The SQL Language - How to create queries to manipulate data of a RDB

1. Introduction

In a RDB, **data manipulation is obtained by making use of a meta-language called Structured Query Language (SQL)**. This language makes use and operationally implements the abstract concepts of relational algebra, which we have discussed in the previous chapter.

Although the acronym SQL refers, directly, to the word 'query', data retrieval and data storing are not the only actions that can be performed using SQL. Indeed, SQL is, in fact, a unified meta-language that enables the execution of all the basic (and complex) operations that are necessary for the management and for the use of a RDB. Even the definition and, lately, the management of the relational schema (i.e., definition of tables, fields, primary keys, relations, etc.) and the execution of administrative tasks can be performed using SQL statements. So, this meta-language may be used by all categories of users (key users, administrators and programmers).

In fact, SQL is based on two different sets of instructions:

- The definition statements known as **Data Definition Language (DDL)** - This is a set of instructions necessary to create the database objects (mainly tables and relations). At present, the DDL is rarely used, given that many development platforms - such as Microsoft Access - allow one to create tables, and other objects in an easier and quicker way, by means of graphical and interactive applications.
- The manipulation instructions known as **Data Manipulation Language (DML)** - This is a set of instructions that allow one to interrogate the database using appropriate queries.

  Essentially the DML consists of four instructions and four fundamental operations, as shown by Table 4.1:

Tab. 4.1
SQL Main Instruction and Operations

FUNDAMENTAL INSTRUCTIONS	BASIC OPERATIONS
✓ DELETE	✓ INNER JOIN
✓ INSERT	✓ LEFT JOIN
✓ UPDATE	✓ RIGHT JOIN
✓ SELECT	✓ UNION

It is worth noting that **SQL is a meta-language, but it is certainly not a programming language** and therefore **does not contain the typical control mechanisms** of both procedural and object-oriented languages (Visual Basic, Basic.Net, Delphi, Java, C#, Python, etc.) such as IF ... THEN, FOR ... NEXT, DO WHILE ... LOOP, etc.

However, for a practical use of SQL **it is sometime useful to include SQL statements inside a conventional programming language** (most of the current programing language allow this possibility); in this case the use of SQL is called embedded, while the programming language is called the host language.

Before proceeding further on, we report in a schematic way the format (syntax), with which the SQL command statements will be presented.

<> - Arrows delimit the names of key elements of the SQL language. For example, we will use expression such as: <*command*>, <*expression*>, <*identifier*>;

[] - Brackets denote an optional element;

... - Three dots (the ellipsis) indicate that the previous item can be repeated a limitless number of tile

{} - Braces are used to group, together, multiple elements of a definition;

| - A vertical bar line between two elements indicates that the preceding element is an alternative to the following one (i.e., OR operator).

We also observe that an SQL command is normally constituted by an operation followed by one or more clauses that specify the effect of the operation. Thus, the general command definition has the following shape:

<*Command*>:: = <action> <clause> [<clause> ...]

2. The Select Operator

As it was said, queries allow one to search specific data in a RDB.

To this aims there is the need to use selection queries that begin with the SELECT keyword. Obviously a selection query acts on one or more tables (or even on tables obtained by other queries) and returns a new table.

The command syntax is as follows:

```
<Select Command> :: =
        SELECT [ALL | DISTINCT] <Selection List>
         <Table Expression>
         [<Sorting Criteria>]
```

Where <*Table Expression*> is defined as follows

```
<Table Expression> :: =
        <FROM Expression>
        [<WHERE Expression>]
        [<GROUP BY Expression>]
        [<HAVING Expression>]
```

As it can be seen, the SELECT operator **mandatorily requires a Selection List** that defines the fields (i.e., the columns) to be returned and that will form the output table. In addition, the **query must also indicate where** (i.e., in which Tables) **the fields** listed in the SELECT section **must be searched**. This part corresponds to the <Table Expression> that is used, exactly, to define the sub set of the table on which the query must operate. Of this part, **only the FROM clause is mandatory**, all the others are optional. Specifically, its syntax is defined as follows:

```
<FROM Expression>  :: = FROM <Table name>
        [{, <Table Name>} …]
```

Practically speaking, **it is necessary to include in the FROM Expression at least the name of a single table**, i.e., the table from which data must be collected. Obviously, to create complex queries operating on more than a single Table, it will be necessary to include in the FROM Expression the name of all the tables that must considered during the search. As shown by the syntax, the names of the table must be separated by a comma.

