

SQL Constraints

SQL constraints are used to specify rules for the data in a table.

If there is any violation between the constraint and the data action, the action is aborted by the constraint.

Constraints can be specified when the table is created (inside the CREATE TABLE statement) or after the table is created (inside the ALTER TABLE statement).

SQL CREATE TABLE + CONSTRAINT Syntax

```
CREATE TABLE table_name
(
  column_name1 data_type(size) constraint_name,
  column_name2 data_type(size) constraint_name,
  column_name3 data_type(size) constraint_name,
  ....
);
```

In SQL, we have the following constraints:

- **NOT NULL** - Indicates that a column cannot store NULL value
- **UNIQUE** - Ensures that each row for a column must have a unique value
- **PRIMARY KEY** - A combination of a NOT NULL and UNIQUE. Ensures that a column (or combination of two or more columns) have an unique identity which helps to find a particular record in a table more easily and quickly
- **FOREIGN KEY** - Ensure the referential integrity of the data in one table to match values in another table
- **CHECK** - Ensures that the value in a column meets a specific condition
- **DEFAULT** - Specifies a default value when specified none for this column

SQL NOT NULL Constraint

By default, a table column can hold NULL values.

SQL NOT NULL Constraint

The NOT NULL constraint enforces a column to NOT accept NULL values.

The NOT NULL constraint enforces a field to always contain a value. This means that you cannot insert a new record, or update a record without adding a value to this field.

The following SQL enforces the "P_Id" column and the "LastName" column to not accept NULL values:

Example

```
CREATE TABLE PersonsNotNull
(
  P_Id int NOT NULL,
  LastName varchar(255) NOT NULL,
  FirstName varchar(255),
  Address varchar(255),
  City varchar(255)
)
```

SQL UNIQUE Constraint

The UNIQUE constraint uniquely identifies each record in a database table.

The UNIQUE and PRIMARY KEY constraints both provide a guarantee for uniqueness for a column or set of columns.

A PRIMARY KEY constraint automatically has a UNIQUE constraint defined on it.

Note that you can have many UNIQUE constraints per table, but only one PRIMARY KEY constraint per table.

SQL UNIQUE Constraint on CREATE TABLE

The following SQL creates a UNIQUE constraint on the "P_Id" column when the "Persons" table is created:

SQL Server / Oracle / MS Access:

```
CREATE TABLE Persons
(
  P_Id int NOT NULL UNIQUE,
  LastName varchar(255) NOT NULL,
  FirstName varchar(255),
  Address varchar(255),
```

```
City varchar(255)
)
```

SQL UNIQUE Constraint on ALTER TABLE

To create a UNIQUE constraint on the "P_Id" column when the table is already created, use the following SQL:

MySQL / SQL Server / Oracle / MS Access:

```
ALTER TABLE Persons
ADD UNIQUE (P_Id)
```

To DROP a UNIQUE Constraint

To drop a UNIQUE constraint, use the following SQL:

SQL Server / Oracle / MS Access:

```
ALTER TABLE Persons
DROP CONSTRAINT uc_PersonID
```

SQL PRIMARY KEY Constraint

The PRIMARY KEY constraint uniquely identifies each record in a database table.

Primary keys must contain unique values.

A primary key column cannot contain NULL values.

Most tables should have a primary key, and each table can have only ONE primary key.

SQL PRIMARY KEY Constraint on CREATE TABLE

The following SQL creates a PRIMARY KEY on the "P_Id" column when the "Persons" table is created:

SQL Server / Oracle / MS Access:

```
CREATE TABLE Persons
(
P_Id int NOT NULL PRIMARY KEY,
```

```
LastName varchar(255) NOT NULL,  
FirstName varchar(255),  
Address varchar(255),  
City varchar(255)  
)
```

SQL FOREIGN KEY Constraint

A FOREIGN KEY in one table points to a PRIMARY KEY in another table.

Let's illustrate the foreign key with an example. Look at the following two tables:

The "Persons" table:

| P_Id | LastName | FirstName | Address | City |
|------|-----------|-----------|--------------|-----------|
| 1 | Hansen | Ola | Timoteivn 10 | Sandnes |
| 2 | Svendson | Tove | Borgvn 23 | Sandnes |
| 3 | Pettersen | Kari | Storgt 20 | Stavanger |

The "Orders" table:

| O_Id | OrderNo | P_Id |
|------|---------|------|
| 1 | 77895 | 3 |
| 2 | 44678 | 3 |
| 3 | 22456 | 2 |
| 4 | 24562 | 1 |

Note that the "P_Id" column in the "Orders" table points to the "P_Id" column in the "Persons" table.

The "P_Id" column in the "Persons" table is the PRIMARY KEY in the "Persons" table.

The "P_Id" column in the "Orders" table is a FOREIGN KEY in the "Orders" table.

The FOREIGN KEY constraint is used to prevent actions that would destroy links between tables.

The FOREIGN KEY constraint also prevents invalid data from being inserted into the foreign key column, because it has to be one of the values contained in the table it points to.

SQL Server / Oracle / MS Access:

```
CREATE TABLE Orders
(
O_Id int NOT NULL PRIMARY KEY,
OrderNo int NOT NULL,
P_Id int FOREIGN KEY REFERENCES Persons(P_Id)
)
```

SQL CHECK Constraint

The CHECK constraint is used to limit the value range that can be placed in a column.

If you define a CHECK constraint on a single column it allows only certain values for this column.

If you define a CHECK constraint on a table it can limit the values in certain columns based on values in other columns in the row.

SQL CHECK Constraint on CREATE TABLE

The following SQL creates a CHECK constraint on the "P_Id" column when the "Persons" table is created. The CHECK constraint specifies that the column "P_Id" must only include integers greater than 0.

SQL Server / Oracle / MS Access:

```
CREATE TABLE Persons
(
P_Id int NOT NULL CHECK (P_Id>0),
LastName varchar(255) NOT NULL,
FirstName varchar(255),
Address varchar(255),
City varchar(255)
)
```

SQL DEFAULT Constraint

The DEFAULT constraint is used to insert a default value into a column.

The default value will be added to all new records, if no other value is specified.

SQL DEFAULT Constraint on CREATE TABLE

The following SQL creates a DEFAULT constraint on the "City" column when the "Persons" table is created:

My SQL / SQL Server / Oracle / MS Access:

```
CREATE TABLE Persons
(
P_Id int NOT NULL,
LastName varchar(255) NOT NULL,
FirstName varchar(255),
Address varchar(255),
City varchar(255) DEFAULT 'Sandnes'
)
```

Defining a query

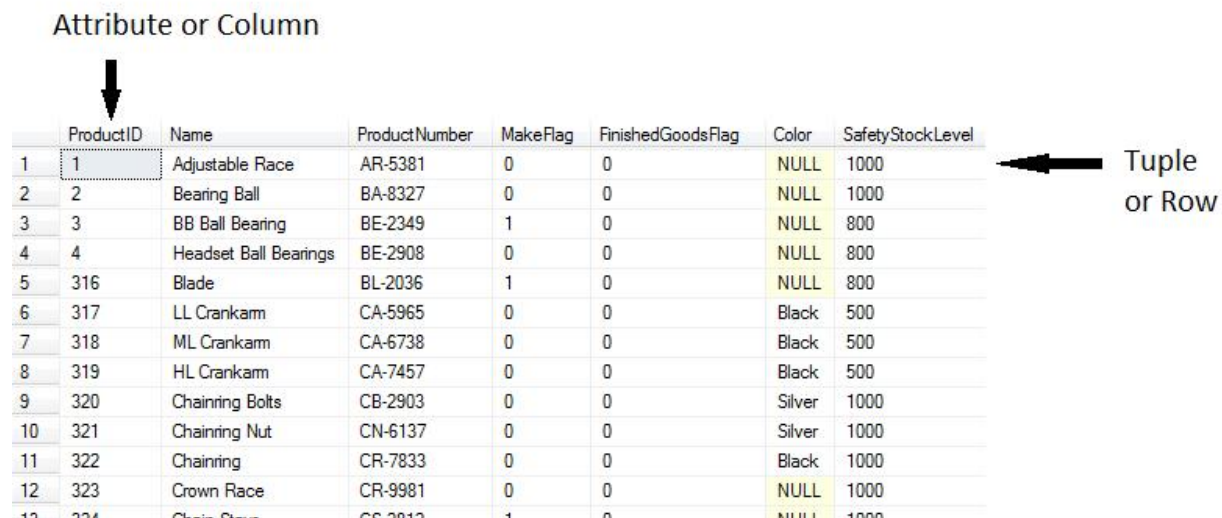
So now that you have AdventureWorks up and running we can actually begin writing our first *query*. Before we do that I want you to think a moment about what a query is. Seriously, think about it. Wikipedia has the following to say about it; "In general, a query is a form of questioning, in a line of inquiry". Alright, so we are going to question the database.

So *how* exactly are we going to question the database? When you ask a friend to get you something, say a drink, you would probably ask your friend something like "Can you get me a drink?". If you want to be polite you would insert the word "please" somewhere in that sentence, but it would not really alter the meaning of your question, or query. Of course a database would not understand such a question. A database can only 'understand' a predefined set of statements in a particular order, it is *structured*. Now guess what, SQL actually stands for *Structured Query Language*! This language is actually the part about databases I will be explaining to you in this article.

I should mention that SQL Server does not use the ANSI and ISO SQL standard syntax completely. Instead it uses T-SQL (Transact-SQL), which is one of the dialects of the SQL standard. This basically means that the queries in the article might also work in other SQL databases such as Oracle, MySQL, PostgreSQL and Firebird, but that this is not guaranteed (also because all these other databases have their own dialects as well). This article is not about differences in SQL dialects, but I did want to mention it here.

