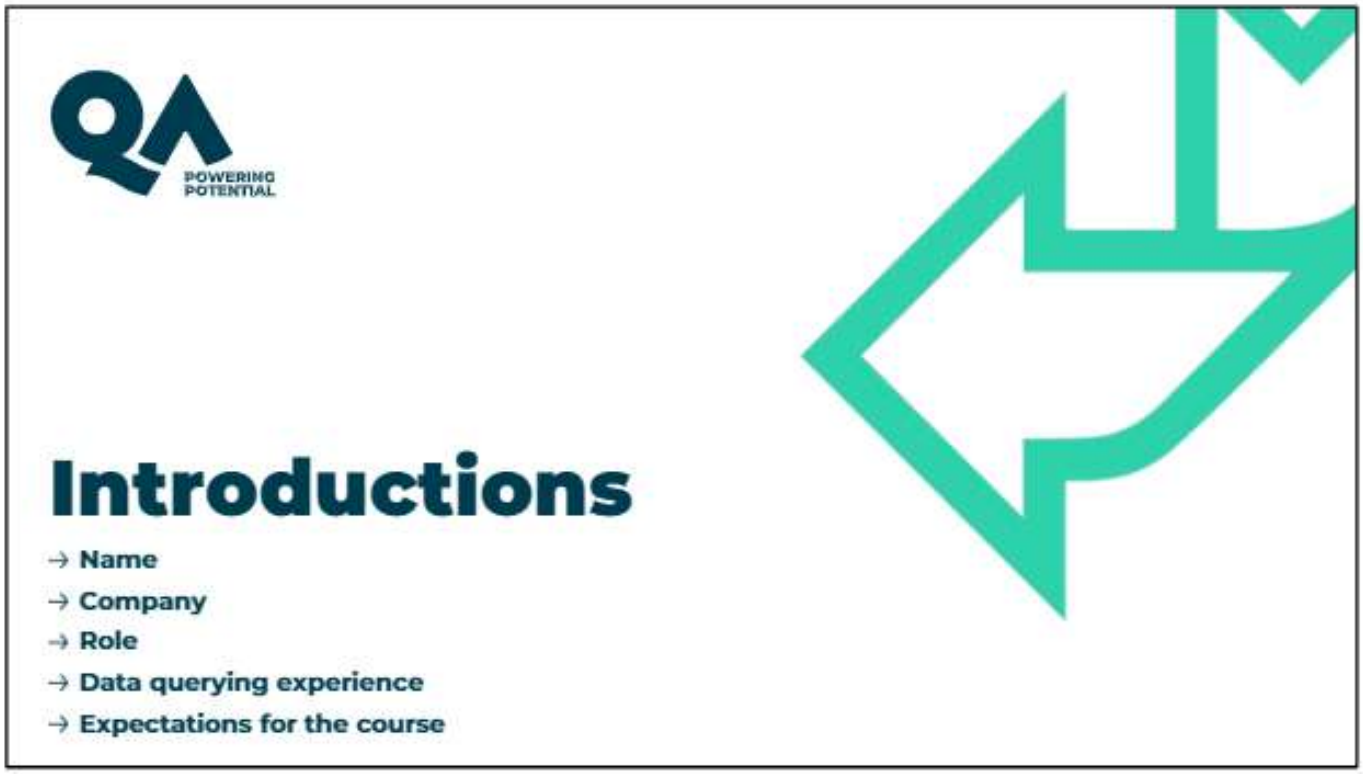
Querying Databases using Transact-SQL

Learner Guide

QA.COM

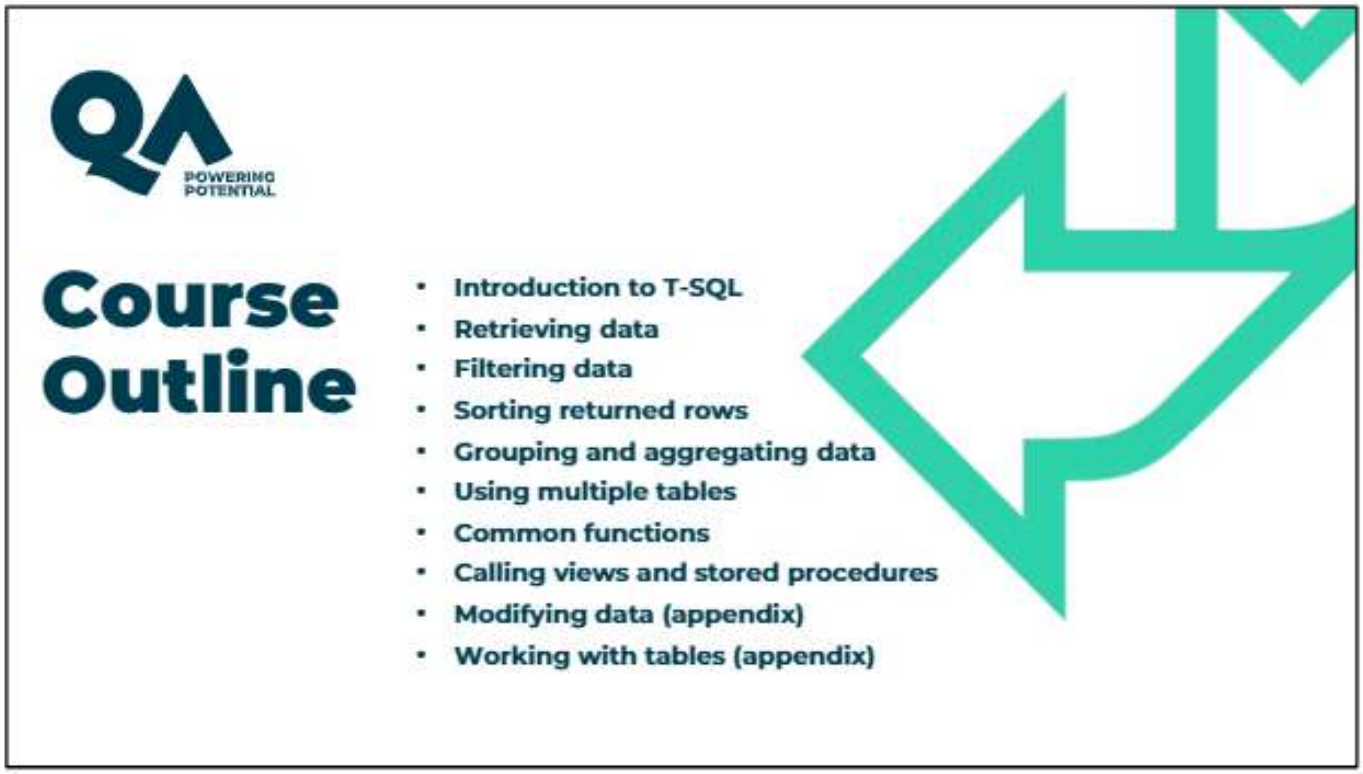
Slides version 1.4


POWERING
POTENTIAL
Introduction

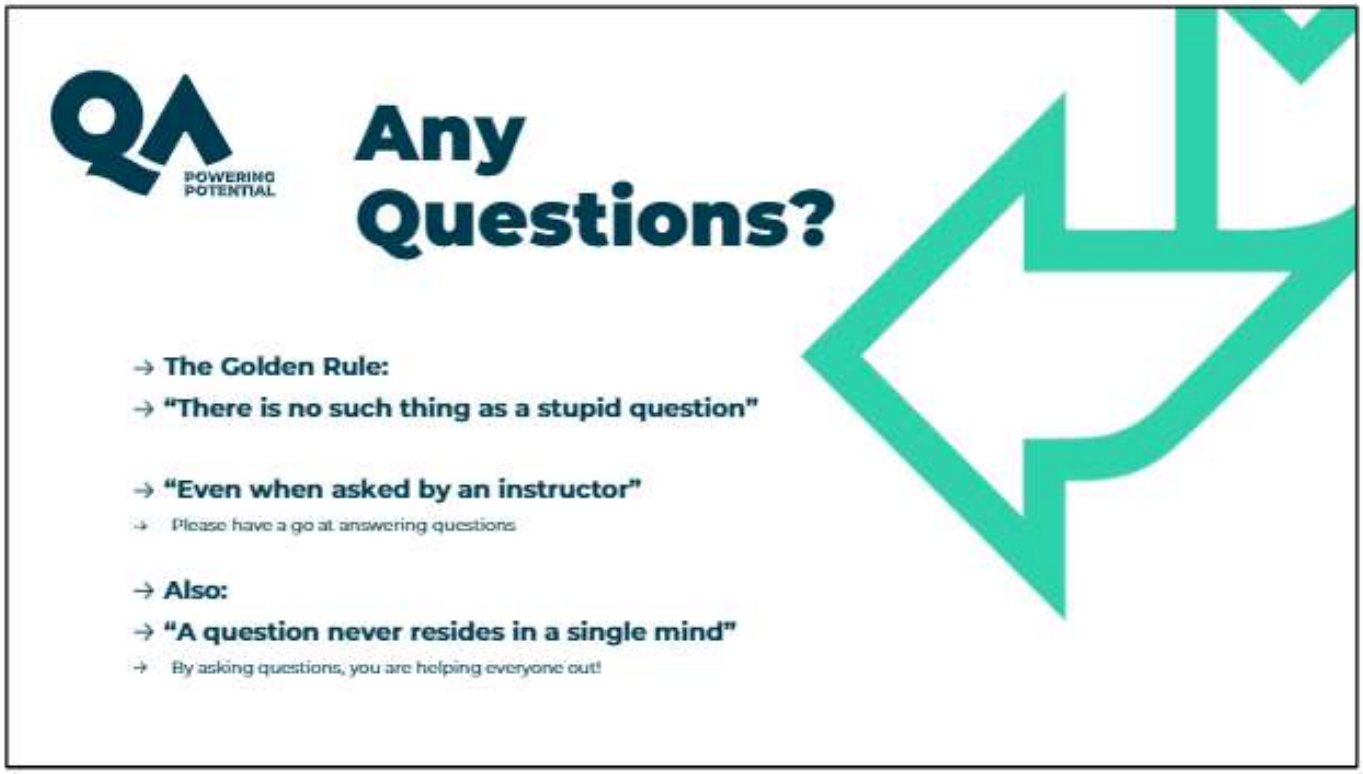

POWERING
POTENTIAL
Introductions
→ Name
→ Company
→ Role
→ Data querying experience
→ Expectations for the course

Q1

4


POWERING
POTENTIAL
Course
Outline
Introduction to T-SQL
Retrieving data
Filtering data
Sorting returned rows
Grouping and aggregating data
Using multiple tables
Common functions
Calling views and stored procedures
Modifying data (appendix)
Working with tables (appendix)


POWERING
POTENTIAL
Objectives
→ At the end of this course, you will be able to write a query that:
→ Filters unwanted data
→ Sorts the results
→ Retrieves data from more than one table
→ Uses built-in functions to calculate some values


POWERING POTENTIAL
Any Questions?
→ The Golden Rule:
→ "There is no such thing as a stupid question"
→ "Even when asked by an instructor"
→ Please have a go at answering questions
→ Also:
→ "A question never resides in a single mind"
→ By asking questions, you are helping everyone out!


POWERING
POTENTIAL
Introduction
to Transact SQL
Version 1.5

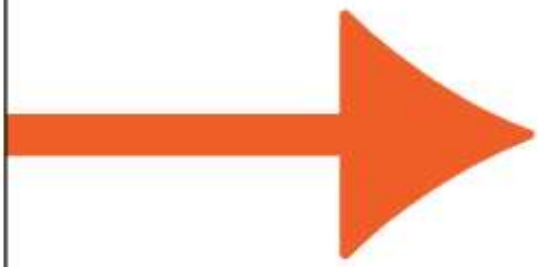

Overview
→ Microsoft SQL Server tools
→ Introduction to T-SQL
→ Hands-on lab


Microsoft SQL
Server Tools

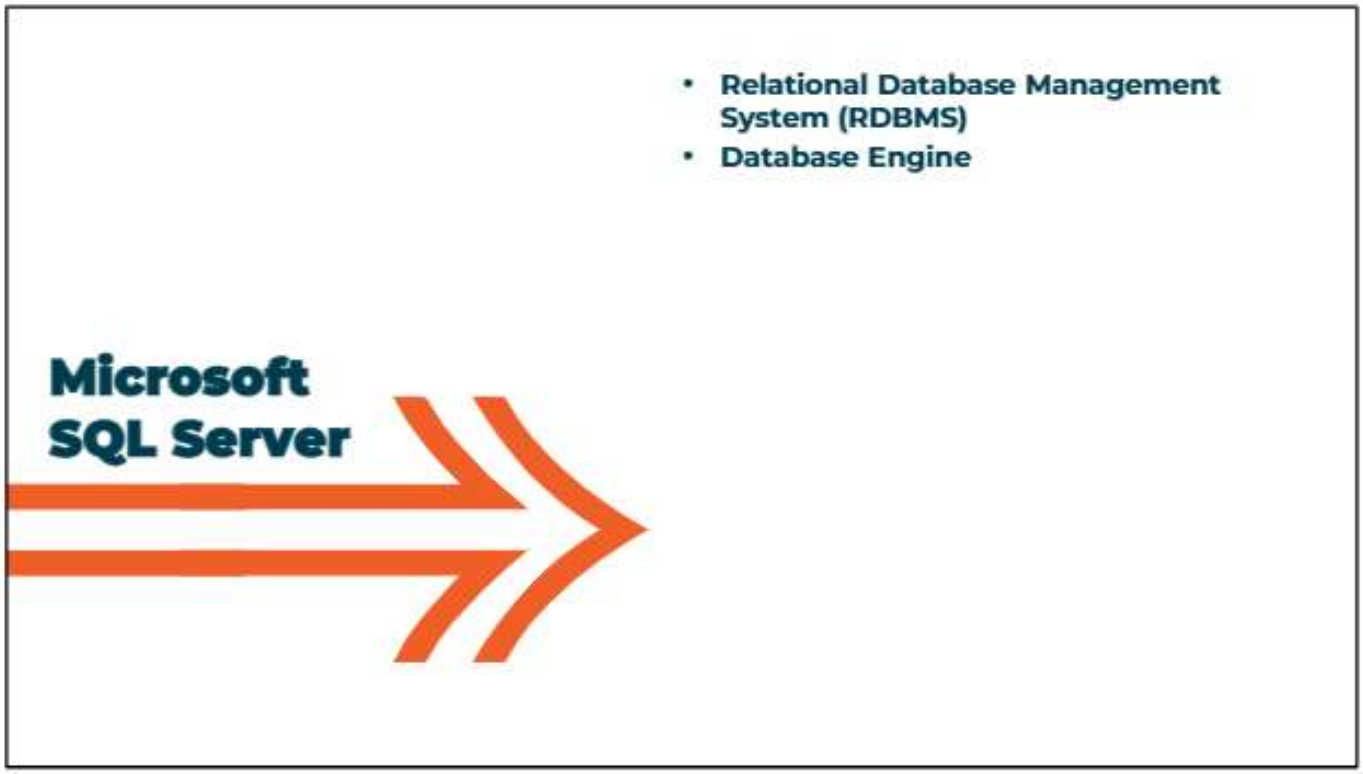
QA

Microsoft SQL Server Tools

- Microsoft SQL Server
- SQL Server Management Studio
- Executing queries



4



For the purposes of this course, the best way to describe Microsoft SQL Server is to call it a Relational Database Management System ("RDBMS"), although there is in fact much more to it.

An RDBMS is a service that stores and processes data and there are many different systems available, from the free such as MySql (www.mysql.com) to the enterprise level such as Oracle (www.oracle.com) and IBM's DB2 (www.ibm.com/software/data/db2).

In an RDBMS, data are stored on rows, in tables, within data files and it is the Database Engine's task to work out how to retrieve the data that we ask it for and where to put data when we modify it.

This course is focused on the retrieval of data but there is an appendix on the basics of data modification for those delegates who are interested in that side of things.

Microsoft SQL Server is Microsoft's offering in the RDBMS field, which itself has many different editions from the free (Microsoft SQL Server Express Edition) to the enterprise level (Microsoft SQL Server Enterprise Edition). Having repeated the

phrase “Microsoft SQL Server” many times in the previous sentence, we will refer to it from now on as SQL Server. The “SQL” is often pronounced “sequel” and we’ll get onto what that means later.

The current version of SQL Server is Microsoft SQL Server 2022, which was released in November 2022.

Microsoft SQL Server Management Studio ("SSMS") is an integrated environment for accessing, configuring, managing, administering and developing all components of SQL Server. SSMS combines a broad group of graphical tools with a number of rich editors to provide access to SQL Server to developers and administrators of all skill levels.

It combines the features of Enterprise Manager, Query Analyzer and other tools from earlier releases of SQL Server.
SSMS works with
•Reporting Services
•Integration Services
•SQL Server Compact Edition
•Analysis Services

With SSMS, database developers and administrators can develop or administer any of the components of the Database Engine.

In September 2018 Microsoft released another development environment for SQL Server, called Azure Data Studio. Unlike SSMS, Azure Data Studio also works on non-Windows operating systems. Azure Data Studio is installed alongside SSMS,

but it can also be installed separately.

