9. Data Sub-Language SQL

9.1 Introduction

In this chapter, we shall learn more about the essentials of the relational model’s standard language that will allow us to manipulate the data stored in the databases. This language is powerful yet flexible, thus making it popular. It is in fact one of the factors that has led to the dominance of the relational model in the database market today.

Following Codd’s papers on the relational model and relational algebra and calculus languages, research communities were prompted to work on the realisation of these concepts. Several implemented versions of the relational languages were developed, amongst the most noted were SQL (Structured Query Language), QBE (Query-By-Example) and QUEL (Query Language). Here, we shall look into SQL with greater detail as it the most widely used relational language today. One often hears of remarks that say, “It’s not relational if it doesn’t use SQL”. It is currently being standardised now as a standard language for the Relational Data Model.

SQL had its origins back in 1974 from IBM’s System R research project as Structured English Query Language (or SEQueL) for use on the IBM VS/2 mainframes. It was developed by Chamberlain et al. The name was subsequently changed to Structured Query Language or SQL. It is pronounced “sequel” by some and S-Q-L by others. IBM’s products such as SQL/DS and the popular DB2 emerged from this. SQL is based on the Relational Calculus with tuple variables. In 1986, the American National Standards Institute (ANSI) adopted SQL standards, contributing to its widespread adoption. Whilst many commercial SQL products exist with various “dialects”, the basic command set and structure remain fairly standard.

Although SQL is called a query language, it is capable of more than just getting data off relations in the databases. It can also handle data updates and even data definitions—add new data, change existing data, delete or create new structures. Thus SQL is capable of:

1. **Data Query**
   The contents of the database are accessed via a set of commands whereby useful information is returned to the end user

2. **Data Maintenance**
   The data within the relations can be created, corrected, deleted and modified
3. *Data Definition*

The structure of the database and its relations can be defined and created.

The end user is given an interface, as we have seen in Chapter 3, to interact with the database via menus, query operations, report generators, etc. Behind this lies the SQL engine that performs the more difficult tasks of creating relation structures, maintaining the systems catalogues and data dictionary, etc.

SQL belongs to the category of the so-called Fourth-Generation Language (4GL) because of its power, conciseness and low-level of procedurality. As a non-procedural language it allows the user to specify *what* must be done without detailing *how* it must be done. The user’s SQL request specification is then translated by the RDBMS into the technical details needed to get the required data. As a result, the relational database is said to require less programming than any other database or file system environment. This makes SQL relatively easy to learn.

### 9.2 Operations

#### 9.2.1 Mapping: The SQL Select Statement

The basic operation in SQL is called *mapping*, which transforms values from a database to user requirements. This operation is syntactically represented by the following block:

```
Select (<Target List>)
From (<Relation>)
Where (<Predicate>)
```

*Figure 9-1. SQL Select*

This uncomplicated structure can used to construct queries ranging from very simple inquiries to more complex ones by essentially defining the conditions of the predicate. It thus provides immense flexibility.

The SQL Select command combines the Relational Algebra operators Select, Project, Join and the Cartesian Product. Because a single declarative-style command can be used to retrieve virtually any stored data, it is also regarded by many to be an implementation of the Relational Calculus. If we need to extract information from only one relation of the database, we may encounter similarities and a few differences between the Relational Calculus-based DSL Alpha and SQL. In this case we may substitute key words of DSL Alpha for matching key words of SQL as follows:
Let us refer back to the earlier example with the Customer relation.

Suppose we wish to “Get the names and phone numbers of customers living in London”. With DSL Alpha, we would specify this query as:

```plaintext
Range Customer X;
    Get (X.Cname, X.Cphone): X.Ccity=London;
```

whereas in SQL its equivalent would be:

```sql
Select  Cname, Phone 
From Customer 
Where Ccity = 'London'
```

In either case, the result would be the retrieval of the following two tuples:

<table>
<thead>
<tr>
<th>C#</th>
<th>Name</th>
<th>City</th>
<th>Phone</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Codd</td>
<td>London</td>
<td>2263035</td>
</tr>
<tr>
<td>2</td>
<td>Martin</td>
<td>Paris</td>
<td>5555910</td>
</tr>
<tr>
<td>3</td>
<td>Deen</td>
<td>London</td>
<td>2234391</td>
</tr>
</tbody>
</table>

This simple query highlights the three most used SQL clauses:

1. The SELECT clause
   This effectively gets the *columns* that we are interested in getting from the relation.
   We may be interested in a single column, thus we may for example write “Select Cphone” if we only wish to list just the telephone numbers. We may also however be interested in listing the customer’s name, city and telephone number; in which case, we write “Select Cname, Ccity, Cphone”.

