TWO STYLES OF DATABASE DEVELOPMENT

Describe the opposition between an atomic, procedural and row-oriented style versus a holistic, SQL and data-set oriented style, within a relational database

Stefan Ardeleanu

TWO STYLES OF DATABASE DEVELOPMENT

TWO STYLES OF DATABASE DEVELOPMENT

Describe the opposition between an atomic, procedural and row-oriented style versus a holistic, SQL and data-set oriented style, within a relational database

STEFAN ARDELEANU

ISBN-10: 1493796976

ISBN-13: 9781493796977

Library of Congress Control Number: 2015914541

CreateSpace Independent Publishing Platform,

North Charleston, South Carolina

The application developer … sees himself as a rider on the row.

But the row is not a horse, but a donkey!

TABLE OF CONTENTS

Introduction and Intended Audience

A story about table aliases, a vision about an inferior skill, how to drink wine mixed with water in France!

Writing correctly is critical for the quality of our software

A basic terminology

All types of software developers!

The same styles of development were used for many years in the user interface and in the database!

The application developer – the main target of this book

Database developers and students in IT universities are especially targeted too

The two sections of the book

Chapter 1 The concept of style

The style of development is dynamic. We need to recognize it first!

The most common styles of programming

The database development starts with the table design

Do we already start the development?

Are we ready for SQL?

Chapter 2 SQL - the beauty and the beast!

How can a query language be so important for a style of development?

What is SQL? What is not SQL?

What about programming? Is there such a thing like database programming?

Programming is a practical activity!

Should we write in the database in a certain way?

The SQL shop metaphor!

An example of bad practice!

Chapter 3 The holistic vision against the data

The concept of data set

The holistic approach versus the atomic approach – an introduction

A different model –does this model deserve to be promoted indeed?

Performance and portability - two advantages for the holistic approach.

Visual development versus SQL development

Chapter 4 What to choose: the data set or the data row?

Choosing the level of detail: the set versus the row!

Performance is poor. Performance completely blocked in any tentative of improvement.

Database programming means query, query and query all the time!

Let’s go back to the SQL shop! One more time, please!

The use of scalar functions –a typical accessory for the atomic approach!

Debugging is so simple! The code is much simpler and readable!

What is a database developer nowadays?

It’s practice time!

Chapter 5 Data transfer paradigm, the first set of examples

The exercises, the context, the goals, ways to illustrate the two approaches!

Holistic versus atomic: incrementally update a target

Chapter 6 Others atomic features used in excess

The use of scalar functions – a challenge to the set-based approach

A simple query!

Chapter 7 Writing SQL versus writing procedurally, other holistic methods

Writing SQL versus writing procedurally, another impediment! An example of update!

Writing SQL versus writing procedurally: the power of union in the holistic approach!

Embedded SQL versus dynamic SQL – another dilemma!

Others holistic solutions: the temporary table, explicit or implicit like with a clause

Chapter 8 Row triggers. When should we follow the atomic way? Some final reflections and thoughts!

The use of row triggers: another common atomic solution used in excess

The atomic approach should be used, whenever is necessary!

Some final reflections and thoughts

The concept of SQL template

Writing horizontally or vertically: a decision to be taken!

The specific software application should be implemented holistically

The SQL itself can be better and better!

Performance, oh performance! This book is for you!

A specific data migration software application can be written in pure SQL more often than you expect!

INTRODUCTION AND INTENDED AUDIENCE

A STORY ABOUT TABLE ALIASES, A VISION ABOUT AN INFERIOR SKILL, HOW TO DRINK WINE MIXED WITH WATER IN FRANCE!

Being an IT contractor means working mostly project based. The project can be a short-term based or long term. For a short-term project, you need to work on a specific task in a team of developers, you need to solve a punctual problem and, after doing it, you are done with the project! The team of developers will continue their work and you will search for a new project and challenge.

I want to share some thoughts from one of my experiences of that type.

Some time ago, I was in a project where most of the members were Java developers, very good professionals with many years of experience. They developed a document management system, metadata based, written almost exclusively in Java. The database was Oracle. The manager and their technical leader invited me to do some Oracle staff, to solve some database-specific tasks for the project. All the team members were pure application developers and no one had too much expertise with databases. Still, in the end, everything was transform in SQL but no one cared too much about that. The developers were good artists and the design was very sophisticated, written after the good principles of Object oriented programing.

One day, I was in the office and listened to some of their discussions about a certain SQL statement. They asked me some questions, which I did not understand. I went over to their desks and looked it up. I was still confused because the SQL statement was quite basic. A self-join was required. I added the self-join and everything worked perfectly well. The team members were amazed! They were not aware about the fact that a table can be referenced many times in a join using various aliases. They did not know what a self-join is!

After a while, during a break, we were having some discussions with the same team of Java developers; they were very confused and surprised by the fact that, the same way they spent their entire career doing Java, I spent my entire career doing database development and especially doing SQL. SQL is so simple, your task is such a trivial task, you have a much easier life than ours, they said! We fight with such complicated concepts to apply and you just manipulate rows and columns in your simple query language! That is not fair, they thought without saying it aloud.

In another train of thoughts, I was in France with two of my good colleagues, Clark and Marjorie. Clark is American, my manager for a long time and a very good friend, and Marjorie is a nice and elegant French woman, my colleague in support in the French area. We were in a café and we were chatting. Among other staff, we ordered for some wine. In addition, I asked for some mineral water. In my country, some people have the habit sometimes to mix sparkling water with wine. I tried to ask Marjorie what she thinks about that. Of course, I knew the answer. To any French people, mixing water with wine is a blasphemy. Being in France, being the quest, I understood and left the water where it was, apart from the glass of wine!

What is the connection between these three, shared memories? Being an application developer does not make you SQL independent, in most of the cases. Whether you like it or not, you will be in contact with SQL, sometimes more or sometimes less! The same way any database developer knows the basics of structured programming, normally most of the application developers know the basics of SQL. There are no such things like superior or inferior languages, but useful and compatible according to the business. When you are in the country of relational databases, even if you are an application developer, you should learn how to write the fair SQL.

WRITING CORRECTLY IS CRITICAL FOR THE QUALITY OF OUR SOFTWARE

The quality of a software application is proportional with many things and it is difficult to measure and qualify all the factors that contribute to a good quality. I believe that one of the most important aspects is the way we write our code, our style of development. It is hard to say what it means to write correctly in the context of software development. It is a degree of subjectivity involved in any judgment of this type. However, I believe that no one can argue with me when I say that the style of development and the way we write our code is proportional with the quality of the software we build.

I am a database developer with many years of experience. During this period spent inside relational databases, I gathered enough experience to be able to explain that the database area requires a certain and distinct style of development.

This is the topic of this book: how to write code inside a relational database in a certain manner, distinct and specific, which may not be exactly a common style.

There are thousands and millions of lines of codes in the databases all over the world that are written in a total inadequate manner and these lines of codes cause many performance issues in many places. All these performance issues can be avoided with one condition: the programmers should try to understand that a certain style of development is required in a relational database.

How to write correctly inside a relational database is the main topic of my book. This is a book about a certain style of development within a relational database. It is a book about database programming and a book about what is the fair way of writing inside a database. When I say database programming, I mean mostly SQL, considering that this is what database programming really means. This is a book about performance. Is a book about a minimal performance that can be achieved in any database if the style of development is the appropriate one, according to some standard of development that I want to share with you later!

The style of the book is not academic although it is a book about database programming. I am a practical person and I consider that programming is part of our lives. I also believe that programmers and IT professionals are among us and are ordinary people that deserve also a common style of argumentation and not only an academic style. There are thousands of libraries with academic papers for IT professionals and students. This book does not fall into that category.

A BASIC TERMINOLOGY

Due to the direct style of the book, on one hand, and to the fact that this book is about database development on another, which means that it is still a technical book, I would like to define some basic terminology. The Internet is full of classifications and manuals, courses and documentation, libraries and practical examples, the concepts are explained and re-explained by specialists. I want to be consistent and to avoid any possible confusion so I clarify some keywords used in this paper.

I understand by data-oriented software application that kind of software application composed by at least one user interface, graphical or not, and the database behind. When I say database I am referring mostly to a relational database. One of the main goals for any software application of this type is to allow data access in the database via the user interface. The end users read and write from the database, via the user interface. This is what I call a classic data-oriented software system. I will use the name classic software application, for simplicity. The focus is on the database, so the topic of the software application is always the database section.

Another type of data-oriented software application is that kind of application that has the purpose to transferring data between classic systems. Medium or large companies will possess many classic software applications. Every software system of that type will have its own purpose, its own database and will cover one part of the business or another. Another type of data-oriented software application is the software system where the goal is to move data between two or many classic systems. I call this a specific software application. In most of the cases, there is no user interface and no classic end users. One or many classic systems are the targets and the one or many other classic systems are the sources. That kind of specific system can be anything like a Replication system, a Data Migration system or an Extract Transform and Load system (ETL) part of a data warehouse system.

Please be aware of the distinction classic-specific. One is to develop in a database in a classic system and another one is to develop in a specific system and maybe my main interest in this book is related to the variety of specific systems where a lot of developers are working in the same manner they are working in classic systems.

Regarding the terminology, this book is full of examples and exercises mostly from two database systems. To eliminate any confusions and ambiguities, dear reader, please read Oracle database as Oracle and Microsoft SQL Server as SQL Server.

ALL TYPES OF SOFTWARE DEVELOPERS!

I am a SQL developer. In other words, I am a classic database developer. For many years in the past, I have felt bad about that, thinking that the only pure and authentic database specialists are the database administrators. With all the respect for the work of a DBA, and for their important responsibilities, there were always other database people on the market apart from them! For a long time in the past, I wrongly thought that, being a database developer, which means a SQL developer, is not a path by itself but a skill among others, an insufficient path that could be associated with something else. People were searching for developers in different combinations like Java plus PL SQL, or like C# and SQL Server. The SQL programming and the classic database development was considered mainly as an addition to application development and still is this way in many cases.

Being a software developer might mean knowing how to develop both: database and non-database layers. Sometimes we may be in the position of doing both and sometimes we do an exclusive work.

“He is a very good Java developer, you will see!” ”He will participate in your project and he will finally build value for the customers with his skills!” “He has many years of experience with C and almost as many with C#. He is a very good programmer!” On the other hand, even more, Joe is an object-oriented language theorist and his code is a model for your Java or C# programmers. These kinds of statements were and still are very common and application developers were and continue to be the most common types of developers on the market. For example, working in a web-based application, developing a web interface is an extremely appreciated occupation nowadays, and the market is full of good web developers being able to satisfy the requirements and build extensive and scalable applications.

What is happening with the database? For many years in the past, after the 90’s, the only authentic database people were the database administrators. To be a DBA is a difficult task and involves enormous responsibilities! If something goes wrong in the factory and the inventories fail, the DBA must be there and must try to find a solution for the work to continue. He is the one who takes care of the database and its optimal use by end users: he is the one who is leading all the things in production. It is far from my intentions to minimize the role of a DBA: his presence is critical and necessary for any production system. In large enterprises, there are armies of database administrators who take care of the databases.

What about the ones that effectively built the databases as part of the data-oriented software application? For many years, the market neglected them, even if some people will perhaps disagree with this idea. I admit there is a degree of subjectivism in this statement that I am not afraid to recognize, you can consider it as a personal point of view. The database developers should have received more recognition in the past and even today. Just by looking at the jobs descriptions in the past, before BI and Data warehouse explosions, looking at the requirements for people within the IT market, you will rarely see explicit requirements for database developers. In the recent years, there was a big change and now you can see a larger variety of requests for database specialists, but this change came due to the explosion of reporting systems and analytics. For example, an ETL specialist is a kind of highly specialized database developer, and not an ordinary SQL or database developer!

THE SAME STYLES OF DEVELOPMENT WERE USED FOR MANY YEARS IN THE USER INTERFACE AND IN THE DATABASE!

This situation is visible very clear in the projects and in the heart of the software industry, in the programmers' offices. The software developers, the programmers think in patterns of structured and object-oriented programming and apply these principles everywhere in their code, including the databases. Because, very often, people do not recognize the database developer as a distinct type of specialist, and because they do not accept a distinct pattern of development for him, the databases is often build in an inappropriate style. This is still happening very often, more often than you can imagine!

Unfortunately, sometimes there is still confusion between two different images of the language. I am referring to the confusion between the apparent simplicity of the language and the potential huge complexity of the written logic with this language. SQL is a paradox by itself; you may learn the basics and understand the language in one week. Still, in order to cover the entire complexity of SQL takes almost the same amount of time necessary for becoming a very good Java programmer, for example. To become a SQL expert is as difficult as becoming a Java expert. Because of this confusion, the knowledge of SQL was included in the section others for many years. Software companies want to hire Java or C# programmers with SQL knowledge, and they often consider database programming a kind of secondary skill. Very often application developers should be able to develop everywhere: in the user interface and in the database, and they worked in the same style and using the same patterns. These patterns are extremely suitable for the user interface but not for the database. A large number of software applications are using the same unique style, the classic style of application development in both user interface and database section.

The idea of having a distinct style of programming within the database, distinct in comparison with the existing ones, the idea of using a different style of development in the database exists for some years. I can say that now things are easily changed and people start to see that a database programmer needs to write code differently. I was very happy to see requirements for set based style projects, I was happy to see explicit requirements for a database programmer and an explicit request for a non-application developer.

In this book, I want to promote a certain style of development specific to the database. I am referring to SQL development in particular as the most important type of database development. The SQL programmer is critical in a large variety of projects and it should not be necessarily specialized like me. A good programmer can be both an application developer and a SQL programmer, with priority on one or another. I don't want to say that we should be one or the other and I accept that in most of the projects with classic software applications the developers can do both. However, if the programmer writes SQL he should write his code differently than Java or C or anything else. My plan is to describe this style of programing in opposition with the typical style of an application developer. I want to show that, in the database, a different style of development is required.

THE APPLICATION DEVELOPER – THE MAIN TARGET OF THIS BOOK

I dare to say that, in one way or another, this book is for everyone to a certain degree, and by everyone, of course I am referring to everyone interested in databases, especially in relational databases. There are so many people involved in the database area like programmers of all types, business analysts, software testers, all kinds of IT consultants, IT project managers, IT technical leaders and the list may continue.

Still, this book is especially for programmers. When I say programmers, I can understand many things. The major category of programmers targeted in this book is composed by the variety of application developers like Java developers or C, C++ or C# or PHP developers. I can surely state that this is the most important category of software developers and most of the software today is built on the top of these technologies and on the shoulders of these developers. These developers are focused in their classic languages and they are especially working at the user interface level. Many of them are good practitioners of the Object-oriented model, for example, and they know how to apply this model to their applications. The Object-oriented model is a complex model and it covers many of our realities better than others models. I offer a great sense of admiration and respect for these developers and I believe what they are doing is great. Still, despite the valuable skills that I recognize with all my modesty and sympathy, I consider that this book might be useful for these kinds of developers in the first instance. The book is not for all of them, but for many of them!

Why is that? Because of a simple question: what about the database? Someone might say.

The application developer can decide: I do not want to be involved. Alternatively, he can choose: I want a minimal involvement, a minimum of work at the database level. I am an application developer and I believe working in the database is something else than what I am regularly doing! Maybe some of them they are really doing this.

Still, even if the application developer questions himself about the possibility to write inside a database, this is a very rare situation. Generally the managers decide and they do not analyze too carefully all these problems. They consider that any application developer can work in the database and do database development and they ignore how they are doing it.

The managers will decide: you can do some SQL and you can do some work at the database level, is not a big deal! Consequently, the developer will start working in the database right away without asking him: do I need to do anything differently? I am moving from C# into Transact SQL, I am changing all the time and write code from Java to Oracle, I am affecting some classes and afterwards I will affect some tables inside some store procedures. Do I need to do anything different? This question should be addressed to the developers, but actually, in most of the cases no one really does that. Very often, moving from the user interface to the database is not seen as a major change and the application developers are not thinking that they should change anything in the way they write their code. Unfortunately, they are simply programming right away in the same way they are doing it at the application level.

Some application developers simply ignore any difference, they are not aware of the distinctions because they consider the same model applies to the database. Others are judging SQL language, see its apparent simplicity, and consider that nothing should be said to them. Many developers try to impose their standard from the user interface level and move it to the database. This book is written especially for application developers that, due to various reasons, subjective or objective, are using the same style of development in the database as the one in the user interface. I consider they should change something in the way they write in the database. This book is mainly for them.

DATABASE DEVELOPERS AND STUDENTS IN IT UNIVERSITIES ARE ESPECIALLY TARGETED TOO

This book addresses to database developers too. Normally, a true database developer will be aware of the style described in this book and will write code in the manner I promote. He still can get some clarifications and clear his mind. Still, other database developers are actually closer to application development even if they believe they are database developers. I am referring to those developers linked to the procedural language of a vendor or another like PL SQL developers, for example. These will work more like an application developer than like a database developer. A true database developer is a SQL developer, in my opinion. The book is dedicated to this mixed database application developers.

The book is also dedicated to students in IT universities and to young programmers who are just starting their work in the field of software development. I believe that they should be aware that a different style and a different vision are required when coding in the database versus coding at the user interface level. In the IT universities, these distinctions are not generally made with enough clarity and, very often, when doing effective development on both sides, they are using the same style and the same vision. The consequence is that the database is always sacrificed because the vision is good for the user interfaces but not optimal for the database.

On the other hand, considering the topic, I can affirm that this book is not exclusively dedicated to programmers. This book can be read by business analysts, by IT consultants apart from programmers, any type of professionals that are commonly dealing with data. This book addresses the managers that lead IT data-oriented projects. Almost everyone is dealing with data nowadays in one way or another.

An IT company, where the goal is to create and maintain software, to build all kind of software applications for one business or another, means infinitely more than programmers. Any software application is related to a certain business that wants to model. So the software applications are mainly surrounded by the business people rather than by programmers. People that know the business form the majority in a software company and not the programmers. It is like a big ship or boat where two categories of people stay and live: the business people and the technical people like programmers, DBA’s, sys admins and others.

The business people are targeted by this book too, even if maybe not in the same measure. The business analysts, the business consultants, even the business testers, they are all not so technical, but some of them are strongly involved in the databases at various levels. This book is for them too, not just for programmers.

To conclude, this book addresses the following categories of persons, in the order of importance and utility for each category:

1. The book is especially for the application developers that write code into relational databases using a classic style of development.
2. The book is for those pseudo database developers that are actually more like application developers and are intensively using the procedural facilities of a

language like PL SQL.

3. These pages are also for true database developers, who are actually SQL developers. They will eventually clear their minds and get a confirmation of what they are doing.
4. The book is for students from IT universities, especially computing, computer science and others. I hope these will be aware from the beginning that the database is something else and will be warned in time before starting effective development.
5. The book is for the young programmers that start their work now in the software industry. I hope they will see that they need to follow a different approach in the database and not developer in the same manner like in the user interface.
6. The book is also for business people like analysts or testers that are dealing with data. I believe is good for them to be familiarized with the distinctions between the two styles of development.
7. Finally, the book is for managers and technical leaders of the software projects where the database is a critical component. They are taking the decisions and I hope that some of them will try to give more importance to the topic, for a better performance of their software.

THE TWO SECTIONS OF THE BOOK

The book is divided into two large sections. The first part of the book is the conceptual area. I wanted to explain the reason for the two styles of development and to justify why we need a different style when we are inside a relational database. I will try to define the concept of the style of development, to explain why this is such a critical component for a developer.

Considering that the book is for students in IT universities too, I have some sections in the first four chapters where I describe some basic aspects of database development like table design and the characteristics of SQL language. These sections are also required for the sake of argumentation but can be ignored by experienced database developers, readers of my book.

The goal in the first four chapters is to show that a separate style of development is required in a relational database. This style of development is revealed during the book by the opposition with the classic or typical style of development used by most of the application developers. The style of development that should be used in the database is holistic and set based by opposition with the atomic style and row-oriented used by many application developers.

The last four chapters of the book are highly practical and are meant to proof of the concepts revealed in the first part of the book. I imagined a series of examples, taken from two of the most popular database systems: Oracle and SQL Server. These examples illustrate the two styles of development and show the differences. The practices are described and I am not afraid to state that my goal is to promote the holistic and set based style of development against the atomic and row- oriented style of development so dear to most of the application developers.

Chapter 1

THE CONCEPT OF STYLE

THE STYLE OF DEVELOPMENT IS DYNAMIC. WE NEED TO RECOGNIZE IT FIRST!

I know that I am taking a risk by approaching a somehow vague concept, like the concept of **style of development**. The concept of style is not a scientific notion and it involves a certain degree of subjectivity. I can consider the concept of style of development similar to the concept of **development approach**, but I prefer the terminology of style because I accept the degree of subjectivity mentioned above.

First, this book is not a scientific one. I am not a scientist. I am a database programmer who developed his style of development by working for many years in the database under various systems like Oracle, SQL Server, IBM DB2 and others. I don't claim to reinvent the wheel and I know that most of the things described in this book are and should be familiar to many specialists, especially to many database programmers.

SQL is such a common language that anyone who would say something new about it might be regarded with curiosity and distrust. There are many books and papers about SQL, many things have been said and written on the subject, and most of all, many software applications are written in SQL. For sure SQL is, along with some few other languages, in the top of popularity and usage. The things are clear in the area.

Therefore, this book is not a book about SQL in the sense that I consider something new to describe about it. What I want to talk about is the fact that writing SQL code involves a certain style that is somehow distinct from the most common styles that are used in others languages. Even in this area, the things are becoming clearer during the last years. I saw the tendencies and I know that, for example, the **set-based approach** is becoming more and more required in the software market, in many projects. What I want to show is the fact that, in certain situations, the software developers should use a certain **style of programming**. They should use a distinct style of development, specific to database programming and mostly for some specific types of software applications, not the classic ones. I am mostly referring to replication systems, data migration systems, Extract-Transform-Load (ETL) systems or any type of application that has the purpose of moving data between various classic software systems.

In the theoretical approaches on SQL, in the multitude of courses that have been written, most of the specialists that wrote these papers are trying to explain the syntaxes. Of course, they are trying to describe the cases in which one is supposed to use one certain syntax or another. They are trying to illustrate the use of SQL using a variety of exercises. They explain that SQL is a query language, so any SQL course is mainly dedicated to the topic of querying. This is the main purpose of SQL: how to get access to data in the relational format. I consider all these pretty much well known. I intend to consider more aspects, for example, the fact that, while working with a database and using a relational language like SQL, you should adopt a certain style of programming which is not the same as the one you are familiar with, as an application developer.

The source of this book is my experience and you can consider it as a **practical guide**. That is why I prefer to use the rather vague concept of style of development instead of trying to use a more scientific concept. In my view, this style means the set–based approach but even more than that. The set-based approach that I am promoting here is just the most important characteristic of that style.

In another train of thoughts, I believe that people are becoming more aware of the set-based approach and I see how more and more demand is required for this approach in the market. I think people are becoming increasingly aware of the need for a certain, distinct and specific style of programming in the database area.

However, I continue to see very often how many programmers and many projects are implemented in the same manner for the user interface and the database, without the distinction that should be made.

I consider that, using a vague concept of style and not a strong scientific concept, I can easily use my experience in the database and illustrate the style that I developed during many years of programming within the database. Without being a theory and without the strength and soundness derived from scientific argumentation, the use of an appropriate style of programming can add more value than a theory.

What is the most important matter for everyone involved in a data-oriented software application, apart from the accuracy of information?

One aspect is for sure the **performance** of the software itself. One is to run a data migration interface in one minute and another is to execute it in five minutes! Getting a good performance in the database is proportional, in many situations, with the use of the most appropriate style of development.

Let’s continue. The question remains, what is a style of programming?

Can we say that a programmer has his own style?

To a certain degree, I would say yes. Most of the programmers have their own style. Without being very strict and considering the classic developer that effectively writes code and not the visual developer, I can affirm that the style of a developer is related to the way he writes his code. The way the developer writes his code is influenced by many factors: his education and background, his experience, his types of projects, his role within the projects, his ambitions and why not, his talent, his seriousness and capacity of organization. We all know how important it is for a programmer to be able to organize his work properly; we all agree that he needs to be able to see the details without losing the whole. That would be the ideal programmer in an ideal world. Still, very often we lose ourselves into the details. From a different perspective, the models the programmer learned, in the college education - for example, and used during his experience influence his style. In one way or another, the style of a programmer describes his ability to write.

I would like to believe that the programmer is similar to a writer. The writer has more or less restrictions than the programmer depending on the way you are looking at it. The writer is restricted by his audience as the programmer is restricted by his end users and testers. The programmer is sometimes very technical but also the writer may be. The degree of creativity is an essential skill of a programmer like is for a writer. However, this degree of creativity is not absolute like it can be for a writer but relative because programming is a practical activity after all and not pure art. Anyway, just as a writer has his own style of writing, so a programmer has his own style of writing development code.

This is the main topic of this book, this is what I want to illustrate and describe a certain style of development and the need for a specific style of database programming.

I consider that a distinct style of programming is required when writing inside a relational database. I was able to develop and especially understand this distinct style of development in many years, in time. Being aware of the fact that I am not the only one, I consider that this style of development is not promoted enough and is not clearly explained in its details. I think that a more detailed description of this style is necessary and I believe that many IT people can take some advantages reading this book. One goal is to convince some application developers to reflect at this proposed style and change something in the way they write when they are switching to the database, if they will accept my arguments. Another goal is related to IT education systems and universities. I consider that this style may be better promoted in the database courses in some universities. Apart from understanding the principles of relational databases, apart from understanding the SQL language, apart from the delivery of specific vendor languages like PL SQL or Transact SQL, it is also very important for the young student to understand how one should adapt his code to be more efficient in the database. I do believe this path, the use of a specific **style of development**, should be followed by the developers that will write codes in the database. I also consider that this path should be described and explained in the universities so the students, the next future of programmers, will be warned that something needs to be changed in the way they write when dealing with data.

THE MOST COMMON STYLES OF PROGRAMMING

Now that I have clarified and described the concept of style of development, after admitting that this is a vague concept and not a scientific notion, I will try to identify some of the major premises for one style of development or another. Many factors influence a certain style of development. There are schools of development and these schools are based on certain models and theories. I want to investigate and to describe some of the factors that contribute to a certain style of programming. I will start with a series of questions. These questions are addressed to a programmer, eventually to an application developer but not only.

What type of programmer are you? What is your area of expertise? What programming languages have you used? What paradigm did you follow along your career as a software developer? During your career in the software industry, what were your main paths? Were you involved in many levels of development, for example did you write code at user interface level and database level too?

As I already mentioned, I am specialized in database programming and I am a database developer. I had been involved at the user interface level during my first years of experience but I liked databases too much and I preferred to specialize quickly in the field of database programming.

Are you that kind of programmer? Not necessary specialized in databases, you may be a highly specialized Java developer, for example.

Alternatively, you are a programmer that can do both and some other things too, very flexible, being able to switch from PHP or C# to Oracle database and PL SQL, for example. Are you that mixed programmer that performs very well in all the sections, in both user interface and database level? I have all the respect for these flexible programmers, as long as they make the distinctions and they are not trying to work in the same way in all the areas of the software.

Theoretically, when designing and developing an application, based on a certain language like C# or Java and using a certain relational database system like SQL Server or Oracle, the application can be written by any kind of IT specialists. In most of the cases, the programmers are able to satisfy both sections. They are able to write code at the user interface level, using Java or C#, and they can write their logic at the database level using SQL and PL SQL or any other specialized vendor database language.

The other approach, not so common, is also possible. Specialized database people are used for the database section and application developers are used for the user interface. Despite the advantages, this is not the general situation, on the contrary. In most of the projects, the software applications are built by application developers that are able to write in the database using SQL and the associated procedural languages like PL SQL. The developers are working on both levels; user interface and database, and they switch from the user interface to the database level, periodically. This is the most common situation, with advantages and disadvantages. It is a matter of resources and availabilities but is also a matter of skills and costs.

Let’s say a company wants to start a new project, it decides to use Java technology and the Oracle database. The managers do not generally search for specialized database developers or for specialized application developers. They imagine that a good programmer needs to know both and manage well both sections. They do not search for highly specialized Java developers to allow them to write Java code and not touch the database under any circumstances! They do not search for specialized database developers to do exclusive database development! Generally the managers want to hire people that can do both. In the most common scenario, the expectations are that the developers will be able to write C# or Java code, on one hand, and write SQL code, on the other. Of course, there are variations and everything is related to the specific of the project and application.

The opposite strategy is not so common, although I noticed that more and more projects try to be organized in the opposite manner. I also noticed an increase in the request for database developers, for example. There is a growing demand in database development that may be explained by the larger number of projects where more specific database expertise is required, like ETL projects, data warehouse projects. However, I believe that, even in normal projects, not so database specific, there are managers to think in this manner. Anyway, this is another discussion and it's difficult to decide. My personal opinion is that finally this mixed approach will continue to be the most common one. The programmers prefer the user interface or database but they are generally able to do both. The question is how are they doing it? Are they doing the job right in both sections? Are they using the same principles of programming in both sections? Alternatively, do they consider that a different approach may be required?

Any project and any software application is the mirror of a certain business to be implemented. Therefore, the business drives everything including the database. The programmers can be involved mainly in the current functionalities of the business, operational systems. For example, they can be involved in classic online transaction processing (OLTP) configurations, in classic production systems. Others developers can participate to specific projects like data warehouse, replication systems or data migration ones. There is a large variety of situations and software applications and the programmers should adapt their capabilities and should try to be flexible, they should write their code in concordance with the specificity of the project.

Apart from the classic programming, there is the new type of programming, very modern and fancy, the visual development. There are new species of programmers specialized in tools, most of them visual, that spent years and years working with certain tools.

This is a new type of programming, based on efficient tools with a lot of code in their back. Once I had a very curious and funny experience! I have been to my daughter high school when she was in her last year and spoke to the college students. Some of them were tempted to enroll to IT universities. I explained them some basic facts regarding our world of software developers and I tried to explained them a bit what are the advantages and disadvantages of a programmer, in my vision. One of the college students came to me and told me he is interested in software development. Moreover, he mentioned to me that he would like to be a programmer without writing a line of code, if possible! His dream was to become a visual developer! I don't want to comment in any way this vision but I believe it is a danger somehow if this vision will persist in the future. I can't imagine the new type of programmers, with no writing experience and working exclusively visual! Maybe I am too conservative!

So, coming back to the list of interrogations, are you a classic programmer that prefers to writes code or are you a visual programmer?

All these questions are reasonable in today's market where the complexity is so high and the number of alternatives is increasing year by year. There are so many technologies, so many languages and so many visual tools!

Why we have all these questions and sociological research?

All these interrogations are necessary when talking about software programming styles. The purpose of this book is to try to define and to clarify a **style of programming**, specific to the database and to compare it with a classic or typical style. The topic is generally addressed to developers, mainly to them, but not exclusively. A project manager, a technical leader, a tester, a business analyst with technical skills, a student in an IT university should be interested in the topic too. This book intends to help them understand that changing their style and adapting it to the necessities of the project is in their interest. For sure, everyone should win. All these questions are designed to fit the concept of **style of development** in the general context.

The next step in the discussion is the programming model, the general paradigm in which the programmers are bounded or linked. I just want to describe a general and relative view, without considering myself an expert into this. There are several common models and, according to these models, certain programming styles are predominant in the market.

The most popular model nowadays is the **Object-oriented** paradigm. This model involves a certain **style of programming**, adapted to the principles of Object-oriented programming. This model may be considered one of the most complexes. I believe this model is close to the reality more than any others are. The most common programming languages and frameworks rely on this model, already classic, like Java or C#. All these languages satisfy the principles of Object-oriented programming (OOP).

Object-oriented programming is properly described in the IT universities all over the world. Young and future programmers become familiarized with the model from the very beginning and they understand that most of their activity will be based on this model and paradigm. The principles of Object-oriented programming, like encapsulation, polymorphism, data abstraction and inheritance are explained both in theory and in practice through a variety of simple applications. Any junior or beginner programmer, when graduating the university, is aware of this model. Apart from the principles, apart from the model itself, a certain style of programming is promoted automatically and by default, most of the young programmers are already converted more or less to this style. The associated style of programming is, to a certain degree, a consequence of this model and most of the application programmers start the development in a similar fashion and adopt a similar style. This is very normal and rational, and everyone can agree with the idea that this is the predominant style on the market and a large part of the written software is based in this model and principles.

When analyzing the database level and the Object-oriented programming model, there are not too many things to be discussed. This model is proven unsuitable for the databases, at least for the relational ones. The data is too simple and the Object-oriented model is too complex. The **Relational model** is the one that drive our world of databases! The simplicity of the Relational model is obvious comparing to the complexity of the Object-oriented model. Consequently, the large number of attempts to convert the relational databases to the Object-oriented model was unsuccessful. Trying to adopt the Object-oriented model and its associated style of programming in the database was one of the reasons for many performance issues in the databases in the past!

Apart from the Object-oriented paradigm, there is another model that stays closer to the relational model and database programming. I am referring, of course, to the older model of programming, comparing to Object-oriented model, the **structured programming model**. The students learn both paradigms and try to understand both models. Later they will decide, depending on the situation and the specific of their projects, which model to choose and what associated style of development to adopt. When I say, "decide", I don't mean, "to say they explicitly adopt one style or another". The process of adopting a certain style of programming is somehow an unconscious process. People generally do not realize this and they adopt it unconsciously.

In the IT universities, while learning the **structured programming**, the students become aware of the simple concepts such as variable, structures and arrays, if-else structures or while loops. They learn how to build a function, a function that returns a void "value". The students learn how to create a procedure in certain languages. They become aware of the more modern and complex concepts of class and object, they learn about data structures, fields and methods. This is the base, and any programmer is aware of all these, any programmer learn these models and most of them start to apply all these notions in their effective work. Software applications are built following these principles of Object-oriented programming or the principles of structured programming.

I believe we can consequently accept as programming styles those styles of programming that are in conformity with the principles and models described above: like Object-oriented programming or the simpler model of structured programming.

All the students understand, more or less, these paradigms and later on they will apply these principles and they will be tempted to adopt one style or another in their software development activity.

To continue the discussion, in a different train of thoughts, one common characteristic of a good programmer is the generality. A good programmer is the one who always tries to be as general as possible. Being so, the developer will be able to cover the particular, and handle all the particular situations that falls into the general.

Considering that, Object-oriented model is the most general and complete programming paradigm someone may agree to the idea that this model can be used as a baseline for any programming activity. Moreover, the style of development that every programmer has is in concordance with the preferred model in most of the cases.

This is generally true but, like any rule and principle, there are exceptions. The database, the relational model is too simple and it's simply not compatible with the Object-oriented programming.

The structured programming and the Object-oriented programming are the most common paradigms and most of the programmers are adopting one associated style of development or another, depending on their projects. Whether they are developing in Java, C, C#, Delphi or anything else, the programmers are adopting one of these two styles of development. All these are well known and there are thousands of books and demonstrations about how to write code in one way or another.

Still, there are situations when these styles are not convenient. There are certain types of projects in which none of these two styles are required. Alternatively, better said, there are situations where these styles should be adjusted to accommodate new features. Still, these styles are used as such because the programmers are not always aware of some distinctions and are not aware that sometimes they need to try to adapt to **others** scenarios.

It is not hard to guess what kind of scenario I have in mind.

What happen if we are in the position of developing in a database?

Moreover, what should we do if we are instructed to write code in a relational database?

Are these styles of development the most suitable ones for working within a relational database?

These questions drive the entire content of this book. In addition, these are the questions for I would like to offer an answer.

Let's remember some simple facts looking back in the history of software development.

The popularity of the Relational model is out of question. A large part of the businesses that we model, either production or sales, maybe supply chain planning or inventory, or flights reservation, all of these and more others, are implemented in relational databases. Apart from the variety of online transaction, processing (OLTP) systems there are more and more historical databases such as data warehouses used for analysis and prediction. All the associated suite of software applications are built mostly for this data stored in our relational databases.

The database is a critical component of the software and huge efforts are oriented to these databases.

The two main goals to be achieved are trivial. The databases should have a correct design so the data will be stored fairly and the end users should be able to see the reality of their business. The logic in the databases and in the associated applications should be consistent and accurate so the performance will be acceptable. This mainly means that the response timings should be good.

Considering the importance of the database component, the use of a certain style of development should not be done automatically. For example, let's see this scenario.

John Doe is a C# developer, he spent many years writing C# code and now he needs to build some logic in an Oracle database. He would write his code in a similar fashion. It is true that the Object-oriented model is not suitable for the database, so he will use a mixture between structure programming and Object-oriented programming, as much as he possibly could! He cannot use classes but he can use, for example, records or types if he is using PL SQL. I am sure you will imagine that the list may continue.

My opinion is that this is one of the biggest issues for many application developers that need to move to the database and need to write logic at the database level. They adopt an inappropriate style of development in most of the cases because they are used to a certain style, to the typical style. They not analyze and they are not aware of the different models. They don't realize the necessity of changing something in the way they write their code when they handle the data.

Being a programmer is a vague definition today. People are specialized in one or another language or tool; they are specialized in one kind of business or another, in one type of software or another. There is a large variety of projects and it is difficult to ask someone to know everything.

Considering all these, I will minimize the area of interest from the beginning and continue the entire topic based on this limited perspective.

I am discussing about database development. I will assume that we are developing in a relational database. From now on, the entire discussion is particular, but still critical considering the importance of the databases.

Let's conclude! Personally, I am a database programmer. I am using a certain style of development that I was able to build in time and I am planning to promote this style now. I am specialized in databases! I chose this path because I like databases, because I like to dig into the data. I cannot pretend to anyone to become as a passionate SQL people as I am. Rather than that, I would like to explain to others programmers my opinion about the way they should write their logic. Especially to those application developers who need to write code in the database, due to the circumstances of their projects. The database is a critical component and things are getting very slow over there sometimes. I believe that one reason for this low performance is the fact that an inappropriate style of development was used in those databases. Let's see how all this was possible!

THE DATABASE DEVELOPMENT STARTS WITH THE TABLE DESIGN

I know that this sub chapter may be seen as inappropriate or too basic by some of my readers. The reason is very simple: the basics of table design should be clear to everyone, readers of this book. Theoretically, everyone knows what a table is, everyone knows what a column is, and everyone should know what a constraint is! Still, considering the large audience as described in the introduction, I would like to describe some of the basics considerations regarding the table design. I hope some of my readers like database developers or simply developers with enough experience in database development will forgive me and quickly read this sub chapter or simply move to the next one. This sub chapter seems to be somehow apart from the topic, although I will try to argue the opposite.

As I mentioned earlier, I am discussing about how people should write their code inside a relational database. Let’s imagine us as part of a team of programmers preparing to start the development. We are already familiarized with the two models, the Object-oriented model and the structured model and we learned the SQL language. The relational database can be anything. It can be Oracle, SQL Server, DB2 or PostgreSQL. Things are pretty much the same when talking about the style of development in a relational database.

I consider that the database development starts with the design. That is why I included this sub chapter here, because this is the starting point in our development activity! So let’s try to review some basics!

There is a list of object types (nothing to do with Object-oriented programming) in any database, and among various classifications of these types, one is the most important. We can classify the objects in **base objects** and **procedural objects**. The developer is obviously involved mainly in the set of procedural objects. However, despite any appearances, the development actually starts from the base objects design. To be more specific, the development starts from the **table design**. A good developer should know that and not undermine its importance.

The database is firstly the sum of its tables and the table is the center of the universe in the universe of databases! I am referring to relational databases and I don’t want to upset the promoters of others types of databases, non-relational ones. An efficient database development means, firstly, the proper design of the tables. The tables design starts from the business, like anything else in software development. The table is the mirror of the business, in the sense that the useful business information is stored in the tables. When I define a student I need to know in what way a student will be stored in the system, I need to know what are the characteristics of the student, not any characteristic, but the characteristics that are relevant for the university.

The database developer needs to be aware of the meanings of the tables. The table design can be implemented by specialized architects, by business analysts or by developers, but the developers need to have a good understanding of it. The set of procedural code they will write have one single purpose, to get everything from those tables.

What does it mean “the table design”? Of course, this is an elementary question.

The table is a combination of **columns** and **rows**, like an excel file. Even a non-technical person, let's say not a programmer, a secretary, an assistant manager, can easily understand what a table is when comparing the table with an excel sheet. I know this may sound like a blasphemy to some orthodox database people but this is a good comparison for the sake of argument! At a closer look, the table means something else but at the first view, and from the logical perspective, there are similarities and it is a good starting point for a non-technical person to understand the table by comparing it with an excel sheet, considering the popularity of excel sheet.

There are two images for a table, two ways of looking at, the design view and the execution view. In the design view, one can see the definition of the table and, he can see the columns. In the execution view, one can see the data, he can see the rows and he is able to analyze and understand the data.

Let's talk about the design view, the table definition. That means simply the set of columns that compose the table, and the columns in the table correspond to the characteristics of that something that needs to be defined, in this case the student or the instructor, for example.

The column should have a **name**, unique for a specific table. The name cannot exceed a certain length, according to the database system. The column should have a certain **data type**, from the available list of data types for each database system. The most important data types are string, numeric and data, with variations and sub types. The column can be a **business** column, with a clear business meaning, like the first name of the student. The column can be a **technical** or **artificial** column, used for implementing the consistency of data, like an **identifier**, for example a student id. This column has no meaning by itself and the end user will not understand anything from it. However, these columns are very important for the developer and he is manipulating these artificial columns with priority.

The table design continues and so the initial development. The table designers define the columns, they specify the relevant names and they associate the correct data types according to the business requirements. The first stage of development, the table design, ends with the layer of **constraints**, a critical aspect of the design but also part of the first layer of database development. The designers, that sometimes are the developers, should be able to use this facility and define all the constraints correctly.

I will shortly remind the most common types of constraints. Various database systems may have them all or not.

1. The first type of constraint is the so-called **NOT NULL**. A column can have such importance that should always be filled with something when data is added into the table. For example, the name of a student should not miss, what is the relevance of a student if we do not know the name! That column will be defined as mandatory. This means that, whenever try to add a new student, for example, the name should be specified, otherwise an error will be raised by the system and the student cannot be added. Adding the **NOT NULL** constraint whenever is possible is a very good practice. As long as we have less null values, it is better for our development, as we know how many problems are

caused by the null values in our logic!

2. Maybe the most important type of constraint is the **primary key**. One of the principles of a relational database is the fact that we should try to identify one **row** in a table. The identification should be unique. Normally, a good design means not accept the existence of tables without a primary key. Any table should have a primary key, at least in a normalized database. It is such a feeling for a developer when doing his logic later to be sure he can touch the row without any doubt! The developer should always be aware of the primary keys and will manipulate these later very often during his development activity. There is one primary key per table. The primary key can be an artificial column or not, but I recommend the use of artificial columns whenever it's possible. That kind of column has no meaning apart from its role, in this case the role being to identify one row in a table. Still, the primary key constraint can be defined on a business column, like Social Security Number, but it is not a common practice. The primary key can also be defined for a combination of columns. If we want to define the constraint for a pair of columns, the combination of columns should be unique.
3. A similar type of constraint is the so-called **unique** constraint. This constraint is similar with primary key, in the sense of the uniqueness of the column, or pair of columns. Still, there is a difference in the meaning. The purpose of the primary key is to identify a row in a table. The purpose is highly artificial. By contrast, the unique constraint is generally a business constraint, specifying that a certain column should be unique due to the business requirements. For example, the social security number is not the perfect column for a primary key, although we can use it if we really want to. However, it is a perfect column for a unique constraint. Comparing it with a primary key, one per table, many unique constraints per table are generally accepted. The unique constraint can also be defined for a combination of columns, which means that combination of columns should be unique. The perfect combination, in my opinion, is to have an artificial primary key and a business unique key, if it exists. It is good to have them both, whenever it's possible!
4. A more complex type of constraint is the **check** constraint. This is a simple formula that should be applied to one column in the table and implement simple rules. The most common one is the affiliation to a list of values, like gender, that can be either male or female. Generally, but not always, a check constraint can be combined with a **NOT NULL** constraint. Combining check and **NOT NULL** is very important because this way we will cover all the situations for that column and leave nothing out. We shouldn't forget that we are preparing for the development and we are in the beginning, the table design. However, we have the logic in our minds, the set of procedures and functions that will follow.
5. Now we will understand why the database is called relational! For that, we will analyze the **foreign key** constraint. This type of constraint implements the relations from the database. Normally, every table has a primary key, so it is uniquely identified by that primary key. Every table should store the distinct type of information in a transactional and normalized system. The tables are

related between them by foreign keys and are the base for most of the **joins** in the logic. Understanding and properly defining foreign keys it's also a critical step and the set of foreign keys is the key to the understanding of the joins, that are to be found everywhere in the logic that will follow.

6. The last type of constraint is the **default** constraint. This is not seen as a constraint by all the systems. It is not actually a constraint because does not restrict the column in any way. This is just a default value that is added in the absence of an explicit value. For example, most of the students in a university in Paris are right from Paris, let's say 80%. In this case, whenever adding a new student, based on the probability for that student to be from Paris, the locality can be skipped at insert time and the default value, Paris, will be added automatically.

This set of constraints is the first layer of consistency of the data within the database. A programmer that is working within the database needs to be aware of this layer. This is not part of the logic itself, our topic of discussion, but it can be considered as such, even if not effectively.

For example, the gender is checked by the values male and female, this can be done in a stored procedure without any problems but why would you do it? The programmer will have an error handling procedure, catch the error identifier and the error message, and detect the name of the constraint, which is violated, and identify the table and column and raise an intelligible message. However, the rule itself is checked by the database layer, the first layer of constraints.

