Table of Contents

[Setup 4](#_Toc520102127)

[Handy keyboard shortcuts: 4](#_Toc520102128)

[Coding Conventions 4](#_Toc520102129)

[Joins 5](#_Toc520102130)

[Inner Join 5](#_Toc520102131)

[Left Outer Join 6](#_Toc520102132)

[Exclusive Left Join 6](#_Toc520102133)

[Right Join 6](#_Toc520102134)

[Exclusive Right Join 6](#_Toc520102135)

[Full Outer Join 6](#_Toc520102136)

[Exclusive Full Outer Join 6](#_Toc520102137)

[Semi Join 7](#_Toc520102138)

[Anti Semi-Join 7](#_Toc520102139)

[Cross Join 7](#_Toc520102140)

[Self Join 8](#_Toc520102141)

[Union 8](#_Toc520102142)

[Intersect 9](#_Toc520102143)

[Except 9](#_Toc520102144)

[Join Order 10](#_Toc520102145)

[Order By 11](#_Toc520102146)

[Group By 11](#_Toc520102147)

[Having 11](#_Toc520102148)

[All these types of joins – which one is best 12](#_Toc520102149)

[Query plans 13](#_Toc520102150)

[Logical Operators 13](#_Toc520102151)

[Physical Operators 13](#_Toc520102152)

[Nested Loop Join 13](#_Toc520102153)

[Merge Join 14](#_Toc520102154)

[Hash Join 14](#_Toc520102155)

[Further Reading 15](#_Toc520102156)

[Index Seeks and Scans 15](#_Toc520102157)

[Complex Selects 16](#_Toc520102158)

[Multiple tables 16](#_Toc520102159)

[Self join 16](#_Toc520102160)

[Sub-queries 16](#_Toc520102161)

[Correlated Subqueries 16](#_Toc520102162)

[Table Aliases 17](#_Toc520102163)

[CTE 17](#_Toc520102164)

[CROSS APPLY / Lateral Join 17](#_Toc520102165)

[Ranking Functions 18](#_Toc520102166)

[Tally Table 18](#_Toc520102167)

[Pivot 18](#_Toc520102168)

[Large Tables 19](#_Toc520102169)

[Primary Keys 21](#_Toc520102170)

[Foreign Keys 21](#_Toc520102171)

[Indexes 22](#_Toc520102172)

[ASC / DESC indexes 22](#_Toc520102173)

[INCLUDE indexes 22](#_Toc520102174)

[Pros/Cons of indexing 23](#_Toc520102175)

[SARGable Queries 23](#_Toc520102176)

[Stored Procedures 24](#_Toc520102177)

[Temporary Tables 24](#_Toc520102178)

[Transactions 24](#_Toc520102179)

[GO 24](#_Toc520102180)

[Functions 25](#_Toc520102181)

[Views 25](#_Toc520102182)

[Can’t you do this in a Stored Procedure? 25](#_Toc520102183)

[RBAR: Row-By-Agonising-Row 26](#_Toc520102184)

[XML / JSON generation 28](#_Toc520102185)

[Select data in JSON format 28](#_Toc520102186)

[Querying JSON data 28](#_Toc520102187)

[Common JSON issues 29](#_Toc520102188)

[Using JSON Output 29](#_Toc520102189)

[Bulk Insert 30](#_Toc520102190)

[Logging: 30](#_Toc520102191)

[bcp utility 30](#_Toc520102192)

[Notes 31](#_Toc520102193)

[Dates 31](#_Toc520102194)

[Collations 31](#_Toc520102195)

[Schemas 31](#_Toc520102196)

[Entity Framework and other ORM wrappers 32](#_Toc520102197)

[Further Reading 33](#_Toc520102198)

[Statistics 33](#_Toc520102199)

[Versions of Cardinality Estimator 33](#_Toc520102200)

[Possible impact of data skew 33](#_Toc520102201)

[Column Stores 33](#_Toc520102202)

[Python 33](#_Toc520102203)

[Useful sites / blogs 33](#_Toc520102204)

# Setup

Install SQL Server – free version is fine

Run instnwnd.sql to create example database

## Handy keyboard shortcuts:

ALT & Mouse to select by column

ALT + Arrow keys to move line

Crtl-E to execute

SHIFT CTRL U to uppercase

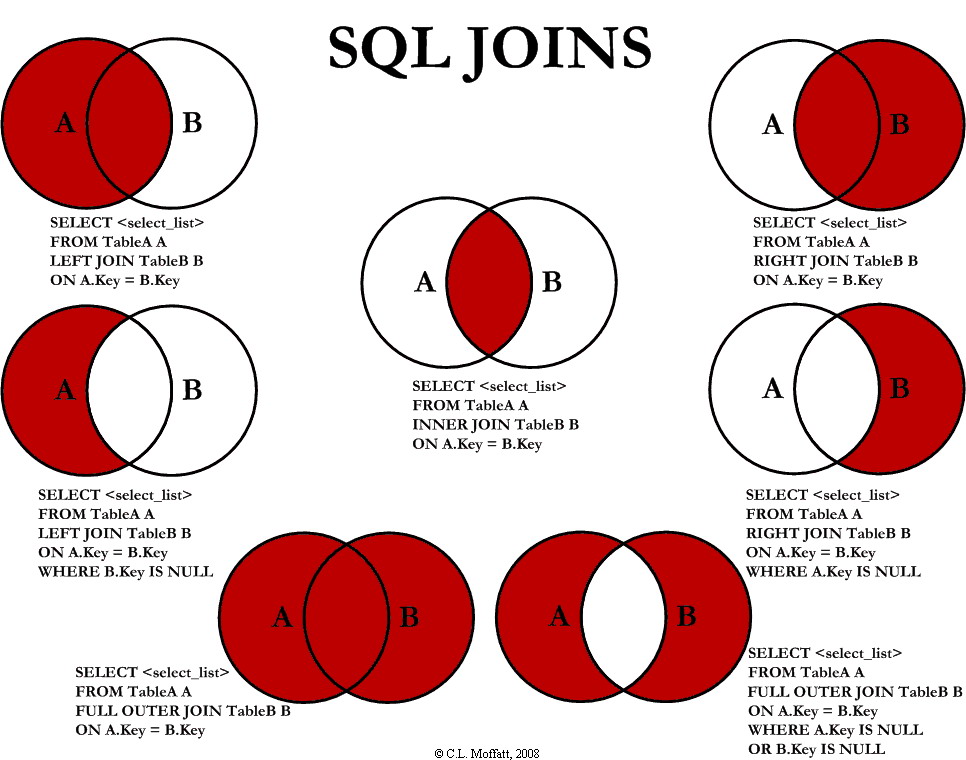
## Coding Conventions

By convention SQL commands should be uppercase

Schemas, table names, columns names etc should match the case they were defined as.

