The SQL Language Part III - Sub-queries and Domain Functions[1]

1. Introduction

A sub-query is a SELECT command, written inside parentheses, that is included inside another SELECT command. More precisely, a **sub-query is an "inner" query, executed inside an "outer" query, which returns a table on which the outer query can operate.**

The syntax is very simple:

```
<sub-query> :: = (<select command>)
```

Before proceeding further on, we also note that, generally, sub-queries are typically used in:

- *Comparison expressions;*
- *Quantified comparison expressions;*
- *Expressions that use the IN operator;*
- *Expressions that use the EXIST operator;*
- *Expressions that are based on complex functions operating on groups.*

2. Type of Sub-queries

There are three different types of sub-queries. These are: *(i) scalar (ii) column and (iii) table subqueries*[2].

A scalar sub-query is **the easiest form of sub-queries. In this case the sub-query returns a single value** (i.e., a scalar value) that, subsequently, is **used as the input of the outer query**.

Let us consider a couple of example.

```
(SELECT MAX (UnitPrice)
FROM PRODUCTS)
```

This query returns a single value, i.e., the maximum price among all available products. Since the value is unique, this is a scalar sub-query.

A second example follows:

```
(SELECT Surname
FROM CUSTOMER
WHERE CustomerID = 1)
```

Also this query returns a single value, i.e., the surname of the costumer whose ID = 1. Since ID is a unique field also the returned value is unique.

[1] *Some of the examples of this chapter can be found in the* ***DFunctions.accdb*** *file*
[2] *The use of table query is rather rare*

A **column subquery is an inner query that returns a column vector** (i.e., a set of mono-dimensional records). An example follows:

```
(SELECT UnitPrice
FROM PRODUCTS)
```

In this case only the UnitPrice field (of all the records belonging to the PRODUCTS tables) is returned.

Finally, a **table sub-query is an inner query that returns a table** (i.e., a set of multi-dimensional records).

3. Using Sub-queries

To show when sub-queries should be used we will make reference to the PRODUCTS and CATEGORIES tables, in OTM relation, shown in Figure 6.1.

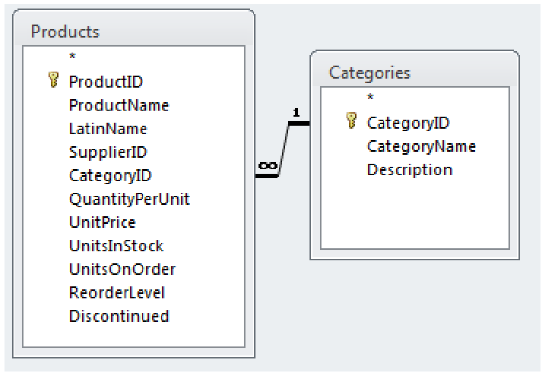


Fig. 6.1. *The PRODUCTS and CATEGORIS tables*

To begin with, let us consider a simple problem: “we want to know which products belong to the Roses category”. A first solution is straightforward, as it is sufficient to use a simple join performed on PRODUCTS and CATEGORIES, as shown below:

```
SELECT PRODUCTS.ProductName
FROM PRODUCTS INNER JOIN CATEGORIES ON PRODUCTS.CategoryId = CATEGORIES.CategoryId
WHERE CATEGORIES.CategoryName = 'Roses'
```

Howeve,r the same result could also be obtained using a scalar sub-query, as shown below:

```
SELECT ProductName
FROM PRODUCTS
WHERE CategoryId = (SELECT CategoryId FROM CATEGORIES WHERE CategoryName = 'Roses')
```

The underlying logic of the SQL code is as follows:

- The inner query (i.e., the sub-query) is executed first;
- The output of this scalar sub-query is the ID of the Roses category;
- This scalar value is passed as the input of the WHERE condition of the outer query;

- In this way the list of all products contained in the PRODUCTS table is filtered, and only the products that belong to the Roses category are retuned in the output table.

Generally speaking a **sub-query included in the WHERE clause of an outer query must fulfill some constraints**:

- The sub-query must be a scalar sub-query;
- The sub-query must be inserted after a comparison operator
- It is not possible to compare the outputs of two sub-queries
- The operator GROUP BY and HAVING cannot be used in the sub-query; however they can be used in the outer query.

Now we are ready to consider a more interesting example; specifically we want to find all the products having a price higher than the average price. To solve this query we cannot write something like:

```
SELECT *
FROM PRODUCST
WHERE UnitPrice > AVG(UnitPrice)
```

This query does not work because a group operating function (i.e., AVG) has been included in the WHERE clause. Thus, to correct the mistake, we need replacing the AVG operator with a scalar sub-query of the form:

```
(SELECT AVG(UnitPrice)
FROM PRODUCTS)
```

Thus the overall query becomes as follows:

```
SELECT *
FROM PRODUCST
WHERE UnitPrice > (SELECT AVG(UnitPrice) FROM PRODUCTS)
```

Let us consider a second example a little bit more complicated. This time we want to find all the information of the most expensive products. We also want to retrieve the information of the category to which the most expensive products belongs to. To this aim we have to write the following SQL based on a concatenated query:

```
SELECT *
FROM PRODUCTS INNER JOIN CATEGORIES ON PRODUCTS.CategoryID = CATEGORIES.CategoryID
WHERE PRODUCTS.UnitPrice = (SELECT MAX(UnitPrice) FROM PRODUCTS)
```

Things would get trickier, if we wanted to see, in the same table, the most expensive product of each category. Indeed, as mentioned above, it is not possible to use neither the GROUP BY nor the HAVING operator in a subquery.

For these reasons, the easiest way to solve the problem is that to combine two queries, Q_1 and Q_2, as follows:

Q_1:

```
SELECT CategoryID, CategoryName,ProductName, UnitPrice
FROM CATEGORIES INNER JOIN PRODUCTS ON Categories.CategoryID = Products.CategoryID
```

Q_2:

```
SELECT CategoryID, Max(UnitPrice) AS [Max Price]
FROM PRODUCTS
GROUP BY CategoryID
```

Note that Q_1 returns the list of all products with an indication of their unit price and of their category.