The programmer writes the logic but he needs to be aware that his logic already started with this layer of table design and constraints. His logic is generally for the data, the data is defined in tables and the developer will continuously manipulate these tables. The potential data should be carefully analyzed; the business should be explained by the persons with knowledge, like the administrator of the University. Based on this information, when designing the tables, the programmer will start the development process by implementing a correct table design including the layer of constraints.

One important matter about the constraints is their names. It is a very good practice to give explicit names and relevant in terms of business. Don't forget that the constraint names are visible in error messages and, when see the message, if the name is explicit and relevant, you will understand right away what it is about and quickly identify the starting point for the investigations. Apart from that, the developer can easily find the objects in the metadata associated with every database: this is also an important matter.

Now let's come back to the application developer, which is thrown in the middle of a database! More than that, this application developer may not be familiarized with the database almost at all, he does not know too much about tables and columns! Even if this is not very common, we know it could happen. He knows what a variable is, he knows what a data type is and in which way he needs to associate a data type with the variable, he knows that he needs to specify a name to the variable and he knows that the name cannot exceed a certain length. He is also familiarized with the excel file! He can easily translate everything and has a basic understanding; he can have a starting point.

So, what he needs to know first when dealing with the database development is not the logic itself, he already knows the principles of structured programming so he has a good background that he will apply, to a certain degree. He firstly needs to be familiarized with the basic objects, with the tables. The table is the object type that will be accessed in his logic almost everywhere.

More or less these are the basics things that can be said about tables and columns. As you can see, it is not too much. Things are very simple and a good table design follows all the above. See below an example. We will create a table and we will add a variety of constraints to enforce certain rules for the columns, rules that have been shortly described earlier.

Code example 01: Design example

```
CREATE TABLE Students
(
      Student_Id INT NOT NULL,
      Student_Name VARCHAR (30) NOT NULL,
      SSN VARCHAR (30) NOT NULL,
      Locality_Id INT,
      Birth_Date DATE,
      Gender VARCHAR (10) NOT NULL
);
ALTER TABLE Students ADD CONSTRAINT PK_Students_Student_Id
PRIMARY KEY (Student_Id);
ALTER TABLE Students ADD CONSTRAINT UQ_Students_SSN
UNIQUE (Student_Id);
ALTER TABLE Students ADD CONSTRAINT CK_Students_Gender
CHECK (Gender IN (‘Male’, ‘Female’));
ALTER TABLE Students ADD CONSTRAINT FK_Students_Localities
FOREIGN KEY (Locality_Id)REFERENCES Localities (Locality_Id);
```

This is an example that illustrates the above considerations. You can see the primary key, an artificial column Student id with no business meaning and with the simple goal to identify one row in the table, the uniquely identify a student. In this way, we will always be sure that we can read and write properly a certain student, without any doubts. You can also see the column Student Name, a descriptive field but mandatory, a column with a clear business meaning. Let’s look at the column SSN. The social security number is a business column that holds the attribute of uniqueness. This is the business key of the table, and a unique constraint was defined for the column. The column gender is a column with a very low selectivity (only two possible values male and female). A **check** constraint was applied to the column according to which we are sure that no other value will be specified for this column. The locality is referenced by the **foreign key** constraint. The tables are linked by relations and these relations are implemented and **checked** in most of the cases by the mechanism of foreign keys. We assume we already built another table with the localities; another primary key that is referenced in students table will identify one locality.

The table design, without being pure development of any type, it can be considered as such for someone working in the database. A developer not being aware of all these cannot develop properly in any database.

DO WE ALREADY START THE DEVELOPMENT?

The example above is extremely simple. These basic considerations are very familiar to most of the programmers that work with data. The goal of this book is not to describe SQL but to promote a development style. Still, an introduction in relational database and the SQL language is required.

We started with the table definition and we tried to illustrate the most common set of constraints attached to the tables. We illustrated a set of Data Definition Statements (DDL).

This is the section of table design.

In the relational database, we may have various classifications. As we saw earlier, we have base objects and procedural objects. The table is the base object by definition, the most important one. In the relational database, everything is for, against, and with the tables. That is why a programmer should firstly understand this simple object. You are in the relational database and you handle tables! You look into the relational database: you are looking at the tables! It is much simpler than what you already know, all you need is to be open and try a paradigm shift.

We are discussing about programming and about the necessity to adopt a certain **style of development** that is different from what we generally know.

The concept of **table**, the idea of **row** and **column** are the ones that show up in front of us.

The first task for the programmer is to understand the table. Rows and columns, is too simple! It is surprising that sometimes even simplicity is a problem. Instead of experiencing satisfaction, some typical developers are complaining about the simplicity of the model.

Someone might say that this has nothing to do with development, eventually that is a work for a DBA! I disagree completely and I consider that a database developer should understand the table design better than the DBA. The DBA generally does not care too much about meanings. He is an administrator; he has a very important role. Still, the DBA does not know the business so well because he is not an expert of that type! The database developer knows it much better! In some books, you can see that the DBA creates the tables. Actually, the table creation is part of the database developer responsibilities not the DBA. The DBA will just build the appropriate scripts for the production environment but the tables were created by the developer in the development environment long time ago! The database administrator (DBA) is aware of the table design because he is managing the system, especially the production. Still, the true creator of the tables is the database developer because he knows the meanings, he is building the logic and he knows what the purpose of one column or another is because he will manipulate these in his logic later during the development process.

The database developer is the one that handles the logic within the database. In a variety of applications, the complexity of the logic within the database is extremely high. This complexity can be handled in many ways. Having application developers working in an inappropriate style in the database, using their specific programming philosophy and not adequate to the database, using their own style in the database will harm the database infinitely even more than some lack of indexes, for example. An application developer should try to understand the principles of database development when he will need to write code in the database.

Let’s see a final review once more!

We will start the database development and we will start writing code into the database. For that, let’s try to see what the responsibilities of a database developer are. Let’s see some examples.

1. The programmer builds the logic that allows data access.
2. The programmer builds the logic that allows end users to read the data and to write the data.
3. The programmer may build the mechanism that may allow transfer of data between various databases.
4. The programmer may build, for example, an ETL, or a replication system or a data migration system etc. In this case, he will be responsible for the data transfer between various systems.

These are just some examples of tasks that a programmer may have. There are various situations and I will try to analyze some of them later. The programmer can be a database developer or can be an application developer that is doing database development along with his application development.

The programmer should be aware of the tables, he can design them or not, but he definitely uses them all the time so he needs to know them very well. Regarding the set of constraints like primary key, check, **NOT NULL**, unique and foreign key, a good and fair implementation is already a key to a successful development. More than that, a correct design of the constraints is part of the primary development.

Everything starts with the table design. Let’s see one example of good design wrongly implemented. I can show you how a good table design may lead to a nightmare in the future development if is not properly understood. Let’s say we have an important entity in an application, for example an invoice. We have an invoice number and an invoice identifier, an artificial primary key. This is the header. In the invoice detail, we have a combined primary key between the invoice identifier from the header and a current number. See the design below.

Code example 02: Design example

```
CREATE TABLE Invoices
(
      Invoice_Id INT NOT NULL,
      Supplier_Id INT NOT NULL,
      Invoice_Date DATE NOT NULL,
      CONSTRAINT PK_Invoices_Invoice_Id
PRIMARY KEY (Invoice_Id)
);
ALTER TABLE Invoices ADD CONSTRAINT FK_Invoices_Suppliers
FOREIGN KEY (Supplier_Id) REFERENCES Suppliers (Supplier_Id);
CREATE TABLE Invoices_Details
```

```
(
      Invoice_Id INT NOT NULL,
      Current_Number INT NOT NULL,
      Quantity INT NOT NULL,
      Currency VARCHAR (30) NOT NULL,
      CONSTRAINT PK_Invoices_Details
PRIMARY KEY (Invoice_Id, Current_Number),
      CONSTRAINT FK_Invoices_Invoices_Details
FOREIGN KEY (Invoice_Id)
      REFERENCES Invoices (Invoice_Id)
);
```

This is an example of a good design. Still, if implemented incorrectly, it may have bad consequences. Imagine that the column Current Number is a volatile column and, every time new details are added to the invoice, the values are recreated based on certain criteria's. Like a true current number. For example, let's assume that the details are shown and the current number is generated ordered by quantity. In this case, the first detail today will become the third one tomorrow. Imagine that the details should be updated. Very often, an update is done based on the primary key. In the given conditions, this is impossible! Another additional field, to satisfy the condition of uniqueness, needs to be added. This is true if the mechanism of current number cannot be changed. Alternatively, the current number will not be recreated but kept with every change in the invoice.

It is critical for the primary key to be set correctly because very often this is the criteria for update. If we want to **update** anything, we need to be able to **identify** it first. The primary key should not just be unique but also **stable**. The **stability** needs to be combined with **uniqueness** for the primary key to be indeed the criteria for row identification and, consequently, for update. Otherwise, we need to use something else, maybe a business unique constraint, for update. Imagine we have a service a part of an invoice, with the current number 12. The invoice id is 100. Therefore, the pair 100, 12 identifies this invoice detail. I want to update something in this detail, first I query the detail based on the pair 100 and 12 and then I update. If the current number is volatile tomorrow will become 20. The pair will become 100, 20 instead of 100 and 12. Trying to update the pair 100 and 12 will update another detail. Therefore, this is a good design but not correctly implemented by the application developer. The primary key should be unique, mandatory. More than that, if the primary key is the criteria for update it needs to be stable. **Stability** is another component of a primary key and this is something that the developer needs to be aware. Most of the developers' work is on these tables, trying to populate them with data. This is the reason why understanding the tables and the associated constraints is critical for them.

Please try to can see how the software programmer is already involved in the logic even before starting it. Later the developer will received some tasks to update the details of the invoice. He will need to make a huge effort because someone did not understand the table design. The combination between the invoice number and the current number is unique. This pair will identify one row in a table. Still, the primary key is not stable because the current number is volatile. The primary key is not persistent, the current number is changing all the time after the table is updated and the developer is forced to find other ways to identify the rows, he will need to define and populate another artificial column to solve the problem. He should add a unique business constraint, but that one will still be artificial. Being involved in the table design from the beginning, possessing the proper knowledge, he would have been able to avoid all these issues.

That is why, when seeing in different books that the DBA is responsible for the design of the base objects like tables I am becoming very confused. He is responsible for the implementation in production eventually adding partitions, indexes, using parallelism, but the design itself is the responsibility of the programmer, of the database developer.

The programmer is familiarized with classic programming and he knows very well one model or another, Object-oriented model or structured model, he might have Java or C# experience. Then, he needs to deal with relational database development. The first contact is with the table. He sees only rows and columns. Some programmers don't like this model of data due to various reasons. Nevertheless, whether they like it or not, the simplicity of the relational model is so obvious that no one can deny it. Moreover, this is not a bad thing in itself. Of course, nowadays, after so many years, everyone is familiarized with SQL and relational databases. Still, many programmers are not able to solve things properly in the database and try to apply a style of development that may be suitable at others levels but for sure is not suitable at the database level.

ARE WE READY FOR SQL?

I just clarified the concept of table as the most important one for a database programmer or application programmer dealing with data. The tables are related to each other by logical relations enforced by the first layer of constraints.

We have a database; we have a variety of tables. We want to access these tables, to populate and read the contents of the tables. What are we using? We may be true database developers, or we may be Java or C developers. Perhaps we are used to deal with systems in the classic way and we need to handle data in a relational database. We can use any database system like Oracle, or SQL Server, or maybe MySQL or PostgreSQL. We are now transferred in the middle of a specialized data warehouse that is using Teradata and we need to work in an ETL interface system. Well, we have good news. We have a standard, as you know. SQL is a standard and the advantages can be seen by anybody who has the chance to deal with many database systems.

SQL is the standard and most of the relational database systems are using it. It is true that this standard was implemented differently from one system to another and it is true that there are differences between syntaxes. Still, the differences are minor. Dealing with one database system and switching into another database system should be relatively easy. Moreover, it is indeed! As you know, any database system has its own programming language that is an extension of SQL. Oracle has the PLSQL language; SQL Server uses Transact SQL language and others. However, the SQL is almost the same, all the software suppliers realize the advantages of having a standard and they tried to adapt the syntaxes to be compliant with it.

This is one reason to say that being a database developer means, first, being a SQL developer. The existence of the standard makes things easy for database developers that can easily switch from one database system to another. We will see more on that soon.

Now it is time to see what SQL actually is. The review will be basic because I consider this quite common and the topic of the book is not to describe the SQL language. SQL is one of the most popular languages nowadays after so many years of usage. Still, the number of true and good SQL developers is not very high.

This is one reason to say that being a database developer means, first, being a SQL developer.

There are PL SQL developers, for example. There are people who say that they work exclusively in Oracle database as developers. Alternatively, there are exclusive Transact SQL developers. This is somehow absurd, in my opinion! I believe that there are mainly SQL developers. A true database developer is mostly a SQL developer. Most of his work is pure SQL. Afterwards, to learn new syntaxes to create a stored procedure in Oracle or SQL Server is not a big deal! The transition from one system to another is very simple. The principles of structured programming apply to all these dedicated database-programming languages. The existence of the standard makes things easy for database developers that can easily switch from one database system to another. Now it is time to see what SQL actually is.

Chapter 2

SQL - THE BEAUTY AND THE BEAST!

HOW CAN A QUERY LANGUAGE BE SO IMPORTANT FOR A STYLE OF DEVELOPMENT?

In the first chapter, I analyze the table design, including the layer of constraints, and I conclude that this is actually the beginning of the development process.

Let's continue with the next step!

When we speak about development, we should speak about a language. Any classic development activity requires a language. Do we have a language? The answer is obvious, yes, and that language is available for many years; it is the universal language for relational databases. The surprise is the fact that this language is not a typical programming language. That language is a **query** language. It is the popular **Structured Query Language**, the so-called **SQL**.

In the beginning, someone may be shocked. The topic is the necessity of a certain **style of programming** inside a database. So we are thinking, obviously, at a classic programming language.

It comes as a big surprise the fact that we are discussing something else!

We are firstly talking about a query language and not about a classic programming language! This seems unusual to anyone not familiarized with the databases.

In a data, oriented software application there is a large variety of people involved in the generic activity of **query**. A customer support people, a business analyst, a consultant, a tester, and a QA analyst they all query the data regularly to get whatever they need. They are not doing any programming, they are simply querying. Almost everyone is querying the data in a certain way, and almost all of them use SQL for that purpose. SQL is not a dedicated language for programmers. SQL is a language for every category of people that, for some reason or another, need access to data. There are so many non-programmers working in software companies with a good level of SQL! This shows once more that SQL is something else than a classic programming language, it is very close to a natural language.

A database will requires a query language useful for data access inside it. This seems to be a valid statement for any kind of database. As I already said, this query language is not necessarily a programming language, in the classic sense of its definition. This query language can be embedded in a programming language and used by database programmers, or it can be used as such by all the categories as the one mentioned above for the simple purpose of data access, for the simple purpose of interrogation.

I assume that the query language should exist, for any kind of database system. A programming language that is specific to a database should use the query language because the main purpose in a database is data access in one way or another. Therefore, despite some opinions, the query language is critical and mandatory for a database programming language and a good programmer working inside a database should have at least an acceptable knowledge about the query language.

A parallel with old times comes to my mind, when Latin was the only academic language and all the others were considered barbarian languages. The books have been written in Latin for many centuries and, for many years, Latin was the only accepted language. Somehow, some programmers consider a classic programming language like Latin and they consider SQL, for example, like a barbarian language. They say, for example, that since SQL is not a dedicated language for their preferred category and it is used by so many other categories apart from theirs, this makes SQL a sort of primitive and barbarian language.

I assume you agree this is an arrogant and irrational vision. I strongly disagree with it, as we all should do.

The software development world is a very practical world. The quality of a language consists firstly in its utility. More than that, the utility should be associated with simplicity. SQL is a very useful language: it was proven to be as such for all these years and the fact that it is used by so many categories of professionals, not just by programmers, is a quality and not a disadvantage. The fact that non-programmers can learn it proves that SQL is a good language and it may be used as a query language by almost everyone in the enterprise.

On the other hand, programmers, either specialized database developers or any other kind of developers doing some work inside a database, use SQL as part of a programming language in their activity of database development.

Even if a programmer should have a more complex understanding of SQL, sometimes we can find a tester or an analyst having a better perception of SQL than an application developer does! Things are also related to everyone’s experience, of course!

When dealing with data in a relational format there are some basic things to do! First, we need to understand the concept of a table, the concept of row and column. We need to understand the table design, eventually with its layer of constraints. Secondly, we need to understand and use the associated query language specific for data access in that format.

This is the first paradox for an application developer that needs to do some work inside a database. He knows classic programming and now he needs to reconsider himself and make contact with another type of language, much simpler than what he already knows, and, more important, a different language than what he is used to, a query language. This is confusing, at least in the beginning.

The application developer needs to understand that a query language is required due to the nature of data. This is a very basic and trivial statement. The data exists to be read and written. We need a way to access the data within our databases. This is the definition of a query language; it allows data access, read and write. More than that, it allows metadata access, both read and write.

This is one difference between the two levels: user interface level versus the database level. The goal in the database is specific and particular, data and metadata access. From the beginning, the expectations are very clear. There are the expectations of a reader, the expectations of a writer, all in a certain format. At the user interface level, we can theoretically and potentially expect anything. At the database layer, we have one main expectation and we want only one thing. I am sure you know very well what that is!

The presence of a query language is already one premise for the necessity of a certain style, distinct from the one we are using at the user interface level, where the style is driven mainly by some general models like those mentioned in the first chapter. The style of development is driven by these, among others, but not exclusively! Sometimes others factors may determine the style of development, like the nature of data, for example.

In the database the required style should be conform to the particular expectations and goals described above. The style of development should be somehow associated with the goal: see the data, get the data, write the data, change the data, delete the data, and move the data! Things are quite clear and there are reasons for the database people to be happy, everything being so obvious. Since the goal is straight and simple, so the style should be. There are no reasons to expect the same complexity that you see at the user interface level, with such complex programming concepts to apply. You don't need to be concerned too much about all these in the database! It is like when you buy some fruits at the market: apples, bananas, oranges and grapes and in the end instead of applying the simple rule of addition and multiplication, you start calculating some differential equations to get the result!

The fact that the query language is a natural language and the basics can be learned in one or two weeks is great! From what I know and always believed, in programming, **simple things are better things**. To me, the simplicity and naturalness of the query language is a big advantage for everyone, including for the application developers that start to deal with the database, and for the students finishing the universities and preparing to write software code.

Nevertheless, some application developers consider the opposite and they rather prefer to make things more complicated because they believe they have some artistic visions against programming! I agree with the idea that a programmer is somehow an artist, but always a practical one. The programmer has some clear goals and these goals are driven by the business. The programmer is not the absolute leader of his work as the artist is.

So the first thing an application developer should do, if he is really involved in database programming, is to try to forget for a while about his classic programming background and dedicate some time to understanding a simple but vital language for his work. He needs to spend time in understanding the popular query language, the **Structured Query Language**. He also needs to comprehend the importance of a query language in his programming activity and he needs to understand that a query language is something vital for his software development activity. He should not complain anymore about the lack of esthetics, for example, when speaking about SQL. You cannot compare apples with pears!

Even in the absence of SQL and not referring explicitly to the relational database, but speaking about any database, things would be similar. The data is for being queried and this activity is critical when implementing the logic inside the database. Therefore, the query language, because there should be a query language, either SQL or something else, is critical for the development process. Even theoretically, we may say that the query language should be integrated in the development process and the style of development is influenced by the query language.

WHAT IS SQL? WHAT IS NOT SQL?

This sub chapter is somehow similar with the one with the table design. Considering the extended audience, including students at IT universities and application programmers with limited experience in the field of databases, I would like to describe the SQL language. I know that most of my readers are completely aware of these basics and I ask them to accept this switch, from the complex considerations described above to the common description of the SQL language that will follow in the next pages.

Let's analyze the query language! The relational model of data is trivial: tables, rows and columns. For such a basic model, there is a trivial language. This language is a dedicated, specific language for accessing rows and columns, in almost any possible way. SQL is a query language and a standard, followed closely by almost all the vendors that build database systems.

As a query language, SQL is a set of instructions with a clear purpose, to allow data and metadata access. SQL is English like, and the statements are very natural and intuitive. The keywords are so common that you don't need to think too much to see what they want to mention!

Learning SQL is easy, even very easy I might say. Still, to become a good SQL professional you need to spend some years on the subject and this is not an easy task, under any circumstances.

First, as application developers, we always need to be aware that SQL is **different** in its nature. I had some discussions with Java programmers or with C# or web programmers that told me that SQL should not be integrated in their philosophy of programming. They don't consider this programming at all, claiming that this is not what they know about programming. Actually, they are right! SQL is indeed very different. More than that, SQL is not a classic programming language, according to their standards. It is true that you cannot compare SQL with C# or Java! SQL has a different nature and it is obviously something else. SQL is firstly a query language and any comparison is completely inappropriate. SQL by itself is a separate language, apart from any other classic programming languages.

Now, since we clarified what SQL is not - an unusual start for a definition but recommended here due to some misunderstandings that exist among some programmers, let us see what SQL is. I will start with the three words: Structured Query Language. There are three characteristics, part of the definition:

1. **SQL is a language**. Surprisingly or not, SQL is apparently not a strong formalized language because it is more like a **natural** language. The main keywords it uses are common words, English trivial words like select, insert, delete. Being a natural language, is one major reason why SQL is available to a large variety of persons, not just programmers and technical people. Many non-programmers are using it, at various levels depending on each one's skills and interest.
2. **SQL is a query language** as part of a classic programming language like PL SQL, for example. The purpose of SQL is to query the data and metadata. By

querying the data, I understand both read and write process.

3. **SQL is a structured language**. This means that SQL is organized in such a way that reminds us of one of the models, the structured model of programming. The principles of structured programming are satisfied in this query language to a certain extent, and this is one reason for which some programmers consider that the use of SQL is based on the use of structured programming. To some extent, this is true. I will explain more about that later. There are differences and these are coming from the nature of SQL, from the nature of the data.

A query language means less than a typical programming language and has a different and particular scope. Querying something is a particular task and, considering this limited goal, we can understand the essence of SQL language.

When we analyze a data-oriented software application, an ordinary application like an inventory system, for example, or a software application for a hospital or university, we see almost the same components. Of course, the technology differs, the methods differ; there are various distinctions between software applications. When talking about data-oriented software applications the purpose is to populate the database via a user interface, either web-based or maybe desktop-based. We have the user interface and the corresponding database in the back-end. In most of the cases, one of the main goals is to populate the database with information and to access this information. The end users will access the data in the database via the user interface and everyone will be happy! These goals, of reading and writing data in the database, are achieved by the use of SQL. By data, we obviously understand business data, like invoices, items or students, patient's information for the hospital, and any kind of business information.

As you all know, the data is written in tables. Apart from this well-known task of reading and writing the data, SQL is responsible for another task. The data within the database can be classified in two categories. A database store **data** and **metadata**. For example, whenever a table is created, metadata information is written in the database automatically. Any database system has a set of tables, called system tables or data dictionary, and these contains information about the business objects. When a table is created, the information that defines the table like the name of the table, the names of the columns, the data types for the columns, the constraint information, and all the design characteristics of the table are written in the set of system tables. This type of information is named **metadata** information, which means data about data. Therefore, when writing with SQL in a relational database, we understand either writing data or writing metadata and similarly when reading, we can read any of two types of data. These considerations are useful and allow me to continue to show the most important characteristics of SQL, the bread of any database developer and the bread of any application developer that should write code in the database.

There are several sections of statements, all of them described in any SQL course. Learning SQL means learning and understanding these statements. This is very easy. I might say everyone can learn these. Then the hard part follows; you need to start the process of understanding the data. Start the process of data selection; start to become friend with the data. This task will take years!

The SQL language is composed, mainly, by four sections of instructions, called sub languages, or subsets of the SQL language. The sections are the followings:

1. The first sub-language is the Data Manipulation Language, the so-called "**DML**". This sub section of SQL is the section responsible with the **data access**, in both ways as active and passive, by read and write. Any instruction responsible with any of these two major actions, read the data or write the data, is part of "DML".
2. The second sub-language is the Data Definition Language or "**DDL**". This is the section of SQL responsible with **metadata access**. Creating new objects, like tables but not only, creating any type of object in a database, is an action that generates metadata in the system. When create a table or a view, many rows are added in the metadata responsible with tables and views.
3. An additional sub-language is the Data Control Language or "**DCL**". In this section of the language, we handle the security within the database like assigning privileges, for example. Various objects are becoming accessible to the users and privileges are given to these. The most common instructions are **grant** and **revoke**.
4. The last from the main sections is the Transaction Control Language or "**TCL**". This section allows us to control the transactions, and this is a critical section for a developer. The transaction is a critical concept and the developer should have a perfect understanding of the transaction. At the database level, inside our database logic, one of the most difficult goals is to have a consistent control of transactions. This is one of the most challenging tasks for a database developer. The commit and rollback statements are critical in any database logic.

Any of these sections contains some specific statements. This limited set of statements is almost everything we need to know, we can say that this is the SQL language! The first section is the most important one for a developer. Most of our database development code is composed from "DML" statements; maybe 90% of the code written by a programmer within the database is composed of these instructions. Let's review them, very shortly.

THE "SELECT" STATEMENT

If we pick any SQL course, and divide it in half, we can see that the first part is dedicated to this type of statement. The "select" statement is dedicated to reading data (or metadata) of any kind. The "select" statement is the query itself, if we understand by query the passive process of reading. This is the most important thing for anyone working at the database level. The purpose of reading is the most elementary and critical goal for anyone dealing with data. The simplicity and the difficulty of SQL is the "select" statement; one able to select the data properly is a good database professional, especially a database developer or data analyst.

Most of the SQL statements can be divided into a sub set of phrases, called clauses. A **clause** is a part of a SQL statement. A "select" statement contains several possible clauses.

1. The first clause is always the “**select”** clause, in a chronological order. In the “select” clause of a query, we specify what we want to visualize, which columns or expressions. An expression can be anything, any combination between various columns and constants combined by the SQL operators, like addition, concatenation. The columns or expressions are separated by a comma. This is always the first clause in a query.
2. The second part of the statement is always the “**from”** clause. In this clause, we specify the source(s) of data. We need to mention the list of tables from where we will get the columns and expressions in the “select” clause. The list of tables or views should be valid objects in the database.
3. After that, we have the “**where”** clause. In this section, we specify the criteria for the data, what are the conditions that need to be satisfied for the data. For example, we want to see the students from Paris; we add criteria for the locality to be Paris. The conditions are separated by logical operators like conjunction or disjunction. Be aware of the distinction between the SQL logic and the classic logic. The SQL logic is a bit different type of logic. In this logic, there is an additional possible choice apart from true and false, the null. The null means the lack of any value.
4. If there are groupings, two additional clauses are required. The first clause is “**group by”**. If the data needs to be grouped by certain columns or expressions, we use this clause. For example, we want a list with the teachers and the numbers of courses delivered by each teacher. For that, we need to group by the teacher and count the courses.
5. If we have another level of filtering, at the level of groups, a new clause is required. This is the “**having”** clause. This clause allows us to filter at the level of groups. The having is correlated with group by clause.
6. If the data should be ordered, the “**ORDER BY”** clause is required. This clause allows us to sort the data according to the columns or expressions that we want to sort.

The queries (SQL “select” statements) and their clauses are maybe the most important ones for a developer dealing with database programming, the simplest ones and the more complicated ones in the same time. Getting the data is simple by itself, theoretically, but it may become very complicated and it happens very often, because the business itself can be very complicated.

The first thing we need to know, when dealing with data, we need to know how to query, how to select the data properly.

THE INSERT, UPDATE AND DELETE STATEMENTS

The rest of the Data Manipulation Language contains the instructions for writing the data. As you know, there is one type of instruction per possible write operation. The insert statement is for adding new information, the update statement is for editing, changing information and the delete statement is for deleting data in the database.

All these statements are well known, in theory and in practice. What is important to me is that all these statements are somehow related to the “select” statement, to the query. In almost all the cases, to be able to write something, you need to be able to read.

You want to insert something. If you want to add some manual new data then you should specify the values one by one for each column, and things are simple here. However, if you want to add some data into a destination from a source of data, you first need to be able to identify the source of data. In addition, that source of data can be a simple but can also be a very complicated query. Therefore, there are two main types of insert statements. The “insert values” statement, which means manually, specifies the values to be added for each column, one value will be inserted at a time. This is very straightforward. Then there it is the “insert select” statement, which means inserting the data into a destination from a source of data. The source of data can be trivial or can be a complicated query. The degree of complexity can vary from infinite simplicity to infinite complexity. This is another example to see how SQL can be extremely simple and extremely complicated, in various situations.

You want to update something. Updates statements can be very simple but can be very complicated, like with the insert. An “update” means changing some values for some columns for some rows, according to the business requirements. The “update” statement can be divided into two phases. You need to identify the rows to be updated. First, you need to execute a “select” statement and identify the rows to be updated. Secondly, you need to identify the new values, because, in an update statement, you update certain rows and you add new values for the columns you want to change. For the identification of the new values, you can specify some manual values or you can get the new values from some simple or complicated subqueries. An “update” can occurs from some sources of data and identifying these sources can be trivial or difficult. An update can be extremely complicated sometimes. You may spend even hours until you get it right!

You want to delete something, you have the “delete” statement. This statement can be again elementary or can be complex. Comparing with an update, things are simpler because you have only one problem here, to identify the rows to be deleted. You do not have any new values because you are simply deleting some data. Still, identifying the rows to be deleted can become a difficult task depending on the requirements or it can be a trivial matter.

To conclude, the general concept of query data can means either read or write and read data. Reading data means the effective query, the “**select**” statement. Writing data means one of the three actions, “**insert”**, “**update”** or “**delete”**. This is the main type of task for a database developer, one of the four described above. This activity should be incorporated into the programmer agenda and the programmer should adjust his style of development according to this task too.

THE CREATE AND ALTER STATEMENTS

Apart from the data, access there is the second layer, of metadata access. The data definition language is a set of instructions for defining objects, which means creating new objects. Let’s say you want to create a new table, a new view, or a new procedure. This is another process where the developer is very active, and a good developer should know how to search in the layer of metadata objects in the database.

WHAT ABOUT PROGRAMMING? IS THERE SUCH A THING LIKE DATABASE PROGRAMMING?

Until now, the discussion was centered on the language as a **query** language. The topic of this book is the style of a programmer, the way he should write and the way he writes software code. It is true that the context is different, because I am analyzing the style of a programmer within the database. Moreover, until now, the only discussion was centered on SQL, mainly a query language! This is a very disappointing experience for a dedicated application developer for who programming means just classes, objects, entities, eventually arrays, structures. We did not mention a word of all these until now, nothing!

As you know, most of the vendors developed their own true database programming languages, apart from the query language. Oracle developed for many years their programming language called PL SQL. Microsoft developed the programming language for SQL Server, the so-called Transact SQL. IBM developed its own programming language for DB2 and this is SQL PL. As you know, the list may continue with a variety of database programming languages, specific for a variety of vendors. Any vendor, any database system has its private, proprietary programming language.

All these languages are structured programming languages and they satisfy the principles of **structured** programming. Any beginner, while studying PL SQL or anything else will learn pretty much and to a certain degree similar things and features that he study in a typical, non-database oriented structured programming language. Any beginner will start with the basics. He will understand what a variable is and how he may use it; he will see what data types are and he will become aware of the available data types. He will study how to use conditional statements, loops, if–else statements; he will learn to handle exceptions. A very important feature of a procedural language, specific for databases, is the **cursor**. The cursor is a critical feature for someone working in the database, and it is a great feature because it allows us, in combination with the loops, to move inside a data set from one row to another and do various manipulations. The application developer is extremely happy when he sees the cursor facility and he learns how to use this feature easily and right away. Unfortunately, this feature is used in excess and is a reason for many performance issues in the databases.

We will see later more exactly, what I am talking about!

Afterwards, the developer learns how to create procedural objects like functions and procedures and he sees how a stored procedure is similar to a void function. The programmer learns all these things, he can use his familiar programming language and he is very happy. Finally, I am home, he says to himself!

Like a parenthesis, I had met people with experience in a certain database system being devoted to that database system. These persons considered themselves programmers for that specific database only. For example, Joe is a PL SQL developer while Joanna is a Transact SQL developer. They would never move to another database system under any circumstance! This is a bad decision, in my opinion. Let’s see why.

All programming languages I shortly described before are actually a **mixture**. PL SQL, for example, is a language composed of two types of instructions: **SQL** statements and **procedural** statements. All these programming languages are a kind of query-programming languages, due to their mixed nature. In the database, in a stored procedure or a function, you are always in the position to execute either a SQL statement or a procedural statement. Whenever you want to query something, mostly to read or write data, you are doing SQL. The database programming language is an extension of the query language, is not like a classic programming language. It is a big difference! The programmer needs to be capable to query the data: that is his first task. Anything else is secondary.

Another advantage of SQL is the fact that it is a **standard**. This means that the set of instructions is, more or less, available in a very similar or even identical manner from one programming language to another. Every database programming language uses SQL in its own way. Still, most of the syntaxes are very close to the standard. For example, a select statement is almost the same in Oracle and SQL Server and in others databases systems. Same considerations apply to an update or an insert statement. There are variations and differences but it’s very easy to move and write from one programming language to another. Under any circumstances you don’t need to fill uncomfortable when switch. Moving from Oracle to SQL Server or vice versa implies some differences. However, the movement is acceptable for any programmer. The fact that SQL is a standard and that most of the vendors try more and more to follow it is a great advantage for database developers and not only. They can easily switch from one system to another. In most of the cases, writing a SQL statement in one database system as Oracle could be executed similarly or even identically in others systems like SQL Server or DB2. From this perspective, for database programmers, it is very comfortable and accessible to change projects from one database system to another: they can use almost the same SQL statements. Still, sometimes there might be differences between database systems and sometimes the SQL statements cannot be written identically. In this case, the programmer needs to investigate and adapt one SQL statement from one database system to another database system. Portability should not be a goal in itself. I would say that, if I need to choose between **portability** and **performance**, I would always choose performance. Sometimes performance means writing SQL according to the specific syntax and not necessarily according to the standard. For example, in SQL Server, the updates are particular and they cannot be migrated to Oracle or others databases systems even if it is pure SQL. In this case, after some years of experience with both systems and handling various applications in both Oracle and SQL Server, I always choose **performance**. After all, it is not very difficult to translate one SQL to another if the performance is better. If there is a similar performance, portability is desired. Anyway, the existence of a standard assures maximum of portability by itself.

To conclude, do we have something like database programming? The answer is definitely yes. However, the programming language means, maybe, 80% query language and maybe 20% the extension. A good database developer is a very good SQL developer, or so he should be. The programming languages, the extensions, were made for the developers to be able to do certain things that cannot be accomplished using basic SQL. The set of procedural statements should be used when the problems cannot be solved with a simple SQL statement. Database programming mostly means SQL development, because data access is our goal when we are in the database. In addition, data access is SQL!

PROGRAMMING IS A PRACTICAL ACTIVITY!

Software development is a highly practical occupation. Most of the programmers are practical persons and not scientists. The most difficult part of their job is the struggle with one business or another, trying to understand it. On the other hand, they learn one language or another, they continue to improve their skills and they continue to learn new features while these are requested by their daily activities.

Normally, the IT universities should be aware of these realities. From what I know, there are serious differences between education systems. Without judging any of these systems, I can objectively say some words about some tendencies. In some places, the programming courses are highly theoretical, too far away from the practical reality of software development. Some of them are maybe good theoreticians, brilliant minds, but far away from the practical world of the programmers.

Maybe some courses are not closed to the real life, are too theoretical, too scientific. Most of the examples are taken from math and are highly non-practical examples. Some students are not aware that, after they will finish their IT University and they will start working, they will do something else. I am not saying math is not important, after all, it is actually our foundation. Math develops our minds as nothing else does. All these programming languages have been built by mathematicians and logicians and we owe enormous gratitude.

Nevertheless, doing software development means something else, the goals are highly practical and the situations we face are mostly practical, common and human situations. Our world is not perfect like the math world and the students should be in accordance with these realities.

Math is the foundation but this does not mean it is going deeper than necessary. It is one thing to be a mathematician or even a math teacher, and something else to be a programmer for a software company. Well, if you are a programmer working for NASA or for a mathematical research institution, you are maybe an elite programmer and you have the privilege to do a special type of work. Please accept my sincere congratulations! Still, most of the programmers are not part of those elite! Most of the programmers work for banks, for factories and companies, for hospitals and health industry, for tourism sector. Most of the programmers are practical people and not scientists.

I was doing database programming for many years and the math that I used in my career was high school level, very rarely more than that.

Math is a world in itself. It is almost a perfect world and certainly a very beautiful world. Programming is our practical world: our economy, our education, our sports, our industry, our hobbies, everything.

A programmer needs to be able to understand different businesses and to adapt his knowledge to all these. He should be one of us, not a savant far away from our world.

If someone wants to build a software application for an inventory system, he needs to understand the inventory business. He may not have the deeper understanding that an analyst has, but at least a sufficient understanding.

If you want to build an application for a game, you need to understand that game, you need to understand the rules that the game needs to follow; you need to play the game.

This is software development and this is programming life experience. Programming is not science, programming is not math and programming is not differential equations! Programming is life, programming means the ability to live in our world and the ability to understand and implement one business or another. This is actually the main reason for which programmers are valuable people: this is why programmers are so appreciated these days. An engineer learns and implements his own technology, a doctor learns all his life to cure and heal others, and a taxi driver knows his own roads like no one else. A programmer needs to be able to understand each ones business and he needs to be aware of how to implement that business into a software application.

A programming language is not a difficult thing by itself. It is true that not anyone can understand and write programs in a certain programming language or another, certain skills are required. You need to have a logical mind, to be able to understand what an algorithm is and implement it in a language. You need to have a solid math background, but it is not necessary to be a mathematician. These skills are not as rare as many people might think. I met many people, young people that were simply afraid of programming and not enough self-confident in their own skills considering aprioristic that they will never be able to understand and write code. Still, very often, their skills were more than acceptable. This is one paradox for our world: it is well known that this is a good occupation nowadays.

A large numbers of teenagers avoid moving into our area due to lack of self-confidence. I have some good news for all these persons: programming is much easier then you believe, just try to see that you can do it!

What it is difficult in programming it is not the language itself and not the set of theoretical knowledge that you learn in universities. The difficulty consists in the ability to understand what is to be implemented, in the ability to live and model the life. Because all the businesses that are modeled by our applications are part of our society, are parts of our life. What is more complex in this world than us?

Moreover, the database section is perhaps one of the most practical sections in a software application. The SQL language itself is so practical and easy to use, exactly in the rhythm described above.

SHOULD WE WRITE IN THE DATABASE IN A CERTAIN WAY?

Now, let's move back to our goal: database development. The students, the young programmers, the application developers with not too much background in the database development, all these categories should be able to understand and deal with the data.

The data is almost everywhere, in every business, and the programmers are dealing with it all the time. In most of the cases, the data is relational. Despite any appearances, the data requires a certain type of understanding and, for that, a different kind of effort is required. The database development is that kind of programming that deals with data in a relational format, rows and columns. We need it, whether we like it or not, and we need to see if our styles are appropriate for this purpose. When I say our styles, I am referring to classic styles that we are familiarized with from the user interface level.

Most of the programmers are following the models they learned in universities. Only that the relational model and the database are somehow different and particular and, in many cases, a different style is required. For example, let's see the reality of a factory or company. This means the sum of processes that are handled in a factory, the multitude of documents that are used in the factory, the different sets of calculations that are done in the production business inside the factory. All these summarize the reality of the factory. We want to build a software application to reflect the reality of this enterprise. For that, we may use different models. Most of the realities contain the segment of data. In most of the cases, relevant business data needs to be stored in a dedicated space, called the database. The list of materials, with the multitude of material attributes, like the type of the products, different classifications, the list of components or bills for materials for the finished products, the list of accountancy documents, company documents and others. Some of these documents may be generated from others. Considering this example of production, it is clear that the segment of data storage is one of the most critical, maybe the most important one. We build nice user interfaces, desktop or web and one of the main goals is to generate a consistent database.

We have the user interface and we have the database. The developer spends time in any of these. He can do his work in the same manner in both sections, at the user interface and at the database level. Alternatively, he can do his work differently when he is involved in the database layer and he can try to use a distinct style in concordance with the nature of data. Any types of the two can be considered as database programming, but my opinion is that the second approach is fundamentally better.

I think that we should write in the database in a certain way. Moreover, the way we should write couldn't be derived directly from the style we are used with from the user interface level and from the two classic models. The fact that this "database way" is particular and is derived from the nature of data does not make the difference and is not a counter argument. The main reason for this paper is to illustrate this way and to promote it.

THE SQL SHOP METAPHOR!

As you are perhaps aware, I am the supporter of the second approach. Try to handle the data component, the database, in its own way and use differently the common models and methods that you are using at the user interface level. When I say, “use differently” I am not referring, of course, to an absurd and total exclusion. We don’t need to forget that SQL itself is a structured language, so it follows the principles of structured programming, although the meaning of structured is not exactly the same. However, there is a big difference in style and this is what I want to explain. This is why I promote and what I want to prove in this book, this is what my experience taught me so many times during my projects. If we follow the second approach, we will start concluding our topic and we will enter more or less in the world of true database developers, we will pass into the SQL area.

I hope you will excuse the apparent deviation from the neutral style of this paper and I want to invite you to step into an imaginary world of metaphors and analogies.

Imagine you fell asleep and you found yourself transported into a fantasy shop. This is the **SQL shop**.

Welcome to my shop! Please take your shoes off!

Try to breath: you will be amazed to see two dominant keywords: **simplicity and naturalness**. This makes a difference.

We are inside the shop now.

Welcome to my SQL shop, I say again. We have many things to see, to visit and eventually to buy. There are so many windows, each of them with so many potential presents. Look, on the left I see some chocolate! There are so many types of chocolate, from different countries. I am going to look. Let’s see what is written on this one! What about the one on the left, I am very curious about it! I like white chocolate, let me look again, please step back and let me see! I will choose some chocolates, I have two children and I eat too much chocolate too! I need to quit soon. Still, now I am in a dream, I can afford to eat tones of chocolate. I will read the names for different types of chocolates: I will **select** different brands and see the different instructions written on the chocolate paper. I will choose some of the brands and **insert** them into the basket.

Oops! I already used, so naturally, some of the most important keywords in SQL: **select**, **insert**. First, I need to have the possibility to read and understand, to select the information before choosing something to buy. I read the instructions on different products in the shop, I am looking at the design of the products, I analyze the utility of each product, and I am looking in my pockets to be reasonable with my budget, of course. This is the select operation, or the read phase. No one is able to write without being able to read first.

When I decide what to buy, I choose the selected products and insert them into my basket. The basket is filled with the **inserted** products. Oops! This inserted keywords sound familiar to some SQL Server developers. Is it similar to a temporary table attached to any insert trigger? Isn’t it fascinating that we are in such a natural topic of a shop, trying to buy something, and we are already in the position to discuss SQL? You will see all over the place in this book why SQL is so valuable: due to the naturalness of the language! SQL is really a part of our live!

The database is like a shop. I will name it the SQL shop. We can assimilate the data in a database with the list of goods in a supermarket, or with the list of books in a library. Whenever we enter a market or a shop, we have the possibility to see, to view, read and understand the utility of the products, the benefits and the disadvantages. This is the select phase. This is the read section. We can also fill our basket with some products, some chosen goods. This might be called the insert phase. We insert some chosen products into the basket. We can also choose, after reflecting and analyzing our budget, to dismiss some products before we pay. In this case, we pick the product and we put it back in the window. We **delete** it from our list with bought products. This delete keyword seems very familiar, isn't it? We may also decide to replace an object with another similar object, and we need to **update** our shopping list again. Everything can change until the moment we arrive to the cash desk and pay. This period of indecision can take some time, until we are completely decided what to buy and end our shopping list. This period of indecision can long even one day if two or three women gather in a huge mall and decide to spend their entire day there watching, shopping and chatting together. It is even possible, at a certain moment, for one of these women to receive an emergency phone call from someone. In this case, they need to leave the shop because an urgent matter occurs and her presence is mandatory. So she **rollbacks** everything, she leaves the basket in the middle of the supermarket and quickly runs to see what happened. In this case, it is possible that one of her mates, or maybe her husband, chooses to take the uncompleted basket and quickly **commit** the wife's shopping list.

Is there someone talking technically here? I don't think so, do you? This is the beauty of SQL! When someone calls us technical, I start to laugh. I do not consider myself a technical person, even if many people that used to work with me considered me so.

Therefore, I am inside this shop, and I have the goal to fill a basket and buy the chosen products. For the shopping period, I look at some of the products, analyze different products, add them into the basket, eventually replace some products with different ones, I ultimately remove some items from the basket. I can do this basket update for some time. Finally, I decide to go and buy the chosen products. I am going to the cash desk and pay the items.

This shop is the database. The items are the rows in different tables, the tables may be considered as the brands. The shopping period is a database transaction. During this period, I have the freedom to add, remove or change items in the basket. Like in the database, I have the option to select, update, insert or delete the rows. After the payment, changes are not permitted anymore to the basket, unless a new shopping session starts. The transaction commit is the payment.

This analogy between a shop and a database, between the list of items in a basket and the list of affected rows in a transaction, between the payment and the transaction commit, this analogy wants to show the simplicity of the database model, and the naturalness of SQL, as we will continue to see along the pages of this book.