SQL Server Solutions

- **A *solution* contains 1 or more *projects***
  - Create an initial project to create the solution
- **A *project* contains files**
- **Add new or existing files to the project**
- **A solution can be added to source control**

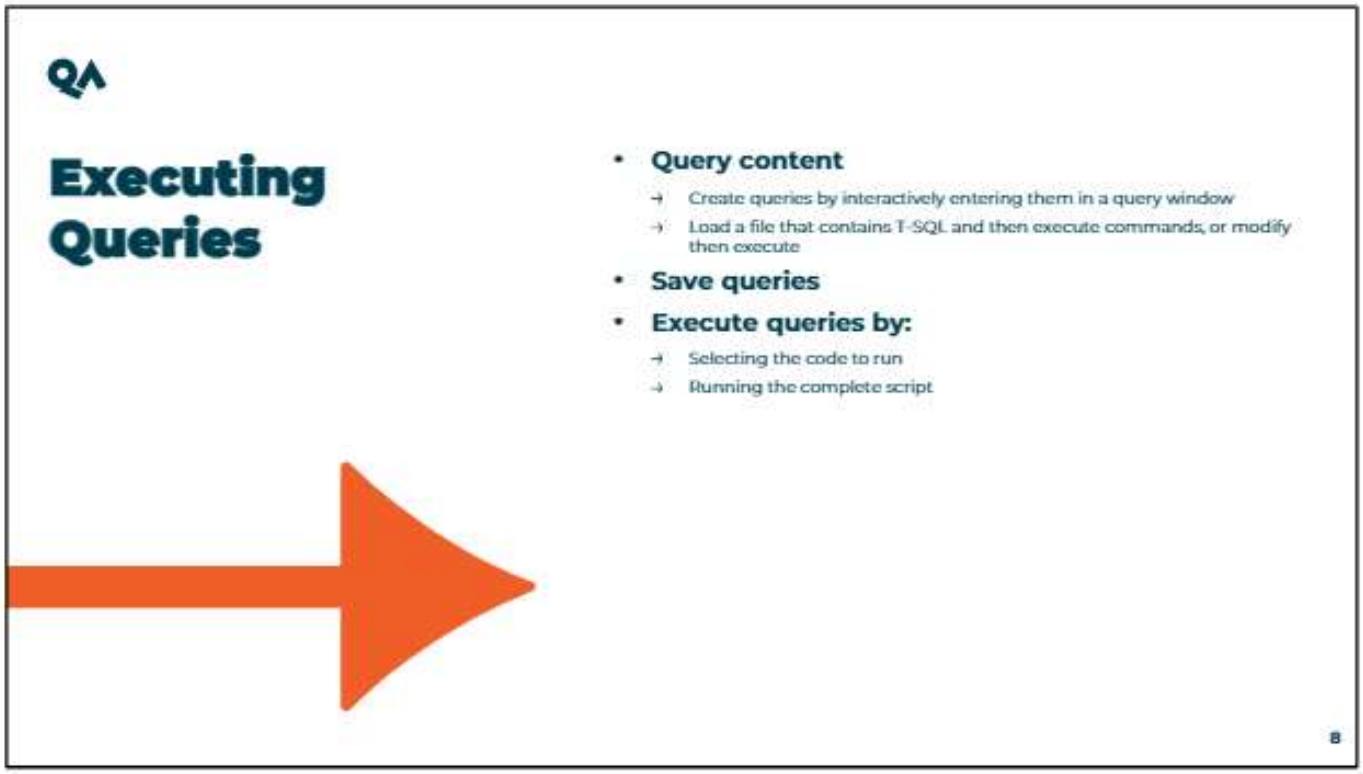
The Solution Explorer window is a component of SSMS that allows us to view and manage items in a solution or a project. It also allows us to use the editors to work on items associated with one of the script projects.

SSMS can be used as a script development platform for:

•SQL Server (TSQL)
•Analysis Services (MDX, DMX or XMLA)
•SQL Server Compact Edition (TSQL)

Solutions are a way of managing script files and folders.

It should be pointed out that we aren't obliged to use projects and solutions – we can manage perfectly fine just by saving all of our scripts to the same folder. However, once there are more than one or two people working on a database we run into the problem of version management and what happens when more than one person tries to update a script at the same time. Source control, also known as SCM (Software Configuration Management), tries to reduce these kind of problems and SSMS Solutions and Projects can be very easily added to source control systems.



To write a query we can either:

•Start with a blank query and enter the code to execute
•Open a pre-written query script

Scripts can be saved to disk using File | Save.

When a script is loaded into SSMS we can either:

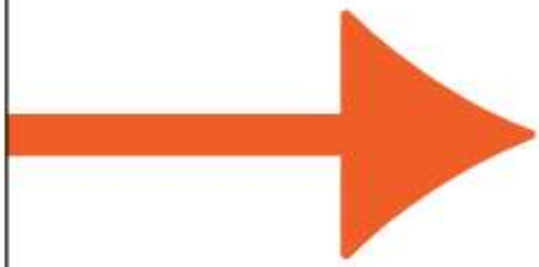
•Select only the statements that we wish to run, then run them (F5 is the keyboard shortcut)
•Run the complete script – no statements selected


Introduction
to T-SQL

QA

Introduction to T-SQL

- What is SQL?
- What is T-SQL?



10

SQL stands for Structured Query Language and is a standard language for accessing (and manipulating) data in databases. SQL is controlled by the American National Standards Institute (ANSI). The current standard is SQL:2016.

As is the case with all standards, the various vendors (Oracle, IBM, Microsoft, et al) all feel that the standard is lacking in certain areas, so whilst all RDBMSs support ANSI-SQL to a greater or lesser extent (few, if any, RDBMSs implement the whole standard), they invariably also have extensions and additions to the standard.

Microsoft's implementation of the standard is called Transact-SQL, or T-SQL for short, or just SQL for even shorter. Most of what we cover in this introductory course is ANSI-compliant.

Oracle's flavour is called PL/SQL ("procedural language") whilst IBM's is SQL PL...

Review

- RDBMSs
- Microsoft SQL Server
- SSMS
- Query Editor


POWERING
POTENTIAL
Retrieving Data
Version 1.5


Overview
• SELECT statement
• Select lists
• Expressions
• Changing column names
• Best practices
• Hands-on lab

Objectives

At the end of this module you will be able to:

- Write a query that retrieves every column of every row in a table
- Write a query that retrieves only some columns from a table
- Write a query that calculates some of its column values


SELECT
statement

QA

SELECT statement

- **SELECT** <<field(s)>>
- **FROM** <<table(s)>>
- WHERE <<condition(s)>>
- GROUP BY <<field(s)>>
- HAVING <<condition(s)>>
- ORDER BY <<field(s)>>

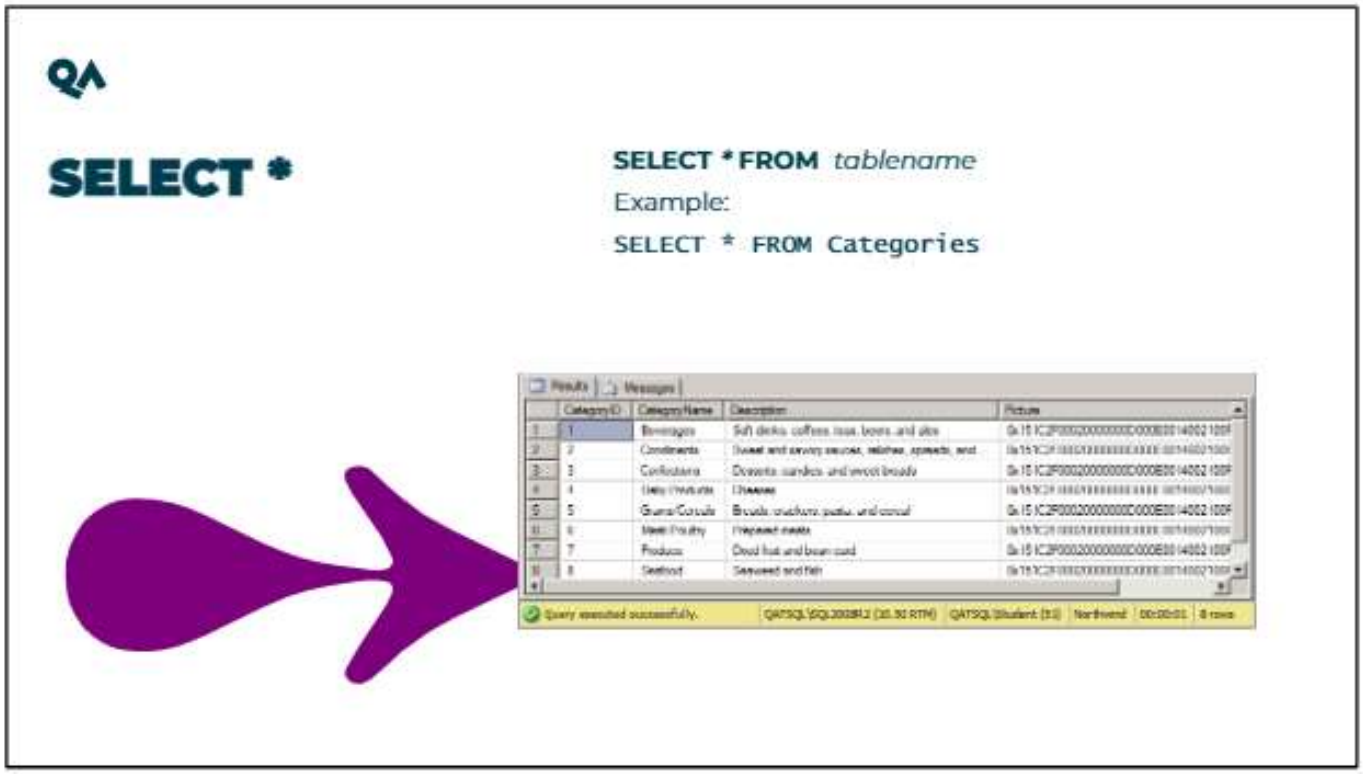
The *SELECT* statement is the basis of all queries in SQL. There are many optional parts to a select statement, as the slide (taken from Books Online) shows. There are dozens of even more esoteric options that haven't been listed here.

The highlighted parts are the bit we're interested in for now; we'll introduce the others as the course progresses.

An explanation is fairly fundamental, and hopefully straightforward. We are telling SQL that we wish to:

SELECT these pieces of information (columns / fields)
FROM these places (tables)

Although for now we'll keep it really simple and only select columns from a single table, it is worth bearing in mind that it is possible to put things other than column names into the select list and that one of the tables in the table list could be another select statement!



The simplest form of a select statement is:

SELECT * FROM *tablename*.

It's generally referred to as a "select star" and is simply telling SQL to go and retrieve everything from the specified table.

In the example on the slide, we're asking for every row and every column from the Categories table (in the Northwind database). We chose this table because it only has four columns and eight rows.

It's useful as a "quick check" to see what data are in a table, as long as the table is quite small. We really wouldn't want to be running a select star query against a table with a hundred columns and a million rows!
It's also useful for "waking up" *Intellisense* in SSMS; once the Query Editor can see which table we're querying, it can populate the completion list with columns that belong to that table, making it easier to avoid spelling mistakes in column names.

Although it is not required in most cases, it is recommended that you terminate statements with a semicolon. Microsoft have said that in a later release they *may*

make it mandatory to terminate your statements, so it could be considered best practice to start using them now just in case. For example:

```
SELECT * FROM Categories;
```


Select lists

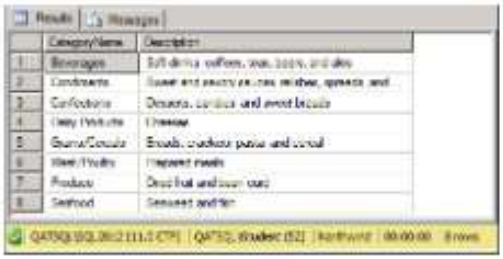
QA

Select lists

SELECT col1, ..., coln FROM *tablename*

Example:

```
SELECT CategoryName, Description
FROM Categories
```



Far more useful and functional than the SELECT * ... statement is the ability to provide SQL with a list of the columns we'd like to see in our results.

We provide a comma separated list of the columns we want to see in the results.

In the example on the slide, we have selected only the CategoryName and Description columns from the Categories table.


POWERING
POTENTIAL
Expressions

QA

Expressions in select lists

- "Calculated" values
- Expression types:

Type	Operators
- Arithmetic operators	+ - * / % *e.g. UnitsInStock + UnitsOnOrder*
- String concatenation operator	+ *e.g. FirstName + ' ' + LastName*

As mentioned earlier, we are not restricted to just asking for column names in our select lists. It is also possible to tell SQL to calculate a value for us using an expression.

As you might expect, all of the standard mathematical operators are available – addition, subtraction, multiplication and division plus another operator that you might not be familiar with: *modulus*, which calculates the remainder of a division, for example 5 / 3 is 1 remainder 2, so 5 % 3 is equal to 2. On that subject, an important "gotcha" with integer maths in SQL is that 5 / 3 equals 1, i.e. the result of operations on two integers is always another integer, with the result rounded down. If you want to ensure that 5 / 3 equals 1.3 recurring, divide 5.0 by 3.0 (in other words, use decimal numbers rather than integers).

There is also the "string concatenation operator", which is a plus sign, which appends one text value onto the end of another.

NOTE: In SQL string literals, like the space character between the FirstName and LastName above, need to be delimited with single quote characters.

Expressions examples (1)

```
SELECT ProductID, ProductName,
       UnitsInStock, UnitsOnOrder,
       UnitsInStock + UnitsOnOrder
FROM Products
```

Results | Messages

	ProductID	ProductName	UnitsInStock	UnitsOnOrder	(No column name)
1	1	Chai	39	0	39
2	2	Chang	17	40	57
3	3	Aniseed Syrup	13	70	83
4	4	Chef Anton's Cajun Seasoning	53	0	53
5	5	Chef Anton's Gumbo Mix	0	0	0
6	6	Grandma's Boysenberry Spread	120	0	120
7	7	Uncle Bob's Organic Dried Pears	15	0	15
8	8	Northwoods Cranberry Sauce	6	0	6
9	9	Mishi Kobe Niku	29	0	29

Query e... | QATSQL\SQL2008R2 (10.50 RTM) | QATSQL\Student (53) | Northwind | 00:00:00 | 77 rows

Expressions examples (2)

```
SELECT FirstName, LastName,
       FirstName + ' ' + LastName
FROM Employees
```

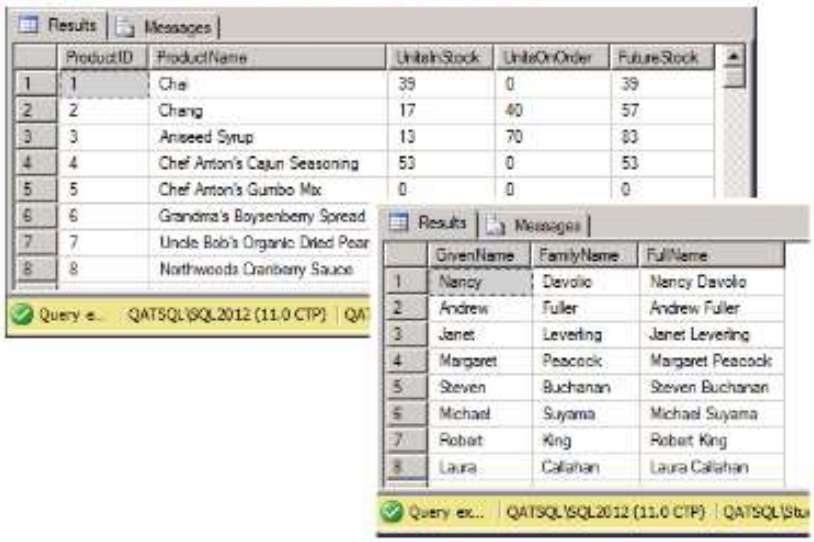
Results | Messages

	FirstName	LastName	(No column name)
1	Nancy	Davolio	Nancy Davolio
2	Andrew	Fuller	Andrew Fuller
3	Janet	Leverling	Janet Leverling
4	Margaret	Peacock	Margaret Peacock
5	Steven	Buchanan	Steven Buchanan
6	Michael	Suyama	Michael Suyama
7	Robert	King	Robert King
8	Laura	Callahan	Laura Callahan
9	Anne	Dodsworth	Anne Dodsworth

Query ex... | QATSQL\SQL2008R2 (10.50 RTM) | QATSQL\Student (53) | Northwind | 00:00:00 | 9 rows


Changing
column names

Changing column names



You might have noticed from the two previous examples that when we use an expression in our select list, the database engine doesn't know what to call the "new" column so it comes out as "(No column name)". In SQL, we can change column names using the *AS* keyword (this is known as *aliasing*), so we could change the first example from before to look as follows:

```
SELECT
        ProductID, ProductName, UnitsInStock, UnitsOnOrder,
        UnitsInStock + UnitsOnOrder AS FutureStock
FROM Products
```

Some additional points about aliasing:

- You can rename any column.
- You can use it for table names as well. We'll see why that's useful later.
- The "AS" is optional but we would **highly recommend** that you get into the habit of using it.

