitoring?” Oracle Support note 1613163.1, 2014.

Oracle Corporation, “Init.ora Parameter CURSOR_SHARING Reference Note.” Oracle Support note 94036.1, 2014.

Oracle Corporation, “Init.ora Parameter OPTIMIZER_SECURE_VIEW_MERGING Reference Note.” Oracle Support note 567135.1, 2013.

Oracle Corporation, “Init.ora Parameter PARALLEL_DEGREE_POLICY Reference Note.” Oracle Support note 1216277.1, 2013.

Oracle Corporation, “Init.ora Parameter SORT_AREA_RETAINED_SIZE Reference Note.” Oracle Support note 30815.1, 2012.

Oracle Corporation, “Init.ora Parameter STAR_TRANSFORMATION_ENABLED Reference Note.” Oracle Support note 47358.1, 2013.

Oracle Corporation, “Installing and Using Standby Statspack in 11g.” Oracle Support note 454848.1, 2014.

Oracle Corporation, “Interpreting Raw SQL_TRACE output.” Oracle Support note 39817.1, 2012.

Oracle Corporation, Java Platform Standard Edition 7 Documentation.

Oracle Corporation, “Master Note for Materialized View (MVIEW).” Oracle Support note 1353040.1, 2013.

Oracle Corporation, “Master Note for OLTP Compression.” Oracle Support note 1223705.1, 2012.

Oracle Corporation, “Multi Join Key Pre-fetching.” Oracle Support note 264532.1, 2010.

Oracle Corporation, “Real-Time SQL Monitoring.” Oracle White Paper, 2009.

Oracle Corporation, “Rolling Cursor Invalidations with DBMS_STATS.AUTO_INVALIDATE.” Oracle Support note 557661.1, 2012.

Oracle Corporation, “Rule Based Optimizer is to be Desupported in Oracle10g.” Oracle Support note 189702.1, 2012.

Oracle Corporation, “Script to produce HTML report with top consumers out of PL/SQL Profiler DBMS_PROFILER data.” Oracle Support note 243755.1, 2012.

Oracle Corporation, “SQLT (SQLTXPLAIN) - Tool that helps to diagnose a SQL statement performing poorly or one that produces wrong results.” Oracle Support note 215187.1, 2013.

Oracle Corporation, “Table Prefetching causes intermittent Wrong Results in 9iR2, 10gR1, and 10gR2.” Oracle Support note 406966.1, 2007.

Oracle Corporation, “A Technical Overview of the Oracle Exadata Database Machine and Exadata Storage Server.” Oracle White Paper, 2012.

Oracle Corporation, “TRCANLZR (TRCA): SQL_TRACE/Event 10046 Trace File Analyzer - Tool for Interpreting Raw SQL Traces.” Oracle Support note 224270.1, 2012.

Oracle Corporation, “Understanding Bitmap Indexes Growth while Performing DML operations on the Table.” Oracle Support note 260330.1, 2004.

Oracle Corporation, Oracle Database Documentation, 10g Release 2.

Oracle Corporation, Oracle Database Documentation, 11g Release 1.

Oracle Corporation, Oracle Database Documentation, 11g Release 2.

Oracle Corporation, Oracle Database Documentation, 12c Release 1.

Oracle Corporation, *Oracle Database 10g: Performance Tuning*, Oracle University, 2006.

Oracle Corporation, "Query Optimization in Oracle Database 10g Release 2." Oracle White Paper, 2005.

Oracle Corporation, "SQL Plan Management in Oracle Database 11g." Oracle White Paper, 2007.

Oracle Corporation, "Optimizer with Oracle Database 12c." Oracle White Paper, 2013.

Oracle Corporation, "SQL Plan Management with Oracle Database 12c." Oracle White Paper, 2013.

Oracle Corporation, "Use Caution if Changing the OPTIMIZER_FEATURES_ENABLE Parameter After an Upgrade." Oracle Support note 1362332.1, 2013.

Oracle Optimizer Blog (http://blogs.oracle.com/optimizer).

Osborne, Kerry, Kerry Osborne's Oracle Blog (http://kerryosborne.oracle-guy.com/)

Pachot, Franck, "Interpreting AWR Report – Straight to the Goal", 2014.