AN EXAMPLE OF BAD PRACTICE!

I have met on a project a case that can be written in anyone's memories. I was on a contract where I needed to build a data migration system between two sources of data. There was an existing system of that type and, starting from this, we were required to build a new data migration interface. I analyzed this migration system and I tried to understand what was going on there.

For example, there was a very simple step inside this migration system: some configuration values were required in a certain table. One table was the target for this step, and three or four columns needed to be updated with some configuration, highly static values. This table was configured once and these configuration values will be used all over the place in the corresponding application. These values were not to be changed afterwards. Let me explain you how this step was build! A stored procedure was written for this purpose and this was normal because the configuration values were critical values so the step was very distinct. Nevertheless, looking at the stored procedure, I was forced to spend some time to understand the goal of this step.

What I found was unbelievable! Some structures were declared and used, with some types, one cursor was used and everything was so complicated! When I realized that, the actual goal of this step was simply to add three or four values in a table and add one row with some configuration data I started to laugh! I said to myself that this is not possible. The programmer that wrote that code, I am referring to a specific PL SQL code, was coming from the user interface and he was decided to completely ignore the fact that he was programming in a database! Even more, he was not aware that he was in the process of building a data migration system! He uses absolutely the same style, the same patterns, like in Java or C++ or whatever application language he was used.

Once I realized that the goal of this particular step was actually to add one line in a table, I simply rewrite the procedure completely, and I simply added the insert statement. The entire data migration interface was written in this style, many structures and types, cursors over cursors!

That data migration system was a very good example of a bad practice, a very good picture of what is happening if a row-by-row approach, and a pure procedural style is used in a data migration application. The PL SQL language was used like a typical and classic language. That programmer tried to develop in a pure Object-oriented style!

I can tell you what that was. It was a disaster! First, the performance was at the **minimum** level. That software application had a very poor performance and whatever indexes or whatever performance features were used these were useless due to the **inappropriate style of programming**. When I rebuilt the data migration interface and built the new one, I completely rewrote everything. More than that: everything was rewritten in pure SQL. I had maybe one or two cursors in the entire data migration interface. There were no structures, no arrays and no cursors: just pure SQL!

After, the performance of the new data migration utility was at the **maximum** level and I challenge someone to make it faster in a different way. Even more, the data migration system was very simple for anyone to understand. It made a huge difference!

Many data oriented software applications are not correctly written. The performance is the first one that suffers but also the code is very difficult, more complicated than it should be. When you are in the database and when your task is to manipulate data, you need to think SQL, you need to think in **data sets** and not follow the row-by-row approach. You need to try to understand the relational model and appreciate the clarity of this model and the naturalness of SQL. Some programmers consider that they are too important in what they are doing and they cannot go down to a simpler model and implement things particularly in that way. Others are not effectively aware of this model and of what they can do with SQL, they don't realize that they need to think differently and they are using the same style that they are using in the other sections of the applications.

The database programming languages are just extensions for SQL. PL SQL is a programming language but is an extension for SQL. A good database programmer knows that SQL is the first priority and the extension (the set of procedural statements) is more like a backup solution for SQL. The philosophy is simple: You have a problem: try to solve it in SQL. If this is not possible, go to the extension and use the set of procedural statements. Why using a cursor when you can solve your problem with a simple SQL statement?

Coming back to our shop, we are with our family at the supermarket. Imagine that we want to buy 20 bottles of wine, 30 bottles of water. We will have a party soon! Try to imagine, for a moment, that you will fill the basket 30 times for the wine and 20 times for the water. You will pay 50 times in total. How much will you stay at the queue? Maybe you will stay 50 minutes, adding one minute per product. I believe the cashier will go crazy and will consider me abnormal. Don't you think so? Who could have this crazy idea in a supermarket? A normal person will fill the basket once and will pay the set of 20 bottles of wine and the second set of 30 bottles of water. He will try to fill one basket and, if it will not be possible, he will fill it 2 times and pay two times. Anyway, there will be quite a difference in terms of waste of time. This is what is happening when developers, good developers, coming from the general models of the user interface and coming from general and complete programming languages like Java and C are trying to use their own style in SQL.

Now you imagine that you have an update statement and you will affect 20 products in the update instruction. There is a quite a number of programmers that would consider normal the fact that the rows are affected one by one, in a cursor or not: this is not important at this moment. Instead of thinking that they do have a set of rows that needs to be affected and try to affect that set of rows in a simple SQL statement they start declaring structures, variables, eventually try to use complex objects only because they cannot or they don't want to think SQL!

Chapter 3

THE HOLISTIC VISION AGAINST THE DATA

THE CONCEPT OF DATA SET

I will continue to analyze and define the rest of the concepts. Some of these I already remind during the previous chapters without explicitly discuss about them yet. I go on and try to explain another notion, maybe the most important one that supports the style of development I am promoting!

There is a concept behind this style and I don’t claim this concept is new!

I know that the market starts to become increasingly more aware of this concept and ask for it, consequently for the style I am promoting. The concept that defines the promoted style of development is the concept of **data set**.

Firstly, the concept is related to SQL language that is mainly a query language. The purpose of the language is straightforward, data access.

Secondly, data in a relational database means a mixture of rows and columns or expressions, in a simplistic vision but quite accurate. In most of the cases when accessing data, we are committed to identify a **combination of rows and columns or expressions**, and that is exactly the definition of a **data set**.

If there is something that can be defined as a permanent thing to be noticed when dealing with data and database programming, that something is a set of data. There is always a data set; there is always a combination of rows and columns or expressions whenever we are dealing with data. “Cherchez la femme” says Alexandre Dumas and I believe we can exclaim similarly, whenever we are doing database development: **search for the data set!** All the mysteries and solutions of database development rely in the proper treatment and fair recognition of the data set.

The data sets can be taken from a variety of tables linked between them by joins, the data sets can be combined between them by set operators and the data sets can be identified in a variety of ways that are not relevant for the moment. The SQL developer is the magician that knows how to get the best from the data set.

The **set-based approach** means to always be aware that, when programming the database, data sets should be affected and the main concern in the development process is the data set.

There is an enormous potential of situations and millions of circumstances and cases can be imagined.

Many examples will follow and I will try to illustrate the **set-based approach** as well as I can. Don’t try to imagine miracles and don’t try to imagine complicated things!

Most of the practices that will follow will illustrate this major characteristic of database programming and SQL. Because database programming means firstly SQL development and SQL are the bread and the butter of any vendor programming language, either PL SQL or anything else! The main activity in the database is the process of query. We are querying the data continuously because this is the essence of database programming. We always try to get a combination of rows and columns or expressions taken from various tables, we always try to get a set of data. This is the secret of SQL, and more than that, this is the secret of database programming: try to think in **sets** of data and do not think atomically, whenever it’s possible. Do not try to divide the data set into smaller units unless you have an authentic business reason and a technical limitation that stops you from affecting the data set as a whole.

Let’s try to have the following vision against the data: see the data as an integer and not as a sum of decimals. If the data set corresponds to one row: this is as a particular situation. Always try to find the set of data, try to think in waves of data like when you are looking at the sea and you see the waves coming one after the other.

When you see a wave coming from the sea you never imagine any division, you just see the wave and admire it! Looking closely you will see many things inside it and you can divide the wave into smaller divisions according to various classifications. However, initially you don’t care about that! Try that with the data set and you will think like a true database developer!

THE HOLISTIC APPROACH VERSUS THE ATOMIC APPROACH – AN INTRODUCTION

I am not a theoretician but I admire theoreticians! They have such a great mind and they are capable of things beyond ordinary people like me. The theoreticians are able to build foundations. These foundations are the base for human creations everywhere. Software programming is a new world, at the history scale. It is a child, a growing one! It is already so much history in software development that we can spend ages to see all of it and try to understand what was happening here during some decades of activity. The idea of programming came from sciences like math and logics and it became so common these days that people are not amazed by it anymore. There are so many artificial programming languages on the market today and the list continues to expand!

Every programming language is a miracle, in one way or another. In addition, the programmers that have the possibility to write pieces of software are like artists and linguists, in a certain way, not just pure technicians. A good programmer may be a technical person but may be an artist too!

Every new programming language that we learn to use is like a new natural language that we learn to speak, in a certain way. Some languages are simpler to learn than others are. In a similar fashion, some programming languages may be easier to be taught than others may. It is a matter of subjectivity and experience, a matter of taste too. Some languages and frameworks can be learned by analogy and comparison.

The principles are similar in many programming languages and this makes our lives easier than multi-lingual's…

Every programming language has its own principles and the programmers try to understand and follow these during their daily activity. Speaking about the set of database programming languages, like PL SQL or Transact SQL, there is another great advantage, the portability.

This is the advantage of the SQL language. This query language is included in every programming language and, even if there are some differences in the syntaxes, learning one makes us eligible for learning another one quite easy.

The procedural facilities are also similar in the syntax, declaring and using a cursor in Oracle and SQL Server or DB2 is not a big difference, you don't need to consider yourself PL SQL developer and to remain in the area of PL SQL necessarily.

Apart from that: in these specific programming languages, there is a principle that should be followed: the principle of **data set**. If there is something to be explained as a principle of work in a dedicated database programming language, it is the **set of data**. A programmer who writes to the database, regardless of the area where it comes from, must realize the need to identify the data set in everything he does.

Let's assume that we are running a query now. We just saw the familiar message about the number of rows affected; let's say we receive five rows. Five rows affected may be interpreted in many ways. We may think at five details, try to analyze the details, and think to affect detail by detail in a certain way. Alternatively, we may think the unity of the five in one, like an integer.

The set of five rows can be seen as a whole, or can be seen as a sum of details. I can see this number of rows affected in its multiplicity: I can see this data as a kind of union composed of five rows. The application developer is tempted to see the multiplicity and is tempted to divide the set into five pieces. Of course, he will be happy to open a cursor right away. By contrast, the database developer will always see the data set, he will be aware of it and he will think SQL. This means he will try to see the data set and ignore the fact that the data set can be divided into five pieces. The application developer is not SQL oriented by default, he is oriented to search for the details, his models are not set oriented and he is trying, by default, to divide and to use the procedural features of the languages.

SQL is data set oriented, SQL is a database specific language and SQL is not incorporated in the paradigm of the classic models of programming that the application developer knows and applies in his daily work. The application developer should make an effort and he needs to try to integrate SQL in his style, he should try to be aware of the data set. The application developer should do that if he wants to have a good performance in the database and especially if he is in the position to write in specific interfaces where his intended goal is to transfer data between systems.

Considering these circumstances, I will define two approaches that can be used in database programming. These are illustrated by the role of the set of data in the process of database development. These approaches are the ones that map the development styles and one developer or another should use a certain style according to the approach he follows.

The first is the **atomic** approach. By contrast, I will name its opposite the **holistic** approach. These are the possible visions against the data in database programming. Consequently, the styles of development will be highly influenced by these. The programmer that follows the atomic vision is the programmer that thinks atomically, the programmer that does not think SQL. If you try to think in a non-SQL fashion means, you will not accept the data set as the main entity that should be firstly identified and handled by the development process. The atomic vision means to divide everything and to see things atomically, especially at the row level. The developer does not accept anything apart from what he knows, he knows structured programming, he knows scalar functions and maybe row triggers and he sees columns per individual rows as parameters for his functions and procedures. He is very happy with the cursor facility and he believes that the purpose of a cursor is to allow him to move everything at the level of a row and scalar variable so it will allow him to read and write almost everything per row.

The holistic approach is the opposite approach. The programmer is aware of another kind of entity apart from what he knows from classic programming: he knows that SQL is something else and he understand that the data set is an integer, a unity and he thinks holistically.

One has a data set and he wants to affect that set with any possible means. He has SQL as the tool to accomplish that. As a last resource, seeing that SQL does not allow him to do his task and to affect the data set holistically, he will divide everything using the cursor and he will solve his problem in a different way.

The two styles of development are the ones associated with the two visions: the **holistic** vision respectively the **atomic** one.

I will try to show and prove that a set of data should to be analyzed in its unity, atomicity, and needs not to be analyzed in its multiplicity. In most of the situations, the data set can be identified as such and affected in its unity. The data set should be targeted and, finally, affected by our actions, write or read. Most of the time when working in the database, in a classic but mostly in a specific application, we are affecting data sets in a continuous process. We should try to be aware of the data set; we should always think at it and try to do not divide it into smaller pieces or rows, whenever is possible.

Let's remember the famous concept of "entity", so dear to application developers. In classic programming, everything is more or less an entity or part of an entity. The entity is a very complex concept and the programmers are always aware of the belongings to a certain entity when they are doing their staff. In the database, the role of the entity is somehow taken by the data set. The data set is the entity and in a relational database, we are usually positioned in a certain data set.

This is the holistic approach and the developer that thinks holistically is the true SQL developer. He is the true database developer. In addition, this is that kind of transformation that an application developer should be capable of when switching to the database. I believe that a good programmer can be both and can do things at a good level in both application and database. In addition, I believe that the programmer should try to think differently and should try to adapt his style when moving to the database.

Another purpose of this book is to explain the main principle that anyone should have when tries to think SQL. This is mostly required for any person that is in the position to write some pieces of code into the database. He needs to try to think SQL! Because the dedicated language for communicating with the data is SQL, its syntaxes are perfect for interrogating rows and columns, and in most of the situations when we are within the database and we are doing programming, we are simply accessing the data in one way or another. I want to read something or I want to write, I use SQL. That's why I was surprised when I heard database developers considering themselves PL SQL developers or PG SQL developers. To me, a true database developer is a SQL developer. Moreover, a SQL developer, a good one, needs to be able to think SQL. Thinking SQL means thinking in sets of data. I cannot imagine anyone saying he thinks PL SQL or Transact SQL! No one can actually imagine that! Still, it should be obvious to anyone what thinking SQL means.

I will continue to insist on the advantages of the SQL language. More than that, the type of actions that we are generally doing in the database is restricted to some basic statements. Let's review them. We are reading some data using a **select** statement, we are adding something using an **insert** statement, we are editing something using an **update** statement or we are deleting something using a **delete** statement. This is what database programming is, this is pretty much it. In almost all the situations, we want to affect a set of rows, one single set. A good SQL developer is always aware of the data set: he is trying to search for the data set. This is the holistic vision: this means thinking SQL and this is true database development! Only when the programmer sees that the problem he needs to solve cannot be done by affecting the entire data set in one SQL statement, whatever complicated this one might be, the programmer will open a cursor, will move the details in variables and do his logic. Only in the situations, very rare I dare to say, when the data set cannot be handled as a **whole**, the programmer will split the data set after its **details** and, in this case, he will use the procedural facilities.

The database programmer is not a limited programmer that does not accept anything else apart from the data set and his precious SQL, like his precious ring! He is not absurd! The database developer knows the principles of structured programming: he knows how to use cursors, records, variables, while statements and the rest of procedural facilities. He is able to solve the problem even if the data set vision does not match all the time. The data set is not indestructible! It can always be divided! Sometimes the data set must be splatted into many pieces. In certain situations, the set of data must be divided into smaller components, even data rows. The holistic vision means accepting that these situations might occur and the holistic vision accepts the possibility of the division, whenever it's necessary. Still, this division will occur, in the holistic approach, only after everything was tried for a holistic solution.

The atomic path means the tendency to divide things right away and to try to solve things procedurally by default. No attempt is made generally for the data set to be handled and the data set is not actually seen as a unity. The atomic vision is very common among many application developers and this is what I try to show here, that this vision should be reconsidered.

A **holistic** vision against the data that will generate a certain distinct style of development in the database and this is highly recommended, in my opinion. This **holistic** vision is reflected in a concept, the well-known concept of **data set**. Having that holistic vision means adopting the set of data as a primary criteria of development. We are developing our logic in the databases in data sets. First, we should identify the data set. After that, we analyze and see if it can be affected in one single SQL statement. If that is possible, and very often it is, SQL is incredible strong and it covers an amazing number of situations, the problem can be solved very often at the level of the entire set. This is the holistic vision and the set prioritization. This is the **set-oriented model of database development**.

A DIFFERENT MODEL –DOES THIS MODEL DESERVE TO BE PROMOTED INDEED?

The education systems are an important component in the IT Industry. The universities teach the students about classes, entities, objects, and everything else! The Object-oriented models, the declarative programming, structured programming paradigms are described in a variety of courses. There are many models and paradigms and each of them should be taught to the future generations of programmers. These principles and models will guide the future programmers in their activity and will influence their work and our lives.

Let’s move back to the database area! It is considered that database programming is part of structured programming. Apart from typical programming and the classic models, and among other things, there are courses about databases where relational databases are explained. The SQL language is on the top of the list. Sometimes SQL is explained by itself in a dedicated course or it is described as part of a vendor programming language like PL SQL. Programing languages as PL SQL can be described in two steps, first the query language and then the extension with all the procedural facilities.

I believe that in the universities, there are not enough discussions about the styles of programming and this is understandable. It is a vague concept and it involves a certain degree of subjectivity. Still, no one can argue that these various styles of development are not important and that our applications are not influenced by this vague concept of style. Only that, because is a subjective matter, it is ignored. These styles are determined by the programming concepts and models, among other things. Among the premises for these styles of development, there are scientific concepts like models and paradigms.

The style of development is influenced by the subjectivity of the programmer too. Some programmers are Java based and they are comfortable with Java, others they do like “INFORMATICA” and visual development and they are comfortable with it. Still, the subjectivity of the programmer is influenced by the concepts. There is a vicious circle, therefore.

Regarding our interest, I believe that, when talking about databases in the universities, the concept of **data set** should be properly described as one of the main concepts in database programming, maybe the most important one. However, in most of the cases, the discussions about a different concept like the set of data that stays behind a certain style of programming are not considered as vital, assuming these are accepted. I believe that the concept is well known but is not considered as vital, as it should be!

I consider that the paradigm of **set-oriented programming** should be promoted in the database courses in universities and explained with priority to the students. Many of them will work on both sides during their careers, they will work at the application level using Java or C# but they will also work at the database level using Oracle or DB2. If familiarized with the data set and the holistic approach, it will be easier for them to adapt to one style to another. Even more, some of them will really want to specialize in database development. For these students, the understanding of the importance of the data set in their development is critical.

For example, let’s pick up a university where databases are supported and are considered as a valuable path for the students. In this university, the database programming courses should contain a second part, after the classic SQL course, where the capabilities of SQL are explained. In this second part, I would discuss about database development and the importance of the data set and the holistic vision of development. I would try to explain that a different style should be used in the database according to the set-based approach and the holistic vision against the data.

Apart from the universities, I believe that these discussions are very important inside software development teams too. There are often mixed teams of developers, application and database developers. Secondly, there are always personal preferences. The software applications written by the developers are influenced by these preferences and this is a reality too.

I believe that discussions about one style of development or another might be very useful inside development teams. The performance of the software applications will be better because a more appropriate style in any section will definitely increase the value and the quality of the written software. The database is always behind, but the end user feels it all the time because of the response timings.

What is more important nowadays apart from the fair implementation of the business? Name something more important than **performance**! Besides, a good performance in the database is firstly done by an appropriate style, a **set-oriented style**, based on a **holistic** vision against the data.

Let’s move from the application developer to the specialized database developer and its role in the market. I believe that his role will increase and I hope I am not too subjective in this statement! After many years of indetermination, I would say that finally, the database developer has its own status, without being in the shadow of a DBA or a developer, that knows Java or C and, among others, he knows some SQL enough to manage himself. You will find more and more specialized database developers and the good ones are always aware of the data set, of the set-oriented style of development that they are definitely use!

Still, many people consider that database programming is not necessarily a distinct path and it does not require a distinct style. Many people believe that an application developer can do SQL easily without any need for any change in his style of work. There are no official arguments for a different style and there are so many programmers doing everything, both application and the database. Even more, there are so many programmers doing everything in the same manner! In a classic system, things may be acceptable even with a procedural, atomic style. This might be true especially if the activity is at the lowest level. Still, who knows how and what is the performance at both levels? Even more, in a **specific** interface like a replication system where the goal is to move data between systems, if the developers are working atomically is a tragedy, the most catastrophic scenario that we can imagine.

Unfortunately, sometimes SQL is seen as a kind of toy, as a kind of light language that can be learned by anyone with a minimal effort. To a certain degree, it is a toy, and it is true that the basics can be accomplished by anyone; the language is extremely natural and intuitive. Still, that is the first stage of learning. Try to imagine that you have a course separated in many modules where the first module is very easy and accessible. The followings are more and more difficult and finally, after graduation, there is no difference against another course where the first module was more difficult and not so accessible. This is SQL.

From this perspective, the concept of **data set** makes a difference in graduating the next modules. A programmer that understands this concept and tries to follow it is a good database developer. For example, take an application developer that is doing his mixture in a classic application and he is doing the switch from the application to the database. If he will be capable to change his style and he will be able to adapt his work from one section to another, he will have only advantages from his work. I really believe that the set-based approach and the holistic style of development deserve to be promoted more intensively, considering the advantages of a much better performance, easier ways to debug, code that is more intelligible and many others.

We will analyze one more example. We assume we want to see the list of customers that satisfy certain conditions, for example customers from London, UK. If we want to think SQL, we should think that we have a set of customers. This is the SQL vision! We do not consider the customers in their multiplicity but we consider the list of customer as one list. This vision is opposed to the atomic vision according to which we consider the customers one by one and try to affect these atomically, customer by customer. This is what I consider as being the principle of SQL. We consider the multiplicity as unity, like when we have a book with one hundred pages we are not thinking at the pages but at the book, we have one book not hundred pages.

Inside the projects, the project managers, the technical leaders have their important role. Considering various styles of development in various sections can be a good decision for them if they really want to have a good performance and good quality in the code written in their teams. Of course, it is difficult to say that the programmers should be convinced to adopt a certain style or another. I don’t want to judge, it is absurd. I still consider that, participating to the discussions with the software programmers and trying to make sure that everyone will try to write in a certain way that the performance will be optimal is a thing that can be done by the leaders of the projects.

Consequently, I believe that the set-oriented model and the holistic vision deserve to be promoted more in the universities and, even more, inside the software companies. A large portion of our code relies on the databases: the vendors continuously try to add new features for the performance section. Performance is always critical. What can be more unpleasant than waiting and waiting for the data to be accessible to the end users? Moreover, one reason for a bad performance in the database, one big reason, is the fact that the style of development is not adapted to the realities of the databases. Despite the fact that SQL is such a trivial language, and it is at least the basics, the data set is not so clearly seen as a goal to be reached by the developers.

PERFORMANCE AND PORTABILITY - TWO ADVANTAGES FOR THE HOLISTIC APPROACH.

I know that most of the programmers are very practical persons and they want to see written code. I understand them and I respect their desire. Soon this will follow! I will try to illustrate the concepts described above and the two alternative programming styles by showing many examples as I can. I will use two of the most important database systems, Oracle and Microsoft SQL Server. I choose these two database management systems for my examples for two simple reasons. First, they are among the most popular database management systems. Secondly, my experience is related to these two systems more than to others. Most of my projects were related to Oracle and SQL Server, but not exclusively. I have experience in classic applications and in specific applications like data warehouse, replication and data migration. These are my favorite types of projects and, working in these projects, I was able to work at my style and adapt it more and more to the **set-oriented approach**.

Nevertheless, the good news is that the examples are very easy to reproduce in any other database management system. Here I will show you the advantage of the standard and the advantage of the fact that most of the SQL statements are almost the same, identical or similar in almost all the relational database systems. This advantage will allow any developer that wants to try the exercises and practice in his own relational system: like Oracle, SQL Server, DB2, PostgreSQL, MySQL, and Teradata! He can practice most of the exercises with minor modifications.

This is another argument in the favor of the holistic approach although I do not consider it as the most important one. The argument is **portability**. It is good for companies building software applications for various database management systems for different customers: portability is an attractive word for the managers. If you are in a software company, where one application is written in three, four systems like Oracle, SQL Server, My SQL, and then portability might be important. Writing SQL allows you to keep the code similar from one system to another. Working procedurally means accepting very different pieces of codes because, even if the procedural languages are quite similar, there are still different syntaxes and it takes some time to translate the code from one system to another!

The examples in the atomic approach are taken from the procedural area of the systems. These are highly particular and non-standard, being non-SQL and procedural. Working atomically forces you to move your center of interest in the procedural area of the database language. In the atomic way, the logic is procedural and specific to the programming language, which materializes in a serious effort in the translation process, if required.

Working SQL moves you to the SQL area, close to the standard and to the true database vision. Therefore, you will see that, if you will try to reproduce some examples from one system to another it will be much easier. Sometimes it may be even very easy, sometimes the logic and the syntax will be identical.

I am discussing about SQL, I am trying to show how to think SQL. This makes us more independent on the database system. In addition, if we are working in an IT company that creates software, it is very easy for us to maintain various logics in different systems if we are working SQL based and not procedurally based. Still, I know there are many database programmers having exclusive experience in one database system. Some of them believe they are Oracle or PL SQL developers, maybe SQL Server, or Transact SQL developers. Actually, these should be considered database developers and any switch from one database system to another should be relatively accessible. Moving from Oracle to SQL Server or from SQL Server to Oracle or to another database system like DB2 should not be a difficult task. Having SQL as a standard makes our life easier and I encourage any developer of this type to consider itself a neutral, database developer, or even more a SQL developer. The programming languages are similar so any movement from PL SQL to Transact SQL or SQL PL is quite accessible.

Even from the procedural facilities, things are not very difficult. Most of the examples can be easily migrated and executed in any relational database system. We have the same types of programming objects, we have similar structures of blocks inside the objects; cursors are declared and used in similar ways. In the field of programming, the database developers have a huge advantage. For a database programmer, one experience in any database system is a gate to any other database system and the database programmers need to have more confidence in themselves regarding their ability. For example, a large category of experienced SQL Server programmers, are not confident to switch to Oracle projects. Of course, an initial effort is required to understand the differences but this is not that difficult. In the beginning, the programmer is a bit slower and he needs an initial period of accommodation with the new database system. However, the change is acceptable and the developer can adapt quite easily to the new database system. The switch is especially straight forward if the style of the programmer is SQL oriented and not procedural oriented. A lot of code is pretty much the same. Of course, we do not need to exaggerate in anything. The portability that SQL gives us is a great advantage. Writing mainly SQL and being in concordance with the standard allows us to write very similarly in different database systems and this is a great advantage!

Still, sometimes, for the sake of portability we may use some syntax that is not very good from the performance point of view. Even if we do write SQL, there are many ways of writing SQL too. There are various SQL syntaxes and some can be more standard and others more specific. Sometimes a specific SQL syntax has a better performance. A good example is the SQL Server particular form of update. This syntax has a better performance and, despite the fact that is particular to SQL Server, it is better to be used. A compromise between performance and portability is required. I always vote for performance!

I want to conclude the discussions about portability and performance in database development and share my opinions gathered mostly from years after years of development inside various databases. Portability means firstly to write SQL and not procedural, due to the advantages of the fact that SQL is a standard and all the vendors have similar syntaxes. So the first rule, to try to write SQL and not procedural, is a golden rule for both performance and portability. The second conclusion is referring to the SQL code itself. The code is similar, sometimes identical in various database systems. When the code is similar, there might be many acceptable syntaxes, some of them similar others identical. Sometimes you may try to use the specific syntaxes and avoid the identical ones because very often the specific syntaxes have better performance. A good example is the specific form of update and delete in SQL Server. That one does not work in Oracle. Still, I encourage you to use it and leave the syntax with subqueries because the performance for the specific “DML” is better. Therefore, my advice for you is this one: choose performance in front of portability. Anyway, compromises are usually required.

VISUAL DEVELOPMENT VERSUS SQL DEVELOPMENT

About databases, we are dealing with data and we are analyzing the two styles of development. We analyze the atomic approach versus the holistic approach.

We analyze the application developer non-aware of the concept of set of data that thinks atomically against the SQL developer aware of the fact that he needs to think in data sets and holistically. Both of them are, in a certain way, classic or typical programmers. They are using a programming language, either a general one like Java or C#, or a specific one like PL SQL or Transact SQL. However, they are using a language. They develop classically, they write code: they are writers!

Apart from classic programming, where the developer writes his code, there is a new generation of programmers, I would call them visual developers. This new category of programmers is becoming more and more appreciated on the market. To remain in the field of databases, good examples can be taken from data warehousing. I am especially thinking at the “ETL” process, the process of extracting, transforming and loading the data from a set of operational system to a large data warehouse system, a historical database. There are many examples of “ETL” visual tools, like “INFORMATICA” or “Oracle Warehouse Builder”, or “Microsoft Integration Services”. An “ETL” is a very complicated process where data needs to be integrated from various operational systems into a historical database, with the purpose of analysis and prediction. An ‘ETL” developer nowadays is sometimes a visual developer and not a classic developer. Of course the best ones are both because, even for a visual developer, when things are not going well he needs to debug and he is going to the sources behind the visual tool. Therefore, there is this alternative too, especially when speaking about specific interfaces like a replication system, an “ETL” or a data migration system, whenever we are discussing the transfer of data between systems, a visual tool might be a solution. Very often, managers choose these types of solutions.

We are slowly moving to a mixed world. The future world will also be the world of tools: this seems to be the tendency. Consequently, the future world will be a world of visual developers, too. To be a good specialist today does not necessarily mean to be a good classic programmer of a certain type, like a C# programmer for example. You can be a very good visual programmer. Still these days companies add a lot of variety in the fields of specializations and now we can see, looking at the list of projects and opportunities, the new concept of visual specialist, visual developer. Some companies are not looking for a database specialist anymore but for a tool one. We have so many tools in the area that we can find ourselves a bit overwhelmed, if I can say that.

For example, I saw descriptions for specific projects where various companies were asking for experienced specialists with many years of experience in “INFORMATICA” or “Oracle Warehouse Builder”. Moving data between various systems, either between two operational systems or like in a data warehouse, being a database developer doesn’t seem to be the best solution anymore.

I have some experience in visual development but I do not intend to move to this area, an attractive area for many people. First, I like SQL too much. This is subjective.

I also consider that, very often, choosing a tool instead of a custom solution with

SQL is not a good decision. I affirm that with the SQL language, you can do miracles and very often, the use of tools could be avoided and replaced with pure SQL systems. I don’t believe you can build a better replication system or data migration system with a tool than with pure SQL, in most of the cases. Of course, the visual developers will disagree and it is their right to do so, I am sure they have solid arguments too. Not being a visual developer like them, I respect their arguments and their options. Anyway, my experience with visual tools showed me that the argument of time is not necessary valid. I am not certain if the development time with a visual tool is indeed better than the development time with a classic, non-visual approach. Moreover, sometimes when you have problems in a visual tool is almost impossible to move forward!

It is clear that there are various alternatives on the market and this is great. Companies can choose. There is more and more demand for the **specific** type of software application, with the clear goal of data movement between existing systems. Companies are moving their data continuously: it is a huge demand for this task almost everywhere and companies try to find better solutions to satisfy their goals. Sometimes they are not satisfied with true database specialists and classic database development and they prefer to move to tools and visual development instead. It is a free market and there are so many ways of solving things. Personally, I strongly believe that, very often, the best solution is SQL. A good solution for a replication system or data migration system is very often a pure SQL system. Nevertheless, of course, tools are an alternative and nowadays many companies prefer to use them instead of traditional solution like a SQL based solution.

I am on the SQL side! I consider that, very often, a task can be achieved with simple and pure database programming, with this trivial and simple SQL and using ordinary database development tools like Oracle SQL Developer or Microsoft SQL Server Management Studio, even in the “ETL” area, which is maybe the most complex case of pure database system. I have built a very complex replication, data migration system similar with an “ETL” and I did it completely in pure SQL, using something like 5% of procedural code. In addition, it works great, I know what is there and I understand everything, I have the perspective of a whole and of the details. People after me can look and understand all the migration steps. This task, of moving data between different systems, can be done in most of the cases with pure SQL. For that, we need to have the proper developers to do that.

Without considering myself an evaluator of the labor market, I can say my opinion as a contractor with experience and as a database person that I am: it is a bit of danger. A person working in a certain tool for five years may become completely dependent on that tool. This is one thing. The second thing is the fact that most of these tools are visual. For example, in the ETL area, almost everything is done within the complex visual diagrams. Many people are not aware of almost anything apart from their diagrams. Maybe I am wrong and I underestimate the significance these people have and maybe I am subjective in considering the SQL and the database importance as critical for any people dealing with data, but this is my feeling: that many of these tool specialists in these products have a limited understanding.

I don’t mention the fact that, using the database knowledge of the true specialists and going back to SQL and traditional methods, there will be also others advantages.

Maybe this tendency is just a fashion and it will pass. Maybe is not just a tendency! The future will decide!

Anyway, using a custom solution with authentic database programmers is possible and if the complexity allows us, it gives the company the advantage of not being dependent of the tool and of the tools specialists, provides the company with the possibility to have a product and to have the understanding of its own work. In most of the cases, when we have this goal of moving data between sources to targets, we can do it using traditional SQL. It is the simplest way and gives us the complete understanding of what we are doing. There are situations where the difficulty of the replication is so complex that a tool is better than a custom solution, this is for sure. What I am saying is that, in many cases, the solutions with the tools are more expensive and less efficient than the traditional solution with the use of SQL and the use of database programmers.

Speaking about tools, the fact that visual development is easier than classic development is an illusion. I can say that learning a visual tool may be even harder than learning a programming language, to a certain extent. The reason for visual tools is speed and efficiency. Things can be done faster and efficient with visual tools. The visual developer works fast comparing with the classic developer. At the first view, this seems true. Allow me to express my doubts and to affirm that, in many cases, using a pure SQL solution for a specific interface can be faster, with a better performance, a better control and with more understanding of the system.

Chapter 4

WHAT TO CHOOSE: THE DATA SET OR THE DATA ROW?

CHOOSING THE LEVEL OF DETAIL: THE SET VERSUS THE ROW!

This book is divided in two parts, the first part describes the concepts and the second part tries to illustrate these concepts with practical examples. We are prepared to start soon a set of practices that will illustrate all the concepts described in the first four chapters. With these practices, I want to show that, very often, the same task within a database can be done either using the **atomic** style of programming, a common style used by many application developers, or using the **holistic** approach for data access, the true SQL style of programming, style that is specific to authentic database programmers, to SQL programmers. Unbelievably, you will also see that the second style of programming I am promoting here: is much simpler and accurate. The code is much simpler, the performance is better; the code is highly portable from one database system to another. Actually, I am not able to see any advantage of the atomic approach versus the holistic approach. To be clear, all these considerations are available in the specific context of the database development.

Besides, to be even clearer, I want to explain one more thing. When I am referring to true database developers and authentic database development approach, I do not want to say that this style is an exclusive style that should be used exclusively by specialized database developers. This style is very accurate and, with some efforts, it could be accommodated by application developers while working in the database relatively easy. They don't need to change their style completely; they will obviously continue to write in the same manner at the application level. Only that, when moving from the user interface to the database, they should adjust their style and think holistically and SQL, as much as they can. A database developer is doing that somehow natively because he is a kind of mercenary of the **set-based approach**, he does not follow the atomic approach unless is required, he finds this as totally inadequate, inefficient, counterintuitive, against the common sense, choose whatever word you prefer to specify an inappropriate style. For the database developer, the set-oriented approach is the obvious way of doing things. This approach is not so clear for the application developer and, despite the differences, he needs to spend some time to understand and accommodate this new approach. However, this is not a very difficult task and, as soon as will understand it, will be infinitely more efficient in the database. All these considerations above are available for **classic** applications, where both approaches are acceptable in the database.

I want to mention again the specific data migration applications, where the **goal is to move data between various database systems**. In these **specific** software systems, the atomic approach is completely forbidden and the application developers should stay away from these applications if they do not want to change something in the way they write code!

One of the main tasks for anyone in a database is the ability to query and to return the required information. That is why a query language like SQL was invented: this language is dedicated for that purpose. This is a natural language and its naturalness derives from the limited purpose of the language. We will never compare SQL with a classic programming language! The first reason for that is the one of different natures. The second reason is the one of degree of generality. A query language like SQL is highly particular and applies to the data organized in a relational manner. That is why, basically, SQL is very easy. It is easy to learn SQL; it is not so easy to understand it and to use it properly.

As we already mentioned, SQL itself is never on its own. The set of SQL instructions is always embedded in a programming language. We are using PL SQL for Oracle or Transact SQL for SQL Server or whatever programming language for whatever database management system. There is a large variety of relational database systems with their associated languages and Oracle and SQL Server are just the examples that I prefer to use. This is a book about database development and SQL, and all the judgements, reflections and thoughts are available in any relational database system. All these programming languages contain two types of instructions: SQL statements and procedural statements. There is an alternative for any programmer. Our logic can rely more on SQL statements or can rely more on procedural statements. We can think more procedurally or we can think more SQL. In most of the cases, the procedural way is associated with the atomic approach. Still, sometimes, programmers can work holistically and procedurally, as we will see in some examples later during the book.

There are many levels of details in our logic when we are writing code in the database. There is always the **lowest** level of detail, the **row** itself. Still, in most of the cases, a **superior** level of detail should be present, the **data set** to be affected. Very often, the data is identified and updated in data sets. In most of the cases, we are in the position to choose: what level to pick up. The traditional models don’t know about the data set because this concept is particular to relational databases. So, the application developer coming from Java or C, familiarized with his traditional models, should become aware of the data set, otherwise he will be tempted to remain at the lowest level of details, the **data row**. The application developer will start opening cursors for every action, declaring variables and using records to supply simple insert statements. By contrast, a database developer is able to think SQL. A database developer knows that he needs to identify the **data set** and will rely on SQL statements instead of procedural statements to make the identification because will choose the level of detail the data set and not the data row. The database developer knows that the SQL language is a **set-oriented language** and knows that this is what should be used in most of his development activity.

For any programmer working in the database and trying to manage the data either by reading something or by writing information, a decision should always be taken. How should I write my logic? What is the starting point, what is the concept that stays behind the scene in this strange and simple world of rows and columns! The application developer, used with his objects and entities, needs now to understand the simpler concept of rows and columns and the concept of data set. He generally understands the concepts of rows and columns but he is not always aware of the data set. Very often, he sees that the procedural facilities available in the dedicated programming languages for data access are applicable to the lowest level of detail. For example a scalar function, a row trigger, even loop and while statements he finds these excellent for data access at the lowest level of detail, that is the row itself. Consequently, the application developer believes that he always needs to try to move the logic to the lowest level of detail, to move to the row. He believes that's normal for the code to be written and applicable to the lowest level of detail. This is a bad decision in my opinion and one reason for a poor performance in many databases. Writing database programming, writing data access logic in this manner when the principle is to move the logic by default to the lowest level of detail and to use the atomic style is the worst programming someone can write in the database!

Do you know what's funny? That this code may look very professional and attractive for classic programmers, all the principles of structured programming are satisfied and this code may look like a piece of art! This shows once more the subjectivity of the concept of style of programming!

PERFORMANCE IS POOR. PERFORMANCE COMPLETELY BLOCKED IN ANY TENTATIVE OF IMPROVEMENT.

The statements above might be true but the effects are dramatic and the consequences are terrible for the database. First, the one that suffers the most is the **performance**. The performance is a disaster. Let’s imagine large data sets affected in cursors all the time, one by one per row. Even if not large data sets, because database programming is not just data warehouse and very large data sets, we still might have issues with performance. If the dimension of the data is moderate and not very large, what happens to our databases?

Do you know what the tragedy is?

Some companies implement large data warehouse systems and deal with large data sets. These companies hire specialized database developers and these developers know how to write set oriented code. In a data warehouse, the style is generally the proper one. This is not happening all the time, but still very often.

If someone tries to use the atomic approach in an “ETL”, for example, the “ETL” will simply become almost unusable and the consequences will be detected immediately. Therefore, in a data warehouse it is less probable to find these inappropriate styles of development, oriented atomically per the lowest level of detail, the row. Still, is still happening very often in the transactional, operational systems, in normal production systems where people cannot afford or not consider necessary to hire specialized database developers and everyone is using application developers with some SQL knowledge. Of course, everyone knows that SQL is very simple and can be learned right away, by anyone!

How will the application programmer learn and use the SQL language? In most of the cases, he will learn by analogy with his application development! So what will be his style? It is probable that will use a style similar with the atomic style of development.

You see the logic of some IT professionals! Everyone can learn SQL, it’s simple, you already know Java or C#, comparing to these SQL is a piece of cake, isn’t it? In these systems, the code can be written and it is written very often at the lowest level of detail. Due to the moderate dimensions of the data, the logic will not always be detected as inappropriate and the application will survive as is written. The bad performance will not be so visible; the timings will be bad but not very bad. There are so many databases written by application developers in the same style like the user interface, and one main reason for performance issue is the style of development.

Choosing the lowest level of detail - the data row and using the atomic style has another disadvantage apart from the poor performance. Any tentative for improvement is effectively blocked, the performance will remain very low due to the use of atomic style.

What are you doing when you have a low performance? You try to improve it, of course. One of the most challenging tasks for a developer is the one of performance, when the programmer is instructed to try to improve the performance in one section or another. I was in the position to do that many times. Do you know what I did, in most of the cases? I **rewrote the entire logic**! The reason for low performance is the use of the atomic approach following to the wrong decision of dealing with the lowest level of detail. If the programmer chooses to write his code at the row level by default, this code simply cannot be optimized. All the nice features for performance, available in any database system, like indexes, materialized views and others; all these features are useless if the code is written in the atomic style.

To conclude, choosing to develop software at the row level and using the atomic style of development in the database has two major consequences. The performance is very low and almost any tentative of improvements is useless. **The code simply cannot be optimized because all the features available for improving performance are set-oriented and not row-oriented**. Any database system has so many great facilities for improving performance. Look at Oracle, see the DB2 system, choose SQL Server, and try to analyze any database system. You will see so many features for improving performance, you have so many things to read and follow and you need, as a developer or a DBA, to learn a lot to apply all these. Unfortunately, all these nice features are applicable for the set-oriented approach! They are almost useless if applied to the atomic style of development. So, if you want a bad performance in your database and if you want to be sure that no one can improve this performance, stay with your favorite atomic approach and work at the row level using cursors and all the others facilities!

Choosing to work SQL or non-SQL is a decision based on each one’s experience too. It is unavoidable and, with all our efforts to be as objective as we can, choosing one style or another is a matter of subjective and personal experience. Experience influences and is influenced by taste. Any programmer has its own taste and the decisions that he makes are influenced by all these factors. We are humans after all and not robots! The experience and taste can be driven by the learning process. For example, I am sure many of us were not completely aware of the level of granularity, of the duality between the data set and data row and the decision that we should always take when writing SQL. This distinction is very clear and simple, but very often we are not aware of clear and simple things. There are certain kinds of applications where choosing the level of detail - the row in our development is a catastrophic decision. Especially in systems where we are planning to simply move data between systems, in the so-called **specific** systems. The data movement process is like a sea. The data movement should be done in waves of data, and the waves are defined according to the business criteria. These waves of data are the data sets. Choosing the data set instead of the data row and trying to identify it whenever is possible should be the principle that drives any developer when writing code in the database.

One of the main tasks for an experienced database developer is the ability to read, understand and eventually drive the execution plan of a SQL statement. The execution plan shows what the optimizer will probably choose when executing a SQL statement. Tuning SQL statements for a better performance is another critical task. Generally, the database administrators are responsible because the production environment is the real world and they are the gatekeepers. However, in reality, the SQL tuning process starts much earlier in the development environment and should be managed by the development. First, a good database developer knows how to write a clean and accurate SQL. Depending on the quantity of data to be processed, the variety of features available in any database engine will be added to the SQL statement by the database administrator. If the logic is written by an application developer determined to follow his favorite and familiar atomic approach, there is almost nothing to tune and almost nothing to improve.

The section of SQL tuning, a very important section for database development, implies two steps. The first step is to try to catch the SQL statements with a bad performance, the SQL statements that seems to cause performance issues. For that, we have a variety of means and tuning tools like Oracle Enterprise Manager or SQL Server Profiler that will allow us to see and analyze the statements. Afterwards, we analyze all the facilities for performance and try to add them so the performance will be improved. There are so many manuals and courses about that and this is another topic, which I don’t plan to discuss in this book. However, what I want to state again is the fact that all these are useless if the development style is atomic. A precondition for all these tuning and performance to be applied is a holistic style of development because the tuning and the performance section rely in a holistic approach and in the set-based style of development.

DATABASE PROGRAMMING MEANS QUERY, QUERY AND QUERY ALL THE TIME!

Things are very clear and straightforward in database programming, only that everyone needs to understand this. I want to mention and reconfirm again the main task of any database developer: the task of query. The database programmer has almost the same tasks all the time. When doing his logic he is continuously manipulating business data, he is doing “DML” statements.

The programmer is mainly doing four things.

1. The programmer is reading business information. This is the **query**, the “**select”** statement. He is simply getting the data according to his needs. Learning how to select is very simple but is also very difficult. This is the paradox in database programming. A query is like a business request and the set of data returned by the query is the response to the business request… A business request may be very simple or very complicated. What may be simple or complicated to the request is not the request itself, even if this one may be in one way or another, but the means and facilities that the programmer should use to write the select statement. One of the first things the programmer should do is to identify the data set behind the request. The business request is always reflected in a data set behind! Being aware of this principle and thinking holistically is highly recommended.
2. The programmer may add new data, this means the “**insert”** statement. The insert can be simply done in classic insert-values syntax, where the values for the columns are manually supplied. The programmer may add data in a certain target from a source of data and he may use the most complex syntax insert-select. This type of insert it may be difficult and the difficult thing is the select statement and the **set-based orientation**. Whenever proposing to insert into a target from a source, we need to identify the source. The source of data may be taken from a table or from a variety of tables, linked by joins, eventually grouped by union. Therefore, the difficult part in the second type of statement that a programmer is doing, in the insert statement, is actually the same select statement. Again, the programmer should try to think **holistically** and try to identify the source of data as a whole. It is not always possible and, if not possible, he would move to a lower level of detail, lower to the data set, for most of the cases to the row.
3. The programmer may edit data, this means changing something. This is the “**update”** statement. An update can be divided in two segments. The first part is the identification of the rows to be updated. What do we want to update? The second part corresponds to the set of new values, what are the new values? When the programmer wants to see the rows to be updated, he wants to get a **data set**. Consequently, he needs to do a select statement first, the same old select statement. Secondly, he needs to find a set of data, again the same holistic vision. When he wants to find the new values, he can manually supply some values or he can get the values from some sources of data. Getting the values from some sources of data can be very difficult, for that, again we need

to build some select statements and use them to identify the sources of data for the new values.

