The SQL Language Part II - How to create queries operating on several tables

1. Introduction

Up to know we have generated queries that operate on a single table. More specifically we have seen that:

- The <*Selection List*> makes it possible to realize a Projection;
- The DISTINCT and the WHERE condition make it possible to realize a Selection;
- The AS operator makes it possible to realize a renaming;
- The GROUP BY and HAVING operators make it possible to realize what, in relational algebra, is called an aggregation;

We still need to see a couple of basic operator (i.e., the UNION and the SUBTRACTION), but we will do it later. Now we want to focus on the way in which the JOIN operator is implemented by the SQL language. In this way we will be able to create complex queries operating on more than one table. Clearly, the hypothesis is made that all the tables we will be operating on are linked by a generic or, most of the times, by an OTM relation with integrity constraint.

2. Inner Join

Let us consider two Table T_1 and T_2 in OTM relation, with T_1 being the father table. Let PK_1 be the PK of T_1 and FK_2 be the forward key of T_2.

If we want to create a (non-normalized) table showing all the data contained in T_1 and all the data contained in T_2, relatively to the records of T_2 that have one or more correspondence in T_1 we need to:

- Indicate all the fields of both tables in the Selection list
- Indicate the name of both Tables in the FROM clause
- Explicit, as if it were a logic condition, the OTM relation between T_1 and T_2

This is shown in the following query:

```
SELECT T1.*,T2.*
FROM T1, T2
WHERE T1.PK1 = T2.FK2
ORDER BY T1.Field1
```

Since we are operating on two different tables, in order to indicate a specific field it is advisable to make reference, explicitly, to the table to which the record belongs to. For instance, the field F_1 (used to sort in ascending order the records of the output table) belongs to table T_1 and so it is referred as $T_1.Field_1$; the **dot operator (.)** is used to separate the name of the table from the name of the field and indicates "ownership", i.e.,

the field belongs to the table. This is not mandatory but it is certainly advisable. Clearly, if a field with the same record exists in both tables, then the use of the dot operator becomes mandatory.

Also note that the same also apply to the ALL (*) operator, as for T_1.* and T_2.*

Lastly we observe that the OTM relation is logically described in the following way:

WHERE $T_1.PK_1 = T_2.FK_2$

This statement assures that only the records of the "son" table that have a correspondence in the "father" one will be returned by the query.

In a more formal way we can say that the OTM relation is expressed by the following logical condition:

$Name\_of\_Tabel_1.Name\_Common\_Field_{1,2} = Name\_of\_Table_2.Name\_Common\_Field_{1,2}$

As an alternative it is possible to use a specific SQL syntax to operate a JOIN on two tables in OTM relation, as shown below:

<*Command* INNER|NATURAL JOIN> : : =

< *Reference to the father table*> [NATUAL] INNER JOIN <*reference to the son table*>

[ON <*join condition*> | USING <join fields list>]

It is important to note that the **INNER JOIN** command must be placed in the **FROM** statement and not, as before in the WHERE condition.

Also note that, as it can be seen, SQL differentiates among the INNER and the NATURAL JOIN. In the first case all the fields of both tables are returned exactly as in the previous query where we wrote SELECT T_1.*,T_2.*; conversely using the keyword NATURAL JOIN common fields (i.e., the forward key of the son table) will not be returned.

Concerning the definition of the join, two possibilities are allowed:

- When the JOIN is based on the use of the ON operator, one must specify the logical condition defying the join between the tables;
- When the JOIN is based on the USING operator, one must list the name of the fields (i.e., the PK of the father table and the FK of the son table) on which the join is made.

Owing to ehat we've just said, the join-query that we made above, can also be implemented in the following ways:

SELECT T_1.*,T_2.*

FROM T_1 **INNER JOIN** T_2 **ON** $T_1.PK_1 = T_2.FK_2$

ORDER BY $T_1.Field_1$

```
SELECT T1.*,T2.*
FROM T1 INNER JOIN T2 USING T1.PK1, T2.FK2
ORDER BY T1.Field1
```

When the Join is made between two tables in OTM relation, the choice between ON and USING is just a matter of preference. Conversely, if the Join is made between tables that have a "generic relation", it is advisable to make use of the ON operator; indeed, this makes it possible to define the join condition using additional logical operators, rather than the standard equality one (=).

