Analytic SQL in SQL Server 2014/2016

Series Editor
Jean-Charles Pomerol

Analytic SQL in SQL Server 2014/2016

Riadh Ghlala

ISTE

WILEY

First published 2019 in Great Britain and the United States by ISTE Ltd and John Wiley & Sons, Inc.

ISTE Ltd
27-37 St George's Road
London SW19 4EU
UK

www.iste.co.uk

John Wiley & Sons, Inc.
111 River Street
Hoboken, NJ 07030
USA

www.wiley.com

Library of Congress Control Number: 2019939237

British Library Cataloguing-in-Publication Data
A CIP record for this book is available from the British Library
ISBN 978-1-78630-412-4

Contents

Introduction

I.1. Motivation

Nowadays, data analysis is a skill that is in high demand in order to develop indicators and metrics to monitor our information systems. In order to pursue this mission, several methods, techniques and tools have been developed to cover this need in various aspects.

Database Management Systems (DBMS), along with their querying language, Structured Query Language (SQL), are considered a very important path in data analysis. Based on analytic queries directly executed on the server side, a wide range of indicators can be developed without going through intermediate application layers.

Since its invention in the 1980s, SQL has demonstrated its dominance in data management. According to a study conducted by Indeed[1], SQL has the highest ranking among skills sought by IT talent researchers.

Figure I.1 shows the position of SQL in relation to other computer languages.

1 *Indeed* is a job search engine: https://www.indeed.com/.

The spread of SQL is explained by its characteristics of simplicity, high performance and diversity. Indeed, this language, which was previously designed for the definition and manipulation of data, has now become essential for analyzing this data.

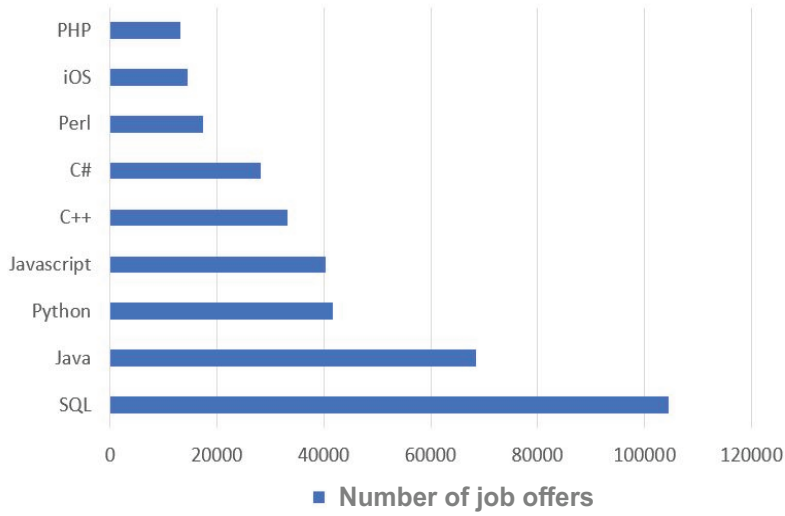


Figure I.1. *Number of Indeed job offers by programming language (January 2019)*

I.2. Outline of the work

This book, composed of four chapters, studies SQL as a tool for data management and analysis.

Although the concepts of this language are standardized, this book presents them with the Microsoft SQL Server Database Management System (DBMS) and focuses primarily on the analytical aspect of this language.

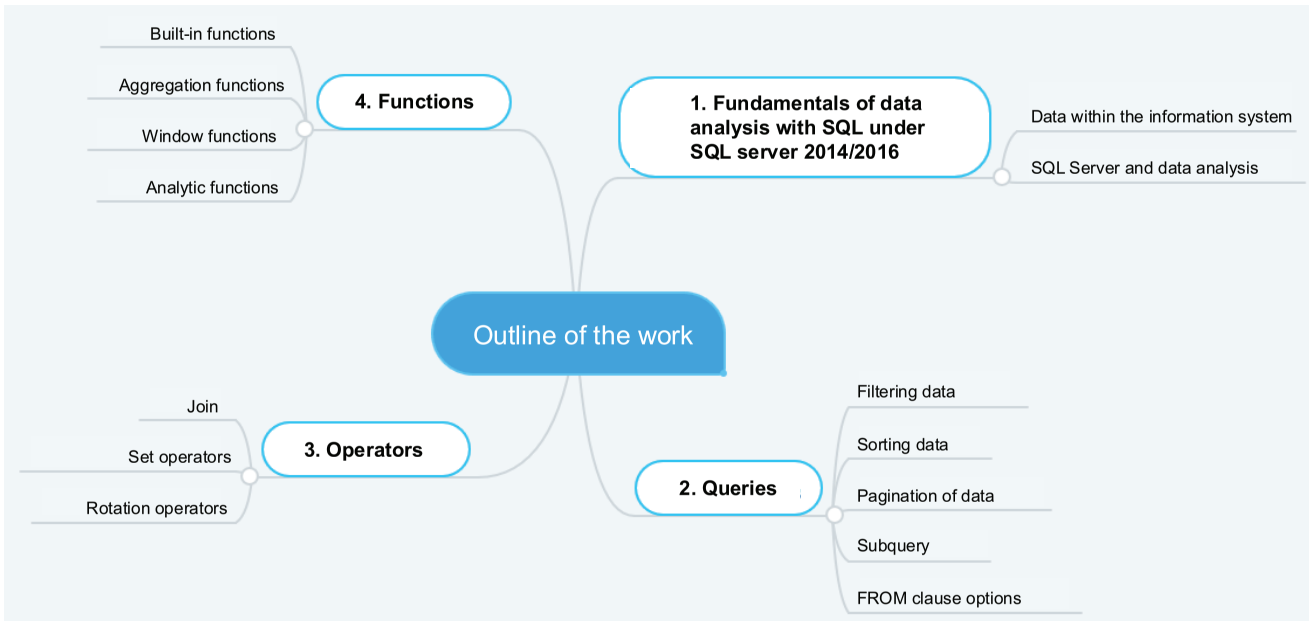


Figure I.2. *Outline of the work*

I.3. Mind map of the introduction

This introduction will be devoted to the presentation of SQL as a querying language for transactional and/or analytical data processing. Figure I.3 presents a mind map of the different concepts that will be covered in this introduction.

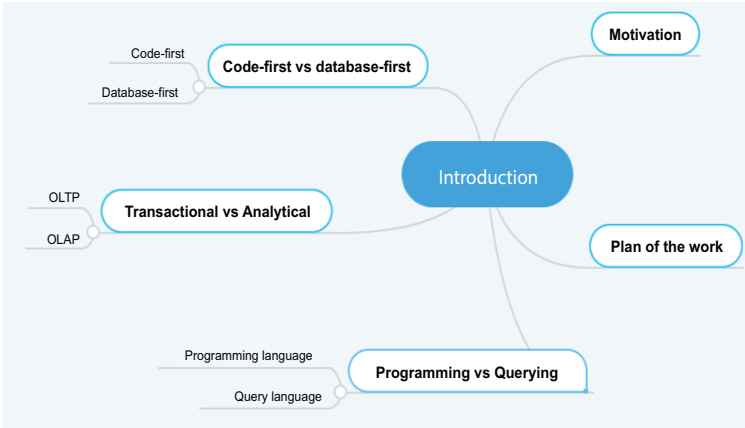


Figure I.3. *Mind map of the introduction*

I.4. Programming language versus querying language

The world of information technology contains a variety of languages, each dealing with a particular area of our information system. Among these languages, we find that the SQL language is designated as a querying language as opposed to the majority of other programming languages such as C#, Java or Python.

I.4.1. *Programming language*

Programming languages, also known as procedural languages (we also consider object-oriented languages which

are procedural languages with object-oriented mechanisms), are languages that allow developers to express their needs when automating particular tasks in the information system and also to specify to the machine (the computer) the method to apply to execution. Figure I.4 shows the process followed for programming with a procedural language.

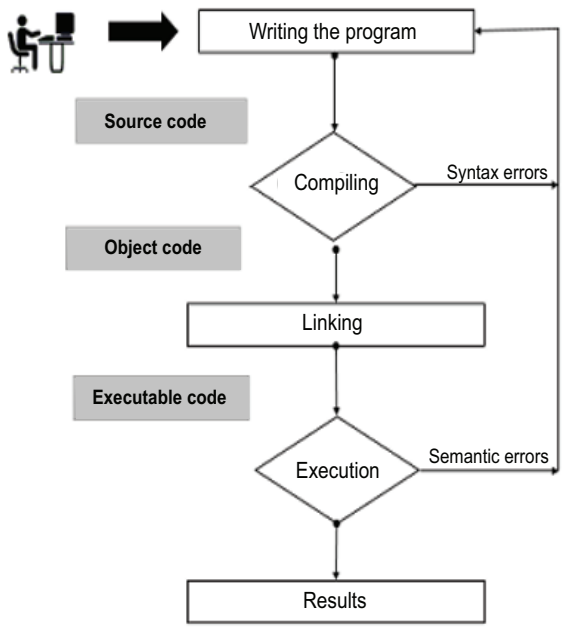


Figure I.4. *Steps of programming with a procedural language*

I.4.2. *Querying language*

The strength of querying languages, and in particular SQL, is that they ask users to focus on expressing their needs and to delegate the development of execution plans reflecting methods of execution to the tool used, which is the DBMS. Figure I.5 shows the process followed for requesting data using SQL.

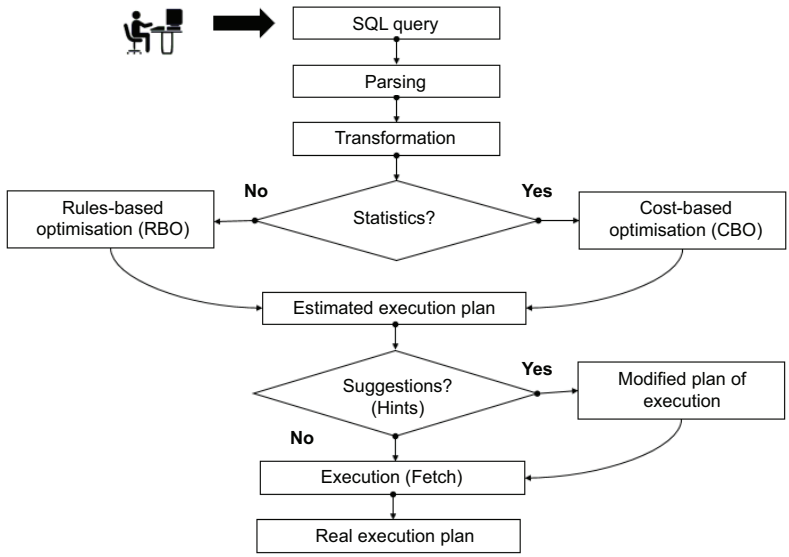


Figure I.5. *Data query with the SQL language*

I.5. Transactional processing versus analytical processing

Data represents the static axis of any information system. The processing of this data, which is done using the SQL language, is divided into two categories: definition and manipulation. In addition to the definition of data usually performed on the database server side, the manipulation of this data is entrusted to the business logic layer to execute transactions or perform queries.

I.5.1. *Transactional processing*

Transactional processing, also known as OnLine Transactional Processing (OLTP), consists of acting upon the data to change its status by addition, modification or deletion. This type of processing must be subject to very strict controls to ensure proper functioning in a multi-user and distributed context.

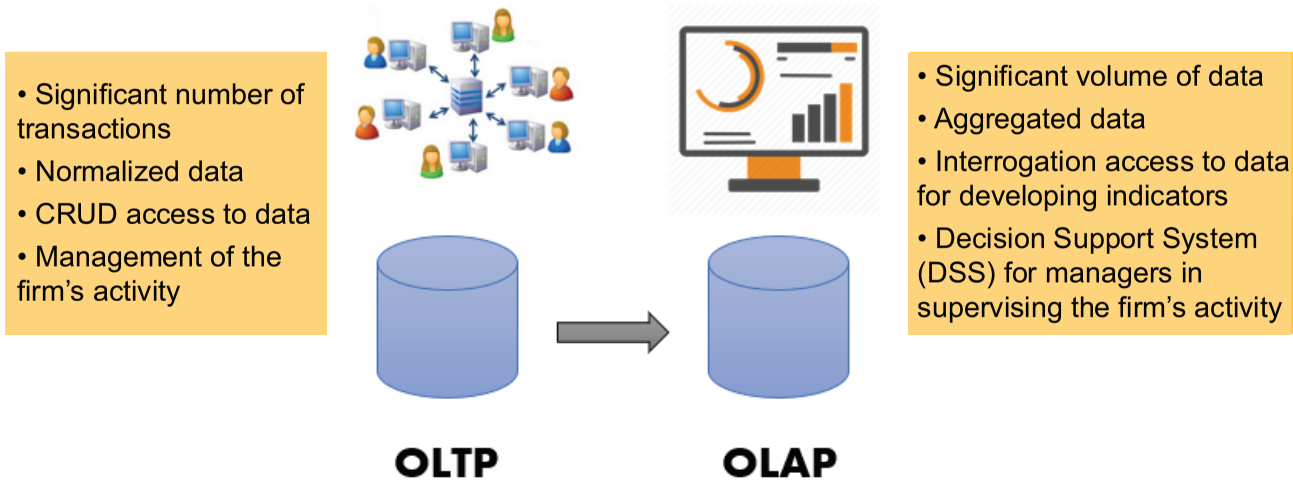


Figure I.6. *OLTP versus OLAP*

I.5.2. *Analytical processing*

Analytical processing, also known as OnLine Analytical Processing (OLAP), consists of interrogating the data in order to develop synthetic information that can be used as indicators for the decision-maker. Figure I.6 compares the two types of processing, OLTP and OLAP.

I.6. Code-first versus database-first

The IT solutions that manage our information systems present numerous variants of OLAP. Indeed, this analytical processing can be performed with SQL queries directly executed on the database server or by integrating these queries into the application layer programs.

I.6.1. *Code-first*

The term “code-first” refers to an approach that favors the use of the business logic layer for the development of a new need in the information system. In the case of analytical processing, this approach consists of developing the indicators requested by managers through programming or through graphical interfaces to generate the necessary programs. This approach has several disadvantages, such as the complexity of the programs to be developed, the security of data exchanges between these programs and the database, and the performance dilemma, especially when handling large amounts of information.

I.6.2. *Database-first*

The term “database-first” refers to an approach that promotes, for the development of a new need in the information system, the implementation of this need directly in the database via a functionality provided by the DBMS.

In the case of analytical processing, this approach consists of developing the indicators requested by decision-makers through appropriate SQL queries designated by the analytic functions of SQL or simply analytic SQL. This approach is important in information systems because of the simplicity, performance and security of the data in these types of requests. The use of this approach requires the mastery of these analytic functionalities of the SQL language.

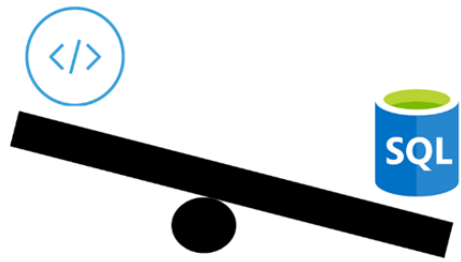


Figure I.7. *Code-first versus DB-first*

I.7. Conclusion

Data analytics is a crucial need in information systems. This need can be met by various IT solutions for the development of the different indicators required by managers to supervise their firm's activities in a highly competitive economic environment. SQL, as a querying language, is constantly being enriched with new features to improve its analytical capacity and therefore to master this aspect of analytical processing (OLAP), especially in database-first mode.

1

Data Analysis Fundamentals with the SQL Language

1.1. Data at the heart of the information system

1.1.1. *Introduction*

The development of IT solutions becomes more complicated from one day to the next due to the ever-increasing complexity of information systems and the need to align business needs with these IT solutions. In every generation of information system, data remains the cornerstone of these IT solutions. This importance is due to the static aspect of data in real-world modeling and the ability to provide any processing requested by application developers to meet the needs of the business logic layer.

Figure 1.1 shows the position of data in the urbanization of information systems according to the Enterprise Architecture (EA) vision[1]. This model puts two of the components on equal footing: *data* and *applications*.

Indeed, the role of data in meeting the needs of the business logic layer is essential, whether through the

1 Enterprise Architecture is a systemic vision of the enterprise in the form of four components (business, data, applications and technologies).

Analytic SQL in SQL Server 2014/2016, First Edition. Riadh Ghlala.

applications component or by direct access to the *data* component by means of a query language such as SQL.

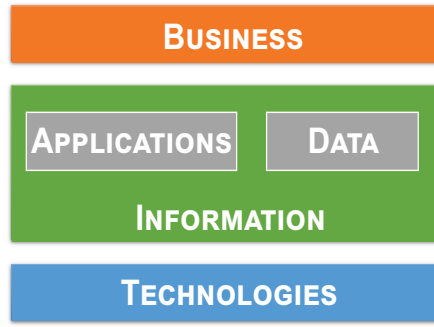


Figure 1.1. *Enterprise architecture*

1.1.2. *Mind map of the first section*

This first section will be devoted to introducing the basics of databases, the SQL language and Microsoft SQL Server DBMS.

Figure 1.2 presents a mind map of the different concepts that will be covered in this section.

1.1.3. *Concept of a database*

A database (referred to as DB) is a collection of homogeneous data relating to a real-world domain (information system), stored in a structured way and with as little redundancy as possible to facilitate and accelerate access to these data later on.

These data must be accessible by different users, possibly simultaneously. Thus, the notion of database is generally coupled with that of computer networks in order to be able to share this information.

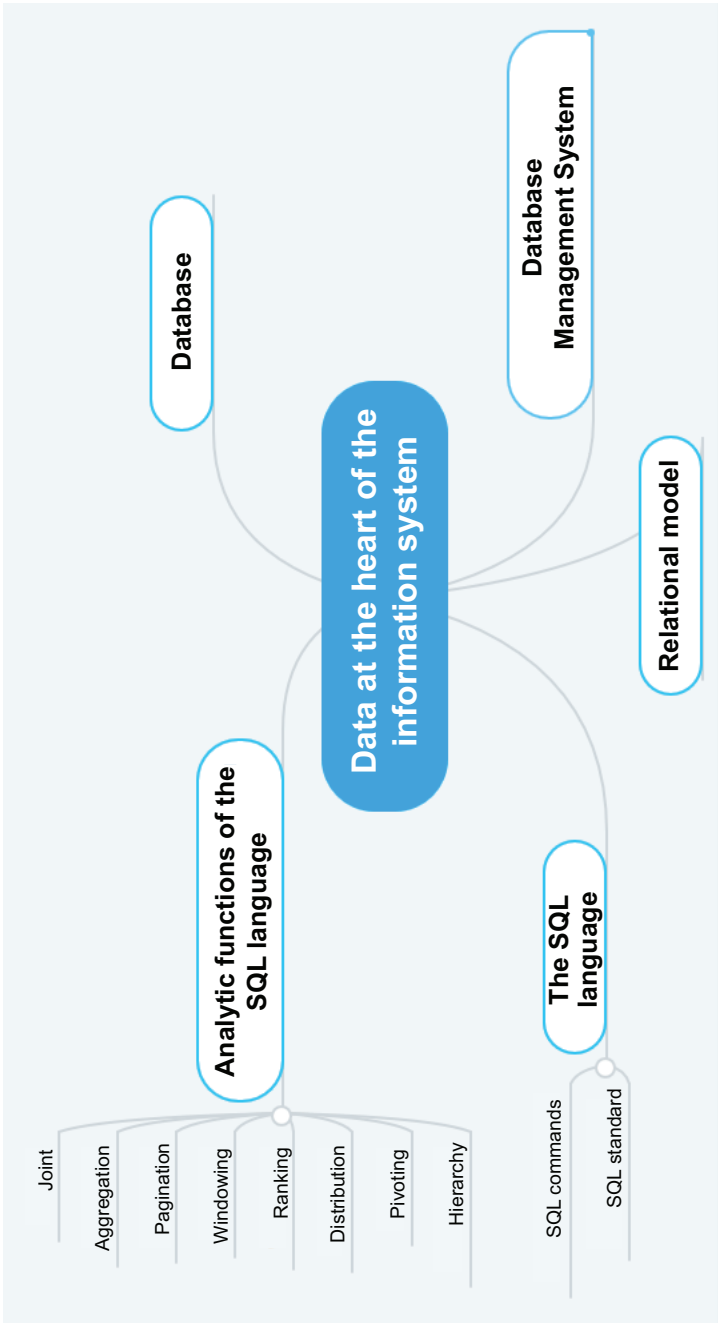


Figure 1.2. *Mind map of the first section*

The engineering process of a database goes through three essential steps: design, generation of the database schema in an appropriate formalization (in our case, the relational model) and finally implementation of this database.

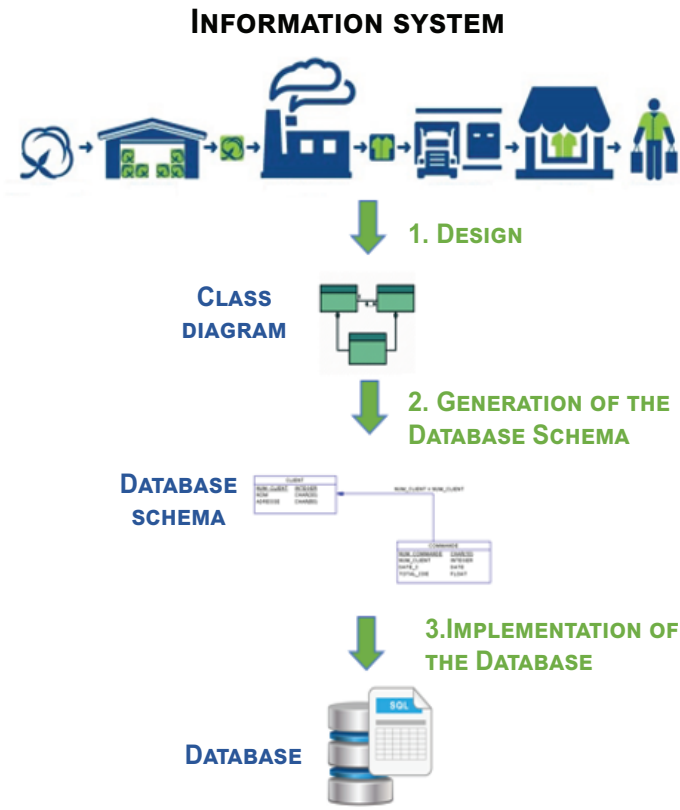


Figure 1.3. *Database engineering process*

1.1.4. *Database Management System (DBMS)*

Database management consists of the implementation, operation and administration of the database. These tasks are performed by software using a set of commands written in a query language called SQL.

The range of DBMSs is very diverse. The choice of a DBMS is made according to several criteria: functionalities provided, cost, performance, security, etc.

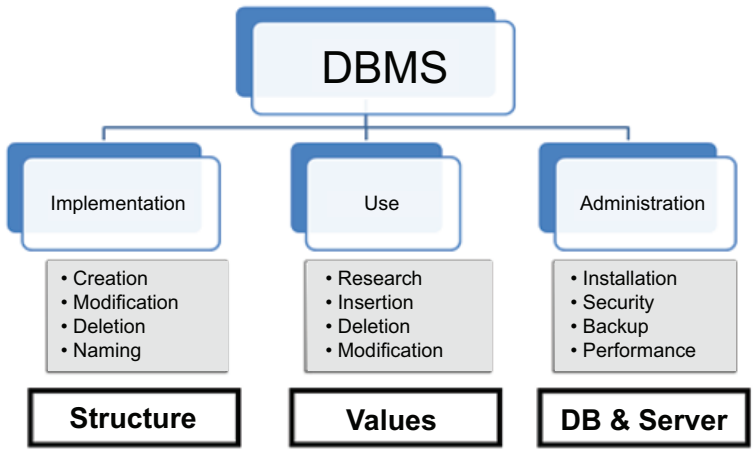


Figure 1.4. *Features of a DBMS*

Figure 1.5. *The main relational DBMSs*

1.1.5. *The relational model*

In 1970, Edgar F. Codd, a researcher at IBM[2], proposed in a mathematical thesis the representation of data, links and even the expected results of a search in a database in table form. However, IBM – which was then working on another type of database – did not begin to take an interest in it until 1978, when the concept interested Lawrence Ellison, the founder of a start-up that became Oracle Corporation.

This proposal is the basis of the relational data model, a model used by almost all DBMSs. These DBMSs are qualified as Relational DBMSs (RDBMSs). In the past, this label was obvious since the majority of DBMSs were relational. Today, with a very diversified landscape including SQL, NewSQL and NoSQL, it is essential to specify the qualification of the DBMS.

This book discusses the analytic functions of the SQL language in relational DBMSs. The illustrative examples are based on the use of Microsoft SQL Server 2014/2016 DBMS.

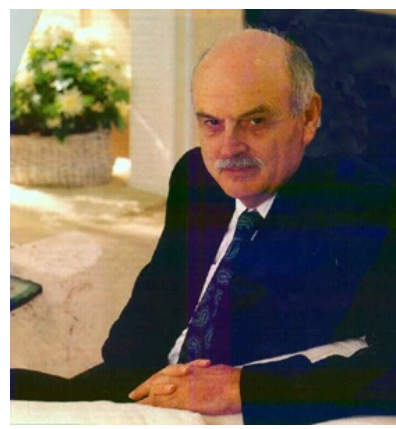


Figure 1.6. *Edgar Frank Codd (23 August 1923–18 April 2003, United Kingdom). Turing Prize in 1981*

2 International Business Machines (IBM) is an American multinational company specialized in the field of computer equipment.

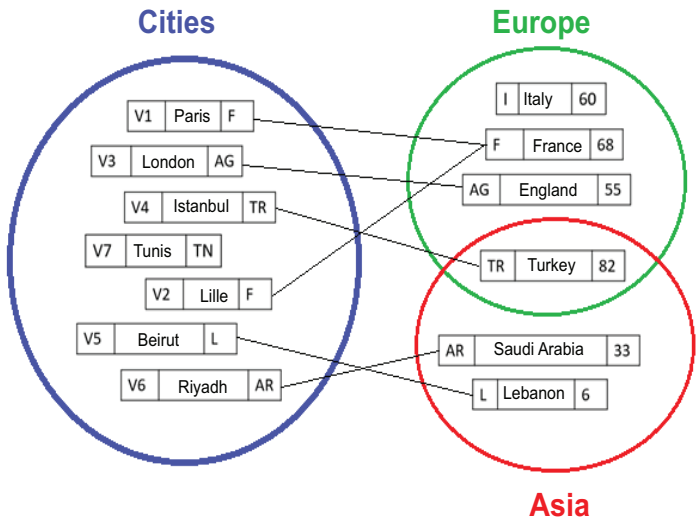


Figure 1.7. *The relational model: a model based on set theory*

1.1.6. *The SQL language*

SQL (Structured Query Language) is a computer language used to communicate with a database. Unlike programming languages where the user is called upon to specify the execution algorithm, SQL makes it possible to provide results by simply formulating our needs in the form of queries close to natural languages.

1.1.6.1. *SQL commands*

The SQL language consists of a set of commands that can be classified into four sub-languages according to their roles in data management within the database.

Figure 1.8 shows the SQL language commands and their classifications which are:

– Data Definition Language (DDL) containing commands for creating, modifying and deleting objects managed in the database, the most important of which are the tables that represent the data storage structures;

– Data Manipulation Language (DML) containing commands to search, insert, modify, delete and merge data into tables;

– Data Control Language (DCL) containing the commands for managing access control (allocation, removal and prohibition) to data;

– Transaction Control Language (TCL) containing transaction control commands.

