



# JOINS, UNIONS and Subqueries

---

.NET CORE

*A JOIN is a means for combining columns from one or more tables by using values common to each.*

[HTTPS://EN.WIKIPEDIA.ORG/WIKI/JOIN\\_\(SQL\)](https://en.wikipedia.org/wiki/JOIN_(SQL))

# SQL JOIN Statements – Overview

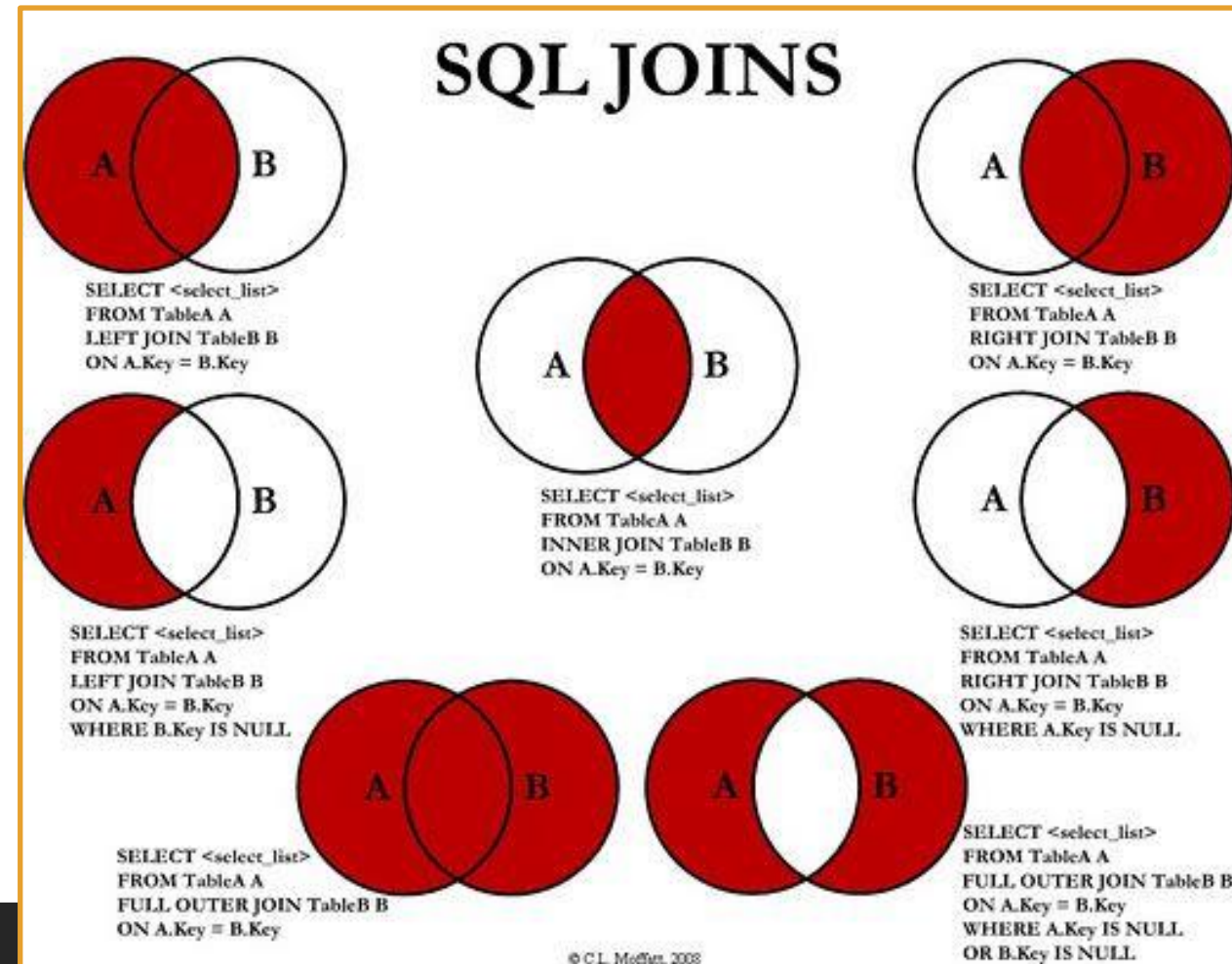
<https://docs.microsoft.com/en-us/sql/relational-databases/performance/joins?view=sql-server-ver15#fundamentals>

*JOINS* tell SQL how to use data from one table to select rows in a different table.