**Figure 9-2.** Similarities of DSL Alpha and SQL Select
2. The FROM clause
We need to identify the *relations* that our query refers to and this is done via the From clause. The columns that we have chosen from the Select clause must be found in the relation names of the From clause as in “From Customer”.

3. The WHERE clause
This holds the *conditions* that allows us to restrict the tuples of the relation(s). In the example “Where Ccity=London” asserts that we wish to select only the tuples which contain the city name that is equal to the value ‘London’.

The system first processes the From clause (and all tuples of the chosen relation(s) are placed in the processing work area), followed by the Where clause (which chooses, one by one, the tuples that satisfy the clause conditions and eliminating those which do not), and finally the Select clause (which takes the resultant tuples and displays only the values under the Select clause column names).

### 9.2.2 Output Restriction

Most queries do not need every tuple in the relation but rather only a subset of the tuples. As described previously in section 5.3, the following mathematical operators can be used in the predicate to restrict the output:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>=</td>
<td>Equal to</td>
</tr>
<tr>
<td>&lt;</td>
<td>Less than</td>
</tr>
<tr>
<td>&gt;</td>
<td>Greater than</td>
</tr>
<tr>
<td>&lt;=</td>
<td>Less than or equal to</td>
</tr>
<tr>
<td>&gt;=</td>
<td>Greater than or equal to</td>
</tr>
<tr>
<td>&lt;&gt;</td>
<td>Not equal to</td>
</tr>
</tbody>
</table>

Additionally, the logical operators AND, OR and NOT may be used to place further restrictions. These logical operators, along with parentheses, may be combined to produce quite complex conditional expressions.

Suppose we need to retrieve the tuples from the Transaction relation such that the following conditions apply:

1. The transaction date is before 26 Jan and the quantity is at least 25
2. Or, the customer number is 2

The SQL statement that could get the desired result would be:

```
Select C#, Date, Qnt From Transaction
   Where (Date < '21.01' And Qnt >= 25) Or C# = 2
```
9.2.3 Recursive Mapping: Sub-queries

The main idea of SQL is the recursive usage of the mapping operation instead of using the existential and universal quantifiers. So far in our examples, we always know the values that we want to put in our predicate. For example,

Where Ccity = ‘London’  
Where Date < ’26.01’ And Qnt > 25

Suppose we now wish to “Get the personal numbers of customers who bought the product CPU”. We could start off by writing the SQL statement:

Select C#  
From Transaction  
Where P#= ?

We cannot of course write “Where P#=CPU” because CPU is a part name not its number. However as we may recall, part number P# is stored in the Transaction relation, but the part name is in fact in another relation, the Product relation. Thus one needs to first of all get the part name from Product via another SQL statement:

Select P#  
From Product  
Where Pname = ‘CPU’

Having obtained the equivalent P#, the value is then used to complete the earlier query. The way this is to be expressed is by writing the whole mapping operator in the right hand side of comparison expressions of another mapping operator. This effectively means the use of an inner block (sub-query) within the outer block (main query) as depicted in the figure below.

![Figure 9-3. Query nesting](image)

The query in the outer block thus executes by using the value set generated earlier by the sub-query of the inner block.

