**SQL Server Join Examples**

**Problem**I am new to SQL Server and want to learn about the JOIN options.  What are all of the JOIN options in SQL Server?  What is the significance of each of the options?  I am a little confused on the differences and syntax, can you provide some examples and explanations?  Are JOINs only for SELECT statements?

**Solution**Joining tables to obtain the needed data for a query, script or stored stored procedure is a key concept as you learn about SQL Server development.  In a nutshell, joins are typically performed in the FROM clause of a table or view for the SELECT, INSERT...SELECT, SELECT...INTO, UPDATE and DELETE statements.  In previous versions of SQL Server, join logic could also have been included in the WHERE clause with = (INNER JOIN), \*= (LEFT OUTER JOIN), =\* (RIGHT OUTER JOIN), etc. syntax, but the support has been reduced and the best practice in SQL Server is to use the syntax outlined in the examples below.

Before we jump into code, let's provide some baseline information on the joins options in SQL Server:

* INNER JOIN - Match rows between the two tables specified in the INNER JOIN statement based on one or more columns having matching data.  Preferably the join is based on referential integrity enforcing the relationship between the tables to ensure data integrity.
  + Just to add a little commentary to the basic definitions above, in general the INNER JOIN option is considered to be the most common join needed in applications and/or queries.  Although that is the case in some environments, it is really dependent on the database design, referential integrity and data needed for the application.  As such, please take the time to understand the data being requested then select the proper join option.
  + Although most join logic is based on matching values between the two columns specified, it is possible to also include logic using greater than, less than, not equals, etc.
* LEFT OUTER JOIN - Based on the two tables specified in the join clause, all data is returned from the left table.  On the right table, the matching data is returned in addition to NULL values where a record exists in the left table, but not in the right table.
  + Another item to keep in mind is that the LEFT and RIGHT OUTER JOIN logic is opposite of one another.  So you can change either the order of the tables in the specific join statement or change the JOIN from left to right or vice versa and get the same results.
* RIGHT OUTER JOIN - Based on the two tables specified in the join clause, all data is returned from the right table.  On the left table, the matching data is returned in addition to NULL values where a record exists in the right table but not in the left table.
* Self -Join - In this circumstance, the same table is specified twice with two different aliases in order to match the data within the same table.
* CROSS JOIN - Based on the two tables specified in the join clause, a Cartesian product is created if a WHERE clause does filter the rows.  The size of the Cartesian product is based on multiplying the number of rows from the left table by the number of rows in the right table.  Please heed caution when using a CROSS JOIN.
* FULL JOIN - Based on the two tables specified in the join clause, all data is returned from both tables regardless of matching data.

Let's walk through examples from the AdventureWorks sample database that is available for SQL Server to provide examples of each type of join then provide some insight into the usage and sample result sets.

***INNER JOIN Example***

In this example we are joining between the Sales.SalesOrderDetail and Production.Product tables.  The tables are aliased with the following:  SOD for Sales.SalesOrderDetail and P for Production.Product.  The JOIN logic is based on matching records in the SOD.ProductID and P.ProductID columns.  The records are filtered by only returning records with the SOD.UnitPrice greater than 1000.  Finally, the result set is returned in order with the most expensive first based on the ORDER BY clause and only the highest 100 products based on the TOP clause.

|  |
| --- |
| USE AdventureWorks;  GO  SELECT TOP 100 P.ProductID,  P.Name,  P.ListPrice,  P.Size,  P.ModifiedDate,  SOD.UnitPrice,  SOD.UnitPriceDiscount,  SOD.OrderQty,  SOD.LineTotal  FROM Sales.SalesOrderDetail SOD  INNER JOIN Production.Product P  ON SOD.ProductID = P.ProductID  WHERE SOD.UnitPrice > 1000  ORDER BY SOD.UnitPrice DESC  GO |

***LEFT OUTER JOIN Example***

In this example we are combining two concepts to show that more than two tables can be JOINed in one SELECT statement and more than one JOIN type can be used in a single SELECT statement.  In the sample code below, we are retrieving the matching data between the Person.Contact and Sales.SalesPerson tables in conjunction with all of the data from the Sales.SalesPerson table and matching data in the Sales.SalesTerritory table.  For records that exist Sales.SalesPerson table and not in the Sales.SalesTerritory table, NULL values are returned for the columns in the Sales.SalesTerritory.  In addition, this code uses two columns to order the data i.e. ST.TerritoryID and C.LastName.