4. The programmer can delete some data. In order to do that, he needs to identify the rows to be deleted. The programmer needs to write a select statement first - even if he does that in his mind - the select statement with the rows to be deleted. That will be, of course another data set.

Let’s look at the descriptions above and try to see some constants that occur everywhere, in all the four major actions. We notice that **select** statements should be executed all the time, before anything else. Even if a write action is required, a read action is usually correlated with it. A select statement is involved in almost anything. Secondly, we see that a **data set is always present** in any action, we are reading and we are writing data sets in any of the major actions involved in database development. Dividing the data set in data rows by default because the followers of structured programming are not aware of the data set is against the definitions of these major actions.

Why are we doing this review of these major actions? The practical section will start soon! A variety of exercises will demonstrate these simple concepts and principles of work previously described.

With this occasion, I wanted to reaffirm again the importance of the query in database programming. To be a database programmer you need to understand the query process, this is the most important chapter for a database programmer.

Secondly, looking at these major actions and reviewing them, we can also understand the importance of the set-based approach. Even in the definition of the four major SQL statements the data set is present everywhere. The data set division, apart from the technical limitations and reasons where the division is required, is against the definitions of the major actions in database programming. What is an update? An update means to change the values for a data set. What is a delete? A delete means to delete a data set. What is an insert into a target from a source of data? This represents an insertion of a data set from a source to the target. Everywhere we see the presence of the data set!

When doing database development, the first task is to get the data, so to use SQL. The only tool that allows us access to data is SQL.

Secondly, we are writing procedural code, we are doing classic development as a last resource, or as an additional resource. A database programmer needs to be able to think SQL, to think in data sets. Like any programming languages, we have all the general facilities like variables, structures, and others and we have specific features like cursors. These are part of any database developer life. However, what is important is to realize that these facilities are not to be used before SQL.

The logic is written in stored procedures, functions, and triggers. However, when looking inside them, we should see mostly SQL. If we see a cursor, this means there was no other way for the problem to be solved in SQL and a cursor was used.

For example, we need to concatenate some values in a column, values taken from another column. We may need to position ourselves in the data set, take the value from the column in a row-by-row approach, move the data in a variable, concatenate in another variable, get the result and update the other column. This is a context where the procedural facilities of the language are required and used accordingly. Pure SQL language was not enough! Therefore, this may happen, and we always have the cursors and the rest of procedural facilities that allow us to handle things in a row-by-row approach.

Let’s say that we are involved in a task like, for example, moving some data from one table to another, from a source to a target. The developer can think in many ways. He can try to declare some variables or structures: he can try to move the data elements into some temporary storage spaces like the variables, he can open a cursor, he can move the values in the cursor and, being in the cursor, he can try to copy the elements from the source into the target. This is the atomic vision of the programmer. This is, in my opinion, the vision that is compatible more with typical programming, the style of programming that is described in most of the courses and used by most of the developers. The atomic vision works, and the goals can be achieved. More than that, the atomic approach is followed by many application developers within the database. Thousands and millions of logics within the databases are written in this manner. It is easier for an application developer to use the same approach. Cursors are the best facility for that purpose. I use them time to time because it happens to be not able to solve the problems with pure SQL.

The problem is that this vision, the atomic vision is not exactly the best vision that matches with the database. The atomic vision is not optimal and is not suitable for data. The cursors match perfectly the atomic vision and this is the best way to write code atomically.

I also declare cursors and variables or records, move the data in a row-by-row approach into variables, manipulate them and finalize the logic sometimes. Still, this should be done only when the data manipulation is not possible with pure SQL. This is a rare situation. The atomic approach will be used when manipulation row-by-row is required and the strength of SQL is not enough to supply this goal. At that time, the cursor and the row-by-row approach is the best choice and it can be used. That is an exceptional situation!

LET’S GO BACK TO THE SQL SHOP! ONE MORE TIME, PLEASE!

I’m going to use again some analogies to illustrate the differences.

Let’s go back to our shop, the database shop. I am with my wife and we have a basket full with products, of course, she is the boss in all these domestic matters! We go to the cashier and we are ready for pay. We have ten chocolates in the basket, with different colors. Instead of summarize them all together we start adding the chocolates one by one, pay for each chocolate, and then continue with the next one. Obviously, a huge queue forms behind us. People in the supermarket get upset. The question is why are upset? If we have just two chocolates, there is near closing time for lunch in Spain, and most of the people are taking their siesta, so actually there are not too much people in the supermarket, it shouldn’t be such an issue.

The fact that I rather pay for the two chocolates in two times instead of paying them in one shot it does not harm anyone. Still, if I have ten chocolates and it’s the rush hour in the afternoon, and the supermarket is full of people coming back from the office, then, paying for ten chocolates, even two, in many shots instead of one single shot may cause serious inconveniences.

Wait! The cashier asked me for a bag. I looked at her very surprised! What do you want me to do with a bag? - I asked her. I need ten bags for my ten chocolates. I believe I will give up, the bags are useless for me I can take everything in my hand and go to my car: my wife will help me so I can deliver the chocolates relatively easy.

The cashier stared at me. She almost refused to believe! Are you joking? – She said! What world are you coming from, are you from Mars or Pluto or something? Don’t you know what these bags are for? They allow you to add many goods in one and transport them easily. If you consider adding the goods one by one, you don’t just act illogically and against the common sense, you don’t just waste your time and others, but you also refuse any tool like a bag, dedicated to this purpose.

So what’s the main reason for these inconveniences apart from the fact that it seems quite a lack of logic to pay for ten chocolates ten times instead of paying them in one shot? The reason is clear: performance. The performance is very poor and the main consequence is the bad timing for the queues. More than that, any feature for performance improvement is useless.

Similarly for the database, by using the atomic vision and being used to affect the data atomically, may not cause too much harm in the testing activity where the quantity of data is generally low and the variety of data is generally reduced.

Nevertheless, in production, there it makes a big difference and very often, this inappropriate style may cause issues.

I can tell you that, at least in my activity, whenever I was involved in performance, almost all the time I was forced to rewrite large pieces of code and replace the atomic vision with the holistic vision described everywhere in this paper.

The alternative to the atomic vision that is so familiar to the application developers is the holistic vision that is intimately related to SQL, to the data in the relational model and to the concept of data set.

Let’s go back to the example in the beginning of the paragraph. We have some data to move from one source of data to a destination. If we think holistically, which means we think SQL, we analyze the entire set of data in the source. We identify the data set to be transported into the target. For that, we need a query: that is all we need! This is actually the most difficult part. We simply take the entire set of data and move it in one shot in the target. There is no need for more: we do not need any programming facilities like cursors and structures. Sometimes the complexity of this task may require these facilities, especially when the set of data from the source needs to be manipulated row-by-row before being added into the target. There are an enormous number of scenarios; I don’t want to generalize in any way. Very often, more often than anyone can imagine, the degree of simplicity for this basic operation, moving data from one place to another, has such a level that a simple SQL should be enough to fulfill the task. In addition, very often, due to the atomic style that is so appropriate to a large number of programmers, this task is done in a very complicated way. This causes many issues, especially in performance.

We all know how important performance is for everyone. My opinion is that, very often, the main reason for a poor performance in the database is not a wrong indexes strategy, or something of that kind, but the use of this atomic approach in the code, a style that is highly unsuitable for the database.

THE USE OF SCALAR FUNCTIONS –A TYPICAL ACCESSORY FOR THE ATOMIC APPROACH!

One valuable principle when studying some languages like C or Pascal is the use of functions. Any student starts by learning the basics like variables, all kind of loops, conditional statements. The student learns arrays, then structures. In a future step of his learning process, the student learns the use of functions.

Before that, he did everything in the main function and then he suddenly realizes the power of functions and one principle of structured programming: he learns how to split every distinct task according to the business, and how to embed it in a function.

The function can return a true value, like an integer or a string, or can do something without returning anything, like a void function. The programmer is seduced by this principle and he keeps it in his mind all his life. He is extremely happy that the use of functions allows him to organize his work better and to divide the complexity of his activity into less complex tasks.

The programmer builds functions over functions and he remains consequent to this principle of divisions: to divide the complexity and separate the task into distinct tasks, embedded in functions.

Afterwards, if he were to use some programming languages, he would learn the use of the procedure and sees if he were going to like a void function or not.

After some time, the programmer starts working in the database too. No one told him that this is something else and he should not necessarily follow exactly the same paths and principles. He sees that there are two main types of procedural facilities: procedures and functions. He acknowledges that a stored procedure aims to perform an action while a function retains original meaning, to return a value. He understand the use of a stored procedure and he sees that in most of the cases he is doing some particular actions, especially read or write data using the set of “DML” statements. The principle of division, divides the logic in smaller pieces according to the business, is equally valuable and applicable in the database like in the user interface.

However, the use of functions should be seen differently. The application programmer is used with functions and he can easily see the combination between a cursor and a function. He sees the **scalar function** right away as an ideal type of procedural facility for his atomic approach. A scalar function is that type of function that acts per variable, per column and row. It is the last type of function the programmer should use if he wants to follow the set-based approach! The application developer is used with the use of functions from the user interface and he is tempted to translate that in the database. This is another bad decision, in my opinion, as I will argue right now.

The scalar function returns a certain value of a certain type. Any scalar function can be applied to a certain column or expression in a certain row. The scalar function is the best facility for the atomic approach and it is one favorite facility for many application developers working inside the databases. Let’s imagine you have one thousand rows in a data set and you create one function and apply it in a cursor in the data set one thousand times, instead of avoiding that function and do the entire logic in the data set directly! This is a very common situation in many databases!

On the other hand, in the database we have many types of functions. We have table functions that apply to data sets, these types of functions are much better than scalar functions and a good database developer should use these ones and apply directly per sets of data. That is the difference. A table function can be called per a data set while a scalar function would always be called per column and row in a cursor.

A scalar function is to be defined and used if we apply it to configuration tables. We know that one certain function will return one single value from a configuration table: this is the ideal scenario for a scalar function.

In the database, the main procedural object is the stored procedure. This is obvious because in the database the main task is the data manipulation, either read or write. This is what we are doing in most of our stored procedures, read data and write data. Using the holistic approach and the set-based approach, we read and write data sets.

If we need to generate data sets and use the generated output as input for something else, we can define a table function. The role of the function is very different in database programing. This is another principle that the application developer should understand before starting creating hundreds of scalar functions and call them in cursors!

DEBUGGING IS SO SIMPLE! THE CODE IS MUCH SIMPLER AND READABLE!

Another advantage of the holistic approach is the easy use of debug.

I am referring now to those **specific** systems, where the goal is to move data between various systems. That might be a data migration system, or a replication system, or an “ETL” system etc. If this system is built entirely in SQL, as I recommend, apart from all the advantages described in the previous pages, I want to mention another one. I am referring to the methods for debug. In these specific systems, we do not have the classic debug functionalities with variables and watch. Still, we have a much simpler debug method to use, for people that are using the holistic approach. Whenever you have an error of a certain type, as a constraint violation, conversion error or anything else, and you have an error handling procedure that shows you the place where the error occurs, it is very simple. All you need to do is to simulate everything until that point; you need to make sure you have the data when the error occurs, and finally take the entire set of data separately, check the data and see what causes the error.

For example, we are adding some new products in a migration system from one “ERP” system to one target system, an inventory system. For some reason, some materials attributes are not added and a constraint violation occurs in the step with the following attributes. We can simply reload the process until that point, make sure we are able to have the same data until that point, and then retake the step with the error, execute the insert statements and the logic for that step and, when we are at the area with the error, start and investigate holistically what is happening.

This kind of debug is very simple for anyone familiarized with the holistic vision, the SQL vision of set of data. I will try to illustrate some examples of debug in one of my future books where I am planning to describe a replication system written in pure SQL.

In most of the cases, the reason for the error should be a data set. All you need is to take that data set, analyze it, and try to understand which details of the data set were causing the error.

If the migration system is properly organized in steps, if the error handling procedure is built coherently then is able to catch the error and store it in in the error table. There we can see the step, and the error identifier and description. We can have constraint violations errors or conversion errors or whatever types of errors. We simply comment the steps after the step with the error and run the interface until that point. We then take the step with the error and check the data. We query the data before the error, analyze it and see what caused it. Again, for debugging, SQL is the required skill. In order to be able to identify the reasons for the error we need to query the data, and see what was happening. It is elementary, very straightforward!

Another advantage of the holistic approach is readability. The code is much clearer, and you can see everything organized in data sets. I admit that this apparent clarity and readability is somehow subjective and relative to the developer. I admit that the developer, being used to classic programming, may find the SQL style not so readable. He may believe that his style, with lots of cursors and structures and loops, is more readable. Honestly, I don’t want to force anyone to accept my idea, maybe this matter is more subjective than I believe it is. My strong opinion is that the code is simpler and readable if written in the SQL style instead of being written in the procedural style. At least, the length of the procedures is for sure smaller if you use the SQL style. That does not mean you cannot have SQL statements to share many pages, if the complexity is beyond the average!

WHAT IS A DATABASE DEVELOPER NOWADAYS?

I want to add some words about database development, the world where I claim I belong too along with so many others professionals!

Let’s look at the market. The database development is an accepted specialization nowadays, after many years when only database administrators were accepted as distinct database specialists.

Now there is an explosion of database specialists and a continuous and increasing demand. This change is because more and more reporting and analytics databases are built over the set of operational systems everywhere.

The sections of data warehouse, not mentioning the big data technologies, are more and present everywhere. The BI technologies are more increasingly popular in the market. Moreover, who are the specialists to handle all these projects? The database developers along with business analysts, reporting analysts are the ones that can sustain all these increasing demands.

From my perspective, as a SQL developer and specialist, despite the appearances, things are not completely different! The SQL is still considered in the “other” section for many projects and many teams, where application developers should know some SQL, among others. The application developers still consider SQL similar to their application development and use their own style in the database and, even more, there are database developers specialized in one language or another that write code in a similar way with the application using the atomic style of development. Let’s analyze a specialized PL SQL developer that is using the atomic approach and the procedural facilities. He calls himself a database developer. I call him an application developer, because he does not think SQL and holistically, even if he is developing in PL SQL.

In the field of data warehousing, where there are so many “ETL” coming up, in the middle or large companies with a variety of systems communicating between them by specific replication systems or data migration systems, more and more database developers are requested to do the job. The set-oriented approach is requested on the market and this shows that the enterprises are aware of the necessity of the holistic approach. The project managers should either decide to use specialized database developers or try to teach application developers to write holistically.

IT’S PRACTICE TIME!

Now, finally, after so many discussions, concepts, and clarifications, it is time for some exercises. I will try in the next following chapters to illustrate the two styles of development with examples. I will try to show that the holistic approach is a better approach when being in the database.

As I mentioned, the examples are taken from two major database management systems, Oracle and SQL Server. I want to reaffirm that all the considerations and thoughts described in this book are general and am referring to any relational database system. The holistic approach is a vision that should be present in any relational database. Oracle database and Microsoft SQL Server database are taken as examples to illustrate the principles described here. Similar examples can be easily reproduced in any other database system.

You may see one advantage of the holistic approach when trying to translate some of the examples in your database, if other than Oracle and SQL Server.

It will be very easy for you to translate the holistic version, because SQL is a standard and the syntax will be very similar. In the atomic approach, you will have some work to do to translate the examples from Oracle or SQL Server to your system. The procedural languages are similar but there are differences, there is no standard between them apart from the same mode of structure programing. Still, we don’t need to exaggerate, a cursor is a cursor, a loop is a loop, and things are similar here too. However, the quantity of work necessary to keep versions written in procedural code for any of the database management systems is infinitely higher than if the logic were written in SQL.

Chapter 5

DATA TRANSFER PARADIGM, THE FIRST SET OF EXAMPLES

THE EXERCISES, THE CONTEXT, THE GOALS, WAYS TO ILLUSTRATE THE TWO APPROACHES!

The title of this book is “Two styles of database development” and it is divided in two parts: the concepts and the practice.

Starting with this chapter, I’m going to detail the two styles by using a large variety of examples. Some people may still argue against the efficiency and advantages of one style or another but no one can deny the distinctions anymore. My main goal in this book is to clarify the two styles of development, to show the differences between them, the advantages and disadvantages of each of them. I also intend to promote the holistic style but this is somehow subjective to a certain degree. Anyone may experiment and may try to apply the two styles and anyone can compare if he wishes to. If people are aware of the distinctions then my main goal is achieved and I am satisfied.

The considerations and the arguments for a certain style of programming, specific to the database, against a classic style specific to the user interface are going to be illustrated in the following chapters by a list of practices and exercises. Many examples will try to explain the two approaches, the holistic approach in opposition to the atomic approach, and the purpose is to show that, inside a relational database, the holistic approach should be used in most of the cases.

I tried to organize the examples so they can be compared and tested in the readers systems. I invite you to run the exercises and see the differences for yourself.

Of course, ideally would be to try to imagine similar practices in your systems and test one approach versus another in your logic inside your databases. Even more ideally would be to try to effectively change your style in the systems you are effectively working at this moment, if that applies to you. Only looking and analyzing real data and real scenarios, you will effectively see the differences.

Take these examples and practices as what they are: some simple exercises!

The most important things have been already said in the previous chapters, the main ideas have been even repeated many times. These examples will just reinforce and try to confirm the statements, advices and ideas expressed in the first part of the book.

One of the most important characteristic of an exercise or practice is the context. The context of an example consists of the business description of the exercise, the technical description that should be generated from the business description, the characteristics and prerequisites of the sample, like the data definition statements that may be used for the exercise, the goals that need to be achieved in each example. I will not insist too much on the business description considering mainly the technical description. Most of the exercises are illustrated in both Oracle and SQL Server, more precisely in both

PL SQL and Transact SQL. Very often, I use stored procedures to illustrate the practices.

Please take note again of the fact that the goal is to illustrate a holistic approach in a relational database system. Almost all the principles and almost all the examples can be reproduced similarly in any other database system like IBM DB2, PostgreSQL, Sybase and others. Oracle and SQL Server are the systems where I choose to show the relevant examples: that is all! These principles are not specific to these two popular database management systems, are common to them and to all the others of the same kind.

To conclude, most of the practices will contain the followings:

1. The practice will be defined by a business description followed by a technical description.
2. The practice will contains a set of data definition language (“DDL”) statements associated with the practice, like the list of prerequisites or the list of tables involved.
3. The practice will contain the descriptions of the goals to be achieved in the practice.
4. I will show a sample of data, whenever will be the case.
5. I will illustrate and describe the various techniques that are used and the explanations and reasons for choosing one technique or another.
6. I will display the scripts in Oracle or SQL Server, or both. These scripts will sometimes be embedded in stored procedures, as the most representative type of object in database programming.
7. I will always compare the two approaches, because the purpose of these examples is to compare and describe the advantages or disadvantages of one style of development or another. I admit that the purpose of most of the practices is to promote the holistic, set-oriented style, specific to database developers, against the atomic and row-oriented style, specific to application developers.

PRACTICE 1: A FULL DATA TRANSFER BETWEEN TWO SYSTEMS - ONE COMMON OPERATION.

I tried to imagine simple and relevant exercises because the purpose is didactic. The first example tries to illustrate one common situation in database programming, the data transfer paradigm.

Very often, when doing database development, we are simply transferring data between a source and a target. By transferring data, I can understand moving data from A to B, eventually updating or deleting information based on certain conditions. The process of data transfer can be made in any kind of application, in a classic application or a specific application. It is one of the most common operations in database programming.

To illustrate some data transfer practices, I choose a simple model.

You can imagine two systems and you can imagine that you are moving data between these two systems. The first system is a normalized system and the second one is a reporting or analytic system where a certain degree of normalization may exist. If you prefer, you can rather imagine one single system where the targets are part of the reporting area of that system.

A data transfer can be full or incremental. I consider the transfer a full one if the target is completely deleted before the transfer. An incremental transfer is committed when the data is moved incrementally, only the changes from the source system are applied to the target.

The first example will illustrate a full transfer and the second will show an incremental data transfer, a more complex scenario.

Let’s define the context! This is going to be the base for most of the practices that will follow!

FILTER ENGLISH OR EUROPEAN COUNTRIES: THE BUSINESS AND TECHNICAL DESCRIPTION AND GOALS

The context specifies the descriptions of both source and target systems. A set of countries, languages and their association is part of the source system and a set of reporting tables per language is part of the destination system. The source system, let’s name it A, contains, among others, a set of three tables to store the following information:

1. The source will contain a list of countries.
2. The source will contain a list of languages.
3. The source will contain the association between countries and languages, including some common information per both country and language.

The target system, let’s say B, may be a separate system or not. You can consider a separate reporting system, where the data is denormalized according to the language. The system contains, among others, a list of tables per language with country information.

As examples, I choose one table for the English language and one for French. You can imagine that the target system contains a list of reporting tables per language. For simplicity, I assume just English and French languages, because these languages are enough to illustrate my goals.

The information from the source system A should be moved into the destination system B. The goal is to generate the set of countries for the English language in the specific table containing only the English countries and, eventually, the set of countries for French language in the specific table containing the French countries.

FILTER ENGLISH OR FRENCH EUROPEAN COUNTRIES: THE PREREQUISITES.

Let’s continue with the context definition. We just saw the business and technical description of the context. Now let’s see the table design. The source system A is composed of three tables, as you can see below:

Code example 03: Countries and languages design

```
CREATE TABLE Countries
(
Country_Id INT CONSTRAINT NN_Country_Id NOT NULL,
Country_Code VARCHAR(3) CONSTRAINT NN_Country_Code NOT NULL,
Country_Name VARCHAR(50) CONSTRAINT NN_Country_Name NOT NULL,
Continent VARCHAR(15) CONSTRAINT NN_Country_Continent NOT NULL,
CONSTRAINT PK_Country_Id PRIMARY KEY (Country_Id),
CONSTRAINT UQ_Country_Code UNIQUE (Country_Code),
CONSTRAINT CK_Country_Continent
CHECK(Continent IN ('Europe', 'North America','South America', 'Asia', 'Africa', 'Australia', 'Central America'))
);
CREATE TABLE Languages
(
      Language_Id INT CONSTRAINT NN_Language_Id NOT NULL,
      Language_Name VARCHAR(50)
CONSTRAINT NN_Language_Name NOT NULL,
      CONSTRAINT PK_Language_Id PRIMARY KEY (Language_Id),
      CONSTRAINT UQ_Language_Name UNIQUE (Language_Name)
);
CREATE TABLE Countries_Languages
(
      CL_Id INT CONSTRAINT NN_CL_Id NOT NULL,
      Language_Id INT CONSTRAINT NN_CL_Language_Id NOT NULL,
      Country_Id INT CONSTRAINT NN_CL_Country_Id NOT NULL,
      Language_Category VARCHAR(10)
CONSTRAINT NN_CL_Category NOT NULL,
      Make_Flag INT,
      CONSTRAINT PK_CL_Id PRIMARY KEY (CL_Id),
      CONSTRAINT UQ_Language_Country
UNIQUE (Language_Id, Country_Id),
      CONSTRAINT CK_CL_Category
CHECK (Language_Category IN ('MAIN', 'SECONDARY')),
      CONSTRAINT CK_CL_Make_Flag CHECK (Make_Flag IN (0, 1)),
      CONSTRAINT FK_CL_Countries
FOREIGN KEY (Country_Id) REFERENCES Countries(Country_Id),
      CONSTRAINT FK_CL_Languages
FOREIGN KEY (Language_Id) REFERENCES Languages (Language_Id)
);
-- Build an Oracle sequence too
CREATE SEQUENCE CL_Id_Seq;
```

According to the above design, there is one table for the countries, one for the languages and one for the association between the countries and the languages. According to the business needs, some languages should be associated with some countries and some common information will be added there.

Despite the fact that these are well-known design tips, for the non-SQL specialists, readers of this book, I'll try to add some basic considerations. By looking at the script, we notice the following aspects, concerning the first chapters when we discussed about the fact that the first stage of development is actually the base-objects design, especially tables design. I want to review some of the design considerations looking at this script.

1. Every table has its own primary key, as it should be in a normalized system. The primary key is enforced by an artificial column, with no business meaning.

This is the most common scenario. I always prefer to keep the primary key away from the business!

2. Sometimes, when the primary key is artificial we can add a unique constraint. That is the so-called unique business key and that defines the table, from the business point of view. For example, the code of the country should be unique and that code has a business meaning, compared with the artificial key that has the only purpose to uniquely identify one row. The combination language and country should also be unique in the association table, so you can see the unique constraint. The business key can be defined as primary key. Technically speaking is a suitable candidate. Still, I prefer to separate things and to always let the primary key away from the business.
3. The constraints are all named constraints, not system generated. The names are very relevant, generally obtained by the concatenation between the table, column and constraint type. The names can be seen in the set of system objects or data dictionary and the developer does not need to investigate very much, he can already understand looking at the names. The constraint names are also visible in error messages and the developer will quickly see what is about. I recommend that the prefix of the constraint to be the constraint type.
4. Add the check constraints or foreign key constraints whenever is possible. If you know exactly the values, do not hesitate and do not leave the columns optional. Combine the check constraints with **NOT NULL** constraints. In most of the cases, a column with low selectivity can have a check constraint. See if that is the case and add both constraints, if possible. If the list with values defined by the check constraint will increase in time, you should transform the check constraint into a foreign key constraint. Add a lookup table first and restrict the table using that lookup table. Before choosing the type of constraint, think in perspective. Even if now there are some few values, if you anticipate the list may grow, add the lookup table from the beginning.
5. It is good to apply the constraints and to restrict the data to be conformed to the business definition. If the layer of constraints is properly set from the beginning, you have a good starting point in your database development activity. You can see in the example that almost every column has at least one constraint attached to it. Of course, there are descriptive fields that will not have any constraints. However, the tip is to try to see if any type of constraint can be applied in one way or another to the column and do this investigation constantly for any column in any table.
6. You can see that this script can be executed, with absolutely no change, in Oracle and SQL Server and in others database systems too. Take note again of the advantage of the standard.1

Let's continue and illustrate the destination system. The target system B is composed of a list of reporting tables, per country.

```
Code example 04: English and French
CREATE TABLE English_European_Countries
(
      English_CL_Id INT CONSTRAINT NN_English_CL_Id NOT NULL,
      Country_Code VARCHAR(3)
```

```
  CONSTRAINT NN_LCountry_Code NOT NULL,
        Country_Name VARCHAR(50)
  CONSTRAINT NN_LCountry_Name NOT NULL,
        Language_Category VARCHAR(10),
        CONSTRAINT PK_English_CL_Id PRIMARY KEY (English_CL_Id),
        CONSTRAINT UQ_ECountry_Code_Category
  UNIQUE (Country_Code, Language_Category));
  CREATE TABLE French_European_Countries
  (
        French_CL_Id INT CONSTRAINT NN_French_CL_Id NOT NULL,
        Country_Code VARCHAR(3)
  CONSTRAINT NN_FCountry_Code NOT NULL,
        Country_Name VARCHAR(50)
  CONSTRAINT NN_FCountry_Name NOT NULL,
        Language_Category VARCHAR(10),
        CONSTRAINT PK_French_CL_Id PRIMARY KEY (French_CL_Id),
        CONSTRAINT UQ_FCountry_Code_Category
  UNIQUE (Country_Code, Language_Category)
  );
  Others reporting tables may be added per various languages.
```

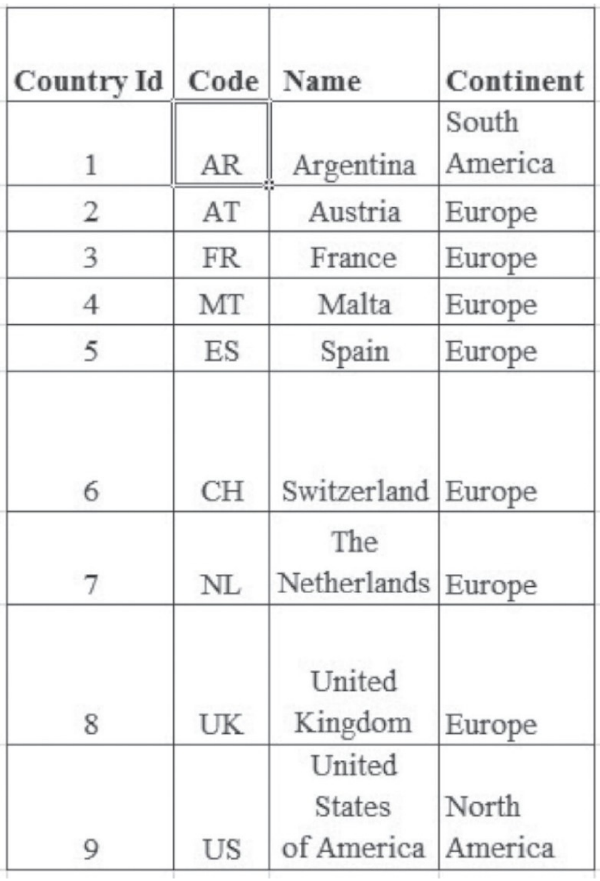
You can imagine a reporting activity where the reports are set per language and the data is divided per language. The examples want to illustrate a style, and this style is recommended in the database in most of the cases, a classic system but mostly a specific system like a replication system, a data migration system or a data warehouse system.

FILTER ENGLISH OR FRENCH EUROPEAN: THE SAMPLE OF DATA

Let’s see the initial data! This is very useful for the illustration of most examples. For anyone working in the database, programmer or analyst or tester, the data should have a strong relevancy. The business reflects in the data. Any database design should be associated with some samplings.

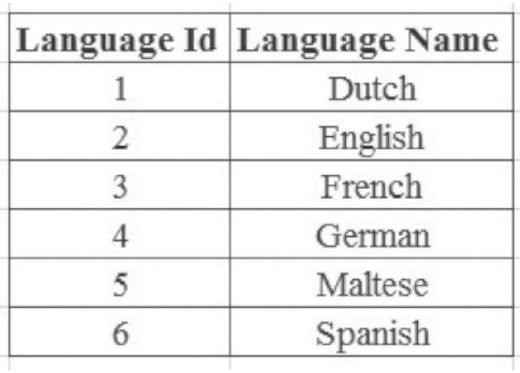
Seeing the initial data is also useful because the reader can simply generate all the examples completely in his machine and he may see with his own eyes the results, he may have a confirmation of all the things that have been explained.

See the data values in tabular format see the table Countries.



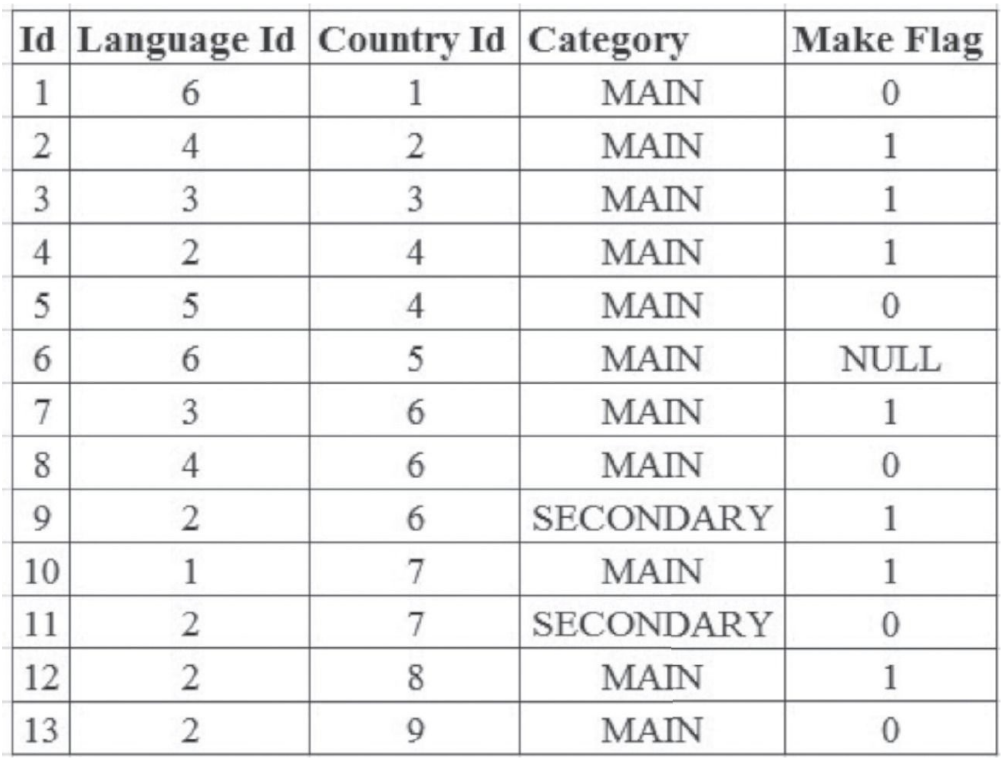
Country Id	Code	Name	Continent
1	AR	Argentina	South America
2	AT	Austria	Europe
3	FR	France	Europe
4	MT	Malta	Europe
5	ES	Spain	Europe
6	CH	Switzerland	Europe
7	NL	The Netherlands	Europe
8	UK	United Kingdom	Europe
9	US	United States of America	North America

See the data values in tabular format see the table Languages.



Language Id	Language Name
1	Dutch
2	English
3	French
4	German
5	Maltese
6	Spanish

See the data values in tabular format see the table Countries Languages.



Id	Language Id	Country Id	Category	Make Flag
1	6	1	MAIN	0
2	4	2	MAIN	1
3	3	3	MAIN	1
4	2	4	MAIN	1
5	5	4	MAIN	0
6	6	5	MAIN	NULL
7	3	6	MAIN	1
8	4	6	MAIN	0
9	2	6	SECONDARY	1
10	1	7	MAIN	1
11	2	7	SECONDARY	0
12	2	8	MAIN	1
13	2	9	MAIN	0

AN EXAMPLE OF INSERT SCRIPT!

Before starting the first example, due to the fact that most of the exercises will be taken from these samplings with countries and languages, I would rather specify the insert script here so anyone who wants to effectively execute all the practices to have the possibility to reproduce the exact conditions as in my system.

This is the so called “insert script” or, in our case, initialization script. The countries, languages and their associations are initialized with the required quantity of data. The tables above show this initial data. Now the same data can be seen in a more technical manner, in an insert script.

Code example 05: Populate countries and languages

```
      INSERT INTO Countries (Country_Id, Country_Code, Country_ Name, Continent) VALUES (1, ‘AR’,
‘Argentina’, ‘South America’);

      INSERT INTO Countries (Country_Id, Country_Code, Country_ Name, Continent) VALUES (2, ‘AT’,
‘Austria’, ‘Europe’);

      INSERT INTO Countries (Country_Id, Country_Code, Country_ Name, Continent) VALUES (3, ‘FR’,
‘France’, ‘Europe’);

      INSERT INTO Countries (Country_Id, Country_Code, Country_ Name, Continent) VALUES (4, ‘MT’,
‘Malta’, ‘Europe’);

      INSERT INTO Countries (Country_Id, Country_Code, Country_ Name, Continent) VALUES (5, ‘ES’,
‘Spain’, ‘Europe’);

      INSERT INTO Countries (Country_Id, Country_Code, Country_ Name, Continent) VALUES (6, ‘CH’,
‘Switzerland’, ‘Europe’);

      INSERT INTO Countries (Country_Id, Country_Code, Country_ Name, Continent) VALUES (7, ‘NL’,
‘The Netherlands’, ‘Europe’);

      INSERT INTO Countries (Country_Id, Country_Code, Country_ Name, Continent) VALUES (8, ‘UK’,
‘United Kingdom’, ‘Europe’);

      INSERT INTO Countries (Country_Id, Country_Code, Country_ Name, Continent) VALUES (9, ‘US’,
‘United States of America’, ‘North America’);

      INSERT INTO Languages (Language_Id, Language_Name) VALUES (1, ‘Dutch’);

      INSERT INTO Languages (Language_Id, Language_Name) VALUES (2, ‘English’);

      INSERT INTO Languages (Language_Id, Language_Name) VALUES (3, ‘French’);

      INSERT INTO Languages (Language_Id, Language_Name) VALUES (4, ‘German’);
      INSERT INTO Languages (Language_Id, Language_Name) VALUES (5, ‘Maltese’);
      INSERT INTO Languages (Language_Id, Language_Name) VALUES (6, ‘Spanish’);
      INSERT  INTO  Countries_Languages  (CL_Id,  Country_Id,  Language_  Id,  Language_Category,
Make_Flag) VALUES (1, 1, 6, ‘MAIN’, 0);
      INSERT  INTO  Countries_Languages  (CL_Id,  Country_Id,  Language_  Id,  Language_Category,
Make_Flag) VALUES (2, 2, 4, ‘MAIN’, 1);
      INSERT  INTO  Countries_Languages  (CL_Id,  Country_Id,  Language_  Id,  Language_Category,
Make_Flag) VALUES (3, 3, 3, ‘MAIN’, 1);
      INSERT  INTO  Countries_Languages  (CL_Id,  Country_Id,  Language_  Id,  Language_Category,
Make_Flag) VALUES (4, 4, 2, ‘MAIN’, 1);
      INSERT  INTO  Countries_Languages  (CL_Id,  Country_Id,  Language_  Id,  Language_Category,
Make_Flag) VALUES (5, 4, 5, ‘MAIN’, 0);
      INSERT  INTO  Countries_Languages  (CL_Id,  Country_Id,  Language_  Id,  Language_Category,
Make_Flag) VALUES (6, 5, 6, ‘MAIN’, NULL);
      INSERT  INTO  Countries_Languages  (CL_Id,  Country_Id,  Language_  Id,  Language_Category,
Make_Flag) VALUES (7, 6, 3, ‘MAIN’, 1);
      INSERT  INTO  Countries_Languages  (CL_Id,  Country_Id,  Language_  Id,  Language_Category,
Make_Flag) VALUES (8, 6, 4, ‘MAIN’, 0);
      INSERT  INTO  Countries_Languages  (CL_Id,  Country_Id,  Language_Id,  Language_Category,
Make_Flag) VALUES (9, 6, 2, ‘SECONDARY’, 1);
      INSERT  INTO  Countries_Languages  (CL_Id,  Country_Id,  Language_  Id,  Language_Category,
Make_Flag) VALUES (10, 7, 1, ‘MAIN’, 1);
      INSERT  INTO  Countries_Languages  (CL_Id,  Country_Id,  Language_Id,  Language_Category,
Make_Flag) VALUES (11, 7, 2, ‘SECONDARY’, 0);
      INSERT  INTO  Countries_Languages  (CL_Id,  Country_Id,  Language_  Id,  Language_Category,
Make_Flag) VALUES (12, 8, 2, ‘MAIN’, 1);
      INSERT  INTO  Countries_Languages  (CL_Id,  Country_Id,  Language_  Id,  Language_Category,
Make_Flag) VALUES (13, 9, 2, ‘MAIN’, 0);
```

FILTER ENGLISH OR FRENCH EUROPEAN COUNTRIES: THE TWO OF THE POSSIBLE SOLUTIONS, ACCORDING TO THE TWO STYLES OF DEVELOPMENT.

The first example is relatively simple. Even an application developer with not so much SQL experience may choose the holistic approach. Still, this is rarely happening.

Let’s analyze the data elements once more before starting to see the examples.

You can see from the sample that there is a list of countries, with an artificial identifier, a name, a unique code and the continent. The languages table has just the artificial identifier and the name of the language. The association table has one language and one country, like any association table. Apart from that, the category of a language can be main or secondary for the country and there is a flag for each country language combination. The destination table has another artificial identifier as primary key, the country code, name and the category. We are positioned in the English or French language so we know that we are within a certain language. That is why any language reference is not required, just the country information.

Let’s suppose that this task is done by an authentic application developer that always sees things atomically.

He understood the cursors functionalities and he is used to implement cursors in combination with the associated loop statements. Let’s see the first solution, and the atomic approach.

I named the procedure as such because I want to illustrate that the procedure is atomic. The transfer is a full transfer because the target table with countries is first deleted. Let’s see the Oracle version.

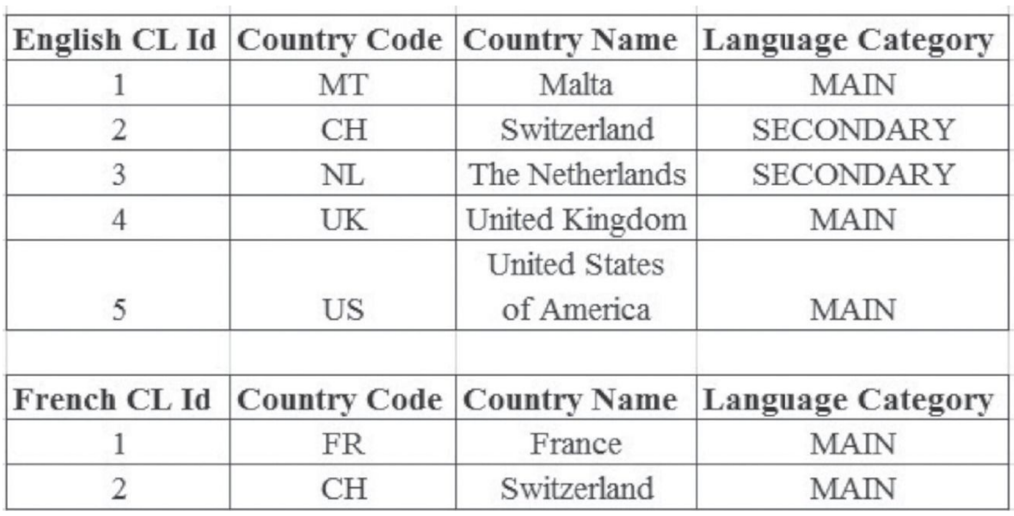
Code example 06: Oracle Atomic Full transfer

```
CREATE PROCEDURE Atomic_Full_Transfer_Country
(p_Language_Name VARCHAR)
AS
      v_Country_Name VARCHAR2(50);
      v_Country_Code VARCHAR2(3);
      v_Language_Category VARCHAR2(10);
      v_New_EEC_Id INT;
      CURSOR c_Get_Countries (p_Language VARCHAR2) IS
      SELECT c.Country_Name, c.Country_Code, cl.Language_Category
      FROM Countries_Languages cl INNER JOIN Languages l
          ON (l.Language_Id = cl.Language_Id)
      INNER JOIN Countries c
          ON (c.Country_Id = cl.Country_Id)
      WHERE l.Language_Name = p_Language;
BEGIN
      v_New_EEC_Id := 1;
      IF p_Language_Name = ‘English’ THEN
          DELETE English_European_Countries;
      ELSIF p_Language_Name = ‘French’ THEN
          DELETE French_European_Countries;
      END IF;
      OPEN c_Get_Countries (p_Language_Name);
      LOOP
          FETCH c_Get_Countries
          INTO v_Country_Name, v_Country_Code, v_Language_Category;
          EXIT WHEN c_Get_Countries%NOTFOUND;
      IF p_Language_Name = ‘English’ THEN
      INSERT INTO English_European_Countries (English_CL_Id,
      Country_Code, Country_Name, Language_Category)
      VALUES (v_New_EEC_Id, v_Country_Code, v_Country_Name, v_
      Language_Category);
      ELSIF p_Language_Name = ‘French’ THEN
      INSERT INTO French_European_Countries (French_CL_Id,
      Country_Code, Country_Name, Language_Category)
      VALUES (v_New_EEC_Id, v_Country_Code, v_Country_Name, v_
      Language_Category);
      END IF;
      v_New_EEC_Id := v_New_EEC_Id + 1;
      COMMIT;
      END LOOP;
      CLOSE c_Get_Countries;
END Atomic_Full_Transfer_Country;
/
```

This is the first solution using the atomic approach and the associated style of development. Just to clarify, even if these details should be straight forward, I will describe the flow for the first example, in the atomic approach.

1. The parameter is the language, either French or English and the list may expand for others languages, of course.
2. The developer is declaring some variables to store the data to be inserted, for country name and code, for the language category and for the key identifier that he should generate. Obviously, he can use a record or structure instead of the variables and achieve the same goal. The purpose is to be able to store atomically the values to be inserted in these variables. You can already see how the programmer sees things atomically and he is preparing to store the data and manipulate it at the atomic (row) level.
3. The application programmer is declaring the cursor that will store the country name, code and the language category from the series of source tables Countries Languages, Languages and Countries. The cursor with the loop facility, that allows us to position wherever we want it in the data set, is the base for the atomic approach.
4. We initialize the value for the identifier to one, using the dedicated variable v_New_EEC_Id. The primary key from the target, an artificial identifier, will have to receive a unique value taken from this variable.
5. Based on the language, either English or French, the logic deletes one reporting table or another. The transfer is full so the target table is deleted first.
6. The programmer opens the cursor and starts adding the values from the cursor into the variables using the fetch instruction. This is the classic series of steps in every cursor.
7. Based on the language, the data is added into the target table, row by row. For every row, the data is inserted from the variables that stores the current set of values from the cursor. The same if-else statement is used to detect the target table based on the language.
8. The developer will increment the key to prepare the next value for the next iteration.
9. This procedural style corresponds to the atomic approach of programming. This is the type of development that the application developer is familiarized by analogy with the models that he knows. Many application developers think in this manner, by default. They think atomically in almost any circumstance and they try to use the same style in the database like in the user interface. They know the principles of structured programming and they apply them here at the row level, trying to use the same style everywhere.