PHP OCI8, *PHP Manual*, 2013 (http://php.net/manual/en/book.oci8.php).

Põder, Tanel, Tanel Poder's Blog (http://blog.tanelpoder.com).

Senegacnik, Joze, "Advanced Management of Working Areas in Oracle 9i/10g." Collaborate, 2006.

Senegacnik, Joze, "How Not to Create a Table." Miracle Database Forum, 2006.

Shaft, Uri and John Beresniewicz, "ASH Architecture and Usage." Miracle Oracle Open World, 2012.

Shee, Richmond, "If Your Memory Serves You Right." IOUG Live! Conference, 2004.

Shee, Richmond, Kirtikumar Deshpande and K Gopalakrishnan, *Oracle Wait Interface: A Pratical Guide to Performance Diagnostics & Tuning*. McGraw-Hill/Osborne, 2004.

Shirazi, Jack, *Java Performance Tuning*. O'Reilly, 2003.

The Data Warehouse Insider Blog (https://blogs.oracle.com/datawarehousing).

Vargas, Alejandro, "10g Questions and Answers." 2007.

Wikipedia encyclopedia (http://www.wikipedia.org).

Williams, Mark, *Pro .NET Oracle Programming*. Apress, 2005.

Williams, Mark, "Improve ODP.NET Performance." *Oracle Magazine*, 2006.

Winand, Markus, *SQL Performance Explained*. 2012.

Wood, Graham, "Sifting through the ASHes." Oracle Corporation, 2005.

Wustenhoff, Edward, *Service Level Agreement in the Data Center*. Sun BluePrints, 2002.

Zait, Mohamed, "Oracle10g SQL Optimization." Trivadis CBO Days, 2006.

Zait, Mohamed, "The Oracle Optimizer: An Introspection." Trivadis CBO Days, 2012.

# Index

# Troubleshooting Oracle Performance

Christian Antognini

IOUG independent oracle users group

Apress®

# Troubleshooting Oracle Performance

ISBN-13 (pbk): 978-1-4302-5758-5

ISBN-13 (electronic): 978-1-4302-5759-2

President and Publisher: Paul Manning
Lead Editor: Jonathan Gennick
Developmental Editor: Curtis Gautschi
Technical Reviewers: Alberto Dell'Era, Francesco Renne, Joze Senegacnik
Editorial Board: Steve Anglin, Mark Beckner, Ewan Buckingham, Gary Cornell, Louise Corrigan, Jim DeWolf, Jonathan Gennick, Jonathan Hassell, Robert Hutchinson, Michelle Lowman, James Markham, Matthew Moodie, Jeff Olson, Jeffrey Pepper, Douglas Pundick, Ben Renow-Clarke, Dominic Shakeshaft, Gwenan Spearing, Matt Wade, Steve Weiss
Coordinating Editor: Kevin Shea
Copy Editors: Corbin Collins and Sharon Wilkey
Compositor: SPi Global
Indexer: SPi Global
Artist: SPi Global
Cover Designer: Anna Ishchenko

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a Delaware corporation.

For information on translations, please e-mail rights@apress.com, or visit www.apress.com.

Any source code or other supplementary material referenced by the author in this text is available to readers at www.apress.com. For detailed information about how to locate your book's source code, go to www.apress.com/source-code/.

*A dédichi chésto libro a*
*chí, che a rasón, i ga l'éva anmó*
*sü con mí perché a gó anmó metú*
*tròpp témp par scrival…*

*a Michelle, Sofia e Elia.*

# Contents

# Foreword by Jonathan Lewis

When I sat down to write this foreword for the Second Edition of *Troubleshooting Oracle Performance (TOP)*, the first thing I did, after reading the draft chapters, was go back to the foreword I wrote for the first edition to see how much it needed changing. Obviously, there were references to chapter numbers that needed to change, but to my great surprise I discovered that I had failed to make a couple of important points about why this book should be required reading for all aspiring Oracle professionals. Being allowed to revise the foreword allows me to address those points.