|  |
| --- |
| USE AdventureWorks;  GO  SELECT C.ContactID,  C.FirstName,  C.LastName,  SP.SalesPersonID,  SP.CommissionPct,  SP.SalesYTD,  SP.SalesLastYear,  SP.Bonus,  ST.TerritoryID,  ST.Name,  ST.[Group],  ST.SalesYTD  FROM Person.Contact C  INNER JOIN Sales.SalesPerson SP  ON C.ContactID = SP.SalesPersonID  LEFT OUTER JOIN Sales.SalesTerritory ST  ON ST.TerritoryID = SP.TerritoryID  ORDER BY ST.TerritoryID, C.LastName  GO |

***RIGHT OUTER JOIN Example***

In an effort to explain how the RIGHT OUTER JOIN and LEFT OUTER JOIN is logically a reciprocal on one another, the code below is re-written version of the LEFT OUTER JOIN above.  As you can see the JOIN order and tables are different, but the final result set matches the LEFT OUTER JOIN logic.   In the sample code below, we are retrieving the matching data between the Person.Contact and Sales.SalesPerson tables in conjunction with all of the data from the Sales.SalesPerson table and matching data in the Sales.SalesTerritory table.  For records that exist Sales.SalesPerson table and not in the Sales.SalesTerritory table, NULL values are returned for the columns in the Sales.SalesTerritory.

|  |
| --- |
| USE AdventureWorks;  GO  SELECT C.ContactID,  C.FirstName,  C.LastName,  SP.SalesPersonID,  SP.CommissionPct,  SP.SalesYTD,  SP.SalesLastYear,  SP.Bonus,  ST.TerritoryID,  ST.Name, ST.[Group],  ST.SalesYTD  FROM Sales.SalesTerritory ST  RIGHT OUTER JOIN Sales.SalesPerson SP  ON ST.TerritoryID = SP.TerritoryID  INNER JOIN Person.Contact C  ON C.ContactID = SP.SalesPersonID  ORDER BY ST.TerritoryID, C.LastName  GO |

***Self Join Example***

In this example, we are actually self joining to the HumanResources.Employee table.  We are doing this to obtain the information about the Employee and Manager relationship in the HumanResources.Employee table.  In conjunction with that JOIN logic we are also joining to the Person.Contact twice in order to capture the name and title data based on the original Employee and Manager relationships.  In addition, another new concept introduced in this query is aliasing each of the columns.  Although we could have done so in the previous examples, we made point of doing so in this query to differentiate between the Employee and Manager related data.

|  |
| --- |
| USE AdventureWorks;  GO  SELECT M.ManagerID AS 'ManagerID',  M1.ContactID AS 'ManagerContactID',  M1.FirstName AS 'ManagerFirstName',  M1.LastName AS 'ManagerLastName',  M.Title AS 'ManagerTitle',  E.EmployeeID AS 'EmployeeID',  E1.ContactID AS 'EmployeeContactID',  E1.FirstName AS 'EmployeeFirstName',  E1.LastName AS 'EmployeeLastName',  E.Title AS 'EmployeeTitle'  FROM HumanResources.Employee E  INNER JOIN HumanResources.Employee M  ON E.ManagerID = M.ManagerID  INNER JOIN Person.Contact E1  ON E1.ContactID = E.ContactID  INNER JOIN Person.Contact M1  ON M1.ContactID = M.ContactID  ORDER BY M1.LastName  GO |

***CROSS JOIN Example***

As indicated above, please heed caution when running or modifying this query in any SQL Server database environment.  The result set is intentionally limited by the TOP 100 clause and the WHERE clause to prevent a Cartesian product, which is the result of each of the rows from the left table multiplied by the number of rows in the right table.

