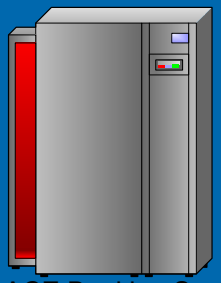
SQL – An Introduction

Using some basic SQL statements to retrieve data from SQL databases

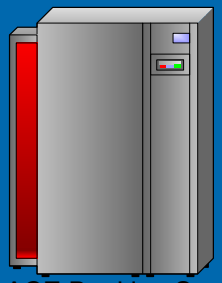
by David Smith

Border Asset Management Ltd

The last 18 months!



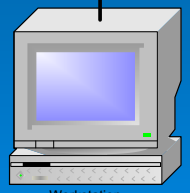
RBS ACE Banking System
All client financial records and
dealing information held in Scotland



RBS ACE Banking System
All client financial records and
dealing information held in Guernsey

In-House SQL Database Server
All client KYC data and
'copy' of data held at RBS for
Fund Manager decision making

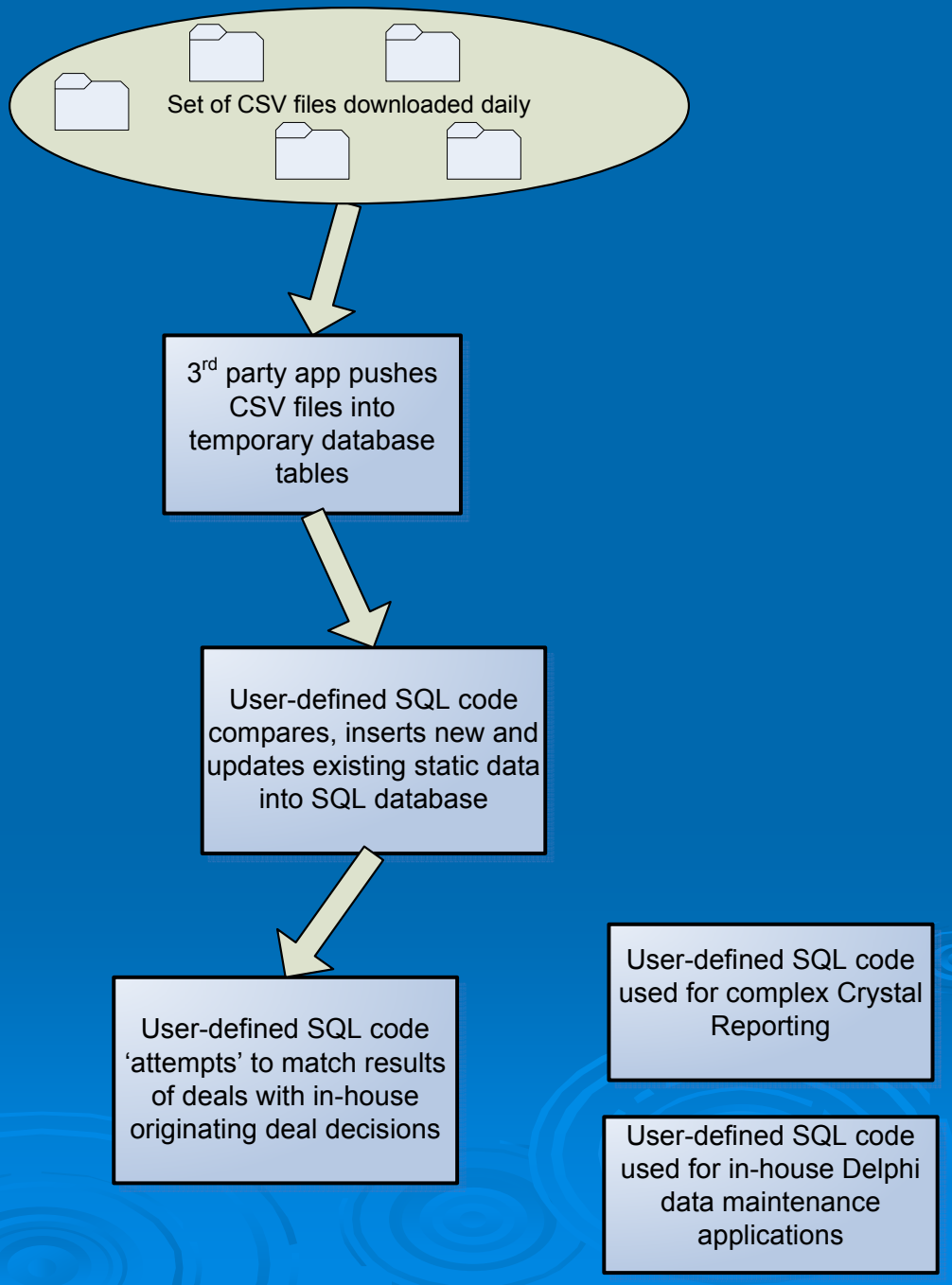
In-House Terminal Server
running Windows based
Fund Management application