In practice, the easiest form of a Select query consists in a list of records (columns) that belongs to the same Table as in the pseudo-code that follows:

```
<Selection List> :: =
        SELECT <First field> [{, <Second field}, …]

<Table Expression> :: =
        FROM <Input_Table>
```

It should be clear that, this query operates a projection of the Input_Table; indeed, all the records of the Input_Table will be returned, but only the fields that appear in the Selection List will be displayed.
Let us consider a simple CUSTOMERS table as the one in Fig. 4.1

CUSTOMERS
ID_Customer - Integer (PK)
Name – Char (252)
Surname – Char (252)
Date of Birth – Date
Nationality – Char (252)
Income – Currency

Fig. 4.1. *The CUSTOMERS Table*

The following query generates a new table with as many records as the original CUSTOMERS table, but with only three fields (per record).

```
SELECT Name, Surname, Nationality, Income
FROM CUSTOMERS
```

A possible example is shown below:

Tab. 4.2 (a)
A simple Selection query performed on the CUSTOMERS table

Name	Surname	Nationality	Income
Allison	Addison	British	€ 45,000
…	…	…	…
…	…	…	…
Zoe	Zuckemberg	Australian	€ 38,000

Additional operators

It is interesting to note that the order of the fields in the outcome table, depends, exclusively, on the order used in the Selection List.

It is also possible to **rename** the fields using the **AS operator**, as shown below:

```
SELECT  <Field name>  AS '<New Name>' [{,<Field Name> AS <new name>} …]
FROM <Table Name>
```

For instance, we could modify the previous query as follows:

```
SELECT Income AS Annual Income, Surname AS Last Name, Nationality
FROM CUSTOMERS
```

In this case the order of the fields has been altered and the first two fields have also been renamed. This is shown by table 4.1 (b).

Tab. 4.2 (b)
A simple Selection query with renamed field

Annual Income	Last Name	Nationality
€ 45,000	Addison	British
…	…	…
…	…	…
€ 38,000	Zuckemberg	Australian

Sometimes, it may be useful to select all the fields of a Table. In this case, instead of writing the name of each field in the Selection List, it is sufficient to use the **All operator** that is indicate with an asterisk **(*).**

The Select operator makes it possible not only to select some fields from a table, but it also permits **to create new Calculated Fields**. To this aims it is sufficient to include in the Selection List a simple mathematical expression operating on one or more fields of the selected tables.

To clarify both concepts, let us consider a PRODUCTS table as the one of Fig.2 where, OO stands for On-Order, OH for On-Hand and R is the reorder level (i.e., the level of the Inventory Position that triggers a new replenishment order). Note that the hypothesis is made that a single warehouse is used and so, data concerning the inventory can be conveniently placed in the PRODUCTS table.

Also note that there is a Forward Key, namely Category_ID, thus it should be clear that this table is in OTM relation with the "father" table CATEGORIES.

PRODUCTS
ID_Product - Integer (PK)
Name – Char (252)
Shelf Life – Integer
OO – Integer
OH – Integer
R - Integer
Price – Currency
Category_ID (FK) - Integer

Fig. 4.2. *The PRODUCTS Table*

It could be interesting to evaluate, for each product:

- The current Inventory Position (IP) = (OH + OO)
- The gap between the IP and the reorder point Gap = (IP - R)

In case of a negative gap a certain amount of product (greater or equal than the Gap) should be ordered.