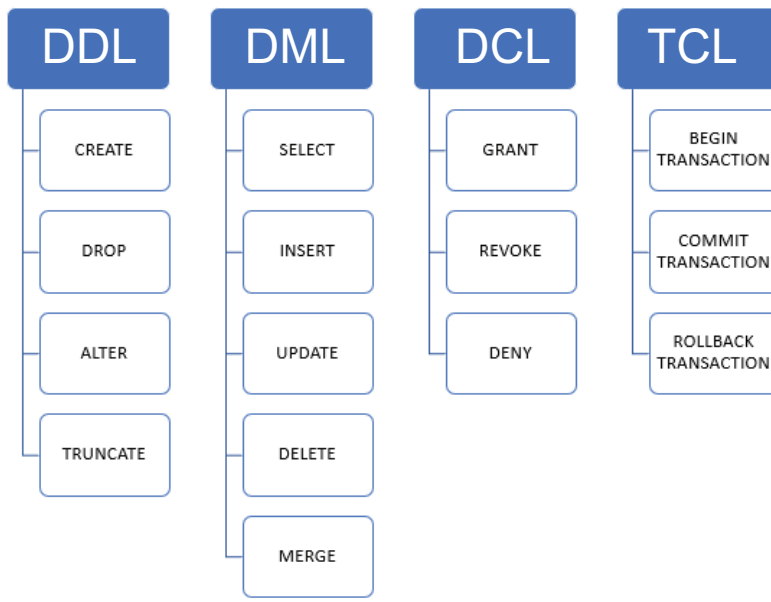


Figure 1.8. *SQL language commands*

1.1.6.2. *SQL standards*

Since its invention in the early 1980s, this language has undergone several changes and has become a standard on the market in terms of the query language used by RDBMSs. These different periods through which this language has gone are marked by standards developed by the ANSI[3] and ISO[4].

Table 1.1 represents the most important features of the different standards of the SQL language.

Standard	Features and functions
SQL1 (86–89)	– Basic operators of relational algebra
	– Simple data types (integers, real, fixed-size strings)
	– Restricted Assembly Operations (UNION)
	– Referential integrity (foreign key)
SQL2 (92)	– Richer data types (intervals, dates, variable-size strings)
	– Different types of joints (JOIN): natural joint, external joint
	– Set operations: difference (EXCEPT), intersection (INTERSECT)
SQL3 (99)	– Extends the relational model with object-oriented features
	– Supports complex, user-defined data types
	– Integrates trigger management
SQL 2003	– Auto-incrementing of keys
	– Calculated column

3 The American National Standards Institute (ANSI) is an American standards organization.
4 The International Organization for Standardization (ISO) is an international standards organization.

	– XML support
	– Window functions
	– Sequences
	– Identity columns
SQL 2008	– Improvement of the MERGE and DIAGNOSTIC instructions
	– The TRUNCATE TABLE instruction
	– Comma separation of WHEN clauses in the CASE instructions
	– INSTEAD OF triggers
	– Support for some XQuery features
SQL 2011	– New features to better manage pagination of results
SQL 2016	– JSON

Table 1.1. *SQL standards*

1.1.7. *Analytic functions of SQL language*

The end of the 20th Century was marked by a computer revolution, specifically in the field of data management with the popularization of relational databases. This technology has contributed greatly to supporting production information systems based essentially on transactional activities. These systems are referred to as OnLine Transactional Processing (OLTP).

Nowadays, we are witnessing new demands from company managers. Indeed, the latter are constantly looking for dashboards that reflect the status of their activities in relation to objectives already set. These dashboards are based on Key Performance Indicators (KPIs).

Business Intelligence (BI) aims to meet managers' requirements in terms of dashboards. It includes techniques, methods and tools for conducting analytical IT solutions. These solutions cover a company's business intelligence system known as OnLine Analytical Processing (OLAP). SQL, as an influential player in this ecosystem, has improved to ensure its position as a leader in the field of transactional data management as well as analytics.

In addition to the mechanisms acquired in SQL that can be used for data analysis such as filters, sorting, joining and subqueries, SQL seeks to reinforce this need for analysis with a series of practical functions to respond to complex analysis requests with all the simplicity, elegance and performance that characterize this language. These functions can be classified into subcategories, namely:

1.1.7.1. *Aggregation functions*

The aggregation functions (MIN, MAX, SUM, COUNT and AVG) represent the backbone of any analytical processing. Several options are added to this category of functions to strengthen their analytical capabilities.

1.1.7.2. *Pagination functions*

Until recently, the pagination of query results followed approaches specific to each DBMS. The SQL 2011 standard unifies this important functionality in data analysis.

1.1.7.3. *Windowing functions*

Windowing functions represent a revolution in the SQL language. They allow the same data stream extracted from the database to be used several times to develop various indicators over these data. The windowing mechanism is ensured by the OVER clause.

1.1.7.4. *Ranking functions*

The classification functions (RANK, DENSE_RANK, NTILE and ROW_NUMBER) represent an application of the windowing functions to develop a specific classification of the results.

1.1.7.5. *Distribution and offset functions*

The distribution and offset functions (LEAD, LAG, FIRST, LAST) represent another application of the windowing functions to develop correlation indicators between the different lines of the result.

1.1.7.6. *Functions for processing null values*

The functions for processing null values (ISNULL and COALESCE) are essential to solve the problem of the data schema constraint in a relational database. Ignoring this can distort the results.

1.1.7.7. *Pivot operators*

Analytical processing proposes new operators for multidimensional data structures called Cubes. Among these operators, we find the PIVOT operator which has been introduced even in the relational context to simulate the rotation operator on two-dimensional tables, in order to develop indicators according to different visions of the data.

1.1.7.8. *Hierarchical functions*

Data modeling can lead to reflexive associations. Data analysis, in this case, must use hierarchical queries to extract relationships that may exist between different occurrences of the same entity.

1.1.8. *Conclusion*

Data are at the heart of any analytical processing. Indicators can be developed through programs that manipulate these data or through direct access through queries using analytic functions dedicated to this type of need.

1.2. SQL Server and data analysis

1.2.1. *Introduction*

Microsoft SQL Server is a relational DBMS. According to the db-engines.com[5] website, it is ranked the third DBMS among a list of 343 DBMS of different categories as of January 2019. The criteria used in this ranking include a list of properties of these DBMSs as well as the functionalities they offer.

Figure 1.9 shows the top 10 DBMSs according to this ranking.

1.2.2. *Mind map of the second section*

This second section will focus on presenting the SQL Server DBMS as a full data management environment.

Figure 1.10 presents the mind map of the different concepts that will be discussed in this section.

5 *db-engines* is a ranking site for database management systems https://db-engines.com/en/ranking.

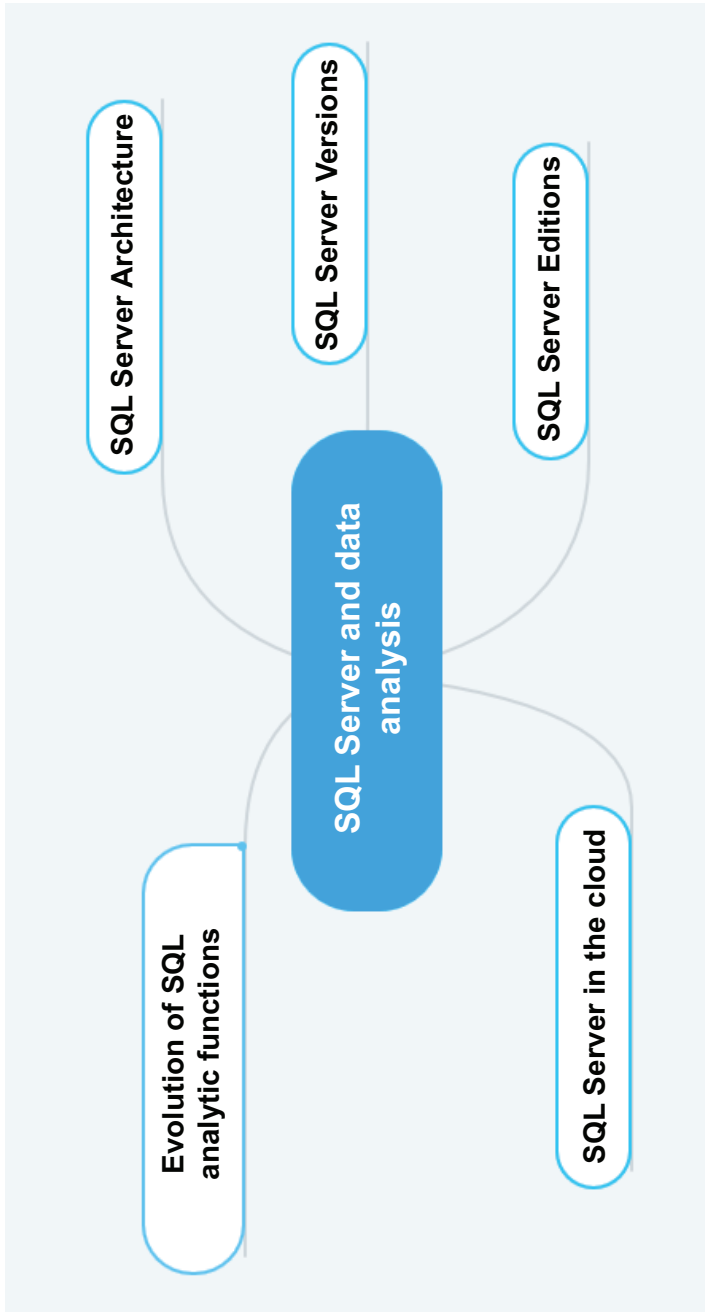


Figure 1.9. *Mind map of the second section*

343 systems in ranking, January 2019

Rank Jan 2019	Rank Dec 2018	Rank Jan 2018	DBMS	Database Model	Score Jan 2019	Score Dec 2018	Score Jan 2018
1.	1.	1.	Oracle	Relational DBMS	1268.84	-14.39	-73.11
2.	2.	2.	MySQL	Relational DBMS	1154.27	-6.98	-145.44
3.	3.	3.	Microsoft SQL Server	Relational DBMS	1040.26	-0.08	-107.81
4.	4.	4.	PostgreSQL	Relational DBMS	466.11	+5.48	+79.93
5.	5.	5.	MongoDB	Document store	387.18	+8.57	+56.24
6.	6.	6.	IBM Db2	Relational DBMS	179.85	-0.90	-10.43
7.	7.	↑ 9.	Redis	Key-value store	149.01	+2.19	+25.88
8.	8.	↑ 10.	Elasticsearch	Search engine	143.44	-1.26	+20.89
9.	9.	↓ 7.	Microsoft Access	Relational DBMS	141.62	+2.10	+14.92
10.	10.	↑ 11.	SQLite	Relational DBMS	126.80	+3.78	+12.54

Figure 1.10. *DBMS ranking (top 10)*

1.2.3. *SQL Server architecture*

Microsoft SQL Server is more than a relational DBMS that is limited to defining and manipulating data. Rather, it is a very rich environment that, in addition to the main task already mentioned, covers the other operations that form the life cycle of this data within the company such as cleaning, consolidation, performance and data analysis.

To ensure these different missions, SQL Server qualifies as a DBMS with a pluggable architecture. It consists of a set of services dedicated to each of the different areas of data management.

Figure 1.11 shows a summary diagram of these different services which can be integrated into the environment in a customized way in the SQL Server DBMS.

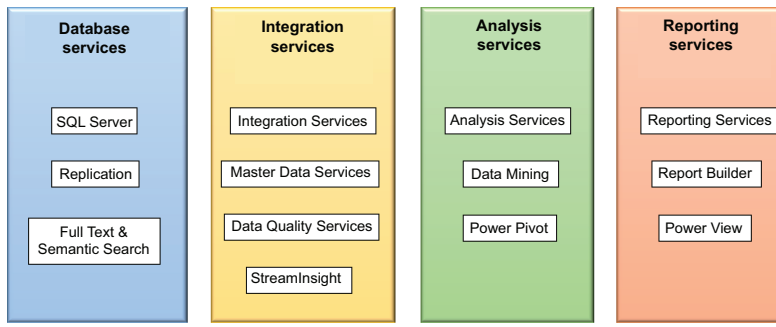


Figure 1.11. *SQL Server services*

The most used services in SQL Server are:

– Database engines on-premise and in the Microsoft Azure cloud (SQL Server Engine);

– the Full-Text Search service;

– SQL Server Integration Services (SSIS);

– Master Data Services (MDS);

– StreamInsight Service;

– Data Quality Services (DQS);

– SQL Server Analysis Services (SSAS);

– SQL Server Reporting Services (SSRS).

1.2.4. *SQL Server versions*

Since its invention at the end of the 1980s, SQL Server has undergone continuous development to support new functionalities and/or different operating systems.

Table 1.2 shows the different versions with their years of appearance, code names and supported operating systems.

Designation	Year	Version	Code	Operating system
SQL Server 2019	2019	–	Aris	Windows and Linux
SQL Server 2017	2017	14.0	Helsinki	Windows and Linux
SQL Server 2016	2016	13.0	–	Windows
SQL Server 2014	2014	12.0	Hekaton	Windows
SQL Server 2012	2012	11.0	Denali	Windows
SQL Server 2008 R2	2010	10.50	Kilimanjaro	Windows
SQL Azure DB	2010	10.25	DB Cloud	Azure
SQL Server 2008	2008	10.0	Katmai	Windows
SQL Server 2005 Analysis Services	2005	–	Picasso	Windows
SQL Server 2005	2005	9.0	Yukon Territory	Windows
SQL Server 2003	2003	–	Rosetta	Windows

Reporting Services				
SQL Server 2003 (64 bit)	2003	8.0	Liberty	Windows
SQL Server 2000 (32 bit)	2000	8.0	Shiloh	Windows
SQL Server 7 OLAP Services	1999	–	Plato	Windows
SQL Server 7	1998	7.0	Sphinx	Windows
SQL Server 6.5	1996	6.5	Hydra	Windows
SQL Server 6	1995	6.0	SQL95	Windows
SQL Server 4.2.1	1994	4.21	–	Windows
SQL Server 4.2	1992	4.2	SQLNT	OS/2
SQL Server 1.1 (16 bit)	1991	1.1	Pietro	OS/2
SQL Server 1.0 (16 bit)	1990	1.0	Filipi	OS/2

Table 1.2. *SQL Server versions*

1.2.5. *SQL Server editions*

SQL Server is proprietary software. It is marketed in different editions based on the functionalities required by the company.

Figure 1.12 shows the different editions of the 2014 version.

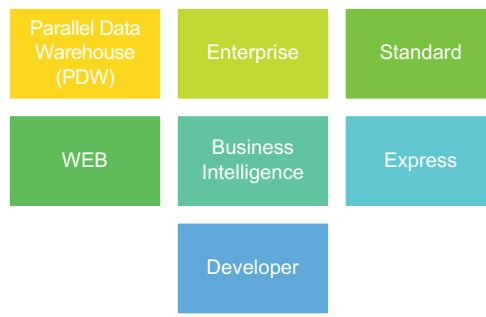


Figure 1.12. *SQL Server editions*

1.2.6. *SQL Server in the cloud*

The cloud is a new opportunity for companies to manage their information systems. All or part of this information system and its applications must be embedded within it. The latest versions of SQL Server have been designed to reside in the cloud to provide the company with a scalable and highly available platform with pay-per-use. Azure SQL Database is a database provided as SaaS (Software as a Service) with the Microsoft Azure platform.

1.2.7. *Evolution of SQL analytic functions in the SQL Server DBMS*

Microsoft® SQL Server 2014 provides a scalable business intelligence platform optimized for data integration, reporting and analysis. This book deals with one aspect of this platform, which is the SQL analytic functions of the Microsoft SQL Server SQL language referred to as Transact SQL or simply TSQL. Figure 1.13 shows the genesis of these functions across the different versions of MS SQL Server.

Although the examples in this book use an SQL Server 2014 environment (Software and Test Bases), the SQL Server 2012 version can also be used because it is already considered a reference version in terms of the SQL language as it implements almost all of the SQL 2011 standard.

1.2.8. *From analytic SQL to "In-Database Analytics"*

"In-Database Analytics" is a technology for analyzing data in the database by integrating analytical logic into the database itself. This eliminates the time and effort required to transform the data and move it between a database and a separate analysis application. SQL Server has joined this trend, as shown in Figures 1.14 and 1.15 respectively, by integrating the R server from version 2016 and the Python language from version 2017.

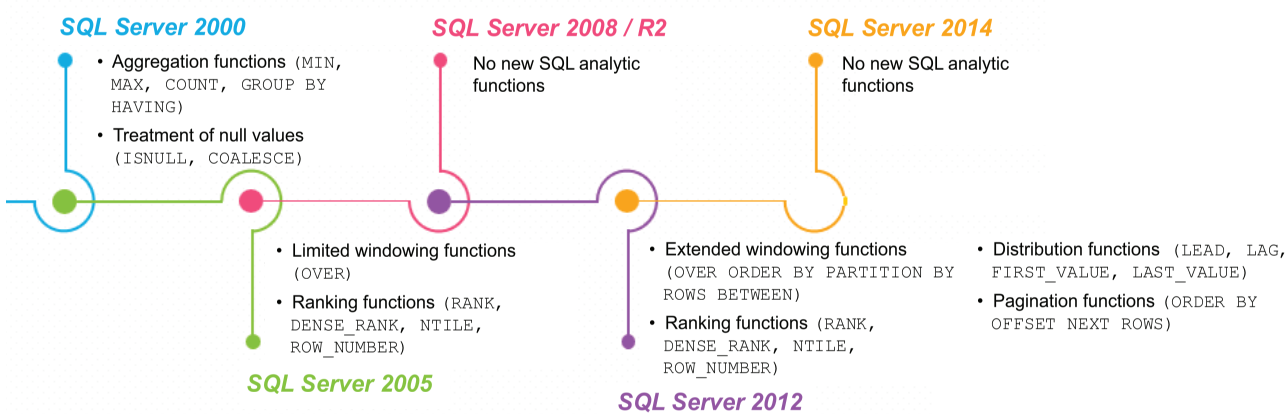


Figure 1.13. *The evolution of SQL analytic functions in Microsoft SQL Server*

Figure 1.14. *Integration of the R server into SQL Server version 2016*

Figure 1.15. *Integration of the Python language into SQL Server version 2017*

1.2.9. *The test environment*

To test the different requests in the following chapters, a test environment must be set up. This environment includes the SQL Server and the AdventureWorks database.

1.2.9.1. *Installing SQL Server 2014*

Microsoft SQL Server is proprietary software and requires a license for use. However, it is possible to install it for educational purposes with an evaluation version (one version for 180 days). The steps for installing the SQL Server DBMS are as follows:

1) Have a PC with a minimum configuration to install Microsoft SQL Server 2012 or later (2014 or 2016).

1) Processor: 2.0 GHz or higher

2) RAM main memory: 1 GB

2) Download Microsoft SQL Server software from the Microsoft website[6].

3) Install Microsoft SQL Server software:

1) Once the download is complete, run the setup.exe installation file (run as administrator).

2) After a few seconds, a window appears to check the required hardware and software configuration and start the installation.

3) In the Scheduling window, click "New stand-alone installation of SQL Server" or "Add functionality to an existing installation".

4) In the Product Key window, enter a product number or select the free edition.

5) In the License Conditions window, accept the license and click on Next.

6) In the global rules window, check the Microsoft update and click on Next.

7) In the Microsoft Update window, check the Microsoft update and click on Next.

8) In the product update window, now install SQL Server offline. It is not necessary to check the option to install the latest updates. Click on Next.

9) In the installation rules window, check all the rules that have been transmitted. The firewall may give a warning, which is not a problem. Click on Next.

6 https://www.microsoft.com/fr-fr/evalcenter/evaluate-sql-server-2014-sp2.

10) In the feature selection window, select the required features and click on Next.

11) In the Functional Rules window, click on Next to continue the installation.

12) In the instance configuration window, create a default instance and click on Next.

13) In the server configuration window, you can create a new user profile and assign a password to it if you want a separate account to be updated for the SQL Server service. You can choose the default option. Click on Next.

14) In the database engine configuration window, select Mixed SQL Server Authentication to provide a password for SQL Server authentication. Click on the ADD current user button to add the current user as administrator.

15) In the database engine configuration window, once the user has been added, click on Next.

16) In the configuration window of the database engine, you can change the data directory path.

17) In the database engine configuration window, check the size of the temporary database. You can change the size of the temporary database and the path.

18) In the configuration window of the database engine, you can activate the file flow. Click on Next.

19) In the Analysis Services Configuration window, select SSAS Server Mode, add the current user to the SSAS administrator, and then click on Next.

20) In the Analysis Services Configuration window, check the installation path of the analysis service. You can change the installation path and click on Next.

21) In the configuration window for report generation services, select Install and configure for native SQL Server mode.

22) In the Ready to Install window, select the functions to be installed and click on the Install button.

23) In the installation progress window, the installation will take some time; click on Next, once the installation is complete.

24) In the completion window, check that all features are successfully installed.

25) In the completion window, click on Close to complete the installation.

1.2.9.2. *The AdventureWorks2014 test database*

The illustrative examples require the existence of a database already implemented to use as a test database. The choice adopted in this book is to use one of the databases provided by Microsoft, which is the AdventureWorks2014 database.

AdventureWorks2014 is designed to introduce the features of SQL Server and can be downloaded from the Microsoft website. It can be integrated into the server according to several scenarios.

1.2.9.2.1. Restoring the database from a backup file

The integration of the AdventureWorks2014 database into the server is done by restoring a backup file (*AdventureWorks.bak*) related to this database. The steps in this scenario are as follows:

1) Download the file "*AdventureWorks2014.bak*"[7].

2) Open SQL Server Management Studio and connect to the target SQL Server instance.

7 https://github.com/Microsoft/sql-server-samples/releases/tag/adventureworks.

3) Right-click on the Databases node, then select Restore Database.

4) Select *Device* and click on the ellipses (…).

5) In the Select Backup Devices dialog box, click Add, access the database backup in the server file system and select the backup.

6) Click on OK.

7) If necessary, change the target location of the data and log files in the Files pane. Note that it is recommended to place data files and log files on different drives.

8) Click on OK. This will initiate the restoration of the database. Once this operation is complete, the AdventureWorks database will be installed on your SQL Server instance.

1.2.9.2.2. Creating the database by executing an SQL script

Integration of the AdventureWorks2014 database into the server is performed by executing an SQL script (*instawdb.sql*) containing the queries related to the generation of the database. The steps in this scenario are as follows:

1) Download "AdventureWorks-oltp-install-script.zip"[8] and extract the zip file to the "C:\Samples\AdventureWorks" folder.

2) Open SQL Server Management Studio and connect to the target SQL Server instance.

3) Open the "*instawdb.sql*" file from the location where the sample database is extracted.

8 https://github.com/Microsoft/sql-server-samples/releases/download/adventureworks/AdventureWorks-oltp-install-script.zip.

4) Now make sure you find the variable *SqlSamplesSourceDataPath* and its value which should be "*C:\Samples\AdventureWorks*". This is a very important task, otherwise your database will be created with empty tables.

5) Check the AdventureWorks database in the Object Explorer.

1.2.9.2.3. Attaching the database using the data file

Integration of the AdventureWorks2014 database into the server is performed by attaching the database from a data file *(AdventureWorks.mdf)* related to this database. This scenario has become obsolete with the new versions of SQL Server since this file is no longer available for download; however, if you have ever had this file (e.g. when moving the database from one server to another), the integration steps will be as follows:

1) Copy the *"AdventureWorks.mdf"* file into the "\MSSQL12.MSSQLSERVER\MSSQLSQL\DATA" folder of the "C:\Program Files\Microsoft SQL Server" folder.

2) Open SQL Server Management Studio and connect to the target SQL Server instance.

3) Right-click on the Databases node, then select Attach.

4) Select Add and browse to the MDF file you want to attach.

5) Select the file and click on OK.

6) The database you have selected should be displayed in the bottom window. If the file is listed as "not found", select the ellipses (…) next to the file name and update the path to the correct path.

7) If you only have the data file (.mdf), not the log file (.ldf), highlight the .ldf file in the lower window and select Delete. This will create a new log file.

8) Select OK to attach the file. Once the file is attached, the AdventureWorks database will be installed on your SQL Server instance.

1.2.9.2.4. Creating the database in the Microsoft Azure cloud

To be able to work with the AdventureWorks2014 database in the cloud, you must follow these steps:

1) Connect to your Azure portal[9].

2) Select "*Create a resource*" in the upper left corner of the navigation pane.

3) Select "*Databases*" and then select "*SQL Database*".

4) Fill in the requested information.

5) In the "*Select Source*" field, select *"Sample (AdventureWorksLT)"* to restore a backup of the last AdventureWorksLT backup.

6) Select "*Create*" to create your new SQL database, which is the restored copy of the AdventureWorksLT database.

1.2.9.3. *The script project*

In SQL Server Management Studio, we can create a project that allows us to logically group a set of queries related to a single project in the same place. The necessary steps for the creation of this project are:

1) Starting SQL Server Management Studio (SSMS);

2) File → New → Project;

9 https://portal.azure.com/.

3) Choose SQL Server Scripts;

4) Check the display of a window in SSMS related to the project. If the window does not appear, force it to be displayed by Display → Solution Explorer;

5) You can now create new connections and queries in the project.

1.3. Conclusion

Microsoft SQL Server is a full environment for data management over both transactional and analytical axes. SAL analytic functions consist of strengthening its analytical capacity to further support managers in supervising their activities via dashboards generated by direct access to data.

2

Queries

2.1. Data filtering

2.1.1. *Introduction*

The notion of filters is essential in SQL language. They consist of restricting the results to be displayed from a database. These filters are provided by different techniques incorporated into the query, from the simple WHERE clause to the use of the full-text search engine.

2.1.2. *Mind map of the first section*

This first section will be devoted to data filters.

Figure 2.1 presents the mind map of the different concepts that will be covered in this section.

2.1.3. *Types of filters*

Using filters in an SQL query is like doing a conditional search.

Analytic SQL in SQL Server 2014/2016, First Edition. Riadh Ghlala.