Workstation

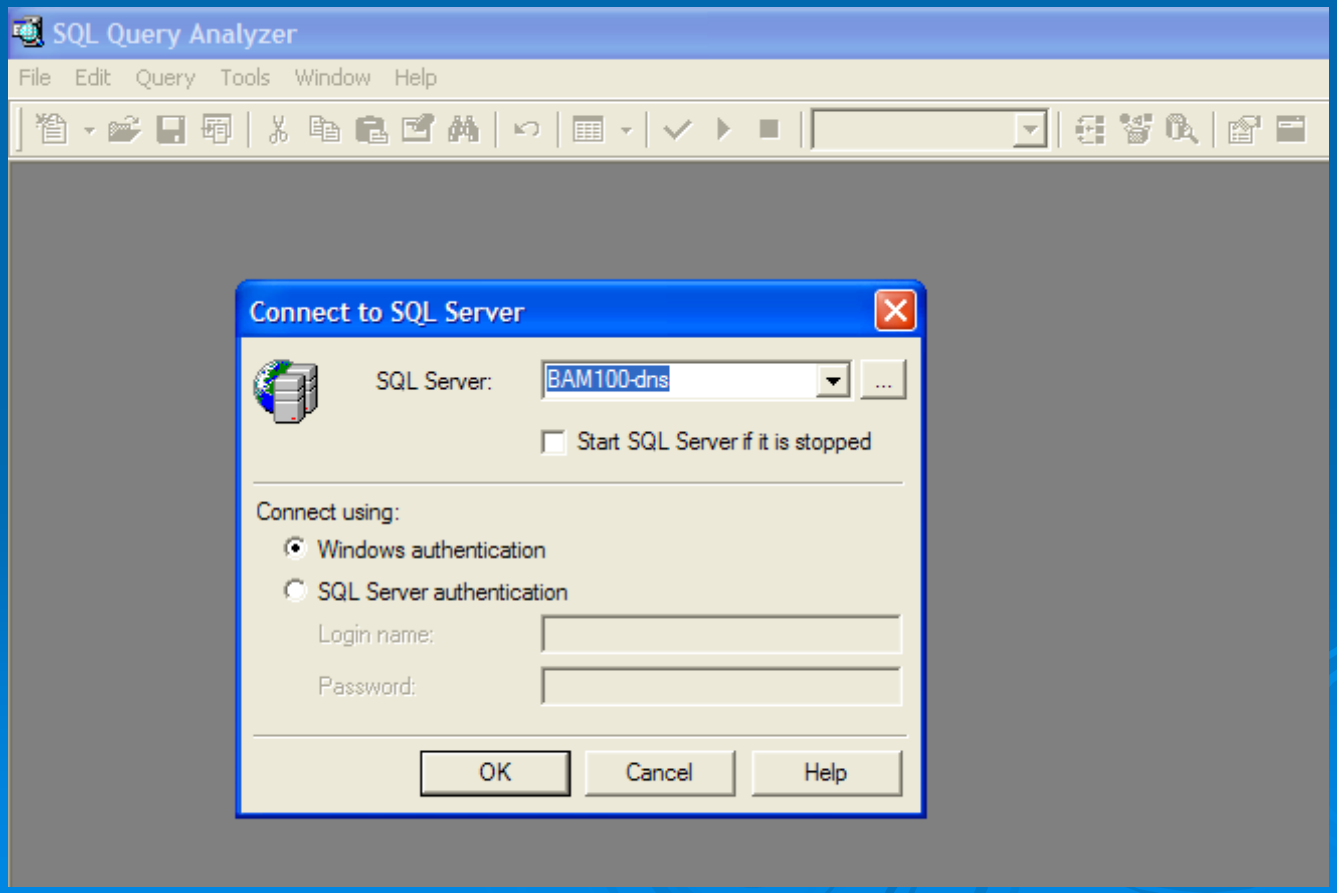
Laptop

SQL in action

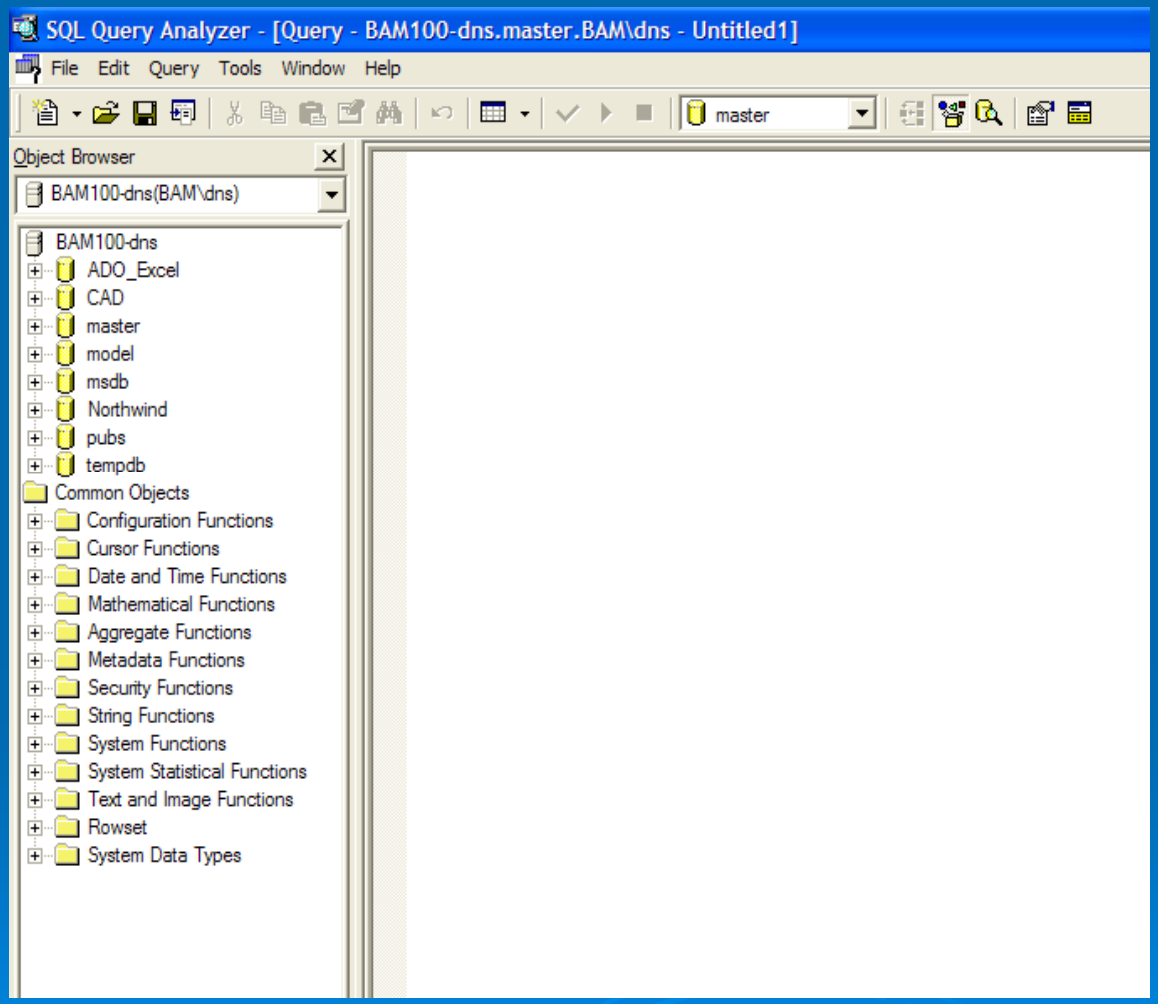


MS SQL 2000 Query Analyzer

SQL 2000 comes with a tool, called the Query Analyzer, which allows the developer to work with SQL databases and develop queries, tables and all manner of objects associated with those SQL databases. When running QA, you are prompted for the server you wish to connect to and how you will authenticate with that server, be it via your security account within Windows or via SQL Server authentication itself.