To this aim we could write a query as the following one:

```
SELECT *, (OH + OO) AS IP, (OH + OO - R) AS Gap
FROM PRODUCTS
```

A possible outcome is shown below:

Tab. 4.3
A simple Selection query performed on the PRODUCTS table

Product_ID	Name	Shelf Life	OO	OH	R	Price	Category_ID	IP	Gap
1	Rose	5	0	100	50	10€	1	100	50
2	Mouse Tail	30	20	40	60	20€	2	60	0
3	Cactus	100	0	30	50	15€	3	30	-20
…	…	…	…	…	…	…		…	…
n	Violet	5	50	30	40	5€	1	80	40

As we have noted above, the table CATEGORIES has a OTM relation with the table PRODUCTS and so, it could be interesting to see how many CATEGORIES are used to categorize all the products contained in the PRODUCTS table. To this aim, we could make a projection on PRODUCTS using Category_ID as the only field included in the Selection List. This corresponds to the following simple query:

```
SELECT Category_ID
FROM PRODUCTS
```

However, the obtained result (an example is given in Tab. 4.5) is not satisfactory, since what we get is a list of ID_Category, but most of them appear more than one time. This is correct, since the relation is of the OTM form, and so the FK ID_Category is not unique.

Tab. 4.4 (a)
A Select query made on the FK

Category_ID
1
2
3
1
1
…
2
…
5

If we do not want duplicated data (i.e., we want the minimum set of the data contained in the FK field), we need to use **the DISTINCT operator**.

Please remember that the syntax of the SELECT operator is as follows:

```
<Select Command> : : =
        SELECT [ALL | DISTINCT] <Selection List>
        <Table Expression>
        [<Sorting Criteria>]
```

Aw we can see, after the keyword SELECT there are two operators (i.e., ALL and DISTINCT) divided by a vertical bar and they are included inside a pair of brackets. The vertical bar means that ALL and DISTINCT are mutually exclusive, the brackets means that them both are optional. Since ALL comes before DISTINCT, it is considered as the default option, in other words typing SELECT is analogous to type SELECT ALL.

What is the difference between SELECT ALL <*field name*> and SELECT DISTINCT <*field name*>? It is easy to guess that, with respect to selected field, using the **ALL** operator, all the records will be shown, using **DISTINCT**, only non-duplicated records will be shown. In other words, the DISTINCT operator is a first way to implement the Selection operation (of relational algebra).

Owing to these issues, we can reformulate the previous query as follows:

```
SELECT DISTINCT Category_ID
FROM PRODUCTS
```

A possible outcome is shown below:

Tab. 4.4 (a)
A Select query with the DISTINCT operator made on the FK

Category_ID
1
2
3
4
5

It is now clear that products have been grouped in five distinct categories.

We conclude by observing that, and this is obvious, the DISTINCT operator can be used only once in a SELECT statement, i.e., DISTINCT can be applied to a single field only.

For instance, the following query would generate an error:

```
SELECT DISTINCT Name, DISTINCT Category_ID
FROM PRODUCTS
```

3. The Where Operator

The **WHERE** operator makes it possible **to specify one or more filtering conditions** that will affect the number of the records returned by a Selection query. In other words, the WHERE operator implements the Selection (or Restriction) operator of the relational algebra.[1]

As above mentioned also DISTINCT performs a Selection, but WHERE is much more powerful and flexible. Its syntax is the following one:

```
<Where Statement> :: = WHERE <condition>
```

Note that <condition> can be any (valid) logical condition obtained through comparison operators, logical operators and any other conditional connectors.

Comparison Operators and Logical connectors

Standard comparison operators (see Table 4.5) can be used to build complex conditions in which the value of a field will be compared with a constant value and/or with the value of one or more fields.

Tab. 4.5.
Comparison Operators

Meaning	Symbol
Equal	=
Different	<>
Greater	>
Less	<
Greater or equal	>=
Less of equal	<=

To make a first example we can consider Table 4.3 again. Now we want to see only the products that need a replenishment order (i.e., those one with a negative gap).

[1] It may seem confusing but the SELECT operator performs a 'projection', whereas it is the WHERE operator that performs a 'selection'.