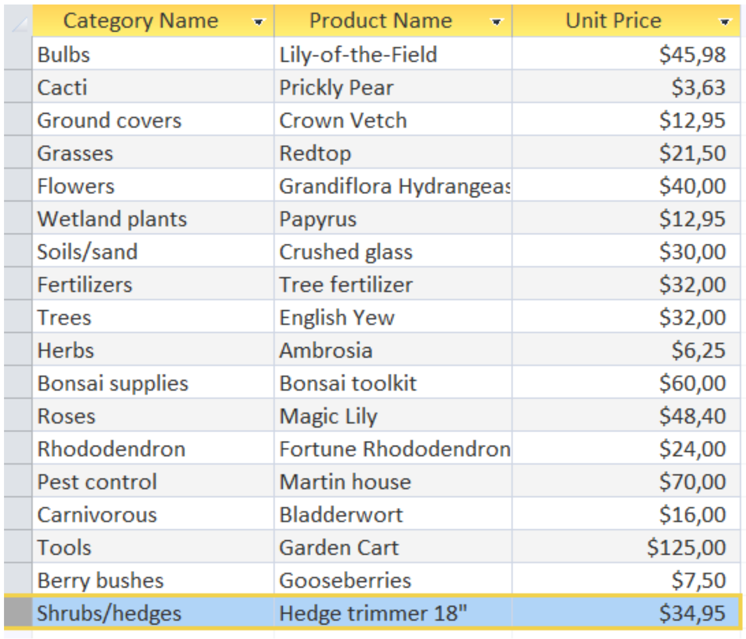
Q_2 returns the list of the ID of each category, with an indication of the maximum unit price of each category.

Lastly, to get the desired result we have to write a third query operating on the previous ones:

```
SELECT Q₁.CategoryName, Q₂.ProductName, Q₂.UnitPrice
FROM Q₁ INNER JOIN Q₂ ON Q₁.CategoryID = Q₂.CategoryID
WHERE Q₁.UnitPrice=Q₂.[Max Price]
```

In this way data are aggregated and filtered using a Join condition (made on the CateoryID field) and a logical condition (based on the UnitPrice field). A possible outcome is shown in Table 6.1:

Tab. 6.1. *A complex query*



Category Name	Product Name	Unit Price
Bulbs	Lily-of-the-Field	$45,98
Cacti	Prickly Pear	$3,63
Ground covers	Crown Vetch	$12,95
Grasses	Redtop	$21,50
Flowers	Grandiflora Hydrangeas	$40,00
Wetland plants	Papyrus	$12,95
Soils/sand	Crushed glass	$30,00
Fertilizers	Tree fertilizer	$32,00
Trees	English Yew	$32,00
Herbs	Ambrosia	$6,25
Bonsai supplies	Bonsai toolkit	$60,00
Roses	Magic Lily	$48,40
Rhododendron	Fortune Rhododendron	$24,00
Pest control	Martin house	$70,00
Carnivorous	Bladderwort	$16,00
Tools	Garden Cart	$125,00
Berry bushes	Gooseberries	$7,50
Shrubs/hedges	Hedge trimmer 18"	$34,95

4. ALL, ANY, IN and EXSIST operators

Up to now we have discussed, only, the use of scalar queries. However it is also possible to use a **set of quantified comparison operators that make it possible to base the comparison on the values of a whole column of fields**. The basic syntax is as follows:

```
<quantified comparison operator> :: =
<comparison operator> <ALL |ANY >
```

where:

```
<comparison operator> :: =
= | <> | < | > | <= | >=
```

Let C be the column vector of the comparison values. **Then if ANY is used, then the comparison returns true if there is at least one value of C that fulfill the comparison condition. Conversely, if ALL is used, then the comparison returns true if and only if all the values of C fulfill the comparison condition**.

For instance if $C \equiv \{1; 2; 3; 4; 5\}$ and the field $c = 3$, then:

- $c \geq$ ANY(C) is true;
- $c \geq$ ALL(C) is false.

Let us consider an example: we want to find all the carnivorous plants (if any) that are more expensive than the cheapest rose in the products catalogue.

To get this result we should write the following query:

```
SELECT PRODUCTS.ProductName, PRODUCTS.UnitPrice
FROM PRODUCTS INNER JOIN CATEGORIES ON PRODUCTS.CategoryId = CATEGORIES.CategoryId
WHERE CATEGORIES.CategoryName = "Carnivorous" AND PRODUCTS.UnitPrice > ANY _
        (SELECT PRODUCTS.UnitPrice
         FROM PRODUCTS INNER JOIN CATEGORIES ON PRODUCTS.CategoryId = CATEGORIES.CategoryId
         WHERE CATEGORIES.CategoryName = "Roses")
```

The inner query returns the unit price of all the products that are categorized as Roses. The outer query returns all the products that are classified as Carnivorous Plant. Also, since the inner query is used in the WHERE clause (of the outer query) and the ANY operator is used too, only those carnivorous plants, that cost more than the cheapest rose, are returned.

It is worth noting that, if we knew that Roses' CategoryId = 16, the same result could be obtained in a much simpler way, by means of the **MIN** operator:

```
SELECT PRODUCTS.ProductName, PRODUCTS.UnitPrice
FROM PRODUCTS
WHERE CategoryName = "Carnivorous" AND UnitPrice > _
                    (SELECT MIN(UnitPrice)
                     FROM PRODUCTS
                     WHERE CategoryID = 16)
```

Two expressions of frequent use are:

- **= ANY** - This expression is similar to the **INTERSECT** operator;
- **<> ALL** - This expression is similar to the **EXCEPT** operator.

To clarify this concept let us consider two tables A and B, respectively and the following query:

```
SELECT *
FROM A
WHERE A.field[i] = ANY (SELECT B.field[j] FROM B)
```

In this case, a record of table A is returned if the value of its i-th field matches the value of the j-th field of one or more records of B.

Conversely, if we had written:

```
SELECT *
FROM A
WHERE A.field[i] <> ALL (SELECT B.field[j] FROM B)
```

Then, only those records of A that do not match any records of B (relatively to the i-th and to the j-th field, respectively), would have been returned.

As an alternative, **instead of ANY and ALL is it also possible to use the IN operator**. More precisely:

- *IN is equivalent to = ANY*
- *NOT(IN) is equivalent to <> ALL*

EXIST is a last important operator, as it allows one to define logical condition operating even on a whole record and not only on a part of it. In this case the sub-query following the EXISTS operator can be a column but also a table sub-query; more specifically, it syntax is as follows:

```
< EXIST condition> : : = [NOT] EXISTS <subquery>
```

Specifically, the subquery must be a SELECT statement. If the subquery returns at least one record in its result set, the EXISTS clause will evaluate to true and the EXISTS condition will be met. If the subquery does not return any records, the EXISTS clause will evaluate to false and the EXISTS condition will not be met.