A Civica coding standards document is being prepared

# Joins



Try the following select queries and note the difference in the data returned, and the total number of rows returned for each.

## Inner Join

SELECT \* FROM dbo.Orders

SELECT \*   
FROM dbo.Orders   
INNER JOIN dbo.[Order Details] ON dbo.Orders.OrderID=dbo.[Order Details].OrderID;

Expected count: 2155

SELECT count(\*)

FROM dbo.Orders

INNER JOIN dbo.Suppliers ON dbo.Orders.ShipRegion = dbo.Suppliers.Region

Expected count: 54

## Left Outer Join

SELECT \* from dbo.Orders

SELECT \* from dbo.Suppliers

SELECT \*

FROM dbo.Orders

LEFT OUTER JOIN dbo.Suppliers ON dbo.Orders.ShipRegion = dbo.Suppliers.Region

Expected count: 843

## Exclusive Left Join

SELECT \*

FROM dbo.Orders

LEFT OUTER JOIN dbo.Suppliers ON dbo.Orders.ShipRegion = dbo.Suppliers.Region

WHERE dbo.Suppliers.Region IS NULL

Expected count: 789

## Right Join

SELECT \*

FROM dbo.Orders

RIGHT OUTER JOIN dbo.Suppliers ON dbo.Orders.ShipRegion = dbo.Suppliers.Region

Expected count: 80

## Exclusive Right Join

SELECT \*

FROM dbo.Orders

RIGHT OUTER JOIN dbo.Suppliers ON dbo.Orders.ShipRegion = dbo.Suppliers.Region

WHERE dbo.Orders.ShipRegion IS NULL

Expected count: 26

Right and left outer joins are functionally equivalent. Neither provides any functionality that the other does not, so right and left outer joins may replace each other as long as the table order is switched.

Try the right/left joins above again, but switch the order of the tables. You should find that when doing the left join you get the results you previously had for the right join

## Full Outer Join

SELECT \*

FROM dbo.Orders

FULL OUTER JOIN dbo.Suppliers ON dbo.Orders.ShipRegion = dbo.Suppliers.Region

Expected count: 869

## Exclusive Full Outer Join

SELECT \*

FROM dbo.Orders

FULL OUTER JOIN dbo.Suppliers ON dbo.Orders.ShipRegion = dbo.Suppliers.Region

WHERE dbo.Orders.ShipRegion IS NULL OR dbo.Suppliers.Region IS NULL

Expected count: 815

## Semi Join

This is a common join type, but is rarely explicitly named as such. . You will see them in the query plan output.

In the joins above you can get multiple results as a single row may match many rows in the other table. But with a semi join only one row is returned, no matter how many matches there are.

Here is an example of a semi join

SELECT \*

FROM dbo.Orders

WHERE EXISTS (

SELECT 1

FROM dbo.Suppliers

WHERE dbo.Suppliers.Region = dbo.Orders.ShipRegion

)

You may be more familiar with semi joins in the form of an IN clause:

SELECT \*

FROM dbo.Orders

WHERE dbo.Orders.ShipRegion IN (

SELECT dbo.Suppliers.Region

FROM dbo.Suppliers

)

If you take a look at the execution plan you should see that the Hash Match is marked as a Right Semi Join

## Anti Semi-Join

As for the semi join, this is a commonly used, but is rarely explicitly named as such. It is the negative join i.e. NOT EXISTS, NOT IN

SELECT \*

FROM dbo.Orders

WHERE dbo.Orders.ShipRegion NOT IN (

SELECT dbo.Suppliers.Region

FROM dbo.Suppliers

WHERE dbo.Suppliers.SupplierID < 10

)

If you take a look at the execution plan you should see that the Nested Loop is marked as a Left Anti Semi Join

This is a good example of comparing the multiple ways to find rows in Table A that do not match Table B: <https://sqlperformance.com/2012/12/t-sql-queries/left-anti-semi-join>

## Cross Join

This selects the Cartesian product of the two tables i.e. each and every row in Table A is joined to each and every row in Table B giving every possible combination. In general this should be avoided as it produces massive result sets. However, it can be very useful in reporting and creating test data though.

SELECT \*

FROM dbo.Orders

CROSS JOIN dbo.Suppliers

Expected count: 24,070

If using the return value of a table valued function, then **cross apply** should be used instead of **cross join**. This allows parameters to be sent to the function.

<http://weblogs.sqlteam.com/jeffs/archive/2005/09/12/7755.aspx>

## Self Join

SELECT e2.FirstName + ' ' + e2.LastName AS [Managed By], e1.\*

FROM dbo.Employees e1

INNER JOIN dbo.Employees e2 ON e1.ReportsTo=e2.EmployeeID

Expected count: 8

**Bonus Question:** How would you re-write the query above to include the Vice President of Sales, who does not report to anyone?

## Union

Not a join in its own right, but allows you to concatenate the results of multiple queries together. The results sets must have the same number columns and their types must match. They can be named differently, in which case the first select defines the column names

**UNION** returns a distinct set of rows after combining all the selected results

SELECT dbo.Employees.EmployeeID, dbo.Employees.LastName, dbo.Employees.FirstName, dbo.Employees.Title, dbo.Employees.TitleOfCourtesy as test

FROM dbo.Employees

WHERE dbo.Employees.TitleOfCourtesy='Ms.'

UNION

SELECT dbo.Employees.EmployeeID, dbo.Employees.LastName, dbo.Employees.FirstName, dbo.Employees.Title, dbo.Employees.TitleOfCourtesy

FROM dbo.Employees

WHERE dbo.Employees.Title='Sales Representative'

ORDER BY dbo.Employees.EmployeeID

Expected count: 7

**UNION ALL** returns every row from all the selects, including exact duplicates

SELECT dbo.Employees.EmployeeID, dbo.Employees.LastName, dbo.Employees.FirstName, dbo.Employees.Title, dbo.Employees.TitleOfCourtesy as test

FROM dbo.Employees

WHERE dbo.Employees.TitleOfCourtesy='Ms.'