The Internet carries a lot of information about Oracle, but it's highly fragmented and desperately in need of collation and cohesion. In fact, many of the books published about Oracle suffer from the same problem—they collect many bits of information together but don't supply any form of cohesive narrative that would allow the reader to grasp a topic in a way that can become a launching point for subsequent learning and understanding. Even the Oracle manuals suffer from the same problem (though to a lesser degree). I've often made the point in my presentations on troubleshooting that your reading should include key manuals from Oracle's documentation set – the *Oracle Database Concepts manual*, the *Oracle Database Administrator's Guide*, and the *Oracle Database Performance Tuning Guide* – and that there will be things in any one of those manuals that you won't really understand until you read the other two. A key feature of *TOP* is the way it structures the information to avoid the historical problem—it tells us what we are trying to achieve, why we want to achieve it, and then how to achieve it.

Sometimes this structure is astonishing in its simplicity. I was particularly struck at one point by the titles of three consecutive chapters. Ignore, for the moment, that the content of the chapters was well worth reading. The titles alone were a startlingly clear presentation of a concept that rarely gets recognized as the first question of troubleshooting:

- Chapter 3: Analysis of Reproducible Problems
- Chapter 4: Real-time Analysis of Irreproducible Problems
- Chapter 5: Postmortem Analysis of Irreproducible Problems

Did you realize that there are only three types of problems, and the strategy you use to solve a problem depends on which of the three classes it belongs to? The fundamental source of the data you can use to solve the problem is the same in all cases, but the availability and granularity of some of that data changes over time. Recognizing this grouping is the first step in a methodical approach to solving problems.

This pattern of collating bits of information and presenting a cohesive picture of what's possible and how to achieve results runs through the whole book. Chapter 6, for example, pulls together a long list of the transformations that Oracle can perform when optimizing a query; Chapter 13 gives a surprisingly long list of all the different ways in which partition-based operations can appear in execution plans.

By the time you've finished reading this book, you will probably find that you already knew a lot more than you realized, but you will also have learned a lot more about what you knew because all the pieces have been pulled together, the gaps have been filled, and the information has been restructured through Christian's knowledge and insights.

—Jonathan Lewis

Jonathan Lewis is well known in the Oracle world. He has worked with the Oracle RDBMS for more than 26 years, written three books about it (two published by Apress), and traveled to more than 50 different countries to supply his skills at troubleshooting, tuning, and design. He is based in the UK, conveniently close to two major airports and one international train station.

# Foreword by Cary Millsap

I think the best thing that has happened to Oracle performance in the past ten years is the radical improvement in the quality of the information you can buy now at the bookstore.

In the old days, the books you bought about Oracle performance all looked pretty much the same. They insinuated that your Oracle system inevitably suffered from too much I/O (which is, in fact, *not* inevitable) or not enough memory (which they claimed was the same thing as too much I/O, which also isn't true). They'd showed you loads and loads of SQL scripts that you might run, and they'd tell you to tune your SQL. And that, they said, would fix everything.

It was an age of darkness.

Chris's book is a member of the family tree that has brought to us . . . light. The difference between the darkness and the light boils down to one simple concept. It's a concept that your mathematics teachers made you execute from the time when you were about ten years old: *show your work*.

I don't mean "show and tell," where someone claims he has improved performance at hundreds of customer sites by hundreds of percentage points [sic], so therefore he's an expert. I mean *show your work*, which means documenting a relevant baseline measurement, conducting a controlled experiment, documenting a second relevant measurement, and then showing your results openly and transparently so that your reader can follow along and even reproduce your test if desired.

That's a big deal. When authors started doing that, Oracle audiences started getting a lot smarter. Since 2000, there has been a dramatic increase in the number of people in the Oracle community who ask intelligent questions and demand intelligent answers about performance. And there's been acceleration in the drowning-out of some really bad ideas that lots of people used to believe.

In this book, Chris follows the pattern that works. He tells you useful things. But he doesn't stop there. He shows you *how he knows*, which is to say he shows you how *you can find out for yourself*. He shows his work.

That brings you two big benefits. First, showing his work helps you understand more deeply what he's showing you, which makes his lessons easier for you to remember and apply. Second, by understanding his examples, you can not only understand the things Chris is showing you, you'll also be able to answer additional good questions that Chris hasn't covered—like what will happen in the next release of Oracle after this book has gone to print.

This book, for me, is both a technical *and* a "persuasional" reference. It contains tremendous amounts of fully documented homework that I can reuse. It also contains eloquent new arguments on several points about which I share Chris's views and his passion. The arguments that Chris uses in this book will help me convince more people to do the Right Things.