MS SQL 2000 Query Analyzer



A typical MDI windows application, QA offers an Object Browser window, multiple script writing windows and an output pane. It offers script syntax checking facilities, drag-and-drop from the object browser to the scripting window ,object scripting and amending and can display estimated execution plans – although the latter may well have to be the subject of another talk.

It also offers quick access to the very detailed Help system.

SELECT * FROM dbtable

SELECT ‘All columns (fields)’ FROM database table called ‘dbtable’

(and by default all records)

where ‘dbtable’ is the name of an existing database table/view.

Naming of objects in SQL follows the dot notation and is of the form:

server_name.database_name.object_owner.object_name

Typically for many of us, especially in the early days of developing SQL expressions, we will be working with components that are attached to the relevant server and database where the data we are requiring is located. Equally we will have ownership of the relevant object and then we can just refer to the object_name on its own.

So

SELECT * FROM myserver.pubs.dbo.employee becomes

SELECT * FROM employee

Remember though that this facility, to use the dot naming convention, allows you to work with multiple objects located in different databases on different servers!

Examples 01,02

SQL SELECT Clause

The excellent SQL Help system defines the SELECT clause as:

```
SELECT [ ALL | DISTINCT ]
    [ TOP n [ PERCENT ] [ WITH TIES ] ]
    < select_list >
< select_list > ::=
   {    *
        | { table_name | view_name | table_alias }.*
        |    { column_name | expression | IDENTITYCOL | ROWGUIDCOL }
           [ [ AS ] column_alias ]
        | column_alias = expression
    }    [ ,...n ]
```

This, however, may well seem complex so let's look at a few examples to show how easy it is!

SELECT fields FROM (vertical partitioning)

SELECT fieldname1, fieldname2 FROM

SELECT fieldname1, fieldname2, * FROM

SELECT fieldname1 AS SomeBetterName, fieldname2 AS SomeOtherName, * FROM

SELECT [my field] AS MyAliasName FROM

SELECT fieldname1 AS [My Nice "Name"] FROM

SELECT 'DataSource_1' AS DataSource, fieldname1, FALSE AS 'Status' FROM

Examples 03, 04, 05

SQL Functions in SELECT Clause

SELECT CAST(fieldname1 AS VARCHAR(12)) AS MyAliasName

SELECT CONVERT(VARCHAR(10), mydate, 103) AS MyAliasName

SELECT DATEDIFF(d, orderdate, shipdate) AS OrderToShipDays

SELECT DATEDIFF(wk, orderdate, shipdate) AS OrderToShipWeeks

SELECT GETDATE() AS Today, fieldname1,

SELECT fields FROM table WHERE condition (horizontal partitioning)

Although there are many ways to return just a subset of the source data, the WHERE clause is used most frequently.

WHERE (condition)

WHERE True (tends to return ALL the records!)

WHERE False (tends to return None!)

WHERE (freight > 10.00)

WHERE ((shipcity = 'France') AND (carrier = 'UPS'))

WHERE ((manufacturer = 'Dell') OR (carrier LIKE '%Walsh Western%'))

WHERE employeeID IN (5, 7,8)

WHERE NOT (employeeID IN (5,7,8))

WHERE (employeeID NOT IN (5,7,8))

ORDER BY clause

Often the result set from a query will be handled by some other application but in many situations you may still want to order the data in some way and we use the ORDER BY clause for this.

The ORDER BY clause has to come after the WHERE or GROUP BY clauses but before any HAVING (covered later) and has the form

ORDER BY fieldname1, fieldname2

with the default being the data is sorted in ascending order. This can be specified, for clarity, with the ASC parameter and sorting in reverse order can be performed with the DESC parameter.