A JOIN statement defines the relationship between the tables by using keywords to:

- Specifying the column from each table to be compared for the JOIN.
- Specifying a logical operator ( = or <>,) for comparing values from the indicated columns.

Typically, a JOIN condition uses a *Foreign Key* from one table and its associated *Key* in the other table.

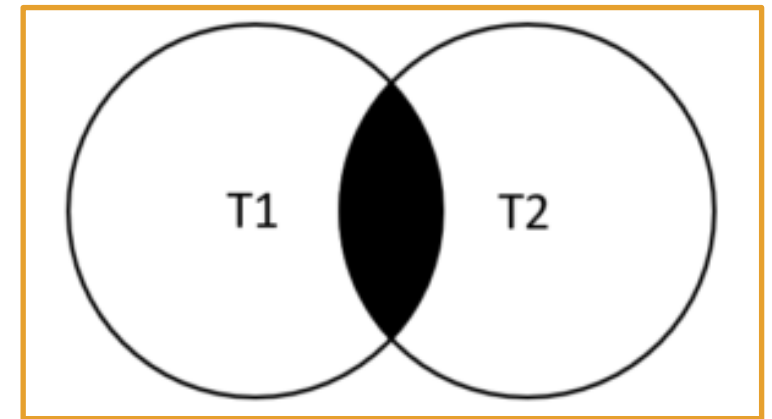


# SQL JOIN Statement

<https://docs.microsoft.com/en-us/sql/relational-databases/performance/joins?view=sql-server-ver15#fundamentals>

The **JOIN** (aka INNER JOIN) Keyword combines with the WHERE and HAVING search conditions to control which rows are selected from the tables referenced in the FROM clause. Specifying the **JOIN** conditions in the **FROM** clause helps separate them from any other search conditions that may be specified in a **WHERE** clause. Below is the recommended clause to use for specifying **JOINS**.

**INNER JOINS** eliminate the rows that do not match with a row from the other table.



**FROM [first\_table] [join\_type] [second\_table] ON [join\_condition]**

This example:

- Specifies 3 unambiguous column names to return from the desired tables,
- Specifies the 2 tables to JOIN,
- Specifies the columns with shared data ON which to JOIN,
- Sets 2 constraints to filter on the results reported

```
SELECT ProductID, Purchasing.Vendor.BusinessEntityID, Name
FROM Purchasing.ProductVendor JOIN Purchasing.Vendor
ON (Purchasing.ProductVendor.BusinessEntityID =
    Purchasing.Vendor.BusinessEntityID)
WHERE StandardPrice > $10 AND Name LIKE N'F%'
GO
```

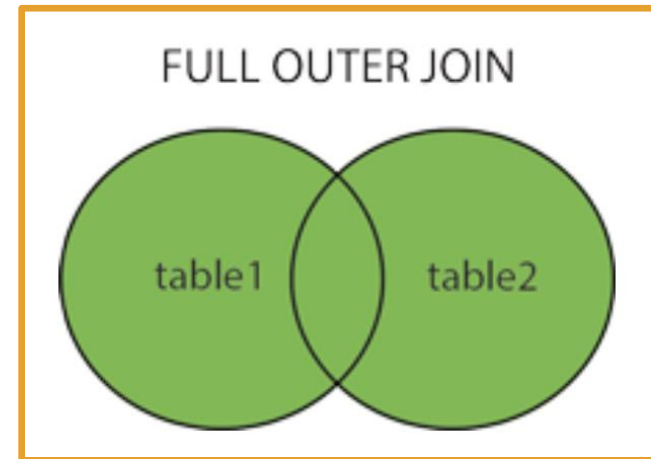


# SQL FULL JOIN Statement

<https://docs.microsoft.com/en-us/sql/relational-databases/performance/joins?view=sql-server-ver15#fundamentals>  
[https://www.w3schools.com/sql/sql\\_join\\_full.asp](https://www.w3schools.com/sql/sql_join_full.asp)

**FULL JOINS** (aka **OUTER JOIN**) return all rows from at least one of the tables or views mentioned in the FROM clause, if those rows meet any **WHERE** or **HAVING** search conditions.

The order in which tables appear in an OUTER JOIN statement is significant. The first table mentioned is the "left" table and the second table is the "right" table. When you specify a left or right outer join, you are referring to the order in which the tables were added to the query and to the order in which they appear in the SQL statement in the SQL pane.