For instance, if we wanted to find all the customers that issued an order after a certain date, the following concatenated query could be used:

```
SELECT Name, Surname
FROM Customers
WHERE EXISTS
        (SELECT Orders.OrderId, Orders.CustomerID
         FROM Orders
         WHERE Orders.OrderDate>#1/1/2012# AND Orders.CustomerID = Customers.CustomerID)
```

The outer query returns the name and the surname of all the customers of the CUSTOMERS table; the inner one returns the ID of the orders issued after a certain data and the ID of the customers who made that orders.

An additional condition, used by the EXIT operator is introduced, too;

```
Orders.CustomerID = Customers.CustomerID
```

Specifically, for each teneric record '*i*' returned by the outer query, the CustomerID is taken and passed to this condition. If the condition is satisfied then the *i*-th record will be kept, otherwise it will be discarded.

For this reason, SQL statements that use the EXISTS condition are very inefficient since the sub-query is rerun for every row in the outer query's table.

There are more efficient ways to write most queries, that do not use the EXISTS condition.

For instance, in this case the following simpler query should have been used:

```
SELECT DISTINCT Name, Surname
FROM Orders INNER JOIN Customers ON Orders.CustomerId = Customers.CustomerId
WHERE Orders.OrderDate>#1/1/2012#
```

Note the use of the **DISTINCT** condition. Could you guess why it is needed in this case?

Similarly, to find all categories that do not have any products we could use the following query:

```
SELECT CategoryName
FROM CATEGORIES
WHERE NOT EXISTS (SELECT * FROM PRODUCTS WHERE PRODUCTS.CategoryId = CATEGORIES.CategoryId)
```

Also in this case a standard select query based on a left join and on the use of the IsNull function can be used to obtain the same result.

```
SELECT Categories.CategoryName, Products.ProductID
FROM Categories LEFT JOIN Products ON Categories.CategoryID = Products.CategoryID
WHERE IsNull(ProductID)
```

5. Concatenated Sub-queries and calculated fields

Up to know we have considered the use of concatenated queries only inside the WHERE clause. However, **concatenated query can be used also in the SELECT statement of an SQL query**.

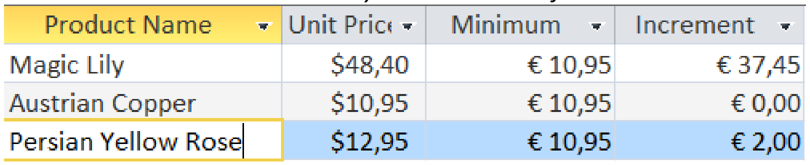
This is very useful when there is the need to perform "complex" algebraic operation on the data.

Let us consider the following example: we want to compute the difference of price between each rose and the cheapest rose in the catalogue. To this aim we could use the following concatenated query:

```
SELECT ProductName, UnitPrice, UnitPrice - (SELECT MIN(UnitPrice) FROM PRODUCTS INNER JOIN CATEGORIES
ON PRODUCTS.CategoryID = CATEGORIES.CategoryID
WHERE CATEGORIES.CategoryName = 'Roses') AS Increment
FROM PRODUCTS INNER JOIN CATEGORIES ON PRODUCTS.CategoryID = CATEGORIES.CategoryID
WHERE CATEGORIES.CategoryName = 'Roses'
```

A possible outcome is shown below:

Tab. 6.2. *Query with calculated field*



Product Name	Unit Pric	Minimum	Increment
Magic Lily	$48,40	€ 10,95	€ 37,45
Austrian Copper	$10,95	€ 10,95	€ 0,00
Persian Yellow Rose	$12,95	€ 10,95	€ 2,00

6. Domain Aggregate Functions

Domain Aggregate Functions are useful functions that mimic selection queries and, as such, they make it possible to find relevant data in a relational DB, without writing specific SQL code. However, this simplification does not come for free; a **domain function is much slower than the corresponding standard SQL** query and so, an excessive use of domain functions may reduce the performance of a DB.

Specifically, the generic syntax of a domain function is as follows:

```
DFunction_Name(Expression, Domain, [Criterion])
```

where:

- **DFunction_Name** - The name of a specific Dfunction;
- **Expression** - A mandatory input parameter, which contains the list of the fields (or even a calculated field) that must be returned as output;
- **Domain** - A mandatory input parameter that specifies the domain (i.e., the table or the query) on which the search must be made;
- **Criterion** - An optional input value that specifies a logical criterion used to filter the data.

In other words, making a parallel with a standard Select query written in SQL, we can write the following correspondence table:

Tab. 6.3. *SQL and DFunction*

SQL		**DFunction**
SELECT	Correspond To →	Expression
FROM		Domain
WHERE/HAVING		Criterion

Another interesting feature is that **Domain Functions can be called “anywhere”**, that is inside a query, inside VBA code, and even in the Text_Boxes of a Form[3].

Dlookup and other useful Dfunctions

A Dlookup performs a simple search and, as such it is very similar to a standard Select query written in SQL. Although this is improper, we could say that a Dlookup is a compact form of select query.

A simple example may be helpful to clarify this concept:

```
DLookup("Surname", "EMPLOYEES", "ID = 1")
```

This function operates on the EMPLOYEES table and returns the value of the Surname field of the (unique) record with an ID equal to 1.

[3] A couple of examples can be found at the end of this chapter

As it should be evident, this Dlookup corresponds to the following (scalar) query:

```
SELECT Surname
FROM EMPLOYEES
WHERE EmployeeID = 1
```

In this case the result is a scalar, as the filter is made on the primary key; anyhow, when there are several records satisfying the filtering condition, it is possible to get the first or the last one of them, by substituting DlookUp with **DFirst** and **DLast**, respectively.

There are also useful Dfunctions that operate on groups and that can be efficiently used when many records satisfy the filtering condition. A list of the main groups-operating Dfunctions, is shown below, where the returned value is relative to the records of the original Table that satisfy the filtering condition.

Tab. 6.4. *Main DFunction operating on groups*

Name	Returned Value
DMax	Maximum value of the "Expression" field
DMin	Minimum value of the "Expression" field
DCount	Number of returned records
DSum	Sum of the value of the "Expression" field
DAvg	Avearate of the value of the "Expression" field
DStd	Standard deviation of the "Expression" field. Returned value are considered as the whole universe
DStdP	Standard deviation of the "Expression" field. Returned value are considered as a sample of the universe
DVar	Variance of the "Expression" field. Returned value are considered as the whole universe
DVarP	Variance of the "Expression" field. Returned value are considered as a sample of the universe

For Example the following code:

```
X = DMin("Freight_Cost", "ORDERS", "ShipCountryRegion = 'UK'")
Y = DVar("Freight_Cost", "ORDERS", "ShipCountryRegion = 'UK'")
```

Assigns to X and to Y, respectively, the less costly freight and the variance of the cost of all the shipments made in UK.