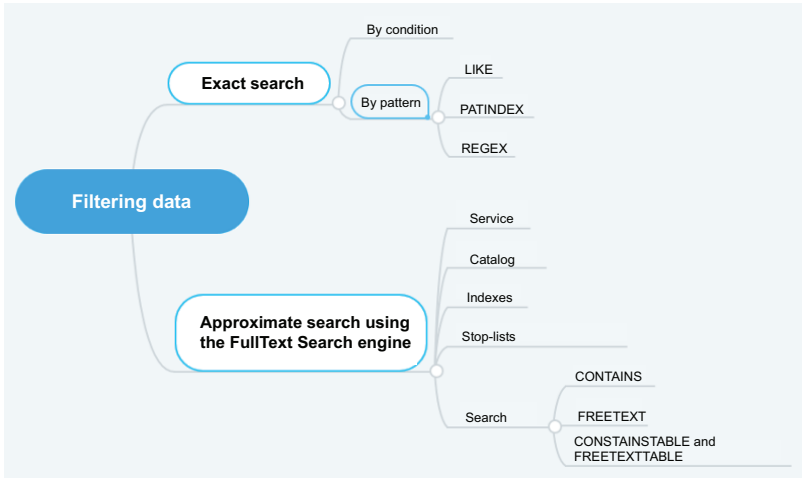


Figure 2.1. *Mind map of the first section*

These searches can be divided into different categories as follows:

– exact conditional search, which consists of returning a dataset fulfilling the condition(s) used in the filter. This search is invoked by the WHERE clause. The exact conditional search can itself be divided into two subcategories:

- exact conditional search with a filter by condition: this type of search consists of generating a particular dataset using one or more specific conditions;

- exact conditional search with a pattern filter: this type of search consists of generating a particular dataset using a pattern instead of one or more specific conditions. The search by pattern can itself be divided into two subcategories:

- exact conditional search with a filter by pattern using the LIKE operator;

- exact conditional search with a filter by pattern using the REGEXP function;

– approximate and semantic conditional search:

- Full-Text Search: complex queries on character data on SQL tables can be performed using full-text queries on SQL and Azure SQL databases. SQL Server Full-Text Search is an easy to use, very fast and extensible solution for indexing and searching over different types of document content (Word, Excel, PDF and HTML);

- Semantic Search: semantic search functionality has been added to version 2012 as an extension of the full-text search available in previous versions. While full-text searching allows you to query the words in a document, semantic searching seeks to improve the accuracy of the search by understanding the researcher's intent and the contextual meaning of the terms as they appear in the searchable data space in order to generate more relevant results. This feature is not covered in this book.

2.1.4. *Exact conditional search*

The exact conditional search makes it possible to specify with certainty the rows to be generated in the dataset. The user has a clear view of the data and thus has perfect command of how to write the filters needed for the search, either by condition or by pattern.

2.1.4.1. *Exact conditional search with one filter per condition*

This type of search consists of generating a dataset containing only the rows that verify the condition(s) entered in the WHERE clause.

IMPORTANT NOTE.– Generally, the attributes used in the conditions in the WHERE clause must be indexed to improve performance.

If you use several conditions, you must impose priority in parentheses to avoid ambiguity in the query.

```
USE AdventureWorks2014;
GO
-- Equal (=) Operator
SELECT P.Name, P.ListPrice
FROM Production.Product AS P
WHERE P.ProductLine = 'R' = 'R'
GO
-- Greater then (>) Operator
SELECT P.Name, P.ListPrice
FROM Production.Product AS P
WHERE P.StandardCost > 100
GO
-- BETWEEN Operator
SELECT *
FROM Production.Product AS P
WHERE P.ListPrice BETWEEN 100 AND 150
GO
-- IN Operator
SELECT *
FROM Production.Product AS P
WHERE P.Color IN ('Red','Yellow','Silver')
GO
-- AND Operator
SELECT *
FROM Production.Product AS P
WHERE P.ListPrice >= 100 AND P.ListPrice < = 150
GO
-- OR Operator
SELECT *
FROM Production.Product AS P
WHERE P.Color ='Red'.
OR P.Color ='Yellow'.
OR P.Color ='Silver'.
GO
-- OR and AND Operator
SELECT *
FROM Production.Product AS P
WHERE ListPrice > 100
AND (P.Color ='Yellow' OR P.Color ='Silver')
GO
```

Box 2.1. *Exact conditional search queries. For a color version of this box, see www.iste.co.uk/ghlala/SQL.zip*

2.1.4.2. *Exact conditional search with a filter by pattern*

Pattern filtering is an application of the regular expression mechanism. This application is done in a limited fashion with the LIKE keyword or in an extended fashion with the REGEX function, which requires additional development since this function is not predefined in SQL Server.

IMPORTANT NOTE.– Management of the indexes associated with the attributes used with the LIKE keyword follows specific rules that may even prevent the use of these indexes.

2.1.4.2.1. Exact conditional search with a filter by pattern using the LIKE operator

This type of search consists of generating a dataset containing only the rows that confirm a mapping with the pattern used with the LIKE keyword in the WHERE clause.

```
USE AdventureWorks2014;
GO
-- Generic character % to replace an
-- indeterminate set of characters.
SELECT *
FROM HumanResources.Employee AS E
WHERE E.JobTitle Like '%Engineer%'
GO
-- Generic character _ to replace a single
-- character.
SELECT *
FROM Person.EmailAddress AS EA
WHERE EA.EmailAddress LIKE'__@@%'
GO
-- Generic character [] to explicitly specify
-- the list of possible characters.
SELECT *
FROM HumanResources.Employee AS E
WHERE E.JobTitle Like '[CS]%'
GO
-- Generic character [] to specify the list of
-- possible characters using an interval.
SELECT *
FROM HumanResources.Employee AS E
WHERE E.JobTitle Like '[A-D]%'
```

```
GO
-- Wildcard character [^] to explicitly specify
-- the list of characters to avoid.
SELECT *
FROM HumanResources.Employee AS E
WHERE E.JobTitle Like '[^CS]%'
GO
-- Wildcard character [^] to specify the list of
-- characters to avoid using an interval.
SELECT *
FROM HumanResources.Employee AS E
WHERE E.JobTitle Like '[^A-D]%'
GO
```

Box 2.2. *Conditional exact search queries with pattern using the LIKE keyword. For a color version of this box, see www.iste.co.uk/ghlala/SQL.zip*

2.1.4.2.2. The PATINDEX function

The PATINDEX function behaves in the same way as the LIKE keyword, but returns the position of a string (pattern) in a target string. The target string can be constant or a column of a table.


```
USE AdventureWorks2014;
GO
-- Use of the PATINDEX function in the - SELECT
clause -- as a constant.
SELECT PATINDEX('%Lock%','Locked')
GO
-- Use of the PATINDEX function in the - SELECT
clause -- as a column of a table.
SELECT PATINDEX('%Lock%', P.Name)
FROM Production.Product AS P
GO
-- Use of the PATINDEX function in the - WHERE
clause.
SELECT *
FROM Production.Product AS P
WHERE PATINDEX('%Lock%', P.Name) = 7;
GO
```

Box 2.3. *Queries with the PATINDEX function. For a color version of this box, see www.iste.co.uk/ghlala/SQL.zip*

IMPORTANT NOTE.– The PATINDEX function returns the value 0 if the specified pattern is not mapped in the target string.

2.1.4.2.3. Exact conditional search with a filter by pattern using the REGEX function

SQL Server does not support the REGEX function in a native way. It is necessary to use the managed code technology known as CLR (Common Language Runtime). This technology consists of creating, within the database, advanced procedural objects (procedures, functions, triggers and aggregations) using programming languages such as C# and VB.NET while making use of the functionalities offered by the .NET Framework.

2.1.5. *Conditional approximate search (full-text search)*

The full-text search is used to remedy deficiencies in searches using the LIKE operator or the REGEXP function. This feature is applicable in a wide range of analytical scenarios for executing complex queries with fuzzy logic rather than exact logic in the data search. The full-text search is based on three essential elements: the execution service, the catalogue and the appropriate indexes.

2.1.5.1. *The full-text search service*

The full-text search is based on an SQL Server service that must be installed and started in order to use this feature. Figure 2.2 shows the stage of installation at which the full-text search service will be installed.

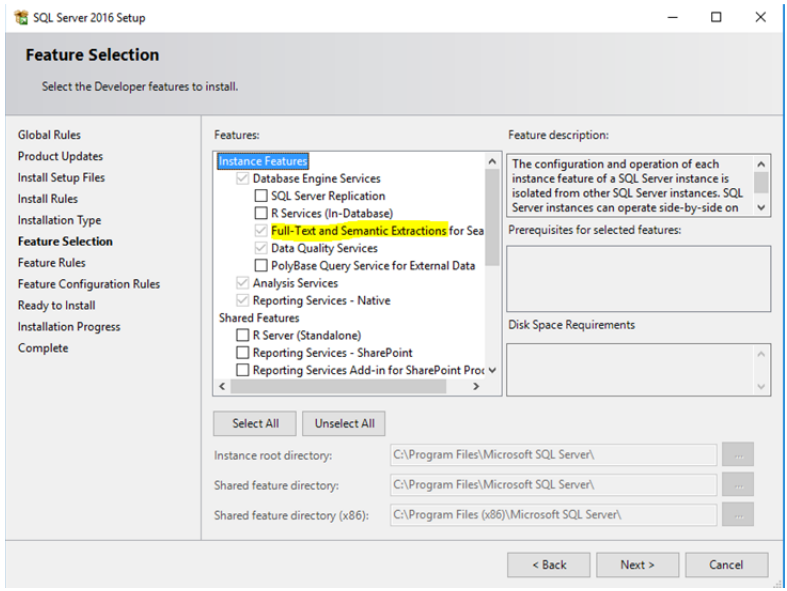


Figure 2.2. *Installation of the full-text search service*

2.1.5.2. *The full-text search catalogue*

The full-text search catalogue is a logical container for a group of indexes related to this type of search. You must create a full-text catalogue before you can create a full-text index. The full-text catalogue is a virtual object that does not belong to any group of files.

```
USE AdventureWorks2014;
GO
CREATE FULLTEXT CATALOG AW_FT_Catalog AS DEFAULT;
GO
```

Box 2.4. *Creation of the catalogue for the full-text search. For a color version of this box, see www.iste.co.uk/ghlala/SQL.zip*

Creation of a full-text catalogue can also be done with Management Studio (SSMS) by following the following steps:

– In the Object Explorer, expand the server, expand Databases and expand the database in which you want to create the full-text catalogue.

– Expand Storage, then right-click on Full-Text Catalogues.

– Select New Full-Text Catalogue.

– In the New Full-Text Catalogue dialog box, specify the catalogue information.

2.1.5.3. *Indexes of type FULLTEXT*

FULLTEXT indexes are required for full-text searches. They can be created in a single column or multiple columns of the same table with or without a "STOPLIST" containing empty words ("STOPWORDS").

```
USE AdventureWorks2014;
GO
CREATE FULLTEXT INDEX ON dbo.Customer(Name)
KEY INDEX PK_Customer_Index
ON AW_FT_Catalog
WITH STOPLIST = OFF
GO
```

Box 2.5. *Creating a FULL TEXT SEARCH index on a single column. For a color version of this box, see www.iste.co.uk/ghlala/SQL.zip*

```
USE AdventureWorks2014;
GO
CREATE FULLTEXT INDEX ON dbo.Customer(Name, Email)
KEY INDEX PK_Customer_Index
ON AW_FT_Catalog
WITH STOPLIST = OFF
GO
```

Box 2.6. *Creating a FULL TEXT SEARCH index on a number of columns. For a color version of this box, see www.iste.co.uk/ghlala/SQL.zip*

IMPORTANT NOTES.– A full-text index can be created on a table or an indexed view in a database in SQL Server. The

full-text index requires a single index, with a single column and which not nullable. Only one full-text index is allowed per table or indexed view. Each full-text index applies to a single table or indexed view. A full-text index can contain up to 1024 columns.

2.1.5.4. *The "STOPLIST"*

A "STOPLIST" is an object manipulated by SQL Server to contain empty words ("STOPWORDS") which are words that have meaning in a specific language. These words are omitted from the full-text index because they are considered unnecessary for searching.

```
USE AdventureWorks2014;
GO
CREATE fulltext stoplist MyStopList;
GO
ALTER fulltext stoplist MyStopList
 ADD'le' LANGUAGE 1036;
GO
-- Index modification with the system STOPLIST.
ALTER fulltext INDEX ON Production.ProductDescription
 SET stoplist system;
-- Index modification with a customised STOPLIST.
ALTER fulltext INDEX ON Production.ProductDescription
 SET stoplist ArcanesStopList;
GO
```

Box 2.7. *Creating a FULL TEXT SEARCH index on a single column with STOPLIST. For a color version of this box, see www.iste.co.uk/ghlala/SQL.zip*

IMPORTANT NOTE.– An index can be created without a STOPLIST, with the system's STOPLIST or with a customized STOPLIST.

2.1.5.5. *The FULLTEXT search*

The FULLTEXT search on one or more indexed columns uses the CONTAINS and FREETEXT full-text predicates used in the WHERE clause of the query to return a Boolean value (true or false) indicating the existence or not of the search term.

Full-text queries are quite powerful and can be of the following types:

– Simple term: searches for a specific word or phrase.

– Prefix term: searches for a prefix word or phrase.

– Proximity term: search for words or expressions close to each other.

– Generation term: search for flexional forms, i.e. the different tenses of a searched verb or the different singular and plural forms of a word.

– Thesaurus: searches for synonyms for a particular word based on a thesaurus (dictionary).

– Weighted term: searches for different search terms according to the weights associated with them.

2.1.5.5.1. Conditional approximate search of the CONTAINS type

CONTAINS offers the flexibility to execute different forms of queries separately.


```
USE AdventureWorks2014;
GO
-- Search in a specified indexed column
-- (Name).
SELECT *
FROM Production.Product AS P
WHERE CONTAINS(P.Name,'flat');
GO
-- Search in several specified indexed
-- columns (FirstName, MiddleName, LastName).
SELECT *
FROM Production.Product AS P
WHERE CONTAINS((P.FirstName, P.LastName),'Hill');
GO
-- Search in all columns of the table (*).
SELECT *
FROM Production.Product AS P
WHERE CONTAINS(P.Name,'flat');
GO
```

```
-- OR Operator
SELECT *
FROM Production.Product AS P
WHERE CONTAINS(*, '"*nut*" OR "*screw*" OR
"*washer*"');
GO
-- AND Operator
SELECT P.ProductID, P.Name
FROM Production.Product AS P
WHERE CONTAINS(P.Name,'"flat" AND "washer"');
GO
-- NOT Operator
SELECT P.ProductID, P.Name
FROM Production.Product AS P
WHERE CONTAINS(P.Name, '"nut" AND NOT "hex"');
GO
-- NEAR Operator
SELECT P.ProductID, P.Name
FROM Production.Product AS P
WHERE CONTAINS([P.Name],'men NEAR shorts');
GO
-- Operator WEIGHT
SELECT P.ProductID, P.Name
FROM Production.Product AS P
WHERE CONTAINS(P.Name,'ISABOUT (nut weight (.8),
bolt weight (.4), washer weight (.2))');
GO
-- Use of variables
DECLARE @Parm1 VARCHAR(50)
SET @Parm1 = 'XL NEAR men NEAR shorts'
SELECT P.ProductID, P.Name
FROM Production.Product AS P
WHERE CONTAINS(P.Name, @Parm1);
GO
```

Box 2.8. *Full-text queries (in red) using the CONTAINS clause. For a color version of this box, see www.iste.co.uk/ghlala/SQL.zip*

2.1.5.5.2. Conditional approximate search of the FREETEXT type

FREETEXT is a more restrictive predicate that, by default, performs a search based on different forms of a word or sentence (in other words, it includes flexional forms as well as the synonym dictionary).


```
USE AdventureWorks2014;
GO
-- FREETEXT search in a specified indexed
-- column (Name)
GO
```

```
SELECT *
FROM Production.Product AS P
WHERE FREETEXT (P.Name,'Book')
GO
-- The same equivalent search but of type CONTAINS
SELECT *
FROM Production.Product AS P
WHERE CONTAINS (P.Name,'Book' or 'books' or 'booked' or
'booking')
GO
```

Box 2.9. *Full-text queries (in red) using the FREETEXT clause. For a color version of this box, see www.iste.co.uk/ghlala/SQL.zip*

2.1.5.5.3. CONTAINSTABLE and FREETEXTTABLE

CONTAINSTABLE and FREETEXTTABLE act in the same way as CONTAINS and FREETEXT, with the addition of storing results in virtual tables during the execution request. These virtual tables can simulate real tables, hence the possibility of their use in joins.

```
USE AdventureWorks2014;
GO
SELECT FT_TBL.ProductID, FT_TBL.[Name], KEY_TBL.RANK
FROM Production.Product AS FT_TBL
INNER CONTAINSTABLE JOIN (Production.Product, *,
'"*washer*" OR "*ball*"') AS KEY_TBL
ON FT_TBL.ProductID = KEY_TBL.[KEY]
WHERE KEY_TBL.RANK > 100
ORDER BY KEY_TBL.RANK DESC
GO
```

Box 2.10. *Join between a table and the result of CONTAINSTABLE. For a color version of this box, see www.iste.co.uk/ghlala/SQL.zip*

```
USE AdventureWorks2014;
GO
SELECT FT_TBL.ProductID, FT_TBL.Name, KEY_TBL.RANK
FROM Production.Product AS FT_TBL
INNER JOIN FREETEXTTABLE(Production.Product, *,
'washer ball') AS KEY_TBL
ON FT_TBL.ProductID = KEY_TBL.KEY
WHERE KEY_TBL.RANK > 100
ORDER BY KEY_TBL.RANK DESC
GO
```

Box 2.11. *Join between a table and the result of FREETEXTTABLE. For a color version of this box, see www.iste.co.uk/ghlala/SQL.zip*

2.1.6. *Conclusion*

Data filtering is one of the essential features of the SQL language. It allows you to specify, with the required accuracy, the dataset to which the data analysis will be applied.

2.2. Sorting data

2.2.1. *Introduction*

Databases have changed the landscape of information processing over multiple axes from the previous technology that was data files. In the past, developers described the physical order of data storage by specifying the organization of the data file as sequential, indexed sequential or relative. Currently, and with database technology, physical storage management is a matter for the DBMS alone in order to ensure maximum optimization of the management of data transfer between the main memory and the hard disk.

This strategic choice relating to the physical storage of data generated a state of affairs resulting in the need for a clause (ORDER BY) in the SQL query to sort the results extracted from the database according to the user's preferences.

2.2.2. *Mind map of the second section*

This second section will be devoted to the different options relating to the ORDER BY clause.

Figure 2.3 presents the mind map of the different concepts that will be covered in this section.

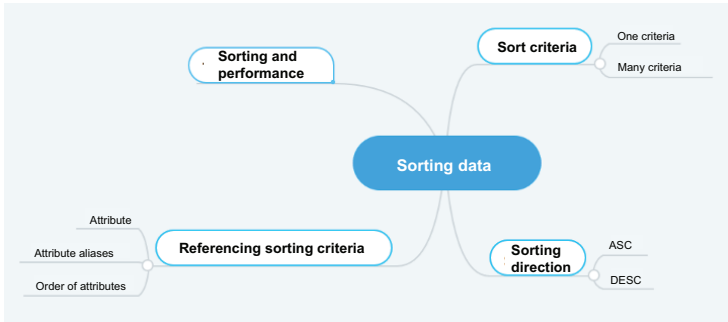


Figure 2.3. *Mind map of the second section*

2.2.3. *Sort criteria*

The dataset returned by the SQL query does not correspond to any semantic order of data. Rather, it is generated according to the storage strategy adopted by the DBMS, which is based on factors that promote the optimization of data restitution and ignores any semantic aspect of the data. The user imposes their own sorting preferences with one or more criteria using the ORDER BY clause.

```
USE AdventureWorks2014;
GO
SELECT P.Name, P.ListPrice, P.Color
FROM Production.Product AS P
ORDER BY P.Color
GO
```

Box 2.12. *Sort the data with only one criterion. For a color version of this box, see www.iste.co.uk/ghlala/SQL.zip*

```
USE AdventureWorks2014;
GO
SELECT P.Name, P.ListPrice, P.Color
FROM Production.Product AS P
ORDER BY P.Color, P.ListPrice
GO
```

Box 2.13. *Sort the data with several criteria. For a color version of this box, see www.iste.co.uk/ghlala/SQL.zip*

IMPORTANT NOTE.– Sorting data with several criteria defaults to sorting the data with the first criterion. The other criteria will only be used if there is a tie.

2.2.4. *Sorting direction*

The dataset sorted with the ORDER BY clause can be in ascending order (from the smallest to the largest) with the ASC option or in descending order (from the largest to the smallest) with the DESC option.

```
USE AdventureWorks2014;
GO
SELECT P.Name, P.ListPrice, P.Color
FROM Production.Product AS P
ORDER BY P.ListPrice ASC
GO
```

Box 2.14. *Sorting data in ascending order. For a color version of this box, see www.iste.co.uk/ghlala/SQL.zip*

```
USE AdventureWorks2014;
GO
SELECT P.Name, P.ListPrice, P.Color
FROM Production.Product AS P
ORDER BY P.ListPrice DESC
GO
```

Box 2.15. *Sorting data in descending order. For a color version of this box, see www.iste.co.uk/ghlala/SQL.zip*

IMPORTANT NOTE.– In case the sorting direction is not specified, it will default to the ascending direction (ASC).

2.2.5. *Referencing of sorting criteria*

To sort a dataset returned by an SQL query, the ORDER BY clause provides several alternatives to reference the attributes (columns) that will be used for this sorting operation. Indeed, we can choose between the name of the

attribute, its alias or even its order number in the SELECT clause.

```
USE AdventureWorks2014;
GO
SELECT P.Name, P.ListPrice, P.Color
FROM Production.Product AS P
ORDER BY P.ListPrice
GO
```

Box 2.16. *Referencing the sort criterion by attribute name. For a color version of this box, see www.iste.co.uk/ghlala/SQL.zip*

```
USE AdventureWorks2014;
GO
SELECT P.Name, P.ListPrice AS Price, P.Color
FROM Production.Product AS P
ORDER BY Price
GO
```

Box 2.17. *Referencing the sort criterion by attribute alias. For a color version of this box, see www.iste.co.uk/ghlala/SQL.zip*

```
USE AdventureWorks2014;
GO
SELECT P.Name, P.ListPrice, P.Color
FROM Production.Product AS P
ORDER BY 2,3
GO
```

Box 2.18. *Referencing sorting criteria by attribute order numbers in the SELECT clause. For a color version of this box, see www.iste.co.uk/ghlala/SQL.zip*

IMPORTANT NOTE.– In the case of displaying a dataset containing all the columns of the table (SELECT *), it will be possible to sort this dataset by attribute name and also by order number, which here represents the order number of the column in the table.

2.2.6. *Sorting and performance*

Sorting is a very resource-intensive operation. Indeed, it consumes about half of the total processor time allocated to the entire request and requires a lot of buffer memory. To get an idea of the time consumed by the ORDER BY clause, you can select the request and display its estimated execution plan. Figure 2.4 shows the high cost of the ORDER BY clause in a request.

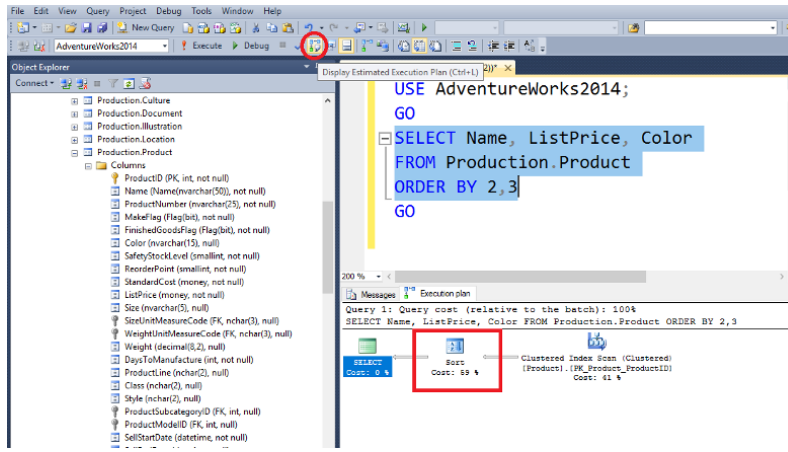


Figure 2.4. *Estimated execution plan of a request*

IMPORTANT NOTES.– The estimated execution plan of a request can be displayed on demand as shown in Figure 2.4 or by using the "Display Estimated Execution Plan" command in the "Query" menu. This plan can also be displayed permanently and in its actual rather than estimated version after each execution of each request by activating the "Include Actual Execution Plan" command in the "Query" menu.

The performance of queries, especially those that use sorting operations, can be improved by various means, the most important of which are indexes.

2.2.7. *Conclusion*

Databases have transformed the operation of sorting data from a characteristic attached to the mechanism of storing this data to a functionality requested by the user when querying this data. This separation between how the data is stored and how it is displayed has provided the DBMS with more flexibility in optimizing storage and has given the user more choice in customizing the display.

2.3. Data pagination

2.3.1. *Introduction*

Pagination in SQL queries consists of customizing the number of lines to be displayed as the result of the query. This functionality is implemented in different ways depending on the DBMS editor, such as Microsoft SQL Server using the TOP option, Oracle using the ROWNUM clause, MySQL using the LIMIT clause, etc.

Data pagination can also be implemented in the application logic. In this case, the program written in C#, PHP, Java or others launches a request to retrieve all the resulting lines and then manages the pagination with its own tools.

The SQL 2011 standard has worked on this functionality to standardize it with the OFFSET… FETCH clause. This new feature has been available since SQL Server 2012.

2.3.2. *Mind map of the third section*

This third section will be devoted to pagination of results with the Microsoft SQL Server-specific approach and the proposed standardized approach of the SQL 2011 standard.

Figure 2.5 presents the mind map of the different concepts that will be covered in this section.

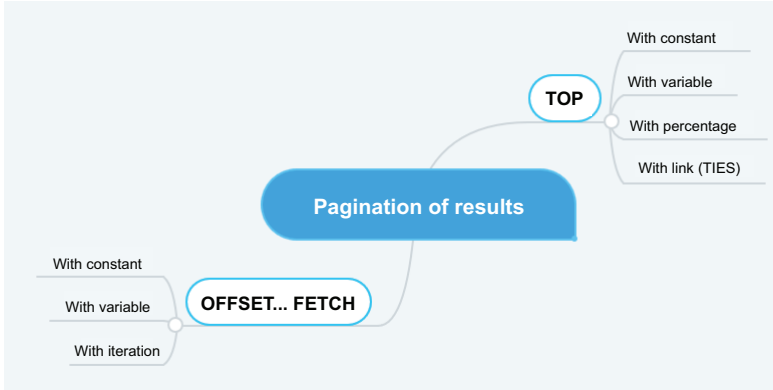


Figure 2.5. *Mind map of the third section*

2.3.3. *The TOP option*

The TOP option has been implemented in the Microsoft SQL Server since its first versions. It provides the user with a simulation of the pagination functionality.

This solution, although not complete, has long been used to satisfy users' needs in terms of pagination of results.

IMPORTANT NOTES.– The TOP option requires the ORDER BY clause in the query and allows you to specify the number of lines to be displayed, always starting from the beginning of the set. The number of lines can be expressed as a constant, an expression or a percentage.

The WITH TIES option added to the TOP option ensures the continuity of the display of lines in the order indicated in the ORDER BY clause, even if the number of lines indicated with TOP is exceeded.

```
USE AdventureWorks2014;
GO
SELECT TOP (10) *
FROM Sales.SalesOrderDetail AS SD
ORDER BY SD.OrderQty
GO
```

Box 2.19. *TOP option with a constant. For a color version of this box, see www.iste.co.uk/ghlala/SQL.zip*

```
USE AdventureWorks2014;
GO
DECLARE @p AS int = 10;
SELECT TOP (@p) PERCENT *
FROM Sales.SalesOrderDetail AS SD
ORDER BY SD.OrderQty
GO
```

Box 2.20. *TOP option with one variable. For a color version of this box, see www.iste.co.uk/ghlala/SQL.zip*

```
USE AdventureWorks2014;
GO
SELECT TOP (10) PERCENT *
FROM Sales.SalesOrderDetail AS SD
ORDER BY SD.OrderQty
GO
```

Box 2.21. *TOP option with a percentage. For a color version of this box, see www.iste.co.uk/ghlala/SQL.zip*

```
USE AdventureWorks2014;
GO
SELECT TOP (10) WITH TIES *
FROM Sales.SalesOrderDetail AS SD
ORDER BY SD.OrderQty
GO
```

Box 2.22. *TOP option with TIES. For a color version of this box, see www.iste.co.uk/ghlala/SQL.zip*

2.3.4. *The OFFSET… FETCH clause*

The importance of pagination has prompted the international community around SQL to integrate this

functionality into the SQL 2011 standard. Indeed, pagination is highly requested by web developers and mobile applications. As soon as it was standardized, Microsoft SQL Server integrated it into its 2012 version.