Anyhow, it is always possible to include additional logical condition in the WHERE clause. For instance, considering once again the AUTHORS and the BOOKS tables introduced in the previous chapter, we could be interested to see all the books written by Joyce. To this aim we could use, at our choice, any one of the following queries (that return a vector containing all the books written by Joyce):

```
SELECT BOOKS.Title
FROM AUTHORS, BOOKS
WHERE (AUTHORS.ID_Authors = BOOKS.ID_Autors) AND (AUTHORS.Surname = 'Joyce')
ORDER BY BOOKS.Title
```

```
SELECT BOOKS.Title
FROM AUTHORS INNER JOIN BOOKS ON AUTHORS.ID_Authors = BOOKS.ID_Autors
WHERE AUTHORS.Surname = 'Joyce'
ORDER BY BOOKS.Title
```

```
SELECT BOOKS.Title
FROM AUTHORS INNER JOIN BOOKS USING AUTHORS.ID_Authors, BOOKS.ID_Autors
WHERE AUTHORS.Surname = 'Joyce'
ORDER BY BOOKS.Title
```

3. Inner Join among more than two tables

An inner join can be defined even if there are several tables linked by OTM relations. The logic does not change. Indeed, it is sufficient to include all the Tables in the FROM clause and, next, to explicit all the OTM relations (i.e., the couples of primary and forward keys) in the FROM or in the WHERE clause.

Let us make an example to clarify this concept. In Figure 5.1 there are two tables, ORDERS and PRODUCTS, which are linked through a MTM relations. Since the relation is of the MTM type (i.e., an order may contain several products and the same product could have been ordered more than once) a bridge table, namely ORDER DETAILS, is used to split the relation into two OTM relations. A last table, namely CUSTOMERS has an OTM relation with the ORDERS table because, evidently, each customer could have made more than one order.

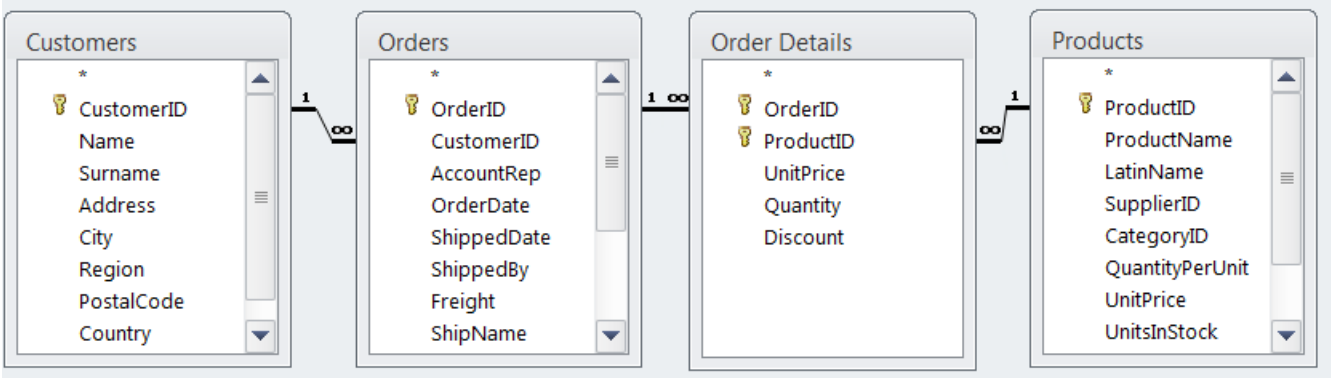

Fig. 5.1. An example of four tables linked by OTM relations

We want to see the composition of all the orders that have been issued in a certain period of time and their total value. We also want to know who made the order.

A possible solution is as follows:

```
SELECT CUSTOMERS.Surname, PRODUCTS.ProductName, ORDERS.OrderDate, _
       [ORDER DETAILS].Quantity*[ORDER DETAILS].UnitPrice*(1-[ORDER DETAILS].Discount) AS Price
FROM PRODUCTS INNER JOIN ((CUSTOMERS INNER JOIN ORDERS ON CUSTOMERS.CustomerID = _
    ORDERS.CustomerID) INNER JOIN [ORDER DETAILS] ON ORDERS.OrderID = [ORDER DETAILS].OrderID) _
    ON PRODUCTS.ProductID = [ORDER DETAILS].ProductID
WHERE (Orders.OrderDate>=[Date In]) AND (Orders.OrderDate<=[Date Out])
ORDER BY Orders.OrderDate
```