Before we start we need to know *what* we are going to query. As you might know SQL Server is a [relational database](#). It is called this because it stores data in [relations](#) (and not because there is a relation between one piece of data and another, as many think). A relation is a mathematical term for a collection of unique and unordered [tuples](#), or sets of values (or attributes). In SQL Server these are represented in [tables](#), a collection of rows and columns. These tables are exactly what we are going to query. The following picture shows a part of the Production.Product table from the database we just installed.

Attribute or Column



| | ProductID | Name | ProductNumber | MakeFlag | FinishedGoodsFlag | Color | SafetyStockLevel |
|----|-----------|-----------------------|---------------|----------|-------------------|--------|------------------|
| 1 | 1 | Adjustable Race | AR-5381 | 0 | 0 | NULL | 1000 |
| 2 | 2 | Bearing Ball | BA-8327 | 0 | 0 | NULL | 1000 |
| 3 | 3 | BB Ball Bearing | BE-2349 | 1 | 0 | NULL | 800 |
| 4 | 4 | Headset Ball Bearings | BE-2908 | 0 | 0 | NULL | 800 |
| 5 | 316 | Blade | BL-2036 | 1 | 0 | NULL | 800 |
| 6 | 317 | LL Crankarm | CA-5965 | 0 | 0 | Black | 500 |
| 7 | 318 | ML Crankarm | CA-6738 | 0 | 0 | Black | 500 |
| 8 | 319 | HL Crankarm | CA-7457 | 0 | 0 | Black | 500 |
| 9 | 320 | Chaining Bolts | CB-2903 | 0 | 0 | Silver | 1000 |
| 10 | 321 | Chaining Nut | CN-6137 | 0 | 0 | Silver | 1000 |
| 11 | 322 | Chaining | CR-7833 | 0 | 0 | Black | 1000 |
| 12 | 323 | Crown Race | CR-9981 | 0 | 0 | NULL | 1000 |
| 13 | 324 | Chain Stay | CS-2012 | 1 | 0 | NULL | 1000 |

Tuple or Row

Let us take a more practical look at this. Let us assume we want to store our CD collection in a database. One table could be a collection of CD objects. For simplicity let us say that a CD has an artist, a title and a release date. So our table is now a collection of tuples holding two names (an artist name and a title) and a date (release date). We can now query the database for all CD's by the artist John Williams or that were released in 2012. Or let us say a CD also has an attribute 'number of songs'. We might want to know the sum of all songs of all CD's by John Williams. Or perhaps the average number of songs on a CD (by artist or in total).

How to create, fill and maintain tables is not in the scope of this article. The reader is expected to have at least some knowledge about what a table is and what it looks like.

5. Our first query; the SELECT statement

I hear you thinking "when do we FINALLY get to see some SQL!?". Well, now.

```
SELECT 'Hello, query!'
```

Remember when you asked your friend for a drink? You could have said "GET Drink". He probably would not have gotten it with that tone (which is commanding rather than asking). Luckily SQL Server does anything you ask. Except we need to replace the GET in this example with SELECT. The SELECT keyword is the first word you will use in most queries. Actually any

query consists of at least the `SELECT` statement. In the above query the database simply returns a table with one row and one column with the value 'Hello query' (excluding the apostrophes).

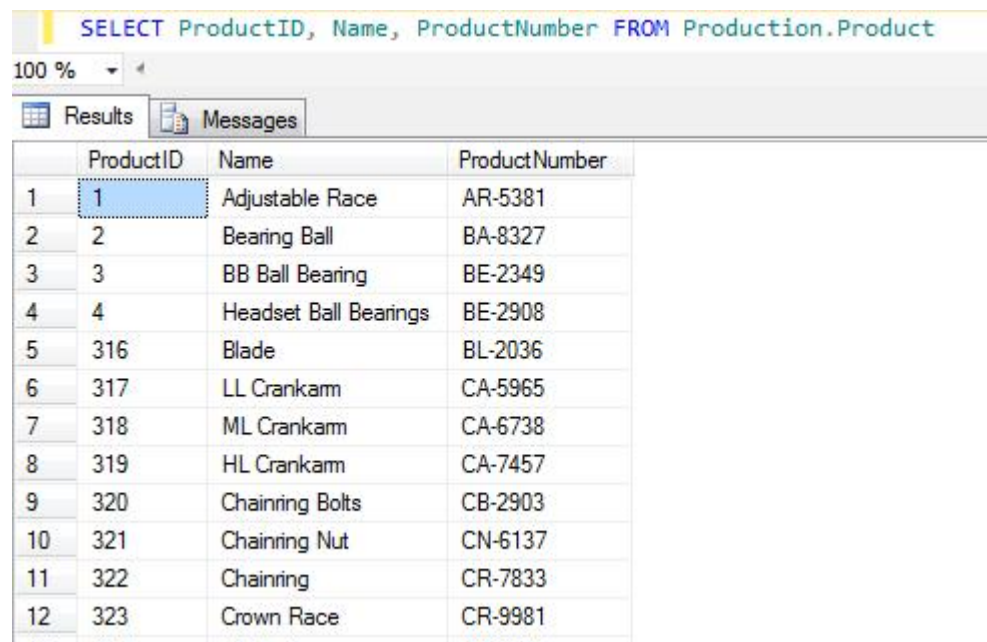
When selecting data we need to separate the 'things' we want to select with a comma. The next example produces one row with two columns. In the `SELECT` list the two values are separated by a comma.

```
SELECT 'Hello, query!', 'A second value'
```

Of course the above query is not very useful. We want to query actual data stored by the database. In order to get this data we need to first indicate where the database should get the data. This is done by using the `FROM` statement. Let us say we want to select some data about products from the AdventureWorks2012 database.

```
SELECT ProductID, Name, ProductNumber FROM Production.Product
```

Actually that query makes a lot of sense, doesn't it? `ProductID`, `Name` and `ProductNumber` are columns of the `Production.Product` table (`Production` is actually the namespace of the table `Product`, the namespace and table name need to be unique across a database). Notice again that the columns are separated by a comma. The result of the query is a table with three columns and as many rows as you have products in the `Production.Product` table.



| | ProductID | Name | ProductNumber |
|----|-----------|-----------------------|---------------|
| 1 | 1 | Adjustable Race | AR-5381 |
| 2 | 2 | Bearing Ball | BA-8327 |
| 3 | 3 | BB Ball Bearing | BE-2349 |
| 4 | 4 | Headset Ball Bearings | BE-2908 |
| 5 | 316 | Blade | BL-2036 |
| 6 | 317 | LL Crankarm | CA-5965 |
| 7 | 318 | ML Crankarm | CA-6738 |
| 8 | 319 | HL Crankarm | CA-7457 |
| 9 | 320 | Chaining Bolts | CB-2903 |
| 10 | 321 | Chaining Nut | CN-6137 |
| 11 | 322 | Chainring | CR-7833 |
| 12 | 323 | Crown Race | CR-9981 |

A shortcut to select all columns from a table is the following syntax.

```
SELECT * FROM Production.Product
```


This will return everything from the `Production.Product` table. It is strongly recommended to NOT use this syntax though. This has mainly to do with underlying table definitions and is outside the scope of this article. However, I have written about a little real-life experience using `SELECT *`, so if you are interested you can read it: [The Evil That is "Select *"](#). Having said that I am still going to use this syntax a lot in this article, just don't use it in production code!

The list of columns to select can appear in any order and do not have to be in the order they appear in columns. The following query shows an example.

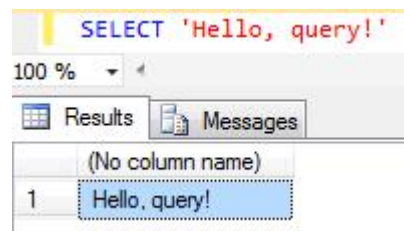
```
SELECT Color, ProductNumber, Name
FROM Production.Product
```

So let us take a look at the general `SELECT... FROM` syntax. In general you can say it look like this:

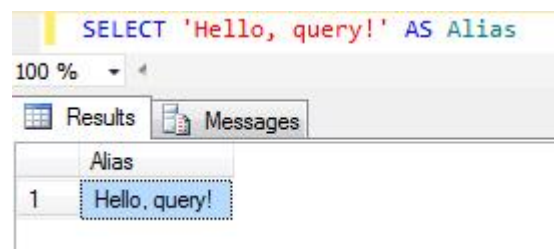
```
SELECT <List of columns> FROM <Table name>
```

5.1 Aliasing

There are a few more things you can do with even the simplest of queries. One of them is aliasing. We are actually going back to the very first query, `SELECT 'Hello, query!'`. Take a look at the output, do you notice something?



The one column returned by this query has no name! Of course it has no name. When we select columns from a table SQL Server takes over the names of the selected columns in the result by default. We did not select from a table in this example though, and as a result SQL Server does not know how to name the column. See what happens when you select an additional value. You now have two columns without a name. Luckily we are not forced to talk about columns like He Who Must Not Be Named. We can actually assign aliases to columns. There are a few ways in which you can do this and I am going to show you one (because it is the recommended syntax).



The `AS <alias>` syntax can give new names to your result columns. Notice that the `AS` keyword is optional.

Aliasing is also allowed when selecting columns from a table.

```
SELECT
    ProductID          AS ID,
    Name                AS ProductName,
    ProductNumber      AS ProductNumber,
    Color
FROM Production.Product
```

As you can see it is possible to give new names to your result columns (such as `ID` and `ProductName`), alias with the same name as the column (`ProductNumber` simply stays `ProductNumber`) or not alias at all (`Color` is not aliased). Aliasing columns becomes a must when you are going to manipulate data or use functions, which we will see later on in the article. Column aliases cannot be used in other parts of your query such as `WHERE`, `HAVING` and `ON` clauses (which are discussed later in this article). The only exception to this rule is the `ORDER BY` list (which will also be discussed later).

It is also possible to alias tables. This is especially useful when selecting from multiple tables. I will come back to that later.

So the general syntax for querying tables now looks as follows:

```
SELECT
    <Column name> (AS Alias)
    ...
FROM <Table name> (AS Alias)
```

We have now looked at the very basics of querying in SQL Server 2012. In the next chapter we are going to look at how we can limit our results.