The power of pagination in its standard version lies in giving the user the possibility of operating and configuring a specific number of lines (FETCH) from any position (OFFSET) of the result set.

```
USE AdventureWorks2014;
GO
SELECT P.BusinessEntityID, P.FirstName, P.LastName
FROM Person.Person AS P
ORDER BY P.BusinessEntityID
OFFSET 10 ROWS
FETCH NEXT 10 ROWS ONLY
GO
```

Box 2.23. *OFFSET… FETCH clause with constant. For a color version of this box, see www.iste.co.uk/ghlala/SQL.zip*

```
USE AdventureWorks2014;
GO
-- Number of page requested (6)
DECLARE @NumPage int = 6
-- Number of lines per page (10)
DECLARE @NbrLigneParPage int = 10
/* Return @Nbr_Line_per_page after a 5-page jump
(@NumPage-1)*@NbrLinePerPage */
SELECT P.BusinessEntityID, P.FirstName, P.LastName
FROM Person. Person AS P
ORDER BY P.BusinessEntityID
OFFSET (@Number-1)*@NbrLinePerPage ROWS
FETCH NEXT @ NbrLigneParPage ROWS ONLY
GO
```

Box 2.24. *OFFSET… FETCH clause with variable. For a color version of this box, see www.iste.co.uk/ghlala/SQL.zip*

```
USE AdventureWorks2014;
GO
-- Declaration of a counter
DECLARE @Counter int = 0
-- Number of lines per page (10)
DECLARE @NbrLineByPage int = 10
/* Return the first six pages */
```

```
WHILE (@Counter < 6)
Begin
 SELECT P.BusinessEntityID, P.FirstName,
P.LastName
 FROM Person.Person AS P
 ORDER BY P.BusinessEntityID
 OFFSET @NbrLineByPage ROWS
 FETCH NEXT @NbrLineByPage ROWS ONLY
 SET @Counter = @Counter +1;
End;
GO
```

Box 2.25. *OFFSET… FETCH clause with iteration. For a color version of this box, see www.iste.co.uk/ghlala/SQL.zip*

2.3.5. *Conclusion*

Pagination of results is a feature that is in high demand. The SQL 2011 standard has developed the necessary mechanisms to ensure this functionality through the DBMS according to the Database-First approach, thus freeing developers from additional pagination work that is generally difficult for them and not optimized for the end-user.

2.4. Subqueries

2.4.1. *Introduction*

Modularity in procedural programming is a practical approach to solving complex problems by subdividing them into elementary, easily manageable sub-problems. This strategy is also valid in SQL language through a very efficient and effective mechanism, namely the subquery technique.

Subqueries are nested queries, i.e. queries within queries. The results of an internal query are transmitted to the external query as intermediate values used in expressions of the latter to generate the final result.

The subquery technique can be used in the different clauses of the main (external) request. In this section, we study the use of subqueries in the WHERE clause.

2.4.2. *Mind map of the fourth section*

This fourth section will be devoted to the subquery technique with its different variants.

Figure 2.6 presents the mind map of the different concepts that will be covered in this section.

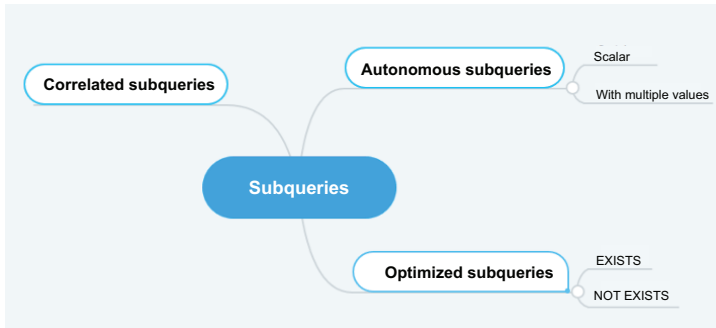


Figure 2.6. *Mental map of the fourth section*

2.4.3. *Autonomous subqueries*

Autonomous subqueries are solutions to respond to requests by writing queries that themselves require values to be determined so that they can be executed. These values, instead of being provided in the main or external query, will be replaced by internal queries to calculate them.

Figure 2.7 shows the operation of autonomous subqueries.

2.4.3.1. *Autonomous scalar subqueries*

Scalar autonomous subqueries are characterized by an internal request that returns a singleton. This value will be

treated in the external request as a constant. To illustrate autonomous scalar subqueries, the following query is used: “Display the names of products that have a higher price than the ‘Blade’ product”.

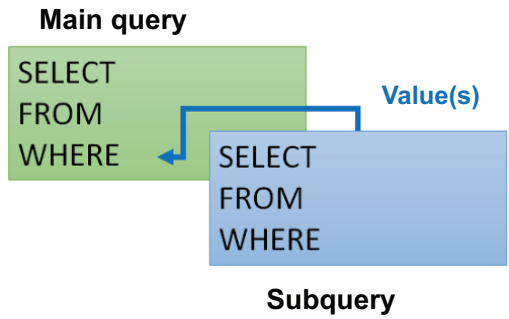


Figure 2.7. *Operation of autonomous subqueries*

This interrogation can be performed with procedural logic. Good practice is to apply the subquery mechanism.

```
USE AdventureWorks2014;
GO
-- Declaration of a variable
DECLARE @price Float
-- Recovery of the price of the ‘Blade’ product.
SELECT @price = P.ListPrice
FROM Production.Product AS P
WHERE P.Name ='Blade'.
-- Display of the names of products whose price is
-- higher than that of the ‘Blade’ product.
SELECT P.Name
FROM Production.Product AS P
WHERE P.ListPrice > @price
GO
```

Box 2.26. *Resolution with procedural logic (method not recommended). For a color version of this box, see www.iste.co.uk/ghlala/SQL.zip*

```
USE AdventureWorks2014;
GO
SELECT PP.Name
FROM Production.Product AS PP
WHERE PP.ListPrice > (SELECT SP.ListPrice
 FROM Production.Product AS SP
 WHERE SP.Name ='Blade')
GO
```

Box 2.27. *Resolution with the subquery technique (recommended method). For a color version of this box, see www.iste.co.uk/ghlala/SQL.zip*

IMPORTANT NOTES.– In this type of subquery, all operators for comparing two numerical values (=, <, >, <=, >=, <>, etc.) are possible.

When the internal request does not return a value, the value NULL will be transmitted to the external request.

The execution plan for this type of subquery shows that the execution of the internal request is first followed by the execution of the external request. It also shows that the subquery technique is implemented by the database engine with joins.

2.4.3.2. *Autonomous multiple-value subqueries*

Autonomous multiple-value subqueries are characterized by an internal query that returns a set of values as a single column set. In the external query, we must use the belonging operator (IN) in expressions using this set returned by the internal query. To illustrate the autonomous multiple-value subqueries, we will execute the following query: “Display the colors of the products ordered in order number 43659”.

```
USE AdventureWorks2014;
GO
SELECT PP.Color
FROM Production.Product AS PP
WHERE PP.ProductID IN
 (SELECT PS.ProductID
 FROM Sales.SalesOrderDetail AS PS
```

```
 WHERE PS.SalesOrderID = 43659)
GO
```

Box 2.28. *Autonomous multi-valued subquery. For a color version of this box, see www.iste.co.uk/ghlala/SQL.zip*

IMPORTANT NOTES.– The other operators used in the autonomous multi-valued subqueries are:

– NOT IN: this refers to non-membership.

– ANY: this is equivalent to the operator IN.

– <> ALL: this is equivalent to the NOT IN operator.

2.4.4. *Correlated subqueries*

Correlated subqueries are solutions to respond to requests that require internal queries executed as many times as the number of lines of the external query. Execution of the internal request follows the receipt of information from the line of the external request. Figure 2.8 shows the behavior of the two internal and external queries, which is similar to the joining of two datasets.

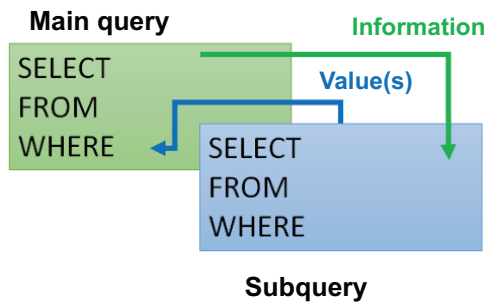


Figure 2.8. *Operation of correlated subqueries*

To illustrate the correlated subqueries, the following query is used: "Display the names of products that are sold with prices (UnitPrice) equal to their cost prices (ListPrice)".

```
USE AdventureWorks2014;
GO
SELECT DISTINCT PP.ProductID
FROM Production.Product AS PP
WHERE PP.ListPrice IN
 (SELECT DISTINCT PS. UnitPrice
 FROM Sales.SalesOrderDetail AS PS
 WHERE PP.ProductID = PS.ProductID)
GO
```

Box 2.29. *Correlated subqueries. For a color version of this box, see www.iste.co.uk/ghlala/SQL.zip*

2.4.5. *Optimized subqueries*

Optimized subqueries are special cases of subqueries that are limited to a partial browse of the dataset in the main query. They are performed with the EXISTS keyword just to check the existence of an element in the main query and not all its possible occurrences. This type of subquery is optimized especially with large datasets processed by the subquery.

To illustrate optimized subqueries, the following query is used: "Display the names of products that have been ordered at least once".

```
USE AdventureWorks2014;
GO
SELECT PP.Name
FROM Production.Product AS PP
WHERE PP.ProductID IN
 (SELECT PS.ProductID
 FROM Sales.SalesOrderDetail AS PS)
GO
```

Box 2.30. *Subqueries not optimized for a total browse. For a color version of this box, see www.iste.co.uk/ghlala/SQL.zip*

```
USE AdventureWorks2014;
GO
SELECT PP.Name
FROM Production.Product AS PP
WHERE EXISTING
 (SELECT *
 FROM Sales.SalesOrderDetail AS PS
 WHERE PP.ProductID = PS.ProductID)
GO
```

Box 2.31. *Subqueries optimized for partial browsing. For a color version of this box, see www.iste.co.uk/ghlala/SQL.zip*

IMPORTANT NOTE.– The negation of existence (NOT EXISTS) is not very useful because in all cases the execution will be done through a total browse. However, the interweaving of two negations of existence can be used to simulate the operation of dividing one dataset by another. This operation, although it is a relational operation, is not implemented in SQL language.

2.4.6. *Conclusion*

The resolution of complex queries in SQL language requires writing different unified queries to generate the final result. Two scenarios are possible: use of the subquery technique or the application of procedural logic by linking these different requests with variables. Good practice recommends the use of the subquery technique to take advantage of these advantages in terms of semantics and optimization.

2.5. Options for the FROM clause

2.5.1. *Introduction*

An SQL query is formed by clauses, each of which has a specific role. Given that the order of writing these clauses must be respected and that the indented style is strongly

recommended, the database engine adopts another order of execution of these clauses.

Figure 2.9 shows the writing order as well as the execution order of the different clauses of an SQL query.

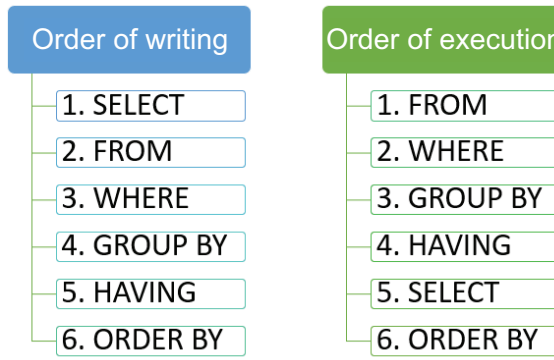


Figure 2.9. *Order of writing and execution of the different clauses of a query*

The importance of the FROM clause comes from its position in the process of executing the query. This first-rank position allows you to specify the source of the data to be analyzed.

IMPORTANT NOTE.– The GROUP BY and HAVING clauses are used only with the aggregation functions that will be discussed in the second section of Chapter 4.

2.5.2. *Mind map of the fifth section*

This fifth section will be devoted to presenting the different alternatives for the FROM clause. Figure 2.10 presents the mind map of the different concepts that will be covered in this section.

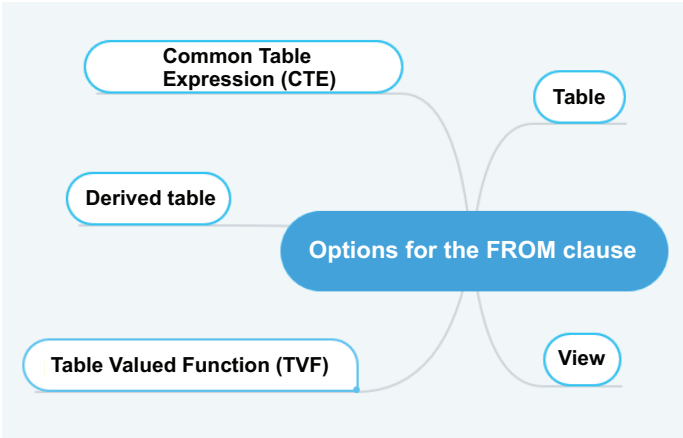


Figure 2.10. *Mind map of the fifth section*

2.5.3. *Tables*

Tables are the most common alternative to the FROM clause. These tables are referenced by their names with optional specification of the schema and even the database.

Data schemas are organizational structures that allow you to distribute database objects and tables in particular to different areas, so as to increase readability and better manage administrative tasks such as access rights.

Databases can also be included in table names. This allows us to write analysis queries for data from different databases in the same SQL Server instance.

IMPORTANT NOTES.– The table name can also contain the name of the server that hosts the database containing this table via the Linked Server technique. This technique allows you to write queries for analysis of data from different heterogeneous servers (from different editors such as Oracle, MySQL, DB2, etc.).

The use of aliases for table names is strongly recommended for readability and optimization reasons.

2.5.4. *Views*

A view is an object in the database. This is a query stored in the server that is created by the CREATE VIEW data definition command. Its creation persists in the server until modification of its associated query with the ALTER VIEW command or its complete deletion with the DROP VIEW command. Views are very useful on several axes depending on the context of their use.

Figure 2.11 shows the four main reasons for using views:

– security: controlling access to table data through filters and projections when creating the view;

– optimization: saving response time through precompiled queries;

– complex calculations: pre-establishing complex calculations that will be performed when the view is queried. This feature overcomes the normalization constraint that recommends not including data calculated when modeling the tables;

– multiple sources: specifying datasets from different sources through complex joins. The end-user later queries the view as a single unified data source with full transparency as regards its definition.

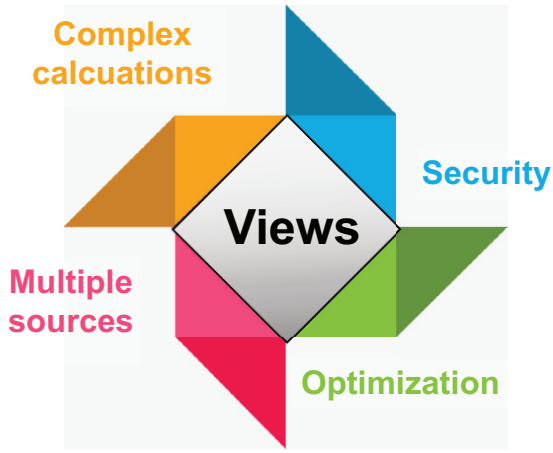


Figure 2.11. *Reasons for using views*

```
USE AdventureWorks2014
GO
CREATE VIEW Detaille_Ventes
WITH ENCRYPTION, SCHEMABINDING
AS
SELECT P.Name, P.ListPrice, OD.UnitPrice, OD.OrderQty,
  (P.ListPrice-OD.UnitPrice)*OD.OrderQty AS Margin
FROM Sales.SalesOrderDetail AS OD
INNER JOIN Production.Product AS P
ON (P.ProductID = OD.ProductID)
GO
SELECT *
FROM Detaille_Ventes AS DV
GO
```

Box 2.32. *Using views in the FROM clause. For a color version of this box, see www.iste.co.uk/ghlala/SQL.zip*

IMPORTANT NOTES.– View management such as creating, modifying, deleting and even querying requires access *privileges* provided by the database administrator to users of the view.

Creating a view consists of compiling and storing the request associated with the view in the server. The physical duplication of data can be ensured by indexing these views.

The ENCRYPTION option hides the view code, and the SCHEMABINDING option means that the table structures referenced by the view cannot be modified in such a way as to affect the view definition.

2.5.5. *Derived tables*

Derived tables are named query expressions embedded in the main query at the FROM clause level. They are treated as virtual tables that are not stored in the database. Derived tables are advantageous since they do not require access privileges to integrate them into other queries, but they have the disadvantage of degrading the readability of the latter.

```
USE AdventureWorks2014
GO
SELECT Name, Marge
FROM (
 SELECT P.Name, P.ListPrice,
 OD.UnitPrice, OD.OrderQty,
 (P.ListPrice-OD.UnitPrice)*OD.OrderQty AS Margin
 FROM Sales.SalesOrderDetail AS OD
 INNER JOIN Production.Product AS P
 ON (P.ProductID = OD.ProductID)
 ) AS Table_Ventes
GO
```

Box 2.33. *Using derived tables in the FROM clause. For a color version of this box, see www.iste.co.uk/ghlala/SQL.zip*

IMPORTANT NOTE.– The query relating to the derived table must be named using an alias.

2.5.6. *Table-valued functions (TVFs)*

The procedural extension of the SQL language provides a very practical mechanism as an option for the FROM clause. Functions that return tables, designated as table-valued functions (TVFs), are functions whose return result is a dataset that can be assimilated to a virtual table.

```
USE AdventureWorks2014
GO
CREATE FUNCTION TVF_Retail_Sales (@C VARCHAR(10))
TABLE RETURNS
SCHEMABINDING, ENCRYPTION
AS RETURN
 SELECT P.Name, P.ListPrice,
 OD.UnitPrice, OD.OrderQty,
 (P.ListPrice-OD.UnitPrice)*OD.OrderQty AS Margin
 FROM Sales.SalesOrderDetail AS OD
 INNER JOIN Production.Product AS P
 ON (P.ProductID = OD.ProductID)
 WHERE P.Color = @C
GO
SELECT *
FROM TVF_Retail_Sales ('Red')
```

Box 2.34. *Using a table-valued function in the FROM clause. For a color version of this box, see www.iste.co.uk/ghlala/SQL.zip*

IMPORTANT NOTES.– The dataset returned by the function is determined by the SELECT query. It is possible to define the structure of the set to be returned through a TABLE variable in the function header and then specify its content in the body of the function through several data manipulation requests (INSERT, UPDATE and DELETE). The body of the function must contain the RETURN instruction to return the result variable containing the dataset.

The management of procedural objects and in particular functions returning tables also requires access rights provided by the database administrator to developers and/or end-users.

The strong point of functions that return tables as FROM clause options is the configuration of datasets.

2.5.7. *Common table expressions (CTEs)*

Common table expressions (CTEs) represent a good compromise between the readability of the code and the security constraints imposed by the database administrator.

Indeed, CTEs are similar to derived tables but defined separately in the FROM clause. This definition, which is done through the keyword WITH and does not require access rights, is temporary and localized to the current request.

```
USE AdventureWorks2014
GO
WITH Details_Ventes
AS (
 SELECT P.Name, P.ListPrice,
 OD.UnitPrice, OD.OrderQty,
 P.ListPrice-OD.UnitPrice)*OD.OrderQty AS Marge
 FROM Sales.SalesOrderDetail AS OD
 INNER JOIN Production.Product AS P
 ON (P.ProductID = OD.ProductID)
 )
SELECT DV.Name, DV. Marge
FROM Details_Ventes AS DV
GO
```

Box 2.35. *Using common table expressions in the FROM clause. For a color version of this box, see www.iste.co.uk/ghlala/SQL.zip*

IMPORTANT NOTES.– The CTE may create a reference several times in the request.

In the same query, several CTEs can be created.

The lifetime of the CTE is that of the request. It does not persist in the server and cannot be shared between different requests.

2.5.8. *Conclusion*

Data analysis always begins with identifying the data source to be analyzed. In an SQL query, this task is assigned to the FROM clause that determines the source(s) of row, selective or derived data.

3

Operators

3.1. Joins

3.1.1. *Introduction*

Joining in relational databases means extracting data from multiple linked tables. The data to be searched is distributed over several tables to comply with the normalization rules of the relational model. This standardization avoids or minimizes data redundancy while ensuring the referential integrity of the data.

The SQL1 standard (SQL'89) did not implement joins, but proposed a mechanism that simulated this operation by using a Cartesian product followed by a filter. This practice, although it continues to exist, is strongly discouraged because it has a very negative impact on the readability and performance of the query.

The SQL2 standard (SQL'92) quickly remedied this problem with the invention of the JOIN operator. The latter normalizes the join operation through improving the readability of the query by giving it semantics. This JOIN operator also reflects considerable processing by the DBMS to improve performance. These efforts are materialized

Analytic SQL in SQL Server 2014/2016, First Edition. Riadh Ghlala.

by three optimization algorithms which are NESTED LOOP, MERGE SORT and HASH. These concepts fall within the theme of SQL query performance generally referred to as SQL Tuning, which will not be addressed in this book.

The following example illustrates this poor practice for displaying the list of products with their respective subcategories.

```
USE AdventureWorks2014;
GO
SELECT P.Name AS Product,
 S.Name AS Category
FROM Production.Product AS P,
 Production.ProductSubcategory AS S
WHERE P.ProductSubcategoryID = S.ProductSubcategoryID;
GO
```

Box 3.1. *Simulation of a join with a Cartesian product followed by a filter. For a color version of this box, see www.iste.co.uk/ghlala/SQL.zip*

3.1.2. *Mind map of the first section*

This first section will be devoted to the study of the JOIN operator.

Figure 3.1 presents the mind map of the different concepts that will be covered in this section.

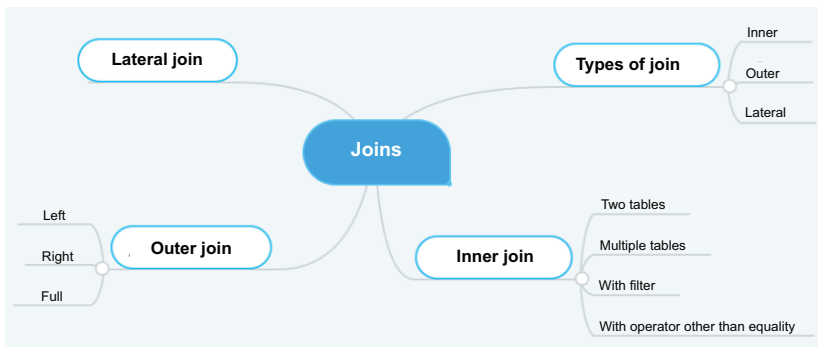


Figure 3.1. *Mind map of the first section*

3.1.3. *Types of joins*

There are two main categories of joins:

– INNER JOIN: consists of searching data from table T1 with additional information from table T2. This complement is based on correspondence between the rows via one or more common columns of the two tables.

– OUTER JOIN: based on table T1 in order to display an exhaustive list of its rows with a complement from table T2 through correspondence between the rows via one or more common columns between the two tables. As described so far, an outer join acts like an inner join; the difference lies in the following points:

- the outer join must be based on a reference table T1 to display a complete list of its lines. This indication is made with the LEFT, RIGHT or FULL clause;

- rows in table T1 that do not match in table T2 will still be displayed with a complement in the form of null values.

IMPORTANT NOTES.– In the case of an external join, the word OUTER is optional, but the direction of the join (LEFT, RIGHT or FULL) is mandatory. LEFT OUTER JOIN ≡ LEFT JOIN.

A join is based on one or more common columns between the processed tables.

The condition of comparison is specified in the ON clause. This condition may be equality or any other condition (<, >, IN, BETWEEN, etc.). In the case of a condition of equality, the join is qualified as an equijoin.

A join is generally made between two tables. In the case of a join of multiple tables, the query will be a nesting of several joins into two tables.

A join can be made with a single table. This is a self-joining in which the table plays a double role. These roles are clarified through aliases.

A join can be performed between a table and the dataset returned by a function. In this case, the join is called a lateral join; it is performed with the APPLY operator and the function must be of TVF the type.

3.1.4. *Inner join*

The inner join is the essential operator in almost all SQL queries to extract data in a relational context based on the concept of standardization.

3.1.4.1. *Inner join between two tables*

To illustrate an inner join between two tables, see the following query: "Display product names with the names of their subcategories".


```
USE AdventureWorks2014
GO
SELECT P.Name AS Product,
 SC.Name AS Subcategory
FROM Production.Product AS P
JOIN Production.ProductSubcategory AS SC
ON (P.ProductSubcategoryID = SC.ProductSubcategoryID);
GO
```

Box 3.2. *Inner join between two tables. For a color version of this box, see www.iste.co.uk/ghlala/SQL.zip*

3.1.4.2. *Inner join between several tables*

To illustrate an inner join between multiple tables, see the following query: “Display product names with their category names”.

```
USE AdventureWorks2014
GO
SELECT P.Name AS Product,
 C.Name AS Category
FROM Production.Product AS P
JOIN Production.ProductSubcategory AS SC
ON (P.ProductSubcategoryID = SC.ProductSubcategoryID)
JOIN Production.ProductCategory AS C
ON (SC.ProductCategoryID = C.ProductCategoryID);
GO
```

Box 3.3. *Inner join between multiple tables. For a color version of this box, see www.iste.co.uk/ghlala/SQL.zip*

3.1.4.3. *Inner join with a filter*

To illustrate an inner join with a filter, see the following request: “Display the names of luxury products (price >= 1000) with the names of their categories”.

```
USE AdventureWorks2014;
GO
SELECT P.Name AS Product,
 P.ListPrice AS UnitPrice,
 C.Name AS Category
FROM Production.Product AS P
JOIN Production.ProductSubcategory AS SC
ON (P. ProductSubcategoryID = SC.ProductSubcategoryID)
JOIN Production.ProductCategory AS C
ON (SC.ProductCategoryID = C.ProductCategoryID)
AND P.ListPrice >= 1000;
GO
```

Box 3.4. *Inner join with a filter (method not recommended). For a color version of this box, see www.iste.co.uk/ghlala/SQL.zip*

```
USE AdventureWorks2014;
GO
SELECT P.Name AS Product,
 P.ListPrice AS UnitPrice,
 C.Name AS Category
FROM Production.Product AS P
JOIN Production.ProductSubcategory AS SC
ON (P.ProductSubcategoryID = SC.ProductSubcategoryID)
JOIN Production.ProductCategory AS C
ON (SC.ProductCategoryID = C.ProductCategoryID)
WHERE P.ListPrice >= 1000;
GO
```

Box 3.5. *Inner join with a filter (recommended method). For a color version of this box, see www.iste.co.uk/ghlala/SQL.zip*

IMPORTANT NOTES.– Do not confuse the AND operator with the WHERE clause in a join with a filter. Even though we get the same result in this case, we must always use the WHERE clause to filter the data and not the AND operator. Confusion between these two elements can generate unexpected results in the case of an external join.