In this case:

- Price is a calculated field
- To filter orders belonging to a certain period the condition Orders.OrderDate>=[Date In] AND Orders.OrderDate <=[Date Out] is used. Please note that, in this case, the brackets have a different meaning, as they define an input provided by the user, i.e., when running this query, the software will ask the user for an initial and for and ending date. Specifically, anytime something written inside brackets is not recognized by the software (i.e., it is neither a keyword nor a field or a table), it is interpreted as a parametric input;

- The set of the relations (or the relational path), is defined in the FROM clause using the INNER JOIN ON syntax. More specifically, as indicated by the parentheses:
  - A first Join, let us call it J_1, is made between CUSTOMERS and ORDERS;
  - A second Join, let us call it J_2, is made between J_1 and [ORDER DETAILS];
  - Finally, a third and last join is made between PRODUCTS and J_2.

Conversely, including the join condition in the WHERE clause, the query can be written in the following way:

```
SELECT CUSTOMERS.Surname, PRODUCTS.ProductName, ORDERS.OrderDate, _
       [ORDER DETAILS].Quantity*[ORDER DETAILS].UnitPrice*(1-[ORDER DETAILS].Discount) AS Price
FROM CUSTOMERS, ORDERS, [ORDERS DETAILS], PRODUCTS
WHERE (CUSTOMERS.CustomerID = ORDERS.CustomerID) AND (ORDERS.OrderID = _
      [ORDER DETAILS].OrderID) AND (= [ORDER DETAILS].ProductID = PRODUCTS.ProductID) _
       AND (Orders.OrderDate>=[Date In]) AND (Orders.OrderDate<=[Date Out])
ORDER BY Orders.OrderDate
```

Note that in this case the join condition is obtained by concatenating three distinct AND condition, that is one AND condition for each one of the three OTM relations among the four original tables.

A possible outcome (inserting dates form 01/01/2010 t0 31/12/2010) is shown in Table 5.1

Tab. 5.1

A possible outcome of the multiple join

Surname	Product Name	OrderDate	Price
Lang	Crushed rock	05/01/2010	62,5
Ackerman	Douglas Fir	05/01/2010	18,75
Ackerman	Fortune Rhododendron	05/01/2010	43,199999928
Lang	Compost bin	05/01/2010	58
Browne	Golden Larch	06/01/2010	27
Browne	Lawn cart	06/01/2010	76,499999873
Khanna	Bat box	06/01/2010	44,25
Koch	Compost bin	08/01/2010	58
Koch	GrowGood potting soil	08/01/2010	6,35
Koch	QwikRoot	08/01/2010	18
Koch	Grass rake	08/01/2010	11,95
Ramos	Cactus sand potting mix	12/01/2010	9
Oveson	Bat box	12/01/2010	29,5
Cox	Blackberries	12/01/2010	27
Cox	Gooseberries	12/01/2010	22,5
Thirunavukkara	Pea gravel	12/01/2010	72
Cox	Ambrosia	12/01/2010	6,25
Miller	Grandiflora Hydrangeas	13/01/2010	40

Note that, sometimes, some records have the same data and the same customer. These are not distinct orders, but lines of the same order issued in a certain date by a certain customer. Thus, it could be useful to group lines in a single order.

A possibility could be that to create a single query performing at the same times the joins and the grouping operator. However, the query would become extremely long and difficult to be understood. Instead, it is easier to make a query that operates on the previous query (the one corresponding to Fig. 5.2). More precisely, if we

save the query as JOIN_QUERY we can use it as a new table and we can call it directly in the FROM statement of a new query. An example follows:

```
SELECT Surname, OrderDate, SUM(Price)
FROM JOIN_QUERY
GROUP BY Surname, OrderDate
```

Please note that the use of the SUM function, that made it possible to sum up all the lines belonging to the same order. Also note the use of both Surname and OrderDate in the GROUP BY statement. This is needed to avoid that orders made by the same customer in different date could be aggregated in a single record.

4. Outer Join

As we have said talking about relational algebra, when executing an Inner/Natural join, all the records of table T_1 that do not have any correspondence with records of table T_2 will get lost. For instance if we add a new category (let us call it "new") in the CATEGORIES table, if we execute an INNER JOIN on CUSTOMERS and CATEGORIES, the "new" category will not be displayed.