Now, if a DFunction corresponds to a standard select query, why should we use a DLookup, if the same result can be obtained, even quicker, with a simpler query? The answer is: <u>"To ensure greater flexibility"</u>.

For instance, what if we wanted to search the employee with ID = 2 instead of the one with ID = 1? Should we write a different query? Of course not!

A first possibility could be that to substitute a static query, with a parametric one, such as:

```
SELECT Surname
FROM EMPLOYEES
WHERE EmployeeID = ["Please insert ID"]
```

In this way, anytime the query is executed, the system shows (to the user) a data input form, asking for the required ID.

This is not bad, but using a DLookup allows one to do something much better than this. Indeed, a very smart solution could be that to create an input form with:

- A TextBox (called TxtInput) where, at run time, the user can insert the required ID;
- A second TextBox (called TxtOupt) that executes the following DlookUp:

```
= DLookup("Surname", "EMPLOYEES", "ID = " & TxtInput)
```

The only "odd" part of this DLookup function is the <u>filtering condition</u> written as:

```
" ID =  "  & TxtInput
```

The meaning is simple:

- "ID = " is a fixed string;
- TxtInput is the name of the text box where the input ID has been placed and, by default, it also indicates the content of the same text box;
- & is a basic operator used to concatenate two or more strings;

So, if the user inserts a value equal to 3 in the text box, the DLookup becomes:

```
= DLookup("Surname", "EPLOYEES", "ID = 3"),
```

which corresponds to the following SQL code:

```
SELECT Surname
FROM EMPLOYEES
WHERE EmployeeID = 3
```

To create the form on Access the following steps must be followed:

- From the Create toolbar, select *Blank Form;*
- Then open the form in Design View and open the properties list;
- In the properties list set the following Format Properties:

- Label (etichetta) = "DFunction Example" - This is the title of the form
- Width (larghezza) = 10 [cm]
- Records Selector (Selettori record) = "No"
- Navigation Buttons (Pulsanti di navigazione) = "No"
- SLiding Bars (Barre di Scorrimento) = "No"

- From the “Other Properties” windows, set Pop Up = "Yes"
- Select the Body part of the mask and, from the Format property menu set High = 5 [cm]

Now we need to add some controls on the form. Specifically let us insert:

- A Text Box – Using the property menu Rename it as TxtInput and set its default value to 1; assign the value "ID" to its associated label (operating directly on the form in design view);
- Another text Box – Rename it as TxtOutput and set its default value to " "; assign the value "Last Name " to its associated label;
- A third Text Box - Rename it as TxtOutput2 and set its default value to " "; assign the value "Last Name " to its associated label;
- Two command buttons - Operating directly on the form opened in design view, label them as "Run Query" and "Clear Name", respectively; also, if you want to, using the property meu, rename the buttons, as BtnQuery and BtnClear.

At this point the form should look like this one:

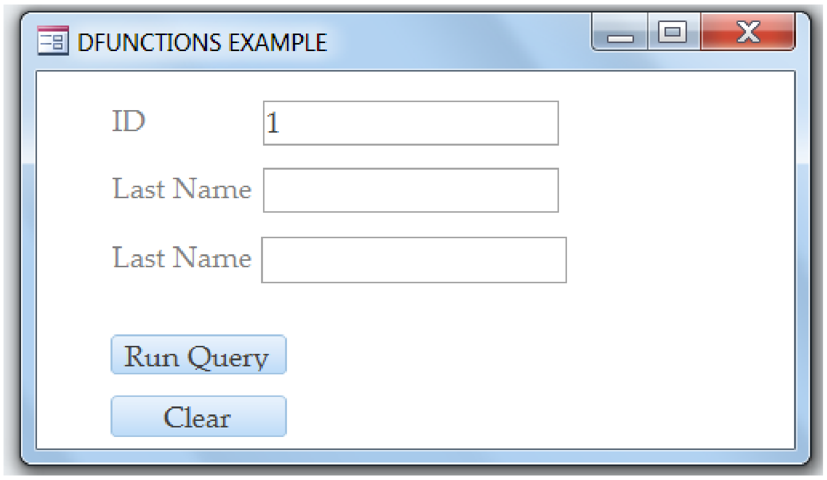
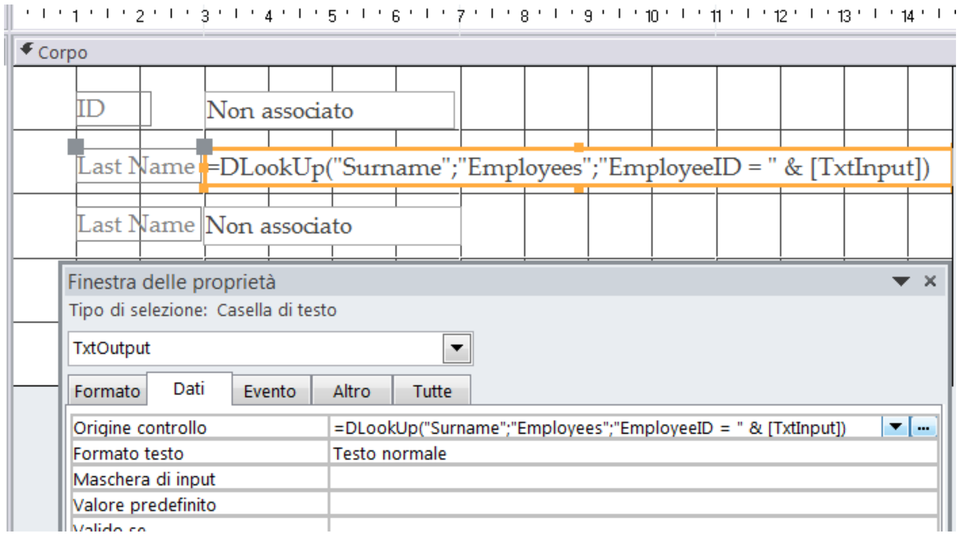


Fig. 6.1. *The Blank Form*

In order to associate a DFunction to a Text Box, in design view, it is sufficient to write the DFunction inside the Text Box or, alternatively, using the property menu, to write the DFunction in the Data Source window, as shown in Figure 6.2.

Now if you try using the form, everything seems fine; however, if the user type in an ID that does not exist (in the EMPLOYEES table) the TxtOutput text box becomes blank. This is still acceptable, yet if the user type in a string instead of a number, the system issues an Error and the whole form begins to flicker. To solve this problem, it is better to call the DlookUp inside a VBA script, so that the validity of the input can be correctly checked.