3.1.4.4. *Inner join with the same table (self-join)*

To illustrate an inner join with the same table, also referred to as the self-join, see the following query: "Display the names of product pairs that have the same prices, colors, styles and sizes".

```
USE AdventureWorks2014;
GO
SELECT P1.Name AS PRODUCT1, P2.Name AS PRODUCT2
FROM Production.Product AS P1
JOIN Production.Product AS P2
ON (P1.ListPrice = P2.ListPrice)
AND (P1.Color = P2.Color)
AND (P1.Style = P2.Style)
AND (P1.Size = P2.Size);
GO
```

Box 3.6. *Self-join (216 lines). For a color version of this box, see www.iste.co.uk/ghlala/SQL.zip*

IMPORTANT NOTES.– This query is a single join with several conditions over several common columns.

The request does not exactly meet our needs because the displayed lines (216 lines) contain different and identical product pairs. It also includes duplications of pairs by changing the positions of the products in these pairs.

```
USE AdventureWorks2014;
GO
SELECT TOP 50 PERCENT
 P1.Name AS PRODUCT1, P2.Name AS PRODUCT2
FROM Production.Product AS P1
JOIN Production.Product AS P2
ON (P1.ListPrice = P2.ListPrice)
AND (P1.Color = P2.Color)
AND (P1.Style = P2.Style)
AND (P1.Size = P2.Size)
AND (P1.ProductID <> P2.ProductID);
GO
```

Box 3.7. *Self-join (three lines). For a color version of this box, see www.iste.co.uk/ghlala/SQL.zip*

IMPORTANT NOTES.– The above examples relate to equijoins (inner joins whose condition of correspondence is equality). In this join, we used a condition other than equality.

3.1.5. *Outer join*

An outer join consists of displaying the complete list of lines of table T1 with possibly additional data from table T2. The keywords LEFT, RIGHT or FULL are used to determine the direction of this join.

3.1.5.1. *Left outer join*

To illustrate a left outer join, see the following query: “Display product names with eventually the names of their subcategories”.

```
USE AdventureWorks2014;
GO
SELECT P.Name AS Product,
 SC.Name AS SubCategory
FROM Production.Product AS P
LEFT OUTER JOIN Production.ProductSubcategory AS SC
ON (P.ProductSubcategoryID = SC.ProductSubcategoryID);
GO
```

Box 3.8. *Left outer join. For a color version of this box, see www.iste.co.uk/ghlala/SQL.zip*

IMPORTANT NOTES.– This query displays all products (504 lines). Even those that are not classified, i.e. have no categories, will be displayed with null values for the columns related to their categories.

The number of rows displayed by this query is easy to determine (cardinality of the `Product` table) because the association between this table and the `ProductSubcategory` table in this joining direction is of type (1/1). The number will be difficult to determine in the opposite direction.

The request can use the LEFT JOIN syntax instead of LEFT OUTER JOIN.

3.1.5.2. *Right outer join*

To illustrate the right outer join (RIGHT OUTER JOIN), see the following query: "Display the names of the categories with eventually the names of the products that correspond to them".

```
USE AdventureWorks2014;
GO
SELECT P.Name AS Product,
 SC.Name AS SubCategory
FROM Production.Product AS P
RIGHT OUTER JOIN Production.ProductSubcategory AS SC
ON (P.ProductSubcategoryID = SC.ProductSubcategoryID);
GO
```

Box 3.9. *Right outer join. For a color version of this box, see www.iste.co.uk/ghlala/SQL.zip*

IMPORTANT NOTES.– This query displays 295 lines. This number does not correspond to the number of subcategories (the cardinality of the `ProductSubcategory` table is equal to 37). It represents the number of subcategories attached to the `Product` table products with possibly the number of subcategories that exist in the `ProductSubcategory` table and do not appear in the `Product` table.

Instead of using the RIGHT OUTER JOIN clause, you can keep the version of the query given in the previous section with the OUTER LEFT JOIN clause while switching the `Product` and `ProductSubcategory` tables.

3.1.5.3. *Full outer join*

FULL OUTER JOIN displays a complete dashboard in which there is an exhaustive list of all the lines of table T1 with possibly additional information from table T2 and vice versa.

To illustrate FULL OUTER JOIN, see the query: "Display the names of the products with eventually the names of their subcategories as well as the names of the subcategories with eventually the names of the products which are associated with them".

```
USE AdventureWorks2014;
GO
SELECT P.Name AS Product,
 SC.Name AS SubCategory
FROM Production.Product AS P
FULL OUTER JOIN Production.ProductSubcategory AS SC
ON (P.ProductSubcategoryID = SC.ProductSubcategoryID);
GO
```

Box 3.10. *Full outer join. For a color version of this box, see www.iste.co.uk/ghlala/SQL.zip*

IMPORTANT NOTE.– Although this feature is interesting in the decision-making world, it should be used with caution

because the interpretation of the generated results is not obvious.

3.1.6. *The Cartesian product*

The Cartesian product is an operation that consists of generating all possible combinations by crossing the rows of table T1 with the rows of table T2. This crossing does not require common columns between the two tables and can even generate completely incorrect results.

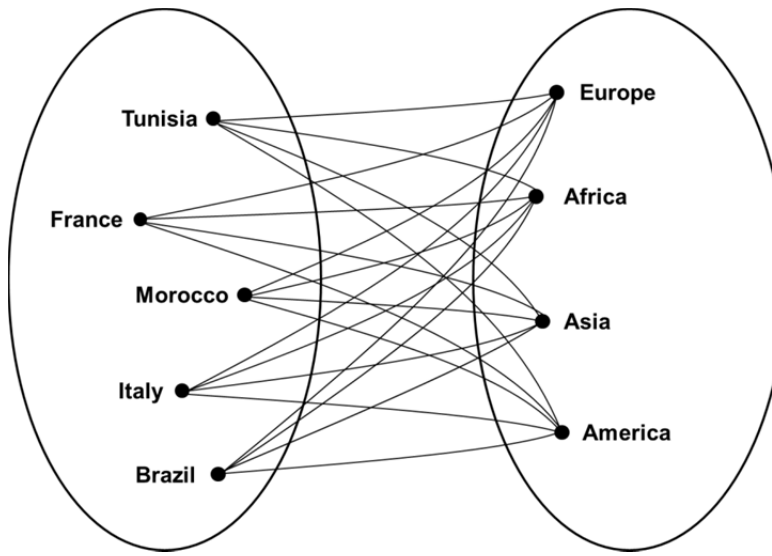


Figure 3.2. *Cartesian product of countries with continents*

This Cartesian product can be made in two ways:

1) by specifying the two tables to be crossed in the FROM clause (not recommended);

2) using the CROSS JOIN operator (recommended).

```
USE AdventureWorks2014
SELECT P.Name AS Product,
 SC.Name AS SubCategory
FROM Production.Product AS P,
 Production.ProductSubcategory AS SC
```

Box 3.11. *Cartesian product (method not recommended). For a color version of this box, see www.iste.co.uk/ghlala/SQL.zip*

```
USE AdventureWorks2014
SELECT P.Name AS Product,
 SC.Name AS SubCategory
FROM Production.Product AS P
CROSS JOIN Production.ProductSubcategory AS SC
```

Box 3.12. *Cartesian product (recommended method). For a color version of this box, see www.iste.co.uk/ghlala/SQL.zip*

3.1.7. *Lateral join*

A lateral join consists of generating the results of correspondence between a table and the dataset returned by a function. This is a special category of functions called TVF (table-valued functions). The operator used to make this join is the APPLY operator.

```
USE AdventureWorks2014
/*Function that returns the list of products by
specifying their colours and a price threshold. */
CREATE FUNCTION UF_PRD(@sc INT)
TABLE RETURNS
AS
RETURN SELECT *
        FROM[Production].[Product] AS P
        WHERE P.ProductSubcategoryID = @sc
        AND P.ListPrice > 1000
GO
```

Box 3.13. *Table-valued function (TVF). For a color version of this box, see www.iste.co.uk/ghlala/SQL.zip*

```
SELECT P.Name AS Product,
 F.Name AS SubCategory
FROM Production.ProductSubcategory AS SC
CROSS APPLY UF_PRD (sc.ProductSubcategoryID) AS F
GO
```

Box 3.14. *Lateral join (CROSS APPLY). For a color version of this box, see www.iste.co.uk/ghlala/SQL.zip*

```
SELECT P.Name AS Product,
 F.Name AS SubCategory
FROM Production.ProductSubcategory AS SC
CROSS APPLY UF_PRD (sc.ProductSubcategoryID) AS F
GO
```

Box 3.15. *Lateral join (OUTER APPLY). For a color version of this box, see www.iste.co.uk/ghlala/SQL.zip*

IMPORTANT NOTES.– The usefulness of the APPLY operator arises especially if the actual parameters used for the function returning a table (TVF) located to the right of the operator are columns of the table used to the left of this operator.

The APPLY operator is used with the CROSS option to simulate an inner join operation (INNER JOIN) or with the OUTER option to simulate a left external join operation (OUTER LEFT JOIN).

3.1.8. *Conclusion*

Joining is a key operation in data analysis. This importance is due to the relational context that requires rigid data normalization. Currently, we are seeing radical changes in data modeling. In this new landscape, the joining operation gives way to mechanisms more adapted to the analytical context, such as the principle of Embedding, which consists of incorporating data from one collection into another. This principle is applied in NoSQL databases, mainly document-oriented databases.

3.2. Set operators

3.2.1. *Introduction*

Data interrogation with SQL language is performed by a set of operations formalized by E. F. Codd (1970) in the form of an algebraic language. The operators of this language can be classified into two main categories: relational operators and set operators.

The relational operations specific to SQL language are as follows:

– selection (restriction): this consists of specifying the lines to be processed. This operation is performed by the WHERE clause of the SQL query.

– projection: this consists of specifying the columns to be displayed among the list of columns of the referenced table(s).

– joining: this consists of processing a dataset from several tables.

Figure 3.3. *Relational and set operators of SQL language*

The set operations, which are UNION, INTERSECT, EXCEPT, CROSS JOIN and division, allow tables to be treated as sets. The operators used, although not standardized, exist in most database management systems under different names.

3.2.2. *Mind map of the second section*

This second section will be devoted to studying the UNION, INTERSECT and EXCEPT operators, since the CROSS JOIN operator was already discussed in the previous section and the division operator is not implemented in all DBMSs including Microsoft SQL Server.

Figure 3.4 presents the mind map of the different concepts that will be covered in this section.

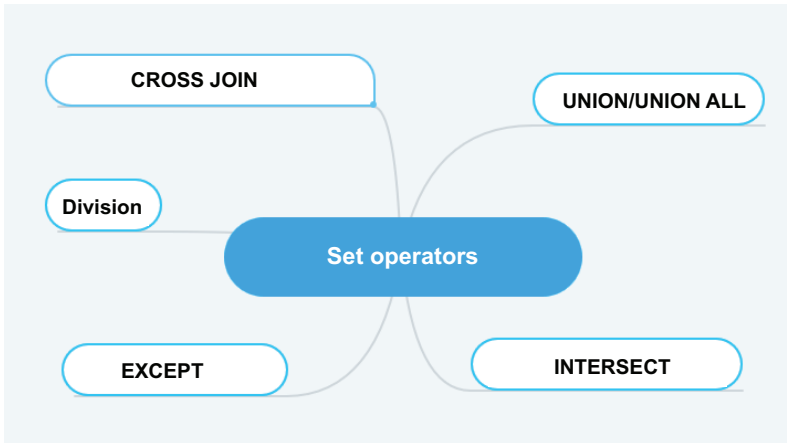


Figure 3.4. *Mind map of the second section*

3.2.3. *The UNION set operator*

This is a set operator that returns together the resultant lines from the dataset of the first query and the resultant lines from the dataset of the second query.

SQL Server has two variants of this operator: UNION and UNION ALL. The first variant displays all lines including duplicates, while the second variant displays all lines while eliminating duplicates.

```
-- Display of all employees and salespeople.
SELECT P.FirstName, P.MiddleName,P.LastName
FROM HumanResources.Employee AS E
JOIN Person.Person AS P
ON (E.BusinessEntityID = P.BusinessEntityID)
-- The first request returns 290 employees.
UNION ALL
SELECT P.FirstName, P.MiddleName, P.LastName
FROM Sales.SalesPerson AS SP
JOIN Person.Person AS P
ON (SP.BusinessEntityID = P.BusinessEntityID)
-- The second request returns 17 salespeople.
GO
-- The union with duplicates returns 307 lines
-- (employees and salespeople).
```

Box 3.16. *UNION ALL set operator. For a color version of this box, see www.iste.co.uk/ghlala/SQL.zip*

```
-- Display of all employees and salespeople.
SELECT P.FirstName, P.MiddleName, P.LastName
FROM HumanResources.Employee AS E
JOIN Person.Person AS P
ON (E.BusinessEntityID = P.BusinessEntityID)
-- The first request returns 290 employees.
UNION
SELECT P.FirstName, P.MiddleName, P.LastName
FROM Sales.SalesPerson AS SP
JOIN Person.Person AS P
ON (SP.BusinessEntityID = P.BusinessEntityID)
-- The second request returns 17 salespeople.
GO
-- The union without duplicates returns 290 employees
-- because the salespeople are already employees.
```

Box 3.17. *UNION set operator. For a color version of this box, see www.iste.co.uk/ghlala/SQL.zip*

IMPORTANT NOTES.– The UNION ALL operator is more efficient than the UNION operator. The latter executes additional instructions to eliminate duplicates.

The union of two tables does not require the two tables to have perfectly identical structures. It is sufficient to have similar columns between the two tables.

Similarity between two columns is based on the type of data in the content of these columns and not their names. Two columns of the same data type are considered similar by the UNION operator even though they do not have the same name and their contents are semantically different.

3.2.4. *INTERSECT set operator*

This is a set operator that returns only those lines that appear in the dataset of both the first query and the second query.

```
-- Display of all sales employees.
SELECT P.FirstName, P.MiddleName, P.LastName
FROM HumanResource.Employee AS E
JOIN Person.Person AS P
ON (E. BusinessEntityID = P.BusinessEntityID)
-- The first query returns 290 employees.
UNION
SELECT P.FirstName, P.MiddleName, P.LastName
FROM Sales.SalesPerson AS SP
JOIN Person.Person AS P
ON (SP.BusinessEntityID = P.BusinessEntityID)
-- The second query returns 17 sellers.
GO
-- The intersection returns 17 sales employees
-- because all salespeople are already employees.
```

Box 3.18. *INTERSECT set operator. For a color version of this box, see www.iste.co.uk/ghlala/SQL.zip*

IMPORTANT NOTES.– The UNION operator with its two variants and the INTERSECT operator check the commutativity characteristic. This feature means that the order of the two queries does not count; it only intervenes in the choice of the names of the columns to be displayed (those of the first query) if they do not already have the same names.

3.2.5. *EXCEPT set operator*

This is a set operator that returns only those lines that appear in the dataset of the first request and not in the dataset of the second request.

```
-- Display non-sales employees.
SELECT P.FirstName, P.MiddleName, P.LastName
FROM HumanResources.Employee AS E
JOIN Person.Person AS P
ON (E. BusinessEntityID = P.BusinessEntityID)
-- The first query returns 290 employees.
EXCEPT
SELECT P.FirstName, P.MiddleName, P.LastName
FROM Sales.SalesPerson AS SP
JOIN Person.Person AS P
ON (SP.BusinessEntityID = P.BusinessEntityID)
-- The second query returns 17 salespeople.
GO
-- The difference returns 273 employees.
```

Box 3.19. *EXCEPT set operator. For a color version of this box, see www.iste.co.uk/ghlala/SQL.zip*

IMPORTANT NOTES.– The EXCEPT operator does not check the commutativity characteristic. The order of the two queries is important because the permutation gives different results.

3.2.6. *The division set operator*

This is a set operator that returns only those rows of a dataset that are related to all the rows of another dataset. This data query requires an intermediate dataset linking the two. To divide customers by products, i.e. to search for customers who have ordered all products, three sets of data are required:

– a dataset for customers;

– a dataset for the products;

– an intermediate dataset representing the links between customers and products in the form of order details (order lines).

```
-- Display of customers who have ordered all the
-- products (division of the order table
-- Sales.SalesOrderHeader' by the product table
--'Production.Product').
SELECT SO.CustomerID
FROM Sales.SalesOrderHeader AS SO
WHERE NOT EXISTS
 (
 SELECT SD.ProductID
 FROM Sales.SalesOrderDetail SD
 WHERE NOT EXISTS
 (
 SELECT P.ProductID
 FROM Production.Product as P
 WHERE P.ProductID = sd.ProductID
 AND SO.SalesOrderID = SD.SalesOrderID
 )
 )
GO
```

Box 3.20. *Simulation of the division set operator. For a color version of this box, see www.iste.co.uk/ghlala/SQL.zip*

3.2.7. *Conclusion*

Modeling the database with the relational model means that the tables are mathematically assimilated to relationships or sets. Operators such as UNION, INTERSECT and EXCEPT seem very useful in applying a set logic to these tables.

3.3. Pivoting operators

3.3.1. *Introduction*

The data managed by the information system is generally divided into two subsystems:

– OLTP: a transactional system in which data is stored in relational databases formed by tables and requested by the SQL language;

– OLAP: a decision-making system in which data is stored in multidimensional databases formed by cubes and requested by the MDX or DAX language.

Pivoting operators, also known as Rotation Operators, are new features integrated into the relational model. These operators are inspired by the multidimensional logic in which operations such as PIVOT, DRILL DOWN, DRILL UP, DRILL THROUGH, SLICE and DICE are applicable to cubes.

The advantage of rotating data is to have a view of the data from different angles that reflect the company's different interests and concerns.

3.3.2. *Mind map of the third section*

This third section will be devoted to presenting the operators for transposing data between the rows and columns of a table to simulate the pivot operator applicable to multidimensional data stored in cubes.

Figure 3.5 presents the mind map of the different concepts that will be covered in this section.

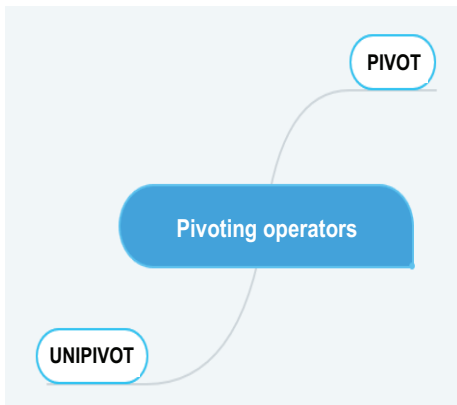


Figure 3.5. *Mind map of the third section*

3.3.3. *The PIVOT operator*

The PIVOT operator allows you to transpose the display of the dataset by transforming the rows of this dataset into columns.

```
USE AdventureWorks2014
GO
-- Display the order amounts per year in a
-- transposed form (the amounts in lines and the
-- years in columns).
SELECT 'Total Amount',[2011],[2012],[2013],[2014]
FROM (SELECT TotalDue, Year(OrderDate) AS ORDER_YEAR
 FROM Sales.SalesOrderHeader AS SOURCE_TABLE
 ) AS TEMP_TABLE
PIVOT (SUM(TotalDue)
 FOR ORDER_YEAR IN ([2011],[2012],[2013],[2014])
 ) AS PIVOT_TABLE
/*---------------------------------
 VERSION with CTE
---------------------------------- */
WITH TEMP_ABLE AS
 (
 SELECT TotalDue, Year(OrderDate) AS ORDER_YEAR
 FROM Sales.SalesOrderHeader AS SOURCE_TABLE
 )
SELECT 'Total Amount',[2011],[2012],[2013],[2014]
FROM TEMP_ABLE
PIVOT (
 SUM(TotalDue)
 FOR ORDER_YEAR IN ([2011],[2012],[2013],[2014])
 ) AS PIVOT_TABLE
GO
/*---------------------------------
 VERSION with view creation
---------------------------------- */
CREATE VIEW Total_Amount_Year as
WITH TEMP_ABLE AS
 (SELECT TotalDue, Year(OrderDate) AS ORDER_YEAR
 FROM Sales.SalesOrderHeader AS SOURCE_TABLE
 )
SELECT 'Total Amount' AS TM,[2011],[2012],[2013],
 [2014]
FROM TEMP_ABLE
PIVOT (
 SUM(TotalDue)
 FOR ORDER_YEAR IN ([2011],[2012],[2013],[2014])
 ) AS PIVOT_TABLE
GO
```

Box 3.21. *PIVOT operator. For a color version of this box, see www.iste.co.uk/ghlala/SQL.zip*

IMPORTANT NOTE.– The values for years must be entered between two brackets [].

Creating a view requires aliases for all its columns.

A problem with the PIVOT operator is specifying all values in the IN selection if you want to see them in columns. One solution to overcome this problem is to create a dynamic IN selection using the Dynamic SQL mechanism.

```
USE AdventureWorks2014
DECLARE @request VARCHAR(4000)
DECLARE @years VARCHAR(2000)
SELECT @years =
 STUFF((SELECT DISTINCT '],['+LTRIM(Year(OrderDate))
 FROM Sales.SalesOrderHeader
 ORDER BY '],['+LTRIM(Year(OrderDate))
 FOR XML PATH('')),1,2,'') + ']'
SET @query =
"SELECT *
FROM (
 SELECT TotalDue, Year(OrderDate) AS ORDER_YEAR
 FROM Sales.SalesOrderHeader AS SOURCE_TABLE
 ) AS TEMP_ABLE
PIVOT ( SUM(TotalDue)
 FOR ORDER_YEAR IN ('+ @years +')
 ) AS PIVOT_TABLE'
EXECUTE (@query)
GO
```

Box 3.22. *Query using the PIVOT operator in a dynamic manner. For a color version of this box, see www.iste.co.uk/ghlala/SQL.zip*

3.3.4. *The UNPIVOT operator*

The UNPIVOT operator allows you to transpose the display of the dataset by transforming the columns of this dataset into rows.

```
USE AdventureWorks2014
GO
SELECT Year, Amount
FROM Total_Amount_Year AS SOURCE_TABLE
UNPIVOT (
 Amount for Year IN ([2011],[2012],[2013],
 [2014]
 )
 ) AS UNPIVOT_TABLE
GO
```

Box 3.23. *UNPIVOT operator. For a color version of this box, see www.iste.co.uk/ghlala/SQL.zip*

IMPORTANT NOTE.– The UNPIVOT operator should not under any circumstances be interpreted as an operation to cancel the work performed by the PIVOT operator. It is a stand-alone operation that acts on the data from a specified source by transposing the columns into rows.

3.3.5. *Conclusion*

Pivoting operators are used to simulate multidimensional logic functionalities in a relational database. They are useful for information systems that do not plan to invest in multidimensional databases, but to rely on relational databases for the analysis of their data.

4

Functions

4.1. Predefined functions

4.1.1. *Introduction*

The SQL language is a stable language that has not had any major changes in its kernel (its commands) since its invention. This does not mean that SQL is a stagnant language, since it has had multiple enhancements through the integration of new functions capable of being used with SQL commands, mainly the SELECT command.

The dynamism of SQL is also due to the possibility of creating user functions using Transact SQL (TSQL). This procedural add-on to almost all DBMSs is a procedural programming environment designed primarily for server-side development to ensure greater integrity, optimization and security when accessing the database.

Functions with their two variants (predefined and user) play an important role in data analysis.

Analytic SQL in SQL Server 2014/2016, First Edition. Riadh Ghlala.

4.1.2. *Mind map of the first section*

This first section will be devoted to presenting and classifying the predefined functions of SQL Server, as well as introducing the user-defined functions.

Figure 4.1 presents the mind map of the different concepts that will be covered in this section.

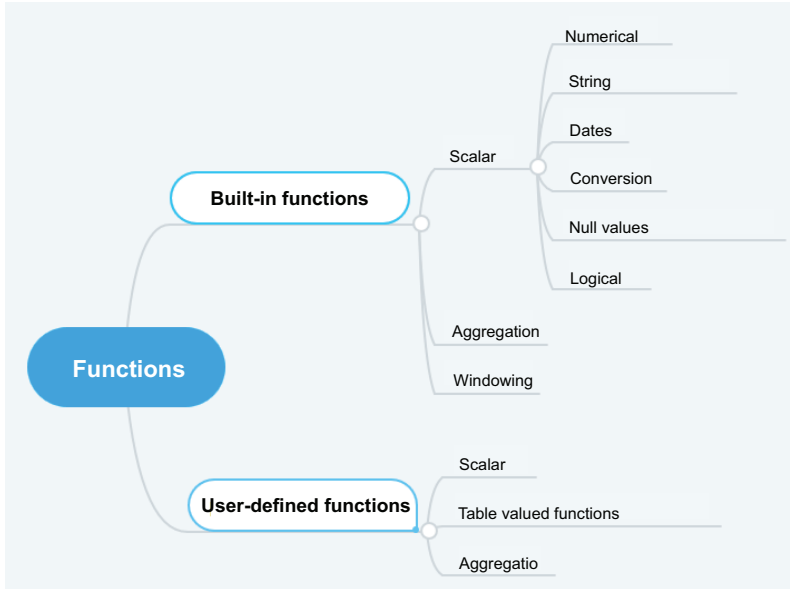


Figure 4.1. *Mind map of the first section*

4.1.3. *Scalar built-in functions*

SQL Server has a very rich set of predefined functions referred to as built-in functions. These functions can be classified into different categories, mainly scalar functions, aggregation functions and windowing functions.

This section is devoted to presenting scalar functions, since the other two categories will be detailed in the

following sections (section 4.2 for aggregation functions and section 4.3 for windowing functions).

Scalar functions, also called single-row functions, are those that work on each line of the processed dataset and return one output per line. Within this category, another classification exists which distinguishes between numerical scalar functions, string processing, processing of dates, conversion, null value processing and logical functions.

In this section, the idea is to present the classification of the different functions with illustrative examples, rather than an exhaustive inventory of all functions.

The complete list of functions can be viewed in the SQL Server documentation or in the SSMS client by following these steps:

1) Open SQL Server Management Studio and connect to the target SQL Server instance.

2) Click on the Database node, choose any database (e.g. *"AdventureWorks2014"*), select Programmability, expand Functions then System Functions.

3) Expand the desired category to see the list of associated functions.

4.1.3.1. *Scalar numerical functions*

Functions operating on numbers accept numerical values as parameters and return numerical values.

Numerical scalar functions can be classified as follows:

– scientific functions;

– trigonometric functions;

– rounding functions;

– random number generation functions.