This example:

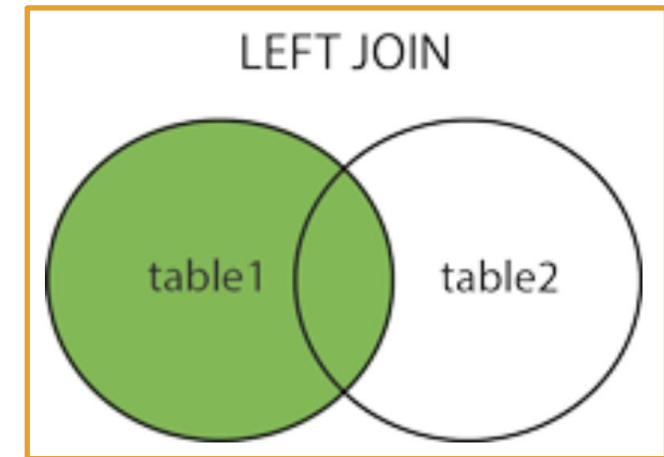
- Specifies 2 unambiguous column names to return from the desired tables, → **SELECT Customers.CustomerName, Orders.OrderID**
- Specifies the 2 tables to FULL OUTER JOIN, → **FROM Customers FULL OUTER JOIN Orders**
- Specifies the columns with shared data ON which to JOIN, → **ON Customers.CustomerID=Orders.CustomerID**
- Sets a constraint to order the result by CustomerName of the Customers table, Ascending (default). → **ORDER BY Customers.CustomerName;**

# SQL LEFT JOIN Statement

[https://www.w3schools.com/sql/sql\\_join\\_left.asp](https://www.w3schools.com/sql/sql_join_left.asp)

When you specify a LEFT or RIGHT OUTER JOIN, you are referring to the order in which the tables were written in the query. The first table mentioned in the query is the "left" table and the second table is the "right" table.

The LEFT JOIN keyword returns all records from the left table (table1), and the matched records from the right table (table2). The result is NULL from the right side, if there is no match.



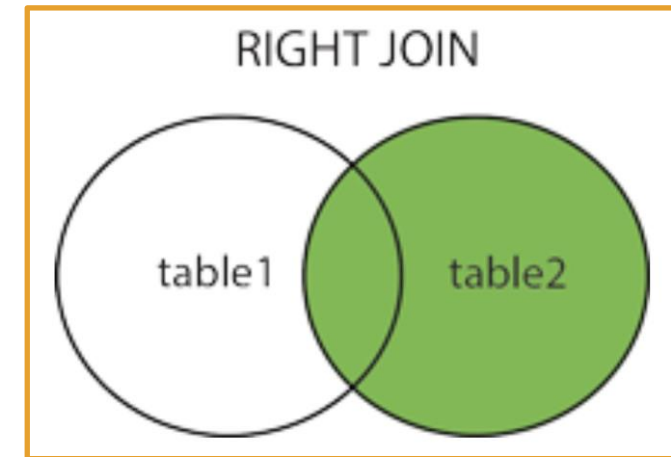
This example:

- Specifies 2 unambiguous column names to return from the desired tables, → `SELECT Customers.CustomerName, Orders.OrderID`
- Specifies the 2 tables to LEFT JOIN, → `FROM Customers LEFT JOIN Orders`
- Specifies the columns with shared data ON which to LEFT JOIN, → `ON Customers.CustomerID = Orders.CustomerID`
- Sets a constraint to order the result by CustomerName of the Customers table, Ascending (default). → `ORDER BY Customers.CustomerName;`

# SQL RIGHT JOIN Statement

The first table mentioned in the query is the "left" table and the second table is the "right" table. When you specify a LEFT or RIGHT **OUTER JOIN**, you are referring to the order in which the tables were written in the query.

The **RIGHT JOIN** keyword returns all records from the right table (table2), and the matched records from the left table (table1). The result is NULL from the left side, if there is no match.