So a modification to the second earlier example might be:

```
SELECT
```

```
        FirstName AS GivenName, LastName FamilyName,
        FullName = FirstName + ' ' + LastName
FROM Employees
```

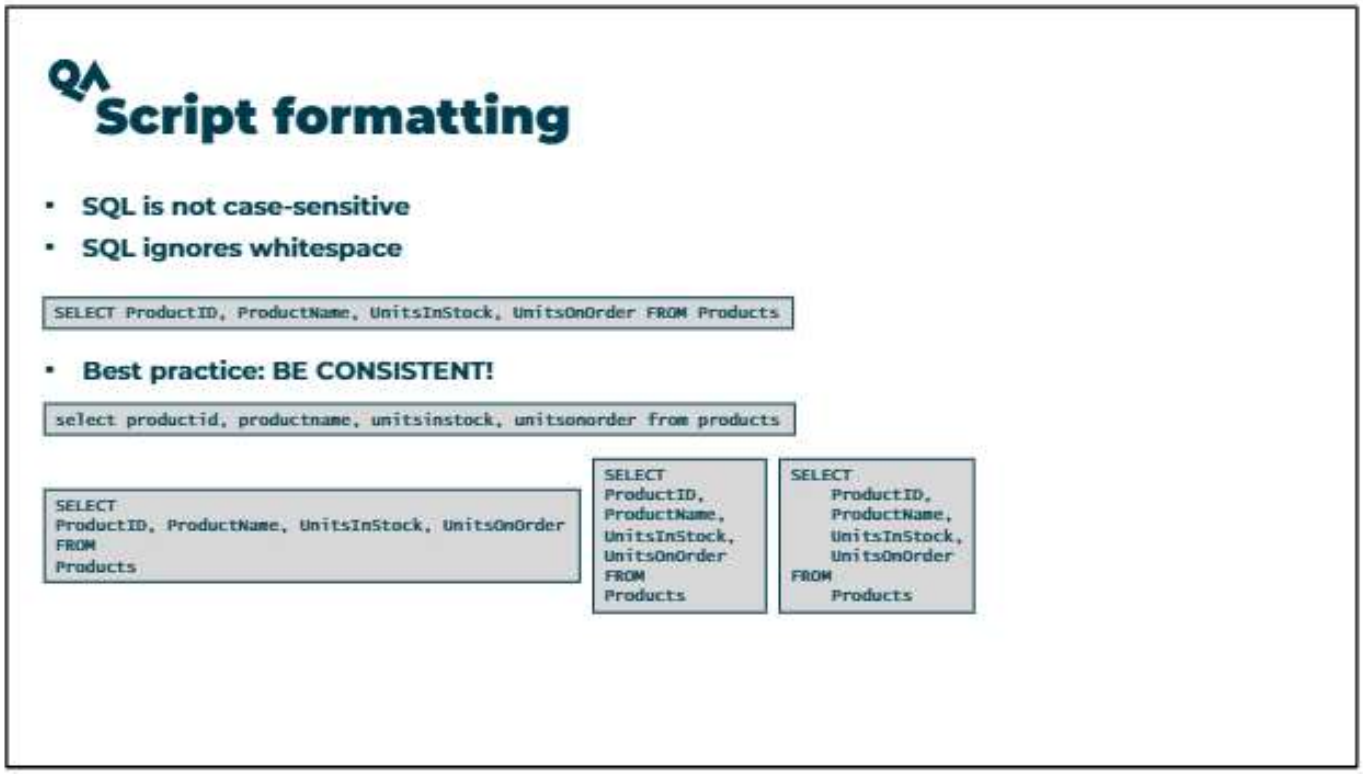
Note the missing "as" in the LastName column and the other alternative syntax using an equals sign on the concatenated column


Best practices

QA

BEST PRACTICES

- Script formatting
- Comments
- Four-part names



The database engine really doesn't care how we format our queries. It ignores most whitespace and is not case-sensitive. This is a good thing, because it means we can format our scripts in whatever way we think is easiest to read. This is also a bad thing, for exactly the same reason; it means that everyone can format their scripts in whatever way they want!

Some organisations have guidelines as to how code should be formatted, but even the best-written guidelines will be unable to take every possible scenario into account, so the key is to try to be consistent with script formatting.

QA

Comments

- **Ignored by SQL**
  - → Add documentation to your scripts
- **Two styles:**

```
→ SELECT
    UnitsInStock + UnitsOnOrder AS
  FutureStock -- calculate FS
  FROM
     Products
  and
→/*
  This query gets a list of employees
  with their full names
  */
  SELECT FirstName + ' ' + LastName AS Name FROM
  Employees
```

- **Best practice: comment your scripts**

Documenting scripts can help enormously when we come back to them in six months' time!

There are two styles of comments in SQL:

- Single line comments – putting two dashes ("--") before some text tells SQL to ignore everything up until the next new line (this is the exception to the rule that SQL doesn't care about whitespace)
- Multi line (or inline) comments – typing a slash then a star ("/*") tells SQL to ignore everything until it comes to a star immediately followed by a slash ("*/"). It's typically used to comment out multiple lines but can also be used to comment out only a part of a single line.

Best practice says that we should comment all but the most basic of scripts. As mentioned before, many organizations have coding standards which may include a requirement for a standard comment block in each script that specifies the author, date and purpose of the script. As a rule of thumb, if you have had to think about how to write the code you've written, it might be worth adding a little comment as a reminder.

/* Author: Bob Cratchit

```
   Date: 25/12/1843
   Purpose: Find out who visited Ebeneezer recently */

SELECT
                'Ghost of ' + GhostPeriod -- add ghost's time to the words "Ghost
of"
FROM Ghosts
```

QA

FOUR-PART NAMES

Referring to tables in queries:

```
→ FROM Employees
→ FROM dbo.Employees
→ FROM Northwind.dbo.Employees
→ FROM QATSQL.Northwind.dbo.Employees
```

Best Practice: use at least two-part names

In SQL Server, tables belong to things called *schemas*, which are (amongst other things) used for managing access to groups of tables.

A schema belongs to a database (also sometimes known as a *catalog*)

A database resides on a database server.

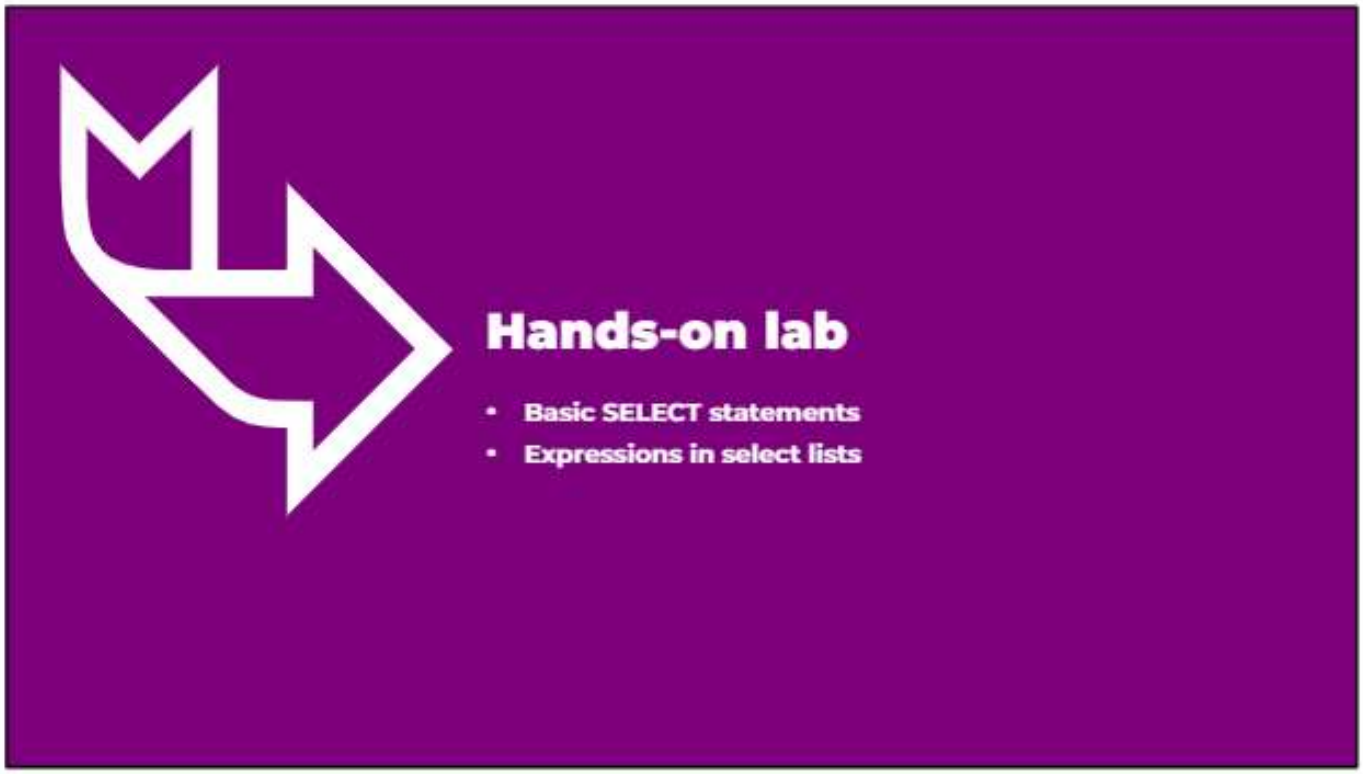
Therefore, to fully qualify a table name, we should use the format *servername.databasename.schemaname.tablename*, but that isn't truly necessary:

- If we don't specify a server, SQL assumes we mean the current server
- If we don't specify a database, SQL assumes we mean the current database (remember we can change the current database with a "USE *database*" statement or choosing from the drop-down list in the Query Editor)
- If we don't specify a schema name, SQL assumes that we mean the *current user*'s default schema. This is an important point as someone who is running one of our stored scripts might have a different default schema than us. To add to the fun, in SQL, a default schema can also be assigned to a group of users.

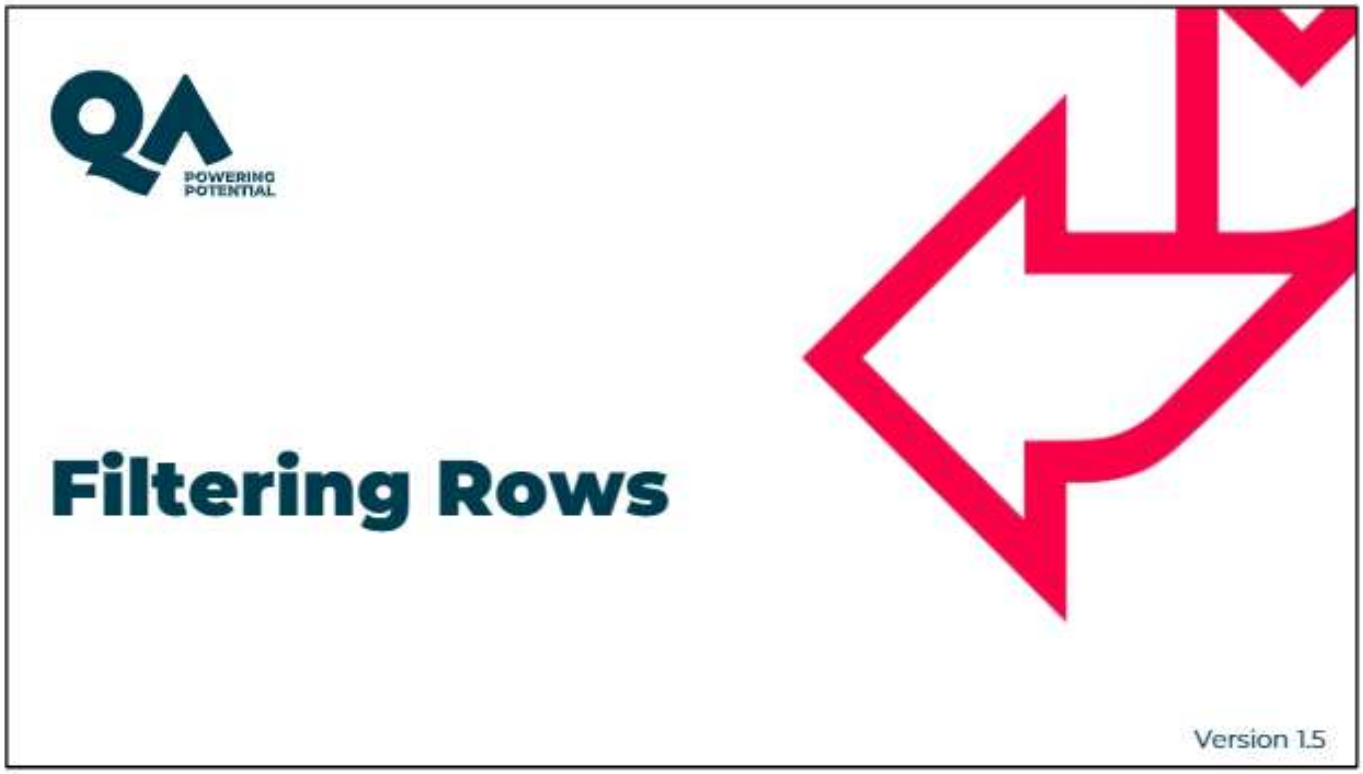
For this reason, when referring to tables, try to get into the habit of specifying two-part names.

In the Northwind sample database that this course uses, all tables belong to the default schema of dbo ("database owner") and therefore it isn't an issue but you are still encouraged to get into the habit now.

NOTE: Up until now, for simplicity, this manual has just been using one-part names but from now on it will switch to two-part names.


Hands-on lab
• Basic SELECT statements
• Expressions in select lists


POWERING
POTENTIAL
Review
• SELECT * FROM ...
• SELECT columns FROM ...


POWERING
POTENTIAL
Filtering Rows
Version 1.5

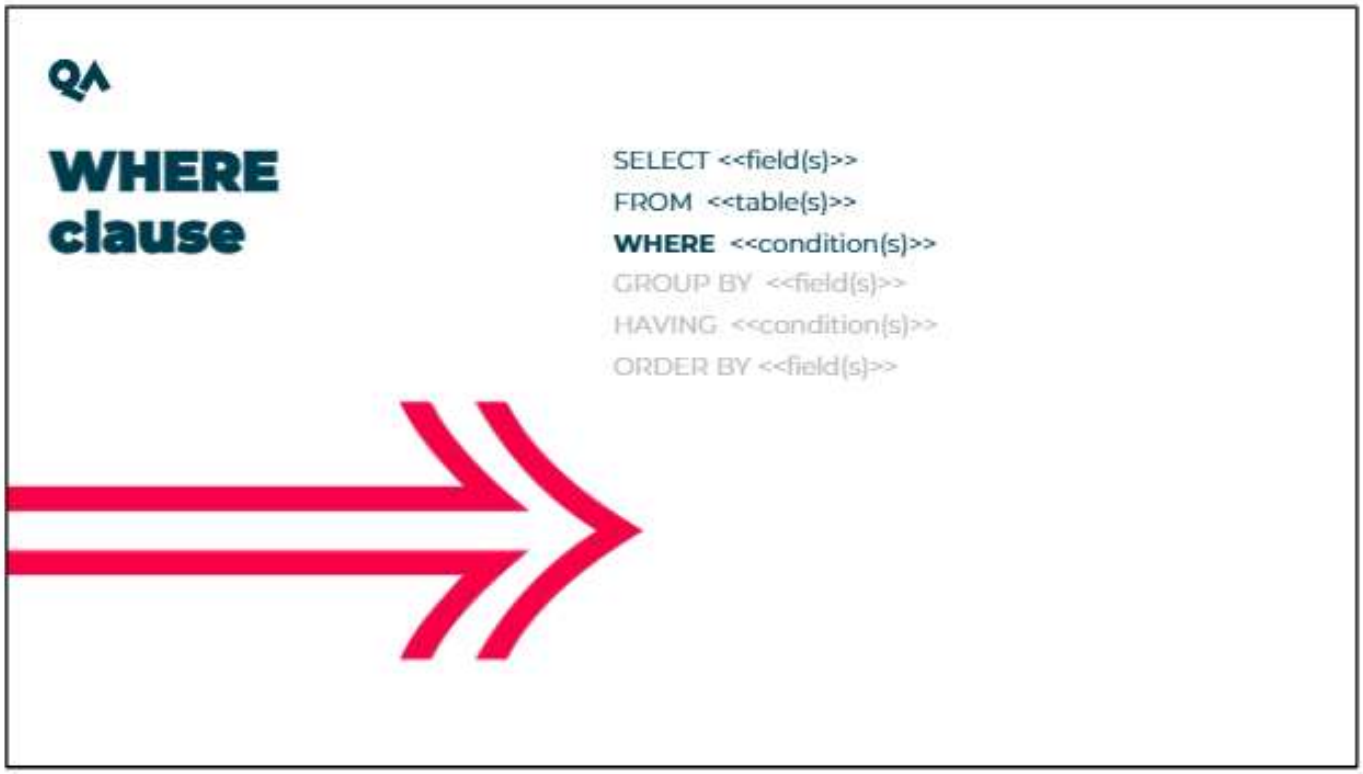

Overview
WHERE clause
WHERE expression
Operators
Unknown values
Best practices
Hands-on lab

Objectives

At the end of this module you will be able to:

- write a query that uses an equality filter
- write a query that use a comparison filter
- write a query that uses the IN and BETWEEN operators
- write a query that looks for text within a column
- write a query that correctly identifies NULL values


WHERE clause



The optional WHERE clause is how we tell SQL that we want to limit the results that a query returns. Without a WHERE clause, all rows in the table are returned.