6. Eliminating duplicates; DISTINCT

Sometimes queries can return duplicate rows. Many products have the same color. If we would select only the color from the products table we would get the color of each individual product. This looks as follows:

| | Color |
|----|--------|
| 1 | NULL |
| 2 | NULL |
| 3 | NULL |
| 4 | NULL |
| 5 | NULL |
| 6 | Black |
| 7 | Black |
| 8 | Black |
| 9 | Silver |
| 10 | Silver |
| 11 | Black |
| 12 | NULL |

Surely this is not what we intended. This is also not relational. A relation is a collection of unique and unordered values, remember? Luckily it is fairly easy to remove duplicate results using the `DISTINCT` keyword. The `DISTINCT` keyword is placed right after `SELECT`. The following query demonstrates this.

| | Color |
|----|--------------|
| 1 | NULL |
| 2 | Black |
| 3 | Blue |
| 4 | Grey |
| 5 | Multi |
| 6 | Red |
| 7 | Silver |
| 8 | Silver/Black |
| 9 | White |
| 10 | Yellow |

All products have nine colors in total (`NULL` is not a color, this is discussed in the next section).

7. Filtering data; the `WHERE` clause

Of course you will want to write more advanced queries. You might want to select all the products of a specific color, products that are more expensive than \$1000 or products that do not have a price at all. This is called *filtering* data.

Filtering in SQL Server can be done in three ways, using the `ON`, `WHERE` and `HAVING` clauses. In this chapter I will discuss the `WHERE` clause. The `ON` and `HAVING` clauses will be discussed later in this article.

Filtering in SQL is based on [predicates](#). A predicate is another mathematical term meaning a function that returns `TRUE` or `FALSE` (also called `Boolean`). For example, imagine a function that checks if a person is older than 18. Such a function would return true or false, someone is either older than 18 or he is not.

Let us look at a simple example in SQL. We want a list of all products that are more expensive than \$1000. We can do this using the `WHERE` clause.

```
SELECT *
FROM Production.Product
WHERE ListPrice > 1000
```

Running this query returns 86 of the 504 products in the `Production.Products` table. Notice how you can use the following signs for comparison:

```
=      = greater than
<      = less than
>=     = greater than or equal to
<=     = less than or equal to
<>     = not equal to (equal to != in C-based languages)
=      = equal to (equal to == in C-based languages)
```

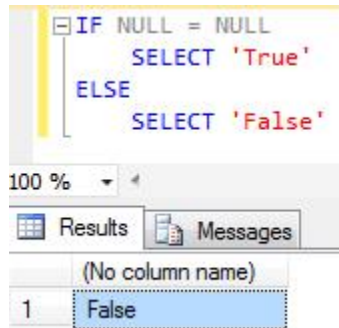
You are probably already used to this kind of syntax. SQL supports it too (among others). In this example `ListPrice > 1000` is the predicate. When using the `WHERE` clause SQL discards all results where the predicate returns `FALSE` (in this case the predicate returns `FALSE` when a products `ListPrice` is not greater than 1000).

There is a catch however. Consider the following query:

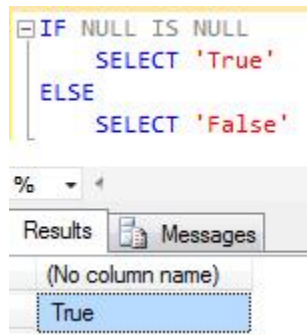
```
SELECT *
FROM Production.Product
WHERE Weight >= 0
```

We have 504 products in the `Production.Product` table and there are no products with a negative weight. Still this query only returns 205 results. What happened to 399 products!? If you take a look at the product table you might notice that many cells have the value `NULL`. This is where stuff gets tricky. `NULL` is actually not a value. It is an indication that the attribute of this tuple (or column of this

row) has no value. So in the above query how are we going to compare `NULL` (or no value) to 0? We might argue that in this case `NULL` is equal to 0, but SQL Server simply treats `NULL` as not equal to anything else. `NULL` is not even equal to `NULL`. Although the following syntax is not covered in this article it should be pretty straightforward.



Luckily not all is lost and we can use the word `IS` to compare values with `NULL`.



So I now have to correct myself. Earlier I said a predicate is a function that returns `TRUE` or `FALSE`. When working with `NULL`s it can return an additional value, `UNKNOWN`. This is called *Three-Valued Logic*. SQL discards results where a predicate in the `WHERE` clause returns `FALSE` or `UNKNOWN`.

The following query returns all products where `Weight` has no value (some might say "where the value is `NULL`" although that is not completely correct).

```
SELECT *
FROM Production.Product
WHERE Weight IS NULL
```

It is possible to filter data by columns that are not in your `SELECT` list. Consider the following query which is completely valid and also returns a correct result (that is the result we would expect it to return).

```
SELECT ProductNumber, Name
```

```
FROM Production.Product
WHERE ListPrice > 1000
```

7.1 The NOT keyword

To negate a predicate you can use the `NOT` keyword. For example, say you want products that are not red. You now have two options, using `<>` for comparison or using `=` combined with the `NOT` keyword.

```
SELECT *
FROM Production.Product
WHERE Color <> 'Red'
```

Or:

```
SELECT *
FROM Production.Product
WHERE NOT Color = 'Red'
```

Both queries return exactly the same results.

You can also use `NOT` in an `IS NULL` statement as `IS NOT NULL`. So the following queries return the same results

```
SELECT *
FROM Production.Product
WHERE Weight IS NOT NULL
```

Returns the same result as:

```
SELECT *
FROM Production.Product
WHERE NOT Weight IS NULL
```

The following query shows perfectly valid syntax, but you should beware of such queries. It returns all rows where `weight` has no value, but the double negative makes it confusing.

```
SELECT *
FROM Production.Product
WHERE NOT Weight IS NOT NULL
```

7.2 Combining predicates

We can combine predicates to create more advanced filters. We can do this using the `AND` or `OR` keywords. The following query returns all products where the `ListPrice` is greater than 1000, but only if it also has the color red.

```
SELECT *
FROM Production.Product
WHERE ListPrice > 1000
      AND Color = 'Red'
```

Try it and see the results for yourself. We know that 86 products are more expensive than \$1000. This query returns 20 rows, so 20 products are more expensive than \$1000 and have a red color.

Next to using the `AND` keyword we can also chain predicates by using `OR`. The following query returns all products that are more expensive than \$1000 or have a red color. This means that products that are \$1000 or less expensive are also returned in the result, but only if they are red.

```
SELECT *
FROM Production.Product
WHERE ListPrice > 1000
      OR Color = 'Red'
```

Now look at the following query. Will it return black and red products that are more expensive than \$1000 or will it return red products that are more expensive than \$1000 and also any priced black products?

```
SELECT *
FROM Production.Product
WHERE ListPrice > 1000
      AND Color = 'Red'
      OR Color = 'Black'
```

When you run this query you can see that it is the second, the query returns red products that are more expensive than \$1000 and also any priced black products. The `AND` keyword takes precedence over the `OR` keyword. Because this is not always clear it is considered best practice to use parenthesis, even when the query already does what you want. Parenthesis always have the highest precedence. That means that what is between parenthesis is evaluated first. So let us say we did not want red products more expensive than \$1000 and all black products. Instead we wanted all products more expensive than \$1000 that are either red or black. With parenthesis this is easily accomplished.

```
SELECT *
FROM Production.Product
WHERE ListPrice > 1000
      AND (Color = 'Red'
```

```
OR Color = 'Black')
```

Run this query and check that all products are now more expensive than \$1000.

One last thing to mention is that `NOT` takes precedence over `AND` and `OR`. So in the following example the results will contain only red products that are not more expensive than \$1000 and all black products.



```
SELECT *
FROM Production.Product
WHERE NOT ListPrice > 1000
      AND Color = 'Red'
      OR Color = 'Black'
```

Now suppose you want all products except red and black products that are more expensive than \$1000. You can use nested parenthesis and the `NOT` keyword to simply accomplish this task.



```
SELECT *
FROM Production.Product
WHERE NOT (ListPrice > 1000
      AND (Color = 'Red'
      OR Color = 'Black'))
```

7.3 Filtering strings

In the previous examples we have filtered numeric values and strings (text). This was easy enough. A numeric value does not need any special treatment and a string value should be placed between apostrophes.

So how will we filter parts of strings? For example we want all products that begin with the letter 'B'. This can be done with the `LIKE` operator. When using the `LIKE` operator you can use the `%` sign to indicate 'none, one or more characters'. Consider the following query that returns all products starting with a 'B'.

```
SELECT *
FROM Production.Product
WHERE Name LIKE 'B%'
```

So the Name should start with B and then have 'none, one or more characters'. So how would we query all products that end with a 'B'? We want none, one or more random characters and then a 'B'.

```
SELECT *
FROM Production.Product
WHERE Name LIKE '%B'
```


Notice that this query actually returns all products that end with 'b' or 'B'. SQL is not case sensitive by default, although you could make it case sensitive. This is not covered in this article.