So, if we wanted to see all the records of both tables, we would execute an OUTER JOIN instead of an INNER one. Clearly, as in relational algebra, also SQL implements three kinds of OUTER JOIN:

- LEFT OUTER JOIN
- RIGHT OUTER JOIN
- FULL OUTER JOIN

The full SQL syntax does not change very much, as shown below:

```
<OUTER JOIN Command> : : =
< Name of first Table> [NATURAL] {LEFT | RIGHT | FULL} [OUTER] JOIN <Name of second table>
[ON <join condition> | USING <list of join fields>]
```

A simple example is shown below (see Table 5.2):

```
SELECT PRODUCTS.ProductName, CATEGORIES.CategoryName
FROM CATEGORIES LEFT OUTER JOIN PRODUCTS _
        _ ON CATEGORIES.CategoryID = PRODUCTS.CategoryID
```

Tab. 5.2.
LEFT OUTER JOIN

Category Name	Product Name
Tools	Revolving sprinkler
Tools	Shade fencing 6'
Berry bushes	Currant
Berry bushes	Red Raspberries
Berry bushes	Blackberries
Berry bushes	Gooseberries
Berry bushes	Strawberries
Shrubs/hedges	Weeping Forsythia
Shrubs/hedges	Winterberry
Shrubs/hedges	Morrow Honeysuckle
Shrubs/hedges	Beautybush
Shrubs/hedges	Hedge trimmer 18"
Shrubs/hedges	Hedge trimmer 16"
Shrubs/hedges	Hedge shears 10"
New	

It is interesting to note that, in Access, a LEFT OUTER JOIN is graphically displayed as shown in Figure 5.2:

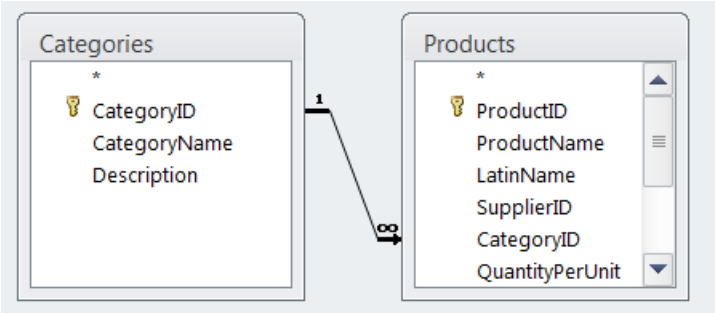

Fig. 5.2. *LEFT OUTER JOIN*

We also note that the FULL OUTER JOIN is not implemented by Access. However, it can be easily reproduced by making an UNION of a LEFT and of a RIGHT OUTE JOIN executed on the same input tables.

We conclude this section mentioning the way in which a simple Cartesian Product can be made. This is very simple, indeed it is sufficient to write a standard Select Query operating on two tables without expressing any join condition, as in the example shown below:

```
SELECT T1.*, T2.*
FROM T1, T2
```

As an alternative, it is also possible **to use the CROSS JOIN** operator:

```
SELECT T1.*, T2.*
FROM T1 CROSS JOIN T2
```

5. Self Join

A JOIN executed on a single table, characterized by a SRR, is called SELF JOIN.

Let us make a classical example. Suppose we want to know all the products of the PRODUCTS table that have exactly the same price. How can we do so?

First of all we need to create a duplicate of the PRODUCTS table (that we could call as PRODUCTS_1) and, next, we need to define a join relation between these two identical tables. This is shown in the SQL below:

```
SELECT PRODUCTS.ProductName, PRODUCTS.UnitPrice, PRODUCTS_1.ProductName, _
                PRODUCTS_1.UnitPrice
FROM PRODUCTS, PRODUCTS AS PRODUCTS_1
WHERE PRODUCTS.UnitPrice = PRODUCTS_1.UnitPrice
```

Note that the copy of the table is created in the FROM clause using the AS operator. This is called Aliasing technique. Formally, its syntax is the following one:

```
<Aliasing> : : =
FROM <original name> AS <new name>
```

Also note that the join condition is made on the UnitPrice Field. Indeed we want all the couples of products having the same price.