We very rarely want every row to be returned, so it's quite an important part of a query!

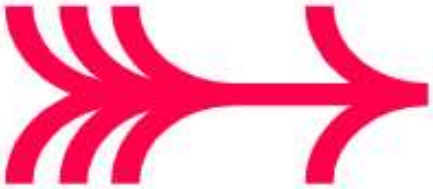

WHERE
expression

QA

WHERE expression

- **Is a predicate**
  - → An expression that returns either true or false
- **SQL retrieves all rows that are true**
  - → Or possibly none at all
- **Example:**

```
SELECT
  ProductID,
  ProductName,
  UnitsInStock,
  UnitsOnOrder
FROM
  dbo.Products
WHERE
  Discontinued = 0
```



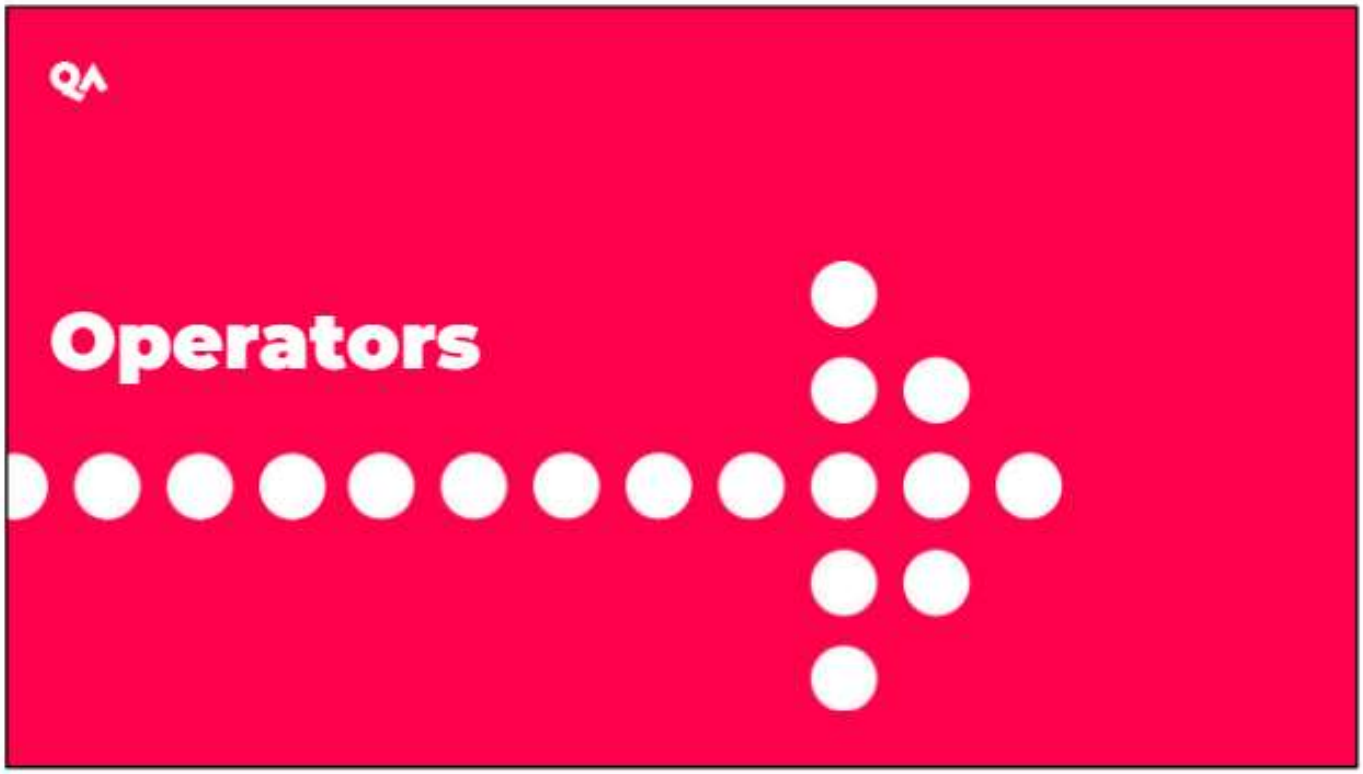
7

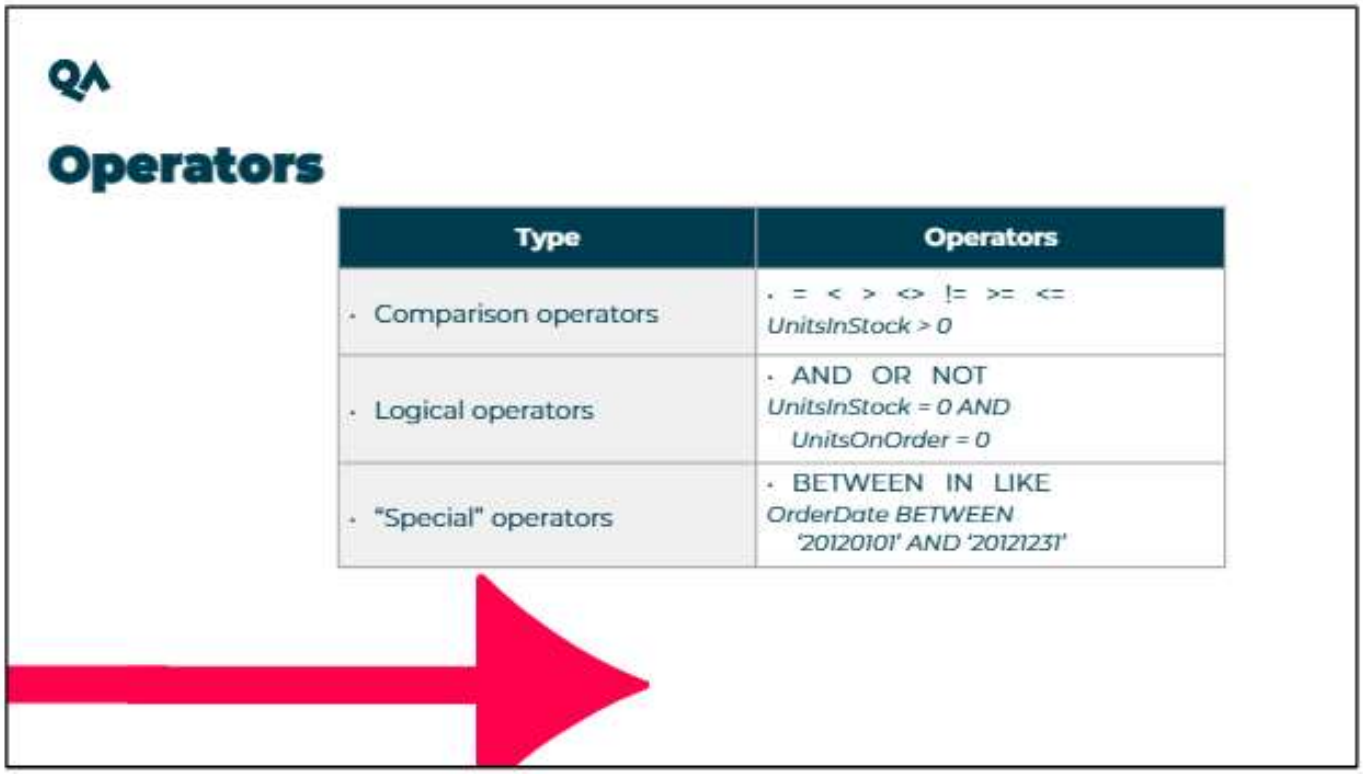
A WHERE clause consists of a predicate, which is an expression that returns either true or false.

SQL will retrieve all the rows in the table for which that predicate returns true.

In the example on the slide, we are telling SQL that we only want to see those products whose discontinued column has a value of zero (i.e. they are *not* discontinued)

NOTE: depending on how the database has been created, SQL may not actually need to perform the test on every single row, so if you're concerned about a table with a million rows taking a long time to filter, don't be.


Operators



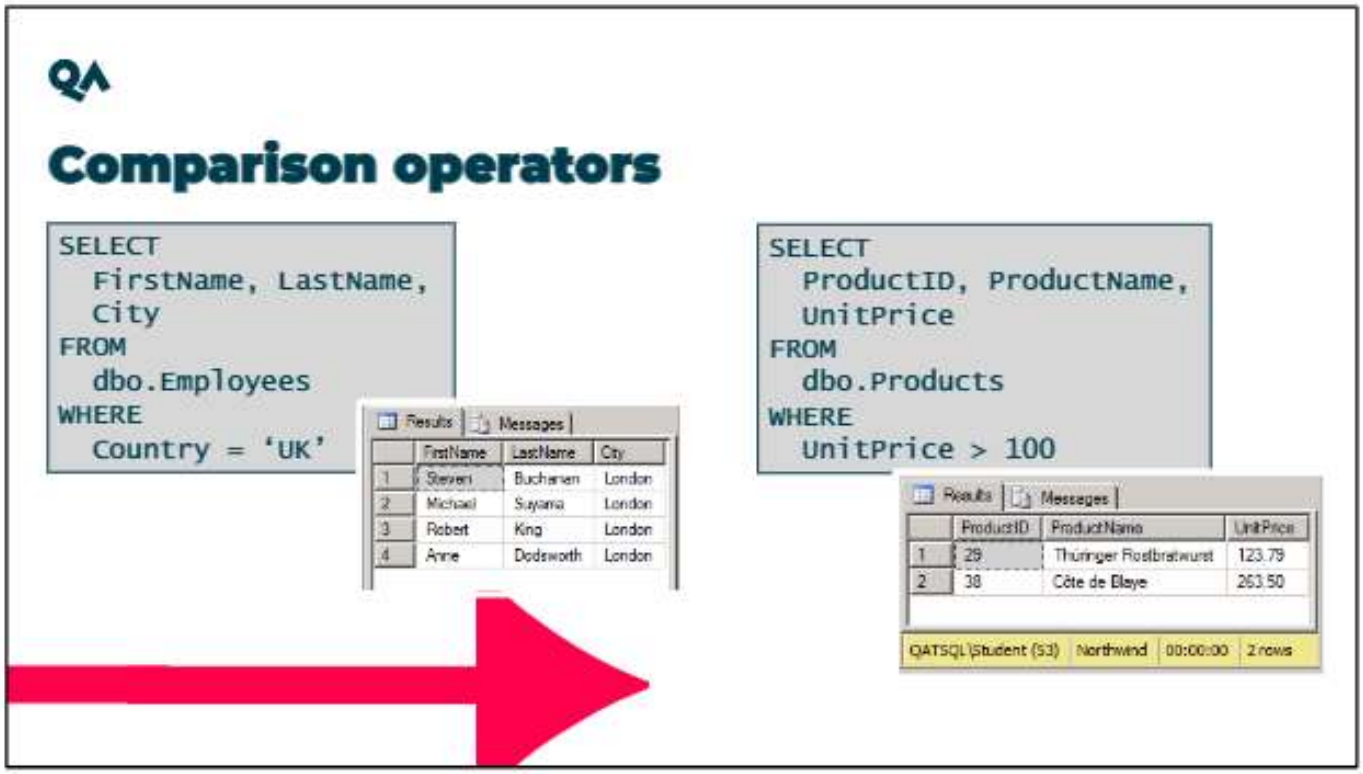
We do not have to use an equality comparison in our WHERE clauses, nor are we limited to only one criteria in our predicates.

In the first example – *UnitsInStock > 0* - the query is looking to return any products which have a UnitsInStock value greater than 0.

In the second example – *UnitsInStock = 0 AND UnitsOnOrder = 0* – the query is looking to return any products which have a UnitsInStock value of 0 but also a UnitsOnOrder value of 0, meaning that none are in stock and none are on order.

In the third example – *OrderDate BETWEEN '20120101' AND '20121231'* – the query is looking to return any orders which were placed between 1st January 2012 and 31st December 2012. As dates are written in different formats around the world (for example 8th April 2022 would be written as 8/4/2022 in the UK, but 4/8/2022 in the USA – it is recommended to use a language neutral format structured as YYYYMMDD.

The next few pages will cover some of the options available when filtering.



Comparison operators perform standard value comparisons :-

- = Equal to
- > Greater than
- < Less than
- >= Greater than or equal to
- <= Less than or equal to and
- <> Not equal to

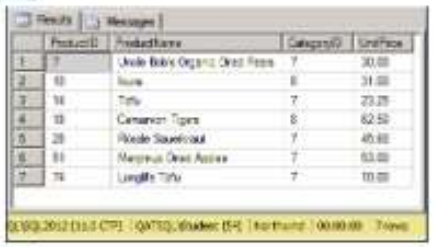
Note that the following can also be used

- != Not equal to (same as <>)
- !< Not less than (same as >=)
- !> Not greater than (same as <=)

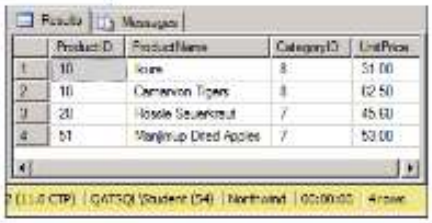
NOTE: The use of *not equal to* should be avoided. It is always more efficient to say what you **do** want to see rather than what you don't, as this way SQL will return a smaller, more specific result set. For example, if somebody asked what you wanted for Christmas it is more helpful to tell them what you **do** want rather than what you **don't**.

Logical operators

```
SELECT
  ProductID, ProductName,
  CategoryID, UnitPrice
FROM
  dbo.Products
WHERE
  CategoryID = 7
  OR CategoryID = 8
  AND UnitPrice > 30
```



```
…
WHERE
  (CategoryID = 7
  OR CategoryID = 8)
  AND UnitPrice > 30
```



Logical operators allow us to combine multiple conditions for our filter:-

- AND - retrieve rows that meet all of the criteria – generally returns fewer rows. For example, asking for a pair of shoes which are size 11 **and** black.
- OR - retrieve rows that meet any of the criteria – generally returns more rows. For example, asking for a shirt which is either red **or** blue.
- NOT - reverse the result of the following expression. For example, asking for a product which is **not** in a certain category.

We need to be careful with the order in which expressions are evaluated though – the two illustrated examples look remarkably similar but give different results.

The first query is saying "give me all the products in Category 7, plus all the products in Category 8 which have a UnitPrice more than 30".

By using brackets around the *OR*ed expression, the second query is saying "give me all the products in Categories 7 or 8 which have a UnitPrice of more than 30".

The complete order in which expressions are resolved is:

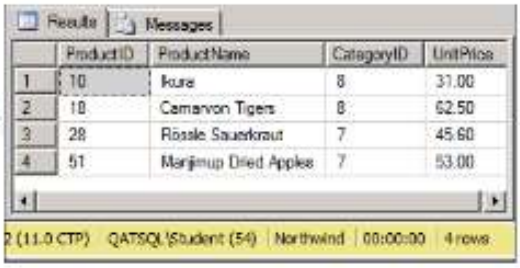
*NOT*s are evaluated first, then *AND*s are evaluated, then the *OR*s.

NOTE: Try to avoid using NOTs.

Specifying a list of values - IN

```
SELECT
  ProductID, ProductName,
  CategoryID, UnitPrice
FROM
  dbo.Products
WHERE
  (CategoryID = 7
  OR CategoryID = 8)
  AND UnitPrice > 30
```

```
SELECT
  ProductID, ProductName,
  CategoryID, UnitPrice
FROM
  dbo.Products
WHERE
  CategoryID IN (7, 8)
  AND UnitPrice > 30
```

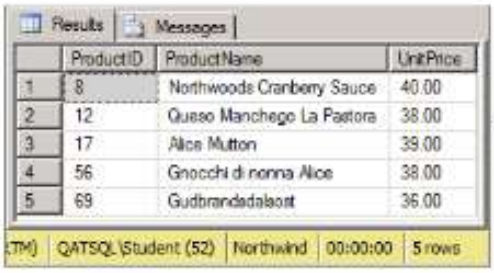


The IN operator enables us to specify a list of values which we want the expression to match. It is a shorthand way of specifying multiple OR statements that apply to the same column.

Specifying a range of values - BETWEEN

```
SELECT
  ProductID, ProductName,
  UnitPrice
FROM
  dbo.Products
WHERE
  UnitPrice >= 35
  AND UnitPrice <= 40
```

```
SELECT
  ProductID, ProductName,
  UnitPrice
FROM
  dbo.Products
WHERE
  UnitPrice
    BETWEEN 35 AND 40
```



Rather than using "greater than or equal to", "*AND*" and "less than or equal to", we can use a *BETWEEN* expression to specify a range of values.

BETWEEN and *IN* can look extremely similar; the difference is that *IN* can be used for a list of distinct values, such as:
CategoryID IN (1, 3, 5) [which excludes 2 and 4]

Whereas *BETWEEN* is always a range, such as:
CategoryID BETWEEN 1 AND 5 [which includes 2 and 4]

NOTE: We cannot exclude the boundary values with a *BETWEEN*. It is always equivalent to a "greater than or equal to", never a "greater than".