ORDER BY employeeID

ORDER BY employeeID ASC, TotalSalesValue DESC

ORDER BY 1, 3 DESC is an alternative way of specifying sort order with the numbers referring to column positions in the result set. So this example would sort by column 1 data (ascending) and column 3 data descending.

Further Record/row Filtering

There are other ways of filtering record/rows from the source data into the result set. The default option in the SELECT clause is in fact the ALL parameter.

SELECT ALL * FROM dbtable

However, the TOP clause can be successfully used when just wanting a snapshot of the type of data held in a large table, or when (more frequently) you want the top n values from a data source when ordered in a particular way. So TOP n (n records) and TOP n PERCENT (n percent of the data source records) are both forms available to developers. The additional WITH TIES parameter allows the query to place additional records/rows in the result set if the n^{th}, $n^{th}+1$, etc records have matching values.

SELECT TOP 10 customerid, ordervalue ORDER BY ordervalue

SELECT TOP 20 PERCENT WITH TIES employeeid, employee_sales
ORDER BY employee_sales DESC

Further Record/row Filtering contd.

Another way of record/row filtering is by using the DISTINCT clause.

SELECT DISTINCT * FROM

Naturally on a keyed table the above DISTINCT clause will have no effect, but when working with non-keyed tables containing duplicate records OR when vertically partitioning the table even on a keyed table, the DISTINCT clause will retrieve only unique records/rows.

SELECT DISTINCT CustomerID, EmployeeID, ShipVia, ShipName FROM

Manipulating Result Set Column Data

Manipulation of the output columns can be done using the CASE function, two versions of which exist

SIMPLE CASE

- **CASE** ***input_expression***
  WHEN ***when_expression*** **THEN** ***result_expression***
  [...*n*]
  [
  ELSE ***else_result_expression***
  ]

  END

SEARCHED CASE

CASE
WHEN ***Boolean_expression*** **THEN** ***result_expression***
[...*n*]
[
ELSE ***else_result_expression***
]
END

Example 09B

Using Aggregate Functions

Using server based aggregate functions makes a lot of sense when relevant, as less data gets pulled from server to client and less client application design and processing is required.

Some of the aggregate functions are:

SUM, COUNT, AVG (these can all use the ALL or DISTINCT keyword)

MAX and MIN

SELECT SUM (freight) AS TotalFreight FROM orders

Naturally the WHERE operator can be used to limit what row values are summed such as

SELECT SUM (freight) AS TotalFreight FROM orders WHERE employeeID = 5

SELECT COUNT(*) AS NumberOfRecords FROM orders WHERE employeeID = 5

However, what you cannot do without additional keywords is an expression of the form

SELECT employeeID, SUM (freight) AS TotalFreight FROM orders

Using Aggregate Functions with GROUP BY

To solve queries where you want to aggregate some column/field value for a different column/field value then you must use the GROUP BY clause

So the incorrect SELECT employeeID, SUM (freight) AS TotalFreight FROM orders becomes

```
SELECT employeeID, SUM (freight) AS Total_Freight FROM orders
        GROUP BY employeeID
SELECT employeeID, MAX(OrderDate) AS Last_Employee_Sale_Date FROM orders
        GROUP BY employeeID
```

Every non aggregated field in the SELECT statement must be included in the GROUP BY clause. On occasions this is a nuisance and as long as the relevant field you want to include in the SELECT statement is directly 1-1 linked with a field in the SELECT statement then you can use expressions of the form

```
SELECT CustomerID, ShipName, MAX(ShipAddress) AS ShipAddressAsString,
SUM(freight) AS TotalFreightCosts FROM orders GROUP BY CustomerID,
ShipName
```

GROUP BY, ORDER BY & WHERE

Naturally we can use the ORDER BY clause with Aggregate functions and the GROUP BY clause to further control the result set.

```
SELECT    SUM(freight) AS TotalFreightCosts, CustomerID FROM orders
GROUP BY CustomerID
ORDER BY 1 DESC
```

Notice the ORDER BY clause uses column data ‘after’ the SELECT command has executed and therefore this can become

```
SELECT    SUM(freight) AS TotalFreightCosts, CustomerID FROM orders
GROUP BY CustomerID
ORDER BY TotalFreightCosts DESC
```

We can add the WHERE clause to this to further enhance the query as in

```
SELECT    SUM(freight) AS TotalFreightCosts, CustomerID FROM orders
WHERE employeeID = 5
GROUP BY CustomerID
ORDER BY TotalFreightCosts DESC
```

Note the order of these clauses – they cannot be entered in any other order.