Lastly, we note that, as an alternative we could have written the query also in the following way:

```
SELECT PRODUCTS.ProductName, PRODUCTS.UnitPrice, PRODUCTS_1.ProductName, _
                                PRODUCTS_1.UnitPrice
FROM PRODUCTS INNER JOIN PRODUCTS AS PRODUCTS_1 ON _
                                PRODUCTS.UnitPrice = PRODUCTS _1.UnitPrice
```

Unfortunately, in both case we get a result that is not fully satisfactory, as shown in Table 5.3. As it can be seen, there are records that have the same product repeated twice. This is correct, because each product costs as itself. Thus, to get rid of duplicated field it is sufficient to filter the returned records. A possible way is the following one:

```
WHERE PRODUCTS.ProductID > PRODUCTS_1.ProductID.
```

Tab. 5.3.

Self-Join with redundant records

Product Name	Unit Price	Product Name	Unit Price
Clay flowerpot 6"	$1,75	Clay flowerpot 6"	$1,75
Lavender	$2,25	Lavender	$2,25
Beebalm	$2,75	Beebalm	$2,75
Gardening gloves (S)	$2,95	Gardening gloves (L)	$2,95
Gardening gloves (L)	$2,95	Gardening gloves (M)	$2,95
Gardening gloves (M)	$2,95	Gardening gloves (M)	$2,95
Gardening gloves (S)	$2,95	Gardening gloves (M)	$2,95
Hose saver	$2,95	Gardening gloves (M)	$2,95
Gardening gloves (L)	$2,95	Gardening gloves (S)	$2,95
Gardening gloves (L)	$2,95	Gardening gloves (L)	$2,95
Gardening gloves (S)	$2,95	Gardening gloves (S)	$2,95

Thus the full query becomes:

```
SELECT PRODUCTS.ProductName, PRODUCTS.UnitPrice, PRODUCTS_1.ProductName, _
                PRODUCTS_1.UnitPrice
FROM PRODUCTS, PRODUCTS AS PRODUCTS_1
WHERE PRODUCTS.UnitPrice = PRODUCTS_1.UnitPrice AND PROUCTS.UnitPrice > PRODUCTS_1.UnitPrice
```

Most of the times a Self Join is made to create a hierarchy, as in the case of an organizational chart. An example is shown in Table 5.4.

Tab. 5.4.

An example of SRR

ID	Last Name	Role	Chief
1	Red	CO	Null
2	Green	Commercial Manager	1
3	White	Production Manager	1
4	Brown	Seller	2
5	Blond	Department Head	3
6	Wolfstail	Worker	5
7	Pizzotti	Worker	5
8	Rolling	Seller	2

In this case the organizational table is very simple, as in the hierarchy there are only three distinct levels i.e., a worker respond to the Department Head that respond to the Production Manager that respond to the Chief Officer. Since there are three levels, if we wanted to build the whole organizational table we should use three Aliasing Tables. However, in this case, we should limit the query to the use of two aliasing tables (and so only two level of the hierarchy will be generated)

Once we have generated the aliasing tables we need to specify the Inner Join Condition and the join fields that, in this case, correspond to the ID and to the Chief fields. Also, since some records (typically one) may have no correspondences in the other tables we need to use a LEFT join, so as to return also those people that are at the vertex of the hierarchy and that do not have a chief.

Owing to what we said above, the resulting query is as follows

```
SELECT ORG_CHART.[Last Name], ORG_CHART.Role, Nz(LEV_I.[Last Name], "Nobody") AS [Responds to:], _
        Nz(LEV_II.[Last Name], "Nobody") AS [Who Responds to:]
FROM (ORG_CHART LEFT JOIN ORG_CHART AS LEV_I ON ORG_CHART.Chief= LEV_I.ID)
LEFT JOIN ORG_CHART AS LEV_II ON LEV_1.Chief = LEV_II.ID
```

Please note the use of the **Null-Zero f**unction NZ(*Field Name, [Substitution Field])*. This function takes two inputs. the first one is mandatory and is the field that must be evaluated. The second one, optional, is the value that will be returned when the value of the first field is null. If this optional value is not passed as input, then in case a null field the NZ function returns a zero. That is the reason why the function is called Null-Zero.

The query, made in Access, is visually shown by Fig. 5.3.