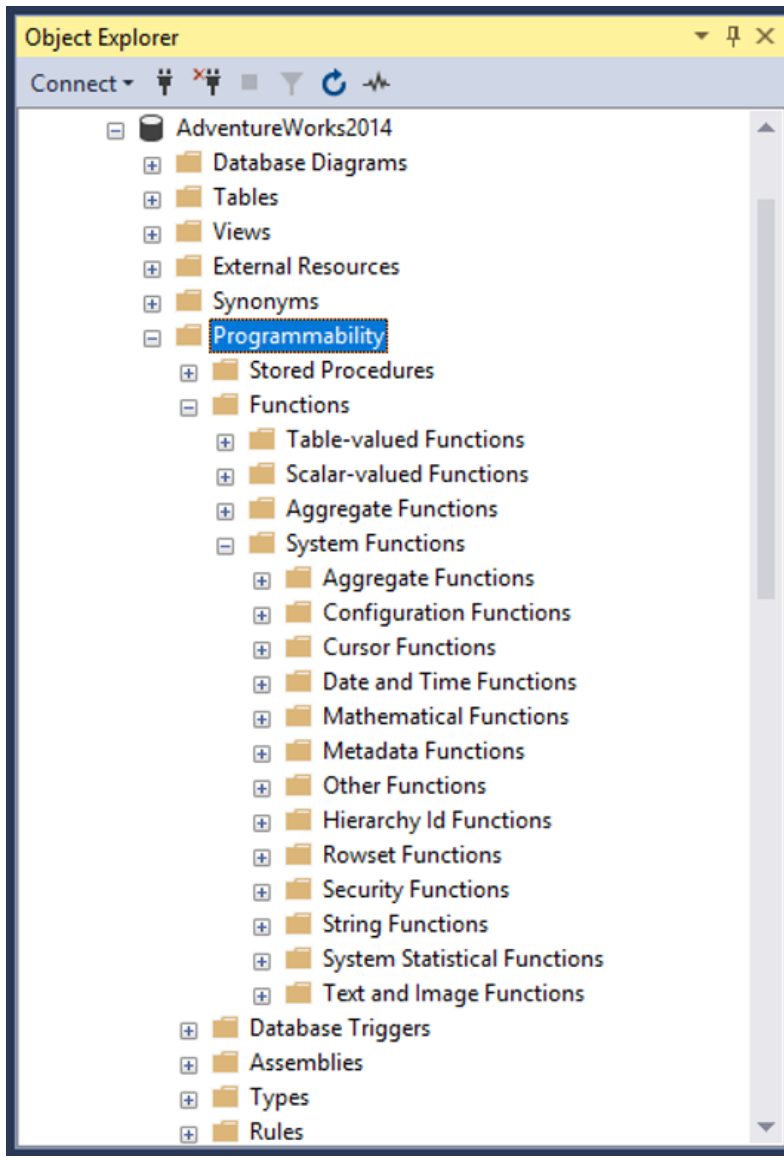


Figure 4.2. *Built-in SQL Server functions*

```
USE AdventureWorks2014;
GO
/*------------------------- *
 * Scientific functions *
 *------------------------- */
SELECT PI() -- Pi
GO
SELECT SQRT(25) -- Square root
GO
SELECT SQUARE(5) -- Square
GO
SELECT POWER(5,3) -- Power
GO
SELECT EXP(5) -- Exponential
GO
SELECT LOG(5) -- Natural logarithm (ln)
GO
SELECT LOG10(5) - Base 10 logarithm
GO
SELECT ABS(-5) -- Absolute value
GO
SELECT SIGN(-5) -- Sign
GO
/*--------------------------- *
 * Trigonometric functions *
 *--------------------------- */
SELECT SIN(1) -- Sinus
GO
SELECT COS(1) -- Cosine
GO
SELECT TAN(1) -- Tangent
GO
/* ----------------------------*
 * Rounding functions *
 *--------------------------- */
SELECT ROUND(5.236,1) -- Single-digit rounding
GO
SELECT FLOOR(5.236) -- Lower integer
GO
SELECT CEILING(5.236) -- Upper integer
GO
/* ----------------------------------*
 * Random number generator *
 *---------------------------------- */
SELECT RAND() - Real number between 0 and 1
GO
SELECT RAND()*10 - Real number between 0 and 10
GO
SELECT FLOOR(RAND()*10) -- Integer between 0 and 10
GO
```

Box 4.1. *Numerical scalar functions. For a color version of this box, see www.iste.co.uk/ghlala/SQL.zip*

4.1.3.2. *Scalar string functions*

Functions operating on character strings accept character strings as parameters and return either a numerical value or a character string.

The scalar functions for processing character strings can be classified as follows:

– functions returning the position of one string within another;

– transformation functions;

– comparison functions;

– character encoding functions.

```
USE AdventureWorks2014;
GO
/*----------------------- */
/* Position functions */
/*----------------------- */
-- String position
SELECT CHARINDEX('in', 'Min') AS RESULT
GO
-- Position of the pattern
SELECT PATINDEX('%in%', 'Min') AS RESULT
GO
-- Length of a string
SELECT LEN('Minimum') AS RESULT
GO
/*---------------------------- */
 * Transformation functions *
 *---------------------------- */
-- Names displayed in upper case
SELECT UPPER(Name) AS NOM
FROM [Production].[Product]
GO
-- Names displayed in lower case letters
SELECT LOWER(Name) AS NOM
FROM [Production].[Product]
GO
-- Extraction of a 3 character sub-string on the left
SELECT LEFT('Analytic SQL',3) AS RESULT
GO
-- Extraction of a 10 character sub-string
-- on the right
SELECT RIGHT('Analytic SQL',10) AS RESULT
GO
-- Extraction of a 10 character sub-string
```

```
-- starting from position 5
SELECT SUBSTRING('Analytic SQL',5,10) AS RESULT
GO
-- Elimination of left-hand spaces
SELECT LTRIM('Analytic SQL') AS RESULT
GO
-- Elimination of right-hand spaces
SELECT RTRIM('Analytic SQL') AS RESULT
GO
-- Elimination of spaces at both ends
SELECT RTRIM(LTRIM('Analytic SQL')) AS RESULT
GO
-- Replacing one sub-string with another
SELECT REPLACE('Analytic SQL', 'Analytic',
 'Tuning') AS RESULT
GO
/*-------------------------- *
 * Comparison functions *
 *-------------------------- */
-- Code returned by a phonetic indexing
-- algorithm of names by pronunciation in
-- British English (SOUNDEX)
SELECT SOUNDEX('Analytic') AS RESULT
GO
SELECT SOUNDEX('Tuning') AS RESULT
GO
-- Comparison of two strings of characters based on
-- the SOUNDEX algorithm and return of a value
-- between 1 and 4 reflecting their similarity
SELECT DIFFERENCE('Analytic', 'Tuning') AS RESULT
GO
/*---------------------------------- *
 * Character encoding functions *
 *---------------------------------- */
-- ASCII code of a character
SELECT ASCII('A') AS RESULT
GO
-- UNICODE code of a character (N for National)
SELECT UNICODE(N'ض') AS RESULT
GO
-- Character related to an ASCII code (from 0 to 255)
SELECT CHAR(97) AS RESULT
GO
-- Character related to a UNICODE code
-- (from 0 to 65535)
SELECT NCHAR(1590) AS RESULT
GO
```

Box 4.2. *Scalar string functions. For a color version of this box, see www.iste.co.uk/ghlala/SQL.zip*

4.1.3.3. *Scalar date functions*

Functions operating on dates accept dates as parameters and return either dates or numerical values.

The scalar functions for processing dates can be classified as follows:

– functions that return the current date and time with low and high precision;

– functions returning part of the date and time;

– validation and format management functions;

– other functions such as comparing and changing a date.

```
USE AdventureWorks2014;
GO
/*------------------------- */
/* Current date and time */
/*------------------------- */
-- Current date (Low precision)
SELECT GETDATE() AS RESULT
GO
-- Current date and time (Low precision)
SELECT CURRENT_TIMESTAMP AS RESULT
GO
-- Current date and time (Low precision) in Coordinated
Universal Time (UTC)
SELECT GETUTCDATE()AS RESULT
GO
-- Current date and time (High precision)
SELECT SYSDATETIME()AS RESULT
GO
-- Current date and time (High precision) in Coordinated
Universal Time (UTC)
SELECT SYSUTCDATETIME()AS RESULT
GO
-- Current date and time (High precision) with
-- time zone
SELECT SYSDATETIMEOFFSET() AS RESULT
GO
/*---------------------------------- */
 * Validation functions and format *
 *---------------------------------- */
-- Verification of the validity of a date
SELECT ISDATE('02/29/2019') AS RESULT
GO
-- View the date format
DBCC USEROPTIONS;
GO
```

```
-- Change the date format (Day/Month/Year)
-- Valid values are mdy, dmy, ymd, ydm, ydm, myd
-- and dym
SET DATEFORMAT dmy;
GO
/*------------------------------------------------- *
 * Functions returning a part of a date or time *
 *------------------------------------------------- */
-- Display part of the date. Possible
-- values are: year, quarter, month, dayofyear,
-- day, week, weekday
-- DATENAME returns a string of characters
-- DATEPART returns an integer
SELECT DATENAME(day, GETDATE()) AS RESULT;
SELECT DATEPART(day, GETDATE()) AS RESULT;
GO
-- Display part of the time. Possible
-- values are: hour, minute, second, millisecond,
-- microsecond, nanosecond
SELECT DATENAME(minute, sysdatetime()) AS RESULT
SELECT DATEPART(minute, sysdatetime()) AS RESULT
GO
-- Display the day of a date
SELECT DAY(GETDATE()) AS RESULT
GO
-- Display the month of a date
SELECT MONTH(GETDATE()) AS RESULT
GO
-- Display the year of a date
SELECT YEAR(GETDATE()) AS RESULT
GO
/*------------------ *
 * Other functions *
 *------------------ */
-- Difference between two dates based on one of them
SELECT DATEDIFF(day,'01/01/2000', GETDATE()) AS RESULT
SELECT DATEDIFF_BIG (nanosecond,'01/01/2000',
 GETDATE()) AS RESULT
GO
-- Formation of a date from its parts
SELECT DATEFROMPARTS (2018, 12, 31) AS RESULT;
GO
-- Modification of a date
SELECT DATEADD (day, 10, GETDATE()) AS RESULT;
SELECT DATEADD (month, -4, GETDATE()) AS RESULT;
GO
```

Box 4.3. *Scalar date functions. For a color version of this box, see www.iste.co.uk/ghlala/SQL.zip*

4.1.3.4. *Scalar conversion functions*

Data processed by functions always has a specific data type. The need to process this data in a format other than its own explains the need for conversion functions. Although SQL Server can do implicit conversions in many cases, conversion functions are required to make this conversion explicit. These functions accept a data in a particular type as a parameter and return the same data in the specified type.

The conversion functions can be classified as follows:

– the CAST standard function;

– Microsoft SQL Server-specific functions CONVERT and PARSE;

– Microsoft SQL Server-specific functions with exception handling TRY_CONVERT and TRY_PARSE.


```
USE AdventureWorks2014;
GO
/*---------------------------------------- *
 * Standard conversion function (CAST) *
 *---------------------------------------- */
SELECT CAST(25.65 AS int);
SELECT CAST('25.65' AS NUMERIC(8,2));
SELECT CAST('2019-01-10' AS datetime);
SELECT CAST(GETDATE() AS VARCHAR);
/*---------------------------------------- *
 * Specific conversion functions *
 * (CONVERT and PARSE) *
 *---------------------------------------- */
SELECT CONVERT(INT, 25.65);
SELECT CONVERT(datetime,'2019-01-10');
SELECT CONVERT(VARCHAR, GETDATE());
SELECT CONVERT(VARCHAR, GETDATE(), 110)
SELECT PARSE('17.25' AS NUMERIC(8,2))
SELECT PARSE('12-31-2018' as DATETIME)
SELECT PARSE('2018/12/31' as DATETIME)
SELECT PARSE('31-12-2018' as DATETIME using 'fr-FR')
GO
/*------------------------------------------------ *
 * Specific conversion functions with exception *
 * management (TRY_CONVERT and TRY_PARSE) *
 *------------------------------------------------ */
SELECT TRY_CONVERT(INT,'B3')
SELECT TRY_PARSE('2018/2/30' as DATETIME)
GO
```

Box 4.4. *Scalar conversion functions. For a color version of this box, see www.iste.co.uk/ghlala/SQL.zip*

4.1.3.5. *Scalar functions for processing null values*

Relational databases are very rigid in terms of data schema. They require that all rows of the same table have the same data schema, i.e. the same structure. To ensure this constraint, the relational model introduced the notion of the null value (NULL) to replace the gap or void, which means the absence of any value for an attribute in a line. The possible operators to use with a NULL to check whether a value is null or not are IS and IS NOT. Another specificity related to this NULL value is the fact that it can distort a calculation if it is part of its expression.

```
USE AdventureWorks2014;
GO
-- Display product names and their weights with
-- the value of shipping costs calculated on the basis
-- of 1.25 euro for each kilo or part of a kilo.
 SELECT P.Name,P.Weight,
 1.25*CEILING(P.Weight) AS Frais
 FROM [Production].[Product] AS P
GO
```

Box 4.5. *Calculation errors due to null values. For a color version of this box, see www.iste.co.uk/ghlala/SQL.zip*

This problem is solved by the functions for processing null values which can be classified as follows:

– the standard function COALESCE() can take several parameters and returns the first parameter NOT NULL from the parameter list;

– the Microsoft SQL Server-specific function ISNULL(), which only takes two parameters. It returns:

- the first parameter if its value is not NULL,

- the second parameter if the first parameter is NULL.

```
USE AdventureWorks2014;
GO
-- ISNULL function
 SELECT P.Name,P.Weight,
 1.25*CEILING(ISNULL(P.Weight,1)) AS Frais
 FROM [Production].[Product] AS P
-- COALESCE function
 SELECT P.Name,P.Weight,
 1.25*CEILING(COALESCE(P.Weight,1)) AS Frais
 FROM [Production].[Product] AS P
GO
```

Box 4.6. *Scalar functions for processing null values with string data. For a color version of this box, see www.iste.co.uk/ghlala/SQL.zip*

IMPORTANT NOTE.– The functions for processing null values cover all types of data and are not reserved only for numerical values.

```
USE AdventureWorks2014;
GO
-- Display product names with the first not null
-- characteristic among the list of
-- requested characteristics
 SELECT P.Name, COALESCE('Size: ' + P.Size,
 'Size : ' + P.Style,
 'Class : ' + P.Class,
 'Color : ' + P.Color,
 'None') AS Characteristic
 FROM [Production].[Product] AS P
GO
```

Box 4.7. *Scalar functions for processing null values with string data. For a color version of this box, see www.iste.co.uk/ghlala/SQL.zip*

4.1.3.6. *Logical scalar functions*

Logical scalar functions are functions to simulate algorithmic reasoning within the SQL query. These functions provide conditional processing without using the IF… ELSE command of the TSQL procedural extension.

The logical scalar functions can be classified as follows:

– the standard function CASE() which evaluates a list of conditions and returns one of the possible result expressions;

– the Microsoft SQL Server-specific function IIF() which returns one of the two values provided as parameters, depending on whether the Boolean expression has a true or false value;

– the Microsoft SQL Server-specific function CHOOSE() which returns the element at the specified index from a list of values provided as parameters;

– the Microsoft SQL Server-specific function NULLIF() which returns a NULL value if the two specified expressions are equal, otherwise it returns the first expression.


```
USE AdventureWorks2014;
GO
-- CASE function with discrete values
SELECT P.ProductNumber, P.Name,
 CASE P.ProductLine
 WHEN 'R' THEN 'Route'
 WHEN 'M' THEN 'Mountain'
 WHEN 'T' THEN 'Tourism'
 WHEN 'S' THEN 'Misc'
 ELSE 'Not yet for sale'
 END AS Category
FROM Production.Product AS P
-- CASE function with expressions
SELECT P.ProductNumber, P.Name,
 CASE
 WHEN ListPrice = 0 THEN 'Not yet for sale'
 WHEN ListPrice < 100
 THEN 'Sale products'
 WHEN ListPrice >= 100 and ListPrice < 500
 THEN 'Mainstream products'
 WHEN ListPrice >= 500 and ListPrice < 1000
 THEN 'New collections'
 ELSE 'Luxury products'
 END AS "Product nature"
FROM Production.Product AS P
GO
-- IIF function
SELECT P.Name, IIF(P.ListPrice = 0,
 'Product for sale',
```

```
 'Product not yet for sale')
FROM [Production].[Product] AS P
-- NULLIF function
SELECT ProductID, MakeFlag, FinishedGoodsFlag,
 NULLIF(MakeFlag,FinishedGoodsFlag)
FROM Production.Product
-- CHOOSE function
DECLARE @c INT = 2;
SELECT P.Name, CHOOSE (@c, P.Color, P.Style, P.Size,
 P.Class) AS Caractérisitique_Optée
FROM [Production].[Product] AS P
GO
```

Box 4.8. *Logical scalar functions. For a color version of this box, see www.iste.co.uk/ghlala/SQL.zip*

4.1.4. *User functions*

TSQL is a procedural language developed by SQL Server to allow users to combine SQL queries with writing code with procedural logic. User-Defined Functions (UDF) are among the means provided by this TSQL language to enable the user to develop their own integrity and/or data analysis needs.

4.1.4.1. *Scalar UDF*

Modularity in procedural programming consists of subdividing the overall problem into sub-modules called procedures and functions. Procedures are generally modules that act upon the environment by executing actions, while functions are modules whose role is to perform a calculation and return the result.


```
USE AdventureWorks2014;
GO
/*Function that returns the price of a product by
specifying its ID. */
CREATE FUNCTION UF_PRIX_PRD(@ID INT)
FLOAT RETURNS
AS
BEGIN
        DECLARES @Result Float
        SELECT @Result = P.ListPrice
        FROM [Production].[Product] AS P
```

```
        WHERE P.ProductID = @ID
        RETURN @Result
END
```

Box 4.9. *Scalar UDF. For a color version of this box, see www.iste.co.uk/ghlala/SQL.zip*

4.1.4.2. *User TVF*

Functions returning a table are referred to as Table-Valued Functions (TVF). These are functions whose return type is the TABLE type, by analogy with the relational object TABLE, which allows the storage of an indeterminate number of data lines.

The TABLE type of the procedural extension of the TSQL is a type that allows the storage of a data collection. It is an appropriate solution for functions that return multiple results.

```
USE AdventureWorks2014;
GO
/*Function that returns the list of products by
specifying their colours and a price threshold. */
CREATE FUNCTION UF_LIST_PRD(@P Float, @C
VARCHAR(20))
RETURNS TABLE
AS
RETURN SELECT *
        FROM [Production].[Product] AS P
        WHERE P.ListPrice > @P
        AND P.Color = @C
GO
```

Box 4.10. *User TVF. For a color version of this box, see www.iste.co.uk/ghlala/SQL.zip*

IMPORTANT NOTE.– Table-Valued Functions (TVF) have an extended variant which consists of customized processing (definition and manipulation) of the result of type TABLE to be returned.

The dataset returned by this type of function can be assimilated to the relational object TABLE and therefore used in the FROM clause of a query.

The TABLE type can also be used as a formal parameter of a function, from which this function can later be called with an effective parameter in the form of an undefined dataset.

4.1.4.3. *User-aggregate function*

SQL Server gives the user the ability to create their own aggregation functions (a concept discussed in the next section). This functionality is not provided in a native way with the TSQL language. It is necessary to use the managed code technology known as CLR (Common Language Runtime). This technology consists of creating, in the database, advanced procedural objects (procedures, functions, triggers and aggregations) using programming languages such as C# and VB.NET while benefiting from the functionalities offered by the .NET Framework.

4.1.5. *Conclusion*

The predefined (Built-in) and user (UDF) functions provide many possibilities for data queries. This flexibility further improves data analysis capacities with both the SQL query language and its procedural extension, TSQL.

4.2. Aggregation functions

4.2.1. *Introduction*

Aggregation functions, also referred to as group functions, were the first and only mechanism to be used to develop indicators from a database.

These functions, also called multiple line functions, as opposed to scalar functions, which are called single line functions, allow a result to be calculated for a set of lines.

These functions are useful for calculating the sum, minimum, maximum, number and average of quantifiable columns in a table.

4.2.2. *Mind map of the second section*

This second section will present the application of these functions to the global dataset or to subgroups of this dataset formed by criteria for grouping. This section also describes the filters associated with these aggregation functions and the options that developed them.

Figure 4.3 presents the mind map of the different concepts that will be covered in this section.

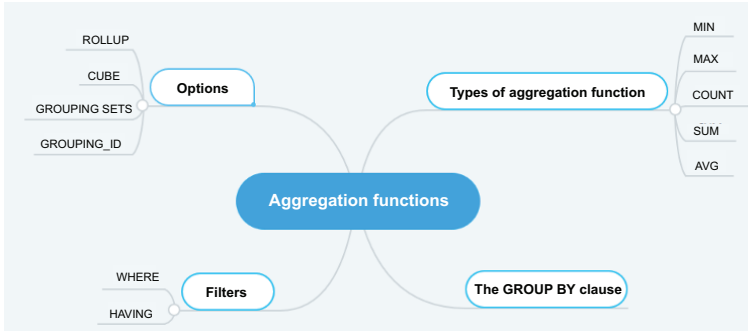


Figure 4.3. Mind map of the second section

4.2.3. *Common aggregation functions*

Table 4.1 lists the most commonly used aggregation functions in data analysis. These functions are standardized and are present in all database management systems.

Function	Description
MIN	Minimum
MAX	Maximum
SUM	Sum of
COUNT	Number of
AVG	Average

Table 4.1. *The most commonly used aggregation functions*

IMPORTANT NOTE.– Microsoft SQL Server offers specific aggregation functions such as COUNT_BIG, CHECKSUM_AGG, STDEV, STDEVP, STDEVP, VAR and VARP.

Microsoft SQL Server gives developers the ability to develop their own aggregation functions using the Common Language Runtime (CLR) technique.

```
USE AdventureWorks2014
GO
-- Standard aggregation functions.
SELECT MIN(P.ListPrice),
 Max(P.ListPrice),
 AVG(P.ListPrice)
FROM Production.Product AS P
GO
```

Box 4.11. *Aggregation functions. For a color version of this box, see www.iste.co.uk/ghlala/SQL.zip*

IMPORTANT NOTE.– It is not possible to display an indicator generated by an aggregation function simultaneously with other elementary data in the same query. The solution lies in using subqueries to answer this type of two-step question.

```
USE AdventureWorks2014
GO
-- Display names of products with the
-- maximum price.
SELECT P.Name, MAX(P.ListPrice)
FROM Production.Product AS P
GO
```

Box 4.12. *Incorrect use of aggregation functions with elementary data. For a color version of this box, see www.iste.co.uk/ghlala/SQL.zip*

```
USE AdventureWorks2014
GO
-- Display names of products with the
-- maximum price
SELECT P.Name
FROM Production.Product
WHERE ListPrice = (SELECT MAX(P.ListPrice)
 FROM Production.Product AS P)
GO
```

Box 4.13. *Permitted use of aggregation functions with elementary data. For a color version of this box, see www.iste.co.uk/ghlala/SQL.zip*

IMPORTANT NOTE.– Processing null values (NULL) in aggregation functions should be done with caution. Indeed, developing indicators based on a column containing null values can distort the results. The use of the functions for processing null values ISNULL() or COALESCE() is essential.

```
USE AdventureWorks2014
GO
-- Display the average weight of products
-- excluding those who have no value for
-- the weight characteristic.
SELECT AVG(P.Weight)
FROM Production.Product AS P
GO
-- Display of the average is 74.069219
```

Box 4.14. *Using aggregation functions with the exclusion of null values. For a color version of this box, see www.iste.co.uk/ghlala/SQL.zip*

```
USE AdventureWorks2014
GO
-- Display average product weight,
-- using value 0 for those which have no
-- value for the weight characteristic
SELECT AVG(ISNULLL(P.Weight,0))
FROM Production.Product AS P
GO
-- Display of the average is 30.127361
GO
```

Box 4.15. *Using aggregation functions including null values. For a color version of this box, see www.iste.co.uk/ghlala/SQL.zip*

4.2.4. *The GROUP BY clause*

The GROUP BY clause consists of subdividing the dataset to be processed by the query into several subgroups according to one or more criteria. Even though several criteria are used, it is still a single subdivision of the global dataset into several subgroups. The group function will then be applied to each subgroup that is formed.

```
USE AdventureWorks2014
GO
-- Display price indicators for
-- products according to colours.
SELECT P.Color,
 MIN(P.ListPrice),
 Max(P.ListPrice),
 AVG(P.ListPrice)
FROM Production.Product AS P
GROUP BY P.Color
GO
-- Display price indicators for
-- products according to colours, styles, sizes
-- and classes.
SELECT P.Color, P.Style, P.Size, P.Class,
 MIN(P.ListPrice),
 Max(P.ListPrice),
 AVG(P.ListPrice)
FROM Production.Product AS P
GROUP BY P.Color, P.Style, P.Size, P.Class
GO
```

Box 4.16. *GROUP BY clause. For a color version of this box, see www.iste.co.uk/ghlala/SQL.zip*

IMPORTANT NOTE.– The criteria used to subdivide the global dataset into subgroups can be displayed by the SELECT clause. Therefore, it does not involve elementary data relating to a particular line, but rather summary information relating to each subgroup.

4.2.5. *WHERE and HAVING filters*

When performing data analysis with aggregation functions, we may often have recourse to the notion of filters. The latter are defined by the two clauses WHERE and HAVING. The role of each in the query is very specific and confusion between the two is not tolerated.

Figure 4.4 shows the location and role of each of these two filters in the query.

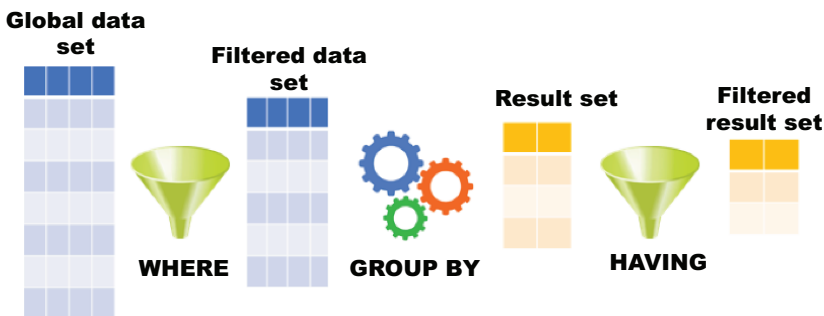


Figure 4.4. *Filters in a query using aggregation functions with the GROUP BY clause*

This figure shows that:

– the WHERE filter is always placed before the GROUP BY clause and its role is to filter the data to be processed (it is an upstream filter);

– the HAVING filter is always placed after the GROUP BY clause and its role is to filter the results obtained (it is a downstream filter).

```
USE AdventureWorks2014
GO
-- Display of price indicators for
-- products according to colour. A filter must
-- be used to eliminate input lines
-- with zero values for colour and
-- another filter must be used to remove
-- indicators < 100 from the output.
SELECT P.Color, Max(P.ListPrice) AS MAX_Price
FROM Production.Product AS P
WHERE P.Color IS NOT NULL -- Filter on inputs
GROUP BY P.Color
HAVING MAX(P.ListPrice) > 10 -- Filter on outputs
GO
```

Box 4.17. *GROUP BY clause with filters. For a color version of this box, see www.iste.co.uk/ghlala/SQL.zip*

IMPORTANT NOTES.– It is not possible to use aliases defined in the SELECT clause in the HAVING clause. We must repeat the entire aggregation function in the HAVING clause, simply because this clause is executed before the SELECT clause, when aliases are not yet defined.

4.2.6. *GROUP BY options*

The GROUP BY clause has been extended significantly to further refine data analyses associated with aggregation functions. These options are ROLLUP, CUBE and GROUPING SETS which were standardized in SQL3 (SQL 99).

4.2.6.1. *The ROLLUP option*

The role of the ROLLUP option is to extend the operation of the GROUP BY clause. It consists, in the case of using

several criteria, of applying several subdivisions of the global set and not just one. Each subdivision gives rise to several subgroups.