Let’s see the results, for both countries.



English CL Id	Country Code	Country Name	Language Category
1	MT	Malta	MAIN
2	CH	Switzerland	SECONDARY
3	NL	The Netherlands	SECONDARY
4	UK	United Kingdom	MAIN
5	US	United States of America	MAIN

French CL Id	Country Code	Country Name	Language Category
1	FR	France	MAIN
2	CH	Switzerland	MAIN

Before continuing to comment more this approach let's see the SQL Server version of the atomic approach. You may see all the steps, and understand the classic, procedural and atomic style of development. Move the data into the variables one by one and populate the table with new rows one by one according to the values stored in the variables.

```
Code example 07: SQL Server Atomic Full transfer
CREATE PROCEDURE Atomic_Full_Transfer_Country
(
@p_Language_Name VARCHAR(50)
)
AS
      DECLARE @v_Country_Name VARCHAR(50);
      DECLARE @v_Country_Code VARCHAR(3);
      DECLARE @v_Language_Category VARCHAR(10);
      DECLARE @v_New_EEC_Id INT;
      DECLARE c_Get_Countries CURSOR FOR
      SELECT c.Country_Name, c.Country_Code, cl.Language_Category
      FROM Countries_Languages cl INNER JOIN Languages l
          ON (l.Language_Id = cl.Language_Id)
      INNER JOIN Countries c
          ON (c.Country_Id = cl.Country_Id)
WHERE UPPER(l.Language_Name) = UPPER(@p_Language_Name);
BEGIN
      SET @v_New_EEC_Id = 1;
      IF @p_Language_Name = ‘English’
          DELETE English_European_Countries;
      ELSE IF @p_Language_Name = ‘French’
          DELETE French_European_Countries;
      OPEN c_Get_Countries
      FETCH NEXT FROM c_Get_Countries
      INTO @v_Country_Name, @v_Country_Code, @v_Language_Category;
      WHILE (@@FETCH_STATUS = 0)
      BEGIN
          IF @p_Language_Name = ‘English’
              INSERT INTO English_European_Countries (English_CL_Id, Country_Code, Country_Name,
Language_Category)
              VALUES (@v_New_EEC_Id, @v_Country_Code, @v_Country_ Name, @v_Language_Category);
          ELSE IF @p_Language_Name = 'French'
              INSERT  INTO  French_European_Countries  (French_CL_Id,  Country_Code,  Country_Name,
Language_Category)
          VALUES (@v_New_EEC_Id, @v_Country_Code, @v_Country_Name, @v_Language_Category);
          SET @v_New_EEC_Id = @v_New_EEC_Id + 1;
      FETCH NEXT FROM c_Get_Countries INTO @v_Country_Name, @v_
Country_Code, @v_Language_Category;
      END
      CLOSE c_Get_Countries;
      DEALLOCATE c_Get_Countries;
END
GO
```

You may see how the data is manipulated atomically and procedurally, row by row. The programmer completely excludes the holistic manipulation of the data set from his logic. He may be aware of the data set because the query that defines the cursor is the data set. However, the developer cannot understand or even worse, he cannot accept that he can handle the data set as a whole with SQL and he is trying to divide it right away and by default. The application developer considers that the row is the only thing he should take into consideration when writing his code. He does not try to analyze and he does not question himself if he can solve the problem in a holistic manner, he is prepared by default to divide everything in rows, at the lowest level of detail. This style of development is a consequence of his classic and typical vision against programming and of his decision to ignore the fact that he is developing now in a specific, data-oriented environment, where the concept of data set should be incorporated into what he already knows, his structured model of programming.

It is true that sometimes we need to solve things atomically. There are business and technical situations where the division is required because some problems cannot be simply solved at the data set level and the division from data set to data row is necessary. Still, very often most of our tasks can be solved holistically by affecting everything as a whole and not piece-by-piece or row by row. This makes the big difference between someone thinking holistically, between a SQL oriented developer and someone thinking atomically, procedural, non-SQL. An application developer is used to handle things in a certain way and this is the first example. One can see how the developer declares those variables, he is declaring the cursor; he is opening it and stores the data row-by-row in an atomic manner, see how he is generating the new identifier in the classical style by incrementing the next value in the cursor. Let's imagine that you have a replication system or a data migration system, that type of specific application I was talking about earlier. Let's imagine you are moving medium to large quantities of data from different sources to different targets and imagine you are following the atomic approach. The first one should suffer is the performance. You can add thousands of indexes; you can do whatever you want. You will not solve too much of the issue. The issue is the improper style of programming.

The programmer thinks atomically and normally he shouldn't! The programmer is using the procedural language, the procedural advanced facilities like cursors, records and others by default. The programmer is following the atomic approach in almost any circumstance even if he may complete his task at the holistic level by using the SQL language and the set-oriented style of development.

Let’s see the holistic, SQL approach for the same example. Let’s notice the differences in terms of readability and simplicity, see how clear the code is in the holistic approach comparing with the code in the atomic approach. Let’s see the Oracle version of the full transfer, the holistic approach.

```
Code example 08: Oracle Holistic Full transfer
CREATE PROCEDURE Holistic_Full_Transf_Country
(
      p_Language_Name VARCHAR
)
AS
BEGIN
      DELETE English_European_Countries
      WHERE p_Language_Name = ‘English’;
      DELETE French_European_Countries
      WHERE p_Language_Name = ‘French’;
      INSERT INTO English_European_Countries (English_CL_Id, Country_Code, Country_Name, Language_Category)
      SELECT ROW_NUMBER() OVER (ORDER BY c.Country_Code, cl.Language_Category) AS English_CL_Id, c.Country_Code,
      c.Country_Name, cl.Language_Category
      FROM Countries_Languages cl INNER JOIN Languages l
          ON (l.Language_Id = cl.Language_Id)
      INNER JOIN Countries c
          ON (c.Country_Id = cl.Country_Id)
      WHERE l.Language_Name = p_Language_Name
AND p_Language_Name = ‘English’;
      INSERT INTO French_European_Countries (French_CL_Id, Country_Code, Country_Name, Language_Category)
      SELECT ROW_NUMBER() OVER (ORDER BY c.Country_Code, cl.Language_Category) AS French_CL_Id, c.Country_Code,
      c.Country_Name, cl.Language_Category
      FROM Countries_Languages cl INNER JOIN Languages l
          ON (l.Language_Id = cl.Language_Id)
      INNER JOIN Countries c
          ON (c.Country_Id = cl.Country_Id)
      WHERE l.Language_Name = p_Language_Name
AND p_Language_Name = ‘French’;
      COMMIT;
END Holistic_Full_Transf_Country;
/
```

Let’s just try, compare, and see the simplicity of the holistic approach. This Oracle stored procedure contains only SQL statements, no procedural instructions. The stored procedure contains two delete statements and two insert statements and that’s all we have! The logic moves the data in data sets, in waves, as I like to say. The entire transfer is seen as one transfer and there was no need for any movement to the row level. Let’s see now the SQL Server version of the holistic approach and then analyze with more details.

```
Code example 09: SQL Server Holistic Full transfer
CREATE PROCEDURE Holistic_Full_Transf_Country
```

```
(
      @p_Language_Name VARCHAR(50)
)
AS
BEGIN
      DELETE English_European_Countries
      WHERE @p_Language_Name = ‘English’;
      DELETE French_European_Countries
      WHERE @p_Language_Name = ‘French’;
      INSERT   INTO   English_European_Countries   (English_CL_Id,   Country_Code,   Country_Name,
Language_Category)
      SELECT ROW_NUMBER() OVER (ORDER BY c.Country_Code, cl.Language_Category) AS English_CL_Id,
      c.Country_Code, c.Country_Name, cl.Language_Category
      FROM Countries_Languages cl INNER JOIN Languages l
          ON (l.Language_Id = cl.Language_Id)
      INNER JOIN Countries c
          ON (c.Country_Id = cl.Country_Id)
      WHERE l.Language_Name = @p_Language_Name AND @p_Language_Name = ‘English’;
      INSERT   INTO   French_European_Countries   (French_CL_Id,   Country_Code,   Country_Name,
Language_Category)
      SELECT ROW_NUMBER() OVER (ORDER BY c.Country_Code, cl.Language_Category) AS French_CL_Id,
      c.Country_Code, c.Country_Name, cl.Language_Category
      FROM Countries_Languages cl INNER JOIN Languages l
          ON (l.Language_Id = cl.Language_Id)
      INNER JOIN Countries c
          ON (c.Country_Id = cl.Country_Id)
      WHERE l.Language_Name = @p_Language_Name
AND @p_Language_Name = ‘French’;
END
GO
```

Let’s analyze the logic in steps as we did for the atomic versions:

1. Delete the table with English or French languages, based on the parameter’s value. The deletion is holistic: all the countries for the respective language are deleted.
2. Insert the English or French countries, based on the value of the parameter, in one single instruction and in a holistic manner: all the countries for the respective languages are added and the new identifier is generated holistically using the function row number. Both steps are set based and holistic. Either 2 countries, or 10 countries, or 100 or 1000 countries, all of them are added in one single statement. We have one set of countries, we do not care how many are within the data set, we visualize the set of countries and we do not care about any detail. We do not care at all!

I believe the difference in terms of simplicity is obvious. Moreover, also in terms of portability things should be clear now with this example. The only difference between Oracle and SQL Server resided in the naming convention for variables because in SQL Server the “@” sign is required! I hope I clarified this advantage of the holistic approach: portability.

The versions of stored procedures are almost identical! For the procedural approach, although similar, there are differences in the specific syntax. You need to know the specific syntax for SQL Server, the procedural language of Transact SQL or you need to know the procedural language for Oracle, you need to know PL SQL. These procedural languages are similar in some aspects, more different in others. There is no standard here, just the same principles of structured programming. Although I consider that is not a difficult task to learn a new programming database language especially when you know another one, it is some work to be done! For the holistic approach, things are much simpler. The standard is followed by all the vendors inside their programming languages. The SQL from SQL Server and the SQL from Oracle are almost the same. Even if different, the differences may be accommodated easily.

Dear reader, I advise you to try and to solve similar exercises in both manners! Please try to identify the data set, affect it in a holistic manner, and try to divide the data set into rows, use the cursors and complete the task using the atomic and procedural approach. Apart from the fact that the code looks so different, apart from the increased complexity of the procedural approach, in most of the cases, you can check the performance when you have medium to large quantities of data. It is obvious that, for 2-3 rows you will not see a big difference in performance!

The first thing that you need to do when implementing a practice in the two ways is to check the results. You need to check the results after the execution in the atomic approach, save the results, go back to the starting conditions and implement the solution using the holistic approach. Compare the results and, if there are differences, try to look and see why. If the logic is well written, there should be no differences. Do that for this example and compare the content of the destination table after both executions, in both systems.

After you make sure the results sets are correct, you may check the performance. I was not able to see many cases where the performance in the database suffered due to the holistic approach versus the atomic approach. Of course, using the holistic approach, we still need to focus on optimizing the SQL itself and that means adding indexes, checking the execution plans to detect possible reasons for performance issues, checking statistics etc. More than that, even the SQL itself can be improved. Writing SQL instead of writing procedurally is the first condition for a good performance. The second condition is to know how to write the best SQL. In addition, for that, a good database developer will take the SQL and rewrite it repeatedly until he has its best code. After that he can apply additional facilities like all kinds of indexes, add materialized views in a static environment like a data warehouse and so many others. All these performance facilities are useless if are applied to the atomic and procedural code. The atomic and procedural code itself will generate a poor performance not the lack of indexes, statistics and the rest of the things that we need to gather during the lifetime of any database.

Normally, in my opinion, any programmer should be aware of the holistic approach. No one should write atomically in the database. At least not simple problems as the ones mentioned here. They should not be solved atomically. Afterwards, if the performance still suffers, we need true database developers that will be able to rewrite the holistic code and improve it. For that, indeed, a specialized SQL developer is required, a programmer that really likes SQL and has a deep understanding of the relational model and SQL. However, before that, before his intervention, the use of a holistic approach is enough for any programmer. This is my vision and I am not afraid to state it here.

If you analyze the holistic approach and compare it with the atomic approach, you will see how the degree of simplicity is obvious in the favor of the holistic approach. To be able to see that, if you are an application developer, you need to try to be objective and you also need to try to free yourself a bit from your paradigms and models, you should try to be able to forget and ignore the classic style of development you are caught in.

Now let's look at the holistic approach and see how the procedural code was completely avoided: even the if-else was excluded from the logic! I removed the if-else and replaced it with the condition for the parameter to be English or French especially to see that, at the limit, we can minimize the use of procedural code completely. Especially in a replication system written in SQL, an "ETL" or a data migration system, this is a great advantage in terms of performance, the data movement to be done in data sets, the fastest possible way!

Working set based, holistically and working SQL is the most suitable style within a relational database, it is the authentic style of a database developer.

Building a specific data migration system in the holistic manner requires having good SQL professionals with a perfect understanding of the data. That means firstly a good understanding of the concept of data set. Some developers that think SQL are more valuable than anyone within this context. You can build a replication system in pure SQL instead of using whatever tool! I believe that very often the use of tools can be avoided.

Let's go back. This first example was relatively easy. This is the general rule: one should always start with a simple example. Maybe some application developers will not follow the atomic approach in the previous example. Although some of them could follow the atomic path, maybe not being aware of the row number function for example. Let's increase the complexity and see more examples to illustrate the two styles of programming.

We just saw one facility that works with the set-based approach: the row number function. Apart from the simple copy example, one other requirement was to dynamically generate an artificial identifier for the key in the target. The tendency for the application developer is clear. A lot of them believe that they need to manipulate data row-by-row to generate the artificial identifier. They cannot imagine that the database system has a set of operators and functions dedicated to set-based approach that will allow them to avoid the row-by-row data manipulation. The only thing that they need to do is to open a google and search for the available features in the database system they are working with. This is what they are doing every! Everything is available now in the Internet, but the developers need to be aware of the set-based approach, they need to be aware that there are a variety of features for the set based approach that will help them working holistically and avoid the row division.

HOLISTIC VERSUS ATOMIC: INCREMENTALLY UPDATE A TARGET

I continue using the same design. We are in the same data migration system and we are moving data from a production system, where all the countries and languages are stored in one place, into a destination system where the data is organized per language. We will increase the complexity of the data transfer and we will assume that the data is not completely erased before the transfer. The data transfer will be transferred **incremental**, only the changes will be applied to the target.

Please look at the data to understand the exercise. Let’s assume we have three changes that occurred in the source tables. For Malta, the English language is becoming a secondary language instead of principal (MAIN), we completely delete the English language from Switzerland and we add the English language as a secondary language for Austria. We also add a new country Algeria, a new language Algerian Arabic. This is the main language in Algeria and the French language is secondary. Therefore, there are changes for both English and French languages, and these should be recorded incrementally now.

```
Code example 10: Change the sources
UPDATE Countries_Languages
SET Language_Category = ‘SECONDARY’
WHERE Country_Id = 4 AND Language_Id = 2;
DELETE Countries_Languages
WHERE Country_Id = 6 AND Language_Id = 2;
INSERT INTO Countries_Languages (CL_Id, Country_Id, Language_ Id, Language_Category)
VALUES (14, 2, 2, ‘SECONDARY’);
INSERT INTO Languages (Language_Id, Language_Name)
VALUES (7, ‘Algerian Arabic’);
INSERT INTO Countries (Country_Id, Country_Code, Country_ Name, Continent)
VALUES (10, ‘Ag’, ‘Algeria’, ‘Africa’);
INSERT INTO Countries_Languages (CL_Id, Country_Id, Language_ Id, Language_Category)
VALUES (15, 10, 7, ‘MAIN’);
INSERT INTO Countries_Languages (CL_Id, Country_Id, Language_ Id, Language_Category)
VALUES (16, 10, 3, ‘SECONDARY’);
```

Obviously, things are more complicated now! We will not simply delete the target table anymore, either English or French reporting table. We should try to keep the reporting tables syncronized with the source tables and apply the changes. More than that, we need to be able to syncronize deletions. Whenever we have a deleted country from the sources we need to remove it from the reporting table. When we have a new country in the sources we need to add it in the target and when we are changing something in the sources we need to update the information in the target. This is the incremental data transfer, our second example.

One should notice the business key apart from the primary key in the source table Countries Languages. The business key is composed of the columns Country Code and Language Category. The role of the artificial key is just for unique identification and the role for the unique business key is to be sure that any unique row has relevance from the business point of view.

One country may have one category at a time. This one is subject to change, as it happened in the Malta case. Based on the language category we can detect the changes, using either the atomic approach or the holistic approach. We can check for the differences one by one using the atomic approach or we can check for the differences for all the countries using the holistic approach.

We will solve this exercise by deleting the non-synchronized values and adding the list of new values. We can try to do that by thinking atomically and defining two cursors: one for the deletions of English countries that were changed in the source that will be applied firstly and the second one for the new values. This should be the atomic approach. We imagine one separate procedure for the English language.

Code example 11: Oracle Atomic Inc transfer

```
CREATE PROCEDURE Atomic_Inc_Transfer_English
AS
      v_Country_Name VARCHAR (50);
      v_Country_Code VARCHAR (3);
      v_Language_Category VARCHAR (10);
      v_Next_EEC_Id INT;
      v_Count INT;
      CURSOR c_Existing_Countries IS
      SELECT Country_Code, Language_Category
      FROM English_European_Countries;
      CURSOR c_New_Countries IS
      SELECT c.Country_Name, c.Country_Code, cl.Language_Category
      FROM Countries_Languages cl INNER JOIN Languages l
          ON (l.Language_Id = cl.Language_Id)
      INNER JOIN Countries_1 c ON (c.Country_Id = cl.Country_Id)
      WHERE l.Language_Name = ‘English’;
BEGIN
      OPEN c_Existing_Countries;
      LOOP
          FETCH c_Existing_Countries INTO v_Country_Code, v_ Language_Category;
          EXIT WHEN c_Existing_Countries%NOTFOUND;
          SELECT COUNT(1) INTO v_Count
          FROM Countries_Languages cl INNER JOIN Languages l
              ON (l.Language_Id = cl.Language_Id)
          INNER JOIN Countries_1 c
      ON (c.Country_Id = cl.Country_Id)
          WHERE l.Language_Name = ‘English’
          AND c.Country_Code = v_Country_Code
      AND cl.Language_Category = v_Language_Category;
          IF (v_Count = 0) THEN
              DELETE English_European_Countries
              WHERE Country_Code = v_Country_Code
AND Language_Category = v_Language_Category;
          END IF;
          COMMIT;
      END LOOP;
      CLOSE c_Existing_Countries;
      OPEN c_New_Countries;
      LOOP
          FETCH c_New_Countries INTO v_Country_Name, v_Country_ Code, v_Language_Category;
          EXIT WHEN c_New_Countries%NOTFOUND;
          SELECT COUNT (1) INTO v_Count FROM English_European_Countries eec
          WHERE NOT EXISTS
          (
          SELECT 1
          FROM English_European_Countries eec1
          WHERE eec1.English_CL_Id = eec.English_CL_Id AND eec1. Country_Code = v_Country_Code
          AND eec1.Language_Category = v_Language_Category
          );
          IF (v_Count = 1) THEN
          SELECT MAX (English_CL_Id) + 1 INTO v_Next_EEC_Id FROM English_European_Countries;
          INSERT  INTO  English_European_Countries  (English_CL_Id,  Country_Code,  Country_Name,
Language_Category)
          VALUES (v_Next_EEC_Id, v_Country_Code, v_Country_Name, v_ Language_Category);
          END IF;
          COMMIT;
      END LOOP;
      CLOSE c_New_Countries;
END Atomic_Inc_Transfer_English;
/
```

Please look and see the way the problem was solved in this example! The logic looks so professional! Let us analyze it:

1. We declare two cursors, one for storing the countries that should be deleted and another one to store the list of new countries.
2. We use the unique business key composed of Country Code and Language Category and store the pair in two variables, row by row. The values are taken from the target table English European countries.
3. We check in the source system, country by country and filter per the combination country code and language category, for the English language, of course. We calculate the count and add it in a dedicated variable.
4. If the count is zero, the data does not exist in the source system anymore, so it should be deleted from the target system.
5. Secondly, we are opening the second cursor from the source system.
6. We store the required information in the dedicated variables: country name, code and the category of the language.
7. We check in the reporting table against the sources to see if we have new countries. We rely in the business key of course. We calculate another count to see if we have something.
8. If the respective count is one, we consider the country as new and we add it to the reporting table.

This task could have been accomplished in different ways in the atomic approach; this is just a scenario and one solution to the problem. The important aspect here is to see the atomic approach in action, to see how somebody can solve things atomically. One imagines everything per unit and programs everything row by row, in a clear atomic vision, so dear to many application developers that are not used to think SQL and holistically!

Let's see the SQL Server version of the atomic approach, very similar to the Oracle version, but still different due to the different nature of the programming language, another disadvantage of the atomic approach.

Using the atomic approach, which generally means using the procedural code, requires a better understanding of the procedural language. The logic may be completely different when using the atomic approach because the programming languages are distinct, even if quite similar.

Code example 12: SQL Server Atomic Inc transfer

```
CREATE PROCEDURE Atomic_Inc_Transfer_Country
(
      @p_Language_Name VARCHAR (50)
)
AS
      DECLARE @v_Country_Name VARCHAR (50);
      DECLARE @v_Country_Code VARCHAR (3);
      DECLARE @v_Language_Category VARCHAR (10);
      DECLARE @v_Next_EEC_Id INT;
      DECLARE @v_Count INT;
      DECLARE c_Get_New_Countries CURSOR FOR
      SELECT c.Country_Name, c.Country_Code, cl.Language_Category
      FROM Countries_Languages cl INNER JOIN Languages l
          ON (l.Language_Id = cl.Language_Id)
      INNER JOIN Countries c ON (c.Country_Id = cl.Country_Id)
      WHERE UPPER (l.Language_Name) = UPPER (@p_Language_Name);
      DECLARE c_Get_Existing_E_Countries CURSOR FOR
      SELECT Country_Code, Language_Category
      FROM English_European_Countries;
      DECLARE c_Get_Existing_F_Countries CURSOR FOR
      SELECT Country_Code, Language_Category
      FROM French_European_Countries;
BEGIN
      OPEN c_Get_Existing_E_Countries;
      FETCH NEXT FROM c_Get_Countries INTO @v_Country_Code, @v_ Language_Category;
WHILE (@@FETCH_STATUS = 0)
BEGIN
      SET @v_Count = (SELECT COUNT (1)
      FROM Countries_Languages cl INNER JOIN Languages l
          ON (l.Language_Id = cl.Language_Id)
      INNER JOIN Countries c ON (c.Country_Id = cl.Country_Id)
      WHERE UPPER(l.Language_Name) = UPPER(@p_Language_Name)
      AND c.Country_Code = @v_Country_Code
AND cl.Language_Category = @v_Language_Category
                              );
      IF (@v_Count = 0)
      BEGIN
```

```
            DELETE English_European_Countries
            WHERE Country_Code = @v_Country_Code AND Language_Category = @v_Language_Category;
        END;
        FETCH NEXT FROM c_Get_Countries INTO @v_Country_Code, @v_Language_Category;
        END
        CLOSE c_Get_Existing_E_Countries;
        DEALLOCATE c_Get_Existing_E_Countries;
        OPEN c_Get_New_Countries
        FETCH    NEXT    FROM    c_Get_New_Countries    INTO    @v_Country_Name,    @v_Country_Code,
@v_Language_Category;
WHILE (@@FETCH_STATUS = 0)
BEGIN
        SELECT @v_Count = COUNT (1)
FROM English_European_Countries eec
        WHERE NOT EXISTS
        (
               SELECT 1
               FROM English_European_Countries eec1
               WHERE eec1.English_CL_Id = eec.English_CL_Id
AND eec1.Country_Code = @v_Country_Code
AND eec1.Language_Category = @v_Language_Category
);
        IF (@v_Count = 1)
        BEGIN
               SET    @v_Next_EEC_Id    =    (SELECT    MAX    (English_CL_Id)    +    1    FROM
English_European_Countries);
               INSERT INTO English_European_Countries (English_CL_Id, Country_Code, Country_Name,
Language_Category)
               VALUES (@v_Next_EEC_Id, @v_Country_Code, @v_Country_Name, @v_Language_Category);
            END;
        FETCH NEXT FROM c_Get_New_Countries INTO @v_Country_Name,
        @v_Country_Code, @v_Language_Category;
        END
        CLOSE c_Get_New_Countries;
        DEALLOCATE c_Get_New_Countries;
        END;
GO
```

Now an application developer working in SQL Server is extremely happy! He did not understand PL SQL but he understands Transact SQL! Working in this style in the database may transform, indeed, the developer in a specialized developer in a certain language. He will not gain the benefits of the standard and he will restrict himself to that particular language. Is the programmer choice, after all!

Still, even so, if you take a closer look you will see a lot of similarities. Things are similar and different, if we want to conclude. Nevertheless, working holistically things are just similar, not so different…

Comparing the two stored procedures, Oracle and SQL Server versions, you can still see almost the same steps. Still, the while is a loop, the cursor is a cursor, the variable is a variable with or without the“@” sign, the fetch is a fetch, the “if” statement is the same statement. In the end, it is not difficult to write in one system or another, things are similar. However, the portability is not the main problem: that is the last problem in the agenda! The performance and the clarity of the code are my problems!

The holistic approach for this example is much simpler. We don’t need anything like cursors or variables here, all we need is to think holistically and understand that we have a set of data to affect with two actions.

Let’s see the SQL Server version for the holistic approach:

Code example 13: SQL Server Holistic Inc transfer

```
CREATE PROCEDURE Holistic_Inc_Transfer_Country
(
      @p_Language_Name VARCHAR (50)
)
AS
BEGIN
      DELETE English_European_Countries
      FROM English_European_Countries eec
      WHERE NOT EXISTS
      (
          SELECT 1
          FROM Countries_Languages cl INNER JOIN Languages l
ON (l.Language_Id = cl.Language_Id)
          INNER JOIN Countries c
ON (c.Country_Id = cl.Country_Id)
      WHERE UPPER (l.Language_Name) = UPPER (@p_Language_Name)
          AND eec.Country_Code = c.Country_Code
          AND eec.Language_Category = cl.Language_Category
      );
       INSERT INTO English_European_Countries (English_CL_Id, Country_Code, Country_Name,
       Language_Category)
      SELECT (SELECT MAX (English_CL_Id) AS Max_English_CL_Id
      FROM English_European_Countries) + ROW_NUMBER() OVER (ORDER
       BY c.Country_Code, cl.Language_Category) AS English_CL_Id, c.Country_Code,
      c.Country_Name, cl.Language_Category
      FROM Countries_Languages cl INNER JOIN Languages l
      ON (l.Language_Id = cl.Language_Id)
      INNER JOIN Countries c ON (c.Country_Id = cl.Country_Id)
      WHERE UPPER (l.Language_Name) = UPPER (@p_Language_Name)
      AND NOT EXISTS
      (
          SELECT 1 FROM English_European_Countries eec
          WHERE eec.Language_Category = cl.Language_Category
AND eec.Country_Code = c.Country_Code
      );
END;
GO
```

Let’s see the steps:

1. First, we need to delete the countries from the reporting table that are not

conform to the business key and were eliminated from the set of source tables. We have all the items that need to be removed in a single select statement. We can simply try to identify the rows to be removed in a simple select statement and then we will transform this in a delete statement.

2. Secondly, we need to add the new English European countries that do not exist in the reporting target table. We can simply generate an artificial key for the reporting key in the select statement using different SQL features.

Now we can compare the two approaches. We have one delete statement and one insert statement, that's all we have! First, we might need to check a select statement to identify the data that needs to be deleted, if we can afford to.

Let's assume we tested the data before. We eventually look at the data and try to understand the nature. We look to see if what we have is indeed what needs to be deleted and, afterwards, we transform this select statement into a delete statement.

We think in these terms. I have some countries in the reporting table with European countries that may be obsolete due to some changes in the source tables. Let's see these countries first! We have a set of countries, one single data set! I don't care and I don't see any reason to take country by country, I have a list of countries, a set of rows, and I want to identify this list. This list, once correctly identified, I take it and use it for the deletion. In the logic, I have one delete statement. In reality, we always think the delete action in two steps. I have a set of rows, I identify it and I change it into a delete.

Secondly, and similarly, I have another set of countries that needs to be added in the reporting table. I take the select statement and make sure I identify the correct set of new countries. I find a way to dynamically generate new artificial identifiers for the key in the reporting table. I don't have any reason to move the logic at the atomic level.

Let's analyze the Oracle logic in the holistic approach and compare with the SQL Server logic. We will be amazed to see how similar the logic is. Actually if we weren't inside a stored procedure and we would have executed everything in a SQL editor, we could copy the entire piece of code from Oracle and execute it in SQL Server or vice versa. So, before continuing, based on these two examples, let's compare the atomic approach with the holistic approach.

Look at the Oracle version.

Code example 14: Oracle Holistic Inc transfer

```
CREATE PROCEDURE Holistic_Inc_Transfer_Country
(
      p_Language_Name VARCHAR
)
AS
BEGIN
DELETE FROM English_European_Countries eec
WHERE NOT EXISTS
(
      SELECT 1
      FROM Countries_Languages cl INNER JOIN Languages l
          ON (l.Language_Id = cl.Language_Id)
      INNER JOIN Countries c
          ON (c.Country_Id = cl.Country_Id)
      WHERE UPPER (l.Language_Name) = UPPER (p_Language_Name)
      AND eec.Country_Code = c.Country_Code
      AND eec.Language_Category = cl.Language_Category
);
      INSERT INTO English_European_Countries (English_CL_Id,
      Country_Code, Country_Name, Language_Category)
      SELECT (SELECT MAX (English_CL_Id) AS Max_English_CL_Id
      FROM   English_European_Countries)   +   ROW_NUMBER()   OVER   (ORDER   BY   c.Country_Code,
cl.Language_Category) AS English_CL_Id, c.Country_Code, c.Country_Name, cl.Language_Category
      FROM Countries_Languages cl INNER JOIN Languages l
          ON (l.Language_Id = cl.Language_Id)
      INNER JOIN Countries c
          ON (c.Country_Id = cl.Country_Id)
      WHERE UPPER (l.Language_Name) = UPPER (p_Language_Name)
      AND NOT EXISTS
      (
          SELECT 1 FROM English_European_Countries eec
          WHERE eec.Language_Category = cl.Language_Category
          AND eec.Country_Code = c.Country_Code
);
END Holistic_Inc_Transfer_Country;
/
```

Let’s conclude this chapter. We just analyzed two simple examples of data transfer. The data transfer, the copy process from a source to a destination, is the most common type of task for the programmer when writing inside the database. The data is transferred from various sources to various targets and this is common in almost any software system. According to our examples, we firstly transfer the data in a simple full approach. That means the data was completely deleted first and replaced with the data from the sources. The second exercise increases the complexity and the data in the target was not deleted first. The data was incrementally updated, only the changes were applied to the target. In both scenarios, the artificial identifier in the target needed to be dynamically generated.

This data transfer problem can be solved in both ways according to the two visions and styles of development. The application developer will be tempted to open a cursor and to move the context and the transfer at the row level. Because he is not fully aware of the fact that he is in a database where the data is affected in data sets and he knows just the principles of structure programming, he imagines he can transfer everything at the row level. He transforms the logic, using the cursors, in row-oriented logic, he sees everything in his variables and he can work almost like in the use interface. The database developer is aware of the data set, he knows that the data should be handled in data sets and he always tries to identify it and affect the data set as a whole, in a holistic manner. He knows that there are many set based facilities in every database system for various things. For example, he uses the row number function to generate the identifier, and this function is applied per the entire data set. His logic is much simpler, is much condensed, is set-oriented, it has a much better performance, is portable because SQL is a standard and the logic is almost the same in any database system. The transfer occurs at the level of the entire data set and this requires a certain style of development that is not very similar with the style in the user interface, maybe is not so attractive for some programmers, but it is efficient and it is what it should be in a relational database.

Chapter 6

OTHERS ATOMIC FEATURES USED IN EXCESS

THE USE OF SCALAR FUNCTIONS – A CHALLENGE TO THE SET-BASED APPROACH

The cursor is the main facility that allows the application developer to move the entire context at the row level. From my point of view, when teaching the applications developers to work in the database, I would completely remove the cursors from their work agenda and forbid them to use them until they learn and **understand the concept of data set**. I would also remove the loop feature from their development activity and don't allow them to loop under any circumstances. Cursors with loops are the favorite tools for many application programmers who are tempted to use them continuously because they permit them to handle things atomically and they let them focus on their favorite classic paradigm outside a real database. Unfortunately, for them they are inside a database and, despite their desires and preferences, they should become aware of the difference.

What is unfair is the fact that the cursor is a great feature and it allows us to solve many problems in the atomic style when the holistic style is not satisfactory. There are many situations when we need them and when the data set cannot be handled as a whole and consequently we are forced to move the context from data set to the row level. Consequently, no one should assume that I reject cursors. On the contrary, I believe that the cursor is a great feature by itself; the abuse of cursors is a disaster.

To be honest, I like working with cursors! It is so interesting and challenging! The steps are very clear. Moreover, even here we start with the concept of data set because the cursor declaration is a data set. Declaring the cursor, opening it, moving intermediates columns or expressions from the cursor data set into the layer of variables, move through the loop from one row to another and do various manipulations, all these actions are challenging and exciting and first time I learned this feature I was very happy!

Like anybody else involved in this kind of business, many of the ideas developed here have been understood and clarified in time. Even now, while I am writing I continue to discover new things and to reveal and clarify another or myself in one matter. For example, the cursors and their mechanism are fascinating sometimes. Especially when trees are involved is so difficult to handle the logic and you need to use cursors, sometimes combine many cursors until you get the proper results. The use of cursors is a perfect feature for the atomic approach. It allows us to change the context from the data set to data row. The atomic approach itself is mandatory and critical and we cannot live without it. Only that should not be used as the default solution because is a backup solution. That is the difference. The atomic approach should be seen as a backup solution for the holistic approach within a relational database. It is similar to the use of antibiotics; no one can say that the antibiotics are bad - they add many years to the average lifespan for most of us. However, the use in excess of antibiotics is a bad practice and the doctors recommend avoiding the use of antibiotics unless they are necessary. The same way the excessive use of antibiotics will damage the body in time, the use of atomic approach will damage the databases in time.

The end users and the customers using the software applications are mainly focused on the accuracy of the results and they accept a bad performance as long as it is not lousy. The fact that most of the developers are application developers within the mixed teams and they are familiarized with the atomic approach, contribute to a software world where a lot of databases are built using the atomic style of development.

The application developer is a great fan of the cursors for another reason too. As we all know, one of the most important feature for an application developer is the function. The application developer loves functions and he is tempted to use them as much as he can. The combination between cursors, movement into variables and loops is perfect for an atomic style. In specific programming languages like PL SQL or Transact SQL or anything else, these functions, suitable for an atomic approach, are named scalar functions. A scalar function returns a certain value of a certain type like a string or number. The scalar functions are very suitable for the atomic approach because they can be applied per variables within loops and cursors. Similarly with stored procedures, if the application developer sees a store procedure like a scalar function returning a void he can understand that a stored procedure should be defined to act in one expression or value and this action can be executed atomically within a loop in a cursor.

So what are the connections between the cursor, the scalar function and the atomic approach? Well, it is very easy to point out. They are all related. The cursor and the loop allow us to go down from the data set to the data row. The scalar function accepts variables as parameters, it returns discrete values, and is a type of function that is executed per row. The scalar function is a row function; it is exactly what the application developer needs to be able to move the context of development from the data set to the row. When I say to move the context from the data set to the data row, I don’t mean that the application developer is necessarily aware of the context switch. In most of the cases, he is not aware of any data set and he is going down to the row level by instinct, automatically.

The principle of structured programming teaches us to create functions and to use them to divide the logic into smaller pieces. One of the first things the student learns in the University is the use of functions. He learns to create functions; he learns that the function will return something, in most of the cases and he learns to add parameters to the functions and recall them later in his logic. When one starts to work in the database, and sees the scalar functions, he makes the connection right away, with what he knows from his classic programming and he is happy: he found his favorite toy and he is glad he can use it in the database!

Of course, those functions are useful in the database too. More than that, we should know that not all the functions are scalar, per row. There are set-based functions like table functions, for example. There are alternatives in almost any database programming language. The function can be applied to a data set and it can return a data set. The function can be used holistically. Unfortunately, nothing compares to scalar functions for an application developer because those remind him of what he knows from the user interface. He is tempted to use scalar functions in excess, he combines these in cursors and loops and he imposes his atomic style in the database.

One of the most important principles in structured programming is the following: divide complex problems into simpler ones by using functions or procedures. This is a great principle and it is followed in database programming too. That doesn’t mean, in the name of that principle, to divide the problem at the data row level instead of trying to solve it at the data set level. Solving the problems at the data set level, as I consider it should be in database programming, it may generate a smaller number of routines as secondary consequence. However, this is not a tragedy. This principle is not valuable by itself but for the benefits, it brings to the software application. We are not in Ethics, Metaphysics or even in a science like Geometry where principles are categorically and should be respected under any circumstances and regardless of any consequences. We are in the practical world of software development and we are in the database. The use of functions is different here than in the user interface and the holistic approach generally means less and different routines, especially in the specific systems like a data migration interface between a system A and a system B.

If we look at the types of available functions, we can see that every database programming language has many types of functions. In SQL Server, there are table functions; some kinds of functions that will return a set of values in the form of an object called table variable. Similarly, there is the possibility to return some complex types in PL SQL like arrays or collections. This shows that all the database engines and their associated programming languages offer the option to use functions in a holistic approach. Some of these functions may be required and may be used. Very often though, the use of functions causes a bad performance and normally these should be generally avoided.

Especially the scalar functions are part of a serious performance issue if applied to large data sets inside cursors. This is a common practice for a lot of application developers that are coming and try to apply the style they are familiarized with in the database. Let’s imagine one set of values with some thousands of rows and you create a function and call that function in a cursor some thousands times. As an alternative, you can manipulate the entire data set using some simple SQL statements. This is something that is happening very often in these kinds of applications that should be set-oriented but are row-oriented. One of the rules I would state in these kinds of applications is this one: use scalar functions for system settings, if you want to identify one setting or another in some configuration tables. Do not use scalar functions for operation that involve data sets, try to avoid and to solve the problem at the level of the data set.

Try to identify the level of granularity required by your software application and, if is a set-oriented application, try to forget about scalar functions. In most of times, these will not be used because these functions apply to details and require the atomic level.

Still, when should we use scalar functions? There is one situation when scalar functions are excellent in the database. Let’s see a query, of a certain type. Let’s define the concept of scalar query, by analogy with a scalar function. A scalar query is that type of query that returns one single value of a certain type, like a string value or numeric one. Sometimes we should know that.

Very often, configuration tables are suitable for scalar queries if these will read from the configuration tables. The scalar queries are ideal for scalar functions and, especially for configurations tables, are excellent and should be used intensively in the database.

EXAMPLE: FILTER ENGLISH OR FRENCH EUROPEAN COUNTRIES, ONLY FOR FLAGGED AND MAIN COUNTRIES.

Let's see one more example, a more complex scenario for the Practice 1, a full data transfer between two systems. The same design applies but the conditions for the report table generation are not so simple anymore. The business and technical requirement may look like this, and now I realize that we can influence the solution, atomic or holistic, even while describing the problem from the beginning.

The Business (technical) request:

We need to generate the same report table, in the full approach like in the previous chapter with one difference. The data should be generated from the set of three source tables into the target-reporting table, either English or French, under certain conditions. If the language is the principal language or main, we need to check for the flag. If the flag is set to a positive value like one, then we will generate the English or French set of countries into the table.

We can already see from the business presentation of the request the structured, atomic vision and we can already anticipate the solution. To me, this solution seems so complicated, so difficult that I barely stop myself from laughing with tears! The application developer, the true application developer will decide to follow his dear classic structured philosophy, completely atomic and procedural. He likes functions so he is ready to build some nice and cozy functions, scalar of course. One may notice that the business (technical) description is clearly written in the atomic style. The problem has been defined atomically before any line of code was written anywhere!

The application developer will see things per business key, the combination between language and country. If one looks at the design of the table, he may see the business key composed from a language and a country corresponding to the unique constraint UQ_Language_Country. This is the key for his atomic approach because the application developer knows that he can have one set of attributes or characteristics per country and language. Therefore, his functions will be oriented per language and country and these fields will be the parameters for his scalar functions.

The first function will return the category of the language for the respective country. The category can be either principal (MAIN) or secondary (SECONDARY). This function is a get function, of course, and the function is get_category. The second function will return the flag (make_flag) for the same combination of language and country, the function will be, guess how: get_flag. Let's see the nice scalar functions!

Code example 15: SQL Server get scalar functions

```
CREATE FUNCTION get_category
(
      @p_language_id INT,
      @p_country_id INT
)
RETURNS VARCHAR(10)
BEGIN
      DECLARE @v_category VARCHAR(10);
      DECLARE @v_count INT;
      SELECT @v_count = COUNT(*) FROM countries_languages
      WHERE language_id = @p_language_id
AND country_id = @p_country_id;
      IF @v_count = 0
          SET @v_category = NULL
      ELSE
          SELECT @v_category = Language_Category FROM countries_languages
          WHERE language_id = @p_language_id
      AND country_id = @p_country_id;
      RETURN @v_category;
END
GO
CREATE FUNCTION get_flag
(
      @p_language_id INT,
      @p_country_id INT
)
RETURNS INT
BEGIN
      DECLARE @v_make_flag INT
      DECLARE @v_count INT
      SELECT @v_count = COUNT(*) FROM countries_languages
      WHERE language_id = @p_language_id
AND country_id = @p_country_id;
      IF @v_count = 0
          SET @v_make_fllag = NULL
      ELSE
          SELECT @v_make_fllag = Make_Flag
FROM countries_languages
          WHERE language_id = @p_language_id
AND country_id = @p_country_id;
      RETURN @v_make_fllag;
END
GO
```

These functions will specify, for each language and country, the value for the flag and category, main or secondary. These functions will allow the developer to identify the conditions specified in the text (see the business requirement). The assumption when building these functions is clearly atomic and procedural, the programmer already sees one country and one language: **he sees himself as a rider on the row! Unfortunately, the row is not a horse but a donkey!**

The house for the scalar functions will be the cursor, of course. Now, when the developer is ready for the final call, he will use a similar logic as the one in previous chapter, but now will be even better because he can use his dear scalar functions and call them in in the cursor, so the database will be the mirror of what he knows from his classic development. Let’s see the atomic style for the SQL Server.

Code example 16: SQL Server Atomic Inc transfer with functions

```
CREATE PROCEDURE Atomic_Transfer_Country_Flag
(
@p_Language_Name VARCHAR (50)
)
AS
      DECLARE @v_Country_Name VARCHAR (50),
@v_Country_Code VARCHAR (3);
      DECLARE @v_Language_Category VARCHAR (10),
@v_New_EEC_Id INT;
      DECLARE @v_Country_Id INT, @v_Language_Id INT,
@v_Make_Flag INT;
      DECLARE c_Get_Countries_Lang CURSOR FOR
      SELECT Country_Id, Language_Id
      FROM Countries_Languages
      WHERE Language_Id IN (SELECT Language_Id FROM Languages
      WHERE Language_Name = @p_Language_Name);
BEGIN
      SET @v_New_EEC_Id = 1;
      IF @p_Language_Name = ‘English’
          DELETE English_European_Countries;
      ELSE IF @p_Language_Name = ‘French’
          DELETE French_European_Countries;
      OPEN c_Get_Countries_Lang
      FETCH NEXT FROM c_Get_Countries_Lang
      INTO @v_Country_Id, @v_Language_Id;
      WHILE (@@FETCH_STATUS = 0)
      BEGIN
          SET @v_Language_Category = dbo.get_category(@v_Language_ Id, @v_Country_Id);
          IF @v_Language_Category = ‘MAIN’
          BEGIN
              SET @v_Make_Flag = dbo.get_flag (@v_Language_Id, @v_ Country_Id);
              IF @v_Make_Flag = 1
              BEGIN
                    SELECT @v_Country_Name = Country_Name, @v_Country_Code = Country_Code FROM
countries
                    WHERE Country_Id = @v_Country_Id;
                    IF @p_Language_Name = ‘English’
                            INSERT INTO English_European_Countries (English_ CL_Id, Country_Code,
Country_Name, Language_Category)
                            VALUES (@v_New_EEC_Id, @v_Country_Code, @v_Country_ Name,
@v_Language_Category);
                    ELSE IF @p_Language_Name = ‘French’
                            INSERT INTO French_European_Countries (French_CL_ Id, Country_Code,
Country_Name, Language_Category)
                            VALUES (@v_New_EEC_Id, @v_Country_Code, @v_Country_ Name,
@v_Language_Category);
                    SET @v_New_EEC_Id = @v_New_EEC_Id + 1;
              END
          END
          FETCH NEXT FROM c_Get_Countries_Lang INTO @v_Country_Id, @v_Language_Id;
      END
      CLOSE c_Get_Countries_Lang;
      DEALLOCATE c_Get_Countries_Lang;
```

```
END;
GO
```

Let’s take a deep breath and let’s analyze this logic, written in a very classic style and in full compliance with the business description of the problem. The business analyst that describes the problem was already under the influence of the atomic approach. This is not an excuse for the developer because he can use his own mind and he can correctly interpret the statements.