|  |
| --- |
| USE AdventureWorks;  GO  SELECT TOP 100 P.ProductID,  P.Name,  P.ListPrice,  P.Size,  P.ModifiedDate,  SOD.UnitPrice,  SOD.UnitPriceDiscount,  SOD.OrderQty,  SOD.LineTotal  FROM Sales.SalesOrderDetail SOD  CROSS JOIN Production.Product P  WHERE SOD.UnitPrice > 3500  ORDER BY SOD.UnitPrice DESC  GO |

***FULL OUTER JOIN Example***

In our last example, we have modified the logic from the LEFT OUTER JOIN example above and converted the LEFT OUTER JOIN syntax to a FULL OUTER JOIN.  In this circumstance, the result set is the same as the LEFT OUTER JOIN where we are returning all of the data between both tables and data not available in the Sales.SalesTerritory is returned as NULL.

|  |
| --- |
| USE AdventureWorks;  GO  SELECT C.ContactID,  C.FirstName,  C.LastName,  SP.SalesPersonID,  SP.CommissionPct,  SP.SalesYTD,  SP.SalesLastYear,  SP.Bonus,  ST.TerritoryID,  ST.Name,  ST.[Group],  ST.SalesYTD  FROM Person.Contact C  INNER JOIN Sales.SalesPerson SP  ON C.ContactID = SP.SalesPersonID  FULL OUTER JOIN Sales.SalesTerritory ST  ON ST.TerritoryID = SP.TerritoryID  ORDER BY ST.TerritoryID, C.LastName  GO |

**Next Steps**

* As you begin to start coding in SQL Server be sure to have a firm understanding of the JOIN options available as well as the associated data that is retrieved.  Be sure to select the correct JOIN logic based on the data that needs to be retrieved.
* Once you have a firm grasp of the JOIN logic with SELECT statements, progress to using the logic with  INSERT...SELECT, SELECT...INTO, UPDATE and DELETE statements

# Joining data and differences of using UNION and UNION ALL in SQL Server

**Problem**Sometimes there is a need to combine data from multiple tables or views into one comprehensive dataset. This may be for like tables within the same database or maybe there is a need to combine like data across databases or even across servers.  I have read about the UNION and UNION ALL commands, but how do these work and how do they differ?

**Solution**In SQL Server you have the ability to combine multiple datasets into one comprehensive dataset by using the UNION or UNION ALL operators.  There is a big difference in how these work as well as the final result set that is returned, but basically these commands join multiple datasets that have similar structures into one combined dataset.

Here is a brief description:

* UNION - this command will allow you to join multiple datasets into one dataset and will remove any duplicates that exist.  Basically it is performing a DISTINCT operation across all columns in the result set.
* UNION ALL - this command again allows you to join multiple datasets into one dataset, but it does not remove any duplicate rows.  Because this does not remove duplicate rows this process is faster, but if you don't want duplicate records you will need to use the UNION operator instead.

Rules to union data:

* Each query must have the same number of columns
* Each column must have compatible data types
* Column names for the final result set are taken from the first query
* ORDER BY and COMPUTE clauses can only be issued for the overall result set and not within each individual result set
* GROUP BY and HAVING clauses can only be issued for each individual result set and not for the overall result set

|  |
| --- |
| **Tip**  If you don't have the exact same columns in all queries use a default value or a NULL value such as:  SELECT firstName, lastName, company FROM businessContacts UNION ALL SELECT firstName, lastName, NULL FROM nonBusinessContacts  or  SELECT firstName, lastName, createDate FROM businessContacts UNION ALL SELECT firstName, lastName, getdate() FROM nonBusinessContacts |

Examples:

Let's take a look at a few simple examples of how these commands work and how they differ.  As you will see the final resultsets will differ, but there is some interesting info on how SQL Server actually completes the process.

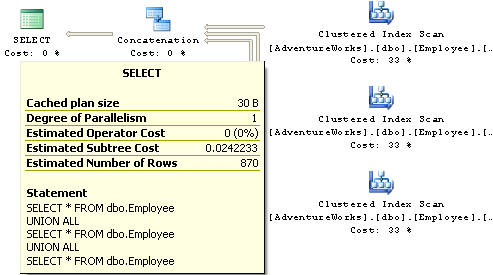
In this first example we are using the UNION ALL operator against the Employee table from the AdventureWorks database.  This is probably not something you would do, but this helps illustrate the differences of these two operators.