Chris is a smart, energetic guy who stands on the shoulders of Dave Ensor, Lex de Haan, Anjo Kolk, Steve Adams, Jonathan Lewis, Tom Kyte, and a handful of other people I regard as heroes for bringing rigor to our field. Now we have Chris's shoulders to stand on as well.

—Cary Millsap

Cary Millsap is chief executive of Method R Corporation, a software performance company. He wrote *Optimizing Oracle Performance* with Jeff Holt in 2003, which earned Cary and Jeff the *Oracle Magazine* 2004 Author of the Year award. Cary is also the author of *The Method R Guide to Mastering Oracle Trace Data*. You can find Cary at `http://method-r.com` and `http://carymillsap.blogspot.com`.

# Foreword from the First Edition

I started using the Oracle RDBMS a little over 20 years ago, and it took about three years for me to discover that troubleshooting and tuning had acquired a reputation verging on the mystical.

One of the developers had passed a query to the DBA group because it wasn't performing well. I checked the execution plan, checked the data patterns, and pointed out that most of the work could be eliminated by adding an index to one of the tables. The developer's response was: "But it doesn't need an index, it's a small table." (This was in the days of 6.0.36, by the way, when the definition of a "short" table was "no more than four blocks long"). So I created the index anyway, and the query ran about 30 times faster—and then I had a lot of explaining to do.

Troubleshooting does not depend on magic, mystique, or myth; it depends on understanding, observation, and interpretation. As Richard Feynmann once said, "It doesn't matter how beautiful your theory is; it doesn't matter how smart you are; if you theory doesn't agree with experiment, it's wrong." There are many "theories" of Oracle performance that are wrong and that should have been deleted from the collective memory many years ago—and Christian Antognini is one of the people helping to wipe them out.

In this book, Christian sets out to describe how things really work, what types of symptoms you should be watching out for, and what those symptoms mean. Above all, he encourages you to be methodical and stick to the relevant details in your observation and analysis. Armed with this advice, you should be able to recognize the real issues when performance problems appear and deal with them in the most appropriate way.

Although this is a book that should probably be read carefully from cover to cover, I think different readers will benefit from it in different ways. Some may pick out the occasional special insight while browsing, as I did in Chapter 4 with the explanation of height-balanced histograms—after years of trying to find an intuitively clear reason for the name, Christian's description suddenly made it blatantly obvious.

Some readers may find short descriptions of features that help them understand why Oracle has implemented that feature and allow them to extrapolate from the examples to situations which are relevant in their applications. The description of "secure view merging" in Chapter 5 was one such description for me.

Other readers may find that they reread a section of the book time and again because it covers so many details of some particularly important and relevant feature they are using. I'm sure that the extensive discussion of partitioning in Chapter 9 is something that many people will return to again and again.

There's a lot in this book—and it's all worth reading. Thank you, Christian.

—Jonathan Lewis

# About the Author

Since 1995, **Christian Antognini** has focused on understanding how Oracle Database works. His main interests include logical and physical database design, the query optimizer, and basically everything else related to application performance management. He is currently working as a senior principal consultant and trainer at Trivadis (www.trivadis.com) in Zürich, Switzerland.

If Christian is not helping one of his customers get the most out of Oracle, he is somewhere lecturing on application performance management or new Oracle Database features for developers. In addition to classes and seminars organized by Trivadis, he regularly presents at conferences and user-group meetings. He is a proud member of the OakTable Network and an Oracle ACE Director.

Christian lives in Ticino, Switzerland, with his wife, Michelle, and their two children, Sofia and Elia. He spends a great deal of his spare time with his wonderful family and, whenever possible, reading books, watching tennis, enjoying a good movie, riding one of his bicycles, or gliding down the Swiss alps on a snowboard.

# About the Technical Reviewers

**Alberto Dell'Era** has been a full-time Oracle Database consultant since 1999, shortly after graduating summa cum laude in electronics engineering.

He currently works for NTT DATA in Italy (a global innovation partner with headquarters in Tokyo and business operations in over 40 countries), where, as an architect, SME, performance tuner, mentor, and sometimes teacher, he helps colleagues and customers to get the most out of Oracle. He especially loves getting his hands dirty working on the development of some critical software module.