Origine controllo	=DLookUp("Surname";"Employees";"EmployeeID = " & [TxtInput])
Formato testo	Testo normale
Maschera di input	
Valore predefinito	

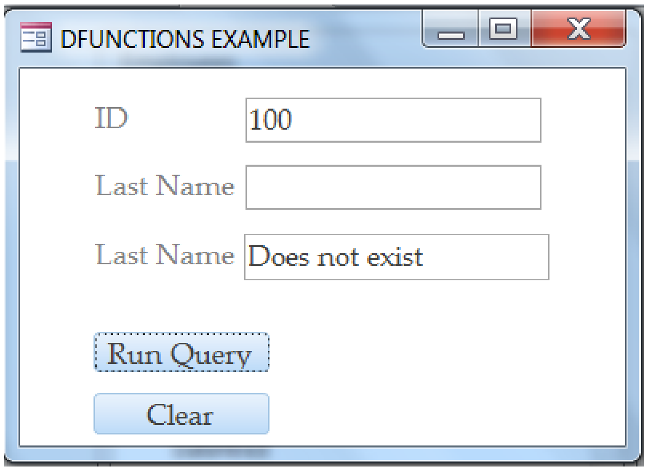
Fig. 6.2. *The DlookUp Function*

To this aim we have to associate to the on-click event of the two command buttons the following VBA code:

```
Private Sub BtnClear_Click()
'This subroutine takes back the form to the original condition (corresponding to ID = 1)
   TxtInput = 1
   Call BtnQuery_Click
End Sub

Private Sub BtnQuery_Click()
'This subroutine checks the input and, next, it executes the DLookUp
  Dim ID As Integer
  Dim Txt As Variant 'We need a variant because the input may be everything
  Dim Out As String
    Txt = TxtInput
    If IsNumeric(Txt) Then
       ID = Txt
       Out = Nz(DLookup("Surname", "Employees", "EmployeeID = " & ID), "Does not exist")
    Else
       Out = "Non valid ID"
    End If
   TxtOutput2 = Out
End Sub
```

Now, if you try to use the form you should get something similar to Figure 6.3 below:


DFUNCTIONS EXAMPLE

ID: 100

Last Name:

Last Name: Does not exist

Fig. 6.3. *The complete Form*

7. Crating dynamic query using domain functions

It is worth remembering that, the execution order of a standard selection query, based on operators working on groups, is the following one:

- SELECT … FROM are executed at first and, in this way, a first table, let say T_1, is returned;
- Next, records of T_1 are filtered using the logical condition specified by the WHERE clause and a second table T_2 is returned;
- Only at this point the function operating on groups (SUM, AVG, COUNT, etc.) and the HAVING and GROUP BY operators are used to make the final computation, on the records of T_2.

It is worth noting that this is exactly the reason why it is not possible to include a function operating on group in the WHERE condition; indeed, due the logical sequence with which a selection query is executed, this would not permit to find a closing condition.

Conversely, the execution sequence of a Domain Function is totally different. Specifically, **records are analyzed one by one** (and not altogether as in a standard query) and both the filtering and the aggregation conditions are evaluated on each single record. In other words, if a table has *n* records both the filtering and the aggregation conditions are executed *n* times. This fact explain the reason why a Domain Function is slower than its homologous selection query; indeed, in the latter one, the filtering and the aggregation conditions are executed only once, as they both operate simultaneously on all the records of the table.

For example, both the following two queries based on Dfunctions are very inefficient in terms of execution time:

```
SELECT DISTINCT DCount("*","CUSTOMERS") AS Total
FROM CUSTOMERS

SELECT ProductName, UnitPrice
FROM PRODUCTS
WHERE UnitPrice > (DAvg("UnitPrice","PROUCTS") + DStDev("UnitPrice","PROUCTS"))
```

Thus these queries should be rewritten as:

```
SELECT COUNT(ID) AS Total
FROM CUSTOMERS

SELECT ProductName, UnitPrice
FROM PRODUCTS
WHERE UnitPrice > (SELECT AVG(UnitPrice)+ STDEV(UnitPrice) FROM PROUCTS)
```

This fact could be seen as a drawback, but conversely it is exactly the biggest advantage offered by Domain Functions. Indeed, since records are analyzed one by one, using Domain Function makes it possible to **create complex queries** based on the relative position of the records of the reference table. In the following Sub-Section we will see how to do so.

How to properly write a criterion

In order to create a dynamic query, all we have to do is to properly define the criterion input. As we have seen, the criterion must be passed as a string but, since Dfunction are VBA functions, this string may be fixed, but it can be made by a variable part, as well.

For instance, both the following instructions are correct:

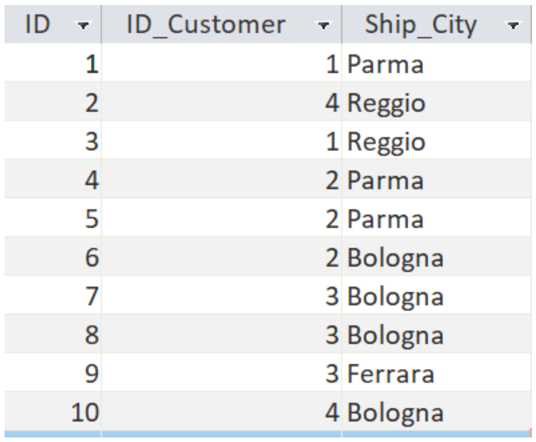
```
DLookup("CustomerID", "Orders", "OrderID = 1")
DLookup("CustomerID", "Orders", "OrderID = " & Forms![Orders]!OrderID)
```

In both case a single CustomerID will be retrieved from the Orders Table, depending on the value of the OrderID. Yet, whereas in the first case the value of the OrderID is fixed, in the latter one it is left free to vary dynamically, depending on the current value inserted (by the user) in the Orders Text Box placed in the Orders Form.

Also, and perhaps more important, to create a variable criterion, we can also make reference, directly, to a specific field of the table. This is a very important feature, but it may sound a little bit confusing; so we will explain it through an example.