String comparisons - LIKE

```
SELECT
  FirstName, LastName,
  Title
FROM
  dbo.Employees
WHERE
  Title LIKE 'Sales%'
```

Results | Messages

	FirstName	LastName	Title
1	Nancy	Davolio	Sales Representative
2	Janet	Leverling	Sales Representative
3	Margaret	Peacock	Sales Representative
4	Steven	Buchanan	Sales Manager
5	Michael	Suyama	Sales Representative
6	Robert	King	Sales Representative
7	Anne	Dodsworth	Sales Representative

QATSQL\Student (54) Northwind 00:00:00 7 rows

```
SELECT
  FirstName, LastName,
  Title
FROM
  dbo.Employees
WHERE
  Title LIKE '%sales%'
```

Results | Messages

	FirstName	LastName	Title
1	Nancy	Davolio	Sales Representative
2	Andrew	Fuller	Vice President, Sales
3	Janet	Leverling	Sales Representative
4	Margaret	Peacock	Sales Representative
5	Steven	Buchanan	Sales Manager
6	Michael	Suyama	Sales Representative
7	Robert	King	Sales Representative
8	Laura	Callahan	Inside Sales Coordinator
9	Anne	Dodswo...	Sales Representative

CTP) | QATSQL\Student (54) Northwind 00:00:00 9 rows

Using comparison operators (=, <, >, <=, >=, <>) against text data performs exact matches.

We can perform wildcard pattern matching on text columns using the *LIKE* operator:

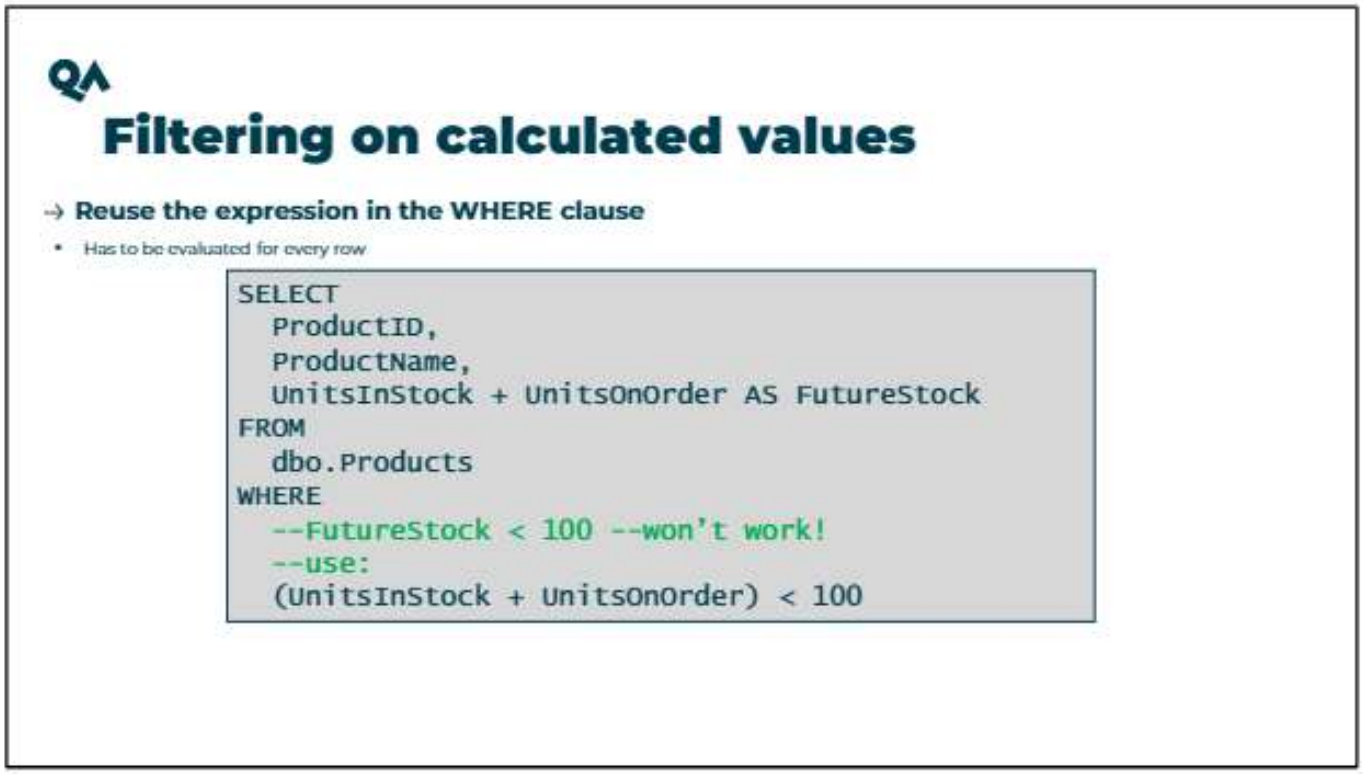
- % Matches any string of zero or more characters
- _ Matches any single character
- [] Matches any single character listed within the square brackets
- [^] Matches any single character *not* listed with the square brackets

Wildcards can appear anywhere within the search text. In the left-hand query, we are asking for all the employees whose job title *begins with* "Sales", whilst in the second we are asking for all employees with a job title that *contains the letters* "sales".

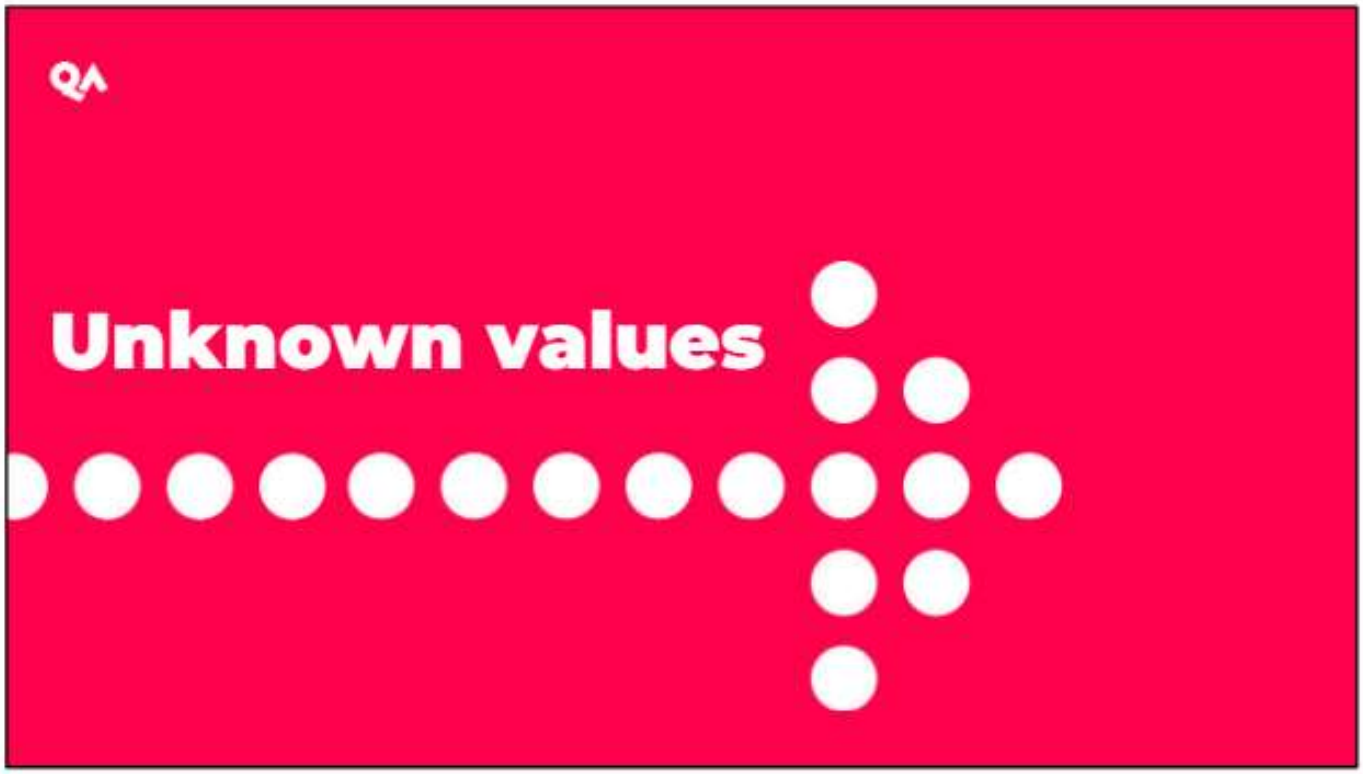
On the installation of SQL Server used for this course, these string searches are not case sensitive. The string "sales" matches "Sales" (and would also match "SALES", "SaLeS", etc.). Your real world database might be case sensitive.

Using leading wildcards (as in the right-hand example) is extremely bad practice.

SQL will *always* have to examine every single row in the table to find your rows. This should be avoided as much as possible. If you really need very flexible string matching, it is possible to use Full-text Indexing, an advanced feature of SQL Server which is beyond the scope of this course.

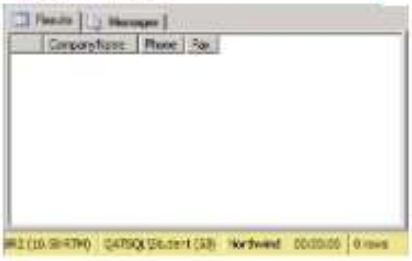


To filter based on a calculation we need to reuse the calculated expression in the WHERE clause. We cannot use the expression's alias, because that alias column hasn't been created at the point in time when we want to start filtering. Effectively, it's only if the WHERE expression returns true that we pick that row and then calculate the calculated value, giving that column a name.


Unknown values

Finding unknown values (NULLs)

```
SELECT
  CompanyName, Phone,
  Fax
FROM
  dbo.Suppliers
WHERE
  Fax = NULL
```



```
SELECT
  CompanyName, Phone,
  Fax
FROM
  dbo.Suppliers
WHERE
  Fax IS NULL
```

Some columns in a database can contain *NULL* values if the column has been set up to allow them and no value was given for the column when the data was entered.

NULL is not zero, nor is it a blank line of text.

NULL typically means one of three things:-

- *not known* (i.e. a person's middle name – we don't know what it is)
- *not applicable* (this person *doesn't have* a middle name)
- *not yet been set* (i.e. a "Units On Order" column – 0 means "no units on order" whilst NULL might mean "*never* been ordered")

If I wanted to look for all of the Suppliers for which we don't have a fax number, my first attempt would very likely be a comparison operator:

WHERE Fax = NULL

This might or it might not work, depending on the settings on the database I'm talking to.

If I do anything with a *NULL* – add, subtract, concatenate or *compare*, the result is UNKNOWN, which is not the same as true or false, which means it won't be selected. SQL Server can be configured to not work this way.

The correct way to perform a *NULL* test is to use *IS*:

```
WHERE Fax IS NULL
```

As opposed to the equality expression, which *might* work, depending on the system and the settings, the IS NULL operator *will always work*.


Best practices

Best practices

Avoid NOTs

→ Including <>

Use brackets around complex expressions

→ To alter the order of execution of logical operators

→ This also makes them easier to read

Use IN and BETWEEN

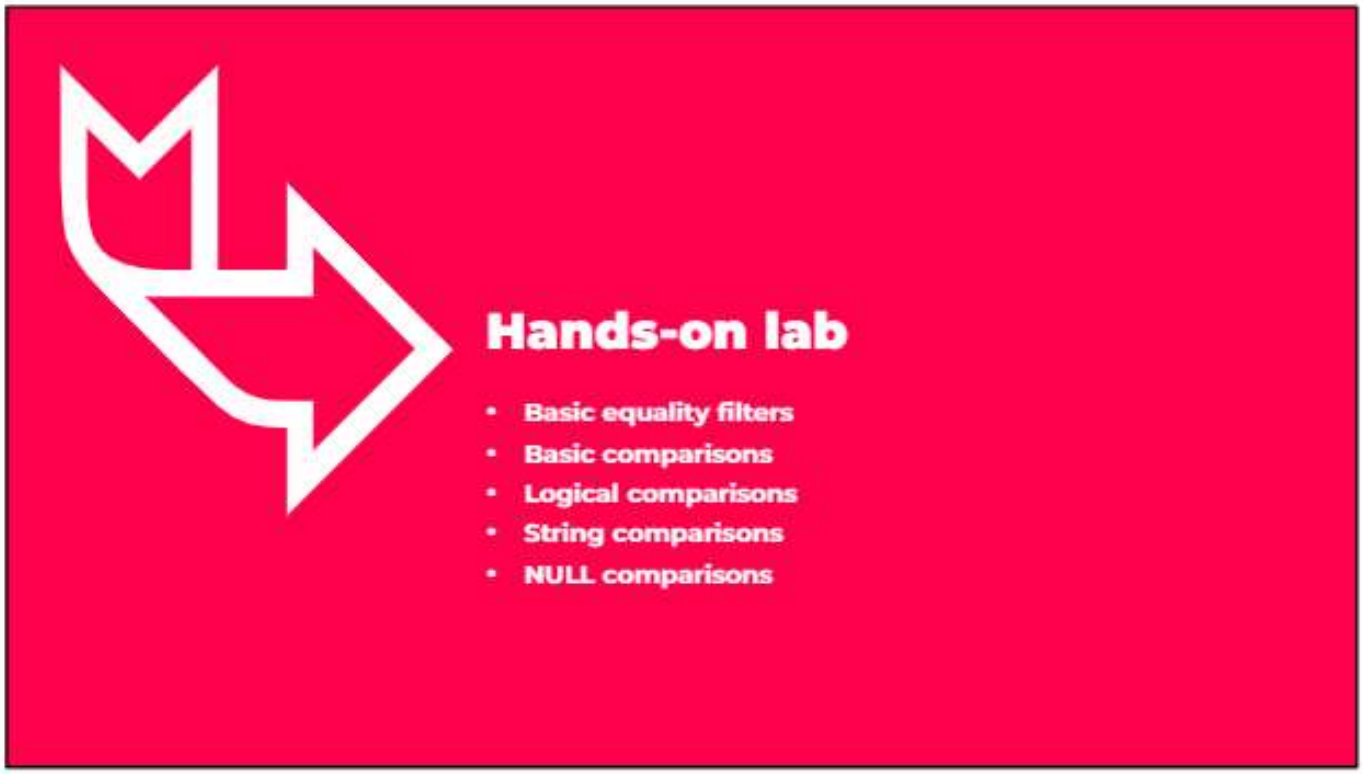
→ Rather than lots of ORs and ANDs

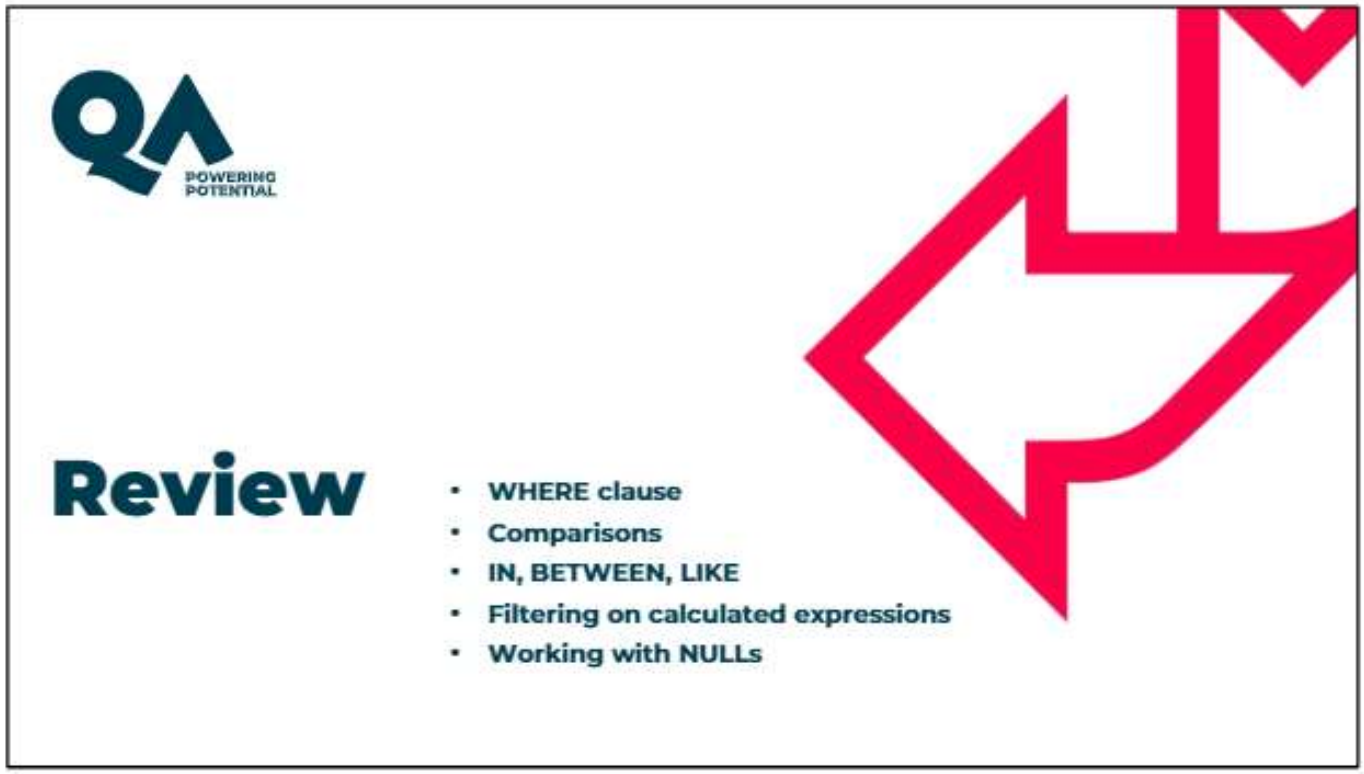
Avoid leading wildcards

→ Inefficient

Use IS NULL

→ Rather than = NULL


Hands-on lab
• Basic equality filters
• Basic comparisons
• Logical comparisons
• String comparisons
• NULL comparisons


POWERING
POTENTIAL
Review
WHERE clause
Comparisons
IN, BETWEEN, LIKE
Filtering on calculated expressions
Working with NULLs


POWERING
POTENTIAL
Sorting Rows
Version 1.5

POWERING POTENTIAL

Overview

- ORDER BY clause
- Sort order
- Sort on multiple columns
- Sort on expressions
- Eliminate duplicates - DISTINCT
- Limit results - TOP
- Hands-on lab

QA

OBJECTIVES

At the end of this module you will be able to:

- → write a query that sorts on a single column
- → write a query that sorts on multiple columns
- → write a query that sorts on a calculated column
- → write a query that selects unique values
- → write a query that returns only a specified number of rows


ORDER BY clause

QA

ORDER BY clause

```
SELECT <<field(s)>>
FROM  <<table(s)>>
WHERE  <<condition(s)>>
GROUP BY  <<field(s)>>
HAVING  <<condition(s)>>
ORDER BY <<field(s)>>
```

The optional *ORDER BY* clause is how we tell SQL to sort the results. Its position, right at the end of the select statement, is accurate.