And of course we can use `LIKE` to find certain words or characters in a string. The following query returns all products that are locks (at least according to their names).

```
SELECT *
FROM Production.Product
WHERE Name LIKE '%Lock%'
```

And the following query returns all gear for road bicycles that are black.

```
SELECT *
FROM Production.Product
WHERE Name LIKE '%Road%Black%'
```

The `%` sign is called a wildcard. The following wildcards can be used in the `LIKE` operator:

```
☐
%           = none, one or more characters
_           = a single character
[<character list>] = a single character from the list
[<character range>] = a single character from the range
[^<character list or range>] = a single character that is not in the list
or range
```

Here are some examples of all wildcards. Notice that the first and second example return the same results.

```
☐
-- First character is anything
-- Second character is an L
-- Third character is a space
SELECT *
FROM Production.Product
WHERE Name LIKE '_L %'

-- First character is an M, H or L
-- Second character is an L
SELECT *
FROM Production.Product
WHERE Name LIKE '[MHL]L%'

-- First character is an A, B or C
SELECT *
FROM Production.Product
WHERE Name LIKE '[A-C]%'

-- First character is not an A, B or C
SELECT *
FROM Production.Product
WHERE Name LIKE '[^A-C]%'
```

7.4 Filtering dates

Another point of interest is filtering dates. SQL Server supports multiple date types such as `DateTime`, `SmallDateTime` and `DateTime2`. The differences between them is outside the scope of this article. There are several ways of filtering options for dates. A date is represented by a string in a certain format. I am from the Netherlands and over here we would write a date as follows: 31-12-2013 (that is 31 december 2013). Other countries have their own format. In America the same date would be written as 12/31/2013. This is problematic because you don't want a query to return different results in different countries. For that reason it is recommended you always use the following format: 20131231, that is four digits for year, two for month and two for the day (also `yyyyMMdd`). January first would be written as 20130101. This format is not specific for any country, or *culture neutral*. So let us try it out.



```
SELECT *
FROM Production.Product
WHERE ModifiedDate = '20080311'
```

Running this query returns no results. What went wrong!? Another 'problem' with dates is that they often have times (other than 00:00:00.000). The following query would give you results (apparently all products were modified in exactly the same milisecond).



```
SELECT *
FROM Production.Product
WHERE ModifiedDate = '20080311 10:01:36.827'
```

Of course such syntax is tedious and selecting all data from a day or a range of dates would become impossible. For that reason you should search for records where the date is greater than or equal to the date you are searching for and less than the day after that date. In this example we would use the following query.



```
SELECT *
FROM Production.Product
WHERE ModifiedDate >= '20080311'
      AND ModifiedDate < '20080312'
```

A last word of caution when filtering dates. The format 2013-12-31 (or `yyyy-MM-dd`) is an ISO standard and might seem to work well, but for historic reasons this format is NOT independent of culture for the `DateTime` and `SmallDateTime` data types.

There are other ways to filter dates such as using functions, but they are not discussed until part II of this article.

7.5 The IN and BETWEEN keywords

You can also filter data using the keywords `BETWEEN` and `IN`. Both let you filter for a range of values. The difference is that `BETWEEN` uses consecutive values while `IN` lets you use any set of values.

The following query returns all products where the price is between \$100 and \$200 (including both \$100 and \$200).

```
SELECT *
FROM Production.Product
WHERE ListPrice BETWEEN 100 AND 200
```

The query is equivalent to the following:

```
SELECT *
FROM Production.Product
WHERE ListPrice >= 100
      AND ListPrice <= 200
```

The `IN` operator allows you to specify the values you want to find delimited by a comma. Let's say we want all products with product model 15, 25 or 30. The following query would return the proper results.

```
SELECT *
FROM Production.Product
WHERE ProductModelID IN (15, 25, 30)
```

The query is equivalent to the following:

```
SELECT *
FROM Production.Product
WHERE ProductModelID = 15
      OR ProductModelID = 25
      OR ProductModelID = 30
```

8. Sorting data; ORDER BY

A lot of times you will want to sort your data that means presenting your data in a specific sorting order. For example, you might want to create an overview of all orders that were made last month and view your latest orders first. As I already mentioned data in SQL Server is stored in relations, which is an unordered collection of (unique) rows. Practically this might not be the case and SQL Server will always show rows in order of the primary key (that what makes a record unique). This also means that whenever you query a table the rows are almost always returned in that order too. There is a catch though. SQL Server can never guarantee that order. That might sound strange if you query a database a thousand times and the order of the returned

rows is the same every time. Still, the order is never guaranteed and if you want a specific ordering you will have to explicitly indicate this. This can be done by using the `ORDER BY` clause.

So let us look at an example. The following query shows all products in alphabetical order.

```
SELECT *  
FROM Production.Product  
ORDER BY Name
```

Check out the results and you might notice that this is (probably) the first time a result set is not ordered by `ProductID`. Since we did not indicate an order direction SQL sorts the data in ascending order. You can explicitly indicate ascending order by using `ASC`, but this is not required. It is also possible to order data in descending order. In this case you do have to use the direction order explicitly. You can use the `DESC` keyword to indicate descending ordering. The following query returns all products from Z to A.

```
SELECT *  
FROM Production.Product  
ORDER BY Name DESC
```

It is also possible to order by multiple columns. This is of course only useful when the first column you order by has multiple rows with the same value. If each value in the first column would be unique (such as `Name`) then it is not possible to order the following columns within that group. The following query returns all products ordered by color name and then by price descending. The first color in the alphabet we have is `Black`. So the most expensive black product is returned first, after that the second most expensive black product etc. After the least expensive black product comes the most expensive `Blue` product etc.

```
SELECT *  
FROM Production.Product  
ORDER BY Color, ListPrice DESC
```

When running this query you might notice something. It is not `Black` that is the color with the 'lowest' value. It is actually `NULL` (or no value). That `NULL` certainly makes things hard, huh? Luckily this is pretty obvious, even if you did not expect it or disagree with it. When ordering 'no value' comes before any value.

You might remember that the `ORDER BY` list can contain column aliases too. This has to do with the order in which SQL Server internally processes queries. The `ORDER BY` is actually the only statement that is internally handled after the `SELECT` (even though `SELECT` appears as the very first keyword in a query) making it possible for `ORDER BY` to use aliases. I will show an example where the `ORDER BY` takes a column alias as parameter.

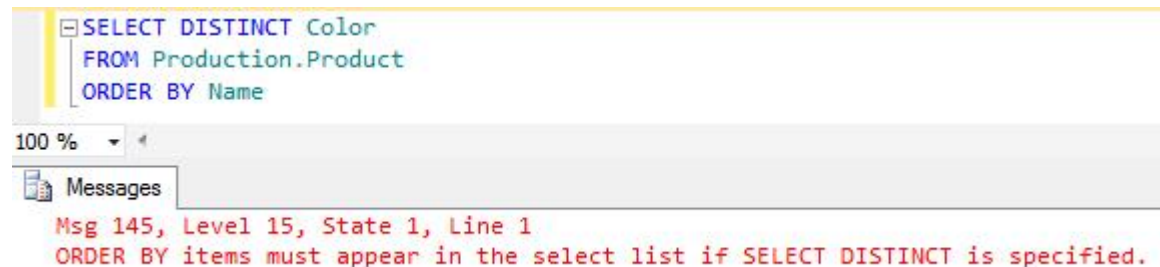
```
SELECT
```

```

        Name,
        ProductNumber AS Number
FROM Production.Product
ORDER BY Number

```

Just like with a `WHERE` clause it is possible to order data by a column that is not in your `SELECT` list. There is one exception to this rule however. When using `DISTINCT` in your query you cannot order by columns that are not in the `SELECT` list. The reason is that when duplicates are removed the result rows do not necessarily map to the source rows in a one-to-one manner. The following query shows this.



This query would return ten rows (as we saw earlier), but the table contains 504 unique names. SQL Server can not order ten rows based on 504 values. Based on what product names should it order the colors black and yellow? That might be a problem...

It is also possible to order by the (1-based) index of the column, but this is considered bad practice. The following example shows this. In this query the result is ordered by `Name` (because it is the first column in the `SELECT` list).

```

SELECT Name, ProductNumber
FROM Production.Product
ORDER BY 1

```

Now imagine that someone would change the order in the `SELECT` list or would add columns before `Name`? Suddenly `Name` is not the first column in the `SELECT` list anymore, but the `ORDER BY` still orders by the first column. This can have unexpected results. When using column names in the `ORDER BY` this would never be a problem.

If you want to specify a `WHERE` clause and an `ORDER BY` the `ORDER BY` comes last in the query.

```

SELECT *
FROM Production.Product
WHERE Color = 'Red'
ORDER BY Name

```

9. Further limiting results; TOP and OFFSET-FETCH

9.1 TOP

You might not always want to return all rows from a query. For example because returning all rows would return so many rows that your application would be unresponsive for hours while fetching the data. Or maybe you are only interested in the top ten most expensive products or perhaps the top ten percent. Of course this is also possible in SQL using the `TOP` statement, which can be used as follows.



```
SELECT TOP 10 *  
FROM Production.Product
```

This query returns the first ten products. This makes little sense though. Remember that SQL Server guarantees no order in its results? Simply selecting a top ten theoretically means we are selecting any ten random products. More useful is to combine `TOP` with an `ORDER BY` clause. The next query gets the top ten most expensive products.



```
SELECT TOP 10 *  
FROM Production.Product  
ORDER BY ListPrice DESC
```

When you run this query you might notice a lot of duplicate prices. We are selecting a top ten, but actually the next three results (which are not returned) are just as expensive as the last row that is returned. And even if we returned those three rows we would still only have our three highest prices (which might or might not be what you want). For this example we really do want the most expensive products even if they only make up for the three highest prices (there are methods to actually get the top ten highest prices, but that is not discussed here). We do want to return the three rows that have the same price as the last row returned though. This can be done by adding `WITH TIES` to the query.



```
SELECT TOP 10 WITH TIES *  
FROM Production.Product  
ORDER BY ListPrice DESC
```

We can also use `TOP` to select a certain percentage of our result. This can be done by using the `PERCENT` keyword.



```
SELECT TOP 10 PERCENT *  
FROM Production.Product  
ORDER BY ListPrice DESC
```

This query returns 51 results. The Product table actually has 504 rows. Ten percent of 504 is actually 50.4. We would not be satisfied if SQL Server returned 0.4 row though, so instead it returns the ceiling (always rounds up). 50.4 thus becomes 51. `WITH TIES` can be added to the query again, which would return 55 rows instead of 51.