UNION ALL

SELECT dbo.Employees.EmployeeID, dbo.Employees.LastName, dbo.Employees.FirstName, dbo.Employees.Title, dbo.Employees.TitleOfCourtesy

FROM dbo.Employees

WHERE dbo.Employees.Title='Sales Representative'

ORDER BY dbo.Employees.EmployeeID

Expected count: 9

## Intersect

Note – this query requires the large table created in section 19

SELECT

dbo.3.EmployeeID

, dbo.ManyEmployees.FirstName

, dbo.ManyEmployees.LastName

, dbo.ManyEmployees.TitleOfCourtesy

, dbo.ManyEmployees.RegionID

FROM ManyEmployees

WHERE FirstName= 'Thomas'

INTERSECT

SELECT

dbo.ManyEmployees.EmployeeID

, dbo.ManyEmployees.FirstName

, dbo.ManyEmployees.LastName

, dbo.ManyEmployees.TitleOfCourtesy

, dbo.ManyEmployees.RegionID

FROM ManyEmployees

WHERE TitleOfCourtesy='Ms'

INTERSECT

SELECT

dbo.ManyEmployees.EmployeeID

, dbo.ManyEmployees.FirstName

, dbo.ManyEmployees.LastName

, dbo.ManyEmployees.TitleOfCourtesy

, dbo.ManyEmployees.RegionID

FROM ManyEmployees

WHERE EmployeeID<3000

## Except

Run the above query again, replacing each intersect one-by-one with EXCEPT, then with both replaced. Can you explain the differences in the result sets?

Except and Intersect can be very useful for data analysis. For example:

DELETE FROM dbo.[Order Details] WHERE ProductID >=70

--discover which orders don't have any order details (i.e. are empty orders and may be invalid)

SELECT dbo.Orders.OrderID FROM dbo.Orders

EXCEPT

SELECT dbo.[Order Details].OrderID FROM dbo.[Order Details]

--discover which products aren't in any order - perhaps these should stop being stocked

SELECT ProductID FROM dbo.Products

INTERSECT

SELECT dbo.[Order Details].ProductID FROM dbo.[Order Details]

Aren’t these just the same as a join? Not quite: union, intersect and except are set operators, not joins. They work to define rowsets, not add columns as a join does.   
Intersect etc works on all columns, not just on the ones specified in the ON clause. This can be a limitation

## Join Order

You may find advice to carefully construct the join order of your tables, debates with AND in the ON versus in the WHERE clauses. Generally, these can be completely ignored. In SQL you describe the results you want, and the database engine figures out the best way to get it for you, using its compiled statistics about the database. If you interfere with this, it is very likely that you’ll actually make things worse, or discover you have optimised for one specific case that does not hold true under production use.

Run the following selects, with execution plan and statistics turned on. You should find that the database engine uses exactly the same plan for all of them.

SELECT \*

FROM dbo.customers

INNER JOIN dbo.orders ON dbo.Customers.CustomerID = dbo.Orders.CustomerID

INNER JOIN dbo.[Order Details] ON dbo.Orders.OrderID = dbo.[Order Details].OrderID

WHERE dbo.orders.EmployeeID=3

SELECT \*

FROM dbo.customers

INNER JOIN dbo.orders ON dbo.Customers.CustomerID = dbo.Orders.CustomerID AND dbo.orders.EmployeeID=3

INNER JOIN dbo.[Order Details] ON dbo.Orders.OrderID = dbo.[Order Details].OrderID

SELECT \*

FROM dbo.Orders

INNER JOIN dbo.Customers ON dbo.Orders.CustomerID = dbo.Customers.CustomerID AND dbo.Orders.EmployeeID=3

INNER JOIN dbo.[Order Details] ON dbo.Orders.OrderID = dbo.[Order Details].OrderID

You can force SQL Server to join in the order you wish, but be very sure that it is necessary to do so. If there is a problem, generally a refresh of the database statistics and/or indexes will fix it. Once the database engine has the updated & valid statistics it will generally make the correct decisions

There are of course, edge cases when it won’t, and if after investigating there are no other options you can add [HINTS](https://docs.microsoft.com/en-us/sql/t-sql/queries/hints-transact-sql?view=sql-server-2017) to your queries. But be very sure of what you are doing & the effects it will have.

Note that forcing the join order in a query will set the order of all the tables in the query – so think about all of your joins.

You can also force join order without hints, try to avoid queries like these as they prevent the database engine from using its knowledge

* <https://www.brentozar.com/archive/2015/05/forcing-join-order-without-hints/>
* <https://bertwagner.com/2017/11/21/does-the-join-order-of-my-tables-matter/>

### Order By

There is no guaranteed order to the results of a SQL query. It will most likely come back in the order of the primary index, but given many inserts/deletes over time, parallelism when scanning indexes and whatever the optimiser chooses to do you cannot depend on any inherent order – there is none.

What you can do is use the ORDER BY clause:

SELECT \*

FROM [Northwind].[dbo].[Products]

ORDER BY SupplierID

You can order by multiple columns:

SELECT \*

FROM [Northwind].[dbo].[Products]

ORDER BY SupplierID, CategoryID

<https://blogs.msdn.microsoft.com/conor_cunningham_msft/2008/08/27/no-seatbelt-expecting-order-without-order-by/>

### Group By

Used with aggregate functions to return the result set in one or more groups

SELECT OrderID, ProductID, SUM(UnitPrice \* Quantity) as Total

FROM [Order Details]

GROUP BY OrderID, ProductID

**Note** that any ‘ordinary’ non-aggregate columns in the SELECT list must be in the GROUP BY clause. They cannot be included otherwise, as if we tried what value would be returned for multiple rows (the ‘aggregate’) that have different values for the non-aggregated column.

For example, the following select will error:

SELECT orderid, unitprice, SUM(unitprice \* quantity) as Total

FROM [order details]

GROUP BY OrderID

As we are not grouping by unitprice, how should the database engine know which of the multiple input unitprices for each order to put into the single, aggregated output row. It can’t, and so you will get an error.

### Having

You cannot use aggregate functions in a WHERE clause.

If you were asked to restrict the above query to orders that SUM to more than 100, your initial instinct might be to try:

SELECT OrderID, ProductID, SUM(UnitPrice \* Quantity) as Total

FROM [Order Details]

WHERE SUM(UnitPrice \* Quantity)>100

GROUP BY OrderID, ProductID

This will error – try it.

The order of the statements is very important. WHERE must follow FROM, and is used to filter the rows that are read into memory for the rest of the statement to work on. Only after this has been done is the GROUP BY applied. So it cannot happen in the WHERE – the aggregation has not happened yet, therefore the SUM cannot be used to filter the rows.

This is why the HAVING statement exists. It allows filtering of the aggregated rows:

SELECT OrderID, ProductID, SUM(UnitPrice \* Quantity) as Total

FROM [Order Details]

GROUP BY OrderID, ProductID

HAVING SUM(UnitPrice \* Quantity)>100

## All these types of joins – which one is best

Unfortunately there is no easy answer to this. ‘Best’ very much depends on the environment, data, indexes, hardware, workload and other factors.

# Query plans

Learn how to turn on and interpret query plans:

* <https://www.mssqltips.com/sqlservertip/1856/sql-server-query-execution-plans-in-sql-server-management-studio/>
* <https://www.mssqltips.com/sqlservertip/1873/how-to-read-sql-server-graphical-query-execution-plans/>

Usually the Estimated and the Actual Plans have similar graphical representation, but they can differ in cases where the statistics are outdated or the query involves parallelism, etc...  Additionally you cannot create Estimated Plans for queries that create objects and work with them (i.e. a query using a temp table).