Let us consider the following (simplified) ORDERS table

Tab. 6.5. *The ORDERS Table*



ID	ID_Customer	Ship_City
1	1	Parma
2	4	Reggio
3	1	Reggio
4	2	Parma
5	2	Parma
6	2	Bologna
7	3	Bologna
8	3	Bologna
9	3	Ferrara
10	4	Bologna

We want to write a query that returns the number of order issued by each customer. Using standard SQL we could easily obtain this result in the following way:

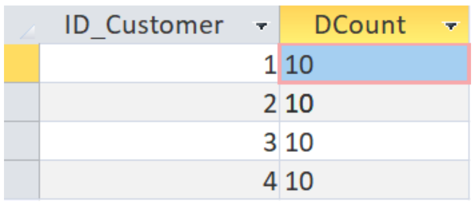
```
SELECT ID_Customer, COUNT(ID) AS [Number of Order Issued]
FROM ORDERS
GROUP BY ID_Customer
```

Using a DFunction embedded in a SQL query, we could try the following (wrong) solution:

```
SELECT ID_Customer, DCOUNT("iD", "ORDERS")
FROM ORDERS
GROUP BY ID_Customer
```

Unfortunately, the result is the following one:

Tab. 6.6. *Unexpected and wrong result*



ID_Customer	DCount
1	10
2	10
3	10
4	10

This is definitely not what we expected! Each customer received the overall number of orders (i.e., 10), and not the total number of orders made by himself. Why so? Mainly, this is due to the following issues:

- We did not specified any filtering criteria;
- DFunctions operate in a positional way, by evaluating one record at a time (of the Table defined in the "Expression" input parameter);

- As for any calculated field, also the result returned by a DFunction is attached (as an additional field) to each record of the standard query, in which the DFunction has been embedded.

For these reasons, although records have been grouped by customer id, the Dsum is repeated 10 times, one for each record of the original ORDERS table and, each time it returns a value of 10 (as the number of records of the table). Also, since we did not defined any criterion, this value is attached to all the 10 records returned by the original Select query.

In a certain way, you could imagine that, at first, the following column query is run:

```
 SELECT ID_Customer
FROM ORDERS
GROUP BY ID_Customer
```

And that next, the following scalar query is run, and the returned value (i.e., 10) is attached to each one of the four records returned by the previous query.

```
 SELECT COUNT(ID)
FROM ORDERS
```

At this point it should be evident that, even a static criterion would not solve the problem. For instance:

```
SELECT ID_Customer, DCOUNT("ID", "ORDERS", "ID_Customer = 1")
FROM ORDERS
GROUP BY ID_Customer
```

Would assign a value equal to 2 (i.e., the number of orders issues by Customer #1) to all the customers.

What we need to do is to <u>write a variable criterion, by referencing directly to the Customer _ID field</u>. This is shown below:

```
SELECT ID_Customer, DCOUNT("ID", "ORDERS", "ID_Customer = " & [ID_Customer])
FROM ORDERS
GROUP BY ID_Customer
```

Specifically,

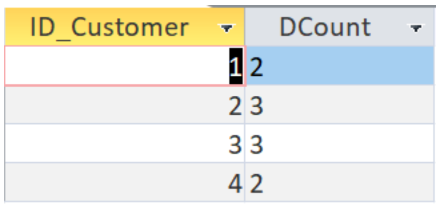
"ID_Customer = " & [ID_Customer]

<u>is a string made by the fixed part "ID_Customer = " concatenated with the value taken (at run time) by the variable [ID_Customer]. Since this variable has the same name of a field of the Table on which the Dfuntion operates, it takes the value of that field, relatively to the record that is currently been evaluated</u> (remember that DFunctions evaluate a record at a time). For instance, when DCount evaluates the third record {ID = 1, ID_Customer = 3, Ship_City = Reggio} of the ORDERS Table, it returns a value equal to 2. Indeed, relatively to this record [ID_Customer] = 3 and the DCount becomes the following one:

```
DCount("ID", "ORDERS", "ID_Customer = 3")
```

The same process is repeated for each record of the ORDERS table, and so we finally get the desired result:

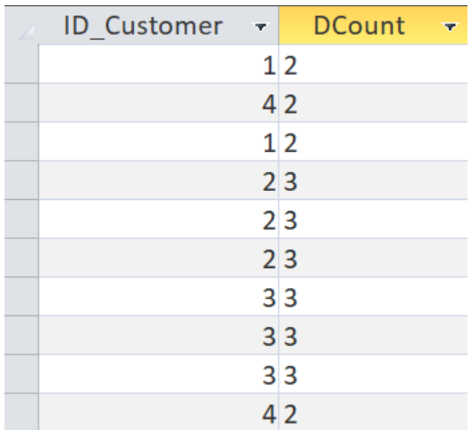
Tab. 6.7. *The correct result*



ID_Customer	DCount
1	2
2	3
3	3
4	2

Please note that, without using the GROUP BY condition, the result would be the following one (can you explain why?):

Tab. 6.8. *The correct result without the GROUP BY Condition*



ID_Customer	DCount
1	2
4	2
1	2
2	3
2	3
2	3
3	3
3	3
3	3
4	2

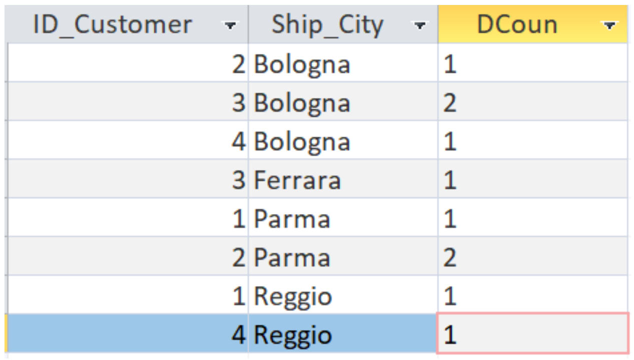
We conclude this sub section noting that the criterion can be extended to include multiple (variable) conditions. For instance, we could add to the previous query an additional filtering criterion, to group orders by customer and by shipping city, at the same time (i.e., we want to see how many orders of a customer have been shipped to the same destination). To this aim it is sufficient to modify the query as it follows (note that the new parts have been highlighted in bold):

```
SELECT ID_Customer, DCOUNT("ID", "ORDERS", "ID_Customer = " & [ID_Customer] & _
                                          _ " AND Ship_City = '"  & _ [Ship_City] & "'")
FROM ORDERS
GROUP BY ID_Customer, Ship_city
```

Please note that, Ship_City is a text field, and so in SQL it must be enclosed between single quotations marks; furthermore, since quotations marks are themselves strings, they must be enclosed in double quotation marks, too. This explains the following notation " ' " used after strings- concatenation symol &.

The final results is shown below:

Tab.6.9. *The result with a double filtering condition*



ID_Customer	Ship_City	DCoun
2	Bologna	1
3	Bologna	2
4	Bologna	1
3	Ferrara	1
1	Parma	1
2	Parma	2
1	Reggio	1
4	Reggio	1

Running Sum and Data Ranking Examples

We are now ready to introduce some more demanding examples, that make it possible to fully exploit DFunction capabilities.