He is a member of the OakTable Network (www.oaktable.net), the well-known organization of Oracle professionals that approaches Oracle the same way the scientific community approaches the world: by careful study and practical experimentation, and by sharing results in reproducible means (scripts, prototypes, and so forth) so they can be peer-reviewed easily.

He strongly believes that knowing how a product really works internally is both fun and the key to being really effective in professional life. For this reason, he spends a lot of time and effort reading technical papers, books and blog posts, and making his own investigations, prototypes, and more.

He owns www.adellera.it and can be contacted at alberto.dellera@gmail.com.

**Francesco Renne** was born in 1962 in Como, Italy. He studied computer science at the University of Milan and after graduating, joined Olivetti, where he worked on the development of the Unix operating system. Francesco has been interested in performance since the beginning of his professional career. He has worked on Unix internals and the Oracle environment to achieve the best possible performance in different environments (new products, benchmarks, international real applications on production, and so on).

In 1994, he joined the Banca Popolare di Bergamo, the only bank in Italy that has rewritten its entire information system using Unix and Oracle. He has made major contributions to improving performance over the whole platform.

In 1999, he co-founded ICTeam and is now that company's CEO. He continues to work on performance, especially on Oracle Data Warehouse environments, for some of the largest companies in Italy.

Francesco lives near Bergamo, Italy, with his wife, Adria, and their two daughters, Viola and Veronica. When not striving to improve something, he enjoys staying with his family, listening to progressive music, and taking pictures.

**Jože Senegacnik** has more than 25 years' experience in working with Oracle products. In 1988, he started working with Oracle version 4. Since 1992, he has been self-employed as a private researcher in the field of computer science. Most of his work time is dedicated to solving performance bottlenecks in different application solutions that are based on the Oracle database. He is also an international speaker, giving talks on the most important Oracle database–related events worldwide. He conducts well-known performance tuning courses together with Oracle University.

# Acknowledgments

Let's face it, writing a book isn't a rewarding task. At all! It takes a huge amount of your free time, that you could spend doing something way more interesting. So, while deciding whether I should embark on the project of writing the second edition of *Troubleshooting Oracle Performance*, I asked myself several times, why should I do it? In the end, the most important factor was the hundreds positive messages I received since I published the first edition in 2008. I discovered that publishing a book that other people regard as useful *is* rewarding! For that reason, the first big thank you goes to all the readers of the first edition who gave me feedback. Without the motivation you gave me, this second edition wouldn't exist.

Motivation isn't the only thing you need to write a book. As already mentioned, lots of time is involved. In that respect, I'm privileged to work for a company that, since 1999, when I started there, gave me all the support I needed not only to develop my skills, but also to pursue ideas (like writing a book) that weren't always self-evident. So, the second belated thank you goes to Trivadis.

When you concentrate on writing a piece of text over an extended period of time, sometimes you miss or forget obvious things. Because of that, I'd say it's of paramount importance to be surrounded by people who check what you're doing. Great thanks go to the technical reviewers Alberto, Franco, and Jože. You helped me improve the quality of the book considerably. Any remaining errors are, of course, my own responsibility. In addition to the "official" technical reviewers, I would also like to thank Dani Schnider, Franck Pachot, Randolf Geist, and Tony Hasler for reading selected parts of the book and providing me with their comments on and impressions of the text.

I also have to thank the people at Apress for supporting me during the development of the book. In particular, Jonathan Gennick, who believed it made sense to publish a second edition of *Troubleshooting Oracle Performance*.

As with the first edition, another person who played a central role was Curtis Gautschi. In fact, he assisted me by proofreading the book once again. And that, even though he doesn't understand most of the things he reads (according to him). Thank you so much, Curtis, for assisting me all these years now.

Finally, special thanks go to Jonathan and Cary for endorsing this book by writing a foreword. You inspired me so much at the beginning of my career, and now I hope a book like this one can inspire more people in the Oracle community to do the right thing.

# Acknowledgments from the First Edition

Many people assisted me in writing the book you now have in your hands. I'm extremely grateful to all of them. Without their assistance, this piece of work wouldn't have seen the light of the day. While sharing with you the brief history of *TOP* (*Troubleshooting Oracle Performance*), let me thank the people who made it all possible.