Aggregate functions & HAVING

When working with Aggregate functions, you often want to limit data appearing in the result set based upon values of the relevant aggregate function output. In this case, the WHERE clause cannot be in this context, and the HAVING clause is used instead.

So the following will not work

```
SELECT  CustomerID, SUM(freight) AS TotalFreightCosts FROM orders
WHERE   SUM(freight) > 100
GROUP BY CustomerID
```

However, rewritten with the HAVING clause, the query now does what you want.

```
SELECT  CustomerID, SUM(freight) AS TotalFreightCosts FROM orders
GROUP BY CustomerID
HAVING  SUM(freight) > 100
```

Note the order of the clauses has now been changed with the HAVING function coming after the GROUP BY but before the ORDER BY. We can still use the WHERE clause to filter records prior to the aggregate function being executed as in

```
SELECT  CustomerID, SUM(freight) AS TotalFreightCosts FROM orders
WHERE   employeeID IN (5, 6, 8) GROUP BY CustomerID
HAVING  SUM(freight) > 100
ORDER BY 2
```

Finding Duplicate Records

One useful trick enabled through aggregate functions is that of being able to find duplicate records easily. To do this, we use the COUNT, GROUP BY and HAVING functions as follows:

```
SELECT  field1, field2, fieldn… , COUNT(*) AS Number_Of   FROM table
GROUP  BY field1, field2, fieldn…
HAVING COUNT(*) > 1
```

So a couple of real examples would be:

```
SELECT ProductID, UnitPrice, COUNT(*) AS OrderNumbers FROM [order details] od
GROUP BY ProductID, UnitPrice
HAVING COUNT(*) > 1
```

Or an example of when we only want ‘single’ entries such as what suppliers only supply one product?

```
SELECT SupplierID FROM products
GROUP BY supplierid
HAVING COUNT(*) = 1
```

Reversing Normal Form – Using JOINS

Oh to work with one table! However, as we all know, 'most' databases (and all the ones we have written!) are 'well designed' and in their respective normal forms. As such, we often require column/field data in our result set that is not contained from within a single table. When this happens, then we need to use the JOIN command.

The JOIN command allows the developer to link two or more tables/views and to retrieve data from them. The JOIN command can be set to link two or more relevant fields. The fields do NOT have to be of the same name, just the same data type; they don't even have to be key fields. A simple example of a JOIN would be:

```
SELECT * FROM Orders
JOIN [Order Details] ON orders.orderID = [Order Details].orderID
```

This query, an INNER JOIN, takes all the records from the Orders table and for each one of them fetches each row in the Order Details table with the same orderID value. This query also outputs every field in both tables.

With long table names like this, and when you want to specify which particular field you want in the output set, it may be helpful to use table aliases such as in:

```
SELECT * FROM Orders o
JOIN [Order Details] od ON o.orderID = od.orderID
```

Joins continued

We can, of course, join more than two tables together such as in:

```
SELECT  o.orderID, o.CustomerID,
CONVERT(VARCHAR(10), o.OrderDate, 103) AS OrderDateAsString,
        od.Quantity, od.UnitPrice, p.ProductName FROM Orders o
JOIN [Order Details] od ON o.orderID = od.orderID
JOIN Products p ON od.ProductID = p.ProductID
```

Note there is no NEED to prefix the column name with the table name or alias as long as that column name is UNIQUE amongst all the tables column names. So as OrderDate, Quantity and ProductName are unique, we can amend this to:

```
SELECT  o.orderID, o.CustomerID,
CONVERT(VARCHAR(10), OrderDate, 103) AS OrderDateAsString, Quantity,
od.UnitPrice, ProductName FROM Orders o
JOIN [Order Details] od ON o.orderID = od.orderID
JOIN Products p ON od.ProductID = p.ProductID
```

However, both Order Details and Products have a column called UnitPrice and hence we have to specify which column (from which table) we require in the result set (or as part of some other function or aggregation function).