This example:

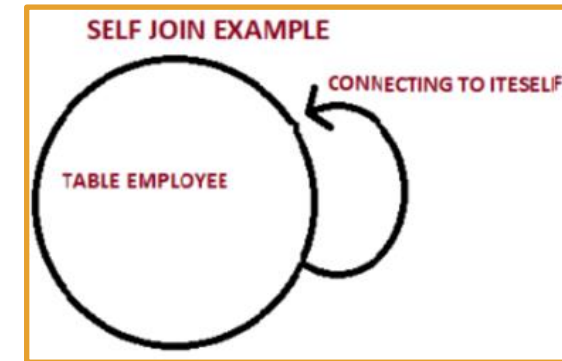
- Specifies 3 unambiguous column names to return from the desired tables,
- Specifies the 2 tables to RIGHT JOIN,
- Specifies the columns with shared data ON which to RIGHT JOIN,
- Sets a constraint to order the result by OrderID of the Orders table, Ascending (default).

```
SELECT Orders.OrderID, Employees.LastName,  
Employees.FirstName  
FROM Orders RIGHT JOIN Employees  
ON Orders.EmployeeID = Employees.EmployeeID  
ORDER BY Orders.OrderID;
```

# SQL SELF JOIN

[https://www.w3schools.com/sql/sql\\_join\\_self.asp](https://www.w3schools.com/sql/sql_join_self.asp)

A **SELF JOIN** is exactly like a **JOIN**. The table is joined with a duplicate version of itself and the result generated.



This example:

- Specifies 3 unambiguous column names to return from the desired tables. The AS keyword allows you to designate a unique identifier.
- Specifies the 2 tables to SELF JOIN. No keyword is required, just a comma between the table names.
- Specifies the filter of CustomerID's that are not equal, but the city is the same.
- Sets a constraint to order the result by City name, Ascending (default).

```
SELECT A.CustomerName AS  
CustomerName1, B.CustomerName AS  
CustomerName2, A.City  
FROM Customers A, Customers B  
WHERE A.CustomerID <> B.CustomerID  
AND A.City = B.City  
ORDER BY A.City;
```

<> means != (not equal to)

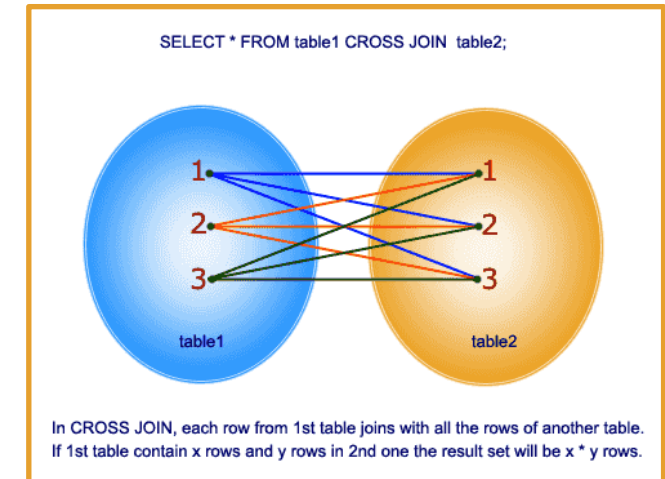


# SQL CROSS JOIN

<https://www.w3resource.com/sql/joins/cross-join.php>

CROSS JOIN produces a Cartesian Product. This is a result set which is the number of rows in the first table multiplied by the number of rows in the second table.

If the WHERE clause is used with CROSS JOIN, it functions like an INNER JOIN. You can also use comma-separated column names after SELECT and enter comma-separated table names after the FROM keyword.



This example:

- Specifies 4 unambiguous column names from the desired tables.
- Specifies the 2 tables FROM which to CROSS JOIN. The first table to multiplied by all items in second table.