It is important to note that because the sub-query replaces the value in the predicate of the main query, the value retrieved from the sub-query must be of the same domain as the value in the main predicate.
9.2.4 Multiple Nesting

It is also possible that may be two or more inner blocks within an outer SQL block. For instance, we next wish to: “Get a date when customer Codd bought the product CPU”. The SQL statement we would start out with would probably look like this:

```
Select Date
From Transaction
Where P#=?
And C#=?
```

As in the earlier query, the part number P# can be obtained via the part name Pname in the relation Product. The customer name, Codd, however has to have its equivalent customer number which has to be obtained from C# of the relation Customer. Thus to complete the above query, one would have to work two sub-queries first as follows:

```
Select Date
From Transaction
Where P# =
  ( Select P#
    From Product
    Where Pname = 'CPU')
And C# =
  ( Select C#
    From Customer
    Where Cname = 'Codd')
```

**Figure 9-4. Interpretation of sub-queries**

Note that the original SQL notation utilises brackets or parentheses to determine inner SQL blocks as:

```
Select Date
From Transaction
Where P# =
  ( Select P#
    From Product
    Where Pname = CPU)
And C# =
  ( Select C#
    From Customer
    Where Cname = Codd)
```
Similarly, an inner block may contain further inner SQL blocks. For instance, if we wish to “Get the names of customers who bought more than 20 pieces of the product CPU” we need to specify:

```
Select Cname
From Customer
Where C# =
    ( Select C#
        From Transaction
        Where P# =
            ( Select P#
                From Product
                Where Pname = CPU )
        And Qnt > 20 )
```

Thus we may visualise the nesting of sub-queries as:

```
Select …
From …. Where <attribute1> <operator>
    ( Select <attribute1>
        From …
        Where <attribute2> <operator>
            ( Select <attribute2>
                From …
                Where <attribute3> <operator>
                    ( Select <attribute3>
                        From …
                        Where … ) ) )
```

The number of inner blocks or levels of nesting may, however, be limited by the storage available in the workspace of the DBMS in use.

### 9.2.5 Multiple Data Items

Standard comparison operators ( =, >, <, >=, <=, <> ) operate on two data items, as in \( x = y \) or \( p \geq 4 \). They cannot be applied to multiple data items. However, a particular SQL block normally returns a set of values (i.e. not a single value which can be used in a comparison).
For instance: “Get the product numbers of items which were bought by customers from London”.

```
Select P#
From Transaction
Where C# =
  ( Select C#
    From Customer
    Where Ccity = ‘London’ )
```

Given the sample database of the earlier examples, the result of the inner SQL block would yield two values for C#, which are 1 and 3, (or more precisely, the set { 1, 3 } ). The outer SQL block, in testing C# = { 1, 3 } would effectively test if { 1,2 } = { 1, 3 } or not. Thus the above SQL statement is not correct!

To overcome the error caused by the testing of multiple values returned by the sub-query, SQL allows the use of comparison expressions in the form:

```
<attribute name> \{ In, Not In \} <set of values>
```

This logical expression is true if the current value of an attribute is included (or not included, respectively in the set of values.

For instance,

- Smith In { Codd, Smith, Deen } is True,
- Smith Not In { Codd, Smith, Deen } is False.

Thus in re-writing the earlier erroneous statement, we now replace the equal operator (=) with the set membership operator ‘In’ as follows:

```
Select P#
From Transaction
Where C# In
  ( Select C#
    From Customer
    Where Ccity = ‘London’ )
```

This time it would yield the outer SQL block would effectively test C# in { 1, 3 }. The outer SQL block would now only retrieve the P#s that are only in the set { 1, 3 }, i.e. testing { 1, 2 } In { 1, 3 }. This would result in returning P# 1 only, which is the expected right answer.
Illustrating with another example, consider the query to “Find the names of customers who bought the product CPU”. Its corresponding SQL statement would thus be:

```
Select Cname From Customer
Where C# In

  ( Select C# From Transaction
    Where P# In

    ( Select P# From Product
      Where Pname = ‘CPU’ ) )
```

Executing this step-by-step:

1. From the inner-most block,
   ```
   Select P# From Product
   Where Pname = CPU
   ```
   would first yield P# 1 from Product, i.e. { 1 }.

2. The next block, would thus be
   ```
   Select C# From Transaction
   Where P# In { 1 }
   ```
   and this would yield C# s 1 and 2 (as they bought P# 1), i.e. { 1, 2 }

3. And finally, the outer-most block would execute
   ```
   Select Cname From Customer
   Where C# In { 1, 2 }
   ```
   would result in the names of customers 1 and 2, which are Codd and Martin respectively.