One thing I should mention is that the number in `TOP` can be parameterized. This is outside the scope of this article, but I will show an example nonetheless.



```
DECLARE @Top AS INT
SET @Top = 10

SELECT TOP (@Top) WITH TIES *
FROM Production.Product
ORDER BY ListPrice DESC
```

9.2 OFFSET-FETCH

What if you wanted to skip a couple of rows instead of selecting a top? This is a common scenario for paging. For this purpose you can use the `OFFSET-FETCH` clause. Let us skip the top ten most expensive products and select the next 25 most expensive. With `OFFSET` we can specify how many rows we want to skip and with `FETCH` we can specify how many rows we do want to return. The following query illustrates this.



```
SELECT *
FROM Production.Product
ORDER BY ListPrice DESC
OFFSET 10 ROWS FETCH NEXT 25 ROWS ONLY
```

You might notice a few things. First of all I started off with including an `ORDER BY` right away. The `ORDER BY` is actually mandatory when using `OFFSET-FETCH`. Second, the `OFFSET-FETCH` clause actually consists of two parts, the `OFFSET` and the `FETCH` part.

Let us take a closer look at the two parts first. The `OFFSET` part is mandatory if you want a `FETCH` part. The `FETCH` part is optional however. Not including the `FETCH` part makes you skip a number of rows and return all the other rows. You might want to skip 100 rows and return all the remaining rows.



```
SELECT *
FROM Production.Product
ORDER BY ListPrice DESC
OFFSET 100 ROWS
```

You can also specify an `OFFSET` of 0. This would select all rows.

Using an `OFFSET` of 0 and specifying a `FETCH` is equivalent to a `TOP` statement. The following query selects the top 25 rows using `OFFSET-FETCH`.



```
SELECT *
FROM Production.Product
ORDER BY ListPrice DESC
OFFSET 0 ROWS FETCH FIRST 25 ROWS ONLY
```

You might notice I used the keyword `FIRST` instead of `NEXT`. They are completely interchangeable, but since we are here selecting a top 25 it might be more correct to use `FIRST` instead of `NEXT`. The `ROWS` keyword can be replaced by `ROW`, which would look better if you have an `OFFSET` or `FETCH` of one.

When using `OFFSET-FETCH` an `ORDER BY` is required. Since order is not guaranteed in SQL Server it would be logical to use an `ORDER BY`, but maybe you really do want to skip a few random rows after which you select a few other random rows. In this case you can use the following `ORDER BY` clause.

```
SELECT *
FROM Production.Product
ORDER BY (SELECT NULL)
OFFSET 1 ROW FETCH FIRST 25 ROWS ONLY
```

Since `OFFSET-FETCH` is a SQL standard and `TOP` is not it is recommended to use `OFFSET-FETCH` whenever possible. There are no `WITH TIES` or `PERCENT` equivalents for `OFFSET-FETCH` though. So if you need that functionality `TOP` would be your only choice.

10. Aggregating data; GROUP BY and HAVING

Quite often you will want to group data. You might want to know how much money you made from all orders, from all orders in a specific year or from each customer. The following query simply counts the rows in the `Sales.SalesOrderHeader` table. The query returns a single value only.

```
SELECT COUNT(*)
FROM Sales.SalesOrderHeader
```

`COUNT` is known as an aggregate function. `COUNT(*)` specifically means count rows. What this query did was group all rows into a single group and count the rows for that group. Now suppose you want to know how much orders each customer has placed. We want to create a single group for every customer and then count the number of rows in that group. This is accomplished by using the `GROUP BY` clause. The following query shows this scenario.

```
SELECT CustomerID, COUNT(*)
FROM Sales.SalesOrderHeader
GROUP BY CustomerID
```

In this query the `GROUP BY` clause is mandatory. Because `COUNT` can only be applied to groups and `CustomerID` by itself is not a group you either need to remove `CustomerID` so a single group is assumed or explicitly create a group per `CustomerID` by using the `GROUP BY` clause.

Let us slightly modify that last query so it shows us how many sales persons each customer has. This can be done by replacing the `*` in `COUNT(*)` with the name of the column you want to count.

```
SELECT CustomerID, COUNT(SalesPersonID)
FROM Sales.SalesOrderHeader
GROUP BY CustomerID
```

Take a little time to look at the result and make sure the result is correct. We see a lot of customers with no sales persons. `COUNT` ignores `NULLS` and simply does not count them. So was the result correct? No, it was not! For example, the customer with ID 29484 only has one distinct sales person, yet our results indicate there were seven! Actually `COUNT` does not count the number of unique values, it simply counts values. So the query above gives us the number of orders per customer where sales persons has a value. Luckily we can easily fix this.

```
SELECT CustomerID, COUNT(DISTINCT SalesPersonID)
FROM Sales.SalesOrderHeader
GROUP BY CustomerID
```

By adding `DISTINCT` inside the `COUNT` function we get a count of all unique values (still excluding `NULLS`). We can also make groups based on multiple columns. For example we want to know the number of sales persons per customer per ship to address. In that example we could see that a single customer uses one or more shipping addresses and the number of sales persons that serve a single shipping address. In this database there is only one shipping address per customer which you can check by running the following query.

```
SELECT
    CustomerID,
    ShipToAddressID,
    COUNT(DISTINCT SalesPersonID)
FROM Sales.SalesOrderHeader
GROUP BY CustomerID, ShipToAddressID
```

We still see a lot of customers that do not have a sales person. For our purposes we are not interested in those. We can still use a `WHERE` clause to order out the `NULL` values.

```
SELECT
    CustomerID,
    COUNT(DISTINCT SalesPersonID)
FROM Sales.SalesOrderHeader
WHERE SalesPersonID IS NOT NULL
GROUP BY CustomerID
```

Notice that the `GROUP BY` clause comes after the `WHERE` clause. So in this example the `WHERE` clause filters out all rows where `SalesPersonID` has no value and then the remaining rows are grouped. But what if we wanted to filter groups? Let us say we want only customers that have

more than one sales person. The `WHERE` clause filters on a row basis and not on a group basis. The following query is not valid syntax.

```
SELECT
    CustomerID,
    ShipToAddressID,
    COUNT(DISTINCT SalesPersonID)
FROM Sales.SalesOrderHeader
WHERE COUNT(DISTINCT SalesPersonID) > 1
GROUP BY CustomerID, ShipToAddressID
```

100 %

Messages

Msg 147, Level 15, State 1, Line 6
An aggregate may not appear in the WHERE clause unless it is in a subquery contained in a HAVING clause.

The error message already gives us the solution. We need to use the `HAVING` clause in order to filter groups. The following query solves our problem.

```
SELECT
    CustomerID,
    ShipToAddressID,
    COUNT(DISTINCT SalesPersonID)
FROM Sales.SalesOrderHeader
GROUP BY CustomerID, ShipToAddressID
HAVING COUNT(DISTINCT SalesPersonID) > 1
```

`HAVING` does largely the same as `WHERE`, except that it filters groups and as such can have aggregate functions. You can use a `WHERE` and `HAVING` clause in the same query. Make sure you understand the difference between them though.

There are other grouping functions as well. The following table lists the most common ones. All can be combined with `DISTINCT` to use only unique values. Like `COUNT` the other functions also ignore `NULLS`.

| | |
|----------------------------|---|
| <code>COUNT(*)</code> | = counts the number of rows |
| <code>COUNT(column)</code> | = counts the number of values in the column |
| <code>AVG(column)</code> | = gives the average value of the values in the column |
| <code>MAX(column)</code> | = gives the maximum value in the column |
| <code>MIN(column)</code> | = gives the minimum value in the column |
| <code>SUM(column)</code> | = adds all values in the column |

The following query shows how the functions can be used.

```
SELECT
    CustomerID,
    COUNT(*) AS NoOfOrders,
```

```

        AVG(SubTotal)      AS AverageSubTotal,
        MAX(SubTotal)      AS MaxSubTotal,
        MIN(SubTotal)      AS MinSubTotal,
        SUM(SubTotal)      AS TotalSubTotal
FROM Sales.SalesOrderHeader
GROUP BY CustomerID

```

In this last example I aliased all of the aggregated columns. You might have noticed, but SQL Server cannot give names to aggregated columns. It is considered best practice to alias these columns. When we order results by aggregated columns we now have two choices. We can either order by the aggregate function, which makes our query rather cluttered or we can order by alias. The following queries show the difference.

```

SELECT
    CustomerID,
    COUNT(*)
FROM Sales.SalesOrderHeader
GROUP BY CustomerID
ORDER BY COUNT(*)

SELECT
    CustomerID,
    COUNT(*) AS NoOfOrders
FROM Sales.SalesOrderHeader
GROUP BY CustomerID
ORDER BY NoOfOrders

```

Just like with `DISTINCT` the `ORDER BY` clause can now not contain columns that are not present in the `SELECT` list.

11. Selecting from multiple tables; using JOINS

You might have noticed that in the last part we worked with `CustomerID`'s and `SalesPersonID`'s without knowing what these ID's mean. The `CustomerID` in the `SalesOrderHeader` table has a corresponding row in the `Customer` table, enforced by a [Foreign Key Constraint](#), where we can get more information about the customer. Take for example the first order in the `SalesOrderHeader` table.

```

SELECT TOP 1
    SalesOrderID,
    SalesOrderNumber,
    CustomerID,
    SubTotal
FROM Sales.SalesOrderHeader
ORDER BY SalesOrderID

```

If we wanted to know the name of this customer we could look it up in a separate query and get the customer with that ID. We do not want that however. We need the name of the customer in

the same result set as the order information. We can select from multiple tables using two methods.