* <https://www.mssqltips.com/sqlservertutorial/2252/estimated-vs-actual-query-plan/>
* <https://www.mssqltips.com/sqlservertutorial/285/query-execution-plans/>

### Logical Operators

These are the joins specified in the sql statements

* Inner Join
* Outer Join
* Cross Join
* Cross Apply (new in SQL 2005)
* Semi-Join
* Anti Semi-Join

See the JOIN section for details.

### Physical Operators

These are what the database engine uses to evaluate the logical operators

* Nested Loop Join
* Merge Join
* Hash join

## Nested Loop Join

This is the most costly join. A nested loops join compares each row from one table (known as the outer table) to each row from the other table (known as the inner table) looking for rows that satisfy the join predicate.

for each row R1 in the outer table  
 for each row R2 in the inner table  
 if R1 joins with R2  
 return (R1, R2)

It’s the nesting of the for loops in this algorithm that gives nested loops join its name.

The total number of rows compared and, thus, the cost of this algorithm is proportional to the size of the outer table multiplied by the size of the inner table. Since this cost grows quickly as the size of the input tables grow, in practice we try to minimize the cost by reducing the number of inner rows that we must consider for each outer row

Read through the following blog post for examples and best practices: <https://blogs.msdn.microsoft.com/craigfr/2006/07/26/nested-loops-join/>

## Merge Join

The merge join works by simultaneously reading and comparing the two sorted inputs one row at a time. Therefore there must either be a sorted index on the merge columns or a pre-sort.

get first row R1 from input 1  
get first row R2 from input 2  
while not at the end of either input  
 begin  
 if R1 joins with R2  
 begin  
 return (R1, R2)  
 get next row R2 from input 2  
 end  
 else if R1 < R2  
 get next row R1 from input 1  
 else  
 get next row R2 from input 2  
 end

Unlike the nested loops join where the total cost may be proportional to the product of the number of rows in the input tables, with a merge join each table is read at most once and the total cost is proportional to the sum of the number of rows in the inputs. Thus, merge join is often a better choice for larger inputs.

Read through the following blog post for examples and best practices:  
<https://blogs.msdn.microsoft.com/craigfr/2006/08/03/merge-join/>

## Hash Join

Hash joins are used to perform the largest joins.  Hash joins parallelize and scale better than any other join. They also require the most memory as the database engine must build hash tables before performing the join.

get first row R1 from input 1  
get first row R2 from input 2  
while not at the end of either input  
 begin  
 if R1 joins with R2  
 begin  
 return (R1, R2)  
 get next row R2 from input 2  
 end  
 else if R1 < R2  
 get next row R1 from input 1  
 else  
 get next row R2 from input 2  
 end

Read through the following blog post for examples and best practices  
<https://blogs.msdn.microsoft.com/craigfr/2006/08/10/hash-join/>

### Further Reading

Work through the examples given here to understand how indexes affect the type of join chosen by the database engine.  
<https://blogs.msdn.microsoft.com/craigfr/2006/08/16/summary-of-join-properties/>

The rest of this blog is also very useful.

## Index Seeks and Scans

A seek uses the index to pinpoint the records that are needed to satisfy the query

An index or table scan is when SQL Server has to scan the data or index pages to find the appropriate records. This means that every leaf-level node was searched to match the criteria.

In general, scans should be avoided and indexes created to allow for a seek operation to occur.

Run the following queries together:

SELECT \* FROM [Order Details] WHERE [Order Details].Quantity = 16

SELECT \* FROM [Order Details] WHERE [Order Details].OrderID = 11077

You should see that although both queries are simply selecting on a value, as Quantity is not covered by an index a scan is used, and takes comparatively much longer than the OrderID query using a seek.

<https://www.mssqltips.com/sqlservertutorial/277/index-scans-and-table-scans/>

# Complex Selects

**TIP** – when comparing query execution times always ignore the first run. Quite often this can be distorted by setting up the initial connection and will not have the benefit of cached plans etc that subsequent queries can take advantage of.

## Multiple tables

You know how to do this right – multiple join statements

## Self join

A self join is used when a table references itself. This is useful when you want to get data from multiple table rows into a single output row.

**Exercise**:

Write a select statement which returns both the employee name and the name of their supervisor in the same output row.

Answer:

SELECT e1.LastName

, e1.FirstName

, e1.Title

, e2.LastName

, e2.FirstName

, e2.Title

FROM Employees e1

INNER JOIN Employees e2 ON e1.ReportsTo=e2.EmployeeID

## Sub-queries

A sub-query is the results of one query fed into another

For pedants: see [terminology definitions](https://www.xaprb.com/blog/2005/09/26/sql-subqueries-and-derived-tables/)

There is a worked example for the Northwind database. Follow it through & see the output you get. Can you explain how it gets the results it does? <https://www.mssqltips.com/sqlservertip/1042/using-derived-tables-to-simplify-the-sql-server-query-process/>

### Correlated Subqueries

Many queries can be evaluated by executing the subquery once and substituting the resulting value or values into the WHERE clause of the outer query. In queries that include a correlated subquery (also known as a repeating subquery or inline-view), the subquery depends on the outer query for its values. This means that the subquery is executed repeatedly, once for each row that might be selected by the outer query.

For example:

SELECT \*

FROM Employees e

WHERE e.Country IN (

SELECT s.Country

FROM [Suppliers] s

WHERE s.City= e.City

)

In that above query the inner select requires data from the outer select, and so must be processed for each outer row

This is generally to be avoided, as it is not good for performance. See the RBAR section for more details.

## Table Aliases

You may have noticed some queries ‘rename’ a table in the select statement. This is known as table aliasing. It is necessary when:

* the same table is referenced more than once
* there is a derived table (i.e. a sub query returning table output) also known as anonymous view (see subquery example from the mssqltips link)

It can also be very useful to alias tables in selects to reduce the length of lines, which can make the query easier to read and understand.

In the query from the link, aliasing the Orders table as o, and Order Details as OG is not strictly necessary, however the alias for the sub-query – OG – is.

## CTE

A [CTE (Common Table Expression)](https://docs.microsoft.com/en-us/sql/t-sql/queries/with-common-table-expression-transact-sql?view=sql-server-2017)  is a temporary named result set, derived from a simple query and defined within the execution scope of a SELECT, INSERT, UPDATE, or DELETE statement.

At the link, there are many examples using the Adventure Works database– can you re-write D, E and F to use the Northwind Employees table.

## CROSS APPLY / Lateral Join

The main purpose is to enable table functions with parameters to be executed once per row and then joined to the results. When the sets of a JOIN are not self-sufficient, a ‘normal’ join cannot be written.