There are 290 rows in table dbo.Employee.

|  |
| --- |
| SELECT \* FROM dbo.Employee  UNION ALL  SELECT \* FROM dbo.Employee  UNION ALL  SELECT \* FROM dbo.Employee |

When this query is run the result set has 870 rows.  This is the 290 rows returned 3 times.  The data is just put together one dataset on top of the other dataset.

Here is the execution plan for this query.  We can see that the table was queried 3 times and SQL Server did a Concatenation step to concatenate all of the data.

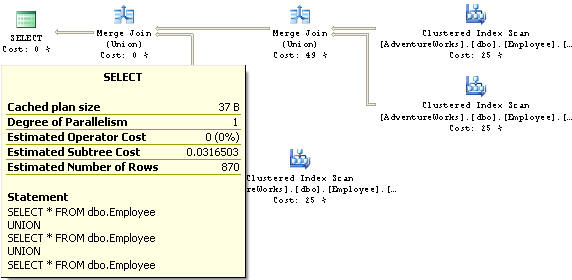


In this next example we are using the UNION operator against the Employee table again from the AdventureWorks database.

|  |
| --- |
| SELECT \* FROM dbo.Employee  UNION  SELECT \* FROM dbo.Employee  UNION  SELECT \* FROM dbo.Employee |

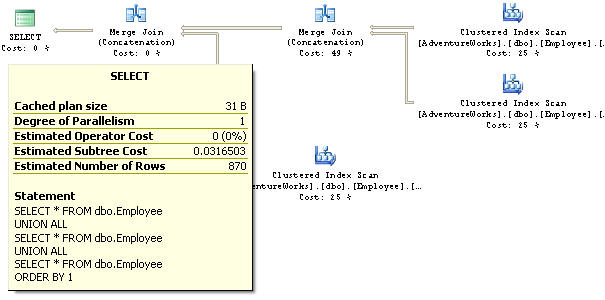
When this query is run the result set has 290 rows.  Even though we combined the data three times the UNION operator removed the duplicate records and therefore returns just the 290 unique rows.

Here is the execution plan for this query.  We can see that SQL Server first queried 2 of the tables, then did a Merge Join operation to combine the first two tables and then it did another Merge Join along with querying the third table in the query.  So we can see there was much more worked that had to be performed to get this result set compared to the UNION ALL.

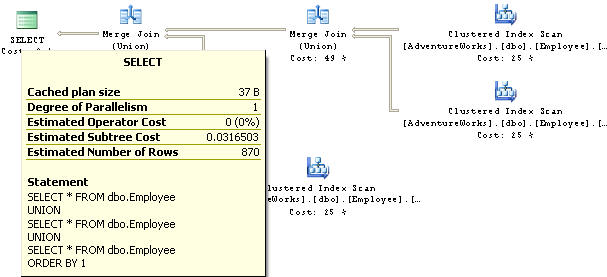


If we take this a step further and do a SORT of the data using the Clustered Index column we get these execution plans. From this we can see that the execution plan that SQL Server is using is identical for each of these operations even though the final result sets will still contain 870 rows for the UNION ALL and 290 rows for the UNION.

UNION ALL query

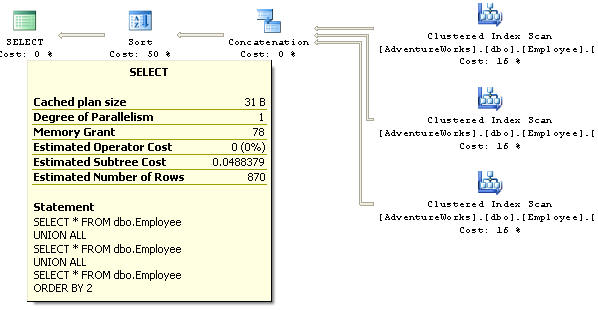


UNION query

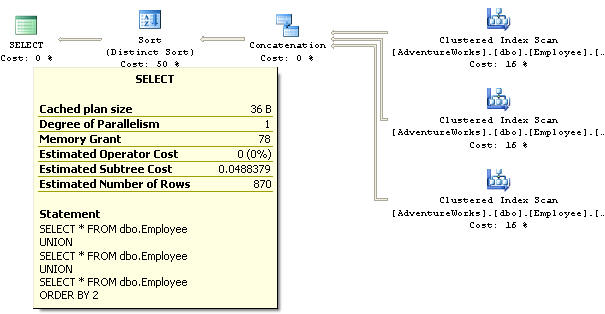


Here is another example doing the same thing, but this time doing a SORT on a non indexed column. As you can see the execution plans are again identical for these two queries, but this time instead of using a MERGE JOIN, a CONCATENATION and SORT operations are used.