To this aim we could reformulate the query as follows:

```
SELECT Name, (OH + OO) AS IP, (R - OH - OO ) AS Reorder Quantity
FROM PRODUCTS
WHERE (OH + OO - R) <= 0
```

A possible outcome is shown below:

Tab. 4.6
Products that should be re-ordered

Name	IP	Reorder Quantity
Mouse Tail	60	0
Cactus	30	20
…	…	…
…	…	…

Let us make a step further. Suppose that the PRODUCTS table has an additional field called Minimum Ordering Lot (MOL), which is the minimum quantity that can be ordered to a supplier. Also, orders must be integer multiple of the MOL quantity, so, for instance, if the gap was -15 and MOL was 10, then an order equal to 20 should be issued.

If so the reorder quantity calculated above is wrong, thus we should correct it. This time the calculation is a little harder, so it may be wise to create a custom-made function, for example using VBA or another language, and to call that function directly in the Selection List. **Public functions, indeed, can be directly used in inside SQL statements.** This is a very important feature!

As a first thing let us see how the function that we need looks like:

```
Public Function RQ (OO, OH, R, MOL As Integer) As Integer
Dim IP, Gap, Q, N As Integer ' Definition of internal variables
  IP = (OO + OH)
  Gap = (IP - R)
  If Gap < 0 Then ' An order is needed
    Gap = - (Gap) ' This is the theoretical quantity to be reorder
    N = Ceiling(Gap / MOL) ' A custom function that rounds up a number
  End If
  RQ = (MOL * N) ' The output
End Function
```

```
Private Function Ceiling(N As Double) As Integer
Dim I As Integer
    Ceiling = N
    If N <> Int(N) Then Ceiling = Int(N) + 1 'Int is a predefined function that returns the integer part of a number[2]
End Function
```

The function takes four input parameters (all integer) and returns, as output, an integer i.e., the Real Quantity to be ordered. At first, the Inventory Position and the Gap are computed. Then, a *If … Then … Else* statement is used to see if an order is needed, that is if the Gap is negative. If so, after changing the sign of the gap, the theoretical number of lots is computed as (Gap/Mol). Since we need to round this quantity to the superior integer, the Ceiling function is invoked. Lastly, the total quantity to be ordered is simply obtained as the product of the number of lots and the dimension of a lot i.e., (MOL*Q).

Once we have created the function, calling it inside the query is straightforward. To this it is sufficient to reformulate the query as follows, where the input parameters must be fields of the table(s) included in the FROM clause of the query:

```
SELECT Name, (OH + OO) AS IP, (R - OH - OO ) AS [Theoretical Reorder Quantity] _
                                                  RQ(OO, OH, R, MOL) AS [Real Reorder Quantity]
FROM PRODUCTS
WHERE (OH + OO - R) <= 0
```

Obviously, it is also possible to build complex logical condition, by aggregating simple condition by means of logical connectors (i.e., AND, OR and NOT), accordingly to the well-known De Morgan's laws[3].

Let us consider a couple of examples to clarify these concepts.

```
SELECT Surname
FROM CUSTOMERS
WHERE NOT (Country = 'USA' ) AND City = 'Vancouver'
```

This query returns the last name of the customers (stored in the CUSTMERS table) that do not leave in the USA and that reside in Vancouver[4]. Note that, in this case the NOT condition is almost redundant. Vancouver is a

[2] *All the VBA functions available in Access can be found at: https://support.office.com/en-us/article/Access-Functions-by-category-B8B136C3-2716-4D39-94A2-658CE330ED83*

[3] De Morgan's laws can be found at: *https://en.wikipedia.org/wiki/De_Morgan%27s_laws*

[4] *Note that the single quotation mark ' ' are used to define a String*

Canadian city and so, it is clear that people leaving in Vancouver are not USA citizens. However, since there is a city named Vancouver in the Washington State, the NOT condition may filter some data.