Joins continued

It is unimportant what order the joins are listed in, so long as the join type is correct and that you do not refer to a table in a join predicate prior to specifying it. So both these queries produce the same results:

```
FROM     Orders o
JOIN [Order Details] od ON o.orderID = od.orderID
RIGHT JOIN Products p ON od.ProductID = p.ProductID
```

```
FROM     [Order Details] od
LEFT JOIN Products p ON od.ProductID = p.ProductID
JOIN Orders o ON o.orderID = od.orderID
```

This query will not work however as the as the reference in the second JOIN predicate to a table p has not yet been defined.

```
FROM     Orders o
JOIN [Order Details] od ON o.orderID = od.orderID
JOIN Suppliers s ON p.supplierID = s.supplierID
JOIN Customers c ON c.CustomerID = o.CustomerID
JOIN Products p ON od.ProductID = p.ProductID
```

Join Types

As you will be aware, depending upon the way in which your data tables are structured, you may need to JOIN tables together in different ways:

INNER JOIN	In an inner join only those records that have a matching value in the relevant tables are returned.
OUTER JOIN	There are 3 types of OUTER JOIN
LEFT OUTER JOIN	With this join all records from the ‘left’ hand table are returned with those matching records from the ‘right’ hand table.
RIGHT OUTER JOIN	With this join all records from the ‘right’ hand table are returned with those matching records from the ‘left’ hand table.
FULL OUTER JOIN	With this join all records from the left hand table are returned complete with all records from the right hand table.
CROSS JOIN	This join, without any WHERE clause, returns the cartesian product of the two tables, meaning for each row in the left hand table all records are returned from the right hand table. Two tables with 5 and 10 records respectively will produce a CROSS JOIN with 50 rows.

Joins - Where, Group By & Aggr' Functions

Naturally we can combine our Join expressions with use of the WHERE clause, Aggregate functions, Group BY and HAVING to provide finer control over what populates the result set. So we can use a 3 component WHERE clause:

```
SELECT o.orderID, o.CustomerID, OrderDate, od.Quantity,
od.UnitPrice, p.ProductName,c.CompanyName AS
CustomerCompanyName, s.CompanyName AS
SupplierCompanyName
FROM   Orders o
JOIN [Order Details] od ON o.orderID = od.orderID
JOIN Products p ON od.ProductID = p.ProductID
JOIN Customers c ON c.CustomerID = o.CustomerID
JOIN Suppliers s ON p.supplierID = s.supplierID
WHERE
s.Country LIKE '%Germany%'
AND p.Discontinued = 0
AND od.Discount <> 0.0
```

Joins – Using Table/Statement Aliases

A JOIN statement, like almost many other statements in SQL, can not only refer to a table, a view, stored procedure or table returning function, but to any other SQL statement. This enables us to join one SQL statement to another such as in:

```
SELECT  e.firstname + ' ' + e.lastname AS EmployeeName, SUM(od.Quantity * od.UnitPrice) AS OrderValue
FROM Employees e JOIN Orders o ON o.EmployeeID = e.EmployeeID
JOIN [Order Details] od ON od.orderID = o.orderID JOIN products p ON od.ProductID = p.ProductID
JOIN            --/now join to the next query using a table alias for it called 'q'
(
SELECT s.Country, p.ProductName, p.ProductID, SUM(od.Quantity * od.UnitPrice) AS OrderValue
FROM Orders o JOIN [Order Details] od ON o.orderID = od.orderID
JOIN Products p ON od.ProductID = p.ProductID
JOIN Customers c ON c.CustomerID = o.CustomerID
JOIN Suppliers s ON p.supplierID = s.supplierID
WHERE   p.Discontinued = 1
GROUP BY s.Country, p.ProductName, p.ProductID
) q             ON q.ProductName = p.ProductName

GROUP BY e.firstname + ' ' + e.lastname
ORDER BY OrderValue DESC
```