The steps are:

1. Declare the cursor with all the combinations language country. Open it. Fetch the identifiers for both country and language. These will be used as parameters for the scalar functions.
2. Initialize the value for the artificial identifier.
3. Delete the reporting table, either English or French.
4. Calculate the category for the language and country, using the function get_category. The parameters are taken from the variables generated from the cursor.
5. If the category is MAIN, continue the logic in the most pure procedural style and calculate the flag using the second function, get_fag.
6. If the flag is positive (value 1) generate the data in the reporting table.
7. You can see after the execution two countries Malta and New Zealand for English European countries and Canada for French European countries.

Let’s see the logic in Oracle, and start with the functions. As you can see, things are similar; of course, the procedural syntax differs but not so much to affirm that it’s difficult to accommodate one from the other.

Code example 17: Oracle get scalar functions

```
CREATE OR REPLACE FUNCTION get_category
(
      p_language_id INT,
      p_country_id INT
)
RETURN VARCHAR2
AS
      v_category VARCHAR2(10);
      v_count INT;
BEGIN
      SELECT COUNT(*) INTO v_count FROM countries_languages
      WHERE language_id = p_language_id
AND country_id = p_country_id;
      IF v_count = 0 THEN
          v_category := NULL;
      ELSE
          SELECT Language_Category INTO v_category
FROM countries_languages
          WHERE language_id = p_language_id
AND country_id = p_country_id;
      END IF;
      RETURN v_category;
END;
/
CREATE FUNCTION get_fllag
(
      p_language_id INT,
      p_country_id INT
)
RETURN INT
AS
      v_make_fllag INT;
      v_count INT;
BEGIN
      SELECT COUNT(*) INTO v_count FROM countries_languages
      WHERE language_id = p_language_id
AND country_id = p_country_id;
      IF v_count = 0 THEN
          v_make_flag := NULL;
      ELSE
          SELECT Make_Flag INTO v_make_fllag
FROM countries_languages
          WHERE language_id = p_language_id
AND country_id = p_country_id;
      END IF;
      RETURN v_make_flag;
END;
/
```

You may see a similar style in any system, a similar logic, similar characteristics and features. The programmer uses the same atomic style, the same mind and the same confusions old habits that cannot be changed, despite the necessities. Hello my dear developers: wake up! We are in the database, we are in a relational database, and it is a different world. Let’s see how the Oracle or PL SQL developer ends his logic in triumph!

```
Code example 18: Oracle atomic transfer
CREATE OR REPLACE PROCEDURE Atomic_Transfer_Country_Flag
(
p_Language_Name VARCHAR
)
AS
      v_Country_Name VARCHAR (50);
      v_Country_Code VARCHAR (3);
      v_Language_Category VARCHAR (10);
      v_New_EEC_Id INT;
      v_Country_Id INT;
      v_Language_Id INT;
      v_Make_Flag INT;
      CURSOR c_Get_Countries_Lang IS
SELECT Country_Id, Language_Id
FROM Countries_Languages
WHERE Language_Id IN (SELECT Language_Id FROM Languages
WHERE Language_Name = p_Language_Name);
BEGIN
      v_New_EEC_Id := 1;
      IF p_Language_Name = ‘English’ THEN
          DELETE English_European_Countries;
      ELSIF p_Language_Name = ‘French’ THEN
          DELETE French_European_Countries;
      END IF;
      OPEN c_Get_Countries_Lang;
      LOOP
          FETCH c_Get_Countries_Lang
INTO v_Country_Id, v_Language_Id;
          EXIT WHEN c_Get_Countries_Lang%NOTFOUND;
v_Language_Category := get_category(v_Language_Id, v_Country_Id);
              IF v_Language_Category = ‘MAIN’ THEN
          v_Make_Flag := get_flag (v_Language_Id, v_Country_Id);
                    IF v_Make_Flag = 1 THEN
                    SELECT Country_Name, Country_Code INTO v_Country_ Name, v_Country_Code
              FROM countries WHERE Country_Id = v_Country_Id;
                           IF p_Language_Name = ‘English’ THEN
                               INSERT   INTO   English_European_Countries   (English_   CL_Id,
Country_Code, Country_Name, Language_Category)
                           VALUES (v_New_EEC_Id, v_Country_Code, v_Country_
                           Name, v_Language_Category);
                           ELSIF p_Language_Name = ‘French’ THEN
                               INSERT   INTO   French_European_Countries   (French_   CL_Id,
Country_Code, Country_Name, Language_Category)
                           VALUES     (v_New_EEC_Id,     v_Country_Code,     v_Country_Name,
v_Language_Category);
              END IF;
              v_New_EEC_Id := v_New_EEC_Id + 1;
              END IF;
          END IF;
      COMMIT;
      END LOOP;
      CLOSE c_Get_Countries_Lang;
END;
/
```

As you can see, we have a very impressive logic and procedural design, according to the business definition. The principles of structured programming are satisfied and the application developer is extremely happy, he is home! It does not matter that his home does not mean a good performance in the database, that the number of lines of codes is triple and the logic is infinitely more complex, he is actually home.

The question is: whose home? For sure, the database is not their home and the application developers should learn the rules from the database house if they want to be welcomed guests. Maybe this home is not so pleasant for some of them but if the business requires their presence, they should try to learn and follow the rules.

Let’s go back to the example and let’s prepare the holistic solution for the exercise. Now we will see that even the business definition could be changed to be SQL oriented.

What do you think about that? Any programmer, before doing his development work, should gather the requirements. He can use the services of a specialized business analyst or he can try to prepare the requirements on his own. Many programmers are preparing their own requirements and few of them have the luck to work with specialized business analysts. Therefore, the first phase - the requirements, is very often done by the programmers. Even here, at this early stage, the vision can be atomic or holistic. Please read the business requirement some pages before, the business requirement written by an authentic application developer that is ignoring completely his presence in the database. Let's see the same business requirement written in a total different style.

The Business (technical) request:

We need to generate the same report table, in the full approach like in the previous chapter with one difference. The data should be generated from the set of three source tables into the target reporting table, either English or French, for the countries where the language is principal (MAIN) and for the countries with the positive flag (1).

Now the text looks like a simple query, don't you think? This is exactly what it is. We need to generate the English or French European countries for the ones with the flag set to one and category main. It is a simple insert select statement, and the difference in the logic is obvious. Let's see the holistic approach for Oracle.

```
Code example 19: Oracle Holistic Inc transfer new fiilter conditions CREATE OR REPLACE PROCEDURE
Holistic_Transfer_Country_Flag

(

      p_Language_Name VARCHAR

)

AS

BEGIN

      DELETE English_European_Countries

      WHERE p_Language_Name = 'English';

      DELETE French_European_Countries

      WHERE p_Language_Name = 'French';

          INSERT  INTO  English_European_Countries  (English_CL_Id,  Country_Code,  Country_Name,
Language_Category)

      SELECT ROW_NUMBER() OVER (ORDER BY c.Country_Code, cl.Language_Category) AS English_CL_Id,
c.Country_Code,

      c.Country_Name, cl.Language_Category

      FROM Countries_Languages cl INNER JOIN Languages l

          ON (l.Language_Id = cl.Language_Id)

      INNER JOIN Countries c ON (c.Country_Id = cl.Country_Id)

       WHERE   l.Language_Name   =   p_Language_Name   AND   p_Language_Name   =   'English'   AND
       cl.Language_Category = 'MAIN' AND cl.Make_ Flag = 1;

            INSERT  INTO  French_European_Countries  (French_CL_Id,  Country_Code,  Country_Name,
            Language_Category)

       ELECT ROW_NUMBER() OVER (ORDER BY c.Country_Code, cl.Language_ Category) AS French_CL_Id,
       c.Country_Code, c.Country_Name, cl.Language_Category

       FROM Countries_Languages cl INNER JOIN Languages l

ON (l.Language_Id = cl.Language_Id)

      INNER JOIN Countries c ON (c.Country_Id = cl.Country_Id)

      WHERE l.Language_Name = p_Language_Name

AND p_Language_Name = 'French'

AND cl.Language_Category = 'MAIN' AND cl.Make_Flag = 1;
```

```
      COMMIT;
END Holistic_Full_Transfer_Country;
/
```

This is the set based approach and the holistic style of development, the one that is recommended in the relational database and that should be used by every programmer, not only by specialized database developers. Let's examine and compare the two logics and see the difference in the style. In the holistic approach we have no procedural logic at all, just pure SQL.

Let's see the steps:

1. The countries for the language specified as parameter are deleted. There is no if-else statement and the language specifies just the filter condition. I intentionally removed the if-else to show that the procedural code can be avoided and replaced with the use of SQL.
2. The countries are populated for the English or French from the set of source tables. Similarly, the use of filter condition after the language parameter replaces the if-else statement.
3. Just to clarify, one language will be transferred based on the value of the parameter. The other will not be affected at all because the filter condition for the language will not be satisfied. The reason for this approach was to use that the procedural code is very often optional I also wanted to maximize the use of SQL and minimize the use of procedural code.

The data in the target will be generated easily and straightforward, in a holistic manner. The data set, depending on the value of the parameter, will be generated according to the language. The difference comparing with practice 1 in the previous chapter is that two new additional filter conditions were added to the logic. These will replace the scalar functions and the rest of additional procedural facilities. Regarding portability, see how the SQL server version is almost the same, these being others advantages of the holistic style.

```
Code example 20: SQL Server holistic transfer
CREATE PROCEDURE Holistic_Full_Transfer_Country
(
      @p_Language_Name VARCHAR (50)
)
AS
BEGIN
          DELETE English_European_Countries
      WHERE @p_Language_Name = 'English';
          DELETE French_European_Countries
      WHERE @p_Language_Name = 'French';
      INSERT   INTO   English_European_Countries   (English_CL_Id,   Country_Code,   Country_Name,
Language_Category)
      SELECT ROW_NUMBER() OVER (ORDER BY c.Country_Code, cl.Language_Category) AS English_CL_Id,
c.Country_Code,
      c.Country_Name, cl.Language_Category
      FROM Countries_Languages cl INNER JOIN Languages l
          ON (l.Language_Id = cl.Language_Id)
      INNER JOIN Countries c ON (c.Country_Id = cl.Country_Id)
      WHERE l.Language_Name = @p_Language_Name
      AND @p_Language_Name = ‘English’
      AND cl.Language_Category = ‘MAIN’ AND cl.Make_Flag = 1;
          INSERT  INTO  French_European_Countries  (French_CL_Id,  Country_Code,  Country_Name,
Language_Category)
      SELECT ROW_NUMBER() OVER (ORDER BY c.Country_Code, cl.Language_Category) AS French_CL_Id,
c.Country_Code,
      c.Country_Name, cl.Language_Category
      FROM Countries_Languages cl INNER JOIN Languages l
ON (l.Language_Id = cl.Language_Id)
      INNER JOIN Countries c ON (c.Country_Id = cl.Country_Id)
      WHERE l.Language_Name = @p_Language_Name
      AND @p_Language_Name = ‘French’
      AND cl.Language_Category = ‘MAIN’ AND cl.Make_Flag = 1;
END
GO
```

To conclude, imagine that the quantity of data is moderate to medium or large and imagine an atomic style. This scenario is not exotic; unfortunately, this is really happening more often that we expect in many databases all over the world. The use of scalar functions is one example of scenario to be avoided. The performance of scalar functions in cursors is very poor and normally they should be avoided, with the exceptions of true scalar functions, when applied to scalar data sets, returning exactly one row, like configuration data.

In the data migration interface that I designed, I have two scalar functions. These are used for getting the values for configuration data, which are highly static data and useful for the functionality of the data migration interface. In this case, because these configuration values actually drive the entire data migration / replication system, I used scalar functions because their meaning is the same like in any classic system. Still, apart from that scenario, I never used them because I don’t need them, they are inefficient and they require the intensive use of cursors.

A SIMPLE QUERY!

Let’s continue to illustrate the two development approaches within the database and take some more examples. Let’s see the context of a query. I know how important these examples are for any developer, experienced or not. Words and concepts are good, words and programming code is even better! Consequently, I will try to satisfy both of them.

The future practices will be based on a simple design composed from two tables: one will contain a list of products and one a list of the types for the products, with the associated foreign key. Let’s see the table design.

Code example 21 products and their types

```
CREATE TABLE Product_Types
(
Product_Type_Id INT CONSTRAINT Nn_Product_Type_Id NOT NULL,
Product_Type_Code VARCHAR (5)
CONSTRAINT Nn_Product_Type_Code NOT NULL,
Name VARCHAR (255) CONSTRAINT Nn_Product_Type_Name NOT NULL,
CONSTRAINT Pk_Product_Type_Id PRIMARY KEY (Product_Type_Id)
);
CREATE TABLE Products
(
Product_Id INT CONSTRAINT Nn_Product_Id NOT NULL,
Name VARCHAR (30) CONSTRAINT Nn_Product_Name NOT NULL,
Product_Code VARCHAR (5)
CONSTRAINT Nn_Product_Code NOT NULL,
Product_Description VARCHAR (255),
Make_Flag INT,
Product_Type_Id INT,
Default_Quantity INT,
CONSTRAINT Pk_Product_Id PRIMARY KEY (Product_Id),
CONSTRAINT Fk_Products_Product_Types
FOREIGN KEY (Product_Type_Id)
REFERENCES Product_Types (Product_Type_Id)
);
```

The table “products” contains the following columns:

1. The column product id is a unique and artificial product identifier, and this is the primary key too.
2. The column product name represents the name of the product.
3. The column product code represents the code of the product
4. The column product description represents the description of the product.
5. The flag called make flag can be either zero or one.
6. The type of the product is specified as a reference to the table with product types.
7. The default quantity for the product will be used later.

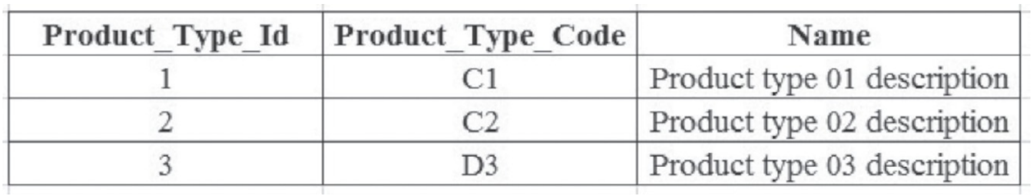
Let’s see some values and see the insert script:

Code example 22: Populate products and types

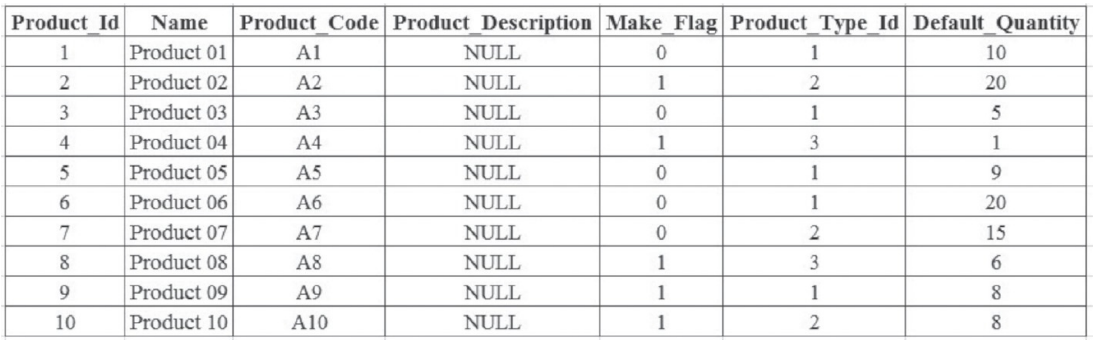
```
INSERT INTO Product_Types (Product_Type_Id, Product_Type_ Code, Name)
```

```
        VALUES (1, ‘C1’, ‘Product type 01 description’);
        INSERT INTO Product_Types (Product_Type_Id, Product_Type_ Code, Name)
        VALUES (2, ‘C2’, ‘Product type 02 description’);
        INSERT INTO Product_Types (Product_Type_Id, Product_Type_ Code, Name)
        VALUES (3, ‘D3’, ‘Product type 03 description’);
        INSERT INTO Products (Product_Id, Name, Product_Code, Make_ Flag, Product_Type_Id,
Default_Quantity)
        VALUES (1, ‘Product 01’, ‘A1’, 0, 1, 10);
        INSERT INTO Products (Product_Id, Name, Product_Code, Make_ Flag, Product_Type_Id,
Default_Quantity)
        VALUES (2, ‘Product 02’, ‘A2’, 1, 2, 20);
        INSERT INTO Products (Product_Id, Name, Product_Code, Make_ Flag, Product_Type_Id,
Default_Quantity)
        VALUES (3, ‘Product 03’, ‘A3’, 0, 1, 5);
        INSERT INTO Products (Product_Id, Name, Product_Code, Make_ Flag, Product_Type_Id,
Default_Quantity)
        VALUES (4, ‘Product 04’, ‘A4’, 1, 3, 1);
        INSERT INTO Products (Product_Id, Name, Product_Code, Make_ Flag, Product_Type_Id,
Default_Quantity)
        VALUES (5, ‘Product 05’, ‘A5’, 0, 1, 9);
        INSERT INTO Products (Product_Id, Name, Product_Code, Make_ Flag, Product_Type_Id,
Default_Quantity)
        VALUES (6, ‘Product 06’, ‘A6’, 0, 1, 20);
        INSERT INTO Products (Product_Id, Name, Product_Code, Make_ Flag, Product_Type_Id,
Default_Quantity)
        VALUES (7, ‘Product 07’, ‘A7’, 0, 2, 15);
        INSERT INTO Products (Product_Id, Name, Product_Code, Make_ Flag, Product_Type_Id,
Default_Quantity)
        VALUES (8, ‘Product 08’, ‘A8’, 1, 3, 6);
        INSERT INTO Products (Product_Id, Name, Product_Code, Make_ Flag, Product_Type_Id,
Default_Quantity)
        VALUES (9, ‘Product 09’, ‘A9’, 1, 1, 8);
        INSERT INTO Products (Product_Id, Name, Product_Code, Make_ Flag, Product_Type_Id,
Default_Quantity)
        VALUES (10, ‘Product 10’, ‘A10’, 1, 2, 8);
```

Let’s see the values for the products and their types:



Product_Type_Id	Product_Type_Code	Name
1	C1	Product type 01 description
2	C2	Product type 02 description
3	D3	Product type 03 description



Product_Id	Name	Product_Code	Product_Description	Make_Flag	Product_Type_Id	Default_Quantity
1	Product 01	A1	NULL	0	1	10
2	Product 02	A2	NULL	1	2	20
3	Product 03	A3	NULL	0	1	5
4	Product 04	A4	NULL	1	3	1
5	Product 05	A5	NULL	0	1	9
6	Product 06	A6	NULL	0	1	20
7	Product 07	A7	NULL	0	2	15
8	Product 08	A8	NULL	1	3	6
9	Product 09	A9	NULL	1	1	8
10	Product 10	A10	NULL	1	2	8

Everything starts with the business requirements. Now we have a simple design, some familiar products and their types. Let’s see what we want from this list of products, for the moment!

Business requirement:

We want to get a simple result set, a data set taken from the products table. We want to select the product identifier, the name for each product, the name for the previous product and the product code for the previous product if the flag is set to one otherwise the current product code. The concept of previous product applies based on the product identifier. For example, the previous product for product id 3 is product id 2 etc.

This exercise seems very suitable for the atomic approach as it looks like we need to access the data row-by-row and gather the information for the previous rows. This is a complex example and the use of atomic approach may be understandable. That does not mean it cannot be avoided but you need to have good eyes and a good SQL oriented mind to be able to see the holistic approach. For the moment, we will focus on the atomic approach that may seem unavoidable.

Any experienced application developer will start dividing everything into small pieces of code, will start imagining how will move the previous values into variables for each row etc. We are happy with the 10 products that we have, so there is no problem in working atomically from the performance point of view. However, this is actually a sample from a large table with, let's say one hundreds or thousands or more products. Things might change if we are digging into larger data set atomically from the beginning without questioning ourselves if there is no other way to solve the problem! We will see that there is, in spite of some evidences.

Most of the developers are familiarized with the atomic style and this is understandable to a certain point, because this style is the one that is used at the user interface level where we don't have the concept data set. Thinking SQL and thinking holistically requires a different vision and a different style. Still, the good news is that this style is simpler and the things are much easier in the holistic approach that offers us the great advantage of a better performance. The differences in performance between the two styles in a significant quantity and variety of data are impressive for anyone who wants to do some testing.

What you need to do, I repeat, is to start getting friend with the data. Try to query more, try to dig into the data and see more meanings, try to correlate the business with the data and try to see the business in the data. More than that, you can try to transfer your atomic vision in the new one, holistic. This means try to find the data sets from the data rows. You need to make an effort and you need to change your vision against data, actually better said, you need to start having a vision against the data. The atomic approach actually starts from a lack of vision against the data. Using the typical approach of development in the database starts from a total neglect of the most intimate resort of the database, the nature of data. The nature of data is holistic, set oriented.

Let's go back to our exercise. This exercise might be solved in many similar ways. I picked up one solution for the atomic approach and one solution for the holistic approach. Let's investigate the atomic approach, and let's see the code for SQL Server.

Code example 23 Get the products atomically, SQL Server.

```
CREATE PROCEDURE Get_Products_Atomically
```

```
AS
BEGIN
      DECLARE @v_Product_Id NUMERIC (10, 0), @v_Prev_Product_Code NVARCHAR (5);
      DECLARE @v_First_Name NVARCHAR (30), @v_First_Product_Code NVARCHAR (200)
      DECLARE @v_Current_Name NVARCHAR (30), @v_Min1_Product_Id NUMERIC (10, 0);
      DECLARE @v_Previous_Name NVARCHAR (30), @v_Product_Code NVARCHAR (200);
      DECLARE @v_Min_Product_Id NUMERIC (10, 0), @v_Make_Flag INT;
      DECLARE @v_Products TABLE(Product_Id INT, Current_Name NVARCHAR (30),
      Previous_Name NVARCHAR (30), Product_Code NVARCHAR (5));
      SELECT @v_Min_Product_Id = MIN (Product_Id) FROM Products;
      SELECT @v_Min1_Product_Id = MIN (Product_Id)
      FROM Products WHERE Product_Id > @v_Min_Product_Id;
      SELECT @v_First_Name = Name FROM Products
      WHERE Product_Id = @v_Min_Product_Id;
          SELECT @v_First_Product_Code = Product_Code
      FROM Products WHERE Product_Id = @v_Min_Product_Id;
      INSERT INTO @v_Products (Product_Id, Current_Name, Product_ Code)
      SELECT Product_Id, Name,
      CASE WHEN Make_Flag = 1 THEN NULL ELSE Product_Code END AS Product_Code
      FROM Products WHERE Product_Id = @v_Min_Product_Id;
      DECLARE c_Products CURSOR FOR
      SELECT Product_Id, Name, Product_Code, Make_Flag FROM Products
      WHERE Product_Id > @v_Min_Product_Id
      ORDER BY 1;
      OPEN c_Products;
      FETCH NEXT FROM c_Products
      INTO @v_Product_Id, @v_Current_Name, @v_Product_Code, @v_ Make_Flag;
      WHILE (@@FETCH_STATUS = 0)
      BEGIN
      IF (@v_Min1_Product_Id = @v_Product_Id)
      BEGIN
          SET @v_Previous_Name = @v_First_Name;
          IF (@v_Make_Flag = 1)
              SET @v_Prev_Product_Code = @v_First_Product_Code;
          ELSE
              SET @v_Prev_Product_Code = @v_Product_Code;
          INSERT INTO @v_Products (Product_Id, Current_Name, Previous_Name, Product_Code)
          SELECT @v_Product_Id, @v_Current_Name, @v_Previous_Name, @v_Prev_Product_Code
          SET @v_Previous_Name = @v_Current_Name;
      END
      ELSE
      BEGIN
              IF (@v_Make_Flag = 0)
                    SET @v_Prev_Product_Code = @v_Product_Code;
              INSERT INTO @v_Products (Product_Id, Current_Name, Previous_Name, Product_Code)
              SELECT @v_Product_Id, @v_Current_Name, @v_Previous_ Name, @v_Prev_Product_Code
              SET @v_Previous_Name = @v_Current_Name;
              SET @v_Prev_Product_Code = @v_Product_Code;
          END
          FETCH NEXT FROM c_Products
```

```
            INTO @v_Product_Id, @v_Current_Name, @v_Product_Code, @v_ Make_Flag;
        END;
        CLOSE c_Products;
        DEALLOCATE c_Products;
        SELECT * FROM @v_Products
END
GO
```

Let’s analyze the logic, step by step.

1. We store in two variables the product identifiers for the first product and the next product. See the variables @v_Min_Product_Id and @v_Min1_Product_Id.
2. We store the name and product code for the first product. See the variables @v_First_Name and @v_First_Product_Code.
3. We add the first product into the table variable see the table variable @v_Products.
4. We open the cursor with all the products apart from the first one. See the cursor named c_Products.
5. For the second product (the first one in the cursor, equal with the variable @v_Min1_Product_Id), we setup the previous name and we check the flag. If the flag is set to one, the previous code is the first product code. Otherwise, the previous code is the current product code. We add the data using the variables into the table variable and set the previous name to current name.
6. If is another product apart from the second one (first one in the cursor), we check the flag. If is set to zero, the previous product code will become the current product code. Add the data to the results table and setup the values for the @v_Previous_Name and @v_Prev_Product_Code.
7. Select the data from the table variable (@v_Products).

Let’s analyze the SQL Server solution for the atomic approach more deeply. The Oracle solution is similar but even more procedural!

As you can see it is quite logic! We are using a table variable and we are using a mixed solution, half-atomic and half-holistic. The table variable is firstly populated in a holistic manner, in a set-oriented style. Then, due to the complexity of the logic, a cursor is opened and the table is updated. This is a kind of **mixed approach** and it is better than a complete atomic approach that we could have used and that could have been used by the fans of classic development.

This is somehow a classic SQL Server style of development, according to which a table, either a table variable or a temporary table if larger, is manipulated several times until it gets the proper data. This approach is the mixed approach I was talking about and is better than a complete atomic approach. Sometimes it is even the best method. As an alternative, we could do everything in a cursor and handle all the things complete in the cursor in the most pure atomic style but I wanted to offer another paradigm, the mixed approach.

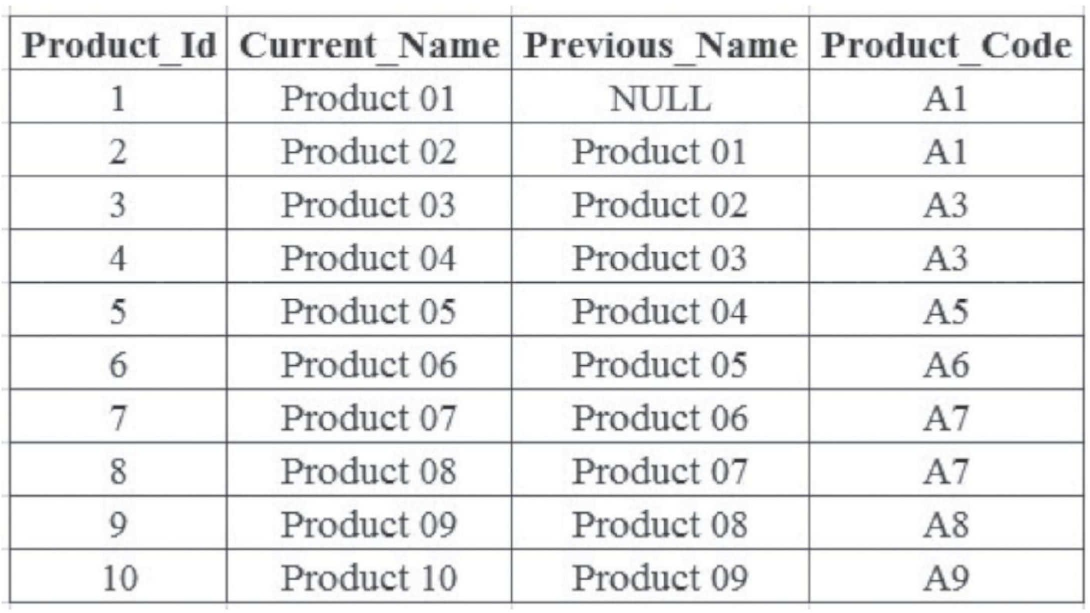
As you perhaps know, the temporary table is very common in SQL Server, not so much used in Oracle. The temporary table or even the variable of type table, or a record or a collection if we look into the garden of Oracle, or whatever kind of structure might be useful features and are part of the development. Sometimes these are to be combined in a mixed approach like in this example and things will be handled partially holistically in data sets and atomically afterwards. One solution is to partially generate a data set in a structure like a temporary table or an array and following a holistic style and populate that structure. Afterwards, we may need to update the data generated in the temporary table and need to follow an atomic approach and handle the rest of the logic atomically. This is what we did here.

This is an example of a semi classic procedural approach that is used to generate the results set according to the requirements. This is what most of the students know, this is what most of the application developers are doing in the database: structured programming and the use of the atomic development approach. The rows in the source table are taken in the cursor row by row, handled atomically: various conditions are checked against the variables stored row by row. In the Oracle version, the things are quite similarly with the difference that the objects are even more procedural than in SQL Server, using real structures (so called records) and arrays. The SQL Server table variable is a simpler structure and it is close to SQL, the Oracle types are more procedural. The meaning and the style are the same; the atomic approach is the same.

This is the general trend and this is the preferred style that is used in many databases, part of software applications despite the fact that we can use the other approach, native to the database, the holistic approach. We can identify situations where the atomic style needs to be used, because these kinds of situations do exist. However, very often, more often than we can imagine, the atomic approach and the procedural style can be avoided and replaced with the holistic approach and the pure SQL style of development.

The application developer is used to see things atomically and he has the tendency to divide everything atomically. He doesn’t necessarily see the data set, the set of rows to be affected, he sees variables and he tries to divide the logic in variables. He is tempted to open cursors almost every time and he is tempted to transfer the logic within the database in the application. In addition, this tendency brings logic within the database that is too sophisticated and to complicated. More than that, the relational model is too simple for the application developer and it builds things in a very complicated way.

Let’s see the results:



Product_Id	Current_Name	Previous_Name	Product_Code
1	Product 01	NULL	A1
2	Product 02	Product 01	A1
3	Product 03	Product 02	A3
4	Product 04	Product 03	A3
5	Product 05	Product 04	A5
6	Product 06	Product 05	A6
7	Product 07	Product 06	A7
8	Product 08	Product 07	A7
9	Product 09	Product 08	A8
10	Product 10	Product 09	A9

Let's see the solution for the holistic approach for this problem and analyze the solution. In this particular scenario, we do have a complete holistic approach. You can compare again the two approaches and you can see the differences. From the performance point of view, you can try to generate some larger data sets and compare the timing in both implementations. You will be amazed to see the differences.

The holistic approach is so simple comparing with the atomic approach! Let's see it first and then analyze it.

```
Code example 24 Get the products holistically, SQL Server.
CREATE PROCEDURE Get_Products_Holistically
AS
SELECT Product_Id, Current_Name, Previous_Name, Product_Code
FROM
(
      SELECT a.Product_Id, p_Current.Name AS Current_Name, p_
      Previous.Name AS Previous_Name,
      CASE WHEN p_Current.Make_Flag = 1 THEN p_Previous.Product_
      Code ELSE p_Current.Product_Code END
      AS Product_Code, 1 AS Type
      FROM
      (
          SELECT Product_Id, ROW_NUMBER () OVER (ORDER BY Product_ Id) Current_Row_No,
          ROW_NUMBER () OVER (ORDER BY Product_Id) -1 AS Previous_Row_No
          FROM Products
          ) a INNER JOIN Products p_Current ON (p_Current.Product_Id = a.Product_Id)
          INNER JOIN
          (
              SELECT Name, Product_Id, Product_Code, ROW_NUMBER () OVER (ORDER BY Product_Id)
Row_No FROM Products
          ) p_Previous ON (p_Previous.Row_No = a.Previous_Row_No)
          UNION
          SELECT TOP 1 Product_Id, Name AS Current_Name, NULL AS Previous_Name,
          CASE WHEN Make_Flag = 1 THEN NULL ELSE Product_Code END AS Product_Code, 0 AS Type
          FROM Products
          ORDER BY Type, Product_Id
```

```
 ) A
GO
```

What is that, someone might ask! Just one select statement for so many things, is that possible? The answer is yes, a simple select statement solves all the problems. There is no need for cursors, no need for records or table or temporary variables to populate etc. There is no need for an atomic approach, not even for a mixed approach. One simple select statement was enough.

First, please try and compare the results. For that, you should execute the store procedures in SQL Server and compare the results. You will see the same results because the logic is the same only that the approaches and styles are totally different.

This is not a simple example and not everybody knows how to do it, it is not a simple exercise. However, it is not impossible; an effort is required for understanding. What you need to do is to look at the data and analyze everything, you should be aware of the set based operators and functions like row number; you should try to imagine how you take the data from the previous row in a direct SQL and not in a cursor. In this scenario, most of the programmers would choose the atomic approach, even some database programmers. It looks like a case for a cursor and for manipulation. Still, this example proves that even practices seeming to require an atomic solution, can be solved holistically. You need to have enough imagination and you need to jump, mentally, from one row to another and get the data.

I always like to say that SQL can do miracles and this is a good example!

Chapter 7

WRITING SQL VERSUS WRITING PROCEDURALLY, OTHER HOLISTIC METHODS

WRITING SQL VERSUS WRITING PROCEDURALLY, ANOTHER IMPEDIMENT! AN EXAMPLE OF UPDATE!

I will continue with more examples following the same purpose, to illustrate the two styles of development. I will use the same simple design taken from an Inventory database, the design with the two tables that store products information: products and product types. If you will take a closer look at the data in the tables, you will notice that there is no description for the products. Our purpose in the following exercise is to update the description based on a certain algorithm. Let's see the description of the problem, a description written in the most pure atomic style and very clear for an application developer.

Business (technical) description:

We want to update the product description in the products table. Maybe you know that in the field of inventory (production) the products codes have their own significance. According to this, you can see an algorithm for the description based on the code of the type (field product type in the table Product_ Types) and the already familiar flag (field Make_Flag from the table Products). If the type of the code starts with C letter then we should look at the flag. If the flag is one, we need to concatenate the constant string value DESC plus the code of the product and the code of the type otherwise we should concatenate the constant DESC plus the code of the product plus the name of the product. If the type of the code is anything else but C we are looking at the flag and add the name of the product and the constant if is 1 otherwise we add the type of the code.

Looking at the above code, you can easily see that is very probable for the analyst or developer to thinks atomically and procedurally. From the business / technical description you can already derive the set of if-else statements, eventually the scalar functions, all of them in a common and popular nice cursor! This will be the atomic approach. I don't want to insist very much with the explanations, this exercise is similar to the others in the previous chapters.

The chosen solution, one of the many, will be described. The developer will create one function get_type with the parameter the product identifier. This procedure will return the type based on the flag, one of the two parts of the algorithms. Let's see the function in Oracle version.

```
Code example 25 Get product type Oracle
CREATE FUNCTION Get_Type
(
      p_Product_Id INT
)
RETURN VARCHAR2
```

```
AS
      v_Type VARCHAR2 (255);
      v_Make_Flag INT;
BEGIN
      SELECT Make_Flag INTO v_Make_Flag FROM Products WHERE Product_Id = p_Product_Id;
      IF (v_Make_Flag = 1) THEN
          SELECT t.Product_Type_Code INTO v_Type
          FROM Product_Types t INNER JOIN Products p
ON (p.Product_Type_Id = t.Product_Type_Id)
          WHERE p.Product_Id = p_Product_Id;
      ELSE
          SELECT t.Name INTO v_Type
          FROM Product_Types t INNER JOIN Products p
ON (p.Product_Type_Id = t.Product_Type_Id)
          WHERE p.Product_Id = p_Product_Id;
      END IF;
      RETURN (v_Type);
END;
/
```

For a given product, based on the value of the flag, we either generate the type's code from the product types table or the name of the product from the products table. This is a classic scalar function and the purpose is obvious, this function will be executed in a cursor for all the products later. This is one of the most common scenarios of work for an application developer.

After building this function, the logic will continue with the update procedure. Let's see it first and then analyze.

Code example 26 Update product description Oracle Atomic

```
CREATE PROCEDURE Upd_Products_Desc_Atomic
AS
      v_Product_Id INT;
      v_Product_Type_Code VARCHAR2 (5);
      v_Product_Code VARCHAR2 (5);
      v_Description VARCHAR2 (255);
      v_Generated_Type VARCHAR2 (255);
      v_rid ROWID;
      CURSOR c_Get_Products IS
      SELECT Product_Id, rowid FROM Products FOR UPDATE OF Product_ Description;
BEGIN
      OPEN c_Get_Products;
      LOOP
          EXIT WHEN c_Get_Products%NOTFOUND;
          FETCH c_Get_Products INTO v_Product_Id, v_rid;
          v_Generated_Type := Get_Type(v_Product_Id);
          SELECT p.Product_Code, t.Product_Type_Code INTO v_Product_ Code, v_Product_Type_Code
          FROM Products p INNER JOIN Product_Types t
ON (p.Product_Type_Id = t.Product_Type_Id)
          WHERE p.Product_Id = v_Product_Id;
          v_Description := 'DESC_';
          IF (SUBSTR (v_Product_Type_Code, 1, 1) = ‘C’) THEN
              v_Description := v_Description || v_Product_Code;
          END IF;
          IF SUBSTR (v_Description, LENGTH (v_Description), 1) <> ‘_’ THEN v_Description :=
v_Description || ‘_’;
          END IF;
          v_Description := v_Description || v_Generated_Type;
          UPDATE Products
          SET Product_Description = v_Description
          WHERE rowid = v_rid;
      END LOOP;
      COMMIT;
      CLOSE c_Get_Products;
END;
/
```

The steps are quite clear, let’s quickly review:

1. Declare the cursor for the table products with the option of updating the description, our goal.
2. Calculate the generated type using the function created before.
3. Use a string inside the cursor for the description and start concatenate to that string (v_Description).
4. Get the code of the product and the product code and product code type for the given product taken from the cursor.
5. If the first letter of the type code is the C letter, then add the product code to the description string.
6. Add the generated type and finalize the description for the given product inside the cursor.
7. Update the description with the calculated value of the description for the given product.
8. That will happen for all the products. Finally, the table will be updated.

This is the classic atomic and procedural approach, so familiar to application developers that can be followed instinctively by many new programmers in the database if they are not warned from the beginning (I might say starting with their college period) that they should write their code in a different way if they are inside a database. Things are so complicated compared with a simple update, one simple update! This is what we call the holistic approach, a simple update statement. If the developer will read the business description in a holistic manner and not in an atomic one, he will find the holistic solution that is actually trivial for a SQL developer.

I will not display the atomic function and update procedure for SQL Server; it is similar with the differences given by the languages. I would rather prefer to mention the holistic approaches for both Oracle and SQL Server for anyone to see how similar these are and to understand the second argument for the holistic approach: portability. So let’s see the holistic approach for Oracle:

```
Code example 27 Update product description Oracle Holistic
CREATE PROCEDURE Upd_Products_Desc_Holistic
AS
```

```
BEGIN
      UPDATE Products
      SET Product_Description = (SELECT
          CASE WHEN Products.Make_Flag = 1 AND SUBSTR (t.Product_ Type_Code, 1, 1) = ‘C’
              THEN ‘DESC_’ || Products.Product_Code || ‘_’ || t.Product_Type_Code
          WHEN Products.Make_Flag = 0 AND SUBSTR (t.Product_Type_ Code, 1, 1) = ‘C’
              THEN ‘DESC_’ || Products.Product_Code || ‘_’ || t.Name
          WHEN Products.Make_Flag = 0 AND SUBSTR (t.Product_Type_ Code, 1, 1) <> ‘C’
              THEN ‘DESC_’ || t.Name
          WHEN Products.Make_Flag = 1 AND SUBSTR (t.Product_Type_ Code, 1, 1) <> ‘C’
              THEN ‘DESC_’ || t.Product_Type_Code
          END
      FROM Product_Types t WHERE t.Product_Type_Id = Products. Product_Type_Id);
      COMMIT;
END;
/
```

This is the power of the “case” statement. This statement can successfully replace the “if-else” statement in a simple SQL. One can see how the simple update statement in the example above will do everything, generate the descriptions according to the algorithm and update the columns. Why do we need to complicate things using sophisticated classic methods instead of using the power of simple SQL language that is a dedicated language for relational databases?

One can easily understand the algorithm from the SQL statement defined in the holistic approach. He can see the conditions specified in the “when” clause of the case statement, nothing is secret or difficult in this update statement. Either a student or an application developer, when working in the database, one cannot act like not being there.

Let’s see the holistic approach for SQL Server, just to see that the two approaches are almost identical.

```
-- Code example 28 Update product description SQL Server Holistic
CREATE PROCEDURE Upd_Products_Desc_Holistic
AS
BEGIN
      UPDATE Products
      SET Product_Description =
          CASE WHEN dest.Make_Flag = 1
      AND SUBSTRING(t.Product_Type_Code, 1, 1) = ‘C’
              THEN ‘DESC_’ + dest.Product_Code + ‘_’ + t.Product_
Type_Code
              WHEN dest.Make_Flag = 0
AND SUBSTRING (t.Product_Type_Code, 1, 1) = ‘C’
              THEN ‘DESC_’ + dest.Product_Code + ‘_’ + t.Name
                    WHEN dest.Make_Flag = 0
AND SUBSTRING (t.Product_Type_Code, 1, 1) <> ‘C’
                           THEN ‘DESC_’ + t.Name
                    WHEN dest.Make_Flag = 1
AND SUBSTRING (t.Product_Type_Code, 1, 1) <> ‘C’
                    THEN ‘DESC_’ + t.Product_Type_Code
```

```
            END
        FROM Products dest INNER JOIN Product_Types t
  ON (t.Product_Type_Id = dest.Product_Type_Id);
  END
  GO
```

Again, I am sorry for repeating this: let’s look at the versions for the two procedures in Oracle and SQL Server! In most of the database systems, they are the same! One would see and understand the advantage of portability, and see how the code is very similar. Things are almost the same because SQL is almost the same. Of course, we are not always working for a software company that will handle the same software application using many database systems. However, we may change the project and we may move from Oracle to SQL Server. Working SQL and holistically offers you a great advantage.

However, the most important reason for a holistic approach is the performance, all the rest are secondary. Performance, portability, simplicity of the code, and the fact that the data set is the keyword that defines a relational database, all these are enough reasons for an application developer to start rethink his code when being in a database and for a student to think and not work identically in both user interface and database.

WRITING SQL VERSUS WRITING PROCEDURALLY: THE POWER OF UNION IN THE HOLISTIC APPROACH!

The battle between the atomic procedural facilities and the holistic SQL methods of work may continue! The most important thing is to be aware of this battle and to try to utilize one kind of facility or another depending on the situation. Most of all, we should not forget that we are within a relational database.

Let's imagine a situation that might occur in various systems, especially in specific systems where the goal is to move data between classic systems. We have a target that should be updated from various sources of data. By updated we understand that new data is added to the target, from various sources of data, according to whatever business conditions. These various conditions can be specified by using the procedural means like if, else – if, else statements. Very often, the same logic can be done by using a union.

Instead of using a variety of if-else and insert statements, we can have only one insert statement based on a "union". This is not a surprise considering that a "union" is a combination of data sets. The meaning of an if-else in this context is the one of adding data in one target from various sources based on a certain condition or another. This can always be equivalent to a union composed of many data sets, and in every data set, we can specify the condition in the if-else. One should be aware that the topic of discussion is related to the duality SQL versus procedural, is not necessarily referring to the opposition atomic holistic. **The approach can be holistic but even the holistic approach can be implemented in a SQL style or in a procedural style**. Moreover, the use of union instead of if-else is a solution to replace a procedural code with SQL code. I don't say it's always better and I don't say it is always recommended, but I just recommend to be kept in mind. Sometimes it might be better, especially in specific systems.

In a **specific** system I consider that, due to the nature of this system of data transfer between various classic systems, it is always better to handle things both holistically and SQL, not just holistically.

Let's imagine that we have geographical criteria for some data and let's imagine we want to populate a target from various sources. The sources are from different countries and we need to insert various pieces of information based on a condition like country. Instead of using an "if else", we can use a "union", maybe even "union all" if we are certain the data sets are distinct. In each query block of the union, we have the respective data set that we would have been written in the if-else. Things are similar and, in this context of an insert into a target from various sources based on various conditions, the use of if-else and union are very often alternatives. If this is the case, sometimes is better to use union, and even better, to use union all if possible, obviously.

The use of union with the use of variables that specify different sets of data can be easily integrated and, very often, large pieces of procedural code (not necessarily atomically, even in the holistic manner) can be replaced with a highly simplified piece of SQL code. For example, any query block of a union can be defined according to a value of a certain application setting. In this way, the query block may be identified either by certain columns that need to satisfy some criteria (like country) but they can also be defined by some variables too.