As a first example, let us considered a table T with a "Price" field of currency type. We want to calculate a running sum of the Price field. In other words, we want to add a calculate field that, at each generic position *i* contains the sum of the prices of all records that comes before position *i* i.e., $Running\ Sum\ (i) = \sum_{j=1}^{i} Price_j$.

We can easily obtain the desired result using a Dsum, as shown below:

```
SELECT *, Format(DSum("Price","T","ID <=" & [ID]),"Currency") AS [Running Sum]
FROM [T]
ORDER BY ID
```

Specifically:

- Format is a function that allows one to properly formatting the value of a certain field; its syntax is **Format(< Name of the string to be formatted>,[<format type>])**. In the present case we use "currency" as format type, since we wanted to format the value returned by DSum as a standard currency value;
- "ID <=" & [ID] is the a dynamic condition, obtained by referencing, directly, the ID field of the original Table T;
- We used the ORDER BY ID condition, to sort records in ascending order depending on their ID;

- We executed the filter on the ID field because, being the PK, it is the only field that cannot be null and because it contains natural numbers in ascending order.

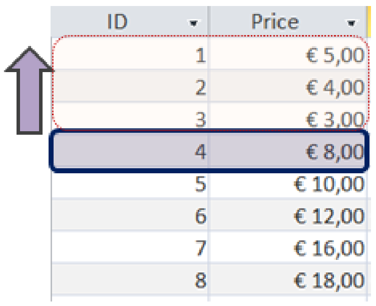
Let us now try to understand how the DSum actually operates. Specifically, when applied to a generic record in position '*r*' the DSum turns into:

```
DSum("Price","T","ID <= r")
```

Which corresponds to:

```
SELECT SUM(Price) AS [Running Sum]
FROM T
HAVING ID <= r
```

Note that the fact that ID = *r*, is due to the fact that records were sorted in ascending order depending on their ID[4]. Also, due records' sorting, the ID of the record in position *r* will be higher than that of all the previous ones: {*(r-1), (r-2)*, …, 1}. This assures that the (running) sum is executed on all the records that precede the current one, as graphically shown by Figure 6.4 (where *r* = 4).

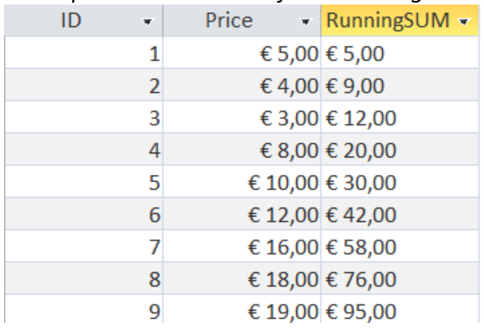


ID	Price
1	€ 5,00
2	€ 4,00
3	€ 3,00
4	€ 8,00
5	€ 10,00
6	€ 12,00
7	€ 16,00
8	€ 18,00

Fig. 6.4. *DSum applied to the forth record*

A possible outcome is shown in Table 6.10:

Tab. 6.10. *A possible outcome of the Running Sum Query*



ID	Price	RunningSUM
1	€ 5,00	€ 5,00
2	€ 4,00	€ 9,00
3	€ 3,00	€ 12,00
4	€ 8,00	€ 20,00
5	€ 10,00	€ 30,00
6	€ 12,00	€ 42,00
7	€ 16,00	€ 58,00
8	€ 18,00	€ 76,00
9	€ 19,00	€ 95,00

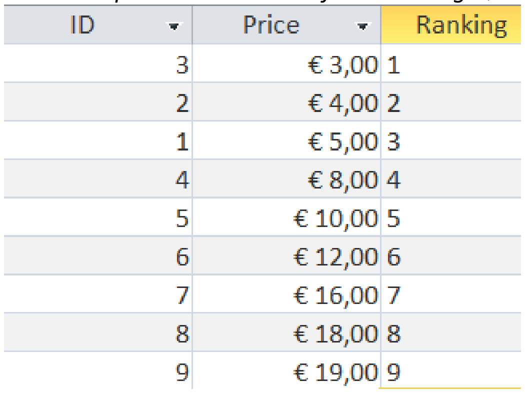
[4] *Although this is not always true, as in the case of missing values of the ID (due to the removal of some records), the previous reasoning can be generalized with no difficulties.*

It is worth noting that, with a simple change of the previous query, we could obtain a sort of ranking of the data, i.e., we could add a calculated field showing how many prices are lower than the current one. The query is the following one:

```
SELECT T.ID, T.Price, DCount("Price", "T", "Price <= " & [Price]) AS [Ranking]
FROM T
ORDER BY T.Price
```

The result is shown in Table 6.11:

Tab. 6.11 *A possible outcome of the Ranking Query*



ID	Price	Ranking
3	€ 3,00	1
2	€ 4,00	2
1	€ 5,00	3
4	€ 8,00	4
5	€ 10,00	5
6	€ 12,00	6
7	€ 16,00	7
8	€ 18,00	8
9	€ 19,00	9

For the sake of completeness we can show how the same result could be obtained using standard SQL. To this aim let us consider a very simple table, called RST (i.e., Running Sum Table), with records made of two fields, Year and Demand, respectively, as shown in Table 6.12 (a) below.

Tab. 6.12 (a) *The RST Table*

Year (PK)	Demand
1	100
2	80
3	70
4	80

To obtain the running sum we will:

1. Perform a Cartesian product between RST and itself (renamed, with the aliasing technique as RST_1),
2. Filter and order data in a “smart” way,
3. Sum and group data so as to generate the running sum column.

Let us start making the Cartesian product.

To do so we have to write the following query:

```
SELECT RST.Year, RST.Demand, RST_1.Year, RST_1.Demand
FROM RST, RST AS RST_1
ORDER BY RST.Year
```

The ORDER BY condition is used to be sure that, after the Cartesian product, records are ordered in terms of the year of the original table. In this way we get a table with 16 records of 4 fields each, as shown in Table 6.12 (b), where the records in grey are those ones that will be removed next.

Tab. 6.4 (b). *The Cartesian Product Table*

RST.Year	RST.Demand	RST_1.Year	RST_1.Demand
1	100	1	100
1	100	2	80
1	100	3	70
1	100	4	80
2	80	1	100
2	80	2	80
2	80	3	70
2	80	4	80
3	70	1	100
3	70	2	80
3	70	3	70
3	70	4	80
4	80	1	100
4	80	2	80
4	80	3	70
4	80	4	80

Now we need to filter these data. What we want is to keep only those records where RST.Year >= RST_1.Year, in order to match to the generic record corresponding to year [i] all the records corresponding to year [1] to year [i]. In this way it will be easy to generate a running sum by grouping and ordering the records, as clearly shown by Table 6.12 (c).