Figure 4.5 illustrates the role of the ROLLUP option in the query.

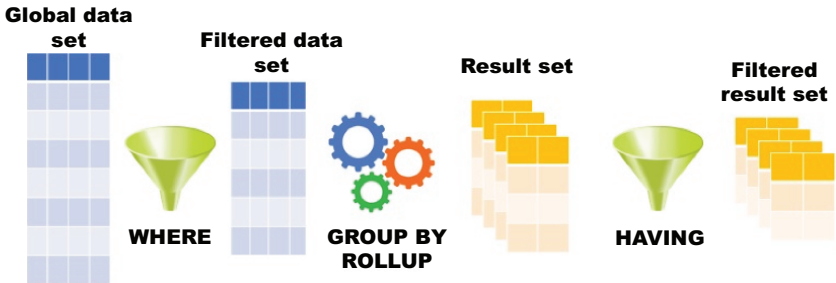


Figure 4.5. *The ROLLUP option*

This figure shows that using the ROLLUP option with the GROUP BY clause generates several result sets displayed together in the same display, i.e. in the form of a dashboard. It is assimilated to the execution of several requests, each using a single subdivision made by the ROLLUP option.

```
USE AdventureWorks2014
GO
SELECT P.Color, P.Style, P.Size, P.Class,
 COUNT(*)AS NOMBRE
FROM Production.Product AS P
GROUP BY ROLLUP (P.Color, P.Style, P.Size, P.Class)
GO
-- On 5 (4+1) subdivisions in this case which are:
-- ((Color, Style, Size, Class), (Color, Style, Size),
-- (Color, Style), (Color), ())
```

Box 4.18. *Using the ROLLUP option in the GROUP BY clause. For a color version of this box, see www.iste.co.uk/ghlala/SQL.zip*

IMPORTANT NOTE.– The number of subdivisions made by the GROUP BY clause with the ROLLUP option using N criteria is equal to N+1 subdivisions, i.e. as if from a merger of the results generated by N+1 requests, each using a single subdivision made by the ROLLUP option.

The order in which the criteria are written in the ROLLUP option affects which subdivisions will be performed, according to a precise algorithm that allows combinations to be generated by eliminating one criterion from right to left of the list each time. Thus, ROLLUP (A, B, C) carries out the following subdivisions: ((A, B, C), (A, B), (A), ()). The last empty subdivision () allows aggregations to be made over the entire dataset.

The number of subgroups generated by each subdivision cannot be determined systematically. It is necessary to refer to the dataset in order to calculate this number.

Interpretation of the results generated by the ROLLUP option is based on whether or not there is a NULL value associated with each criterion to designate its intervention in the calculation or not. This interpretation is further complicated by the presence of null values in the dataset itself. In this case, we will have an ambiguity regarding the origin of the NULL value: is it derived from the data or the operation of the ROLLUP option?

4.2.6.2. *The CUBE option*

The role of the CUBE option is to extend the operation of the GROUP BY clause. It consists, in the case of the use of several criteria, of applying several subdivisions of the global set and not just one. Each subdivision gives rise to several subgroups. The CUBE option is an extension of the ROLLUP option with all possible combinations of the criteria used.

Figure 4.6 shows the role of the CUBE option in the query.

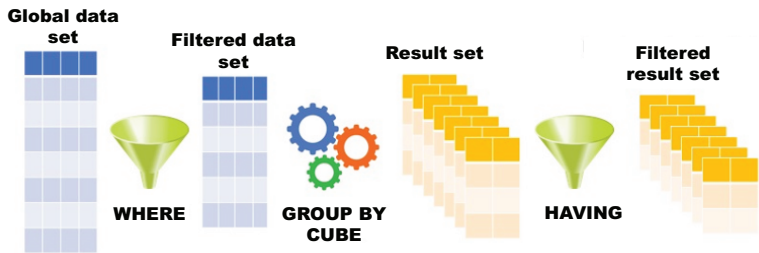


Figure 4.6. *The CUBE option*

```
USE AdventureWorks2014
GO
SELECT P.Color, P.Style, P.Size, P.Class,
 COUNT(*)AS NOMBRE
FROM Production.Product AS P
GROUP BY CUBE (P.Color, P.Style, P.Size, P.Class)
GO
-- On 16 (2^4) subdivisions in this case which are:
-- ((Color, Style, Size, Class),
-- (Color, Style, Size), (Color, Style, Class),
-- (Color, Size, Class), (Style, Size, Class),
-- (Color, Style), (Color, Size), (Color, Class),
-- (Style, Size), (Style, Class), (Size, Class),
-- (Color), (Style), (Size), (Class), ()).
```

Box 4.19. *Using the CUBE option in the GROUP BY clause. For a color version of this box, see www.iste.co.uk/ghlala/SQL.zip*

IMPORTANT NOTE.– The number of subdivisions processed by the GROUP BY clause with the CUBE option using N criteria is equal to 2^N subdivisions.

4.2.6.3. *The GROUPING SETS option*

The GROUPING SETS option is used to customize subdivisions with combinations of your choice.

Figure 4.7 shows the role of the GROUPING SETS option in the query.

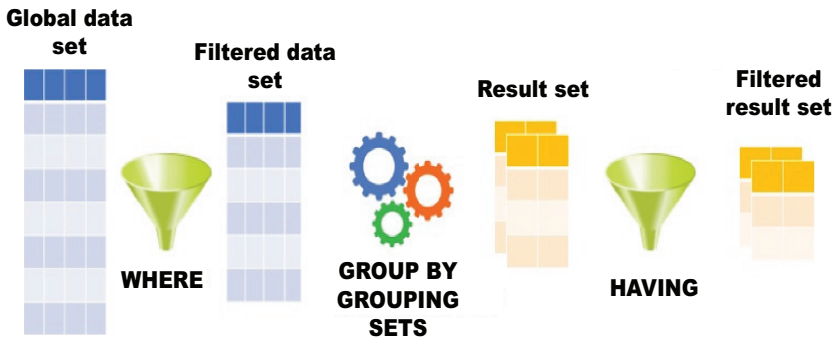


Figure 4.7. *The GROUPING SETS option*

```
USE AdventureWorks2014
GO
SELECT P.Color, P.Style, P.Size, P.Class,
 COUNT(*)AS NOMBRE
FROM Production.Product AS P
GROUP BY GROUPING SETS ((P.Style, P.Class),
 (P.Color, P.Style, P.Size),
 (P.Class, P.Color), (P.Size),
 (P.Color), ()
 )
GO
```

Box 4.20. *Using the GROUPING SETS option in the GROUP BY clause. For a color version of this box, see www.iste.co.uk/ghlala/SQL.zip*

IMPORTANT NOTE.– The GROUPING SETS option of the GROUP BY clause allows you to customize the subdivisions in terms of number and order.

4.2.6.4. *The GROUPING_ID option*

The GROUPING_ID option is in reality a Boolean function used in the SELECT clause. It takes as its parameter the criterion used in the GROUP BY clause regardless of the option used. It allows you to specify

whether the criterion was used (display value 0) in the calculation or not (display value 1).

```
USE AdventureWorks2014
GO
SELECT P.Color, P.Style, P.Size, P.Class,
 GROUPING_ID (P.Color) AS Color_ON,
 GROUPING_ID (P.Style) AS Style_ON,
 GROUPING_ID (P.Size) AS Size_ON,
 GROUPING_ID (P.Class) Class_ON,
 COUNT(*) AS NOMBRE
FROM Production.Product AS P
GROUP BY ROLLUP (P.Color, P.Style, P.Size, P.Class)
GO
```

Box 4.21. *Using the GROUPING_ID function in the SELECT clause. For a color version of this box, see www.iste.co.uk/ghlala/SQL.zip*

IMPORTANT NOTES.– The GROUPING_ID function removes the ambiguity related to null values displayed in the dashboard. Their origins can now be clearly determined, whether they are null data or indicators of the ROLLUP option (or the other options of the GROUP BY clause).

The use of a logical function such as IIF further improves the readability of the dashboard.

```
USE AdventureWorks2014
GO
SELECT P.Color, P.Style, P.Size, P.Class,
 IIF(GROUPING_ID (P.Color)=0,'ON','OFF') AS Color_ON,
 IIF(GROUPING_ID (P.Style)=0,'ON','OFF') AS Style_ON,
 IIF(GROUPING_ID (P.Size)=0,'ON','OFF') AS Size_ON,
 IIF(GROUPING_ID (P.Class)=0,'ON','OFF') AS Class_ON,
 COUNT(*) AS NOMBRE
FROM Production.Product AS P
GROUP BY ROLLUP (P.Color, P.Style, P.Size, P.Class)
GO
```

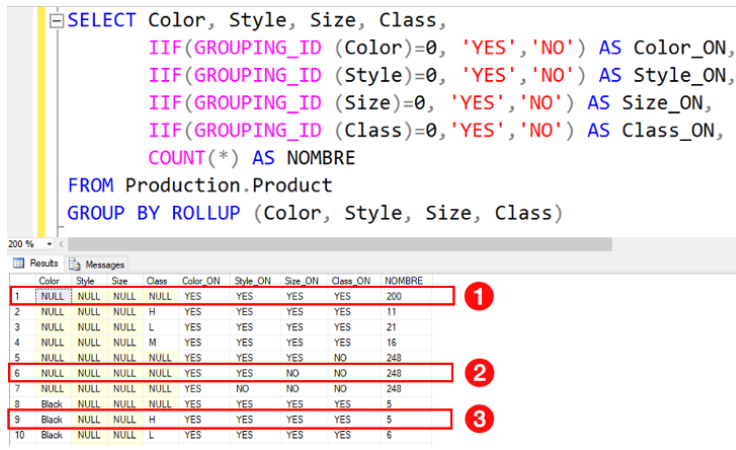
Box 4.22. *Using the GROUPING_ID function with the logical function IIF in the SELECT clause. For a color version of this box, see www.iste.co.uk/ghlala/SQL.zip*

Interpreting the dashboard has now been made easy and pleasant. In fact, Figure 4.8 shows the execution of the request, and its interpretation is as follows:

1) the number 200 represents the number of products with null values for the four characteristics: color, style, size and class;

2) the number 248 represents the number of products with null values for both characteristics: color and style regardless of their sizes and classes;

3) the number 5 represents the number of black and Class H products that do not have style and size, i.e. null values for the latter two characteristics.



```
SELECT Color, Style, Size, Class,
       IIF(GROUPING_ID (Color)=0, 'YES','NO') AS Color_ON,
       IIF(GROUPING_ID (Style)=0, 'YES','NO') AS Style_ON,
       IIF(GROUPING_ID (Size)=0, 'YES','NO') AS Size_ON,
       IIF(GROUPING_ID (Class)=0,'YES','NO') AS Class_ON,
       COUNT(*) AS NOMBRE
FROM Production.Product
GROUP BY ROLLUP (Color, Style, Size, Class)
```

200 % Results Messages

	Color	Style	Size	Class	Color_ON	Style_ON	Size_ON	Class_ON	NOMBRE
1	NULL	NULL	NULL	NULL	YES	YES	YES	YES	200
2	NULL	NULL	NULL	H	YES	YES	YES	YES	11
3	NULL	NULL	NULL	L	YES	YES	YES	YES	21
4	NULL	NULL	NULL	M	YES	YES	YES	YES	16
5	NULL	NULL	NULL	NULL	YES	YES	YES	NO	248
6	NULL	NULL	NULL	NULL	YES	YES	NO	NO	248
7	NULL	NULL	NULL	NULL	YES	NO	NO	NO	248
8	Black	NULL	NULL	NULL	YES	YES	YES	YES	5
9	Black	NULL	NULL	H	YES	YES	YES	YES	5
10	Black	NULL	NULL	L	YES	YES	YES	YES	6

Figure 4.8. *Dashboard with aggregation functions. For a color version of this figure, see www.iste.co.uk/ghlala/SQL.zip*

4.2.7. *Conclusion*

Despite the diversification of business intelligence solutions today, aggregation functions are still the cornerstone of any data analysis. Mastery of these functions is an essential skill, either to apply the “in-database first” approach and access the data directly for analysis, or to

exploit graphical data visualization solutions in an optimal way.

4.3. Windowing functions

4.3.1. *Introduction*

SQL windowing has marked a revolution in this query language. This mechanism consists of sampling from a dataset several times as if we were making several separate queries with a single display unifying all the results.

The dataset is determined only once in the FROM clause of the request while its multiple sampling is done in the SELECT clause with the OVER function, as if we were processing the same dataset several times.

The use of this technique gives us the possibility of:

– using the same query to display different results simultaneously, hence the development of a summary dashboard;

– group complex analyses in the same query, considered without the use of the OVER() function to be syntactic and semantic errors. An example of this analysis is displaying elementary data relating to each of the data lines simultaneously alongside summary indicators such as the minimum, the average or the sum relating to the entire dataset;

– combining this mechanism with SQL language analytic functions such as aggregation functions, ranking functions and distribution functions.

4.3.2. *Mind map of the third section*

This third section will be devoted to presenting data windowing in SQL language based on examples of aggregation functions. The use of analytic functions such as ranking, distribution and offset functions with the OVER() windowing function will be detailed in the following section.

Figure 4.9 presents the mind map of the different concepts that will be covered in this section.

4.3.3. *Operating mode of the OVER() windowing function*

The main objective of data analysis is to develop dashboards that simultaneously include basic data and summary indicators for the processed data. This requirement was not possible before because of the syntactic constraints of the SQL language. The windowing function (OVER) has recently solved this problem and extended the SQL language with new analytic features.

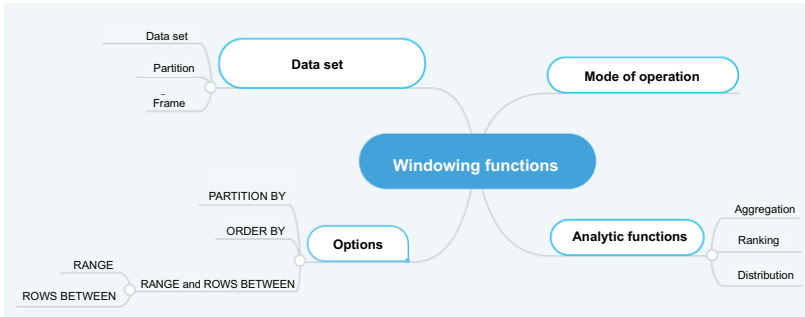


Figure 4.9. *Mind map of the third section*

```
USE AdventureWorks2014
GO
SELECT Name, ListPrice AS Prix,
 Max(ListPrice) AS MAX_Prix
FROM Production.Product AS P
WHERE Color IS NOT NULL
AND ListPrice > 0
GO
```

Box 4.23. *Syntax error due to the combination of elementary data with synthetic data. For a color version of this box, see www.iste.co.uk/ghlala/SQL.zip*

```
USE AdventureWorks2014
GO
SELECT Name, ListPrice AS Prix,
 Max(ListPrice) OVER() AS MAX_Prix
FROM Production.Product AS P
WHERE Color IS NOT NULL
AND ListPrice > 0
GO
```

Box 4.24. *Using the OVER windowing function to combine elementary data with synthetic data. For a color version of this box, see www.iste.co.uk/ghlala/SQL.zip*

In this example, the query behaves as if we have two separate queries; one displays the name and price of each product and the other displays information related to the entire group that is the maximum price, based on the same dataset and providing unified results in the same display.

4.3.4. *Analytic functions associated with the windowing mechanism*

Data windowing in an SQL query only makes sense with one of the analytic functions. The latter can be classified into four categories (Table 4.2).

4.3.5. *Dataset, partition and frame*

Data windowing means sampling a dataset arising from one or more tables several times using the OVER() function. This clause can be used in an advanced way to apply the analytic function associated with each partition of the dataset. The partition can also be subdivided into various parts called "Frames". This advanced operating mode is done by means of options of the OVER() function.

Categories	Functions
Aggregation	MIN, MAX, SUM, AVG, COUNT
Ranking	RANK, DENSE_RANK, NTILE, ROW_NUMBER
Distribution	STDEV, STDEVP, VAR, VAR, VARP, PERCENT_RANK, CUME_DIST, PERCENTILE_CONT, PERCENTILE_DISC
Offset	LEAD, LAG, FIRST_VALUE, LAST_VALUE

Table 4.2. *Categories of analytic functions used with the windowing mechanism*

Figure 4.10 shows the operation of the windowing mechanism performed by the OVER() function.

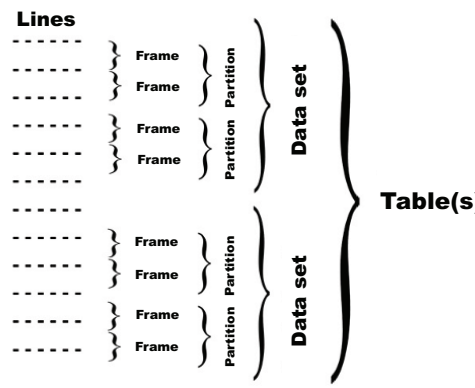


Figure 4.10. *Data windowing with dataset partitioning*

4.3.6. *Windowing options*

For advanced use of the OVER() windowing function, three options are available: PARTITION BY to work with partitions, ORDER BY to sort the dataset processed by the OVER function and ROWS/RANGE to manage frames within partitions.

4.3.6.1. *Option PARTITION BY*

The PARTITION BY option allows you to partition the dataset into several partitions according to one or more criteria. The associated analytic function will be applied to each partition and not to the entire dataset.

```
USE AdventureWorks2014
GO
SELECT Name AS Nom, ListPrice AS Prix,
 Color AS Coleur,
 Max(ListPrice) OVER() AS MAX_P_GENERAL,
 Max(ListPrice) OVER(PARTITION BY Color) AS
 MAX_P_COLOR
FROM Production.Product AS P
WHERE Color IS NOT NULL
AND ListPrice > 0
GO
```

Box 4.25. *OVER() function with the PARTITION BY option. For a color version of this box, see www.iste.co.uk/ghlala/SQL.zip*

IMPORTANT NOTE.– The PARTITION BY option is comparable to the GROUP BY clause used in a query without data windowing. GROUP BY is executed only once while PARTITION BY is executed as many times as there are OVER() clauses in the request.

4.3.6.2. *Option ORDER BY*

Some analytic functions such as Ranking, which will be detailed in the next section, require sorting of the dataset so that they can develop the requested indicators. This sorting operation is also necessary to form frames within the

partition through the ROWS BETWEEN or RANGE option which will be detailed in the following section.

IMPORTANT NOTE.– The ORDER BY option of the OVER() function should not be confused with the ORDER BY clause of the main request. The first is necessary for the application of the analytic function associated with the OVER() function or to manage frames in different partitions of the dataset, while the second is optional for sorting the final result according to one or more specified criteria.

4.3.6.3. *The RANGE and ROWS BETWEEN options*

The ROWS BETWEEN and RANGE options are used to manage frames in the partitions of the dataset processed by the OVER() windowing function.

4.3.6.3.1. ROWS BETWEEN option

The ROWS BETWEEN option specifies that the aggregation functions in the current partition of the OVER() function will take into account the frame defined by the current line and a specific number of lines before or after the current line. The keyword PRECEDING specifies a backward direction and the keyword FOLLOWING specifies a forward direction.

```
/*---------------------------
 PRECEDING
---------------------------- */
-- Display a set of order lines
-- (current line plus any 3 lines
-- previous).
SELECT SalesOrderID, LineTotal,
 SUM(LineTotal) OVER(
 PARTITION BY SalesOrderID
 ORDER BY ModifiedDate
 ROWS BETWEEN 3 PRECEDING AND CURRENT ROW
 ) AS PREC
FROM Sales.SalesOrderDetail
/*---------------------------
 FOLLOWING
---------------------------- */
-- Display a set of order lines
-- (current line plus any 3 lines
-- following).
SELECT SalesOrderID, LineTotal,
 SUM(LineTotal) OVER(
```

```
 PARTITION BY SalesOrderID
 ORDER BY ModifiedDate
 ROWS BETWEEN CURRENT ROW AND 3 FOLLOWING
 ) AS FOLLOW-UP
FROM Sales.SalesOrderDetail
/*-----------------------------------------
 PRECEDING AND FOLLOWING
----------------------------------------- */
-- Display a set of order lines
-- (current line plus any 3 lines
-- following and any 3 lines previous).
SELECT SalesOrderID, LineTotal,
 SUM(LineTotal) OVER(
 PARTITION BY SalesOrderID
 ORDER BY ModifiedDate
 ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING
 ) AS PREC_NEXT
FROM Sales.SalesOrderDetail
/*-----------------------------------------
 ROWS UNBOUNDED PRECEDING
----------------------------------------- */
-- Calculation based on a frame from the
-- beginning of the partition to the current line.
SELECT SalesOrderID, LineTotal,
 SUM(LineTotal) OVER(
 PARTITION BY SalesOrderID
 ORDER BY ModifiedDate
 ROWS UNBOUNDED PRECEDING
 ) AS PREC_NEXT
FROM Sales.SalesOrderDetail
/*-----------------------------------------
 ROWS UNBOUNDED FOLLOWING
----------------------------------------- */
-- Calculation based of a frame from the
-- current line to the end of the partition.
SELECT SalesOrderID, LineTotal,
 SUM(LineTotal) OVER(
 PARTITION BY SalesOrderID
 ORDER BY ModifiedDate
 ROWS BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING
 ) AS PREC_NEXT
FROM Sales.SalesOrderDetail
GO
```

Box 4.26. *Frame specified with the ROWS BETWEEN option. For a color version of this box, see www.iste.co.uk/ghlala/SQL.zip*

IMPORTANT NOTE.– Both the ROWS BETWEEN and the RANGE options require the ORDER BY option.

4.3.6.3.2. RANGE option

The RANGE option specifies that the aggregation functions in the current partition of the OVER() function will

take into account the frame defined by the current value and the lower or upper bound of the partition.

```
/*-----------------------------------------
 RANGE UNBOUNDED PRECEDING
----------------------------------------- */
-- Calculation based on a frame from the
-- start of the partition up to the current value.
SELECT SalesOrderID, LineTotal,
 SUM(LineTotal) OVER(
 PARTITION BY SalesOrderID
 ORDER BY ModifiedDate
 RANGE UNBOUNDED PRECEDING
 ) AS PREC_NEXT
FROM Sales.SalesOrderDetail
/*-----------------------------------------
 RANGE UNBOUNDED FOLLOWING
----------------------------------------- */
-- Calculation based on a frame from the
-- current value up to the end of the partition.
SELECT SalesOrderID, LineTotal,
 SUM(LineTotal) OVER(
 PARTITION BY SalesOrderID
 ORDER BY ModifiedDate
 RANGE UNBOUNDED FOLLOWING
 ) AS PREC_NEXT
FROM Sales.SalesOrderDetail
GO
```

Box 4.27. *Frame specified with the RANGE option. For a color version of this box, see www.iste.co.uk/ghlala/SQL.zip*

IMPORTANT NOTE.– The impact of the RANGE option will be clearly noted with duplicated values. In this case, the calculation will be carried out over the frame defined by the value, regardless of its line, and the indicated boundary (upper or lower).

4.3.7. *Conclusion*

Windowing features have given SQL a new lease of life so that it can continue to excel in data analysis.

4.4. Analytic functions

4.4.1. *Introduction*

Analytic functions are a powerful mechanism for succinctly representing complex analytic operations. They allow efficient evaluations that would otherwise involve costly auto-joins or calculations outside the SQL query language. These analytic functions can be classified into three categories: ranking, distribution and offset functions.

4.4.2. *Mind map of the fourth section*

This fourth section will be devoted to presenting the analytic functions associated with the OVER() data windowing function.

Figure 4.11 presents the mind map of the different concepts that will be covered in this section.

4.4.3. *Ranking functions*

4.4.3.1. *Introduction*

Ranking functions are an important facet of data analysis. They allow us to assign a rank to each row of data in the processed dataset.

The role of these functions is comparable to that of the RANK function in Microsoft Excel, and they are used in SQL language as analytic functions associated with the data windowing function which is OVER().

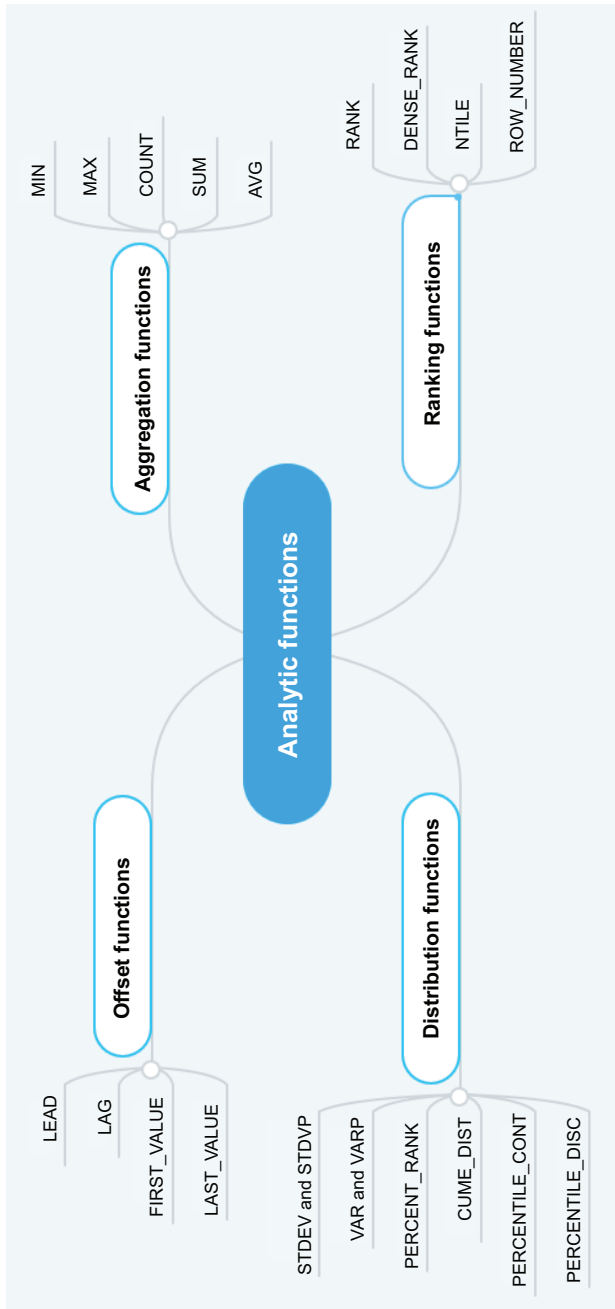


Figure 4.11. *Mind map of the fourth section*

This type of analytic function should not be confused with data sorting, because their role is not to act on the order in which the results are displayed but rather to assign ranks to the different data lines according to an algorithm suitable for each of them.

4.4.3.2. *The RANK function*

The RANK() function allows us to apply the classification method which consists of assigning the same rank for ties while omitting the next rank(s), which can generate a series of non-consecutive ranks.

```
USE AdventureWorks2014
GO
-- Use of the RANK function to classify
-- totals of order lines (LineTotal).
-- Processing carried out for Order.
SELECT ProductID, LineTotal,
 RANK() OVER(PARTITION BY SalesOrderID
 ORDER BY LineTotal)
 AS Classement_Rank
FROM Sales.SalesOrderDetail
GO
```

Box 4.28. *RANK function. For a color version of this box, see www.iste.co.uk/ghlala/SQL.zip*

IMPORTANT NOTE.– For all classification functions, the OVER() function requires the ORDER BY option.