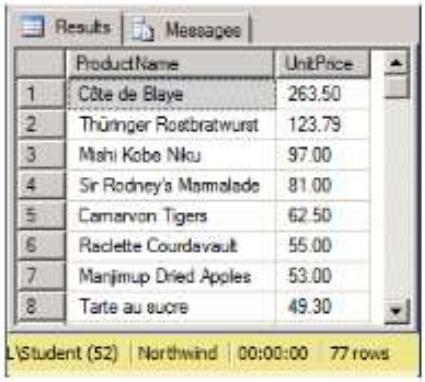
Sorting is the *very last thing* to happen to our result set. All the filtering and calculating and manipulating has been done by the time SQL gets around to sorting the data, because sorting can be very time-consuming; imagine if SQL sorted a million rows from a table before realising that *none* of the rows meet the WHERE clause!


Sort order

Sort order

- **ASC - ascending**
  - Default
- **DESC - descending**

```
SELECT
  ProductName,
  UnitPrice
FROM
  dbo.Products
ORDER BY
  UnitPrice DESC
```

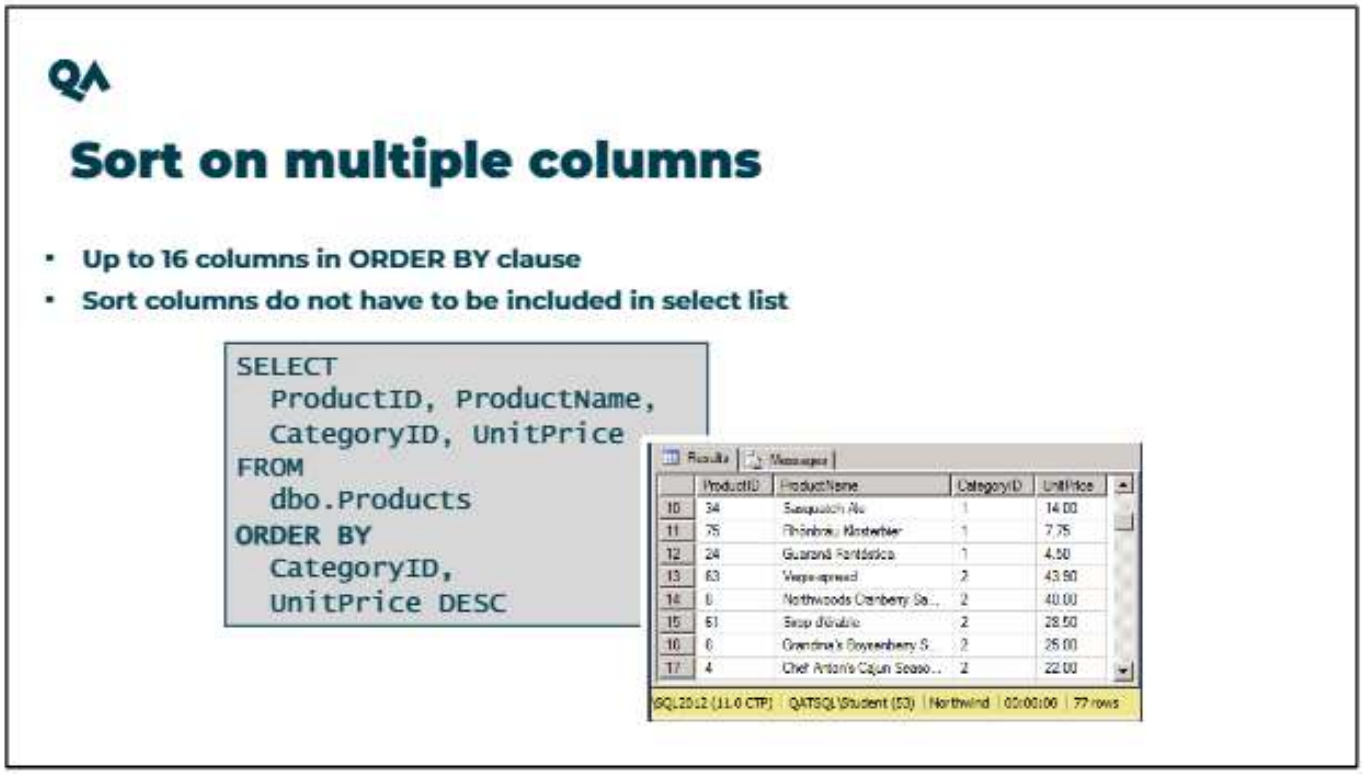


If we do not specify an *ORDER BY* clause, SQL does not guarantee the order in which rows will be returned.

The default sorting rules ("collation settings") for text (language, case sensitivity, accent sensitivity) will have been set up when the server was installed. Usually, in the UK, they are case insensitive and follow the rules of English. Individual databases, tables and even columns can be given alternative collations.

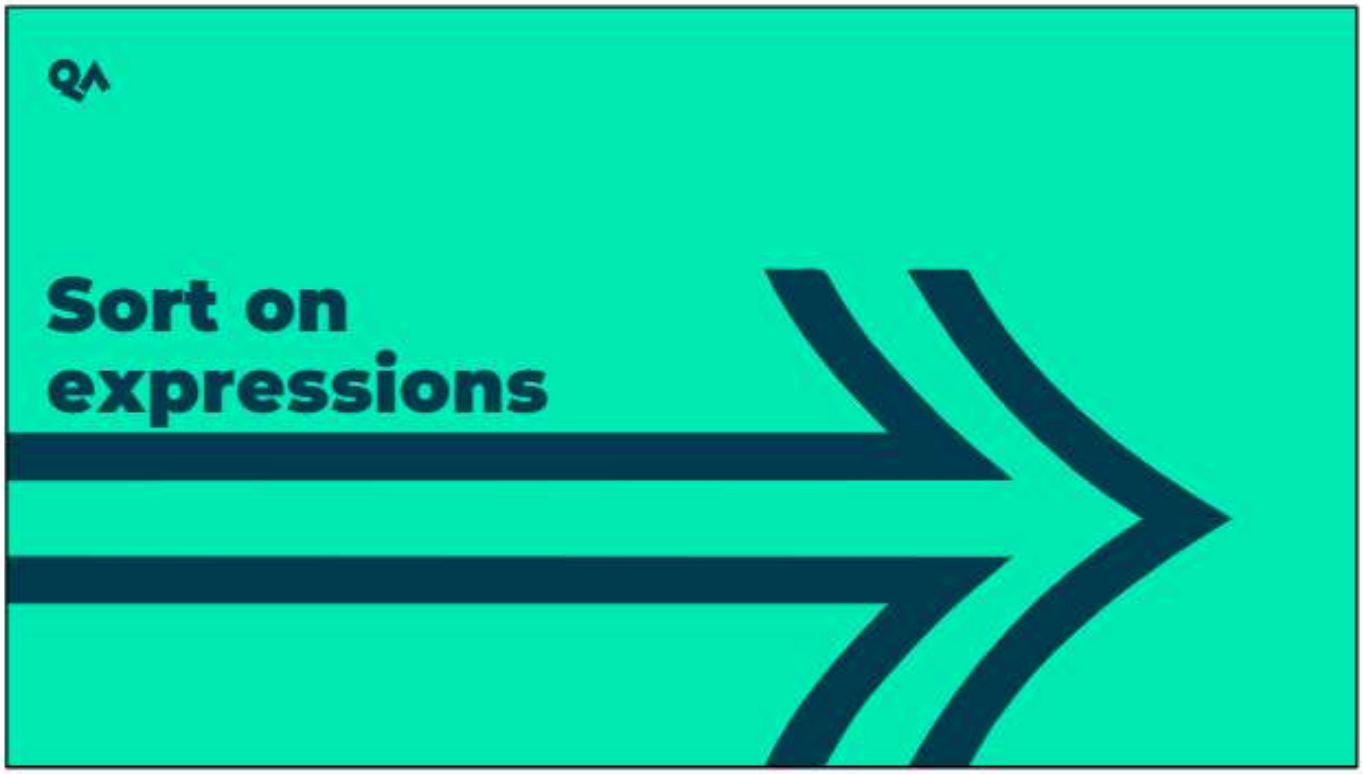
By default, columns are sorted in ascending order (1 to 10, a to z, etc.) but it is possible to tell SQL to sort them in descending order (10 to 1, z to a and so on) using the *DESC* keyword. There is also an *ASC* keyword but it doesn't tend to get used very often...


Sort on multiple columns



We can sort on up to sixteen columns in one ORDER BY clause, using a comma-separated list of column names.

The columns that we're sorting on do not have to be in the list of columns we're selecting out, but it can be confusing for us humans if the sort order isn't obvious. Imagine if the example on the slide didn't include the CategoryId column in the select list – it would be difficult to understand why the results were in that order.


Sort on
expressions

Sort on expressions

→ **ORDER BY**

- <<expression>>
- <<*expression-alias*>>
- <<expression-ordinal>>

```
SELECT
  ProductName, UnitsInStock,
  UnitsOnOrder,
  UnitsInStock + UnitsOnOrder
    AS FutureStock
FROM
  dbo.Products
ORDER BY
  FutureStock DESC
```

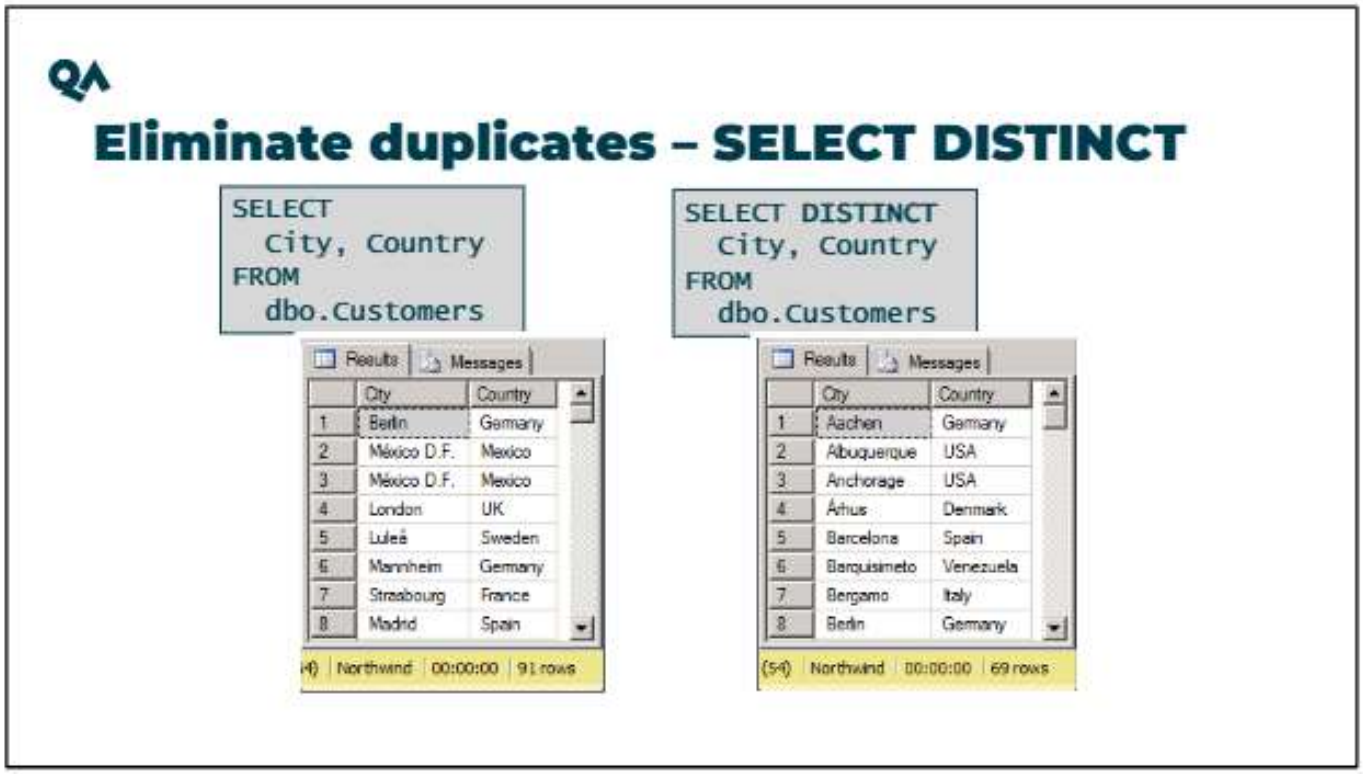
Just like in a where clause, we are not restricted to just columns; we can sort on calculated values as well.

Unlike filtering on expressions, we do not have to duplicate the expression in the ORDER BY clause. Remember, sorting is the very last thing to happen, after all of the rows have been selected and most importantly after all the expressions have been evaluated.

We can use the column alias if there is one or the expression's ordinal number if not.

In fact, we can use column ordinals – column numbers relating to their position from left to right in the result set - to sort on any column, calculated or not, but this is not considered best practice. Firstly, queries using ordinals can be more confusing to read – which column name does "ORDER BY 17" refer to? – plus if you modify the SELECT list your query may no longer return the data sorted in the order you require, or may even return an error, unless you modify the number in the ORDER BY clause.


Eliminate
duplicates -
DISTINCT



If we require a list of unique values, we can use the *DISTINCT* modifier on our SELECT statement to eliminate duplicates. The reason this is being mentioned during the sorting chapter is that SQL usually needs to sort the data to be able to find duplicates. The check for duplicate data is based on all the values in the select list, so a

SELECT DISTINCT CompanyName, City FROM dbo.Customers

would be a fairly pointless exercise in the Northwind database because every customer only appears once in the customers table. If we just wanted to see which customers have at least one address in a given city, then it might make sense. As in:

SELECT DISTINCT CompanyName, City FROM dbo.Customers
WHERE City = 'London'

Limit results - TOP

QA

Limit results – SELECT TOP

- **SELECT TOP *n***
- **SELECT TOP *n* PERCENT**
- **SELECT TOP *n* ... WITH TIES**

```
SELECT TOP 11 WITH TIES
  ProductName,
  UnitPrice
FROM
  dbo.Products
ORDER BY
  UnitPrice DESC
```

Results | Messages

	ProductName	UnitPrice
5	Carnarvon Tigers	62.50
6	Raclette Courdavault	55.00
7	Manjimup Dried Apples	53.00
8	Tarte au sucre	49.30
9	Ipoh Coffee	46.00
10	Rössle Sauerkraut	45.60
11	Schoggi Schokolade	43.90
12	Vegie-spread	43.90

\Student (52) | Northwind | 00:00:00 | 12 rows

We might only want to see a few of the possible rows returned, especially with a sorted query.

SELECT TOP n enables us to answer questions like "what are our 10 most expensive products". We can order our results by price descending and then select just the top 10 results.
We can also specify a percentage of the available rows, so 10% of the rows in a table with 77 rows would give us back 8 rows (the .7 of a row is rounded up).

Finally, we can specify the *WITH TIES* option, which tells SQL to return all the rows that are tied for "last place". Note that WITH TIES requires an ORDER BY clause to work.

It is best practice to only ever use a TOP in conjunction with an ORDER BY clause. This is the only way to predictably indicate which rows are affected by TOP.