Create the sample tables from the following link and run the queries: <http://explainextended.com/2009/07/16/inner-join-vs-cross-apply/>

Further reading on Apply:

* <http://www.sqlservercentral.com/articles/APPLY/69953/>
* <http://www.sqlservercentral.com/articles/APPLY/69954/>

When would you actually use APPLY – as for anything, there’s StackOverflow: <https://stackoverflow.com/questions/9275132/real-life-example-when-to-use-outer-cross-apply-in-sql>

## Ranking Functions

Read through this blog post and worked examples: <http://codingsight.com/methods-to-rank-rows-in-sql-server-rownumber-rank-denserank-and-ntile/>

Further work

Rewrite the examples to use the Northwind database: <https://docs.microsoft.com/en-us/sql/t-sql/queries/select-over-clause-transact-sql?view=sql-server-2017>

### Tally Table

If you are working on an old project, you may come across a Number or Tally table, which is a table with one integer column with values from 0-N, with N being a very large number. This was used for ranking before the ranking functions were introduced. It can still be very useful for string manipulation.

For details, have a read of:

<http://www.sqlservercentral.com/blogs/dwainsql/2014/03/27/tally-tables-in-t-sql/>

<http://www.sqlservercentral.com/articles/T-SQL/62867/>

### Pivot

Pivoting a table allows you to make rows columns, and columns to rows. This can be useful for reporting

Example:

SELECT SupplierID, [Alice Mutton],[Aniseed Syrup],[Boston Crab Meat],[Chai],[Chang],[Genen Shouyu]

FROM (

SELECT SupplierID, ProductName, UnitsInStock FROM Products

) pv

PIVOT (SUM(UnitsInStock) FOR ProductName IN ([Alice Mutton],[Aniseed Syrup],[Boston Crab Meat],[Chai],[Chang],[Genen Shouyu])) as pvt

ORDER BY SupplierID

The general problem with Pivot is that you need to know the values to pivot on in advance, and the list can get pretty long. You can [dynamically](https://stackoverflow.com/questions/10404348/sql-server-dynamic-pivot-query) pivot, but this adds complexity

#### Unpivot

See <https://blog.sqlauthority.com/2008/06/07/sql-server-pivot-and-unpivot-table-examples/>

# Large Tables

We’re going to create a new table with a large amount of data. You may need to adjust the following SQL to match the capabilities of you laptop

USE [Northwind]

GO

CREATE TABLE [dbo].[ManyEmployees](

[EmployeeID] [int] NULL,

[LastName] [nvarchar](20) NULL,

[FirstName] [nvarchar](10) NULL,

[Title] [nvarchar](30) NULL,

[TitleOfCourtesy] [nvarchar](25) NULL,

[BirthDate] [datetime] NULL,

[HireDate] [datetime] NULL,

[Address] [nvarchar](60) NULL,

[City] [nvarchar](15) NULL,

[Region] [nvarchar](15) NULL,

[PostalCode] [nvarchar](10) NULL,

[Country] [nvarchar](15) NULL,

[HomePhone] [nvarchar](24) NULL,

[Extension] [nvarchar](4) NULL,

[Photo] [image] NULL,

[Notes] [ntext] NULL,

[ReportsTo] [int] NULL,

[PhotoPath] [nvarchar](255) NULL,

[RegionID] [int] NULL

) ON [PRIMARY] TEXTIMAGE\_ON [PRIMARY]

GO

Increase Max degree of parallelism to 100 with the following query  
Or you can set it through Sever Properties > Advanced > Max degree of parallelism  
See <https://www.brentozar.com/archive/2013/09/five-sql-server-settings-to-change/>

EXEC sp\_configure 'show advanced options', 1;

GO

RECONFIGURE WITH OVERRIDE;

GO

EXEC sp\_configure 'max degree of parallelism', 100;

GO

RECONFIGURE WITH OVERRIDE;

GO

Now run the following scripts to create large amounts of data. Replace the csv filenames with the full path to the file

CREATE TABLE #firstnames (

firstname NVARCHAR(100)

)

GO

CREATE TABLE #surnames (

surname NVARCHAR(100)

)

GO

CREATE TABLE #titles (

title NVARCHAR(100)

)

GO

BULK INSERT #firstnames

FROM 'firstnames.csv'

WITH

(

FIELDTERMINATOR =',',

ROWTERMINATOR ='\n',

FIRSTROW = 1

);

GO

BULK INSERT #surnames

FROM 'surnames.csv'

WITH

(

FIELDTERMINATOR =',',

ROWTERMINATOR ='\n',

FIRSTROW = 1

);

GO

BULK INSERT #titles

FROM 'titles.csv'

WITH

(

FIELDTERMINATOR =',',

ROWTERMINATOR ='\n',

FIRSTROW = 1

);

GO

--you might find it interesting to watch Task Manager while this executes

-- and watch what happens to the log file

INSERT INTO dbo.ManyEmployees (EmployeeID, LastName, FirstName, TitleOfCourtesy, RegionID)

SELECT ROW\_NUMBER() over (ORDER BY (SELECT NULL)), #surnames.surname, #firstnames.firstname, #titles.title, r.RegionID

FROM #firstnames

CROSS JOIN #surnames

CROSS JOIN #titles

CROSS JOIN (SELECT RegionID FROM Region) r

If your machine is struggling with this, delete a few rows out of the temp tables

DROP TABLE #firstnames

DROP TABLE #surnames

DROP TABLE #titles

SELECT COUNT(\*) FROM dbo.ManyEmployees

Decrease Max degree of parallelism to 1

Your log file is probably quite big now, so it would be a good idea to truncate it before continuing.

Run the following queries with execution plan & client statistics switched on:

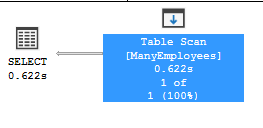
SELECT \* FROM dbo.ManyEmployees WHERE EmployeeID=429194

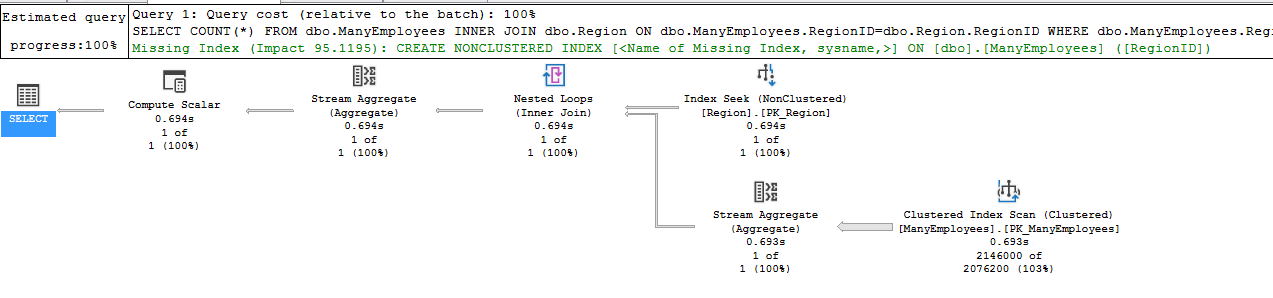
SELECT COUNT(\*) FROM dbo.ManyEmployees WHERE dbo.ManyEmployees.LastName IN('THOMSON', 'WALKER', 'KERR', 'WALLACE');