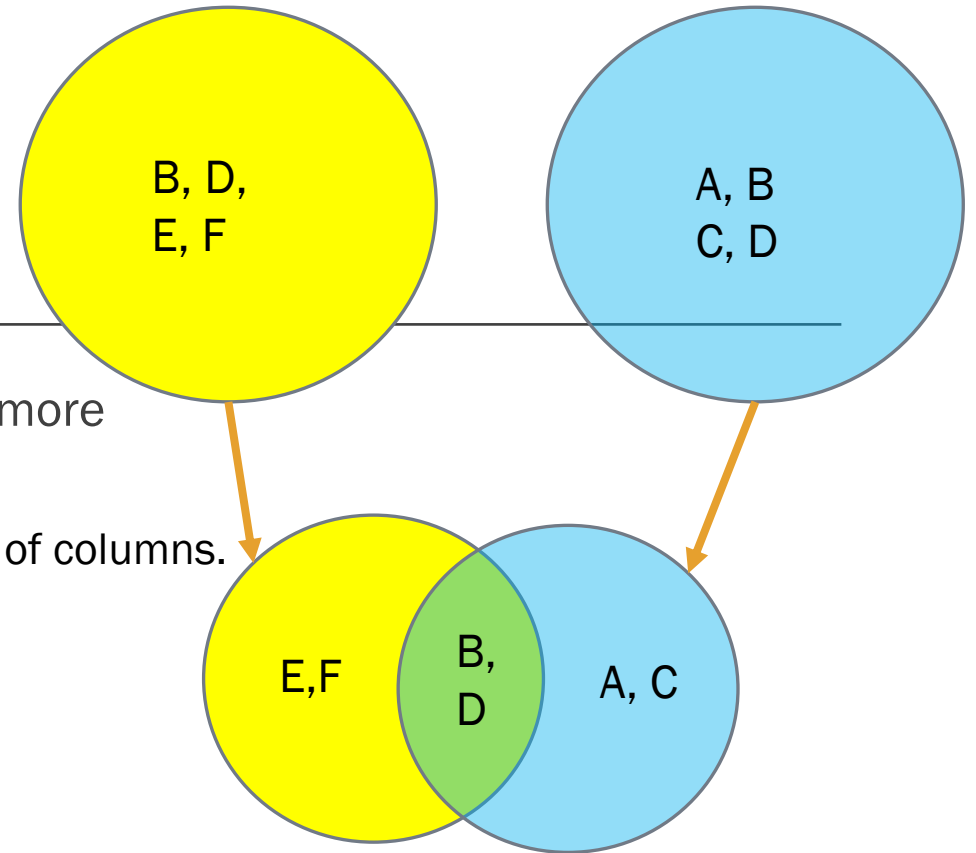
```
SELECT foods.item_name,foods.item_unit,  
company.company_name,company.company_city  
FROM foods CROSS JOIN company;
```

# SQL UNION Operator

[w3schools.com/sql/sql\\_union.asp](http://w3schools.com/sql/sql_union.asp)

The **UNION** operator is used to combine the result-set of two or more **SELECT** statements.

- Each **SELECT** statement within **UNION** must have the same number of columns.
- The columns must also have similar data types.
- The columns in each **SELECT** statement must be in the same order.
- **UNION** selects only distinct values.



This example:

- Creates a complete **SELECT** statement
- Uses the **UNION** keyword
- Creates a separate **SELECT** statement identical to the first but querying a different table
- Sets a constraint to order the result by City name, Ascending (default).

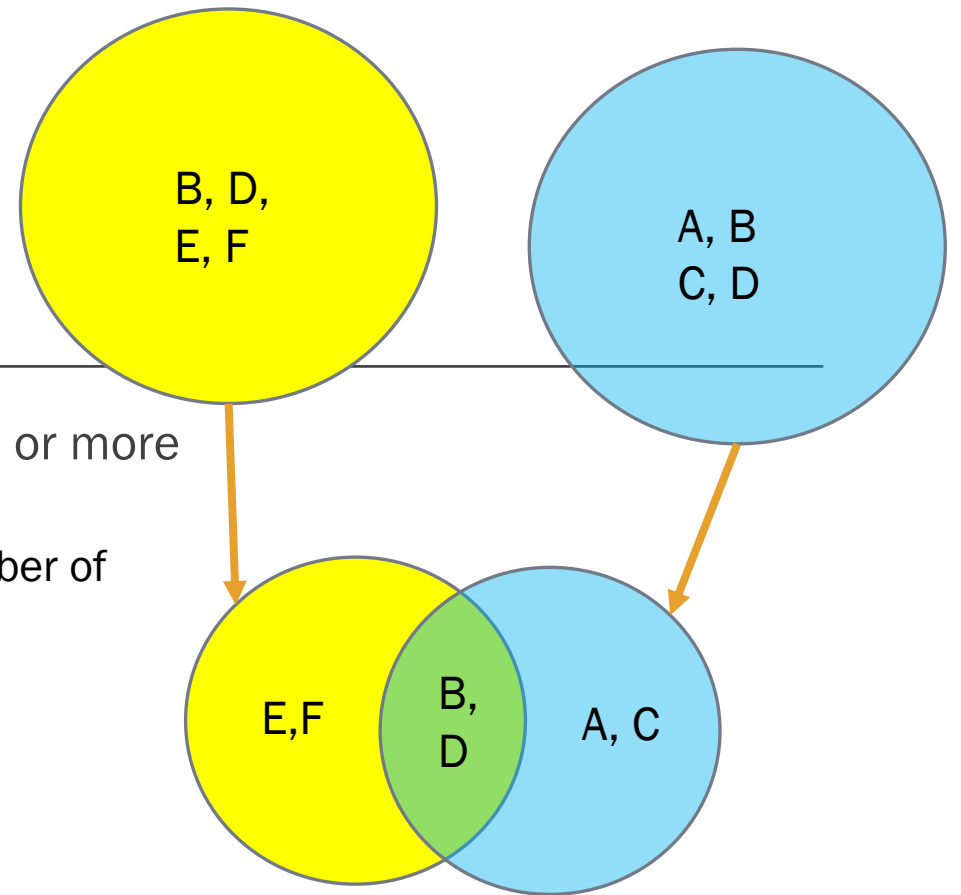
```
SELECT City, Country FROM Customers
WHERE Country='Germany'
UNION
SELECT City, Country FROM Suppliers
WHERE Country='Germany'
ORDER BY City;
```