UNION ALL query



UNION query



# Comparing Multiple SQL Server Datasets with the INTERSECT and EXCEPT operators

**Problem**  
When joining multiple datasets you have always had the ability to use the UNION and UNION ALL operator to allow you to pull a distinct result set (union) or a complete result set (union all).  These are very helpful commands when you need to pull data from different tables and show the results as one unified distinct result set.  On the opposite side of this it would be helpful to only show a result set where both sets of data match or only where data exists in one of the tables and not the other. This could be done with using different join types, but what other options does SQL Server offer?

**Solution**With SQL Server 2005, Microsoft introduced the INTERSECT and EXCEPT operators to further extend what you could already do with the UNION and UNION ALL operators.

* INTERSECT - gives you the final result set where values in both of the tables match
* EXCEPT - gives you the final result set where data exists in the first dataset and not in the second dataset

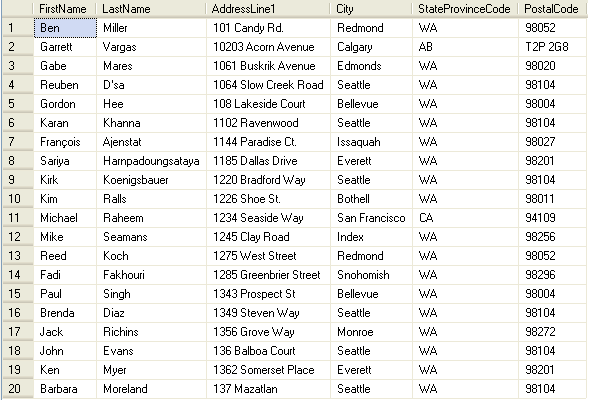
The advantage of these commands is that it allows you to get a distinct listing across all of the columns such as the UNION and UNION ALL operators do without having to do a group by or do a comparison of every single column.

Like the UNION and UNION ALL operators the table structures need to be consistent as well as the columns need to have compatible data types.

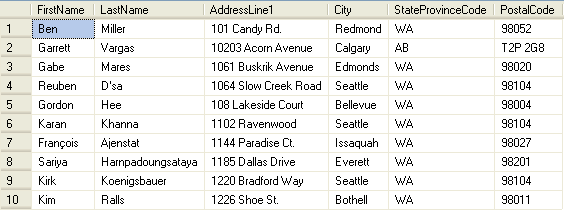
Let's take for example we have two tables manager and customer.  Both of these tables have somewhat the same structure such as the following columns:

* FirstName
* LastName
* AddressLine1
* City
* StateProvinceCode
* PostalCode

Manager table sample data



Customer table sample data



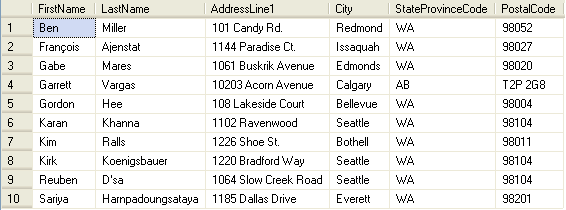
We want to do two queries:

1. Find the occurrences where a manager is a customer (intersect)
2. Find the occurrences where the manager is not a customer (except)

INTERSECT  
If we want to find out which people exist in both the customer table and the manager table and get a distinct list back we can issue the following command:

|  |
| --- |
| SELECT FIRSTNAME,         LASTNAME,         ADDRESSLINE1,         CITY,         STATEPROVINCECODE,         POSTALCODE  FROM   MANAGER  INTERSECT   SELECT FIRSTNAME,         LASTNAME,         ADDRESSLINE1,         CITY,         STATEPROVINCECODE,         POSTALCODE  FROM   CUSTOMER |