SELECT COUNT(\*) FROM dbo.ManyEmployees

INNER JOIN dbo.Region ON dbo.ManyEmployees.RegionID=dbo.Region.RegionID

WHERE dbo.ManyEmployees.RegionID=1





SELECT \* FROM dbo.ManyEmployees WHERE LastName='GRIFFITHS' AND firstname= 'FINN' AND TitleOfCourtesy='Ms'

Requires full table scan, note missing indexes on execution plan

# Primary Keys

Increase transaction timeout to 300 SMSS > Tools > Options > Designers > Transaction time-out after: 300

Add primary key on EmployeeID, (may need to up parallelism again)

ALTER TABLE [dbo].[ManyEmployees] ADD CONSTRAINT [PK\_ManyEmployees] PRIMARY KEY CLUSTERED

(

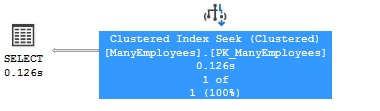
[EmployeeID] ASC

)WITH (PAD\_INDEX = OFF, STATISTICS\_NORECOMPUTE = OFF, SORT\_IN\_TEMPDB = OFF, IGNORE\_DUP\_KEY = OFF, ONLINE = OFF, ALLOW\_ROW\_LOCKS = ON, ALLOW\_PAGE\_LOCKS = ON) ON [PRIMARY]

GO

Repeat previous queries, note execution plan changes & time taken

SELECT \* FROM dbo.ManyEmployees WHERE EmployeeID=429194



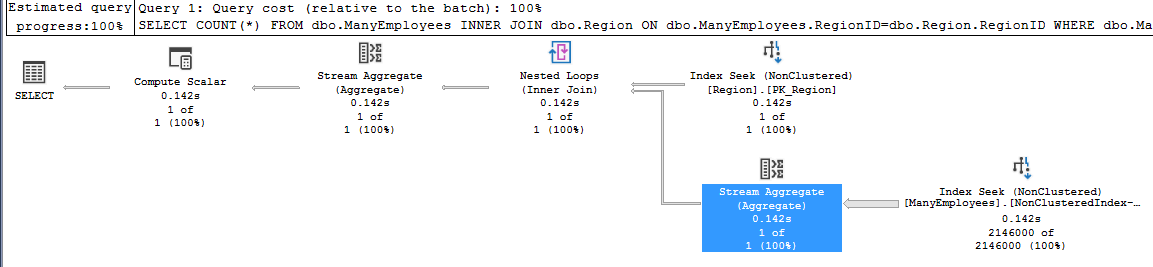
# Foreign Keys

Add foreign key to ManyEmployees.RegionID. Create index on ManyEmployees.RegionID

SELECT COUNT(\*) FROM dbo.ManyEmployees

INNER JOIN dbo.Region ON dbo.ManyEmployees.RegionID=dbo.Region.RegionID

WHERE dbo.ManyEmployees.RegionID=1



# Indexes

Add indexes to column used in queries

Open database file directory, watch what happens to data file

An index scan is when SQL Server reads all the data in the index pages. The cost of an index scan is very expensive for the SQL Server Engine.

An index seek is when SQL Server reads only matching data in the index pages. This method is more efficient for query performance because it will reduce IO and time consumption.

One seek is of course much faster than scanning a whole table, but when you have to seek a lot of rows, a scan could make more sense.

### ASC / DESC indexes

DESC useful when last in is needed more often – e.g. when sorting by date

Can also be useful when covering multiple rows – have one field sorted asc, other desc

Apply the Indexes and Selects as described here using the AdventureWorks database to the Northwind database used in our examples.

<https://www.mssqltips.com/sqlservertip/1337/building-sql-server-indexes-in-ascending-vs-descending-order/>

### INCLUDE indexes

In order to perform queries most efficiently, all columns used in a query should be included in an index. This is known as a **covering** index.

If the full table has a lot of data, it can be expensive to lookup the record values in the main table after the matches have been found in the index. In this case, it can be useful to **include** extra fields required by the select clause, which are not used in the where / joins. Including extra columns this way means that all the data required by the select is available in the leaf node of the index, and so it is not necessary to perform the lookup back to the table. This can save a great deal of time.

<https://docs.microsoft.com/en-us/sql/relational-databases/indexes/create-indexes-with-included-columns?view=sql-server-2017>

## Pros/Cons of indexing

Make selects faster, inserts / deletes slower. Generally this is considered an acceptable trade-off, especially as many refinements have been made to improve index storage. See <https://sqlity.net/en/2445/b-plus-tree/>

Also requires lots of disk space, so only index column sets that will be queried on

### SARGable Queries

Make your queries SARGable

As per wikipedia SARGable is defined as “In relational databases, a condition (or predicate) in a query is said to be sargable if the DBMS engine can take advantage of an index to speed up the execution of the query. The term is derived from a contraction of Search ARGument ABLE”

Advantage of sargable queries include:

* consuming less system resources
* speeding up query performance
* using indexes more effectively

The main things that prevent a query from being sargeable is using functions (custom or inbuilt) in the where clause. If you find yourself doing that, see if you can re-write your query to remove them.

Do not blindly use functions / views in queries without knowing what they do, and how they do it. As well as preventing the use of indexes this is a common way for RBAR to creep in

Work through the following for examples:

<https://www.sqlshack.com/how-to-use-sargable-expressions-in-t-sql-queries-performance-advantages-and-examples/>

# Stored Procedures

Stored Procedures are the workhorses of database programming. They are batches of SQL statements that can be executed on demand.

## Temporary Tables

Local temporary tables (start with #) are limited to your session; other sessions, even from the same user/connection string, can't see them. A local temporary table that is created in a stored procedure is dropped when the procedure ends; other stored procedures, or the calling process, can't see them. Other local temporary tables are dropped when the session ends.  
  
Global temporary tables (start with ##) are shared between sessions. They are dropped when the session that created them ends and no other session is referring to them

## Transactions

A transaction is a single unit of work. It can contain many operations, but if one operation fails none are applied. A transaction can either be ***implicit*** e.g. running ad-hoc SQL statements in the editor or in a Stored Procedures that doesn’t explicitly specify transactions. ***Explicit*** transactions are started by the **BEGIN TRANSACTION** command and completed by either **COMMIT TRANSACTION** or **ROLLBACK**. If an error occurs the whole transaction is rolled back.