# SQL UNION ALL Operator

[w3schools.com/sql/sql\\_union.asp](http://w3schools.com/sql/sql_union.asp)

The **UNION ALL** operator is used to combine the result-set of two or more **SELECT** statements.

- Each **SELECT** statement within **UNION ALL** must have the same number of columns.
- The columns must have similar data types.
- The columns in each **SELECT** statement must be in the same order.
- **UNION ALL** returns all values, even duplicates.



This example:

- Creates a complete **SELECT** statement
- Uses the **UNION ALL** keyword
- Creates a separate **SELECT** statement identical to the first but querying a different table
- Sets a constraint to order the result by City name, Ascending (default). This result includes duplicates.

```
SELECT City, Country FROM Customers  
WHERE Country='Germany'  
UNION ALL  
SELECT City, Country FROM Suppliers  
WHERE Country='Germany'  
ORDER BY City;
```

# SQL INTERSECT and EXCEPT Operators

<https://docs.microsoft.com/en-us/sql/t-sql/language-elements/set-operators-except-and-intersect-transact-sql?view=sql-server-ver15>

---

## EXCEPT:

- Returns any distinct values from the query left of the **EXCEPT** operator. Those values return as long as the right query doesn't return those values as well.

## INTERSECT:

- Returns any distinct values that are returned by both the query on the left and right sides of the INTERSECT operator.

To combine the result sets of two queries that use EXCEPT or INTERSECT, the basic rules are:

- The number and the order of the columns must be the same in all queries.
- The data types must be compatible (implicitly convertible).
- **EXCEPT** and **INTERSECT** return the result set's column names that are the same as the column names that the query on the operator's left side returns.

# SQL INTERSECT and EXCEPT Operators - Examples

<https://docs.microsoft.com/en-us/sql/t-sql/language-elements/set-operators-except-and-intersect-transact-sql?view=sql-server-ver15>

---

This *INTERSECT* example returns any distinct values that are returned by both the query on the left (above) and right (below) sides of the *INTERSECT* operator. This shows all products that have been ordered.

```
SELECT ProductID FROM Production.Product  
  
INTERSECT  
  
SELECT ProductID FROM Production.WorkOrder ;
```

This *EXCEPT* example returns any distinct values from the query left of the *EXCEPT* operator that are NOT also found on the right query. This shows all products that have NOT been ordered.

```
SELECT ProductID FROM Production.Product  
  
EXCEPT  
  
SELECT ProductID FROM Production.WorkOrder ;
```



# SQL Subquery

<https://docs.microsoft.com/en-us/sql/relational-databases/performance/subqueries?view=sql-server-ver15>

---

A subquery is a query that is nested inside a **SELECT**, **INSERT**, **UPDATE**, or **DELETE** statement, or inside another subquery. A subquery can be used anywhere an expression is allowed. The subquery is evaluated then the outer query is evaluated against it.

Many SQL statements that include subqueries can be formulated as **JOINS**. There is usually no performance difference between a statement with a subquery and an equivalent **JOIN**. In some cases, where existence must be checked, a **JOIN** yields better performance. The nested query must be processed for each result of the outer query to ensure elimination of duplicates. In such cases, a **JOIN** approach would yield better results.