To this aim we just need to add the following filtering condition to the previous query:

```
WHERE RST.YEAR >= RST_1.Year
```

By operating in this way we obtain something like Table 6.4 (c):

Tab. 6.12 (c). *The Filtered and Ordered Cartesian Product Table*

RST.Year	RST.Demand	RST_1.Year	RST_1.Demand
1	100	1	100
2	80	1	100
2	80	2	80
3	70	1	100
3	70	2	80
3	70	3	70
4	80	1	100
4	80	2	80
4	80	3	70
4	80	4	80

As it can be see, at this point, getting a running sum column is straightforward, as it is sufficient to sum RST_1.Demand grouping by RST.Year. For instance, if we call Table 6.12(c) as OCPT, then the desired outcome can be obtained as follows:

```
SELECT RST.Year, RST.Demand, SUM(RST_1.Demand) AS [Running Sum]
FROM OCPT
GROUP BY  RST.Year, RST.Demand
```

Tab. 6.12 (d). *The Running Sum Table*

Year	Demand	Running Sum
1	100	100
2	80	180
3	70	250
4	80	330

Clearly, it is also possible to write it all in a single query as shown below:

```
SELECT RST.Year, RST.Demand, SUM(RST_1.Demand) AS [Running Sum]
FROM RST, RST AS RST_1
GROUP BY RST.Year, RST.Demand
WHERE RST.YEAR >= RST_1.Year
ORDER BY RST.Year
```

Moving Average Example

Let us consider a table SS with records of two fields, namely Year and Demand, respectively. Specifically, the demand field contains historical data concerning the (past) yearly demand of a certain product. We want to add a calculated field containing the **moving average** of order three of the past demand, that is:

$$MAvg_3(r) = \frac{\sum_{j=(r-2)}^{r} Demand(j)}{3}$$

To this aim we could use a *DSum* function as follows:

```
SELECT Year, Demand, (Dsum("Demand","SS", "Year <=" & [Year]) - _
                                             _ Nz(Dsum ("Demand","SS", "Year <=" & [Year] - 3),0))/3 AS MAV3
FROM SS
ORDER BY Year
```

As it can be seen, records are ordered using their Year and next, for the generic record *r* the yearly demand from record *1* to record *r* is summed up by means of Dsum("Demand","SS", "Yera <=" & [Year]). However we need to sum demand only from year *(r-2)* to year *r*, thus, we need to rectify this value by subtracting to the overall sum, the sum made on year 1 to year *(r-3)*. This is made using Dsum ("Demand","SS", "Year <=" & [Year] - 3). Lastly the Moving average is obtained dividing by three the precedent sum:

```
(Dsum("Demand","SS", "Year <=" & [Year]) - Dsum ("Demand","SS", "Year <=" & [Year] - 3))/3
```

For example if *r* = 4 we have:

```
(Dsum("Demand","SS", "Year <= 4") - Dsum ("Demand","SS", "Year <= 1"))/3
```

This corresponds to the following and correct calculation: $[(D_1 + D_2 + D_3 + D_4) - (D_1)]/ 3 = [(D_2 + D_3 + D_4) - (D_1)]/ 3$

Obviously, this operation works only starting from the third record. For instance for the first record [ID] - 3 would be (1 - 3) = -2 and this would generate an error. This is the reason why in the query we have written:

```
Nz(Dsum ("Demand","SS", "ID <=" & [ID] - 3),0)
```

Where **Nz is the Null Zero function** and has the following syntax:

```
Nz(<field to be evaluated>, [<value if null>]).
```

Specifically this function receive a field, if the field is not null it returns the value contained in the field; conversely, if the field is null Nz returns a value equal to zero or the value passed as the optional input parameter [<value if null>].

We also note that the same result can be obtained using the DAvg domain function:

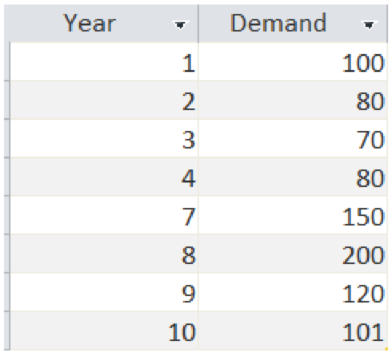
```
SELECT Year, Demand, (DAVg("Demand","SS", "Year <=" & [Year] & "Year > " & [Year] – 3]) AS MAV3
FROM SS
ORDER BY Year
```

Note that, in this case we use a double condition in order to correctly define the domain on which the average has to be made.

Lastly we note that, to avoid missing values (i.e., some years that are missing as in Table 6.13 (a)), it is convenient to create, at first, a second table with continuous values (for the years), using the following query, similar to the one, that we previously made to generate a ranking of the data:

```
SELECT Year, Demand, DCount("Year", "SS", "Year <=" & [Year]) AS Year2
FROM SS
ORDER BY Year
```

Tab. 6.13 (a). *The SS table with missing years*



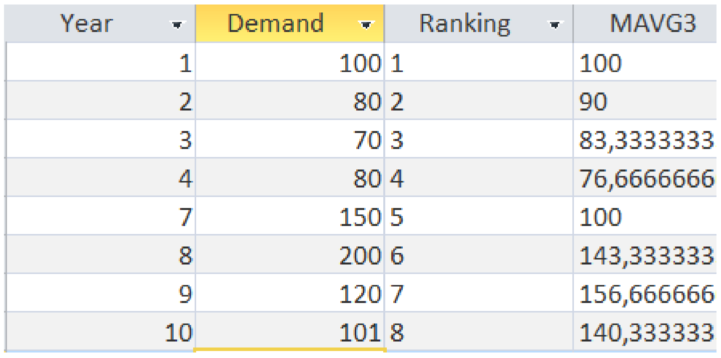
Year	Demand
1	100
2	80
3	70
4	80
7	150
8	200
9	120
10	101

Lastly if we call the generated table as SSRK, we can obtain the moving average with the following query:

```
 SELECT Year, Demand, Ranking, DAvg("Demand","Year2","Year2g <=" & [Year2] & _
                                                     _ "AND Year2 > " & [Year2] - 3) AS [MAVG3]
FROM SSRK
ORDER BY Ranking
```

The final result is shown in Table 6.13 (c):

Tab. 6.13 (b). *The Moving Average of Order Three*



Year	Demand	Ranking	MAVG3
1	100	1	100
2	80	2	90
3	70	3	83,3333333
4	80	4	76,6666666
7	150	5	100
8	200	6	143,333333
9	120	7	156,666666
10	101	8	140,333333