We next go on to a slightly more complex example. Suppose we now wish to “Get a name of such customers who bought the product CPU but did not buy the product VDU”.

In SQL, the statement would be:

```
Select Cname From Customer
Where C# In

  ( Select C# From Transaction Where P# In
    ( Select P# From Product Where Pname = ‘CPU’ )
  And C# Not In
    ( Select C# From Transaction Where P# In
      ( Select P# From Product Where Pname = ‘VDU’ )
    )
)
Why don’t you try to figure out, step-by-step, the sequence of results from the inner-most blocks up to the final result of execution of the outer-most block?

Note that the comparison operators

\[
\begin{align*}
\text{<attribute name>} & \quad \text{In} \quad \{ \text{set of values} \} \\
\text{<attribute name>} & \quad \text{Not In} \\
\end{align*}
\]

are used instead of existential qualifiers (\(\exists\)). It is an implementation of multiple logical OR conditions which is more efficiently handled.

Similarly, comparison expressions

\[
\text{<attribute name>} = \text{ALL} \quad \text{<set of values>}
\]

are used instead of universal qualifiers (\(\forall\)).

This logical expression is valid (i.e. produces the logical value “True”) if the collection of attribute name values in the database includes the given set of values.

For instance, “Get personal numbers of those customers who bought all kinds of company’s products”, would have the following SQL statement for it:

\[
\begin{align*}
\text{Select C# From Transaction} \\
\text{Where P# =} \\
\text{ALL ( Select P#} \\
\text{From Product )}
\end{align*}
\]

The inner block would yield the set \{ 1, 2 \} of P# values. Executing the outer block would effectively test if the 3 customers in the Transaction relation, i.e. C# 1, 2 and 3 would have P# in \{ 1, 2 \}.

This test is as follows:

<table>
<thead>
<tr>
<th>C#</th>
<th>Transaction (C#, 1)</th>
<th>Transaction (C#, 1)</th>
<th>All P#</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>True</td>
<td>True</td>
<td>True  !</td>
</tr>
<tr>
<td>2</td>
<td>True</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>3</td>
<td>False</td>
<td>True</td>
<td>False</td>
</tr>
</tbody>
</table>

The only customer that has P# equal to all P# as found in Product would be C# 1.

9.3 Further Retrieval Facilities

9.3.1 Joining Relations

In the examples that have been used so far, our retrievals have been of values taken from one relation, as in “Select C# From Transaction”. However, often we have to
retrieve information from two or more relations simultaneously. In other words, a
number of relations names may be used in the From clause of SQL. For example, if
we wish to access the relations Customer and Transaction, we may write the SQL
statement as follows:

Select …
From Customer, Transaction
Where…

The target list in the Select clause may contain the attributes form various relations, as in

Select Cname, Date, Qnt
From Customer, Transaction
Where…

where, if you recall, Cname is an attribute of Customer and Date and Qnt are
attributes of Transaction.

Similarly, comparison expressions in the Where clause may include attribute names
from various relations,

Select Cname, Date, Qnt
From Customer, Transaction
Where (Customer.C# = Transaction.C#) And P# = 1

Note that a so-called qualification technique which is used to refer to attributes of the
same name belonging to different relations. Customer.C# refers to the C# of the
Customer relation whereas Transaction.C# refers to the C# of the Transaction relation.

Thus the query “Get customer names, dates and number of pieces for transactions of
the product number 1” will result in:

<table>
<thead>
<tr>
<th>Cname</th>
<th>Date</th>
<th>Qnt</th>
</tr>
</thead>
<tbody>
<tr>
<td>Codd</td>
<td>21.01</td>
<td>20</td>
</tr>
</tbody>
</table>

It must be noted that the two (or more) relations that must be combined on at least one
common linking attribute (as in the Relational Algebra’s JOIN operator). As in the
above example, the link is established on C# as in the clause

Where Customer.C# = Transaction.C#
9.3.2 Alias

In order to avoid a possible ambiguity in a query definition SQL also allows to use an alias for the relation name in the From clause. The alias is an alternate name that is used to identify the source relation and the attribute names may include an alias as a prefix:

<alias>.<attribute name>

Suppose we use T and C as the aliases for the Transaction and Customer relations respectively. We may use these to label the attributes as in:

Select ... From Customer C, Transaction T
Where C.C# = T.C# And …

An alias is especially useful when we wish to join a relation to itself because of grouping as in the query to “Find the names and phone numbers of customers living in the same city as the customer Codd”:

Select C2.Cname, C2.Cphone
From Customer C1, Customer C2
Where C2.Ccity = C1.Ccity
And C1.Cname = ‘Codd’

The resulting interpretation of the SQL statement is depicted in Figure 9-6 below:

9.4 Library Functions and Arithmetic Expressions

The SQL Select clause (target list) may contain also so-called SQL library functions that will perform various arithmetic summaries such as to find the smallest value or to sum up the values in a specified column. The attribute name for such library functions must be derived from the relations specified in the From clause as follows:
The common SQL functions available are:

<table>
<thead>
<tr>
<th>Function name</th>
<th>Task</th>
</tr>
</thead>
<tbody>
<tr>
<td>COUNT</td>
<td>To count the number of tuples containing a specified attribute value</td>
</tr>
<tr>
<td>SUM</td>
<td>To sum up the values of an attribute</td>
</tr>
<tr>
<td>AVG</td>
<td>To find the arithmetic mean (average value) of an attribute</td>
</tr>
<tr>
<td>MAX</td>
<td>To find the maximum value of an attribute</td>
</tr>
<tr>
<td>MIN</td>
<td>To find the minimum value of an attribute</td>
</tr>
</tbody>
</table>

Examples

(1) Get the average quantity of VDUs per transaction