4.4.3.3. *The DENSE_RANK function*

The DENSE_RANK() function allows you to apply the classification method which consists of assigning the same rank for ties without omitting the next rank(s), which always generates a series of consecutive ranks.

```
USE AdventureWorks2014
GO
-- Use of the DENSE_RANK function to classify
-- the totals of order lines (LineTotal).
-- Processing carried out for Order.
SELECT ProductID, LineTotal,
 DENSE_RANK() OVER(PARTITION BY SalesOrderID
```

```
 ORDER BY LineTotal)
 AS Classement_Dense_Rank
FROM Sales.SalesOrderDetail
GO
```

Box 4.29. *DENSE_RANK function. For a color version of this box, see www.iste.co.uk/ghlala/SQL.zip*

4.4.3.4. *The NTILE function*

In descriptive statistics, a quartile is the subdivision of the sorted data into four equal parts. Each share represents 1/4 of the population sample and each element of the population will be classified into one of these four segments. The quartile is a special case of the quantile function implemented in SQL with the NTILE(N) function. Therefore, the quartile is simply NTILE(4).


```
USE AdventureWorks2014
GO
-- Use of the NTILE function to apply
-- the Quartile statistical function on
-- totals of order lines (LineTotal).
-- Processing carried out for Order.
SELECT ProductID, LineTotal,
 NTILE(4) OVER(PARTITION BY SalesOrderID
 ORDER BY LineTotal)
 AS Quartile
FROM Sales.SalesOrderDetail
GO
```

Box 4.30. *NTILE function. For a color version of this box, see www.iste.co.uk/ghlala/SQL.zip*

4.4.3.5. *The ROW_NUMBER function*

The ROW_NUMBER() function assigns a unique number for each line beginning with 1. For rows containing duplicate values, numbers are assigned arbitrarily.

```
USE AdventureWorks2014
GO
-- Use of the ROW_NUMBER function to
-- assign ordinal numbers for the totals of the
-- order lines (LineTotal).
-- Processing carried out for Order.
SELECT ProductID, LineTotal,
 ROW_NUMBER() OVER(PARTITION BY SalesOrderID
 ORDER BY LineTotal)
 AS Num_Order_Ligne
FROM Sales.SalesOrderDetail
GO
```

Box 4.31. *ROW_NUMBER function. For a color version of this box, see www.iste.co.uk/ghlala/SQL.zip*

IMPORTANT NOTE.– The numbers generated by the ROW_NUMBER() function are part of the query result dataset and are not part of the software interface used.

4.4.3.6. *Conclusion*

The choice of the ranking function to be applied reflects a business rule specifying the method used by the information system to rank the data.

4.4.4. *Distribution functions*

4.4.4.1. *Introduction*

Distribution functions provide another alternative in data analysis. They allow us to enrich the SQL language with functions from the theory of statistics and probability.

4.4.4.2. *The standard deviation STDEV and STDEVP*

The standard deviation measures the dispersion of values around the mean. Standard deviation is calculated either from a sample-based approach (STDEV) or over the whole population (STDEVP).

```
USE AdventureWorks2014
-- Calculation of standard deviation of order amounts
-- per client using the sample-based approach.
GO
SELECT CustomerID, STDEV(TotalDue)
FROM Sales.SalesOrderHeader
GROUP BY CustomerID
GO
```

Box 4.32. *STDEV function. For a color version of this box, see www.iste.co.uk/ghlala/SQL.zip*

```
USE AdventureWorks2014
-- Calculation of standard deviation of order amounts
-- per client with the approach using the full
-- population.
GO
SELECT CustomerID, STDEVP(TotalDue)
FROM Sales.SalesOrderHeader
GROUP BY CustomerID
GO
```

Box 4.33. *STDEVP function. For a color version of this box, see www.iste.co.uk/ghlala/SQL.zip*

4.4.4.3. *The variance VAR and VARP*

Variance is an interesting statistical function for checking whether the values of a data population are very scattered or whether they are close to the average. In the same way as the calculation of the standard deviation, which is only the square root of the variance, calculation of the variance may use a sample-based approach (VAR) or the whole population (VARP).

```
USE AdventureWorks2014
-- Calculation of the variance of order amounts
-- per client with the sample-based approach.
GO
SELECT CustomerID, VAR(TotalDue)
FROM Sales.SalesOrderHeader
GROUP BY CustomerID
GO
```

Box 4.34. *VAR function. For a color version of this box, see www.iste.co.uk/ghlala/SQL.zip*

```
USE AdventureWorks2014
-- Calculation of the variance of order amounts
-- per client with the approach using the full
-- population.
GO
SELECT CustomerID, VARP(TotalDue)
FROM Sales.SalesOrderHeader
GROUP BY CustomerID
GO
```

Box 4.35. *VARP function. For a color version of this box, see www.iste.co.uk/ghlala/SQL.zip*

IMPORTANT NOTE.– The functions for calculating the standard deviation and variance do not necessarily require the OVER() windowing function.

```
USE AdventureWorks2014
GO
SELECT CustomerID, TotalDue,
 STDEV(TotalDue) OVER(PARTITION BY CustomerID)
 AS SD_Sample,
 STDEVP(TotalDue) OVER(PARTITION BY CustomerID)
 AS SD_Population,
 VAR(TotalDue) OVER(PARTITION BY CustomerID)
 AS Var_Sample,
 VARP(TotalDue) OVER(PARTITION BY CustomerID)
 AS Var_Population
FROM Sales.SalesOrderHeader
GO
```

Box 4.36. *Standard deviation and variance calculation functions with windowing. For a color version of this box, see www.iste.co.uk/ghlala/SQL.zip*

4.4.4.4. *The cumulative distribution PERCENT_RANK and CUME_DIST*

The functions PERCENT_RANK and CUME_DIST allow us to calculate the cumulative distribution, with a slight difference. In fact, both functions return a value between 0 and 1 calculated as follows:

PERCENT_RANK(x) = Rank(s) – 1/Total number of rows – 1

CUME_DIST(x) = Number of previous rows/Total number of rows

```
USE AdventureWorks2014
GO
-- Display percentages of sums
-- lower than the current sum of the group,
-- excluding the highest value.
-- Processing performed for each customer.
SELECT CustomerID, TotalDue,
 PERCENT_RANK() OVER(PARTITION BY CustomerID
 ORDER BY TotalDue)
 AS Pourcentage_Inf
FROM Sales.SalesOrderHeader
GO
```

Box 4.37. *PERCENT_RANK function. For a color version of this box, see www.iste.co.uk/ghlala/SQL.zip*

```
USE AdventureWorks2014
GO
-- Display percentages of values lower
-- or equal to the current amount.
-- Processing performed for each customer.
SELECT CustomerID, TotalDue,
 CUME_DIST() OVER(PARTITION BY CustomerID
 ORDER BY TotalDue)
 AS Pourcentage_Inf_Equal
FROM Sales.SalesOrderHeader
GO
```

Box 4.38. *Function CUME_DIST. For a color version of this box, see www.iste.co.uk/ghlala/SQL.zip*

IMPORTANT NOTE.– The first value returned by PERCENT_RANK is always 0. The last values returned by both PERCENT_RANK and CUME_DIST are equal to 1.

4.4.4.5. *Reverse distribution PERCENTILE*

Percentiles are values of a quantitative variable that divide the data sorted into groups by hundredths. The most typical application is that of PERCENTILE (0.5), which is the median. We can also specify specific percentiles (e.g. PERCENTILE (0.95) which represents the 95th percentile, a value higher than 95% of observations).

Percentiles can be discrete or continuous. In the first case, the returned values are from the dataset, while in the second case, these values will be calculated as a percentage on a continuous basis.

IMPORTANT NOTE.– Functions related to the calculation of percentiles should not be confused with the NTILE() classification function, which returns a ranking of the line in relation to the group it belongs to, while the percentile functions return cumulative percentage data.

```
USE AdventureWorks2014
-- Display the median of the order amounts.
-- Processing performed for each customer.
GO
SELECT CustomerID, TotalDue,
 PERCENTILE_CONTENT(0.5)
 WITHIN GROUP ( ORDER BY TotalDue)
 OVER (PARTITION BY CustomerID) AS DICDCTDBCID
FROM Sales.SalesOrderHeader
GO
```

Box 4.39. *Continuous reverse distribution using the PERCENT_CONT function. For a color version of this box, see www.iste.co.uk/ghlala/SQL.zip*

```
USE AdventureWorks2014
-- 75th percentile display of orders.
-- Processing performed for each customer.
GO
SELECT CustomerID, TotalDue,
 PERCENTILE_DISC(0.75)
 WITHIN GROUP ( ORDER BY TotalDue)
 OVER (PARTITION BY CustomerID) AS DICDCTDBCID
FROM Sales.SalesOrderHeader
GO
```

Box 4.40. *Discrete reverse distribution using the PERCENTILE_DISC function. For a color version of this box, see www.iste.co.uk/ghlala/SQL.zip*

4.4.4.6. *Conclusion*

The benefits of analytic functions and in particular distribution functions should not hide certain limitations

reflected by the complexity of applying these functions. This complexity lies in writing queries using analytic functions, but above all in interpreting the results provided by these queries.

4.4.5. *Offset functions*

4.4.5.1. *Introduction*

Offset functions contribute in their turn to the development of data analysis. They aim to present relationships between a datum and its successors and/or predecessors. The contribution of these functions is important since they provide a summary view of the data in relation to the rest of the processed dataset.

4.4.5.2. *The LEAD function*

The LEAD function displays the successor of the current value in the current dataset or partition.

```
USE AdventureWorks2014
GO
-- Display the successor (one step) of each
-- amount in the total list of orders.
SELECT CustomerID, TotalDue,
 LEAD(TotalDue) OVER(ORDER BY TotalDue)
 AS SUCC_Montant_TOT_1_Pas
FROM Sales.SalesOrderHeader
-- Display the successor (more than one step) of
-- each amount in the total list of orders.
SELECT CustomerID, TotalDue,
 LEAD(TotalDue, 2) OVER(ORDER BY TotalDue)
 AS SUCC_Montant_TOT_2_Pas
FROM Sales.SalesOrderHeader
-- Display the successor (one step) of each
-- amount in the list of orders relating to
-- each customer.
SELECT CustomerID, TotalDue,
 LEAD(TotalDue) OVER(PARTITION BY CustomerID
 ORDER BY TotalDue)
 AS SUCC_Montant_Par_Clt_1_Pas
FROM Sales.SalesOrderHeader
-- Display the successor (more than one step) of
-- each amount in the order list
-- relating to each customer.
SELECT CustomerID, TotalDue,
```

```
 LEAD(TotalDue,2) OVER(PARTITION BY CustomerID
 ORDER BY TotalDue)
 AS SUCC_Montant_Par_Clt_2_Pas
FROM Sales.SalesOrderHeader
GO
```

Box 4.41. *LEAD function. For a color version of this box, see www.iste.co.uk/ghlala/SQL.zip*

4.4.5.3. *The LAG function*

The LAG function displays the predecessor of the current value in the current dataset or partition.

```
USE AdventureWorks2014
GO
-- Display the predecessor (one step) of each
-- amount in the total list of orders.
SELECT CustomerID, TotalDue,
 LAG(TotalDue) OVER(ORDER BY TotalDue)
 AS PRED_TOT_Amount_1_Step
FROM Sales.SalesOrderHeader
-- Display the predecessor (more than one step)
-- of each amount in the total list of
-- orders.
SELECT CustomerID, TotalDue,
 LAG(TotalDue, 2) OVER(ORDER BY TotalDue)
 AS PRED_Montant_TOT_2_Pas
FROM Sales.SalesOrderHeader
-- Display the predecessor (one step) of each
-- amount in the list of orders
-- relating to each customer.
SELECT CustomerID, TotalDue,
 LAG(TotalDue) OVER(PARTITION BY CustomerID
 ORDER BY TotalDue)
 AS PRED_Montant_Par_Clt_1_Pas
FROM Sales.SalesOrderHeader
-- Display the predecessor (more than one step)
-- of each amount in the order list
-- relating to each customer.
SELECT CustomerID, TotalDue,
 LAG(TotalDue,2) OVER(PARTITION BY CustomerID
 ORDER BY TotalDue)
 AS PRED_Montant_Par_Clt_2_Pas
FROM Sales.SalesOrderHeader
GO
```

Box 4.42. *LAG function. For a color version of this box, see www.iste.co.uk/ghlala/SQL.zip*

IMPORTANT NOTES.– Both LEAD and LAG have a third parameter indicating the default value to be considered in the absence of possible values. If this parameter is not specified, both functions use the default value NULL.

All offset functions require the ORDER BY option.

4.4.5.4. *The FIRST_VALUE function*

The FIRST_VALUE function displays the first value in the current dataset or partition.

```
USE AdventureWorks2014
GO
-- Display the first amount in the total list
-- of orders.
SELECT CustomerID, TotalDue,
 FIRST_VALUE(TotalDue) OVER(ORDER BY TotalDue)
 AS FIRST_Montant_TOT
FROM Sales.SalesOrderHeader
-- Display the first amount in the list of
-- orders related to each customer.
SELECT CustomerID, TotalDue,
 FIRST_VALUE(TotalDue) OVER(PARTITION BY CustomerID
 ORDER BY TotalDue)
 AS FIRST_Montant_Par_Clt
FROM Sales.SalesOrderHeader
GO
```

Box 4.43. *FIRST_VALUE function. For a color version of this box, see www.iste.co.uk/ghlala/SQL.zip*

IMPORTANT NOTE.– The difference between the FIRST_VALUE() offset function and the MIN() aggregation function is that the latter does not take into account zero values. If the first value of a dataset is NULL, the offset function FIRST_VALUE always returns the NULL value, while the MIN() aggregation function first returns the NULL value and then the minimum value of the set or partition.

4.4.5.5. *The LAST_VALUE function*

The LAST_VALUE function displays the last value in the current dataset or partition.

```
USE AdventureWorks2014
GO
-- Display the last amount in the total list
-- of orders.
SELECT CustomerID, TotalDue,
 LAST_VALUE(TotalDue) OVER(ORDER BY TotalDue)
 AS LAST_Montant_TOT
FROM Sales.SalesOrderHeader
-- Display the last amount in the list of
-- orders related to each customer.
SELECT CustomerID, TotalDue,
 LAST_VALUE(TotalDue) OVER(PARTITION BY CustomerID
 ORDER BY TotalDue)
 AS LAST_Amount_Per_Clt
FROM Sales.SalesOrderHeader
GO
```

Box 4.44. *LAST_VALUE function. For a color version of this box, see www.iste.co.uk/ghlala/SQL.zip*

4.4.5.6. *Conclusion*

Enhancing SQL language with analytic functions such as offset is very cost-effective. This contribution is explained by the simplicity and optimization of processing that would have required a great deal of development effort with procedural logic if we did not have this type of function.

References

BEN-GAN I., *T-SQL Fundamentals*, Third Edition, Microsoft Press, Redmond, 2014.

BEN-GAN I., *Querying Data with Transact - SQL Exam Ref 70-761*, Microsoft Edition, 2016.

BEN-GAN I., SARKA D., MACHANIC A. *et al.*, *T-SQL Querying*, Microsoft Press, Redmond, 2015.

BEN-GAN I., SARKA D., TALMAGE R., *Querying Microsoft SQL Server 2012 Exam 70-461*, Microsoft Press, Redmond, 2012.

BRIARD G., *Oracle 10g Sous Windows*, Eyrolles, Paris, 2006.

CLOUSE M., *Algèbre relationnelle: Guide pratique de conception d'une base de données relationnelle*, ENI Editions, Saint-Herblain, 2008.

CODD E.F., "A relational model of data for large shared data banks", *Communications of the ACM*, vol. 13, no. 6, pp. 377–387, 1970.

GABILLAUD J., *Oracle 11g SQL, Pl/SQL, SQL* Plus*, ENI Editions, Saint-Herblain, 2009.

GABILLAUD J., *SQL Server 2014 SQL Transact-SQL: Conception et réalisation d'une base de données*, ENI Editions, Saint-Herblain, 2014.

Analytic SQL in SQL Server 2014/2016, First Edition. Riadh Ghlala.

GABILLAUD J., *Oracle 12c SQL, Pl/SQL, SQL* Plus*, ENI Editions, Saint-Herblain, 2015.

GARDARIN G., *Bases de données*, Eyrolles, Paris, 2003.

HAINAUT J.L., *Bases de données: Concepts, utilisation et développement*, Dunod, Paris, 2018.

JORGENSEN A., LEBLANC P., CHINCHILLA J. *et al.*, *Microsoft® SQL SERVER®2012 BIBLE*, John Wiley & Sons, Hoboken, 2012.

LEBLANC P., *Microsoft® SQL SERVER®2012 Step by Step*, Microsoft Press, Redmond, 2013.

SOUTOU C., *Apprendre SQL avec MySQL: Avec 40 exercices corrigés*, Eyrolles, Paris, 2006.

SOUTOU C., *SQL Pour Oracle Applications avec Java, PHP et XML*, Eyrolles, Paris, 2015.

WATSON J., RAMKLASS R., *OCA Oracle Database 11g: SQL Fundamentals I Exam Guide (Exam 1Z0-051)*, The McGraw-Hill Companies, New York, 2008.

Index

C, D, E

F, J, O

Analytic SQL in SQL Server 2014/2016, First Edition. Riadh Ghlala.

R, S, T, V

Other titles from

in

Computer Engineering

2019

CLERC Maurice
Iterative Optimizers: Difficulty Measures and Benchmarks

TOUNSI Wiem
Cyber-Vigilance and Digital Trust: Cyber Security in the Era of Cloud Computing and IoT

2018

ANDRO Mathieu
Digital Libraries and Crowdsourcing
(Digital Tools and Uses Set – Volume 5)

ARNALDI Bruno, GUITTON Pascal, MOREAU Guillaume
Virtual Reality and Augmented Reality: Myths and Realities

BERTHIER Thierry, TEBOUL Bruno
From Digital Traces to Algorithmic Projections

CARDON Alain
Beyond Artificial Intelligence: From Human Consciousness to Artificial Consciousness

HOMAYOUNI S. Mahdi, FONTES Dalila B.M.M.
Metaheuristics for Maritime Operations
(Optimization Heuristics Set – Volume 1)

JEANSOULIN Robert
JavaScript and Open Data

PIVERT Olivier
NoSQL Data Models: Trends and Challenges
(Databases and Big Data Set – Volume 1)

SEDKAOUI Soraya
Data Analytics and Big Data

SALEH Imad, AMMI Mehdi, SZONIECKY Samuel
Challenges of the Internet of Things: Technology, Use, Ethics
(Digital Tools and Uses Set – Volume 7)

SZONIECKY Samuel
Ecosystems Knowledge: Modeling and Analysis Method for Information and Communication
(Digital Tools and Uses Set – Volume 6)

2017

BENMAMMAR Badr
Concurrent, Real-Time and Distributed Programming in Java

HÉLIODORE Frédéric, NAKIB Amir, ISMAIL Boussaad, OUCHRAA Salma, SCHMITT Laurent
Metaheuristics for Intelligent Electrical Networks
(Metaheuristics Set – Volume 10)

MA Haiping, SIMON Dan
Evolutionary Computation with Biogeography-based Optimization
(Metaheuristics Set – Volume 8)

PÉTROWSKI Alain, BEN-HAMIDA Sana
Evolutionary Algorithms
(Metaheuristics Set – Volume 9)

PAI G A Vijayalakshmi
Metaheuristics for Portfolio Optimization
(Metaheuristics Set – Volume 11)

2016

BLUM Christian, FESTA Paola
Metaheuristics for String Problems in Bio-informatics
(Metaheuristics Set – Volume 6)

DEROUSSI Laurent
Metaheuristics for Logistics
(Metaheuristics Set – Volume 4)

DHAENENS Clarisse and JOURDAN Laetitia
Metaheuristics for Big Data
(Metaheuristics Set – Volume 5)

LABADIE Nacima, PRINS Christian, PRODHON Caroline
Metaheuristics for Vehicle Routing Problems
(Metaheuristics Set – Volume 3)

LEROY Laure
Eyestrain Reduction in Stereoscopy

LUTTON Evelyne, PERROT Nathalie, TONDA Albert
Evolutionary Algorithms for Food Science and Technology
(Metaheuristics Set – Volume 7)

MAGOULÈS Frédéric, ZHAO Hai-Xiang
Data Mining and Machine Learning in Building Energy Analysis

RIGO Michel
Advanced Graph Theory and Combinatorics

2015

BARBIER Franck, RECOUSSINE Jean-Luc
COBOL Software Modernization: From Principles to Implementation with the BLU AGE® Method

CHEN Ken
Performance Evaluation by Simulation and Analysis with Applications to Computer Networks

CLERC Maurice
Guided Randomness in Optimization
(Metaheuristics Set – Volume 1)

DURAND Nicolas, GIANAZZA David, GOTTELAND Jean-Baptiste, ALLIOT Jean-Marc
Metaheuristics for Air Traffic Management
(Metaheuristics Set – Volume 2)

MAGOULÈS Frédéric, ROUX François-Xavier, HOUZEAUX Guillaume
Parallel Scientific Computing

MUNEESAWANG Paisarn, YAMMEN Suchart
Visual Inspection Technology in the Hard Disk Drive Industry

2014

BOULANGER Jean-Louis
Formal Methods Applied to Industrial Complex Systems

BOULANGER Jean-Louis
Formal Methods Applied to Complex Systems:
Implementation of the B Method

GARDI Frédéric, BENOIST Thierry, DARLAY Julien, ESTELLON Bertrand, MEGEL Romain
Mathematical Programming Solver based on Local Search

KRICHEN Saoussen, CHAOUACHI Jouhaina
Graph-related Optimization and Decision Support Systems

LARRIEU Nicolas, VARET Antoine
Rapid Prototyping of Software for Avionics Systems: Model-oriented Approaches for Complex Systems Certification

OUSSALAH Mourad Chabane
Software Architecture 1
Software Architecture 2

PASCHOS Vangelis Th
Combinatorial Optimization – 3-volume series, 2nd Edition
Concepts of Combinatorial Optimization – Volume 1, 2nd Edition
Problems and New Approaches – Volume 2, 2nd Edition
Applications of Combinatorial Optimization – Volume 3, 2nd Edition

QUESNEL Flavien
Scheduling of Large-scale Virtualized Infrastructures: Toward Cooperative Management

RIGO Michel
Formal Languages, Automata and Numeration Systems 1: Introduction to Combinatorics on Words
Formal Languages, Automata and Numeration Systems 2: Applications to Recognizability and Decidability

SAINT-DIZIER Patrick
Musical Rhetoric: Foundations and Annotation Schemes

TOUATI Sid, DE DINECHIN Benoit
Advanced Backend Optimization

2013

ANDRÉ Etienne, SOULAT Romain
The Inverse Method: Parametric Verification of Real-time Embedded Systems

BOULANGER Jean-Louis
Safety Management for Software-based Equipment

DELAHAYE Daniel, PUECHMOREL Stéphane
Modeling and Optimization of Air Traffic

FRANCOPOULO Gil
LMF — Lexical Markup Framework

GHÉDIRA Khaled
Constraint Satisfaction Problems

ROCHANGE Christine, UHRIG Sascha, SAINRAT Pascal
Time-Predictable Architectures

WAHBI Mohamed
Algorithms and Ordering Heuristics for Distributed Constraint Satisfaction Problems

ZELM Martin *et al.*
Enterprise Interoperability

2012

ARBOLEDA Hugo, ROYER Jean-Claude
Model-Driven and Software Product Line Engineering

BLANCHET Gérard, DUPOUY Bertrand
Computer Architecture

BOULANGER Jean-Louis
Industrial Use of Formal Methods: Formal Verification

BOULANGER Jean-Louis
Formal Method: Industrial Use from Model to the Code

CALVARY Gaëlle, DELOT Thierry, SÈDES Florence, TIGLI Jean-Yves
Computer Science and Ambient Intelligence

MAHOUT Vincent
Assembly Language Programming: ARM Cortex-M3 2.0: Organization, Innovation and Territory

MARLET Renaud
Program Specialization

SOTO Maria, SEVAUX Marc, ROSSI André, LAURENT Johann
Memory Allocation Problems in Embedded Systems: Optimization Methods

2011

BICHOT Charles-Edmond, SIARRY Patrick
Graph Partitioning

BOULANGER Jean-Louis
Static Analysis of Software: The Abstract Interpretation

CAFERRA Ricardo
Logic for Computer Science and Artificial Intelligence

HOMES Bernard
Fundamentals of Software Testing

KORDON Fabrice, HADDAD Serge, PAUTET Laurent, PETRUCCI Laure
Distributed Systems: Design and Algorithms

KORDON Fabrice, HADDAD Serge, PAUTET Laurent, PETRUCCI Laure
Models and Analysis in Distributed Systems

LORCA Xavier
Tree-based Graph Partitioning Constraint

TRUCHET Charlotte, ASSAYAG Gerard
Constraint Programming in Music

VICAT-BLANC PRIMET Pascale *et al.*
Computing Networks: From Cluster to Cloud Computing

2010

AUDIBERT Pierre
Mathematics for Informatics and Computer Science

BABAU Jean-Philippe *et al.*
Model Driven Engineering for Distributed Real-Time Embedded Systems 2009

BOULANGER Jean-Louis
Safety of Computer Architectures

MONMARCHE Nicolas *et al.*
Artificial Ants

PANETTO Hervé, BOUDJLIDA Nacer
Interoperability for Enterprise Software and Applications 2010

SIGAUD Olivier *et al*.
Markov Decision Processes in Artificial Intelligence

SOLNON Christine
Ant Colony Optimization and Constraint Programming

AUBRUN Christophe, SIMON Daniel, SONG Ye-Qiong *et al.*
Co-design Approaches for Dependable Networked Control Systems

2009

FOURNIER Jean-Claude
Graph Theory and Applications

GUEDON Jeanpierre
The Mojette Transform / Theory and Applications

JARD Claude, ROUX Olivier
Communicating Embedded Systems / Software and Design

LECOUTRE Christophe
Constraint Networks / Targeting Simplicity for Techniques and Algorithms

2008

BANÂTRE Michel, MARRÓN Pedro José, OLLERO Hannibal, WOLITZ Adam
Cooperating Embedded Systems and Wireless Sensor Networks

MERZ Stephan, NAVET Nicolas
Modeling and Verification of Real-time Systems

PASCHOS Vangelis Th
Combinatorial Optimization and Theoretical Computer Science: Interfaces and Perspectives

WALDNER Jean-Baptiste
Nanocomputers and Swarm Intelligence

2007

BENHAMOU Frédéric, JUSSIEN Narendra, O'SULLIVAN Barry
Trends in Constraint Programming

JUSSIEN Narendra
A TO Z OF SUDOKU

2006

BABAU Jean-Philippe *et al.*
From MDD Concepts to Experiments and Illustrations – DRES 2006

HABRIAS Henri, FRAPPIER Marc
Software Specification Methods

MURAT Cecile, PASCHOS Vangelis Th
Probabilistic Combinatorial Optimization on Graphs

PANETTO Hervé, BOUDJLIDA Nacer
Interoperability for Enterprise Software and Applications 2006 / IFAC-IFIP I-ESA'2006

2005

GÉRARD Sébastien *et al.*
Model Driven Engineering for Distributed Real Time Embedded Systems

PANETTO Hervé
Interoperability of Enterprise Software and Applications 2005