The first method to select from multiple tables, which is not the recommended one, is to add more tables to the `FROM` clause. The following example shows how this can be done. Do NOT run the following query.

```
SELECT *
FROM Sales.SalesOrderHeader, Sales.Customer
```

This query returns the [*cartesian product*](#) of the `SalesOrderHeader` and the `Customer` table. That means that every order is combined with every customer. The query would run for hours on my machine and should return 623636300 rows (31456 orders * 19820 customers). Clearly this is not what we want. We only want to return a subset of the cartesian product. So which rows do we want to see? We only want to return the rows where the `CustomerID` from the `Customer` table matches the `CustomerID` from the `SalesOrderHeader` table. That is fairly simple using a `WHERE` clause. Note that if you would have orders that do not have a value for `CustomerID` this order would not be returned.

```
SELECT *
FROM Sales.SalesOrderHeader, Sales.Customer
WHERE Sales.SalesOrderHeader.CustomerID = Sales.Customer.CustomerID
```

Now that we are selecting from multiple tables we might need to use the column names including table names. After all if we only specify `CustomerID` do we mean the `CustomerID` from the `SalesOrderHeader` table or the `CustomerID` of the `Customer` table? This is where table aliasing comes in handy. You have already seen the syntax for aliasing, so this should not be a problem. Here is the query written with aliases.

```
SELECT *
FROM Sales.SalesOrderHeader AS s, Sales.Customer AS c
WHERE s.CustomerID = c.CustomerID
```

Notice how for table aliases we use short names, like a single character (for example the first letter of the table name). When using table aliases you must use the alias. Using the full table name is not allowed anymore. In some cases aliasing is mandatory like when you select from the same table twice. Unfortunately the `Customer` table does not have the `Name` we are looking for. A `Customer` has a `PersonID` which corresponds to a row in the `Person` table. This `Person` has the name we need. So we are going to need another table to select from and make sure we select only the correct person for every row in the `WHERE` clause.

```
SELECT *
FROM Sales.SalesOrderHeader AS s,
     Sales.Customer AS c,
```

```

    Person.Person      AS p
WHERE s.CustomerID = c.CustomerID
    AND c.PersonID = p.BusinessEntityID

```

So we now select all columns from the SalesOrderHeader, Customer and Person tables. As you can see our result has multiple columns with the same name like CustomerID, rowguid and ModifiedDate. So let us select only the columns we want.

```

SELECT
    s.SalesOrderID,
    s.SalesOrderNumber,
    s.CustomerID,
    p.FirstName,
    p.LastName,
    s.SubTotal
FROM Sales.SalesOrderHeader AS s,
    Sales.Customer AS c,
    Person.Person      AS p
WHERE s.CustomerID = c.CustomerID
    AND c.PersonID = p.BusinessEntityID

```

As mentioned this is not the recommended method to select from multiple tables, so I will not elaborate on this further.

11.1 CROSS JOIN

Instead we should use the JOIN operator. There are multiple kinds of JOINS. The simplest is the CROSS JOIN, which returns the cartesian product. Once again, do not run this query for it will return over 600 million records and takes hours to run (unless you add a where clause of course).

```

SELECT *
FROM Sales.SalesOrderHeader
    CROSS JOIN Sales.Customer

```

11.2 INNER JOIN

The next JOIN, and probably the one you will use most, is the INNER JOIN. Actually INNER JOIN is the most used JOIN and the INNER keyword is optional. The INNER JOIN operator uses the ON clause to match rows instead of WHERE. The following example illustrates this.

```

SELECT *
FROM Sales.SalesOrderHeader AS s
    INNER JOIN Sales.Customer AS c ON s.CustomerID = c.CustomerID

```

When using the INNER JOIN results from the main table that do not have at least one match in the join table are discarded. When the join table has multiple matches then rows from the main table are duplicated in the result. The following query illustrates that rows are discarded. The

SalesPersonID column in the SalesOrderHeader table contains NULLS. Obviously the rows where there is no value for SalesPersonID cannot be matched with a sales person from the SalesPerson table. If you run the following query you will see that these rows are not included in the result.

```
SELECT *
FROM Sales.SalesOrderHeader
      JOIN Sales.SalesPerson ON SalesPersonID = BusinessEntityID
```

The following query shows that the SalesOrderHeader rows are duplicated when joined with the SalesOrderDetail table. An order can have more than one detail and for each detail the sales order header is duplicated.

```
SELECT *
FROM Sales.SalesOrderHeader AS soh
      JOIN Sales.SalesOrderDetail AS sod ON sod.SalesOrderID = soh.SalesOrderID
ORDER BY soh.SalesOrderID
```

The ON clause can contain more advanced filters, just like the WHERE clause. We can, for example, join only on details with a discount.

```
SELECT *
FROM Sales.SalesOrderHeader AS soh
      JOIN Sales.SalesOrderDetail AS sod ON sod.SalesOrderID = soh.SalesOrderID
                                         AND sod.UnitPriceDiscount > 0
ORDER BY soh.SalesOrderID
```

An alert reader might have noticed that the sod.UnitPriceDiscount > 0 part could just as well be in the WHERE clause of the query. Indeed, in an INNER JOIN the ON and WHERE clause are interchangeable. The following query is equivalent to the query above.

```
SELECT *
FROM Sales.SalesOrderHeader AS soh
      JOIN Sales.SalesOrderDetail AS sod ON sod.SalesOrderID = soh.SalesOrderID
WHERE sod.UnitPriceDiscount > 0
ORDER BY soh.SalesOrderID
```

Any row that does not have a match with a row from the other table is discarded in the result, so it does not matter if the ON clause filters a row in the SalesOrderDetail table which is then also discarded from the SalesOrderHeader table or if the WHERE clause filters the rows after a match was found in the ON clause.

The ON clause is always mandatory when working with INNER JOINS. That does not mean you have to check for equality between two rows. The following is also perfectly valid syntax and is

equal to a `CROSS JOIN` (or cartesian product) because every row in the `SalesOrderDetail` table matches any row in the `SalesOrderHeader` table (because the `1 = 1` predicate always returns `TRUE`).



```
SELECT *
FROM Sales.SalesOrderHeader
      JOIN Sales.SalesOrderDetail ON 1 = 1
```

11.3 OUTER JOIN

We just saw that with an `INNER JOIN` rows that do not have a match in the joined table are discarded. With an `OUTER JOIN` we can preserve these results. There are three kinds of `OUTER JOINS`, one that preserves rows from the main table if no match is found, one that preserves rows from the joined table and one that preserves both. Let us join the `SalesOrderHeader` table with the `SalesPerson` table again, but this time we want to preserve sales orders that do not have a sales person.



```
SELECT *
FROM Sales.SalesOrderHeader
      LEFT OUTER JOIN Sales.SalesPerson ON SalesPersonID = BusinessEntityID
```

As you can see the `LEFT OUTER JOIN` preserves `SalesOrderHeaders` (the `LEFT` part of the `JOIN`) when no `SalesPerson` row can be found. In the result set all `SalesPerson` columns have no value for these rows.

The following query joins orders with customers. Not every customer has placed an order though, so we can choose to keep these customers in the result set and return empty orders for these customers.



```
SELECT *
FROM Sales.SalesOrderHeader AS soh
      RIGHT OUTER JOIN Sales.Customer AS c ON c.CustomerID = soh.CustomerID
ORDER BY c.CustomerID
```

As you can see the `RIGHT OUTER JOIN` preserves `Customers` (the `RIGHT` part of the `JOIN`) when no `SalesOrderHeaders` rows can be found.

The last type of `OUTER JOIN` is the `FULL OUTER JOIN` and preserves rows from both sides of the `JOIN`, so basically it is both a `LEFT OUTER JOIN` and a `RIGHT OUTER JOIN`. In the `SalesOrderHeader` table there are orders that have a `CurrencyRateID`, but there are also orders without a `CurrencyRateID`. That would at least qualify for a `LEFT OUTER JOIN`, but there are also `CurrencyRates` that do not have orders, which would also qualify a join between the tables for a `RIGHT OUTER JOIN`. In this case, if we want to keep both orders without a currency rate and currency rates without orders we can use the `FULL OUTER JOIN`.



```
SELECT *
FROM Sales.SalesOrderHeader AS soh
      FULL OUTER JOIN Sales.CurrencyRate AS c ON c.CurrencyRateID =
soh.CurrencyRateID
ORDER BY soh.SalesOrderID
```

The `OUTER` keyword is optional in all `OUTER JOINS`. So `LEFT JOIN`, `RIGHT JOIN` and `FULL JOIN` are also valid syntax.

Remember that with an `INNER JOIN` the `ON` clause and `WHERE` clause had the same filtering function? Well, that is not the case for the `OUTER JOIN`. The `OUTER JOIN` discards rows on one side of the join, but keeps them on the other side. So the `ON` clause is used for matching purposes only. When adding an extra predicate to the `ON` clause it just means the right or left side of the join will have no values if the predicate results in `FALSE` (in contrary to the `INNER JOIN` where the result was discarded as a whole). For example, the following two queries both return very different results.



```
SELECT *
FROM Sales.SalesOrderHeader AS soh
      FULL JOIN Sales.CurrencyRate AS c ON c.CurrencyRateID =
soh.CurrencyRateID
                                AND c.AverageRate > 1
ORDER BY c.AverageRate
```

```
SELECT *
FROM Sales.SalesOrderHeader AS soh
      FULL JOIN Sales.CurrencyRate AS c ON c.CurrencyRateID =
soh.CurrencyRateID
WHERE c.AverageRate > 1
ORDER BY c.AverageRate
```

The first query returns all orders, but discards the currency rate for an order if it has an average rate that is higher than one. The second query only returns orders that have an average rate that is higher than one.

