Components of SQL SELECT Statement

SQL Scripts can be used to perform any function in a database. SQL can be used to delete, update, alter and insert data. It can also be used to update a database's inner workings, rules and structures of data-tables, as well as many further functions such as which users have rights to access which parts of the database. This "Manipulation" SQL is **not** the subject of this document.

This document focuses on the SQL used to retrieve data from a database, known as "SELECT SQL". All "SELECT SQL" statements begin with the key-word SELECT.

Full SQL, including all its functionality is a complex computer language, which can take many years to master fully. SELECT SQL is a small sub-set of the whole language, and is much easier to grasp.

> **Do not get too worried about the complex nature of SQL. The EXE you use has hundreds of SQL Statements already written which are used to run it. You can copy these statements and use them yourself. Just making small changes to existing scripts will probably be enough to allow you to make many useful reports when you start working with SQL. After experimenting in this way your knowledge of SQL can expand until you become more knowledgeable. There are extremely widespread sources of information and training about SQL on the internet, and at the ElevateDB website.**

SQL SELECT

> **In this section of help we will only look at the SQL SELECT Statement. This is any SQL statement sent to the database which starts with the word SELECT and does not usually contain complex fuctions, scripts or procedures.**

> **The current section will briefly introduce each section of the SQL SELECT statement, subsequent sections will go into the language in more detail.**

> **SQL SELECT Statements always start with the keyword "SELECT".**

SELECT Statements are used to retrieve data from a database so that it can be shown in a report, list, grid on screen or any other application GUI such as a chart or pivot table. Third party products such as Microsoft Excel or Word also include tools (called ODBC drivers) which allow database data to be displayed in spreadsheets and documents through the use of SQL SELECT statements.

SELECT Statements retrieve data from database tables, but they can also include data from database views or return and use data from database functions. Detailed understanding of these different database data formats is not expected. Simple examples will show how these different types of data can be incorporated.

In SQL it is normal for Keywords (such as SELECT) to be written in capital letters, while other elements (such as the names of tables or data-fields) are written in a mix of lower case or with the first letter capitalized. When a SQL Script is sent to the database is it "compiled" and then run so the database can return the data. The database does not care whether words are capitalized or not, but it is important to follow capitalization rules in order to keep your scripts readable.

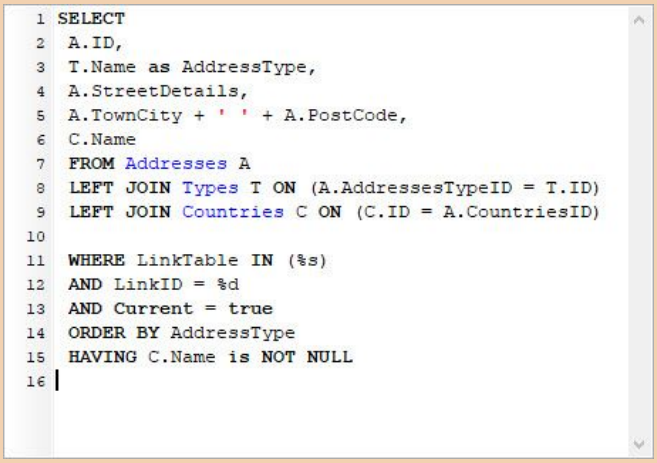

Image 01: Example SQL Script, displayed in an editor which formats the code to make it easier to read

An example of a simple SQL SELECT Statement

```
SELECT
A.ID,
T.Name as AddressType,
A.StreetDetails,
A.TownCity + ' ' + A.PostCode,
C.Name
FROM Addresses A
LEFT JOIN Types T ON (A.AddressesTypeID =
T.ID)
LEFT JOIN Countries C ON (C.ID =
A.CountriesID)

WHERE LinkTable IN (%s)
AND LinkID = %d
AND Current = true
ORDER BY AddressType
HAVING C.Name is NOT NULL
```

SQL "SELECT" Keyword

This word is used to start a SELECT statement. After the SELECT keyword, but before the FROM Keyword script elements are listed which refer to columns in data-tables, constants and functions.

The section of a SELECT statement between the SELECT Keyword and the FROM Keyword is called the SELECT Section of the SELECT script.

The SELECT Section contains elements separated by commas. Usually these elements are simply column-names. When multiple data-tables are referenced in a the FROM Section of a SELECT statement these column-names can be prefixed with a table-identifier. In the above image "A.TownCity refers to the "TownCity" column in the Addresses data-table.

Constants and Functions in the database can also be used. In the above example (Image 01) the empty text ' ' is added as a constant that separates the "TownCity" and "PostCode" fields. No functions are used in the example.

SQL "FROM" and "JOIN" Keywords

At the end of the SELECT Section the final column element is **not** followed by a comma. Instead the FROM keyword is added.