Now, what happen if we modify the query as follows?

```
SELECT Surname
FROM CUSTOMERS
WHERE NOT (Country = 'USA' AND City = 'Vancouver')
```

In this case the scenario is totally different. Since the NOT condition applies to both statements, this time the filter is almost useless. Indeed, the query will return all the customers, except those few ones living in Vancouver in the Washington State.

Using the De Morgan's laws we could have written the previous query also in this alternative way:

```
SELECT Surname
FROM CUSTOMERS
WHERE NOT (Country = 'USA') OR NOT (City = 'Vancouver')
```

A last example follows:

```
SELECT Surname
FROM CUSTOMERS
WHERE NOT (Country = 'USA' OR City = 'Vancouver')
```

In this case the query will return all the non-American customers that do not live in Vancouver. Again, using the De Morgan's laws the query can be written also in the following way:

```
SELECT Surname
FROM CUSTOMERS
WHERE NOT (Country = 'USA') AND NOT (City = 'Vancouver')
```

Special comparison Operators

There are other comparison operators of frequent use. These are:

- **BETWEEN**
- **IN**
- **LIKE**
- **IS NULL**

The **BETWEEN** operator is used to check if a value belongs to a specific interval. Its syntax is the following one:

```
<field> BETWEEN <low value> AND <high value>
```

The **IN** operator is used to check if a value belongs to a list/set of values. Its syntax is the following one:

```
<field> IN ({<'first value of the sequence'>, } …)
```

The **LIKE** operator is used to check if a string has a predefined format; the format is defined using jolly or wildcard chars. Its syntax is the following one:

```
<field> LIKE <wildcard sequence>
```

The IS NULL operator is used to check if a field is null. IS NULL is commonly used together with the NOT operator (i.e., NOT IS NULL), so as to return only the records with non-null fields.

Some examples follow.

The first one concerns the use of the IN operator.

```
SELECT Surname
FROM CUSTOMERS
WHERE City IN ('Vancouver', 'New York', 'Chicago') AND Country = 'USA'
```

In this case the query returns the American customers that live in one of the cities of the list.

Also note that an OR condition could be used instead of the IN operator:

```
SELECT Surname
FROM CUSTOMERS
WHERE (City = 'Vancouver' OR City = 'New York' OR City = 'Chicago') AND Country = 'USA'
```

The second example concerns the use of the BETWEEN Operator:

```
SELECT Surname
FROM CUSTOMERS
WHERE Date_Of_Birth BETWEEN DateAdd("yyyy", -19, Date()) AND DateAdd("yyyy", -13, Date())
```

This query may look odd, but is just returns all the customers that are teenagers, i.e., customers aged between 13 and 19 years old. Specifically, **DateAdd()** is a very useful function that allows one to add or to subtract a certain time interval to a specified date. This function takes three values as input: the standard time units (seconds, minutes, …, years), the time interval to be added or subtracted (if negative) and the reference date. In the query we typed: “yyyy” to indicate that we are measuring time in years, -19 to indicate that we want to subtract 19 years from the reference date, and Date() as the reference date. Note that **Date()** is another function that returns the current date. So, DateAdd("yyyy", -19, Date()) returns the date of 19 years ago and, similarly, DateAdd("yyyy", -13, Date()) returns the date of 13 years ago. As a results, the BETWEEN condition will remove all the customers that have less than 13 years and more than 19 years (i.e., those ones that are not teenager).

The third example concerns the use of the LIKE operator:

```
SELECT Surname
FROM CUSTOMERS
WHERE City LIKE '[abc]*' AND Country = 'USA'
```

In this case the query returns all the customers that are American citizen and that live in a city with a name composed of any number of characters of which the first is 'a' or 'b' or 'c'. For instance, Chicago would be fine, but New York would be not.

This is because:

- chars included in brackets represent a list of chars that are allowed;
- the asterisks indicate a limitless series of characters.

These conventions are summarized in Table 4.7.