11.4 Self JOIN

A table can be joined with the same table. Examples of why you would want to do this might be because an `Employee` has a `ManagerID`, which corresponds to a manager who is also an `Employee`. Unfortunately I could not find an example in the `AdventureWorks2012` database where a self `JOIN` would make sense. That does not mean I cannot show you. The next example selects from the `Person.Person` table and joins with the `Person.Person` table, selecting the `Person` with the current `Persons ID + 1`. Of course this is not really a valid business case and the join fails when `ID`'s are not consecutive, but it shows how you can join a table with itself.



```
SELECT
      pl.BusinessEntityID      AS CurrentID,
```



```

    p1.Title           AS CurrentTitle,
    p1.FirstName       AS CurrentFirstName,
    p1.LastName        AS CurrentLastName,
    p2.BusinessEntityID AS NextID,
    p2.Title           AS NextTitle,
    p2.FirstName       AS NextFirstName,
    p2.LastName        AS NextLastName
FROM Person.Person AS p1
    LEFT JOIN Person.Person AS p2 ON p2.BusinessEntityID =
p1.BusinessEntityID + 1
ORDER BY CurrentID, CurrentFirstName, CurrentLastName

```

11.5 Multiple JOINS

In previous examples we selected Customer data based on the CustomerID in the SalesOrderHeader and then we selected Person data using the PersonID in the Customer table. We can also use JOINS for this task. In fact, it is possible to use all kinds of joins in a single query. Beware that one JOIN may influence the other. In the following example you might think that only orders that ultimately have a customer with a person will be selected (because the JOIN on Person is an INNER JOIN). But because the JOIN with customer is a RIGHT JOIN and the person is joined on customer not every person has an order.

```

SELECT
    s.SalesOrderID,
    s.SalesOrderNumber,
    c.CustomerID,
    p.BusinessEntityID,
    p.FirstName,
    p.LastName,
    s.SubTotal
FROM Sales.SalesOrderHeader AS s
    RIGHT JOIN Sales.Customer AS c ON c.CustomerID = s.CustomerID
    JOIN Person.Person AS p ON p.BusinessEntityID = c.CustomerID
ORDER BY p.BusinessEntityID

```

12. Multiple groups; GROUPING SETS

12.1 GROUPING SETS

We have taken a little break from grouping operators to explore the JOIN options that are available to us. There is more to grouping than we have so far discussed though. A query can have multiple groupings in a single query. We might want to know the number of orders in total (or avg, max, min or sum), the number of orders per customer, per sales person and per customer and sales person. This can be achieved with the GROUPING SETS clause. The following query shows an example.

```

SELECT
    CustomerID,

```

```

        SalesPersonID,
        COUNT(*) AS NoOfOrders
FROM Sales.SalesOrderHeader
GROUP BY GROUPING SETS
(
    (CustomerID
        ),
    (SalesPersonID
        ),
    (CustomerID, SalesPersonID
        ),
    (
        )
)
ORDER BY SalesPersonID,
        CustomerID

```

The results of this query are a little difficult to interpret. The first results look as follows:

| | CustomerID | SalesPersonID | NoOfOrders |
|---|------------|---------------|------------|
| 1 | NULL | NULL | 31465 |
| 2 | NULL | NULL | 27659 |
| 3 | 11000 | NULL | 3 |
| 4 | 11000 | NULL | 3 |
| 5 | 11001 | NULL | 3 |
| 6 | 11001 | NULL | 3 |
| 7 | 11002 | NULL | 3 |
| 8 | 11002 | NULL | 3 |

The first result we see is the total amount of orders. The second row looks like it has the same data except the NoOfOrders is different from the first. This is actually the number of orders that do not have a sales person (SalesPersonID is NULL). The rows after that look exactly the same. One represents a sales order for a specific customer and a specific SalesPersonID (which just happens to be NULL) and the other represents a sales order for a specific customer regardless of SalesPersonID.

When we look at later results where `SalesPersonID` actually has a value this may become clearer.

| | CustomerID | SalesPersonID | NoOfOrders |
|-------|------------|---------------|------------|
| 38270 | 30071 | 284 | 4 |
| 38271 | 30105 | 284 | 7 |
| 38272 | 30112 | 284 | 7 |
| 38273 | NULL | 285 | 16 |
| 38274 | 29488 | 285 | 1 |
| 38275 | 29512 | 285 | 3 |
| 38276 | 29556 | 285 | 1 |
| 38277 | 29602 | 285 | 1 |
| 38278 | 29652 | 285 | 1 |
| 38279 | 29657 | 285 | 1 |
| 38280 | 29665 | 285 | 1 |
| 38281 | 29700 | 285 | 1 |
| 38282 | 29754 | 285 | 1 |
| 38283 | 29821 | 285 | 1 |
| 38284 | 29823 | 285 | 1 |
| 38285 | 29829 | 285 | 1 |
| 38286 | 29972 | 285 | 1 |
| 38287 | 30015 | 285 | 1 |
| 38288 | NULL | 286 | 109 |
| 38289 | 29488 | 286 | 3 |
| 38290 | 29512 | 286 | 1 |
| 38291 | 29516 | 286 | 4 |
| 38292 | 29529 | 286 | 4 |
| 38293 | 29537 | 286 | 3 |
| 38294 | 29538 | 286 | 4 |
| 38295 | 29547 | 286 | 3 |
| 38296 | 29556 | 286 | 3 |
| 38297 | 29595 | 286 | 4 |
| 38298 | 29602 | 286 | 3 |
| 38299 | 29628 | 286 | 4 |
| 38300 | 29648 | 286 | 4 |

Here we see that the first three rows contain the number of orders for a specific customer with a specific sales person. The fourth row has no customer and thus shows the number of orders for that sales person regardless of customer. The rows that follow break down his sales numbers per customer until another row follows where customer is empty and the next sales person starts.

How can we know if `SalesPersonID` is part of a group if it has no value? You might be tempted to remove orders that have no sales person from our query, but beware! The following query gives a much better overview of what each sales person has sold, but the totals for many

customers and the grand total are not correct anymore! After all, orders without a sales person are not counted for customers and for the grand total.

```
SELECT
    CustomerID,
    SalesPersonID,
    COUNT(*) AS NoOfOrders
FROM Sales.SalesOrderHeader
WHERE SalesPersonID IS NOT NULL
GROUP BY GROUPING SETS
(
    (CustomerID),
    (SalesPersonID),
    (CustomerID, SalesPersonID),
    ()
)
ORDER BY SalesPersonID,
    CustomerID
```

Check that this is actually not the result you expected. The total number of orders has suddenly shrunk to only 3806, but this is only the total number of orders that have a sales person! This might or might not be what you want. You have been warned. I will get back to this problem later in this section.

Windowing functions; the OVER clause

Sometimes you want to show the results of grouping data, like the number of orders, without actually grouping the data in your result. This is possible using [window functions](#). A window function works like the GROUP BY clause we have seen earlier, except it calculates the result and returns a single value, making it possible to use these values in your query without grouping it. The 'window' is defined by your result set. So the result set is passed into a window function, the window function does the calculation (by grouping and ordering the window) and returns a single value for each row in your actual result set.

Window functions are internally executed after the WHERE and HAVING clauses. That means you cannot use them in filters or other clauses except the ORDER BY. Common Table Expressions or CTE's offer the solution, but they are not discussed until part II of this article.

13.1 Aggregate functions

We can use the aggregate functions COUNT, AVG, MAX, MIN and SUM in a window and apply them to each row in your result set by using the OVER clause. The following query shows an example of each.

```
SELECT
    SalesOrderID,
```

```

SalesOrderNumber,
COUNT(*) OVER() AS NoOfOrders,
COUNT(SalesPersonID) OVER() AS OrdersWithSalesPerson,
AVG(SubTotal) OVER() AS AvgSubTotal,
MAX(SubTotal) OVER() AS MaxSubTotal,
MIN(SubTotal) OVER() AS MinSubTotal,
SUM(SubTotal) OVER() AS TotalSubTotal
FROM Sales.SalesOrderHeader

```

As you can see we can apply aggregating functions without grouping SalesOrderID and SalesOrderNumber. Each row now simply shows the number of orders, the number of orders that have a sales person and the average, maximum, minimum and total subtotal.

That is pretty useful, but often you want to show these results per group. For example you need to show the total subtotal for the customer in the row. This can be done using PARTITION BY in the OVER clause. The following query shows how to use this.

```

SELECT
    SalesOrderID,
    SalesOrderNumber,
    CustomerID,
    SUM(SubTotal) OVER(PARTITION BY CustomerID) AS TotalSubTotalPerCustomer,
    SUM(SubTotal) OVER() AS TotalSubTotal
FROM Sales.SalesOrderHeader

```

When looking at the result you can see the TotalSubTotal is still the same for every row, but the TotalSubTotalPerCustomer is the same for each row with the same customer and different for each row with another customer. Your result set, or window, is grouped by CustomerID (defined in PARTITION BY) and a single result is returned for each row.

And you can specify multiple columns to partition by. For example you can show the total subtotal per customer per sales person.

```

SELECT
    SalesOrderID,
    SalesOrderNumber,
    CustomerID,
    SalesPersonID,
    SUM(SubTotal) OVER(PARTITION BY CustomerID, SalesPersonID)
        AS TotalSubTotalPerCustomerPerSalesPerson,
    SUM(SubTotal) OVER() AS TotalSubTotal
FROM Sales.SalesOrderHeader
ORDER BY CustomerID, SalesPersonID

```

Ranking functions

Ranking functions can rank rows in a window based on a specified ordering. There are four ranking functions within SQL Server; ROW_NUMBER, RANK, DENSE_RANK and NTILE. An ORDER BY clause is mandatory, a PARTITION is not (if it is not specified the entire window is considered one group). The following example shows how to use the functions.

```

SELECT
    SalesOrderID,
    SalesOrderNumber,
    CustomerID,
    ROW_NUMBER() OVER(ORDER BY CustomerID) AS RowNumber,
    RANK() OVER(ORDER BY CustomerID) AS [Rank],
    DENSE_RANK() OVER(ORDER BY CustomerID) AS DenseRank,
    NTILE(5000) OVER(ORDER BY CustomerID) AS NTile5000
FROM Sales.SalesOrderHeader
ORDER BY CustomerID

```

ROW_NUMBER speaks for itself. It simply returns the number of the current row.

RANK and DENSE_RANK assign a number to each row with a unique order value. That means that rows with the same order value (in this case CustomerID) get the same number. The difference between RANK and DENSE_RANK is that RANK assigns the current rank number plus the number of rows that have the same order value to the next row while DENSE_RANK always assigns the previous rank number plus one, no matter how many rows had the same order value.