This sub chapter opens a new topic and describes another opposition inside a relational database. The main topic is focused on the opposition between the holistic and atomic style of development inside a database. Now, apart from that opposition, we have another opposition between procedural and SQL code. As we realize now, it is possible to write holistic and either procedural and SQL, although in most of the cases the procedural style is associated with the atomic style and the SQL style is associated to the holistic style. Still, we can write procedurally and holistically. The example with the opposition between the two types of statements, the "if-else "and the "union", confirms the above considerations. Even if we write holistically and we affect data sets, we can use any of "if-else" or "union" statements. The first approach is procedural and the second one is SQL. I promote the "union". Still, I do not say that we always have justified reasons to choose this solution. Sometimes it may be better to use if-else. Still, the alternative is to be kept in mind.

Let's imagine you have a warehouse of SQL templates that is called multiple times in your software application, at the database level of course. The temptation is to avoid the use of procedural code in this warehouse, as being a warehouse of SQL statements. In this case, you can use the union instead of if-else and your warehouse will really be a warehouse of SQL statements. I have built that kind of warehouse and it has almost no procedural code inside it, only pure SQL and almost everywhere, I found solutions to replace the procedural code.

Let's see one example with the opposition between the "if-else" and "union". Let's assume the same tables with the de-normalized English and French countries and the normalized table. Now we need to assume the other direction. We will assume we have data in English and French countries and we need to populate the set of normalized tables like Countries_Languages. We assume we have the countries and the languages and we just need to populate the association tables. Based on a variable p_Language_Name we can add either French languages or English languages or both. Of course, the atomic solution can be used and I will not insist on that too much, simply ignoring it. The same holistic solution can be made in a more specific procedural way or in a more SQL oriented manner.

Let's see the procedural solution.

```
-- Code example 29 Holistic and Procedural Oracle
CREATE PROCEDURE Holistic_Full_Tr_Country_Proc
(
      p_Language_Name VARCHAR
)
AS
      v_English_Language_Id INT;
      v_French_Language_Id INT;
      v_Max_CL_Id INT;
BEGIN
      SELECT Language_Id INTO v_English_Language_Id
      FROM Languages WHERE Language_Name = 'English';
      SELECT Language_Id INTO v_French_Language_Id
      FROM Languages WHERE Language_Name = 'French';
      IF p_Language_Name = 'English' THEN
```

```
            DELETE Countries_Languages
            WHERE Language_Id = v_English_Language_Id;
        ELSIF p_Language_Name = ‘French’ THEN
            DELETE Countries_Languages
            WHERE Language_Id = v_French_Language_Id;
        ELSIF p_Language_Name = ‘Both’ THEN
            DELETE Countries_Languages
            WHERE Language_Id = v_English_Language_Id;
            DELETE Countries_Languages
            WHERE Language_Id = v_French_Language_Id;
        END IF;
        SELECT MAX(CL_Id) INTO v_Max_CL_Id
        FROM Countries_Languages;
        IF v_Max_CL_Id IS NULL THEN
            v_Max_CL_Id := 0;
        END IF;
        IF p_Language_Name = ‘English’ THEN
                INSERT INTO Countries_Languages (CL_Id, Country_Id, Language_Id, Language_Category)
                SELECT  v_Max_CL_Id  +  CL_Id_Seq.NextVal,  c.Country_Id,  v_English_Language_Id,
eec.Language_Category
                FROM English_European_Countries eec INNER JOIN Countries c ON (c.Country_Code =
eec.Country_Code);
        ELSIF p_Language_Name = ‘French’ THEN
                INSERT INTO Countries_Languages (CL_Id, Country_Id, Language_Id, Language_Category)
                SELECT  v_Max_CL_Id  +  CL_ID_Seq.NextVal,  c.Country_Id,  v_French_Language_Id,
eec.Language_Category
                FROM French_European_Countries eec INNER JOIN Countries c ON (c.Country_Code =
eec.Country_Code);
        ELSIF p_Language_Name = ‘Both’ THEN
                    INSERT   INTO   Countries_Languages   (CL_Id,   Country_Id,   Language_Id,
                    Language_Category)
                SELECT  v_Max_CL_Id  +  CL_ID_Seq.NextVal,  c.Country_Id,  v_English_Language_Id,
eec.Language_Category
                FROM English_European_Countries eec INNER JOIN Countries c ON (c.Country_Code =
eec.Country_Code);
                INSERT INTO Countries_Languages (CL_Id, Country_Id, Language_Id, Language_Category)
                SELECT  v_Max_CL_Id  +  CL_ID_Seq.NextVal,  c.Country_Id,  v_French_Language_Id,
eec.Language_Category
                FROM French_European_Countries eec INNER JOIN Countries c ON (c.Country_Code =
eec.Country_Code);
        END IF;
        COMMIT;
END Holistic_Full_Tr_Country_Proc;
/
```

The logic is very clear. Based on the language, either that can be of type English, French or both, the data for that language in the normalized table like Countries_Languages is deleted and replaced with the data from the one reporting table or the other or both. It is a classic “if-else” statement, nothing else. A lot of programmers will do it like this, or in a similar fashion, especially the application developers. This is a pure procedural solution, and fortunately is a holistic solution. Unfortunately, this exercise could have been made atomically using a cursor and that would have been the worst scenario. Being a holistic solution is good, the application developer already learned to think holistically. That is great! Now I want to show him more and I want to show him that sometimes simple SQL statements are enough and the quantity of procedural code can be completely minimized. There is a lot of code in this procedure. One should imagine hundreds of pieces like this everywhere in the software application within the database. Even more, let's imagine that these can be avoided and replaced with pure SQL code; instead of 10,000 lines of code, you will have 3,000, for example. In most of the cases, the performance will be better. Let's see the holistic and SQL oriented solution for the practice.

```
--Code example 30 Holistic and Procedural Oracle
CREATE PROCEDURE Holistic_Full_Tr_Country_SQL
(
      p_Language_Name VARCHAR
)
AS
BEGIN
      DELETE Countries_Languages dest
      WHERE EXISTS
      (
          SELECT 1 FROM Languages lang
          WHERE lang.Language_Name = p_Language_Name
AND p_Language_Name IN (‘English’, ‘French’)
          AND lang.Language_Id = dest.Language_Id
      )
      OR EXISTS
      (
          SELECT 1 FROM Languages lang
          WHERE lang.Language_Name IN (‘English’, ‘French’)
AND lang.Language_Name = ‘Both’
          AND lang.Language_Id = dest.Language_Id
      );
          INSERT INTO Countries_Languages (CL_Id, Country_Id, Language_Id, Language_Category)
          SELECT v_Max_CL_Id + RowNum AS CL_Id, Country_Id, Language_ Id, Language_Category
      FROM
      (
          SELECT c.Country_Id, eec.Language_Category,
          (SELECT COALESCE(Max(CL_Id), 0)
      FROM Countries_Languages) AS v_Max_CL_Id,
          (SELECT Language_Id FROM Languages
WHERE Language_Name = ‘English’) AS Language_Id
          FROM  English_European_Countries  eec  INNER  JOIN  Countries_1  c  ON  (c.Country_Code  =
eec.Country_Code)
          WHERE p_Language_Name IN (‘English’, ‘Both’)
          UNION
          SELECT c.Country_Id, eec.Language_Category,
          (SELECT COALESCE(Max(CL_Id), 0)
      FROM Countries_Languages) AS v_Max_CL_Id,
          (SELECT Language_Id FROM Languages WHERE Language_Name = ‘French’) AS Language_Id
      FROM  French_European_Countries  eec  INNER  JOIN  Countries_1  c  ON  (c.Country_Code  =
eec.Country_Code)
```

```
        WHERE p_Language_Name IN (‘French’, ‘Both’)
        ) lf;
        COMMIT;
  END Holistic_Full_Tr_Country_SQL;
  /
```

As you can see, this logic contains one delete statement and one insert statement. The reason is simple: SQL contains by its own a lot of “procedural” facilities, almost all of them set-oriented. If you understand the power of SQL language and if you understand the concept of data set, you will be able to see its strength and use it properly. The code is much simpler and readable; at least for the persons that finally are able to think SQL, the performance is much better because SQL is set-based and is very fast and native for working with data sets, and this is what we are doing when we are inside a database.

Speaking about the power of SQL and about the possibilities for a better performance for SQL itself, this is a completely another topic, very challenging and interesting. However, before thinking to improve our SQL code we should write it properly.

An artist, a writer or a poet works continuously to improve his work. For example, a poet wants to use the metaphors better and he works intensively to improve. However, before doing this, he makes sure his writing is accurate from the grammar point of view and all other aspects.

First of all the programmer needs to write correctly, he should write holistically and SQL if he is inside the database. Secondly, when he is able to write set oriented and SQL, he can work to improve his SQL. For that, there are so many techniques and methods, starting from a continuous process of rewriting for the SQL itself and continuing with a lot of features and techniques available for the database engine. Here one can work with the DBA of course. At this level, the developer needs to know how to read an execution plan, he needs to be able to see the statistics and eventually gather new ones at various levels etc.

If you compare the two approaches above, you will see the difference between the classic procedural style, with a lot of if-else statements, variables and all the staff and the set of simple SQL statements! You will see the mixture between the two types of statements, SQL and procedural. If you look at the last example, you will see just some pure SQL statements, all the logic is included in the SQL itself. In a data-oriented specific software **application** where the goal is to simply move data between two classic systems, this SQL style, along with holistic and set-oriented, is the one that is highly recommended, at least according to my opinion and experience. I will not show the SQL Server version of the logic, because the things are pretty much the same and the differences are minimal. This is especially true because of the style and the holistic approach common to both examples. Of course, in the procedural version, there are more differences specific to the language.

EMBEDDED SQL VERSUS DYNAMIC SQL – ANOTHER DILEMMA!

I remember even now, after so many years, when I first saw the use of dynamic SQL! If was very fascinating for me, at that time, to see how you can concatenate and how you can add more and more filter conditions to a string. The filter conditions were dynamic generated by the user, so there were good reasons for the use of dynamic SQL. The context was one of a report, the conditions in the report contains many parameters, the filter conditions. The string continues to grow and grow, becoming a nice and relevant SQL statement. Finally, in the end, when the concatenation ends, the string was executed and the results set was displayed. This was my first scenario of the use of dynamic SQL. There are many good reasons for the use of dynamic but I would say that the most common one, and the most rational, is the one when there are uncertainties at run time.

For example, sometimes you want to query a table. You don't know exactly which table: it can be one or the other depending on the execution context. You need to select from the unknown and that is not quite normal and common, is it? In these cases, when uncertain things are waiting at the door of execution, you may consider the use of dynamic SQL.

As you know, for any possible reader not exactly familiarized with the distinction, there are two types of SQL. The SQL can be embedded or dynamic. An embedded SQL is the normal SQL, a clear and fair statement or set of statements, written as such and interpreted as such, clearly compiled and determined before the execution. If everything goes fine with the compilation step and the syntax of the statement, the semantic of the statement are checked and the primary and basic elements of the statement are consistently determined. Afterwards, at execution, other surprises or reasons for error might occur but all these are part of the execution context. This is the default type of SQL, the embedded SQL. Normally, a good SQL developer will try to use embedded SQL as much as he can and he should be confident in the separation between compilation phase and execution phase, as two distinct phases that should be analyzed separately and chronologically. If nothing is certain at run time, embedded SQL is an absolute favorite. Very rarely, you can imagine reasons for something else if everything is clear and known at run time.

Still, sometimes the execution may contain questions. Let's see the example when we need to select in a table, or update the table in a certain way. We don't know the name of the table when we build the logic: that one might change at run time. In these cases, we build a string containing all the known elements of the statement and the unknown elements of the statements specified as parameter for the string. After that, the string is generated it is executed. The string will become known at execution time so the compilation and execution will be made at the execution time. This is a reason for concern because the execution will take the burden of compilation and its impact may be infinitely more severe. This is the main reason for which the use of dynamic SQL is to be avoided. It is always better to solve the basic problems of parsing before execution. No one wants to check the syntax and no one wants to check the semantics of the statements at execution time.

I remembered some of my vacations with my family. Most of the preparations for the trip should be made at least in the previous day of our departure. I don't want to imagine what would happen if we would have to prepare everything in the morning of our departure. I would need to be twice as concentrated to be sure I do not forget anything! Things are Similar with the use of dynamic SQL, to a certain degree!

In one of the data migration interfaces I used to work on it happened to have large lists of attributes that should have been updated quite often. The statement was pretty much the same but it contained, of course, different variables, like the attributes to be updated. Therefore, it was a repetitive code and this could have been transformed in dynamic SQL. Still I would rather prefer to avoid this. I continued to use dynamic SQL but I kept that separately in what I named a list of SQL generators. That list contained many strings that concatenate constants like keywords and variables like columns from tables. I had tens of lists and I kept them safely but I did not add them in the software application. These lists were executed and the content of the lists were sets of embedded SQL statements. These statements were copied into the software applications. In this way, the logic was consistent and classic containing only embedded SQL. If I was to have an error, I knew exactly where it was and this was another reason for embedded SQL.

Finding and locating the errors when using dynamic is an extremely difficult task and you need to dig like a marathon runner to get the exact place of the error and start the debug activity. If anything changes, I can use the SQL generators and generate everything again.

So this is a good example where dynamic SQL is a good choice, when is acting like a shadow or like a ninja worrier, in the back of the actual code. No one knows it exists! I believe it is better this way!

The topic of what is more important: the clarity of the code or the length? For example, you have 50 attributes of a certain type in a migration system that should be updated, some of them depending on the changes. You can write the 50 similar statements in one string and execute that string in a cursor. The length of the code will be very small, the clarity and readability will be at the lowest level. The second solution is to continue to keep that string but separately in a file with all sort of strings like this, I called them SQL generators. You will execute the string and you will have 50 statements that you will copy in the logic. The length of the procedure will be large and the clarity and readability at the maximum level.

The difference between having dynamic SQL versus embedded SQL.

If there are no questions at execution time these are alternatives solutions. In my opinion, the increased length of a stored procedure is not an important matter. I find this even completely irrelevant if I am thinking at the clarity we have with everything clearly written. Apart from that, if you have an error mechanism you can follow everything and catch everything without any doubts.

Let's see one example. Let's imagine we have more reporting tables not just English and French but 50 reporting tables. You want to generate periodically the data in the 50 tables from the normalized table. You can think at the solution of writing a stored procedure where to update the 50 tables and use embedded SQL or you can think at a solution to write a string and execute it in a cursor changing the name of the country and populate every country from the 50 tables in the cursor. In the first case, you need to maintain the file with the set of 50 embedded SQL statements, new countries are added and the file should be maintained. It is some work to be done. Using dynamic SQL, you don't need to do anything, but you lose the clarity and the significance of the code. However, you have a third option: to use these strings as part of the backup logic or you can name it metadata logic, to use these strings as SQL generators and execute them to generate the real files of the software.

Before that, let's add one column to the table Languages. Let's add some data too. Let's see the changes below:

```
Code example 31: Add metadata information.
-- Execute Code example 05 before apply these changes.
ALTER TABLE Languages
ADD Language_Table_Name VARCHAR(30);
UPDATE Languages
SET Language_Table_Name = 'English_European_Countries'
WHERE Language_Id = 2;
UPDATE Languages
SET Language_Table_Name = 'French_European_Countries'
WHERE Language_Id = 3;
```

As you know, we have two reporting tables: one for the English language and one for the French language. These tables are linked to the language so I will add this information to the languages tables as you can see above. More reporting tables will follow: this information will be updated in the Languages table. If tomorrow the reporting table for Spanish will be made, the name of the table will populate the row with the Spanish language. In this way, the table languages will contain both data and metadata information and will double its utility. Try to rebuild the data in Countries_Languages and add the initial script and the changes reflected in the initial script (Code example 05).

The update above allows us to see data and metadata information in one place and allows us to use dynamic SQL. The table Languages contains the rows 2 and 3 with the values English_European_Countries and French_ European_Countries. These are values in the "business" table so these are data. On the other hand, these values correspond to some objects in the database, respectively to the tables with the same names. Therefore, these are metadata. This is a common scenario for dynamic SQL, trying to generate your own set of metadata and use that one for various purposes. The set of custom metadata offers the advantage that allows the developer to generate SQL logic, especially if things are repetitive. This logic can be hidden in dynamic SQL or can be revealed in embedded SQL and the mechanism for embedded SQL can be separated in a parallel logic. I am the promoter of this second approach, use metadata and dynamic SQL in a parallel layer, generate, and use embedded SQL. At execution time, the software application will see exclusively embedded SQL.

Let's move back to the example. After recreating the three base table again using the script in Code example 05 let's see one classic example of dynamic SQL in action.

```
Code example 32 Holistic with dynamic SQL for SQL Server
CREATE PROCEDURE Holistic_Full_Country_Dynamic
AS
```

```
        DECLARE @v_Language_Id INT;
        DECLARE @v_Language_Name VARCHAR(50);
        DECLARE @v_Language_Table_Name VARCHAR(30);
        DECLARE @v_SQL_Statement NVARCHAR(1000);
        DECLARE c_Get_Languages CURSOR FOR
        SELECT Language_Id, Language_Name, Language_Table_Name
        FROM Languages
        WHERE Language_Table_Name IS NOT NULL;
BEGIN
        OPEN c_Get_Languages
        FETCH NEXT FROM c_Get_Languages INTO @v_Language_Id, @v_
Language_Name, @v_Language_Table_Name
        WHILE @@FETCH_STATUS = 0
        BEGIN
SET @v_SQL_Statement = ‘DELETE ‘ + @v_Language_Table_Name;
EXECUTE sp_executesql @v_SQL_Statement
        SET @v_SQL_Statement = ‘INSERT INTO ‘ + @v_Language_Table_ Name + ‘ (‘ + @v_Language_Name +
‘_CL_Id, Country_Code, Country_Name, Language_Category)’ + ‘ SELECT ROW_NUMBER() OVER (ORDER BY
c.Country_Code, cl.Language_Category) AS CL_Id, c.Country_Code,
        c.Country_Name, cl.Language_Category
        FROM Countries_Languages cl INNER JOIN Languages l ON (l.Language_Id = cl.Language_Id)
        INNER JOIN Countries c ON (c.Country_Id = cl.Country_Id)
        WHERE l.Language_Name = ‘’’ + @v_Language_Name + ‘’’’;
        EXECUTE sp_executesql @v_SQL_Statement
        FETCH NEXT FROM c_Get_Languages INTO @v_Language_Id, @v_ Language_Name,
@v_Language_Table_Name
        END
        CLOSE c_Get_Languages
        DEALLOCATE c_Get_Languages
END
GO
```

This is dynamic SQL! It is a special kind of programming and it has a certain degree of popularity. I was somehow a fan of dynamic SQL in my beginnings but not anymore. Anyway, dynamic SQL is very useful and of great help in certain situations. Let’s look at the code: it seems a bit like a Morse code, isn’t it? You should be able to guess and know what is behind to be able to understand, it has a lack of meaning. Of course, you can always print the strings before executing anything and you can try to understand the meanings, and this is what we all do when we have issues.

In this example, we have just two reporting tables. We generated and executed one delete and one insert statement for two tables. This is not a serious economy in code. However, you should imagine we have 20 reporting tables and you generate one insert statement and one delete statement instead of twenty. The amount of code will be reduced to a maximum level. The procedure will have 20 lines instead of 200 lines. This is one consequence of dynamic SQL. The length of the code is multiple times smaller but also the meaning. Actually, the use of dynamic SQL minimizes the length of the code and the length of the meaning. Not too many people will understand what is there, many people will wonder and try to guess, start printing various strings, the final string or intermediates: it will be a battle for understanding that will take some time for new programmers working in dynamic SQL. What is more important in a procedure, what is more important when you are looking at the code? Is it the length of the code? Are we happy to see one procedure with 10 lines but no meaning? Alternatively, are we happy to see one procedure with 100 lines but with a clear meaning? Of course, those things are also relative to each ones experience, there are no general answers to these questions.

Now if we will execute the procedure above we will populate the French and English reporting tables. You will see the same data as in Chapter 05 after executing the procedure Atomic_Full_Transfer_Country for both parameters English and French (see the values). There are five English countries and two French ones.

The programmer can choose to use embedded SQL and to write in white and black everything, to explicitly specify all the tables and objects. The logic will contain everything. With embedded SQL you see what you execute, you understand everything and you can catch the errors, if any. The handle errors procedure generator will take you to the error exactly and you can safely debug everything. With dynamic SQL, you need to guess and it is extremely difficult sometimes to identify the place with the error, this is another disadvantage of dynamic SQL.

However, the use of dynamic SQL can be extremely useful sometimes even apart from the classic situation with uncertain things at run time. That case is classic and, if it cannot be avoided, we are somehow forced to use dynamic SQL. Apart from that, the dynamic SQL can be used in a parallel layer, let's call it generator.

Let's take the example above. Let's change that code and let's remove the essential call to the string, the sp_executesql and replace it with a simple print. For simplicity, I will just print the string instead of saving it directly for example in a custom metadata table with SQL generators.

Now the new procedure will be similar with the previous one but with one major difference. Let's see it first.

Code example 33 SQL generator for SQL Server

```
CREATE PROCEDURE Holistic_Full_C_Generator
AS
      DECLARE @v_Language_Id INT;
      DECLARE @v_Language_Name VARCHAR(50);
      DECLARE @v_Language_Table_Name VARCHAR(30);
      DECLARE @v_SQL_Statement NVARCHAR(1000);
      DECLARE c_Get_Languages CURSOR FOR
      SELECT Language_Id, Language_Name, Language_Table_Name
      FROM Languages
      WHERE Language_Table_Name IS NOT NULL;
BEGIN
      OPEN c_Get_Languages
      FETCH NEXT FROM c_Get_Languages INTO @v_Language_Id, @v_ Language_Name,
@v_Language_Table_Name
      WHILE @@FETCH_STATUS = 0
      BEGIN
SET @v_SQL_Statement = 'DELETE ' + @v_Language_Table_Name;
PRINT @v_SQL_Statement
SET @v_SQL_Statement = 'INSERT INTO ' + @v_Language_Table_Name + ' (' + @v_Language_Name +
'_CL_Id, Country_Code, Country_ Name, Language_Category)' + ' SELECT ROW_NUMBER() OVER (ORDER BY
c.Country_Code,    cl.Language_Category)    AS    CL_Id,    c.Country_Code,    c.Country_Name,
cl.Language_Category

     FROM Countries_Languages cl INNER JOIN Languages l ON (l.Language_Id = cl.Language_Id)

      INNER JOIN Countries c ON (c.Country_Id = cl.Country_Id)

      WHERE l.Language_Name = ''' + @v_Language_Name + '''';

      PRINT @v_SQL_Statement

FETCH NEXT FROM c_Get_Languages INTO @v_Language_Id, @v_Language_Name, @v_Language_Table_Name

      END

      CLOSE c_Get_Languages

      DEALLOCATE c_Get_Languages

END

GO
```

The difference is the difference between a print instruction and an execute instruction. The same string is printed instead of being executed. Consequently, the difference is actually huge. The previous procedure was named Holistic_Full_Country_Dynamic and this one is called Holistic_ Full_C_Generator. Now we will execute the last procedure and we will display the data in a new code example. This new code example will be actually a new logic, the logic itself, the logic I consider the one that should be used everywhere, the embedded SQL. The printed string is the text that should be executed. Instead of executing the string in dynamic SQL, I will generate the text with embedded SQL. That text will actually be the effective logic. So let's see the execution results, after calling the last procedure.

Code example 34 The execution results for Holistic_Full_C_ Generator

```
      DELETE English_European_Countries

      INSERT   INTO   English_European_Countries   (English_CL_Id,   Country_Code,   Country_Name,
Language_Category)

      SELECT  ROW_NUMBER()  OVER  (ORDER  BY  c.Country_Code,  cl.Language_Category)  AS  CL_Id,
c.Country_Code,

          c.Country_Name, cl.Language_Category

          FROM Countries_Languages cl INNER JOIN Languages l ON (l.Language_Id = cl.Language_Id)

          INNER JOIN Countries c ON (c.Country_Id = cl.Country_Id)

          WHERE l.Language_Name = 'English'

      DELETE French_European_Countries

      INSERT   INTO   French_European_Countries   (French_CL_Id,   Country_Code,   Country_Name,
Language_Category)

      SELECT  ROW_NUMBER()  OVER  (ORDER  BY  c.Country_Code,  cl.Language_Category)  AS  CL_Id,
c.Country_Code, c.Country_Name, cl.Language_Category

          FROM Countries_Languages cl INNER JOIN Languages l

      ON (l.Language_Id = cl.Language_Id)

          INNER JOIN Countries c ON (c.Country_Id = cl.Country_Id)

          WHERE l.Language_Name = 'French'
```

Now this is amazing. What we have here is so clear and readable! Everyone can see and understand. If I have an error that one will be identified correctly very fast, after the line id and I will know exactly where to go and debug. The code is pure SQL, native database code, the same code as before but the difference is that now the code is visible and not hidden. What do I need to do? Take this execution result and add it, create a new procedure and store that in your database. That will be the real procedure, containing embedded SQL. Let's display it!

Code example 35 Holistic with embedded SQL for SQL Server

```
        CREATE PROCEDURE Holistic_Full_Country_Embedded
        AS
        DELETE English_European_Countries;
        INSERT  INTO  English_European_Countries  (English_CL_Id,  Country_Code,  Country_Name,
  Language_Category)
        SELECT  ROW_NUMBER()  OVER  (ORDER  BY  c.Country_Code,  cl.Language_Category)  AS  CL_Id,
  c.Country_Code,
            c.Country_Name, cl.Language_Category
            FROM Countries_Languages cl INNER JOIN Languages l
        ON (l.Language_Id = cl.Language_Id)
            INNER JOIN Countries c ON (c.Country_Id = cl.Country_Id)
            WHERE l.Language_Name = 'English';
        DELETE French_European_Countries;
        INSERT  INTO  French_European_Countries  (French_CL_Id,  Country_Code,  Country_Name,
  Language_Category)
        SELECT  ROW_NUMBER()  OVER  (ORDER  BY  c.Country_Code,  cl.Language_Category)  AS  CL_Id,
  c.Country_Code,
            c.Country_Name, cl.Language_Category
            FROM Countries_Languages cl INNER JOIN Languages l
        ON (l.Language_Id = cl.Language_Id)
            INNER JOIN Countries c ON (c.Country_Id = cl.Country_Id)
            WHERE l.Language_Name = 'French';
        GO
```

This is the real procedure and the real logic that should be displayed and executed. The approach is purely holistic and SQL, the reporting tables are updated from the normalized system: everything is clear and straightforward. If tomorrow we add two more reporting tables, we have two options: I can add the new two insert and delete statements directly, or I can simply update the custom metadata information and execute the generator again. I can copy the text into the procedure and recreate the logic. This is my recommended way of work for these kinds of situations.

The use of dynamic SQL is great and it helps us solve some delicate situations like the one when you want to select from a table but the name of the table is unknown. Now I want to generate the English countries and tomorrow I want the French countries. The decision, the English or French language or anything else, is taken somehow at run time by the execution user. In these scenarios, dynamic SQL is a good decision. However, in a scenario where we simply have repetitive code and when we have a list of actions that should be generated, like a list of insert statements, we can use dynamic SQL in a parallel layer called generator. That generator will be executed periodically and will be the base for our effective code. The effective code will be pure embedded SQL with all the advantages of embedded SQL. This is the scenario that I recommend and consider the best in this context.

OTHERS HOLISTIC SOLUTIONS: THE TEMPORARY TABLE, EXPLICIT OR IMPLICIT LIKE WITH A CLAUSE

We will move back to the opposition between the atomic approach and the holistic one. The atomic style is defined in most of the cases by the use of one or more cursors, by the use of variables or records or anything similar, where the data in the cursor is stored atomically, by the loop through the cursor and whatever manipulation is required. This strategy and style is adopted by many application developers that are not fully aware of the fact that inside a relational database the manipulation should be done in most of the cases per data sets, in a holistic manner. My suggestion in most of the examples is that the holistic approach means solving the problem by using a simple SQL statement. By simple, I do not mean that any solution should be very simple and actually sometimes, a SQL statement, like a select statement, can be very complicated and can contain even tens or hundreds of lines. The code length, the simplicity or complexity of the statement is not an argument in itself and, very often, the holistic solution is better than the atomic approach. Still, sometimes, a simple SQL statement is not enough to solve the problem. Sometimes the data need to be manipulated and intermediary results should be stored and set before getting the result. Sometimes, to be able to reach the result, you need to intermediary update the data, to generate all kinds of identifiers, to do whatever concatenations, to do whatever calculations. There are so many possible reasons for this data manipulation. In these situations, a simple SQL statement will not be enough to solve the problem in a holistic manner. It is even possible to be forced to use an atomic solution and to go to the cursor and to the loop within the cursor. Nevertheless, before following that approach, we can try something else. We can try to use various holistic techniques for data manipulation before choosing the cursor. Some examples are the temporary tables, either explicit or implicit.

The temporary table is a kind of table very dear to SQL Server developers and not so common for Oracle developers. There are some differences between the statuses of the temporary table in the two systems. This is not relevant in our context. The temporary table is a kind of table that exists in any database system, like Oracle, SQL Server, DB2, PostgreSQL and any other. This is one holistic method for storing intermediates in a data manipulation and is often an alternative to the cursor facility and to the atomic approach. The temporary table can be explicit or implicit, if we consider the “with clause”, more and more used in the database software development. The temporary table generated by the “with clause” is very common nowadays and offers great advantages, one of them being exactly this one, avoiding the atomic approach for a better performance and a holistic data manipulation.

During one of my projects, I was in the position to improve performance. I already mentioned that, and I need to repeat because is related to my actual topic. There was a nice and big cursor, like Big Brother. This Big Brother drives the entire process of a report generation. To be able to generate that report, a lot of intermediates results were generated before the result. Of course, the Big Brother was extremely slow and the report was terrible. After some deliberation, I decided, as almost always when try to improve performance, that a holistic solution was required. A simple SQL was not possible, due to the fact that the data to be manipulated required updates. Therefore, I needed to search for something else.

The “with clause” is a great feature. I tried to use that first, but it was not enough. So finally, I used a classic temporary table instead. The atomic approach was used but not with a direct SQL but with a classic temporary table. The table was populated first and updated afterwards based on certain conditions. The difference in performance against the cursor was huge. Even if you use a normal table and not a temporary table, the difference is impressive comparing with the cursors and the atomic approach. I know that temporary tables are not loved by everyone! There are voices against them! I do not intend to consider them more than they should be. However, comparing with the atomic approach, the temporary table is a better solution because it works holistically. Not all the time and not in any situation but in most of the cases I strongly believe is a better solution. In addition, by the way, the solution that I used and where the temporary table worked much better than the atomic approach it was, surprisingly, in an Oracle environment and not in the classic SQL Server environment.

We all know that cursors in Oracle have a good performance. Despite all these considerations, the atomic approach remains an abnormal style that should not be used unless a holistic solution is available. At least this is my opinion and this is what I try to prove everywhere in this book.

Another holistic feature is the table function, or a type of function that returns a data set or a similar holistic object. These functions are available in many database systems and these are a better alternative than scalar functions executed in cursors. We already discussed about that, I want to introduce another example of performance improvement that I did, now in a SQL Server environment.

There was a complex logic of data transfer between two systems. It was that kind of specific application very suitable for the holistic approach. In this logic, one portion was very expensive with a very poor performance. Analyzing the logic I noticed the procedural and atomic style, and how everything was written in cursors, a variety of scalar functions were defined and called everywhere in those cursors. The performance was a disaster and, of course, any improvement of any type was not possible, except for the rewrite mechanism, of course! The rewrite procedure is very expensive, whenever you rewrite the logic you need to continuously check the results, not just the final but also the intermediates. It is a difficult work, but the satisfaction when you are done is enormous! So what have I done? I replaced almost all the scalar functions and the cursors with a table function. The table function returns a data set, like a temporary table. Instead of updating all the items in the cursor several times using various scalar functions, I updated the table to be returned in the table function. Instead of acting atomically, I acted holistically. That was the big difference. Once more, the difference in performance was impressive.

These are just two examples of situations where the holistic approach was implemented using other types of objects like temporary tables, explicit or implicit, or table functions instead of cursors with scalar functions. The first type of solution whenever you try to replace an atomic solution is to try to use a simple SQL statement. It will work more often than you expect. If one simple SQL statement cannot solve the problem, because the data set should be updated somehow and the simple SQL select statement is not sufficient, there are others possible solutions like the ones specified above. The atomic approach will still be used if the row-by-row functionality is really requested by the logic. This will happen but very rarely, more rarely than you can imagine.

Let's see one more example.

We will go back to the set of product tables, the products and their descriptions (see Code example 21). Let's see the business description.

We want to calculate the quantities per type of the product according to an algorithm. For each product, if the type is C, I am reading the flag. If the flag is set to one (1), I will take the quantity squared and if the flag is zero (0) I will take the double of the quantity. If the type is D, I will look again at the flag and I will take the quantity squared minus simple quantity if the flag is positive and I will take the triple quantity if the flag is zero.

The same question again, do we see this exercise atomically? Are we already riding on the row like John Wayne in the old times? Let's see the pure atomic solution, although it is quite clear that this can be easily avoided.

Before executing the procedure, reinitialize the list of products and types. This means run the statements in Code example 22 again not before deleting the products and the associated types.

Let's see the atomic solution in PL SQL.

Code example 36: Atomic get the default quantity per type Oracle

```
CREATE PROCEDURE Atomic_Get_Qtty_Per_Type
AS
      v_Product_Type_Code VARCHAR(5);
      v_Make_Flag INT;
      v_Default_Quantity INT;
      v_First_Letter_Type CHAR(1);
      v_Current_Qtty INT;
      v_Current_Qtty_C INT;
      v_Current_Qtty_D INT;
      CURSOR c_Get_Products_Qtty IS
      SELECT pt.Product_Type_Code, p.Make_Flag, p.Default_Quantity
      FROM Product_Types pt INNER JOIN Products p
          ON (p.Product_Type_Id = pt.Product_Type_Id);
BEGIN
          v_Current_Qtty_C := 0;
          v_Current_Qtty_D := 0;
          v_Current_Qtty := 0;
          OPEN c_Get_Products_Qtty;
          LOOP
              FETCH c_Get_Products_Qtty
              INTO v_Product_Type_Code, v_Make_Flag, v_Default_Quantity;
              EXIT WHEN c_Get_Products_Qtty%NOTFOUND;
          v_First_Letter_Type := SUBSTR(v_Product_Type_Code, 1, 1);
              IF v_First_Letter_Type = 'C' THEN
                    IF v_Make_Flag = 1 THEN
          v_Current_Qtty := v_Default_Quantity * v_Default_Quantity;
```

```
                    ELSIF v_Make_Flag = 0 THEN
                    v_Current_Qtty := 2 * v_Default_Quantity;
                    END IF;
              ELSE
              IF v_Make_Flag = 1 THEN
              v_Current_Qtty := v_Default_Quantity * v_Default_
              Quantity - v_Default_Quantity;
              ELSIF v_Make_Flag = 0 THEN
                    v_Current_Qtty := 3 * v_Default_Quantity;
              END IF;
          END IF;
          IF v_First_Letter_Type = 'C' THEN
          v_Current_Qtty_C := v_Current_Qtty_C + v_Current_Qtty;
          ELSE
          v_Current_Qtty_D := v_Current_Qtty_D + v_Current_Qtty;
          END IF;
          END LOOP;
          CLOSE c_Get_Products_Qtty;
          DBMS_OUTPUT.PUT_LINE ('The total default quantity for the products with C type is ' ||
TO_CHAR(v_Current_Qtty_C));
          DBMS_OUTPUT.PUT_LINE ('The total default quantity for the
          products with D type is ' || TO_CHAR(v_Current_Qtty_D));
END;
/
```

This solution is very traditional and I believe there is no need for any explanation. The same procedural logic applies, for each row in the data set with the list of products, we check the type if is C or D, then we look at the flag and, depending on the value, we calculate one quantity or another. Using a kind of global variable to the cursor, actually two, one for C and one for D, we add the current quantities to either C or D. Finally, the two global variables will store the quantities.

After the execution of the procedure, you will obtain the following results:

```
anonymous block completed
The total default quantity for the products with C type is 646 The total default quantity for the
products with D type is 30
```

It is clear that this solution will vary based on the database system. Even if it would be similar, there would be serious differences between Oracle, SQL Server DB2 and PostgreSQL for example. The procedural language is different and you need to familiarize with one or another. I continue to say that this matter is not the most important one because the procedural languages, even if different, are still similar. I don't find it very difficult to accommodate one knowing another. As I said, a cursor is a cursor, a loop is a loop, the syntax is different but the meaning is the same. The atomic style is the same.

Now let's see the holistic approach for Oracle. Seeing this complicated logic, you might believe is going to be difficult. Actually it is not, and the famous "with clause" that we discussed earlier will transform the previous exercise in a simple SQL statement. Let's see the holistic solution.

```
Code example 37 Holistic get the default quantity per type Oracle WITH types_quantities AS (
      SELECT SUBSTR(pt.Product_Type_Code, 1, 1) AS Type_Code,
      CASE WHEN p.Make_Flag = 1 THEN p. Default_Quantity * p.Default_Quantity
      WHEN p.Make_Flag = 0 THEN 2 * p.Default_Quantity
      ELSE NULL END AS Current_Qtty
      FROM Product_Types pt INNER JOIN Products p
      ON (p.Product_Type_Id = pt.Product_Type_Id)
      WHERE SUBSTR(pt.Product_Type_Code, 1, 1) = 'C'
      UNION ALL
      SELECT SUBSTR(pt.Product_Type_Code, 1, 1) AS Type_Code,
      CASE WHEN p.Make_Flag = 1 THEN p. Default_Quantity * p.Default_
      Quantity - p.Default_Quantity
      WHEN p.Make_Flag = 0 THEN 3 * p.Default_Quantity
      ELSE NULL END AS Current_Qtty
      FROM Product_Types pt INNER JOIN Products p
      ON (p.Product_Type_Id = pt.Product_Type_Id)
      WHERE SUBSTR(pt.Product_Type_Code, 1, 1) = 'D'
)
SELECT Type_Code, SUM(Current_Qtty) AS Current_Qtty
FROM types_quantities
GROUP BY Type_Code;
```

As you can see this is a simple SQL statement, with the help of the “with clause”. A similar solution can be offered in SQL Server too. Still, maybe the “with clause” is not implemented yet in all the database systems.

Let’s see another implementation for this practice using the SQL Server database. The atomic solution will be very similar with the Oracle solution. Let’s look at it first.

Code example 38: Atomic get the default quantity per type SQL Server

```
       CREATE PROCEDURE Atomic_Get_Qtty_Per_Type
       AS
       DECLARE @v_Product_Type_Code VARCHAR(5),
       @v_Make_Flag INT, @v_Default_Quantity INT,
       @v_First_Letter_Type CHAR(1),
       @v_Current_Qtty INT, @v_Current_Qtty_C INT, @v_Current_Qtty_D INT;
       DECLARE c_Get_Products_Qtty CURSOR FOR
       SELECT pt.Product_Type_Code, p.Make_Flag, p.Default_Quantity
       FROM Product_Types pt INNER JOIN Products p
       ON (p.Product_Type_Id = pt.Product_Type_Id);
BEGIN
       SET @v_Current_Qtty_C = 0;
       SET @v_Current_Qtty_D = 0;
       SET @v_Current_Qtty = 0;
       OPEN c_Get_Products_Qtty;
       FETCH NEXT FROM c_Get_Products_Qtty
       INTO @v_Product_Type_Code, @v_Make_Flag, @v_Default_ Quantity;
       WHILE @@FETCH_STATUS = 0
       BEGIN
           SET @v_First_Letter_Type = SUBSTRING (@v_Product_Type_ Code, 1, 1);
           IF @v_First_Letter_Type = 'C'
BEGIN
           IF @v_Make_Flag = 1
```

```
            SET @v_Current_Qtty = @v_Default_Quantity * @v_Default_ Quantity;
        ELSE IF @v_Make_Flag = 0
            SET @v_Current_Qtty = 2 * @v_Default_Quantity;
        END
        ELSE
    BEGIN
    IF @v_Make_Flag = 1
    SET @v_Current_Qtty = @v_Default_Quantity * @v_Default_ Quantity - @v_Default_Quantity;
        ELSE IF @v_Make_Flag = 0
        SET @v_Current_Qtty = 3 * @v_Default_Quantity;
    END
    IF @v_First_Letter_Type = 'C'
        SET @v_Current_Qtty_C = @v_Current_Qtty_C + @v_Current_Qtty;
    ELSE
    SET @v_Current_Qtty_D = @v_Current_Qtty_D + @v_Current_Qtty;
    FETCH NEXT FROM c_Get_Products_Qtty
    INTO @v_Product_Type_Code, @v_Make_Flag, @v_Default_Quantity;
    END
    CLOSE c_Get_Products_Qtty;
    DEALLOCATE c_Get_Products_Qtty;
    PRINT('The total default quantity for the products with C type is ' +
CAST(@v_Current_Qtty_C AS VARCHAR(20)));
    PRINT ('The total default quantity for the products with D type is ' +
CAST(@v_Current_Qtty_D AS VARCHAR(20)));
END;
GO
```

You can easily see the similarities between Oracle and SQL Server, at least in the simple examples of atomic approaches. With this, I want to illustrate the fact that the style is similar even if the approaches are atomic. Still, there are some differences but these can be easily passed. You can check the results and you should see the same values.

The 'with clause' can be used in SQL Server too. Still, I rather prefer to use a classic temporary table instead. The reason is that sometimes we should manipulate the data before getting the results and sometimes the "with clause" can be insufficient. Let's see this last example in the holistic approach.

Code example 39 Holistic get the default quantity per type SQL Server

```
CREATE PROCEDURE Holistic_Get_Qtty_Per_Type
AS
BEGIN
    CREATE TABLE #types_and_quantities (Type_Code_Prefix CHAR(1),
    Current_Qtty INT);
    INSERT INTO #types_and_quantities (Type_Code_Prefix, Current_ Qtty)
    SELECT SUBSTRING (pt.Product_Type_Code, 1, 1) AS Type_Code_ Prefiix,
    CASE WHEN p.Make_Flag = 1 THEN p.Default_Quantity * p.Default_ Quantity
    WHEN p.Make_Flag = 0 THEN 2 * p.Default_Quantity
    ELSE NULL END AS Current_Qtty
    FROM Product_Types pt INNER JOIN Products p
    ON (p.Product_Type_Id = pt.Product_Type_Id)
    WHERE SUBSTRING (pt.Product_Type_Code, 1, 1) = 'C';
    INSERT INTO #types_and_quantities (Type_Code_Prefix, Current_ Qtty)
```

```
        SELECT SUBSTRING (pt.Product_Type_Code, 1, 1) AS Type_Code,
        CASE   WHEN   p.Make_Flag   =   1   THEN   p.Default_Quantity   *   p.Default_   Quantity   -
  p.Default_Quantity
        WHEN p.Make_Flag = 0 THEN 3 * p.Default_Quantity
        ELSE NULL END AS Current_Qtty
        FROM Product_Types pt INNER JOIN Products p
        ON (p.Product_Type_Id = pt.Product_Type_Id)
        WHERE SUBSTRING (pt.Product_Type_Code, 1, 1) = 'D';
        SELECT Type_Code_Prefiix, SUM(Current_Qtty) AS Current_Qtty
        FROM #types_and_quantities
        GROUP BY Type_Code_Prefix;
  END;
  GO
```

The difference is the fact that, if the algorithm would have been even more complicated, this temporary table could have been updated several times, eventually combined with other temporary tables. This approach may not have the best performance sometimes, but it is a holistic approach. Very rarely, this style of work with temporary tables, if required due to the complexity, will have a lower performance comparing with the atomic approach. This is one more scenario for a holistic solution that can be used against a classic atomic and procedural solution.

With this, we close this chapter and we approach the end.

In the last chapter, I'll try to conclude most of the things that have been discussed and I will try to add some more examples to illustrate the two styles of development.

Chapter 8

ROW TRIGGERS. WHEN SHOULD WE FOLLOW THE ATOMIC WAY? SOME FINAL REFLECTIONS AND THOUGHTS!

THE USE OF ROW TRIGGERS: ANOTHER COMMON ATOMIC SOLUTION USED IN EXCESS

I discussed a lot about scalar functions and I consider them one suitable facility for the atomic approach. That is not the only one. Apart from scalar functions, a feature coming from structured programming, there is another one, coming from the database area. I am referring to table triggers, of course.

When learning database programming and languages like PL SQL or Transact SQL, the programmer starts with the basics and finishes with the set of procedural objects like functions and procedures. These two are already well known from the classic languages so he can understand them relatively quickly. However, soon after learning these types of procedural objects, the programmer learns about a new type of object, specific to databases and especially specific to tables: triggers.

The developer is sometimes seduced by the idea of implicit and automatic execution of the trigger. The main difference between a trigger and a stored procedure is well known and it resides in execution. If a stored procedure is always executed manually and we have full control over the execution, which means we know exactly when we execute it, we cannot say the same thing about the trigger. The trigger and I am referring to the most common type of trigger, table trigger is always executed indirectly and automatic based on a certain event in the table. The events are generally “DML” statements. Consequently, we can have insert, update or delete triggers. The trigger is executed automatically based on an action against the table to which the trigger belongs. The trigger is a dependent object because it is linked to the table. In a way, the trigger is like a constraint, a similar type of object. Moreover, the complexity of the trigger is high, we can do very complex data manipulation inside a trigger and the complexity of a trigger is similar to the complexity of a stored procedure.

The programmer is firstly seduced by the fact that the execution is automatic and he can remove this task completely from his head and leave this for the table event. Sometimes it is required.