```sql
Select AVG(Qnt) From Transaction
Where P# =
  ( Select P# From Product
    Where Pname = 'VDU' )
```

Working first with the inner Select clause, we get a P# of 2 from the Product relation as the part number for the product named VDU. Thus the query is now reduced to

```sql
Select AVG(Qnt) From Transaction
Where P# = 2
```

Accessing the Transaction relation now would yield the following two tuples

<table>
<thead>
<tr>
<th>C#</th>
<th>P#</th>
<th>Date</th>
<th>Qnt</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>21.01</td>
<td>20</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>23.01</td>
<td>30</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>26.01</td>
<td>25</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>29.01</td>
<td>20</td>
</tr>
</tbody>
</table>

where the average quantity value is easily computed as \((30+20)/2\) which is 25.
Get the total quantity of VDUs transacted would similarly be expressed as:

```
Select SUM (Qnt) From Transaction
Where P# =
    ( Select P# From Product
      Where Pname = ‘VDU’ )
```

where the total value is easily computed as \((30 + 20)\) giving 50.

An asterisk (*) in the Select clause is interpreted as “all attributes names of the relations specified in the From clause”.

```
Select * From Transaction
is equivalent to
Select C#, P#, Date, Qnt From Transaction
```

Thus a query to “Get all available information on customers who bought the product VDU” can be written as:

```
Select * From Customer
Where C# In
    ( Select C# From Transaction
      Where P# In
        ( Select P# From Product
          Where Pname = ‘VDU’ ) )
```

The interpretation of this query would be worked out as shown in the following sequence of accesses, starting from the access of the product relation to the Transaction and finally to the Customer relation:

![Figure 9-8](image)

The outcome would be the following relation:

<table>
<thead>
<tr>
<th>C#</th>
<th>Cname</th>
<th>Ccity</th>
<th>Cphone</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Codd</td>
<td>London</td>
<td>2263035</td>
</tr>
<tr>
<td>2</td>
<td>Martin</td>
<td>Paris</td>
<td>5555910</td>
</tr>
</tbody>
</table>
(3) Get a total number of such customers who bought the product VDU, would be written as:

```
Select COUNT (*) From Customer
Where C# In
  ( Select C# From Transaction
    Where P# In
      ( Select P# From Product
        Where Pname = 'VDU' ) )
```

and this would yield a value of 2 for Count (*).

Arithmetic expressions are also permitted in SQL, and the possible operations include:

- addition +
- subtraction -
- multiplication *
- division /

Expressions may be written in the Select clause as:

```
Select C#, P#, Qnt*Price From Transaction, Product
Where Transaction.P# = Product.P#
```

which is used to “Get a total price for each transaction” resulting in:

<table>
<thead>
<tr>
<th>Product</th>
<th>Transaction</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>P#</td>
<td>Pname</td>
<td>Price</td>
</tr>
<tr>
<td>1</td>
<td>CPU</td>
<td>1000</td>
</tr>
<tr>
<td>1</td>
<td>VDU</td>
<td>1200</td>
</tr>
<tr>
<td>2</td>
<td>VDU</td>
<td>1200</td>
</tr>
<tr>
<td>2</td>
<td>VDU</td>
<td>1200</td>
</tr>
</tbody>
</table>

Arithmetic expressions, likewise, can also be used as parameters of SQL library functions. For example, “Get a total price of all VDUs sold to customers” may be written as the following SQL statement:

```
Select SUM (Qnt*Price) From Transaction, Product
Where Transaction.P# = Product.P#
And Product.Pname = 'VDU'
```

Work this out. You should get an answer of 60000.

The attribute names for both library functions and arithmetic expressions must be derived from the relations specified in the From clause.
Thus, it should be noted that the following query definition is NOT correct.

```
Select SUM (Qnt*Price) From Transaction
Where Transaction.P# = Product.P#
And Product.Pname = ‘VDU’
```

Additionally, SQL also permits the use of library functions not only in the Select clause but also in the Where clause as a part of comparison expressions.

The query to “Get all available information on such customers who bought the most expensive product” would be:

```
Select * From Customer
Where C# In
  ( Select C# From Transaction
    Where P# In
      ( Select P# From Product
        Where Price = (SELECT MAX(Price) FROM Product)))
```

## 9.5 Additional Facilities

### 9.5.1 Ordering

The result of a mapping operation may be sorted in ascending or descending order of the selected attribute value.

The form of the Order clause is

```
Order By <attribute name> Up | Down
```

#### Examples

(1) Get a list of all transactions of the product CPU sorted in descending order of the attribute Qnt

```
Select * From Transaction
Where P# In
  ( Select P# From Product
    Where Pname = ‘CPU’ )
Order By Qnt Down
```

The result would be

<table>
<thead>
<tr>
<th>C#</th>
<th>P#</th>
<th>Date</th>
<th>Qnt</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1</td>
<td>26.01</td>
<td>25</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>21.01</td>
<td>20</td>
</tr>
</tbody>
</table>
If instead, the last clause had been “Order By Qnt Up”, the result would be listed in ascending order:

<table>
<thead>
<tr>
<th>C#</th>
<th>P#</th>
<th>Date</th>
<th>Qnt</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>21.01</td>
<td>20</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>26.01</td>
<td>25</td>
</tr>
</tbody>
</table>

The Order By clause is only a logical sorting process, the actual contents of the original relations are not affected.

Multi-level ordered sequence may also be performed as in:

Select * From Transaction
  Order By C# Up,
      Qnt Down

9.3.2 Handling Duplicates

The result of an SQL mapping operation is however not perceived as a relation, i.e. it may include duplicate tuples. Consider for example:

Select C# From Transaction
  Where P# In
      ( Select P# From Product
          Where Price >= 1000 )

The result is actually

<table>
<thead>
<tr>
<th>C#</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
</tbody>
</table>

Imagine if we have thousands of transactions and yet a handful of customers. The result would yield hundreds (even thousands) of duplicates. Fortunately, duplicate tuples can be removed by using the Unique option in the Select clause of the operation as follows:

Select C# Unique From Transaction
  Where P# In
      ( Select P# From Product
          Where Price >= 1000 )
and this will yield a much reduced result with only the distinct (unique) customer numbers:

```
<table>
<thead>
<tr>
<th>C#</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
</tbody>
</table>
```

### 9.3.3 Grouping of Data

Usually, the result of a library function is calculated for the whole relation. For example, consider wanting to find the total number of transactions,

```
Select Count (*)
From Transaction
```

However, sometimes we need to calculate a library function, not for the entire relation, but only for a subset of it. Such subsets of tuples are called groups. For instance, in the relation Transaction, a collection of tuples with the same value of attribute C# is a “group”. In this case, C# is called “Group By” attribute.