Transactions should be used to ensure related operations either all succeed or are not applied. They are an extremely useful way of ensuring the integrity of the data, and there are no partially successful commits.

Run the following code – you should see that although the name is changed inside the transaction, as it is rolled back instead of committed, once outside of the transaction the name reverts to the original value

BEGIN TRANSACTION

SELECT LastName, FirstName, Title FROM Employees WHERE EmployeeID=1;

UPDATE Employees SET FirstName = 'FRED' WHERE EmployeeID=1;

SELECT LastName, FirstName, Title FROM Employees WHERE EmployeeID=1;

ROLLBACK;

SELECT LastName, FirstName, Title FROM Employees WHERE EmployeeID=1;

### GO

You may be familiar with GO statements in SQL scripts. They are commonly used as transaction markers, but this is not what they are for and do not work as such. GO is not a SQL command. It is a command recognised by the SSMS and command line utilities to send a batch of statements to SQL Server. This may or may not match the transactions (it does not unless you explicitly make it do so).

It can also be useful to repeat SQL statements e.g. for testing:

SELECT GETDATE();

GO 10

<https://docs.microsoft.com/en-us/sql/t-sql/language-elements/sql-server-utilities-statements-go?view=sql-server-2017>

# Functions

SQL Server user-defined functions are routines that accept parameters, perform an action, such as a complex calculation, and return the result of that action as a value. The return value can either be a single scalar value or a result set.

They are useful for abstracting complex parts of queries, just as functions would do in programming languages. The database engine can also cache the execution plans leading to faster performance.

They cannot have any side effects such as modifying table data.

<https://docs.microsoft.com/en-us/sql/relational-databases/user-defined-functions/user-defined-functions?view=sql-server-2017>

* Write a function that return an employee’s full name: TitleOfCourtesy, FirstName, LastName
* Then write a View or Stored Procedure that takes an employeeID and returns the full name and address

For extra fun, combine this with the hierarchal query used previously to return both the full name of the employee and their manager in the same row.

# Views

A view is a virtual table. It presents information from multiple source tables in a single result set

* Create a View that combines the Customers and Employees table, joined on City  
  Select \* FROM <your\_view> WHERE City = London

### Can’t you do this in a Stored Procedure?

Yes, but a View has a different purpose to a Stored Procedure. A view is for looking at the data in a specific way and acts as a filter on the underlying tables. It can also be indexed (a materialised view), which can vastly improve performance.

Stored procedures are simply stored SQL statements. They can have parameters, you can choose which parts execute or not with control logic, they can create and destroy objects in the database, and can return any number of result sets. There are for operating on data.

For more information: <https://docs.microsoft.com/en-us/sql/relational-databases/views/views?view=sql-server-2017>

# RBAR: Row-By-Agonising-Row

You don't tell SQL *how to do it*, you tell SQL *what you want*

SQL is not procedural, as you will be used to from other programming languages. It is set based, and you describe the output you need & the database engine provides it to you.

<http://weblogs.sqlteam.com/jeffs/archive/2007/04/30/thinking-set-based-or-not.aspx>

If you are used to procedural languages it can be very easy to drop into row-by-row processing, but there is usually a set-based approach that will vastly improve performance.

If you are having trouble thinking in sets, it can be useful to draw [Venn Diagrams](https://en.wikipedia.org/wiki/Venn_diagram) of the data and the subsets you want. These then can be mapped to the JOINS you require.

The classic case of RBAR is using cursors to select some data, and then process it in some way

For example:

DECLARE @Dep\_Id INT

DECLARE @Emp\_Id INT

DECLARE EmpCursor CURSOR FOR

SELECT dep\_id, id FROM employee

OPEN EmpCursor

FETCH NEXT FROM EmpCursor INTO @Dep\_Id, @Emp\_Id

DECLARE @DepName NVARCHAR(50)

WHILE(@@FETCH\_STATUS = 0)

BEGIN

SELECT @DepName = dep\_name FROM department where id = @Dep\_Id

IF(@DepName = 'Sales')

BEGIN

UPDATE employee SET salary = (salary \* 1.1) WHERE id = @Emp\_Id

END

ELSE IF(@DepName = 'HR')

BEGIN

UPDATE employee SET salary = (salary \* 1.2) WHERE id = @Emp\_Id

END

IF(@DepName = 'IT')

BEGIN

UPDATE employee SET salary = (salary \* 1.5) WHERE id = @Emp\_Id

END

FETCH NEXT FROM EmpCursor INTO @Dep\_Id, @Emp\_Id

END

CLOSE EmpCursor

DEALLOCATE EmpCursor

This can be replaced by:

UPDATE employee

SET salary = CASE

WHEN dep\_name = 'Sales' THEN (salary \* 1.1)

WHEN dep\_name = 'HR' THEN (salary \* 1.2)

WHEN dep\_name = 'IT' THEN (salary \* 1.5)

END

FROM employee

JOIN department

ON department.id = employee.dep\_id

WHERE (dep\_name = 'Sales' OR dep\_name = 'HR' OR dep\_name = 'IT')

Other cases where RBAR can creep in:

* function calls in select / where
* correlated subqueries

If you are using Views or Functions in your query always check what they are doing and how.

# XML / JSON generation

## Select data in JSON format

If you are using SSMS you may need to change to Results to Text or Results to File as the returned string can be quite large

This simplest way to select as JSON is to use FOR JSON AUTO at the end of a query

SELECT \*

FROM Orders

FOR JSON AUTO

If need be you can also do this for XML. In this case, the query would end with FOR XML AUTO

SELECT \*

FROM Orders

FOR XML AUTO

The standard output does not have a root node meaning that the output may not be readable by parses. You can specify a root node by adding the Root option:

SELECT \*

FROM Orders

FOR JSON AUTO, Root('Orders')

More information on JSON output options is available here: <https://docs.microsoft.com/en-us/sql/relational-databases/json/format-query-results-as-json-with-for-json-sql-server?view=sql-server-2017>

FOR JSON will also nest joined tables as required:

SELECT \*

FROM Orders

INNER JOIN [Order Details] ON [Order Details].OrderID=[Orders].OrderID

FOR JSON AUTO

If you need more control over the nesting PATH gives you much more control: <https://docs.microsoft.com/en-us/sql/relational-databases/json/format-nested-json-output-with-path-mode-sql-server?view=sql-server-2017>

**Exercise:**

* Write a query that outputs the Ship columns in the Orders table as a node named 'shippingDetails'
* Add the INCLUDE\_NULL\_VALUE option and compare the output for the ShipRegion column
* Extend your query to include the [Order Details] table JOINED on OrderID.