Tab. 4.7.
Wildcard operators of the LIKE operator

Name	SQL Symbol	ACCESS Symbol	Meaning
Underscore	**_**	**?**	A single char
Hashtag	**#**	**#**	A single digit (number)
Percentage	**%**	*****	A (limitless) sequence of chars
List	**[charlist]**	**[charlist]**	One of the chars of the charlist must be found in the specified position of the input string
Not in List	**[^charlist]**	**[!charlist]**	None of the chars of the charlist must be found in the specified position of the input string

For instance, LIKE 'c[h-m]?????' requires a string of seven chars starting with the letter 'c', followed by a letter in the range [h-m]. Also in this case, Chicago would be fine, but Chelsea would be not.

Although rarely, it may happen that the input string (to be analyzed) contains the underscore ( ) or the percentage (%) or the hashtag (#). Let us suppose that some customers have been coded using one char for the name, three chars for the surname and two chars for the nationality, and that the hashtag has been used to separate these chars, as for F#ZMM#IT. What should we write if we wanted to identify all the Italian customers (i.e., those ones having IT as the last two chars of their code)? The solution is shown below:

```
WHERE Code LIKE '?$#???$#IT' ESCAPE '$'
```

The dollar ($) is called Escape character and it is used to say to the compiler that a wildcard char (in this case the hashtag) has to be considered as a standard char. More precisely any wildcard that is preceded by the escape char will be considered as a normal char and not as a wildcard one. So, in this case the sequence '?$#???$#IT' indicates a string of eight letters (obviously the two escape characters are not counted) having the following structure: the first char is a letter, the second is an hashtag, the following three chars are letters, the sixth char is an hashtag and the last two chars are IT. Please note that the escape chars must not be a wildcard and it must not be part of the string to be analysed (exactly as the dollar $, in the present case).

4. Functions and Variables that can be included in VBA

As we have seen it is possible to include public functions as part of SQL statements. What about public variables? Unfortunately, this is not possible and global variables cannot be included, directly in a SQL statement.

Let us consider a simple example: we have a table, named REGISTRY, corresponding to a customer registry and we want to create and save a query that returns data relating to a specific customer.

This query will look like this one:

```
SELECT Name, Surname, [...]
FROM REGISTRY
WHERE ID = [Enter ID]
```

However, formulated in this way, this is a parametric query and, therefore, when executed, the user will be asked to fill in an input form, to specify the ID value to be searched.

To avoid this problem, it would be nice to modify the WHERE clause as follows:

```
WHERE ID = I
```

with 'I' being a public variable evaluated (at run time) by VBA code. Unfortunately, as above mentioned. even if 'I' was a global variable, this query would not work. In fact, a query can only receive public functions, not public variables. To overcome this fact, we should then write (in a public module) a trivial function of type Get that reads and returns the public variable 'I'.

```
Public Function Get_ID() As Integer
        Get_ID = I 'It reads and return the value of the public variable I
End Function
```

In this way, we can now modify our query as follows:

```
SELECT Name, Surname, [...]
FROM REGISTRY
WHERE ID = Get:ID()
```

For some reasons, although public variables are not allowed inside SQL code, references to the values of some Components placed on a Form are admitted. For instance, suppose that the user can operate on a Form (named F_Id) with a Combo Box (named Cbx_Id) populated with all the customers' ID from the REGISTRY table.

In this case, the previous query could have also been written as:

```
SELECT Name, Surname, [...]
FROM REGISTRY
WHERE ID = Forms(“F_Id”).Controls(“Cbx_Id).Value
```

where: Forms and Controls are the collections containing all the forms of the DB and all the controls placed on a specific form.

Another interesting possibility is the use of **the TempVars()** collection. This collection (indexed from zero) is automatically created by the application and is immediately ready for use, without having to create it either with a Dim instruction or, subsequently, with a Set statement.