Afterwards, the programmer learns about the fact that a trigger can be a holistic trigger or an atomic trigger, so a statement or row trigger. The statement triggers act holistically so they are in concordance with the holistic approach. On the other hand, the row trigger is very similar with the scalar function and it is an ideal facility for the applications developers working inside relational databases. The row triggers are intensively used by them and not only. Maybe too intensively, I might say.

The row triggers offer the advantage of allowing direct access per every field and every row, an ideal facility for application developers working atomically in relational databases. Sometimes, row triggers might be useful. Very often, row triggers can be avoided. It is well known that the performance of triggers is not very poor and normally these should be avoided and used as a last resource. Still, because these are so intimately related to the atomic style of development, especially row triggers, these are intensively used in a variety of data oriented software applications. The programmers will see right away that row triggers match their atomic vision against data and they start using these intensively. This way, you can see a large variety of databases filled with row triggers and scalar functions!

One common example is the one where an artificial identifier should be generated from a sequence. Sequences are great facility, an independent logical object responsible for the generation of numbers starting from a minimum value and growing with a step specified at the time of creation. Sequences are very common and they are an ideal method for artificial numeric generation.

Unfortunately, the application developer likes the combination between the insert trigger and the sequence, actually better said between a before insert trigger and a sequence. As you perhaps know, a trigger is also classified by the timing compared to the event that raises the trigger. The trigger can be before the event, after the event of instead of the event, for views. The before insert triggers and sequences are one favorite method for the population of artificial identifiers.

Let’s go back to chapter 5 and review the Oracle atomic full transfer, first exercise. The solution was written in the Code example 06. You can see how the identifiers for English and French languages were generated in the logic in the loop. Now we will change this logic and use some sequences for the keys generation for the reporting tables. We can create either one sequence for both reporting tables or two sequences, one per language. I prefer to create two dedicated sequences, let’s see them below:

Code example 40 Create sequences

```
CREATE SEQUENCE English_CL_Id_Seq START WITH 1 INCREMENT BY 1
/
CREATE SEQUENCE French_CL_Id_Seq START WITH 1 INCREMENT BY 1
/
```

We will use these sequences to generate the new values for the artificial keys for the reporting tables. For that, we will create two triggers, one per table. Let’s see the triggers and then analyze the logic inside, actually very simple and classic.

Code example 41 The set of before triggers

```
CREATE TRIGGER English_CL_Id_Tg
BEFORE INSERT ON English_European_Countries
FOR EACH ROW
DECLARE
BEGIN
      IF :new.English_CL_Id IS NULL THEN
          :new.English_CL_Id := English_CL_Id_Seq.nextval;
      END IF;
END;
/
```

```
CREATE TRIGGER French_CL_Id_Tg
BEFORE INSERT ON French_European_Countries
FOR EACH ROW
DECLARE
BEGIN
      IF :new.French_CL_Id IS NULL THEN
          :new.French_CL_Id := French_CL_Id_Seq.nextval;
      END IF;
END;
/
```

These two triggers are similar, one for English and one for French. These triggers will be executed every time before a new row is inserted into the base table. The new value will be taken from the sequence. Afterwards, it will populate the artificial identifier. This is a very common technique and many application developers are using it, especially Oracle developers. Of course, using sequences with triggers means that the sequence will be the only accepted method for the identifiers population. We cannot combine with any other methods.

Let’s update the code example 06 accordingly and generate the full transfer again. Let’s see it and analyze again on the topic.

```
Code example 42: Oracle Atomic Full transfer, with triggers
CREATE PROCEDURE Atomic_Full_Transfer_Country_t
(p_Language_Name VARCHAR)
AS
      v_Country_Name VARCHAR2(50);
      v_Country_Code VARCHAR2(3);
      v_Language_Category VARCHAR2(10);
      CURSOR c_Get_Countries (p_Language VARCHAR2) IS
      SELECT c.Country_Name, c.Country_Code, cl.Language_Category
      FROM Countries_Languages cl INNER JOIN Languages l
      ON (l.Language_Id = cl.Language_Id)
      INNER JOIN Countries c
      ON (c.Country_Id = cl.Country_Id)
      WHERE l.Language_Name = p_Language;
BEGIN
      IF p_Language_Name = ‘English’ THEN
      DELETE English_European_Countries;
      ELSIF p_Language_Name = ‘French’ THEN
      DELETE French_European_Countries;
      END IF;
      OPEN c_Get_Countries (p_Language_Name);
      LOOP
      FETCH c_Get_Countries
      INTO v_Country_Name, v_Country_Code, v_Language_Category;
      EXIT WHEN c_Get_Countries%NOTFOUND;
      IF p_Language_Name = ‘English’ THEN
          INSERT INTO English_European_Countries (Country_Code, Country_Name, Language_Category)
          VALUES (v_Country_Code, v_Country_Name, v_Language_Category); ELSIF p_Language_Name =
‘French’ THEN
```

```
            INSERT INTO French_European_Countries (Country_Code, Country_Name, Language_Category)
            VALUES (v_Country_Code, v_Country_Name, v_Language_Category);
  END IF;
  COMMIT;
  END LOOP;
        CLOSE c_Get_Countries;
  END Atomic_Full_Transfer_Country_t;
  /
```

If you compare this version with the version in Code example 06, you will see the differences. The identifier is not visible anymore: it is updated in the back, by the trigger. If you look at the target tables, you will not see the key because the trigger updates the key.

This technique is a disaster if medium to large sets of data are handled in a data transfer. When affecting data sets, and using row triggers, you are in a dilemma. Triggers are not visible you need to search for them. Try to imagine that you work holistically and set based, you are affecting 100 rows now in a clean and pure SQL logic, holistic and set based. Even more, you are very proud of your style and you know that the flow is set based and as fast as it can be. Still, the logic is very slow and you don't know why! Suddenly, you realize that you have a row trigger that change the entire flow, instead of being a set based flow and a holistic style it is transformed into an atomic flow. Let's rewrite the code example 08, the Oracle full transfer.

Let's see it now:

```
  Code example 43: Oracle Holistic Full transfer, triggers
  CREATE PROCEDURE Holistic_Full_Transf_Country
  (
  p_Language_Name VARCHAR
  )
  AS
  BEGIN
        DELETE English_European_Countries
        WHERE p_Language_Name = 'English';
        DELETE French_European_Countries
        WHERE p_Language_Name = 'French';
            INSERT INTO English_European_Countries (Country_Code, Country_Name, Language_Category)
        SELECT c.Country_Code, c.Country_Name, cl.Language_Category
        FROM Countries_Languages cl INNER JOIN Languages l
            ON (l.Language_Id = cl.Language_Id)
        INNER JOIN Countries c
            ON (c.Country_Id = cl.Country_Id)
        WHERE l.Language_Name = p_Language_Name AND p_Language_Name = 'English';
            INSERT INTO French_European_Countries (Country_Code, Country_Name, Language_Category)
        SELECT c.Country_Code, c.Country_Name, cl.Language_Category
        FROM Countries_Languages cl INNER JOIN Languages l
            ON (l.Language_Id = cl.Language_Id)
        INNER JOIN Countries c
            ON (c.Country_Id = cl.Country_Id)
        WHERE l.Language_Name = p_Language_Name p_Language_Name = 'French';
```

```
      COMMIT;
END Holistic_Full_Transf_Country;
/
```

This example looks like an example of holistic style of development. Actually it isn’t! Despite the fact that the logic itself is set based and holistic, the row triggers change everything and the logic is atomic instead of being holistic.

Personally, I rarely use triggers, especially row triggers. They act per row, and they have all the disadvantages of the atomic vision of development. Of course, sometimes are necessary. You can decide yourself but be aware of these disadvantages and especially be aware of the fact that a set based logic, when associated with row triggers, transforms everything. The set based is like a shadow, a mask and the people behind the mask are revealed by the row trigger!

THE ATOMIC APPROACH SHOULD BE USED, WHENEVER IS NECESSARY!

As I often mentioned, sometimes the atomic approach might be necessary. There are situations when we need to think atomically, to open cursors and move the data at the row level into variables, do various manipulations and all the rest of the staff. The combination SQL and procedural, the combination holistic and atomic, all together are composing the database programming language. I never imagined and I never considered that the variety of procedural facilities and atomic ones are not to be used. A database programming language is composed by all of them and all the features are needed, more or less often. Still, it is very important to remember, whenever we are developing inside a relational database, the simple fact that we are inside a relational database! That always means, simply, rows and columns.

The set based approach and the holistic style of development are referring to tendencies and statistics. In most of the cases, we should answer to our business questions using set based solutions and holistic answers. However, most of the cases do not exclude exceptions. The data set is composed of a number of rows and sometimes, to be able to solve our problems, we need to move back to the row level and think atomically. The row based approach and the atomic styles of development are useful solutions but they are to be used as last resources and not as default solutions.

Let’s analyze one type of scenario where an atomic solution can be used.

Business / technical description

WE WANT TO DISPLAY THE LIST WITH ALL THE LANGUAGES AND THE COUNTRIES ATTACHED, AS PRINCIPAL OR SECONDARY. WE NEED TO CONCATENATE IN A STRING THE LIST OF COUNTRIES SEPARATED BY COMMAS, FOR EACH CATEGORY.

This is an example that can be solved atomically. This is one of these situations where the programmer can work atomically. Very often, he needs to. He may try to look for set based facilities like analytics functions (for example row number). However, if he is not able to find these he may think at the cursor and he may start to divide everything at the row level.

When is the division actually necessary? There are situations when we are forced to divide at the row level and work atomically.

From my experience, this division may occur when we are forced to do various manipulations row by row and do whatever calculations, store some intermediates results. I don’t believe we can state any rule for the division to be accepted, things are related to the particularities of the situation. Still, in most of the cases, the division is required when we need to see things row by row. In this case, a simple SQL statement is not enough.

Let’s analyze this example. We have the three tables: one with the languages, one with the countries and their associations. For every language, we have a list of countries and every language is either principal or secondary for the country. The data is highly normalized. From this design, we need to get a kind of report, a situation per language with two lists. The first list will contain all the countries where the language is principal and the second list will contain the countries where the language is secondary. For that, we need to be able to move into the tables, in a row-by-row approach, to concatenate the countries, store them somehow, and finally generate the report. This example illustrates the need for the atomic approach in certain situations.

Let’s see the version for SQL Server.

Code example 44: display the list with languages and countries

```
CREATE PROCEDURE Atomic_List_Of_Countries
AS
      DECLARE @v_Country_Name NVARCHAR(50);
      DECLARE @v_Language_Category NVARCHAR(10);
      DECLARE @v_Language_Name NVARCHAR(50);
      DECLARE @v_Language_Id INT;
      DECLARE @v_List_Of_Countries_Main NVARCHAR(4000);
      DECLARE @v_List_Of_Countries_Sec NVARCHAR(4000);
      DECLARE @v_Previous_Language_Name NVARCHAR(50);
      DECLARE c_Get_Languages CURSOR FOR
      SELECT l.Language_Id, l.Language_Name
      FROM Languages l
      WHERE EXISTS
      (
          SELECT 1 FROM Countries_Languages cl
          WHERE cl.Language_Id = l.Language_Id
      )
      ORDER BY 2;
BEGIN
      CREATE TABLE #List_Of_Countries (Language_Name NVARCHAR(50), List_Of_Countries_Main
NVARCHAR(4000), List_Of_Countries_Sec
NVARCHAR(4000));
      OPEN c_Get_Languages;
      FETCH NEXT FROM c_Get_Languages
      INTO @v_Language_Id, @v_Language_Name;
      WHILE @@FETCH_STATUS = 0
      BEGIN
          SET @v_List_Of_Countries_Main = ‘’;
          SET @v_List_Of_Countries_Sec = ‘’;
          DECLARE c_Get_Countries CURSOR FOR
          SELECT c.Country_Name, cl.Language_Category
          FROM Countries_Languages cl INNER JOIN Countries c
          ON (c.Country_Id = cl.Country_Id)
          WHERE cl.Language_Id = @v_Language_Id
          ORDER BY cl.Language_Category, c.Country_Name;
          OPEN c_Get_Countries;
          FETCH NEXT FROM c_Get_Countries
          INTO @v_Country_Name, @v_Language_Category;
          WHILE @@FETCH_STATUS = 0
          BEGIN
          IF @v_Language_Category = ‘MAIN’
              SELECT @v_List_Of_Countries_Main = @v_List_Of_Countries_Main + @v_Country_Name +
‘,’
```

```
            ELSE
            SELECT @v_List_Of_Countries_Sec = @v_List_Of_Countries_ Sec + @v_Country_Name + ‘,’
            FETCH NEXT FROM c_Get_Countries
            INTO @v_Country_Name, @v_Language_Category;
        END
        CLOSE c_Get_Countries;
        DEALLOCATE c_Get_Countries;
        IF LEN(@v_List_Of_Countries_Main) > 1
            SET @v_List_Of_Countries_Main = SUBSTRING (@v_List_Of_ Countries_Main, 1,
LEN(@v_List_Of_Countries_Main) - 1);
        IF LEN(@v_List_Of_Countries_Sec) > 1
            SET @v_List_Of_Countries_Sec = SUBSTRING (@v_List_Of_ Countries_Sec, 1,
LEN(@v_List_Of_Countries_Sec) - 1);
            INSERT INTO #List_Of_Countries (Language_Name, List_Of_ Countries_Main,
List_Of_Countries_Sec)
            VALUES (@v_Language_Name, @v_List_Of_Countries_Main, @v_ List_Of_Countries_Sec);
            FETCH NEXT FROM c_Get_Languages
            INTO @v_Language_Id, @v_Language_Name;
        END
        CLOSE c_Get_Languages
        DEALLOCATE c_Get_Languages;
        SELECT * FROM #List_Of_Countries
        DROP TABLE #List_Of_Countries
END
GO
```

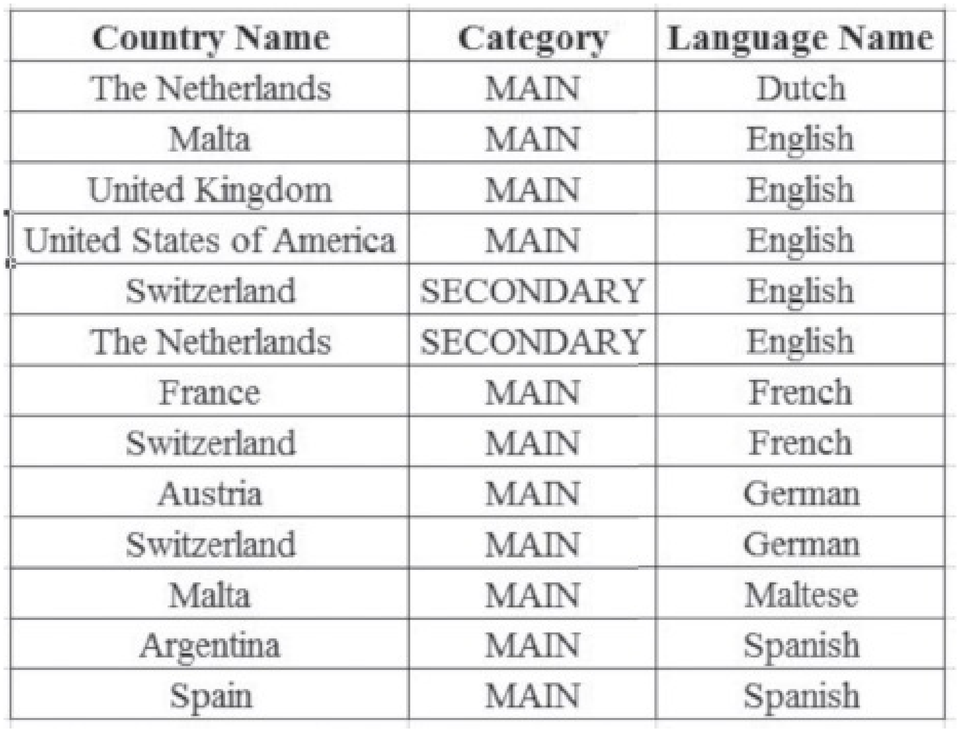
Let’s analyze the logic above. Of course, that this could have been done in many ways, this is one of the many possible solutions.

Even if the solution is atomic, we are still in the world of data. This means that again everything starts from a data set. Let’s call it the base data set or the detail data set.

Let’s see this base data set:

```
Code example 45: the base data or detail set
SELECT c.Country_Name, cl.Language_Category, l.Language_Name
FROM Countries_Languages cl INNER JOIN Languages l
      ON (l.Language_Id = cl.Language_Id)
INNER JOIN Countries c
      ON (c.Country_Id = cl.Country_Id)
ORDER BY l.Language_Name, cl.Language_Category, c.Country_ Name;
```

Let’s look at the data because this is the key to the solution.



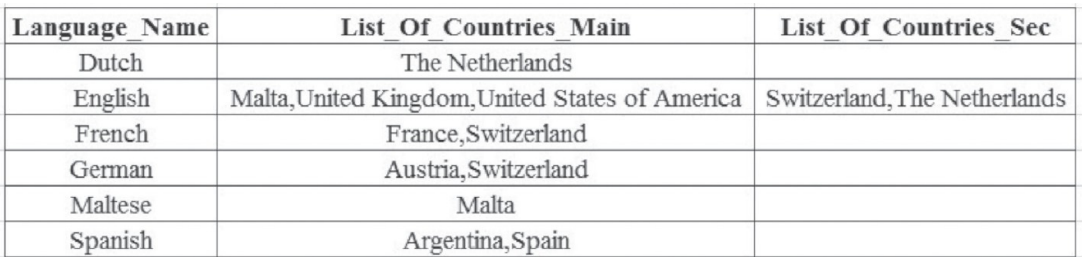
Country Name	Category	Language Name
The Netherlands	MAIN	Dutch
Malta	MAIN	English
United Kingdom	MAIN	English
United States of America	MAIN	English
Switzerland	SECONDARY	English
The Netherlands	SECONDARY	English
France	MAIN	French
Switzerland	MAIN	French
Austria	MAIN	German
Switzerland	MAIN	German
Malta	MAIN	Maltese
Argentina	MAIN	Spanish
Spain	MAIN	Spanish

This data set is the starting point for the solution. Starting from here, we need to generate the report. We can see a variety of languages and associated countries, with the associated category. For example, for English, the countries are Malta, UK and US as main languages. There are also Switzerland and Netherlands as secondary countries. We can imagine the list is larger and we can see that, to be able to get the list, we need to position in the language and then, from there, we need to start concatenate the countries based on category. We need to concatenate the countries for main and for secondary. A simple SQL seems to be not enough because the list of countries, in order to be obtained, requires row-by-row access, requires row division.

Let’s see the proposed solution, see the above code example 44:

1. We declare one cursor for the languages. We will store the languages that have at least one country assigned to it. We store from this cursor, for every row, the language name and language identifier.
2. We create a temporary table, in the pure SQL Server style, to get the results.
3. We declare two variables dedicated for storing the lists of countries, based on category, one for main and one for secondary.
4. For every language, we declare another cursor and use with the list of countries. For every language, we store in dedicated variables the country and the category.
5. In the second cursor, the inner cursor, we start concatenation and link the countries, separated by a comma.
6. When we are done, and move back to the outer cursor, we remove the last comma from the string, from both lists.
7. Being in the outer cursor, with the languages, we add the data from the dedicated variables and populate the reporting table.
8. In the end, we display the desired list.

Let’s see the results:



Language_Name	List_Of_Countries_Main	List_Of_Countries_Sec
Dutch	The Netherlands	
English	Malta,United Kingdom,United States of America	Switzerland,The Netherlands
French	France,Switzerland	
German	Austria,Switzerland	
Maltese	Malta	
Spanish	Argentina,Spain	

This example shows that, sometimes, we can use the atomic approach and sometimes we need to follow the atomic approach. Cursors are a great facility and the main reason, in my opinion, is exactly this one. The cursors, in combination with the use of loops and the use of fetch allow us to position ourselves from one row to another and do various manipulations. It would never cross my mind to deny their utility. What I intend to show almost everywhere in this book is the fact that we need to be aware that we are in a database, we are affecting data sets and we need to keep in mind the holistic vision first. That does not mean we should not be aware of the possibility of division from data set to data row, when necessary.

It is necessary when the business requirements are of a nature that forces us to position at the row level and do various manipulations. These manipulations can be handled by various set based facilities like row number, but if we do not have such facilities or if we are not able to find them, we have always the option to use cursors and move the context to row level.

Sometimes we need to handle things atomically but we don’t need to think atomically unless it’s necessary, this is the meaning of the entire book. Considering our house, the house of relational databases and the house of rows and columns, and considering that the main goal is to handle data sets, it’s even against the logic, reason and common sense to think atomically by default. This is what I wanted to show with all my examples with the SQL shop. Apart from performance, portability, simplicity and naturalness, it is obvious and normal to try to think holistically and SQL, due to the nature of data. This example shows that we have always on our minds an atomic solution, if the set based approach is not sufficient.

SOME FINAL REFLECTIONS AND THOUGHTS

In the end of this final chapter, I propose to share some thoughts taken from my experience as a database specialist, mainly database developer.

Without considering myself a guru, who I am definitely not, I am a passionate and dedicated SQL person. I spent many years in SQL development and I tried to develop my own style. During these years of development, I gathered many experiences, discovered many ideas, and rediscovered the wheel many times. I don’t dare to say I am original and I know very well that most of the ideas explained here are well known by many professionals all over the world. However, I hope to offer a better view and I hope to add some clarifications to some of our realities.

Apart from the distinction between the two styles of development, I want to share some other thoughts and ideas from my experience as SQL developer. Maybe some of the developers will follow some of my advices, maybe not. We leave in a free world, so every reader of this book will take the best decision in concordance with his personality and, why not, with this style!

THE CONCEPT OF SQL TEMPLATE

When considering the holistic approach for our database and decided to develop accordingly, depending on the type of system, we should try to follow the classic principles of programming. Working holistically does not mean ignoring the classic principles of programming, but trying to combine what is more suitable from both. For example, working holistically does not mean having large procedures with a lot of logic inside. We may have that due to the business requirements, but not because of the holistic style of development.

The principle of division, so popular in programming like anywhere in life, is available in the holistic approach too. This principle states that, if a problem is complex, it can be divided into simpler problems; and these ones eventually can be divided too. The division may occur in time not necessarily from the beginning. For example, we may generate a complex logic and we are not aware from the beginning about the principle of division. Very often, we are focused on the general task, the big problem without being aware of the principle of division. When we are done with the logic and we look at our masterpiece, we realize that we may divide the logic into many pieces like functions and procedures. Afterwards, when we have time, we divide the work accordingly. However, the principle of division has the same value for both atomic and holistic approach. I don't see any reason for the holistic approach to do not follow this principle. Still, working holistically will automatically offer a more consistent perspective and maybe, sometimes we will have fewer objects and the logic may appear more condensed. For example, not using scalar functions as much as possible will minimize the number of these kinds of routines.

On the other hand, considering that we are in a database and we work holistically, we have a variety of SQL statements to handle and manage. Very often, some of these SQL statements are similar. Sometimes they are not. Anyway, we may consider the use of SQL templates. At least in a replication or a data migration system, that kind of specific software application that I consider as being very suitable for the holistic approach, we may consider the use of SQL warehouses. A SQL warehouse is a collection of SQL templates. I name it template because it is used repeatedly with minor changes that can be various parameters and I name it SQL because it is a pure SQL statement. That collection of statements can embrace a large number of SQL statements that can be executed in one context or another. At execution time, the SQL templates will receive effective values for the parameters.

The holistic approach means the use of SQL. I estimate that most of the logic will be composed from SQL statements. Especially if we are in a specific system, where the goal is always the same, moving data from A to B, the data movement process will and can be managed in pure SQL. Consequently, the only thing you need to do is to get the set of SQL statements. For a better organization, we can gather many SQL statements in the warehouse and call one template or another from various places in the specific software application.

The SQL warehouse can be placed in a table. It will become data and metadata in the same time. The metadata table can contain many fields that will be related to the template and one field with the template itself. The SQL templates will be executed using dynamic SQL, of course, in various context of the software application.

The custom metadata table is one possible house for the SQL templates. The warehouse can also be placed in a stored procedure with a series of parameters. The stored procedure will be executed in one context or another. I used both approaches. Now, if I have to consider both, I would prefer the stored procedure because I see only embedded SQL. I see all the templates here, I can easily read and understand everything, and I can eventually compare various templates. Using the metadata table, all the templates will be hidden. To look into one SQL template or another I need to query the table, take the template separately in another window and debug that template. The procedure is not so convenient. Finally, both methods are acceptable and are good places for the SQL templates warehouse. I am planning to show you some of these techniques when I will describe a replication system written in pure SQL, my goal for my future book.

Thinking holistically is the first thing that should be acquired, especially in a specific system like an ETL or a data migration system. However, not exclusively, the holistic style and vision are welcomed in the classic software system too, only that there sometimes things are atomic due to the nature of the business. For example, we are in an invoicing system. We are now positioned in an invoice, one invoice. We are now updating that invoice. We are already in one invoice, so the level is close to the atomic level. Consequently, it does not matter too much if the approach is holistic or atomic. Still, if we consider the details of the invoice, and maybe the invoice has 200 details, it is one thing to update 200 details in one action in a holistic manner and another to try to affect every detail in a cursor. As you can see, even in a classic system is recommended to follow the holistic approach, but it is not always necessary. However, it is necessary in a specific system, where the goal is to move data between various systems. In these kinds of software systems, at the database level, we should always consider the holistic approach. Having the fair holistic vision and having so much SQL, we can classify the statements and try to organize them properly. For this reason, we may think to organize them in these kinds of SQL warehouses, embedded in either custom metadata tables or stored procedures.

The fact that a large number of SQL templates are grouped in one single place like a stored procedure is a facility close to the classic vision of programming. Based on business criteria, many SQL warehouses can be created in a system. For example, we might have a staging area between the source and the target system. We can have one SQL warehouse for all the templates that are used for the data movement from the source to the staging and we can have another warehouse composed of most SQL templates used for the movement from the staging to the target. This is just an example.

In a specific system where the goal is simply to move data from one system to another, the data should be moved in data sets as much as possible. The main type of statement responsible for the data set flow is the SQL statement. The decision to group the SQL statements according to certain criteria is a good decision and it is going to increase the level of organization of the system.

WRITING HORIZONTALLY OR VERTICALLY: A DECISION TO BE TAKEN!

The programmer style of development is defined by a set of principles and rules, described in the models and paradigms. We discussed about that and we somehow determine the two alternate styles in the relational database. It is a point of view. We can imagine even more! The style of development is also relative to the area of interest and work. It is one thing to develop mainly in Oracle and another to develop mainly in Java. The style is influenced by the technologies used in our projects, and the style is dynamic, it may change in time. That is one reason for which I am optimistic and I hope some application developers will reconsider their work in the database and try to change something. Even an intention to change is a victory for the database, in general, and for me, in particular.

Working holistically or atomically is part of each ones style of programming in the database. This decision and this tendency, to work holistic or atomic, is a major component of the style of development. The style is reflected in the code, one may easily recognize it.

If you see cursors over cursors for every operation, and a variety of scalar functions or row triggers, it means that you are in front of an application developer working in the database in his own way. If you see data sets manipulated holistically almost everywhere you notice the opposite style. This is the most important criteria for the style determination.

Moreover, this is not the only one. There are others signs that may indicate one style or another. Some of them are critical others are not.

One secondary characteristic of a style of development is the way we effectively write. The aspect of our code is an important component but has its degree of subjectivity. That is why this topic is not regulated and it should not be regulated, indeed. Still, we can discuss about it. We cannot force people to write in a certain way, but we can explain that a certain writing rule is better than another one! A procedural object, like a procedure or function, can have hundreds of lines or even more. The most important thing for this object is to work properly. This is a trivial statement. However, it’s not only that! The procedure needs to be intelligible. Others programmers should be able to understand it. Even us, sometimes, we do not understand our code! One reason is the way we write.

I will explain using a clear and common example. In the database, we have many SQL statements. Let’s imagine one procedure with 10 insert statements, with 5 update statements, with 7 delete statements, and others procedural logic in it apart from the SQL. I noticed a bad habit, and I am sorry for being so tough here. Some programmers write the SQL statements vertically. This means that, if you have an insert statement, and you have 20 columns, you will have between 40 and 50 lines for this insert statement. The vertical writing means to specify one column per line and effectively write vertically. Imagine that you have 10 insert statements and you will have between 400 and 500 lines for these insert statements. If you want to understand this logic, is almost impossible. I cannot comprehend how these people understand what they did, is beyond my capacities!

Let’s see one example. Imagine you have one insert statement in a procedure. Let’s see the two ways you can write the insert statement.

```
Code example 46 Writing vertically
INSERT INTO Countries_Languages (
      CL_Id,
      Country_Id,
      Language_Id,
      Language_Category
      )VALUES (
      14,
      2,
      2,
      ‘SECONDARY’);
-- Writing horizontally
INSERT INTO Countries_Languages (CL_Id, Country_Id, Language_Id, Language_Category)
VALUES (14, 2, 2, ‘SECONDARY’);
```

Imagine you are in a specific system where you have many insert, update and delete statements and you write everything vertically. Will you be able to understand anything from the logic? It will be extremely difficult. See the difference in the example above and answer me to one question. Why would you write an insert statement vertically? Give me one good reason. Still, I need a rational reason! Your taste only is not a good reason in this case. For sure, the intelligibility of the procedure will be badly affected.

The insert statement is one statement. If you are declaring 10 variables, you may add them in 10 lines, each variable declaration on one line. Every variable is distinct and deserves its own line! By contrast, a column in an insert or a value specified is not a distinct component to be added in distinct lines. One insert statement is one single statement. The purpose is to be able to understand the insert statement. For that, when writing horizontally, it is much easier to see and understand what it is about. You may follow each column to each corresponding value; you may check the data types. Eventually, you will check the compatibility. An insert statement, to be intelligible, should be written mostly horizontally.

Another possible reason for which an insert statement is written vertically is the fact that it is the mirror of the tool. Some development tools generate by default the insert statements vertically. Some tools present us with a certain way to write. That does not mean we should write blindly without thinking. Writing SQL vertically is a catastrophe for our logic because it will make our code completely unreadable. For example, in an insert statement, one should be able to follow the correspondences between every column and every associated value. Writing vertically makes this task impossible. The tools should not drive us but we should drive the tools. A good programmer should write in his way according to his reason and should not follow a certain way because he saw it in a tool.

Regarding the style in the database, I consider that writing SQL statements horizontally is extremely important for the readability of our software. We need to do everything to be able to understand what we write. The SQL statement is the most important type of statement and we need to do everything to understand it exactly. Even if it seems a matter of form and not substance, it is important. In addition, one more thing, writing horizontally does not mean writing 30 columns in one line, we will have the same problems and the same lack of understanding. Writing horizontally means writing that number of columns or expressions that fill the line. The important matter is to have the proper visibility and to understand the code without the need to move to the right all the time with the cursor.

Sometimes, the programmers are generating the logic from the tools. They have an initial version, and they update everything after. I never did that but I know some people who do this. This is another reason for the existence of unreadable code. If the logic is very simple, you may leave with that. If not, it is difficult to work in this style.

Writing vertically may also show a kind of lack of respect for the SQL statement. The developer should understand that the SQL statement is one statement, is a unit of work. He needs to do everything in his power to catch the whole, or as much as possible from it. A SQL statement can have hundreds of lines even in a horizontal manner anyway.

Writing horizontally is a decision based on reason and not on taste. Some programmers will say it is their taste. Even if this may be true to a certain extent, they should change this because all the arguments for intelligibility are in the favor of the horizontal writing of SQL statements. Taste should be a factor of decision for a programmer as long as it is not against reason, don’t you think?

THE SPECIFIC SOFTWARE APPLICATION SHOULD BE IMPLEMENTED HOLISTICALLY

A specific software application is very common nowadays, when most of the large enterprises have multiple systems and databases that should communicate to each other. Either a continuous transfer between operational systems, like a replication system or a data migration system, or a transfer between a set of operational systems and an analytics, like in an ETL process, the logic of data transfer between systems is more and more present everywhere.

These kinds of systems should be made by database developers and should not be let in the hands of pure application developers. I know many people may disagree with this categorical statement. I also recognize there is a degree of subjectivity in this statement. Actually I can rephrase it and affirm that these kind of systems should be made by either specialized database developers or by mixed developers with an open mind, so open that they should be able to write classically in the user interface and to change and write holistically and set-oriented in the database. These application developers, although being specialized in languages like Java or C++, are able to understand the simpler model of the database, they understand the set-based approach and they are able to change the way they write when switch to the database. There are very good programmers, brilliant minds and very flexible, that are able to do the switch appropriately.

Either a specialized database developer or a mixed developer, with a good vision against the database, is the proper person to work on an ETL, or a replication system or data migration system written in pure SQL. Both categories of professionals are able to think holistically and SQL, are able to understand and apply the set-oriented programming. These professionals will build proper systems of that type, the so-called specific systems. The first condition for this kind of specific system to work properly and to have a good performance is to be written correctly. If this condition is satisfied, we are fine. We are not necessarily happy, but fine. We have the tools to try to become happy.

By having the proper style in a specific system, and adopting the holistic style of development, we can analyze the logic in data sets and search for the weaknesses. There is no such thing like perfect software system of any type. Consequently, despite the fair style of programming, we still may have performance issues. Now at least we know that we are on the right path and we know that we may start to look further. Because the fair style of development and the set-based oriented style of development is the first condition for a good performance for a data migration process.

The holistic style of development means using mostly SQL. The entire data migration system will be composed mainly of SQL statements. The data is transferred, as it should be. Now the programmer will move to the next phase, trying to improve the performance of the SQL itself!

Regarding the SQL itself and the performance, there are so many possibilities that we need five books of this type to mention them all. Improving one SQL statement or improving certain logic composed of mainly SQL statements is a challenging task. Without entering a new topic now, we should only mention some things that have to be done.

THE SQL ITSELF CAN BE BETTER AND BETTER!

First, when mentioning SQL, we should have a deeper understanding of the SQL paradigm, apart from the language, apart from the business. As I already mentioned several times, we generally have two main goals when dealing with the software. We first need to implement the business. For example, in a specific system where we want to move some products from an ERP system to a production system, we need to make sure the products are moved correctly. We have ten products to be moved from the ERP system, we may check them all in the target system, attribute by attribute. If everything is accurate, we successfully complete the first and the most important goal. This is the most important thing to be done. Once this is completed, we look further and question the performance. If, for the ten products we wait one minute, we are maybe happy. If, for the ten products, we wait ten minutes we are unhappy even if we see the products in the target. The conclusion is that our second goal is the performance. Especially in the database, you can see the performance issues easily by looking at the response times.

If we have a log in the data migration system, we may find the steps with the performance issues. We will detect the place with the issue. We need to take the SQL or the set of SQL statements separately and try to investigate. The first phase of the investigation is to check if the SQL was accurate. Let’s see some of the performance checks that can be done against our SQL in the database.

When you have to choose between joins and subqueries is always better to use joins. In most of the cases, this is possible. An excessive use of subqueries and a replacement of the joins with subqueries will decrease the performance of the SQL statement. In SQL, the same result set can be achieved in many ways. A SQL programmer should know which technique has a better performance another one. Some things are quite well known, others could be found by looking at the execution plan of the statement. The decision between subqueries and joins should be taken, in most of the cases, in the favor of joins.

The use of union should be made with caution and always check to see if you can use a union all instead. As you know, the union operator will always involve a sort operation. This operation is very expensive. We should check the data sets whenever are grouped in a union. We should check if there is a need for duplicates removal. Sometimes we know that the data sets in the union are distinct, we may know that there are no duplicates. If that is true, we can replace the union with union all and we will have a much better performance. One union may take five minute and the union all against the same blocks may take five seconds!

Try to avoid the left outer join because the indexes may not work on the tables. Sometimes left outer join can be avoided or replaced with something else, add them only if they are really required by the business and they cannot be replaced by anything else.

Use the specific SQL to the vendor if it has a better performance than a more standard SQL. As I mentioned before, the SQL Server form of update is specific but it has a better performance than the more general form with the subquery. Many SQL statements are available in many different forms and you should try to know if the performance is the same or not.

There are more and more things to be said. Still, this is another topic and deserves a separate space. The most important thing is to try to write a clean SQL, from the business point of view but also from the performance point of view. For that, the developer should try to read and find more things about the database system he is dealing with.

The ability to read an execution plan is, of course, crucial for a good developer. Not all the developers know that. Still, if we want to be able to improve the performance, we should be able to read and understand the execution plan of a SQL statement. From what I noticed, if you write clean and you know the basics, you have all the chances to have a good performance.

There are many other things like indexes, partitions, parallelism, materialized views and other facilities. These facilities will improve the performance and the developer that knows all these is a true database developer.

PERFORMANCE, OH PERFORMANCE!
THIS BOOK IS FOR YOU!

Let’s conclude! Performance is the second goal in the database, the first one being the accurate implementation of the business. All the books mention performance, everyone says that performance is critical but, in reality, performance is generally neglected and we start looking into it when is too late, when we already have issues. In most of the projects, when we do the design, we are thinking at the immediate goal, to implement the target business.

We never think at performance from the beginning, we analyze it later when the software application is over and already implemented in production. For example, we have an invoicing system. We are thinking at the design, user interface design, database design, we analyze and analyze the invoicing business, but we rarely think seriously at performance. Despite the fact that most of the professional books recommend that performance should be carefully analyzed from the beginning we are only doing that very seldom. Unfortunately, it is like with our health. We begin to care about our health when we get sick and we start taking medicines, eventually we change our life style trying to improve our health.

I read many books about performance and I had the chance to work in this section many times. I am a contractor and very often contractors are called to solve performance issues that were there for years. There are many techniques for improving performance and many features. I already described some of my interventions on some of my projects. It is a valuable skill for a professional to be able to work on performance, to be able to improve it. You need to have a distinct type of knowledge for that, apart from simply development. In the field of databases, performance is a separate section and the database professionals involved in performance are half developers half DBA’s: they are the doctors of the databases, very appreciated and respected.

I read some magazines about the Nordics that they have the best hearts in the world. Their health is incredible and they are happy nations. I can tell you it is true because I had been in all the Nordic countries many times. I was amazed of what I have seen. First, I am a sports person. I practiced sports since I was a child and continue to practice. I love sports, my daughter is a tennis player, I am going to the gym, I play squash and badminton regularly, I like to run, I also like to watch various sports like tennis and handball. I was even a sports journalist for a short period. As you can see, sport is part of my life. Still, I have never imagined that I will ever find a nation dedicated to sports. This is what I saw when I was in Finland. I had the same feeling in Sweden, Norway or Denmark. These nations are healthy nations firstly because they are doing massive sports. Everyone is running, everyone is biking; the pools are full with people all ages, from children to seniors, I never imagined I would generalize in such a manner and be able to state that I found a true sports nation! This statement applies to the Nordics. Why are they doing so much sport? Not because they are necessarily great fans like me, many of them do not watch sports and do not care about professional sport. They are simply doing sports because they do care about performance. They care about their body and they know that, apart from their normal life, they need to do sport to have a good performance. During my database courses in the Nordics, I was at lunch with my students. They were eating only salads; they were not eating bread almost at all, the entire nation was on a diet. The reason is the same, performance.

You just saw how a nation is capable to analyze and to take into account the performance in their daily activities. Why can't we do the same in our projects? Going back and speaking about the database, the first measure we should take is to write correctly in the database. That means effectively to stop using an inappropriate style of development in our databases, that means to stop killing the performance with cursors over cursors and scalar functions over scalar functions called everywhere, and structures and records and arrays instead of using the classic and native SQL facilities. Looking at the performance of our databases it means, in the first place, to try to do not consider SQL as an additional skill that can be achieved easily by anyone and to allow more respect for this language. Looking at performance firstly means explaining to the variety of application developers working in our databases that they should try to think differently due to the simple fact that they are in a distinct environment, the database, where the main concept and concern for them is to handle everything in data sets. To explain to the application developers that there are no such things as good or superior languages and bad or inferior languages, that there are no absolute criteria in software development. In software development, we are clearly adopting a utilitarian and practical thinking.

I noticed very often that, if we write correctly from the beginning, we start already with the premises of a good performance. Which is why I insisted so much on the necessity to start explaining the concepts of data set and the set-oriented style of development during the college years? The young generation of programmers should be aware from the beginning, before starting to write code that they need to think differently in the database. No one told them that explicitly, or very rarely, and they are tempted to write in the same manner in the database. It is such a simple solution but it may have very good consequences! Let's imagine for a moment that all the data-oriented software applications will be written correctly in the databases. I can guarantee that it will be a better world, from the performance point of view. The remaining portion will be for true SQL specialists. They will need to refine, to take the SQL statements and try to improve them; they will replace a poor syntax with a better one, until they will get the best SQL. What will remain? The DBA's and the specialized database specialists will enter the scene and they will start to add valuable features for performance, if this one will still suffer. The main reason for a poor performance will be eliminated. Of course, this will never happen; I am not so naive to believe that, it was just a game of imagination.

The winner in the game of the two styles of development will always be the performance. Choosing the proper style of development means to choose a good performance or not. This book is not to be considered a book about performance. A book about performance is the one where various techniques are described like indexes, partitioning, materialized views, explaining execution plans, and gathering statistics. That is a book about performance, but this one here is a book about how to improve performance, or about how to have a good performance in a fair development environment. All the principles described in this second book are to be applied to a database where the style of development is the holistic, set-oriented style of development. If the database is written in the atomic style of development those principles are useless. Consequently, from this perspective, my book is a book about performance, even the first part of it. If you don’t write holistically in the database, and mostly in some kinds of databases like the ones that implement specific software applications, you choose the worst performance you may have. It is up to you to decide, as a programmer, as a manager of the project, you can decide if you really care about the performance of your software application and let the developers’ work, as they want to. The game is yours: you should properly learn the rules! The style of the player is always a critical component of the game. Apart from learning the rules, look at the style and choose wisely in concordance with your reason. I tried to offer you good reasons for a certain style, you can check your self the differences. The ball is in your hand!

A SPECIFIC DATA MIGRATION SOFTWARE APPLICATION CAN BE WRITTEN IN PURE SQL MORE OFTEN THAN YOU EXPECT!

These are the last words before concluding this final chapter of my book. I hope to be able to write another book that will continue this one. The main topic and my concern is the use of a proper style of development in one layer or another of a software application. It is true that all my discussions and all my thoughts were related to the particular layer of the relational database. I believe we can imagine other layers and sections in the field of software development. The way we write our code and our style of development should be a main concern to all of us, programmers. We should consider this matter and we should try to analyze ourselves and wonder if we are using a proper style of development in the places where we effectively do our work. Even if the concept of style of development is not clearly defined and it has its degree of subjectivity, as I mentioned before, it is very important. I believe everyone will agree that our software applications and databases are influenced by our style of development. This will be reflected especially in the section of performance because, after all and in the end, we are generally able to build the software application but at what costs? Moreover, when I say at what costs I am referring to the cost of development and to the cost of the performance. All of these are influenced by our style of development.

I believe I had the chance to say my word in this matter. Some people will not be quite happy and satisfied with my ideas. It is their right to think so. I respect the work of an application developer. I don’t have such a strong perspective of the user interface and application development and I never had any intention to evaluate in any way their activity in their classic fields and sections. On the other hand, I have a very good perspective of the relational database; I have a very good understanding of the SQL language and I have been very often in the position to clean the code written by application developers in relational databases. That is why I strongly consider that the application developers need to make an effort and understand the concept of data set, to follow more the set-oriented development inside the database and to be aware that they need to write their code differently inside a relational database. With this, I finish this topic and I hope at least some application developers will understand and act consequently.

Apart from the topic of the style of development, I want to conclude referring to my intentions for my next book, which I hope I will have the energy and strength to complete. As I mentioned several times during the pages of this book, I had the chance to specialize in specific software systems written in pure SQL. I had been involved in replication systems and data migrations systems written in pure SQL and I strongly believe we can do miracles with the simple SQL language.

More and more enterprises have a variety of systems and the request for these types of specific systems will increase. There are dedicated tools for that and I don’t have the knowledge to judge these tools. What I can say again is the fact that you can always consider SQL an alternative.

My plan is to describe a data migration system written in pure SQL, considering that I had the privilege to design and build one from the scratch. I want to explain the principles, I want to illustrate the migration system with a variety of examples and show exactly how this can be done. I want to show that using a simple language like SQL may allow you to have a consistent product and to handle many customers and even in various database systems. I want to show you that the maintenance can be relatively easy; I want to show you how to debug, I want to show you how you can check the differences if the migration interface will be incremental, all these with no expensive tool just with pure SQL. I will show you that everything will flow and that the quantity of procedural code will be somewhere to five percent of the code.

The main advantages of this system are performance, portability, and readability of the code, naturalness, and simplicity. Of course, some businesses may be in such a way that simple SQL cannot achieve the goal and something else may be required, like a tool or something else. However, very often, more often than you expect, the simple SQL language will be enough.

My plan is to describe a methodology of a replication system written in pure SQL. This methodology may be useful for the software companies and for various customers that may want to build such a system and they may decide to use a simple solution with the trivial and simple SQL language. Of course, there are variations between the vendors like Oracle or SQL Server or PostgreSQL. However, if you write SQL oriented you don’t care so much about the differences.

With these final considerations, I finish this book and I hope you’ll enjoy it. I would be the happiest person in the world if at least one application developer will learn something from my book or if a student will start his first project knowing that the database is something different and he should not blindly apply the same principles learned in the University. In addition, if the student will be aware about the differences between the holistic and the atomic style of development from the University, that would be another reason of great satisfaction.