```
<table>
<thead>
<tr>
<th>C#</th>
<th>P#</th>
<th>Date</th>
<th>Qnt</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>21.01</td>
<td>20</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>23.01</td>
<td>30</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>26.01</td>
<td>25</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>29.01</td>
<td>20</td>
</tr>
</tbody>
</table>
```

Given this relation, the result of Count (*) is 4

```
<table>
<thead>
<tr>
<th></th>
<th>C#=1</th>
<th>C#=2</th>
</tr>
</thead>
<tbody>
<tr>
<td>11.01</td>
<td>20</td>
<td>25</td>
</tr>
</tbody>
</table>
```

**Figure 9-9.** Grouping by customer numbers

The form of the Group By clause is

**Group By <attribute name>**

**Examples**

1. “Get the list of all customer numbers and the quantity of products bought by each of them”. Note that the relation will have many transactions for any one customer. The
transactions for each customer will have to be grouped and the quantities totaled. This is then to be done for each different customer. Thus the SQL statement would be:

Select C#, Sum(Qnt) From Transaction Group By C#

Thus all transactions with the same C#s are grouped together and the quantities summed to yield the summarised result:

<table>
<thead>
<tr>
<th>Transaction</th>
<th>C#</th>
<th>P#</th>
<th>Date</th>
<th>Qnt</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>21.01</td>
<td>20</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>2</td>
<td>23.01</td>
<td>30</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>1</td>
<td>26.01</td>
<td>25</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>2</td>
<td>29.01</td>
<td>20</td>
</tr>
</tbody>
</table>

C# = 1

C# = 2

Why would the following statement be impossible to execute?

Select * From Transaction Group By P#

(2) Normally, the Where clause would contain conditions for the selection of tuples as in:

Select Cname, Sum (Qnt) From Customer, Transaction
Where Customer.C# = Transaction.C#
Group By C#

This statement will “Get a list of all customer names and the quantity of products bought by each of them” as follows:

<table>
<thead>
<tr>
<th>Transaction</th>
<th>C#</th>
<th>P#</th>
<th>Date</th>
<th>Qnt</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>21.01</td>
<td>20</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>2</td>
<td>23.01</td>
<td>30</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>1</td>
<td>26.01</td>
<td>25</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>2</td>
<td>29.01</td>
<td>20</td>
</tr>
</tbody>
</table>

C# = 1

C# = 2

<table>
<thead>
<tr>
<th>Customer</th>
<th>C#</th>
<th>Cname</th>
<th>Ccity</th>
<th>Cphone</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>Codd</td>
<td>London</td>
<td>2263035</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>Martin</td>
<td>Paris</td>
<td>5555910</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>Deen</td>
<td>London</td>
<td>2234391</td>
</tr>
</tbody>
</table>

Result:

<table>
<thead>
<tr>
<th>Cname</th>
<th>Sum(Qnt)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Codd</td>
<td>50</td>
</tr>
<tr>
<td>Martin</td>
<td>45</td>
</tr>
</tbody>
</table>

Figure 9-10. Restriction followed by Grouping
9.3.4 Further Filtering: Having

We can further filter out unwanted groups generated by the Group By clause by using a “Having” clause which will include in the final result only those groups that satisfy the stated condition. Thus the additional “Having” clause provides a possibility to define conditions for selection of groups.

For example, if we wish to just “Get such customers who bought more than 45 units of products”, the SQL statement would be:

```
Select * From Customer
Where C# In
  ( Select C# From Transaction
    Group By C#
    Having SUM (Qnt) > 45 )
```

![Figure 9-11. Grouping followed by Restriction](image)

In this case, those grouped customers with 45 units or less will not be in the final result. The result will thus only be:

```
C# | Cname | Ccity | Cphone
---|-------|-------|-------
1  | Codd  | London| 2263035
```

It is important to note that in the further filtering of values, the Where clause is used to exclude values before the Group By clause is applied, whereas the having clause is used to exclude values after they have been grouped.