This collection serves to contain public variables (that can be shared among forms), which can be managed through the following properties and the following methods:

- TempVars.Add (Variable_name, Value)
- TempVars(Item), with both positional and string items
- TempVars.Remove (variable_name)
- TempVars.RemoveAll
- TempVars.Count

So, for example if we had created a TemVar with the following code:

```
TempVars("Customer_Id").Value = 100
' Which is equivalent to:
TempVars.Add "Customer_Id", 100
```

We could rewrite the previous query as it follows:

```
SELECT Name, Surname, [...]
FROM REGISTRY
WHERE ID = TempVars(“Customer_Id”) ' Or TempVars!Customer_Id
```

5. Functions operating on groups

As we have seen above, it is possible to include simple mathematic computation inside the Selection List of a query. It is also possible to use functions operating on groups of records and that returns, as a result, a single value (i.e., a scalar value). The main functions operating on groups are the following ones:

- **MAX** - It returns the maximum value of a set of records
- **MIN** - It returns the minimum value of a set of records
- **AVG** - It returns the average value of a set of records
- **COUNT** - It returns the total number of records that satisfy a certain condition
- **SUM** - It returns the sum of the values of a set of records

For instance, if we wanted to find the minimum unit price among the products of the "Herbs" category we would have to write the following query[5]:

```
SELECT MIN([Unit Price])
FROM PRODUCTS
WHERE [Category ID] = 'Herbs'
```

The above-mentioned query returns the minimum unit price. But which is the product that has this price? In order to answer this question, we could be tempted to write something like that:

```
SELECT Name, MIN([Unit Price])
FROM PRODUCTS
WHERE [Category ID] = 'Herbs'
```

Or:

```
SELECT Name
FROM PRODUCTS
WHERE [Category ID] = 'Herbs' AND [Unit Price] = MIN([Unit Price])
```

Unfortunately, none of the two queries does work. Indeed, it is not possible to include in the same selection list elements that operates on a single record (the Name field in this case) and functions that operates on group of records (the MIN operator in this case). Indeed, this create an incompatibility error.

Similarly, it is not possible to include a function operating on groups in the WHERE clause; indeed, also this would generate an incompatibility error.

[5] *If the name of a field is made of two or more words separated by a space, the whole string must be placed in brackets*

For the sake of completeness, we anticipate that the only possible solution is that to use a combined query as the following one[6]:

```
SELECT Name
FROM PRODUCTS
WHERE [Unit Price] = (SELECT MIN[Unit Price] FROM PRODUCTS WHERE [Category ID] = 'Herbs')
```

The inner query (the one used in the WHERE clause of the outer query) returns a scalar value which is then used as comparison value of the outer query. This is absolutely licit, and the query does work.

Distinct, Group By and Having operators

It is often useful to combine the group operating functions with the **DISTINCT** and or with the **GROUP By** operator.

This is because, when we use a group operating function, by default, the compiler associates it to the ALL (*) operator. In other words, by default, the compiler applies the group operating function to all the records of a table. For instance, writing COUNT ([Unit Price]) we get, as result, the number of unit prices listed in the PRODUCTS table. What if some values are null? Fortunately, these values are not counted and so the total amount that is returned is correct. But what if we wanted to know how many different prices are there? In this case a straightforward use of COUNT ([Unit Price]) would be wrong as repeated values would be counted as well. To solve this problem, it is sufficient to use the DISTINCT operator as shown below:

```
SELECT DISTINCT COUNT([Unit Price])
FROM PRODUCTS
```

A problem that occurs more frequently is the following one. Let us consider the previous example, but this time we want to know the minimum price for each category. How to do so? A first idea could be that to repeat the same query changing anytime the name of the category used in the WHERE clause:

```
SELECT MIN([Unit Price])
FROM PRODUCTS
WHERE [Category ID] = <Insert a category here>
```

However, this approach would be tedious and time consuming.

Fortunately, a solution is offered by the use of the GROUP BY operator that has the following syntax:

```
<GROUP BY clause> :: = GROUP BY <field name>
```

[6] *This topic will be better explained in the following chapters*

So, the query has to be rewritten as follows:

```
SELECT [Category ID], MIN([Unit Price]) AS Min_Price
FROM PRODUCTS
WHERE [Category ID] = <Insert a category here>
GROUP BY [Category ID]
ORDER BY [Category ID] ASC
```

In this case the MIN operator is not applied to all the records of the PRODUCTS table, but rather it is applied, sequentially, to each record belonging to a specific category (i.e., in this case the minimum is evaluated considering all the records that have a common value in the [Category ID] field).