NTILE partitions, or tiles, the rows in groups of equal size. In this case the returned result had 31465 rows and we requested 5000 tiles of equal size. 31456 divided by 5000 equals 6 with a remainder of 1456. That means the NTILE value is increased after every six rows. Because there is a remainder of 1456 the first 1456 tiles get an additional row.

The following result shows the ranking functions output.

| | SalesOrderID | SalesOrderNumber | CustomerID | RowNumber | Rank | DenseRank | NTile5000 |
|----|--------------|------------------|------------|-----------|------|-----------|-----------|
| 1 | 43793 | SO43793 | 11000 | 1 | 1 | 1 | 1 |
| 2 | 51522 | SO51522 | 11000 | 2 | 1 | 1 | 1 |
| 3 | 57418 | SO57418 | 11000 | 3 | 1 | 1 | 1 |
| 4 | 43767 | SO43767 | 11001 | 4 | 4 | 2 | 1 |
| 5 | 51493 | SO51493 | 11001 | 5 | 4 | 2 | 1 |
| 6 | 72773 | SO72773 | 11001 | 6 | 4 | 2 | 1 |
| 7 | 43736 | SO43736 | 11002 | 7 | 7 | 3 | 1 |
| 8 | 51238 | SO51238 | 11002 | 8 | 7 | 3 | 2 |
| 9 | 53237 | SO53237 | 11002 | 9 | 7 | 3 | 2 |
| 10 | 43701 | SO43701 | 11003 | 10 | 10 | 4 | 2 |
| 11 | 51315 | SO51315 | 11003 | 11 | 10 | 4 | 2 |
| 12 | 57783 | SO57783 | 11003 | 12 | 10 | 4 | 2 |
| 13 | 43810 | SO43810 | 11004 | 13 | 13 | 5 | 2 |
| 14 | 51595 | SO51595 | 11004 | 14 | 13 | 5 | 2 |
| 15 | 57293 | SO57293 | 11004 | 15 | 13 | 5 | 3 |
| 16 | 43704 | SO43704 | 11005 | 16 | 16 | 6 | 3 |

In this case the ordering by `CustomerID` is not unique. As such the order within this window is not guaranteed by SQL Server and neither is your row number. That means that the record that is now row number two might be row number one or three the next time you run this query. If you want your row numbering to be guaranteed repeatable you should add another order by column which makes the ordering unique, for example `SalesOrderID`. This goes for all your windowing functions including aggregating functions, framing and offset functions.



```
SELECT
    SalesOrderID,
    SalesOrderNumber,
    CustomerID,
    ROW_NUMBER() OVER(ORDER BY CustomerID, SalesOrderID) AS RowNumber,
    RANK() OVER(ORDER BY CustomerID) AS [Rank],
    DENSE_RANK() OVER(ORDER BY CustomerID) AS DenseRank,
    NTILE(5000) OVER(ORDER BY CustomerID, SalesOrderID) AS NTile5000
FROM Sales.SalesOrderHeader
ORDER BY CustomerID
```

Notice that in this last example I have not added the `SalesOrderID` ordering to the `RANK` functions. These return the same value for rows with the same `CustomerID` no matter their ordering within their group. Adding an extra ordering column will actually change the meaning of the `RANK` functions!

Isolation levels

Isolation levels in SQL Server control the way locking works between transactions.

SQL Server 2008 supports the following isolation levels

- Read Uncommitted
- Read Committed (The default)
- Repeatable Read
- Serializable
- Snapshot

Before you run through each of these in detail you may want to create a new database to run the examples, run the following script on the new database to create the sample data. Note : You'll also want to drop the `IsolationTests` table and re-run this script before each example to reset the data.

```
CREATE TABLE IsolationTests
```

```
(
```

```
    Id INT IDENTITY,
```

```
    Col1 INT,
```

```
    Col2 INT,
```

```
Col3 INTupdate te
)
```

```
INSERT INTO IsolationTests(Col1,Col2,Col3)
SELECT 1,2,3
UNION ALL SELECT 1,2,3
UNION ALL SELECT 1,2,3
UNION ALL SELECT 1,2,3
UNION ALL SELECT 1,2,3
UNION ALL SELECT 1,2,3
UNION ALL SELECT 1,2,3
```

Also before we go any further it is important to understand these two terms....

Dirty Reads – This is when you read uncommitted data, when doing this there is no guarantee that data read will ever be committed meaning the data could well be bad.

Phantom Reads – This is when data that you are working with has been changed by another transaction since you first read it in. This means subsequent reads of this data in the same transaction could well be different.

Read Uncommitted

This is the lowest isolation level there is. Read uncommitted causes no shared locks to be requested which allows you to read data that is currently being modified in other transactions. It also allows other transactions to modify data that you are reading.

As you can probably imagine this can cause some unexpected results in a variety of different ways. For example data returned by the select could be in a half way state if an update was running in another transaction causing some of your rows to come back with the updated values and some not to.

To see read uncommitted in action lets run Query1 in one tab of Management Studio and then quickly run Query2 in another tab before Query1 completes.

Query1

```
BEGIN TRAN
```

```
UPDATE IsolationTests SET Col1 = 2
```



```
--Simulate having some intensive processing here with a wait  
WAITFOR DELAY '00:00:10'  
  
ROLLBACK
```

Query2

```
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED  
  
SELECT * FROM IsolationTests
```

Notice that Query2 will not wait for Query1 to finish, also more importantly Query2 returns dirty data. Remember Query1 rolls back all its changes however Query2 has returned the data anyway, this is because it didn't wait for all the other transactions with exclusive locks on this data it just returned what was there at the time.

There is a syntactic shortcut for querying data using the read uncommitted isolation level by using the NOLOCK table hint. You could change the above Query2 to look like this and it would do the exact same thing.

```
SELECT * FROM IsolationTests WITH(NOLOCK)
```

Read Committed

This is the default isolation level and means selects will only return committed data. Select statements will issue shared lock requests against data you're querying this causes you to wait if another transaction already has an exclusive lock on that data. Once you have your shared lock any other transactions trying to modify that data will request an exclusive lock and be made to wait until your Read Committed transaction finishes.

You can see an example of a read transaction waiting for a modify transaction to complete before returning the data by running the following Queries in separate tabs as you did with Read Uncommitted.

Query1

```
BEGIN TRAN  
  
UPDATE Tests SET Col1 = 2
```

```
--Simulate having some intensive processing here with a wait  
WAITFOR DELAY '00:00:10'  
  
ROLLBACK
```

Query2

```
SELECT * FROM IsolationTests
```

Notice how Query2 waited for the first transaction to complete before returning and also how the data returned is the data we started off with as Query1 did a rollback. The reason no isolation level was specified is because Read Committed is the default isolation level for SQL Server. If you want to check what isolation level you are running under you can run “DBCC useroptions”. Remember isolation levels are Connection/Transaction specific so different queries on the same database are often run under different isolation levels.

Repeatable Read

This is similar to Read Committed but with the additional guarantee that if you issue the same select twice in a transaction you will get the same results both times. It does this by holding on to the shared locks it obtains on the records it reads until the end of the transaction, This means any transactions that try to modify these records are forced to wait for the read transaction to complete.

As before run Query1 then while its running run Query2

Query1

```
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ
```

```
BEGIN TRAN
```

```
SELECT * FROM IsolationTests
```

```
WAITFOR DELAY '00:00:10'
```

```
SELECT * FROM IsolationTests
```

```
ROLLBACK
```

Query2

```
UPDATE IsolationTests SET Col1 = -1
```

Notice that Query1 returns the same data for both selects even though you ran a query to modify the data before the second select ran. This is because the Update query was forced to wait for Query1 to finish due to the exclusive locks that were opened as you specified Repeatable Read.

If you rerun the above Queries but change Query1 to Read Committed you will notice the two selects return different data and that Query2 does not wait for Query1 to finish.

One last thing to know about Repeatable Read is that the data can change between 2 queries if more records are added. Repeatable Read guarantees records queried by a previous select will not be changed or deleted, it does not stop new records being inserted so it is still very possible to get Phantom Reads at this isolation level.

Serializable

This isolation level takes Repeatable Read and adds the guarantee that no new data will be added eradicating the chance of getting Phantom Reads. It does this by placing range locks on the queried

data. This causes any other transactions trying to modify or insert data touched on by this transaction to wait until it has finished.

You know the drill by now run these queries side by side...

Query1

```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE
```

```
BEGIN TRAN
```

```
SELECT * FROM IsolationTests
```

```
WAITFOR DELAY '00:00:10'
```

```
SELECT * FROM IsolationTests
```

```
ROLLBACK
```

Query2

```
INSERT INTO IsolationTests(Col1,Col2,Col3)
```

```
VALUES (100,100,100)
```

You'll see that the insert in Query2 waits for Query1 to complete before it runs eradicating the chance of a phantom read. If you change the isolation level in Query1 to repeatable read, you'll see the insert no longer gets blocked and the two select statements in Query1 return a different amount of rows.

Snapshot

This provides the same guarantees as serializable. So what's the difference? Well it's more in the way it works, using snapshot doesn't block other queries from inserting or updating the data touched by the snapshot transaction. Instead row versioning is used so when data is changed the old version is kept in tempdb so existing transactions will see the version without the change. When all transactions that started before the changes are complete the previous row version is removed from tempdb. This means that even if another transaction has made changes you will always get the same results as you did the first time in that transaction.

So on the plus side your not blocking anyone else from modifying the data whilst you run your transaction but.... You're using extra resources on the SQL Server to hold multiple versions of your changes.

To use the snapshot isolation level you need to enable it on the database by running the following command

```
ALTER DATABASE IsolationTests
```

SET ALLOW_SNAPSHOT_ISOLATION ON

If you rerun the examples from serializable but change the isolation level to snapshot you will notice that you still get the same data returned but Query2 no longer waits for Query1 to complete.