Here is the result set:



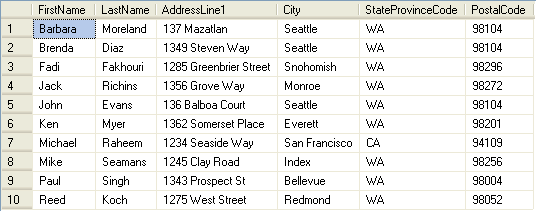
To do this same thing with a regular T-SQL command we would have to write the following:

|  |
| --- |
| SELECT   M.FIRSTNAME,           M.LASTNAME,           M.ADDRESSLINE1,           M.CITY,           M.STATEPROVINCECODE,           M.POSTALCODE  FROM     MANAGER M           INNER JOIN CUSTOMER C             ON M.FIRSTNAME = C.FIRSTNAME                AND M.LASTNAME = C.LASTNAME                AND M.ADDRESSLINE1 = C.ADDRESSLINE1                AND M.CITY = C.CITY                AND M.POSTALCODE = C.POSTALCODE  GROUP BY M.FIRSTNAME,M.LASTNAME,M.ADDRESSLINE1,M.CITY,           M.STATEPROVINCECODE,M.POSTALCODE |

EXCEPT  
If we want to find out which people exists in the manager table, but not in the customer table and get a distinct list back we can issue the following command:

|  |
| --- |
| SELECT FIRSTNAME,         LASTNAME,         ADDRESSLINE1,         CITY,         STATEPROVINCECODE,         POSTALCODE  FROM   MANAGER   EXCEPT  SELECT FIRSTNAME,         LASTNAME,         ADDRESSLINE1,         CITY,         STATEPROVINCECODE,         POSTALCODE  FROM   CUSTOMER |

Here is the result set:

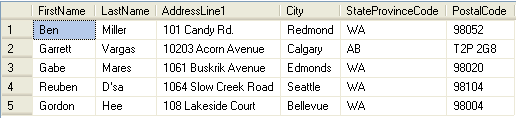


To do this same thing with a regular T-SQL command we would have to write the following:

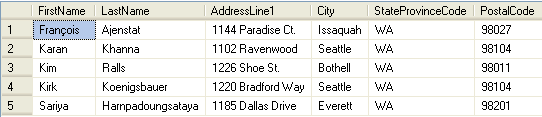
|  |
| --- |
| SELECT   M.FIRSTNAME,           M.LASTNAME,           M.ADDRESSLINE1,           M.CITY,           M.STATEPROVINCECODE,           M.POSTALCODE  FROM     MANAGER M  WHERE    NOT EXISTS (SELECT \*                       FROM   CUSTOMER C                       WHERE  M.FIRSTNAME = C.FIRSTNAME                              AND M.LASTNAME = C.LASTNAME                              AND M.ADDRESSLINE1 = C.ADDRESSLINE1                              AND M.CITY = C.CITY                              AND M.POSTALCODE = C.POSTALCODE)  GROUP BY M.FIRSTNAME,M.LASTNAME,M.ADDRESSLINE1,M.CITY,           M.STATEPROVINCECODE,M.POSTALCODE |

From the two examples above we can see that using the EXCEPT and INTERSECT commands are much simpler to write then having to write the join or exists statements.To take this a step further if we had a third table (or forth...) that listed sales reps and we wanted to find out which managers were customers, but not sales reps we could do the following.

SalesRep table sample data



|  |
| --- |
| SELECT FIRSTNAME,         LASTNAME,         ADDRESSLINE1,         CITY,         STATEPROVINCECODE,         POSTALCODE  FROM   MANAGER  INTERSECT   SELECT FIRSTNAME,         LASTNAME,         ADDRESSLINE1,         CITY,         STATEPROVINCECODE,         POSTALCODE  FROM   CUSTOMER   EXCEPT  SELECT FIRSTNAME,         LASTNAME,         ADDRESSLINE1,         CITY,         STATEPROVINCECODE,         POSTALCODE  FROM   SALESREP |

Here is the result set:

As you can see this is pretty simple to mix and match these statements.  In addition, you could also use the UNION and UNION ALL operators to further extend your final result sets.