A possible result is shown in Table 4.8 where, for each category, the minimum unit price is displayed.

Tab. 4.8
An example of the Group By operator

Category ID	**Min_Price**
1	10€
2	5€
…	…
…	…
N	15€

It is important to note that, if a group operating function is d in the Selection list of a query, then, all the other elements included in the Selection list must be part of the GROUP BY list, too. For instance, in the query above, [Category ID] appear both in the Selection and in the Group By list.

Also note that, in the query, an additional operator has been used. This is the ORDER BY operator that is used to sort (in Ascending ASC or Descending DES order) the records returned by a query; its syntax is as follows:

<Ordering Statement> : : = ORDER BY *<Ordering condition* > [{, *<other condition>*} …]

where:

<Ordering Condition> : : = *<Field name>* | *<Column Number>* [ASC|DESC]

Briefly, it is sufficient to indicate the field or the fields on which the ordering have to be based. Fields can be indicated by name or, making reference to their position in the table (i.e., column number). The field name is the standard option. Also, there is the need to indicate if data have to be sorted in Ascending or in Descending order. Ascending is the default option, even if ASC is not explicitly indicated, it is considered as the standard way to order data.

For instance, if we write:

```
SELECT Surname, Region, City
FROM CUSTOMERS
WHERE Country = 'Canada'
ORDER BY Region ASC, City DES
```

Customer will be ordered (in ascending order) for Region of origin first, and for city of residence (in descending order), next.

HAVING is the last interesting operator to be described. If used conjointly with the GROUP BY operator, **HAVING** makes it possible to add logical conditions on the group operating functions, exactly as the WHERE operator makes it possible to define logical condition on the function that operates on single records.

To make a simple example, we can consider the following condition. We want to know how many products belong to each category, but we want to limit this analysis to the categories that include at least five different products. This is a typical case requiring the use of the having condition; the query that we need is, in fact, the following one:

```
SELECT CategoryID, COUNT(ProductsID) AS #Prod
FROM PRODUCTS
GROUP BY CategoryID
HAVING COUNT(ProductsID) >=5
```

A possible result is as follows:

Tab. 4.9
An example of the Having operator

Category ID	#Prod
1	6
5	5
6	15
…	…
N	7

Note that not all the Category ID are shown in the table; indeed, only the categories that comprehend more than five products are returned by the query.

Also note that the **GROUP BY** operator must always operate on a group operating function as in the present case where, in fact, we wrote: HAVING COUNT(ProductsID) >=5.

Conversely, as we have already noted, a group operating function must never be included in the WHERE clause. For instance, that following query is wrong and does not work:

```
SELECT CategoryID, COUNT(ProductsID) AS #Prod
FROM PRODUCTS
GROUP BY CategoryID
WHERE COUNT(ProductsID) >=5
```

It is also worth noting that the HAVING operator can be based on any one of the comparison operators (i.e., LIKE, IN and BETWEEN). Also, and perhaps more important, **it is possible to use** in the same query **both the WHERE and the HAVING operators**:

- The WHERE clause is executed first and, in this way, all the records that do not fulfill the logical condition (included in the WHERE clause) are eliminated.
- The HAVING operator acts on the remaining records, next.

For instance, let us consider the following query:

```
SELECT CategoryID, COUNT(ProductsID) AS #Prod
FROM PRODUCTS
WHERE CategoryID <= 6
GROUP BY CategoryID
HAVING COUNT(ProductsID) BETWEEN 5 AND 12
```

In this case we get the following outcome:

Tab. 4.10
Having operator used conjointly with WHERE

Category ID	#Prod
1	6
5	5

As it can be seen, only two records are returned. Indeed, all the records with a Category ID greater than six are erased by the WHERE condition. Next, the HAVING operator limits the analysis to those categories with a number of products comprised between 5 and 12. So, since the 6th category has 15 products it is also erased.