Though I didn't realize it at the time, this story began on July 16, 2004, the day of the kickoff meeting I had organized for a new seminar, Oracle Optimization Solutions, which I had planned to write with some colleagues of mine at Trivadis. During the meeting, we discussed the objectives and structure of the seminar. Many of the ideas developed that day and while writing the seminar in the months afterward have been reused in this book. Big thanks to Arturo Guadagnin, Dominique Duay, and Peter Welker for their collaboration back then. Together, we wrote what, I'm convinced to this day, was an excellent seminar. In addition to them, I also have to thank Guido Schmutz. He participated in the kickoff meeting only, but strongly influenced the way we approached the subjects covered in the seminar.

Two years later, in the spring of 2006, I started thinking seriously about writing this book. I decided to contact Jonathan Gennick at Apress to ask for his opinion about what I had in mind. From the beginning, he was interested in my proposal, and as a result, a few months later I decided to write the book for Apress. Thank you, Jonathan, for supporting me from the very beginning. In addition, thanks to all the people at Apress who worked on the book. I only had the pleasure of working with Sofia Marchant, Kim Wimpsett, and Laura Esterman, but I know that several others contributed to it as well.

Having an idea and a publisher are not enough to write a book. You also need time, a lot of time. Fortunately, the company I work for, Trivadis, was able to support me and the project in this way. Special thanks to Urban Lankes and Valentin De Martin.

In order to write a book, it's also essential to be surrounded by people who carefully check what you're writing. Great thanks go to the technical reviewers, Alberto Dell'Era, Francesco Renne, Jože Senegacnik, and Urs Meier. They helped me considerably in improving the quality of the book. Any remaining errors are, of course, my own responsibility. In addition to the technical reviewers, I would also like to thank Daniel Rey, Peter Welker, Philipp von dem Bussche-Hünnefeld, and Rainer Hartwig for reading parts of the book and providing me with their comments on and impressions of the text.

Another person who played a central role is Curtis Gautschi. For many years, he has proofread and enhanced my poor English. Thank you so much, Curtis, for assisting me for so many years now. I know—I should really try to improve my English skills someday. Unfortunately, I find it much more interesting (and easier) to improve the performance of Oracle-based applications than my command of foreign languages.

Special thanks also go to Cary Millsap and Jonathan Lewis for writing the forewords. I know that you spent a considerable amount of your valuable time writing them. I'm very much indebted to you both for that.

Another special thank goes to Grady Booch for giving me the permission to reproduce the cartoon in Chapter 1.

Finally, I would like to thank all the companies for which I have had the privilege to consult over the years, all those who have attended my classes and seminars and asked so many good questions, and all the Trivadis consultants for sharing their knowledge. I have learned so much from all of you.

⟨IOUG⟩ independent oracle users group

*For the Complete Technology & Database Professional*

**IOUG** represents the **voice of Oracle technology and database professionals** - empowering you to be **more productive in your business** and career by **delivering education,** sharing **best practices** and providing technology direction and **networking opportunities.**

---

## Context, Not Just Content

IOUG is dedicated to helping our members become an #IOUGenius by staying on the cutting-edge of Oracle technologies and industry issues through practical content, user-focused education, and invaluable networking and leadership opportunities:

- ***SELECT Journal*** is our quarterly publication that provides in-depth, peer-reviewed articles on industry news and best practices in Oracle technology
- Our #IOUGenius blog highlights a featured weekly topic and provides **content driven by Oracle professionals and the IOUG community**
- Special Interest Groups provide you the chance to collaborate with peers on the specific issues that matter to you and even take on leadership roles outside of your organization
- COLLABORATE is our once-a-year opportunity to connect with the members of not one, but three, Oracle users groups (IOUG, OAUG and Quest) as well as with the top names and faces in the Oracle community.

**Who we are...**

... **more than 20,000** database professionals, developers, application and infrastructure architects, business intelligence specialists and IT managers

... a **community of users** that share experiences and knowledge on issues and technologies that matter to you and your organization

**Interested? Join IOUG's community of Oracle technology and database professionals at www.ioug.org/Join.**

Independent Oracle Users Group | phone: (312) 245-1579 | email: membership@ioug.org
330 N. Wabash Ave., Suite 2000, Chicago, IL 60611