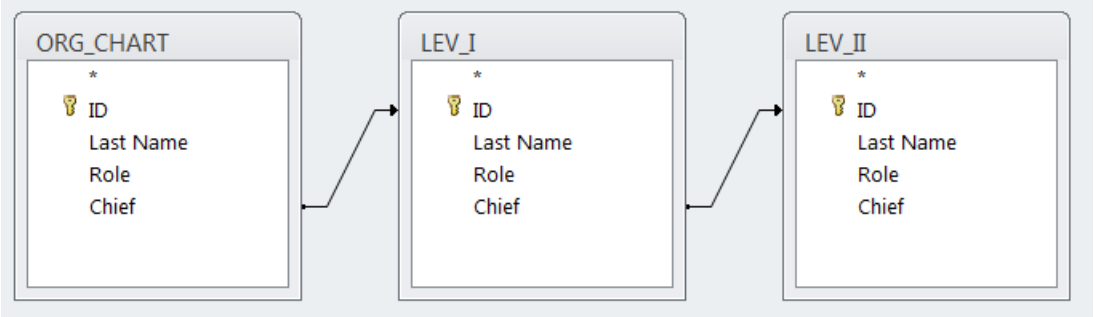

Fig. 5.3. *Aliasing and Left Join to recreate a hierarchy*

By executing the query the following result is obtained:

Tab. 5.5.
A query based on a SRR

Last Name:	Role	Responds To:	Who responds To:
Red	CO	Nobody	Nobody
Green	Marketing Manager	Red	Nobody
White	Production Manager	Redi	Nobody
Brown	Sellert	Green	Red
Blond	Department's Head	White	Redi
WolfsTail	Worker	Blond	White
Pizzotti	Worker	Blond	White
Rolling	Seller	Green	Red

6. Other operators

We conclude this section presenting three more useful operators: UNION, INTERSECT and EXCEPT

Union

The UNION performs the union of two homologous tables. Its syntax is as follows:

```
<First Select Command> UNION [ALL]
<Second Select Command>
```

The <*First Select Command*> and <*Second Select Command*> may be two Select queries that return, as output, two compatible table, or they may be a reference to a saved query.

ALL is an optional operator and it is used to return also duplicated records. Indeed, if ALL is not explicitly used, the UNION operator returns, exclusively, non-duplicated records.

Except and Intersect

EXCEPT implements, in SQL, the Difference operator of relational algebra.

The syntax is similar to that of the UNION operator, as shown below:

```
<First Select Command> EXCEPT [ALL]
<Second Select Command>
```

However, this time, the ALL operator has a slightly different meaning. Let us suppose that Except is made on T_1 and T_2. If T_1 and T_2 have a common record *r* that recurs *n* times in T_1 and *m* times in T_2, then in the output table <T_1 EXCEPT ALL T_2> the common record *r* will be repeated (*n*-*m*) times.

INTERSECT implements, in SQL, the Intersection operator of relational algebra.

The syntax is similar to that of the UNION operator, as shown below:

```
<First Select Command> INTERSECT [ALL]
<Second Select Command>
```

However, this time, the ALL operator has a slightly different meaning: if it is used all duplicated records will be displayed.

We conclude this section showing some simple query used as example. The first query is used to show all the items for which one Work Order (WO), at least, has been generated, i.e., the items that have been manufactured at least once.

```
SELECT ProductID
FROM PRODUCTS
INTERSECT
SELECT ProductID
FROM WORK_ORDERS
```

Conversely, if we were interested in the items that have never been manufactured, the following query should be written:

```
SELECT ProductID
FROM PRODUCTS
EXCEPT
SELECT ProductID
FROM WORK_ORDERS
```

Let us consider one more query:

```
SELECT ProductID
FROM WORK_ORDERS
EXCEPT
SELECT ProductID
FROM PRODUCTS
```

With respect to the previous one, only the order of the tables has changed. However, if we execute this query the output is an empty set. This result is not surprising as all Work orders must be associated to an Item. So it is not possible to find a WO of an item that is not present in the product's catalogue.

Unfortunately, Access does implement neither the EXCEPT nor the INTERSECT operator. However, these operators can be mimicked using some simple tricks. For example to get all the products that have never been manufactured we could write a concatenate query as the following one[1]:

```
SELECT ProductID
FROM PRODUCTS
WHERE ProductID NOT IN(SELECT ProductID FROM WORK_ORDERS)
```

Obviously the INTERSECT operator can be obtained in a similar way, provided that the NOT IN operator is substituted by the IN operator.

[1] Concatenated queries are the topics of the following chapter