Joins – Using A Different Syntax

We can write join statements in another way but whether this is as easy to 'read' later is debatable.

```
SELECT  o.orderID, o.CustomerID,
        CONVERT(VARCHAR(10), o.OrderDate, 103) AS OrderDateAsString,
        od.Quantity, od.UnitPrice,
        p.ProductName,
        c.CompanyName AS CustomerCompanyName,
        s.CompanyName AS SupplierCompanyName
FROM    Orders o, [Order Details] od, Products p, Customers c, Suppliers s
WHERE   o.orderID = od.orderID
        AND od.ProductID = p.ProductID
        AND c.CustomerID = o.CustomerID
        AND p.supplierID = s.supplierID
        AND s.Country LIKE '%Germany%'
        AND p.Discontinued = 0
        AND od.Discount <> 0.0
```

Using this syntax we can use operators such as *=, =*, != to represent LEFT OUTER JOIN, RIGHT OUTER JOIN and NOT EQUAL

The UNION Operator

The UNION operator allows us to combine the results of two or more SQL expressions into a single result set. The only ‘rules’ for the UNION to work is that both expressions have the same number of fields in each and each respective one is of the same data type (i.e. you cannot UNION an Integer with a Date).

```
SELECT  'Pubs Employee‘ AS Company,  emp_id AS employee_ID,
lname AS last_name, fname AS first_name,  pj.job_desc AS Title
FROM pubs..employee pe JOIN pubs..jobs pj ON pe.job_id = pj.job_id

UNION

SELECT  'Northwinds Employee', CAST(EmployeeID AS VARCHAR(10)) AS emp_id,
LastName AS last_name,  firstname,  title
FROM northwind..employees
```

Note that the result set column names are determined by the first half of the UNION, so in the above example the first two columns in the result set will be called ‘Company’ and ‘employee_ID’; using the column alias in the second half of the UNION is in fact irrelevant.

The UNION Operator & Table Aliases

Naturally you can use further SELECT statements, using a table alias, to filter the result of a UNION as in:

```
SELECT * FROM
(
SELECT  'Pubs Employee' AS Company, emp_id AS employee_ID,
lname AS last_name,  fname AS first_name,  pj.job_desc  AS Title
FROM pubs..employee pe JOIN pubs..jobs pj ON pe.job_id = pj.job_id

UNION

SELECT 'Northwinds Employee', CAST(EmployeeID AS VARCHAR(10)) AS emp_id,
LastName AS last_name,  firstname,  title
FROM northwind..employees
) tblAllCompanies  --/ ← this is the table alias

WHERE tblAllCompanies.Title  LIKE  '%Sales%'
ORDER  BY Title
```

Views

Views are used widely by developers for many reasons. They are really just a set of SQL statements that produce a table like output. Their advantage over tables are:

1. Zero-user-access security can be applied to the actual data tables, with users working solely with VIEWS. This prevents users knowing anything about and being able to edit your data tables, helping protect your development investment. The definition of what the VIEW is will in these cases be encrypted.
2. VIEWS can be updateable (otherwise the above to some extent wouldn't work) although there is a limit to how many underlying tables can updated with a single UPDATE statement.
3. VIEWS allows users access to 'real-world' type data rather than the data typically found in well normalised data tables; this enables them to more easily work with report generators etc.
4. VIEWS can be almost always be used in place of a table.

A disadvantage with a VIEW is that you cannot pass it a parameter. To do this you should use UDFS that return tables OR Stored Procedures.

User Defined Functions

UDFs allow you to create functions that can be used in any SQL statement, VIEW or Stored Procedure and can simplify coding. Routines that you would regularly require in a variety of circumstances can be turned into a UDF returning to the calling code either a scalar value or a table. The advantages of a UDF are:

1. Write code once, use many times!
2. Hide underlying table data by encrypting the UDF definition.

Stored Procedures

Stored Procedures offer the same for execution code that UDFs offer for ‘informational’ code. Naturally SPs can be used, and often are used, for returning data to the user but are frequently used for data manipulation. The advantages of SPs are similar to that for UDFs.

SQL An Introduction – The End

I hope you have enjoyed this short introduction to what I am spending more and more time with – and learning all the time.

I hope that it was interesting and that, if you didn't know too much about SQL before, you have learnt something, and that if you did, then either you had a little revision or a chance for a snooze before tea and biscuits!

Thank you for listening

David