Since SQL 2012, if you want to implement a paging solution, the new keyword combination of OFFSET and FETCH should be used. As in:

SELECT ProductName, UnitPrice FROM dbo.Products ORDER BY UnitPrice DESC

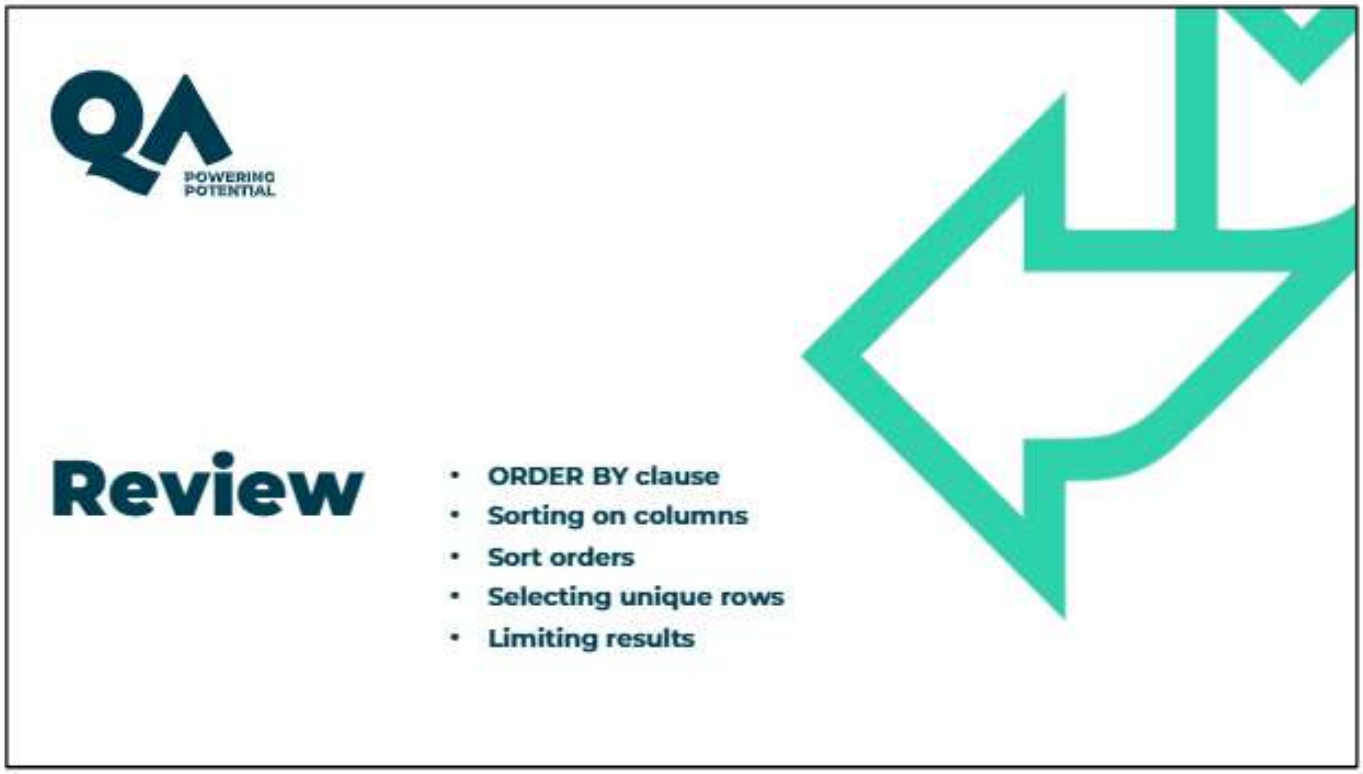
OFFSET 0 ROWS **FETCH** NEXT 10 ROWS ONLY

Then, to return the next 10 rows the following would be used:

SELECT ProductName, UnitPrice FROM dbo.Products ORDER BY UnitPrice DESC
OFFSET 10 ROWS **FETCH** NEXT 10 ROWS ONLY

Hands-on lab

- Basic sorting
- Sorting on expressions
- Unique rows
- Limiting results


POWERING
POTENTIAL
Review
ORDER BY clause
Sorting on columns
Sort orders
Selecting unique rows
Limiting results


POWERING
POTENTIAL
Grouping and
Aggregating
Data
Version 1.5


Overview
• Aggregate functions
• GROUP BY clause
• HAVING clause
• Hands-on lab

QA

OBJECTIVES

At the end of this module you will be able to:

- write a query that calculates single aggregates
- write a query that calculates aggregates for grouped information
- write a query that aggregates, groups and filters on calculated values


Aggregate
functions

QA

Aggregate functions

- **COUNT(*)**
- **COUNT(**<<*expression*>>**)**
- **SUM(**<<*expression*>>**)**
- **AVG(**<<*expression*>>**)**
- **MIN(**<<*expression*>>**) / MAX(**<<*expression*>>**)**
- **Statistical aggregates**
  → STDEV / STDEVP / VAR / VARP

5

You've already seen that we can ask SQL to perform calculations on a row-by-row basis.

We can also ask SQL to perform mathematical operations over a set of records.

With the exception of *COUNT(*)*, all the functions operate only on non-null values for *expression*.

For instance, given the following data:

Product	**Quantity**
A	250
B	0

The *COUNT*(Quantity) would be 2 and the *AVG*(Quantity) would be 125, whilst for the following data:

Product	**Quantity**
A	250
B	*null*

The *COUNT*(Quantity) would be 1 and the *AVG*(Quantity) would be 250.

COUNT(*) in both cases would return 2.


GROUP BY clause

QA

GROUP BY clause

- SELECT <<field(s)>>
- FROM <<table(s)>>
- WHERE <<condition(s)>>
- **GROUP BY** *<<field(s)>>*
- HAVING *<<condition(s)>>*
- ORDER BY <<field(s)>>

Aggregate functions calculate a single value for all the rows in a column.

We use GROUP BY to tell SQL that we wish to calculate the aggregate for each of a number of values in another column.

Grouping aggregates

```
SELECT
  CategoryID,
  AVG(UnitPrice)
FROM
  dbo.Products
GROUP BY
  CategoryID
```

Results | Messages

	CategoryID	(No column name)
1	1	37.9791
2	2	23.0625
3	3	25.16
4	4	28.73
5	5	20.25
6	6	54.0066
7	7	32.37
8	8	20.6825

ent (55) | Northwind | 00:00:00 | 8 rows

```
SELECT
  CategoryID,
  AVG(UnitPrice)
FROM
  dbo.Products
GROUP BY
  CategoryID
ORDER BY 2 DESC
```

Results | Messages

	CategoryID	(No column name)
1	6	54.0066
2	1	37.9791
3	7	32.37
4	4	28.73
5	3	25.16
6	2	23.0625
7	8	20.6825
8	5	20.25

Aggregate functions calculate a single value for all the rows in a column.

We use *GROUP BY* to tell SQL that we wish to calculate the aggregate for each of the values in another column.

In fact, SQL will not let us include anything in the select list that isn't an aggregate unless it is also in the *GROUP BY* clause.

Effectively *GROUP BY* tells SQL to generate a unique list of the values in the *GROUP BY* column(s) using a DISTINCT, and then to calculate the aggregate for all of the rows with that value using a WHERE.

We can use an ORDER BY to sort the results and we should probably alias the aggregate column(s)

If we use a *GROUP BY* on a column with null values, the null values are treated as a group


HAVING clause

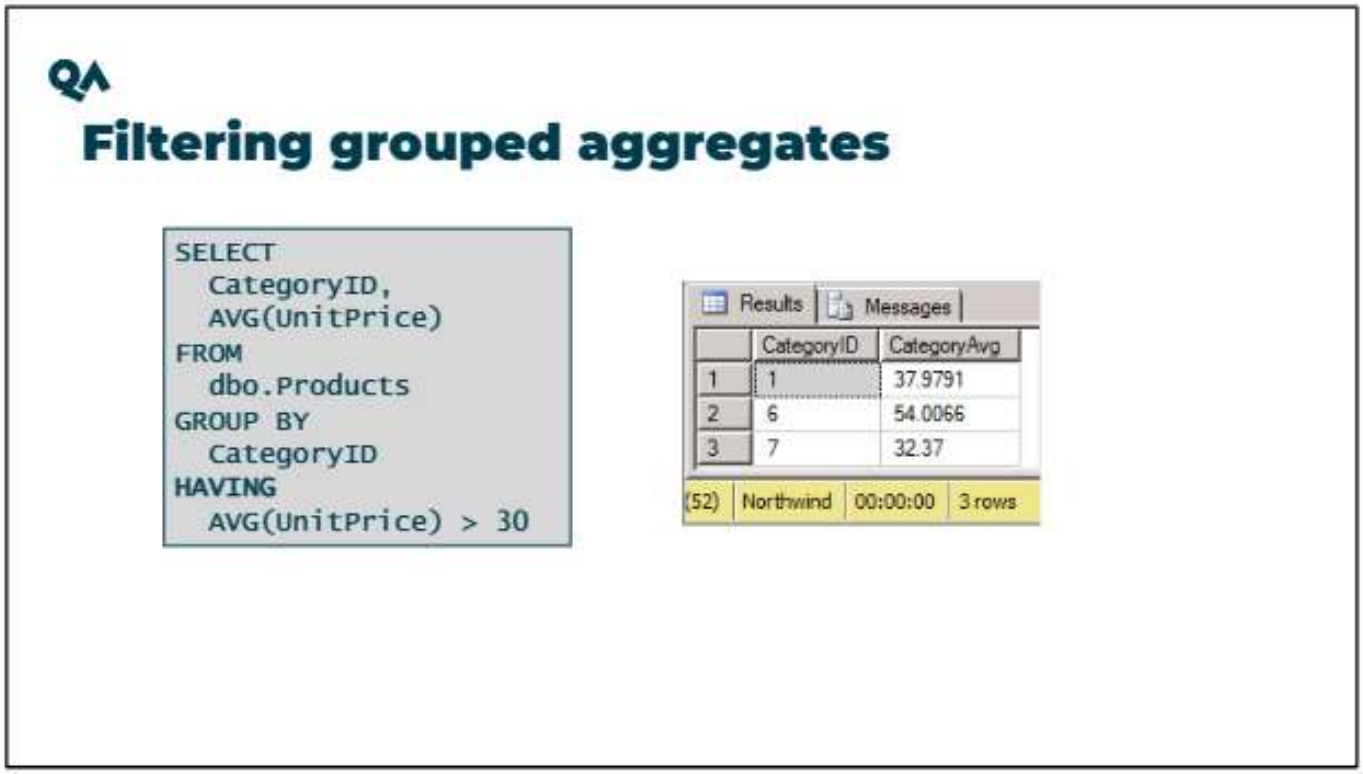
QA

HAVING clause

- SELECT <<field(s)>>
- FROM <<table(s)>>
- WHERE *<<row-level filter(s)>>*
- **GROUP BY** *<<field(s)>>*
- **HAVING** *<<group-level filters(s)>>*
- ORDER BY <<field(s)>>

If we want to filter the grouped aggregates, we need to use a *HAVING* clause.

Remember, the WHERE clause applies at the row level. We now want to wait until the aggregates have been calculated before performing the filter.

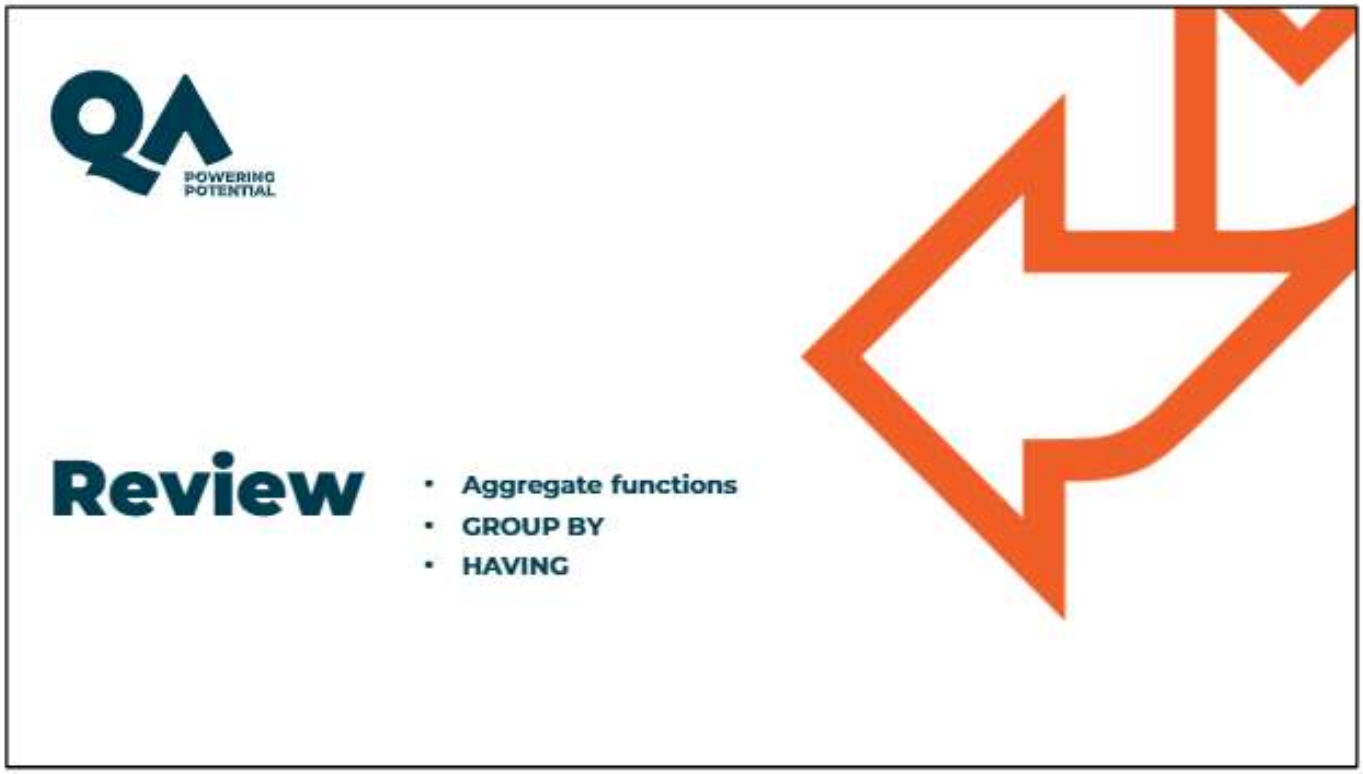


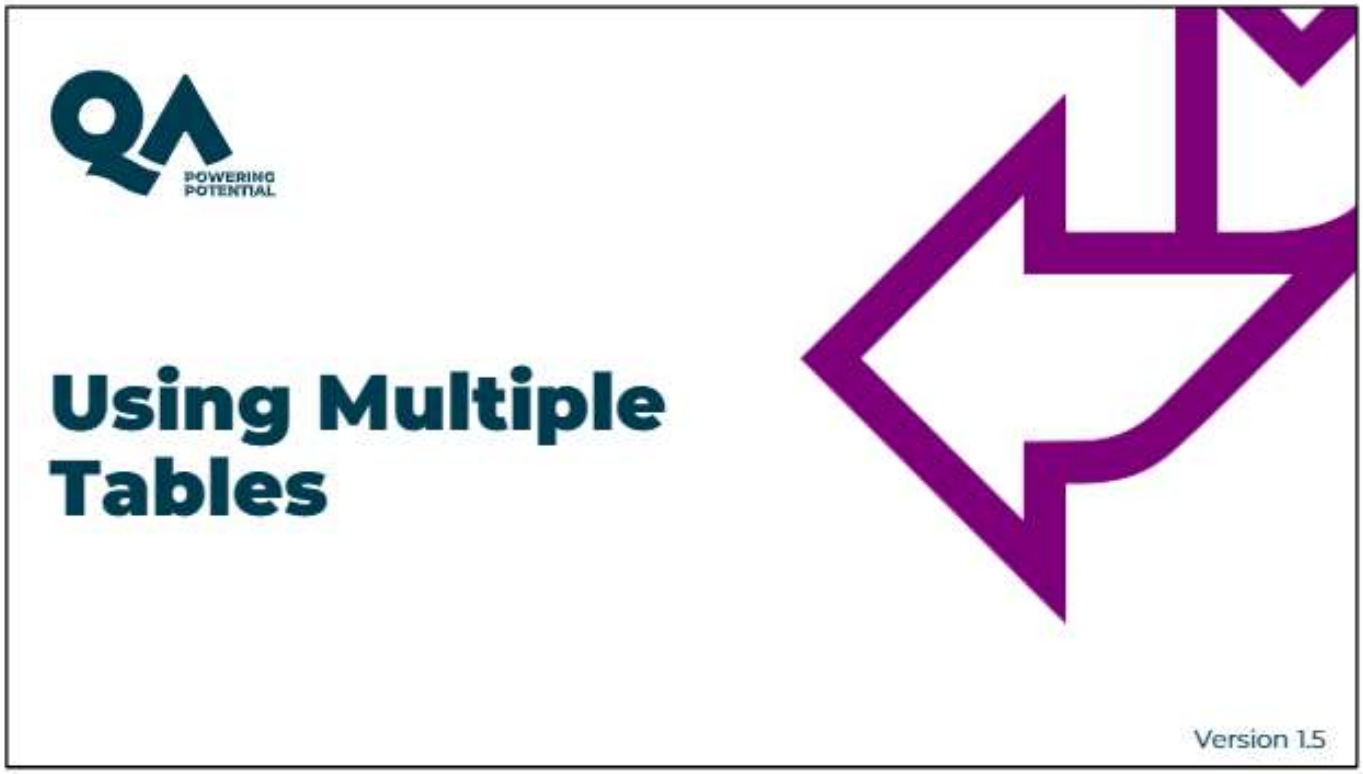
You can use the *HAVING* clause on any of the columns in the select list.