The FROM Keyword must be followed by the name of a data-table. This is the "lead table" for the SELECT statement. Usually this will be the table from which most data in the SELECT statement is retrieved.

The lead-table name can be followed by a space, and then an optional identifier for the table-name. In the image "Addresses A" has been written. This means the SQL statement is looking for records in the "Addresses" table, and that any column-name prefixed with "A." in the SELECT section must be from the Addresses table. The "A." statement is used as a reference for "Addresses.", as it is shorter and easier to write.

After the lead-table name other data-tables can be JOINed into the SELECT statement. JOINing tables is optional, if all the data you require is in a single table no JOINs are necessary.

The JOIN is the key aspect of **relational** databases which makes them powerful. Using JOINs allows data from different tables to be brought together and viewed.

In Orixa JOINs are almost always written in the form:

```
LEFT JOIN [child or extension data-table name] AS [identifier] ON [identifier].[foreignkeyID-fieldname] = [leadtable-identifier].ID
```

Orixa enforces the standard that all tables have an ID column as their primary key The ID is always an integer (whole number). This makes it fairly easy to write all JOINs and to create all relationships between data-tables.

Occasionally it may be necessary to use other forms of JOIN. Orixa supports JOINs of all forms that are standard to SQL-ISO2003. However these more complex JOINs are beyond the scope of this help document.

SQL "WHERE" Keyword

The FROM Section of the SELECT statement is followed by an optional WHERE Section. This starts with the WHERE Keyword, and consists of a list of boolean (true/false) statements.

When the SELECT statement is run only records matching the values laid out on the WHERE statement will be returned.

Sections of a WHERE statement must be separated by **AND** or **OR** keywords**.** Paranthases can be added to ensure evaluation takes place in the desired order.

WHERE statements can contain **SUB SELECT** statements, ie secondary SELECT Statements which themselves return a set of values.

Examples:

```
WHERE ID = 123
```

Only 1 record will be returned. If no record with ID = 123 exists, no records will be returned.

```
WHERE ID IN (123, 455, 432)
```

A maximum of 3 records will be returned.

```
WHERE Name LIKE '%Bill%'
```

Records containing Bill with any text before or afterwards will be returned.

```
WHERE DateDone > Current_Date
```

Records where the "DateDone" field is greater than the date on which the SELECT statement is run will be returned.

```
WHERE ID IN (SELECT ID FROM [SomeTable] WHERE StaffID = 123)
```

An example of a simple sub-select statement. When the main SELECT statement runs, this sub-select statement will run first, and provide a list of IDs that will be returned for this WHERE statement

```
WHERE (ID, Name) IN (SELECT ID, Name FROM [SomeTable] WHERE StaffID = 123)
```

Note that the sub-select statement can contain more than one field, provided that within the WHERE statement the number of fields on each side of the "IN" keyword are the same.

SQL "GROUP BY" Keyword

It is frequently useful to SUM together rows in a SELECT statement. For example we might SUM the total value of sales for each customer. This is done in a SELECT statement by returning **all** the data rows, and then **grouping** them.

It is important to understand the logic: The SELECT returns all the rows, then the GROUP BY statement takes these rows and reduces them so that those columns listed in the GROUP BY section of the statement are summarized.

```
SELECT
  O.Name + ' ' + C.CustomerCode as CustomerDetails,
  SUM(TotalValue) as CustomerTotalSales
FROM Customers C
LEFT JOIN Organisations O ON O.ID = C.ID
LEFT JOIN Sales S ON C.ID = S.CustomersID
WHERE C.CustomerCode LIKE 'WS-%'
AND S.DateDone > DATE '2020-01-01'
GROUP BY CustomerDetails
```

The above SELECT Statement would return a dataset with one row for each Customer who's Customer code started with "WS-", each row would include a column "CustomerTotalSales" which would contain a SUM of all the TotalValue columns for that customer.

GROUP BY sections are necessary to allow systems to display all sorts of summary data including SUMs AVG (average) MIN, MAX and COUNT.

In the above SQL statement note the use of a date constant. The script: "DATE '2020-01-01' " has been added. This means that only sales with DateDone greater than first January 2020 will be returned.

The SQL standard for dates is that the string constant used in the SQL statement should always be in the form: YYYY-MM-DD.

SQL "HAVING" Keyword

The HAVING keyword is optional and functions in the same way as the WHERE keyword.

The main difference between the WHERE and the HAVING sections of a SELECT statement is that the WHERE section is triggered**before** the dataset is returned, but the HAVING section is run on the returned dataset.

It is rare that a HAVING section is needed. It is really only useful with more complex SELECT statements that include computed columns returned by a funtion, since the outcome of this computation is not always available during the operation of the WHERE section.