In this example, a subquery is used as a column expression named `MaxUnitPrice` in a **SELECT** statement. The subquery obtains the maximum price for each unit and then becomes a column in the outer query result.

```
SELECT Ord.SalesOrderID, Ord.OrderDate,  
       (SELECT MAX(OrdDet.UnitPrice)  
        FROM Sales.SalesOrderDetail AS OrdDet  
        WHERE Ord.SalesOrderID = OrdDet.SalesOrderID) AS MaxUnitPrice  
FROM Sales.SalesOrderHeader AS Ord;
```

# Subquery

<https://docs.microsoft.com/en-us/sql/relational-databases/performance/subqueries?view=sql-server-ver15>

---

A subquery nested in an outer **SELECT** statement has the following components:

- A regular **SELECT** query including the regular select list components.
- A regular **FROM** clause including one or more table or view names.
- An optional **WHERE** clause.
- An optional **GROUP BY** clause.
- An optional **HAVING** clause.

This example shows two queries obtaining an identical result. One uses a subquery and the other uses a **JOIN**.

```
SELECT Name
FROM Production.Product
WHERE ListPrice =
    (SELECT ListPrice
     FROM Production.Product
     WHERE Name = 'Chainring Bolts' );
```

```
SELECT Prd1.Name
FROM Production.Product AS Prd1
JOIN Production.Product AS Prd2
    ON (Prd1.ListPrice = Prd2.ListPrice)
WHERE Prd2.Name = 'Chainring Bolts';
```

# CTE (Common Table Expression)

<https://docs.microsoft.com/en-us/sql/t-sql/queries/with-common-table-expression-transact-sql?view=sql-server-ver15>

A *Common Table Expression* is an expression that first creates a temporary table that is then queried by the following query. A *Recursive Common Table Expression* is a CTE which references itself.

1. Define the CTE expression name and column list.
  1. Use the WITH keyword followed by the CTE name and a list of comma-separated columns.
2. Define the CTE query.
  1. Use the AS keyword and write a query (in parenthesis) to query an existing table.
3. Define the outer query referencing the CTE name.
  1. Create a regular query with one exception.
  2. After **SELECT** and the column list,
  3. use the FROM keyword and state the name of the desired CTE
  4. Complete the query as normal
4. Now your outer query will use the CTE as its source.

```
WITH Sales_CTE (SalesPersonID, SalesOrderID, SalesYear)
AS
(
    SELECT SalesPersonID, SalesOrderID, YEAR(OrderDate)
        AS SalesYear
    FROM Sales.SalesOrderHeader
    WHERE SalesPersonID IS NOT NULL
)
SELECT SalesPersonID,
        COUNT(SalesOrderID) AS TotalSales,
        SalesYear
FROM Sales_CTE
GROUP BY SalesYear, SalesPersonID
ORDER BY SalesPersonID, SalesYear;
```

# Pushpinder Lecture

- She is using her DB backup. IT has modules for learning! Seems like a great resource.
- She's using the whiteboard to explain what Joins are Join – Combine multiple tables and query them.
- Which command executes first?
- FROM -> WHERE -> GROUPBY -> HAVING -> SELECT -> ORDERBY
- FROM creates a virtual table. That populates the virtual table with everything from the indicated table.
- Then it passes the virtual table through all the commands till select. Then another virtual table is created and ordered
- SET THEORY – She explains that SQL is based on set theory. shares a ppt she created.
- This ppt is great for a general SQL intro, too. (The slideshow is Basic Joins.pdf)
- PDF defines what each type of join is., what is a 'predicate' (is from, etc)
- She moves over to SSMS to model join types.
- Left join, right join, outer join. Full outer join.

After modeling everything, she gives them an assignment to create certain tables with attributes. And perform assigned

joins.

Joins : Combine multiple tables and query them

DQL-> Select

```
use TSQL;  
Select empid, year(orderdate) as OrderYear  
from Sales.Order  
where custid=71  
group by empid, year(orderdate)  
Having count(*)>1  
order by empid. OrderYear
```

from->where->groupby->having->select->orderby

- Combine all possible combinations of two sets
- In T-SQL queries, usually not desired
- Special case: table of numbers

Name	Product
Davis	Alice Mutton
Davis	Crab Meat