### Querying JSON data

<https://docs.microsoft.com/en-us/sql/relational-databases/json/json-data-sql-server?view=sql-server-2017>

Create a new column in the Orders table called JSONData of type nvarchar(MAX)

Use the queries to you wrote above to insert the values of the Order row into JSONData. Have the Ship columns as a node named ‘ShippingDetails’ and a root node of ‘Order’

Before you continue select out the value of the JSON inserted in the first row. Look at it in a JSON parser e.g <http://json.parser.online.fr/> Is it what you expected?

You should only have the values for the columns in the same row, not the entire table. If you do have the entire table, what changes do you need to make to your UPDATE query to only insert the data for the row?

Read though the following link and then try the exercises: <https://docs.microsoft.com/en-us/sql/relational-databases/json/validate-query-and-change-json-data-with-built-in-functions-sql-server?view=sql-server-2017>

* Select the value of ShipCountry from JSONData for ID 10253
* Select the ShippingDetails node for ID 10253
* Update ShippingDetails node for ID 10253 by setting the value of ShipCity to Sao Paulo

### Common JSON issues

<https://docs.microsoft.com/en-us/sql/relational-databases/json/solve-common-issues-with-json-in-sql-server?view=sql-server-2017>

## Using JSON Output

<https://docs.microsoft.com/en-us/sql/relational-databases/json/use-for-json-output-in-sql-server-and-in-client-apps-sql-server?view=sql-server-2017>

Answers

UPDATE Orders set JSONData = (

SELECT

[OrderID]

,[CustomerID]

,[EmployeeID]

,[OrderDate]

,[RequiredDate]

,[Freight]

,[ShippedDate] AS 'ShippingDetails.ShippedDate'

,[ShipVia] AS 'ShippingDetails.ShipVia'

,[ShipName] AS 'ShippingDetails.ShipName2'

,[ShipAddress] AS 'ShippingDetails.ShipAddress'

,[ShipCity] AS 'ShippingDetails.ShipCity'

,[ShipRegion] AS 'ShippingDetails.ShipRegion'

,[ShipPostalCode] AS 'ShippingDetails.ShipPostalCode'

,[ShipCountry] AS 'ShippingDetails.ShipCountry'

FROM [dbo].Orders o

Where o.OrderID=Orders.OrderID

FOR JSON PATH, Root('Order'), INCLUDE\_NULL\_VALUES

)

SELECT JSON\_QUERY(JSONData, '$.Order') [Order] FROM Orders where orderid=10253

SELECT JSON\_VALUE(JSONData, '$.Order[0].ShippingDetails.ShipCity') FROM Orders where orderid=10253

UPDATE Orders SET JSONData = (JSON\_MODIFY(JSONData, '$.Order[0].ShippingDetails.ShipCity', 'Sao Paulo')) where orderid=10253

# Bulk Insert

<https://docs.microsoft.com/en-us/sql/t-sql/statements/bulk-insert-transact-sql?view=sql-server-2017>

## Logging:

A Bulk Insert operation can massively increase the log file size. To reduce this, and the execution time, as much as possible it is a good idea to drop indexes on the table, have simple logging enabled. If possible, batch the inserts so that the log can be cleared between transactions.

<https://docs.microsoft.com/en-us/sql/relational-databases/import-export/prerequisites-for-minimal-logging-in-bulk-import?view=sql-server-2017>

## bcp utility

The Bulk Copy Utility can also be very useful to insert data. Setting the batch size option can reduce the impact on the logs

<https://docs.microsoft.com/en-us/sql/tools/bcp-utility?view=sql-server-2017>

# Notes

String searches are case insensitive by default. Unless you’re querying JSON/XML, then it is.

COALESCE is a useful command to use in SELECT statements to handle nulls

If you need to generate some SELECT / INSERT / UPDATE scripts for re-use later, a quick way of producing them from existing data is to use CONCAT in a SELECT statement. Each row of the output is then a statement that can be executed

## Dates

Most installs use defaults, so will be American English

Depending on settings, SMSS may show dates in UK style, but selects will output in American style.

Always specify date format string YYYY-MM-DD is preferred.

## Collations

Normally you don’t need to think about collations, and string operations will just work. However if for some reason the server, database and/or column collations do not match (e.g. if a database with a specified collation has been restored to a server with a different collation) you can get unexpected results. If you have weird bugs in your queries this is worth checking. You can set the collation to use in a query by using COLLATE

For more info: <https://www.red-gate.com/simple-talk/sql/sql-development/questions-sql-server-collations-shy-ask/#post-70501-_Toc479001452>

## Schemas

Schemas can be very useful to control access to data, logically group related objects, and prevent naming conflicts in large systems.

However this increases the complexity of administering the database, and ORMs can get in a muddle with them.

If you’re not absolutely sure the schemas will benefit you – use the default dbo

# Entity Framework and other ORM wrappers

Writing and maintaining Stored Procedures for CRUD operations for all tables is very time-consuming, so this is where ORM wrappers come in extremely useful.

However for complicated selects and updates then can generate extremely inefficient SQL – as found in the PROACT project (mentioned by Simon Kelly on Capability Call). A general approach is to start with an ORM for the basics, and write Stored Procedures / Views as required.

Be careful with security: ORMS generally must have access to tables. On some projects, this can rule out ORMs. Users can be given execute rights only to specific Stored Procedures and Views, thus protecting the tables themselves.

Stops SQL Server from building query plans.

Can also have lots of problems if done as code-first in Visual Studio. This generally leads to ppor database design, and lack of needed indexes. It also often creates problems when there are overlapping changes in code repository branches. It is much more reliable to have a set of curated scripts. E.g.an initial Create script and then Update scripts run in a specific order to get to the latest version of the database.

# Further Reading

## Statistics

<https://docs.microsoft.com/en-us/sql/relational-databases/statistics/statistics?view=sql-server-2017>

## Versions of Cardinality Estimator

<https://docs.microsoft.com/en-us/sql/relational-databases/performance/cardinality-estimation-sql-server?view=sql-server-2017>

## Possible impact of data skew

<https://www.sqlshack.com/understanding-skewed-data-in-sql-server/>

## Column Stores

<https://docs.microsoft.com/en-us/sql/relational-databases/indexes/columnstore-indexes-overview?view=sql-server-2017>

Useful instead of indexes for reporting

## Python

<https://blogs.msdn.microsoft.com/cdndevs/2015/03/11/python-and-data-sql-server-as-a-data-source-for-python-applications/>

# Useful sites / blogs

<https://www.brentozar.com/blog/>

<http://sqlblog.com/blogs/dejan_sarka/default.aspx>

<https://sqlbits.com/> - over for this year, but excellent for finding knowledgeable people