HAVING doesn't make any sense if you don't have a GROUP BY clause. In fact, if you try to use a HAVING clause without a GROUP BY, you will get an error message

Much like a WHERE clause, if we are filtering on the result of an expression, we need to duplicate that expression in the HAVING clause.


Hands-on lab
• Single aggregates
• GROUP BY
• HAVING


POWERING
POTENTIAL
Review
Aggregate functions
GROUP BY
HAVING


POWERING
POTENTIAL
Using Multiple Tables
Version 1.5


Overview
• JOINs
• Join types
• UNION, EXCEPT and INTERSECT
• Hands-on lab

OBJECTIVES

At the end of this module you will be able to:

- → write a query that joins two tables together
- → write a query that joins multiple tables
- → write a query that uses an OUTER JOIN
- → write a query that uses a UNION operator to join tables


JOINS

QA

JOINs

```
SELECT <<field(s)>>
FROM  <<table1>>
type-of JOIN <<table2>> ON <<predicate>>
WHERE  <<condition(s)>>
GROUP BY  <<field(s)>>
HAVING  <<condition(s)>>
ORDER BY <<field(s)>>
```

All of the queries we've written so far in this course have been against single tables.

In the wild, we very rarely actually want to run such queries, pulling in information from various tables in the database.

In fact, this is the whole point of Relational Database Theory, the idea of putting data into many different tables to improve the speed of updating and inserting data and also to reduce the amount of disk space used by databases. For querying, this data then needs to be pulled back together again.


Join Types

QA

Join Types

JOINs

- → Inner
- → Outer (right, left and full)

There are two techniques for joining data together – JOINs, which will be the main focus of this chapter, and Set-based operations.

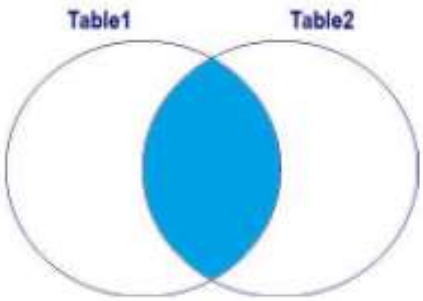
Within JOINs, there are three types: *inner join*s, *outer join*s and *cross join*s. Of the three, this course will only concern itself with the first two as cross joins are rarely, if ever, useful.

QA

Inner join

- **Returns columns from both tables where the ON clause is true**

```
→ SELECT *
→       FROM Table1 INNER JOIN Table2
→                ON Table1.Col1 = Table2.Col2
```



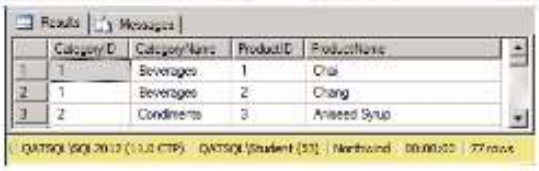
Using two tables the *INNER JOIN* returns columns from both tables where the *ON* clause is satisfied. This is the most common type of join.

Used for joining tables previously split during the design phase into related tables, usually using a primary key – foreign key relationship.

Inner join

```
SELECT
  Products.CategoryID,
  CategoryName,
  ProductID,
  ProductName
FROM
  dbo.Products
INNER JOIN
  dbo.Categories
ON
  Products.CategoryID =
  Categories.CategoryID
```

```
SELECT
  p.CategoryID,
  CategoryName,
  ProductID,
  ProductName
FROM
  dbo.Products AS p
JOIN
  dbo.Categories AS c
ON
  p.CategoryID =
  c.CategoryID
```



When joining tables, we often use a column from one table (in this case the CategoryID column in the Products table) that has exactly the same name as a column in another table (in this case the CategoryID column in the Categories table). From the point of view of someone talking to the database, this makes perfect sense and actually makes the database's structure easier to understand – clearly there is a relationship between products and categories and that relationship is defined via a CategoryID column. Unfortunately, computers are still nowhere near as clever as us humans and so SQL gets confused in both our select list and our *ON* predicate if there are columns with the same names in more than one table. So we need to qualify these "ambiguous" column names with the name of the table, which can become time-consuming.

This is where the ability to alias table names as well as column names starts to become useful – we can use a simple prefix to disambiguate the columns without making the query too much harder to read.

NOTE [1]: *INNER JOIN* is the default type of join, so we can leave out the *INNER* part in our queries.

NOTE [2]: It's only the ambiguous column names that we *must* use the table name

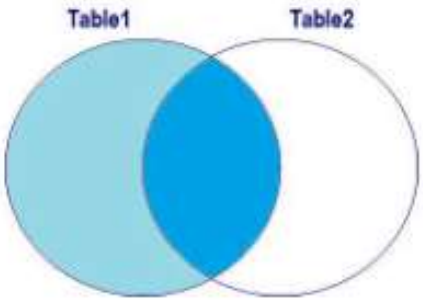
or alias for, but we often use it for every column, especially in queries that join more than two tables together, to make it easier to see at a glance which columns are coming from which tables. The query on the slide loses nothing and possibly gains readability, rewritten as:

```
SELECT p.CategoryID, c.CategoryName, p.ProductID, p.ProductName
FROM dbo.Products AS p INNER JOIN dbo.Categories AS c
ON p.CategoryID = c.CategoryID
```

QA

Left (outer) Join

- Returns columns from both tables where the ON clause is true and columns from Table1 in all cases




```
→ SELECT *
→       FROM Table1 LEFT JOIN Table2
→                ON Table1.Col1 = Table2.Col2
```

Using two tables the *LEFT JOIN* returns columns from both tables where the *ON* clause is satisfied and also the columns from the first table where it is not satisfied.

Used for joining tables previously split during the design phase into related tables, usually using a primary key – foreign key relationship, and returning the mismatches as well.

Left join

```
SELECT
  o.OrderID,
  o.OrderDate,
  c.CompanyName
FROM
  dbo.Customers AS c
LEFT JOIN
  dbo.Orders AS o
ON
  c.CustomerID =
    o.CustomerID
```

Results | Messages

	OrderID	OrderDate	CompanyName
826	11073	1998-05-05 00:00:00.000	Pericles Comidas clásicas
827	11074	1998-05-06 00:00:00.000	Simons bistro
828	11075	1998-05-06 00:00:00.000	Richter Supermarkt
829	11076	1998-05-06 00:00:00.000	Bon app'
830	11077	[illegible]	[illegible] Canyon Grocery
[illegible]	NULL	NULL	Paris spécialités
832	NULL	NULL	FISSA Fabrica Inter. Salchi...

(16.0 RTM) | northwind | 00:00:00 | 832 rows

In the example on the slide, we are running a query to get a list of all OrderIDs, their dates, and the companyname of the customers who placed the orders. If we had used an *INNER JOIN*, we would have got 830 rows back because there are 2 customers who have not placed any orders.

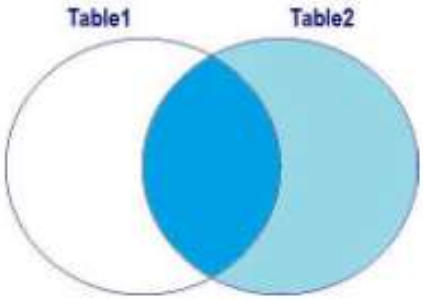
Instead, we use a *LEFT JOIN* to ask for **all** of the records in the Customers table, plus those records in the Orders table which have a match. Rows in the right-hand table without a match show up as *null*s.

NOTE: Strictly speaking, a *LEFT JOIN* is a *LEFT* OUTER *JOIN* but the *OUTER* is implicit and therefore optional.

QA

Right Join

- Returns columns from both tables where the ON clause is true and columns from Table2 in all cases




```
→ SELECT *
→       FROM Table1 RIGHT JOIN Table2
→               ON Table1.Col1 = Table2.Col2
```

Using two tables the *RIGHT JOIN* returns columns from both tables where the *ON* clause is satisfied and also the columns from the second table where it is not satisfied.

Used for joining tables previously split during the design phase into related tables, usually using a primary key – foreign key relationship, and returning the mismatches as well.

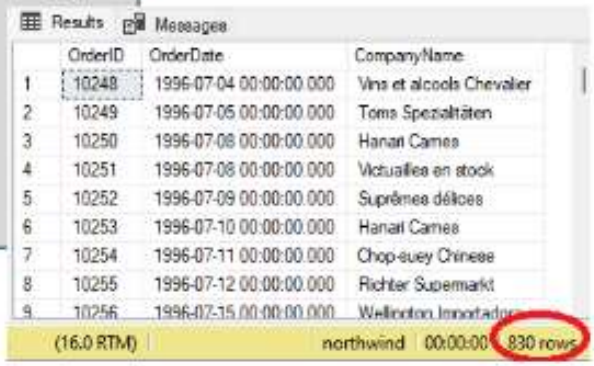
If you're thinking that looks a lot like the definition of a *LEFT JOIN*, you're absolutely correct. The only difference between the two is which side of the *JOIN* keyword the table is. The one before the keyword is the left-hand table and the one after is the right-hand table. All *LEFT JOIN*s can be rewritten as *RIGHT JOIN*s and vice versa.

It's usually just a matter of which table you start thinking about first.

A *LEFT JOIN* can have null values in columns from the right-hand table whereas a *RIGHT JOIN* can have null values in columns from the left-hand table.

Right join

```
SELECT
  o.OrderID,
  o.OrderDate,
  c.CompanyName
FROM
  dbo.Customers AS c
RIGHT JOIN
  dbo.Orders AS o
ON
  c.CustomerID =
    o.CustomerID
```



In the example on the slide, we have simply changed the LEFT keyword from the previous example into a RIGHT. The result set only contains 830 rows because we are asking for the rows in the Customers table that have corresponding entries in the Orders table, so any customers who have not placed orders will no longer exist.

To recrate the results of the previous example with the code above, either change the word RIGHT to read LEFT, or switch the order of the dbo.Customers and dbo.Orders tables in the FROM and JOIN clauses as follows:

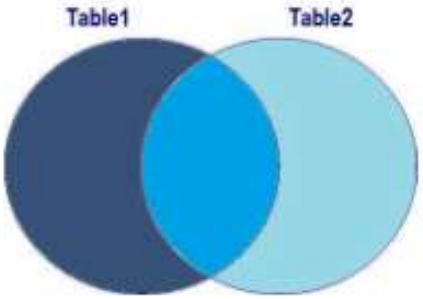
```
SELECT
  o.OrderID,
  o.OrderDate,
  c.CompanyName
FROM
  dbo.Orders AS o
RIGHT JOIN
  dbo.Customers AS c
ON
  c.CustomerID =
```

o.CustomerID

QA

Full Join

- **Returns columns from both tables where the ON clause is true and disjointed rows where matches are not found**




```
SELECT *
    FROM Table1 FULL JOIN Table2
        ON Table1.Col1 = Table2.Col2
```

Using two tables a *FULL JOIN* returns columns from both tables where the ON clause is satisfied, and also the columns from both tables separately where it is not satisfied

Commonly used for examining two versions of the same table to find mismatches.

So if a *LEFT JOIN* can have *nulls* in the right-hand table's columns and a *RIGHT JOIN* can have *nulls* in the left-hand table's columns then a *FULL JOIN* can have *nulls* on both sides.

Joining more than two tables

```
SELECT
o.OrderID, o.OrderDate,
c.CompanyName, e.FirstName, e.LastName
      FROM dbo.Customers AS c
LEFT JOIN dbo.Orders AS o
      ON c.CustomerID = o.CustomerID
LEFT JOIN dbo.Employees AS e ON e.EmployeeID =
o.EmployeeID
```

Results | Messages

	OrderID	OrderDate	CompanyName	FirstName	LastName
509	10833	1998-01-15 00:00:00.000	Ottilies Käseladen	Michael	Suyama
510	10999	1998-04-03 00:00:00.000	Ottilies Käseladen	Michael	Suyama
511	11020	[illegible]	Ottilies Käseladen	[illegible]	Fuller
512	NULL	NULL	Paris spécialtés	NULL	NULL
513	10322	[illegible]	Pericles Comidas clásicas	Robert	King
514	10354	1996-11-14 00:00:00.000	Pericles Comidas clásicas	Laura	Callahan
515	10474	1997-03-13 00:00:00.000	Pericles Comidas clásicas	Steven	Buchanan
516	10502	1997-04-10 00:00:00.000	Pericles Comidas clásicas	Andrew	Fuller

Query executed successf... (16.0 RTM) northwind 00:00:00 832 rows

We can of course join as many tables together as we like, as long as we have some way of telling SQL how to find the rows we're interested in.

In the example on the slide, we're completing the query on Customers and their Orders by adding another join to the earlier examples. We are in effect joining the Employees table to the combined Customers & Orders result set, so we need to use a *LEFT JOIN* to make sure we see all 832 rows, plus any employees we can find. Obviously, any rows that have a *null* for OrderID and OrderDate from the Orders table are going to have *null*s for employee names.

Joining a table to itself

```
SELECT
  emp.FirstName, emp.LastName,
  mgr.FirstName + ' ' + mgr.LastName
    AS Boss
FROM
  dbo.Employees AS emp
JOIN
  dbo.Employees AS mgr
ON
  emp.ReportsTo =
    mgr.EmployeeID
```

Results | Messages

	FirstName	LastName	Boss
1	Nancy	Davolio	Andrew Fuller
2	Janet	Leverling	Andrew Fuller
3	Margaret	Peacock	Andrew Fuller
4	Steven	Buchanan	Andrew Fuller
5	Michael	Suyama	Steven Buchanan
6	Robert	King	Steven Buchanan
7	Laura	Callahan	Andrew Fuller
8	Anne	Dodsworth	Steven Buchanan

QATSQL\Student (57) | Northwind | 00:00:00 | 8 rows

Some tables can contain columns that refer back to the same table. A classic example is in the Northwind database's Employees table, where there is a ReportsTo column which contains the EmployeeID of the employee's manager.

If we want to write a query that lists employees and their managers, we will need to join that table to itself. In these circumstances, we will have to use aliases for the table names to avoid ambiguity.

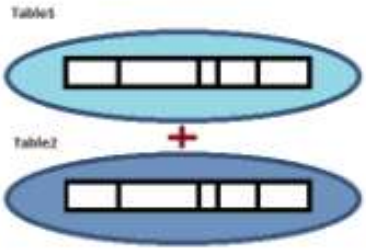
In the example on the slide, a keen observer might notice that there is an employee missing.


UNION, EXCEPT
and INTERSECT

QA

Union [All]

- Accumulation of both sets
- Duplicates may occur with ALL




```
SELECT * FROM Table1
UNION
SELECT * FROM Table2
```

Using two tables a *UNION* returns rows from both tables. If duplicates exist in the two tables they are removed unless we specify *UNION ALL*. Columns in both select lists must match in quantity and data type (or the data types must be convertible by SQL)
The query below returns the ProductID of all products that have either total sales order quantity greater than 500 or a list price greater than 150.

```
SELECT ProductID
FROM dbo.[Order Details]
GROUP BY ProductID
HAVING SUM(Quantity) > 700
UNION
SELECT          ProductID
FROM dbo.Products
WHERE UnitPrice > 100
```

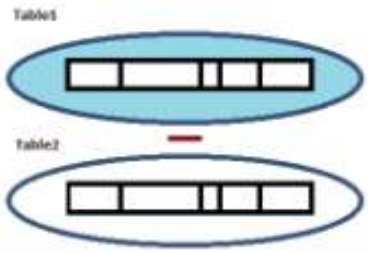
This query should return 39 rows in ProductID order. With a UNION ALL instead, it will return 40 rows in no particular order.

Within this group of operations, *except*s and *intersect*s are more often written using WHERE clauses while *union*s can be very useful.

QA

Except

- Rows from Table1 where they do not exist in Table2




```
SELECT * FROM Table1
EXCEPT
SELECT * FROM Table2
```

Using two tables the *EXCEPT* returns rows that exist in the first table but not in the second. As with a UNION, columns in the select list must match in quantity and data type. It is something like a set-based subtraction.

The query below returns the ProductID of all products that with a list price greater than 100 that *have not* sold more than 700 units:

```
SELECT ProductID
FROM dbo.Products
WHERE UnitPrice > 100
EXCEPT
SELECT          ProductID
FROM dbo.[Order Details]
GROUP BY ProductID
HAVING SUM(Quantity) > 700
```

This query should return 1 row – ProductID 38. It could be rewritten using joins, which would also enable us to get more columns back in the results:

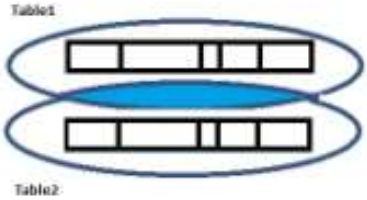
```
SELECT p.ProductID, p.UnitPrice, SUM(Quantity) AS TotalSold
```

```
FROM dbo.Products AS p
JOIN dbo.[Order Details] AS od
ON p.ProductID = od.ProductID
WHERE p.UnitPrice > 100
GROUP BY p.ProductID, p.UnitPrice
HAVING SUM(Quantity) <= 700
```

QA

Intersect

- **Rows that exist in both sets will be returned**



SELECT * FROM Table1

INTERSECT

SELECT * FROM Table2

Using two tables the *INTERSECT* returns rows that exist in both tables. Columns in the select list must match in quantity and data type.

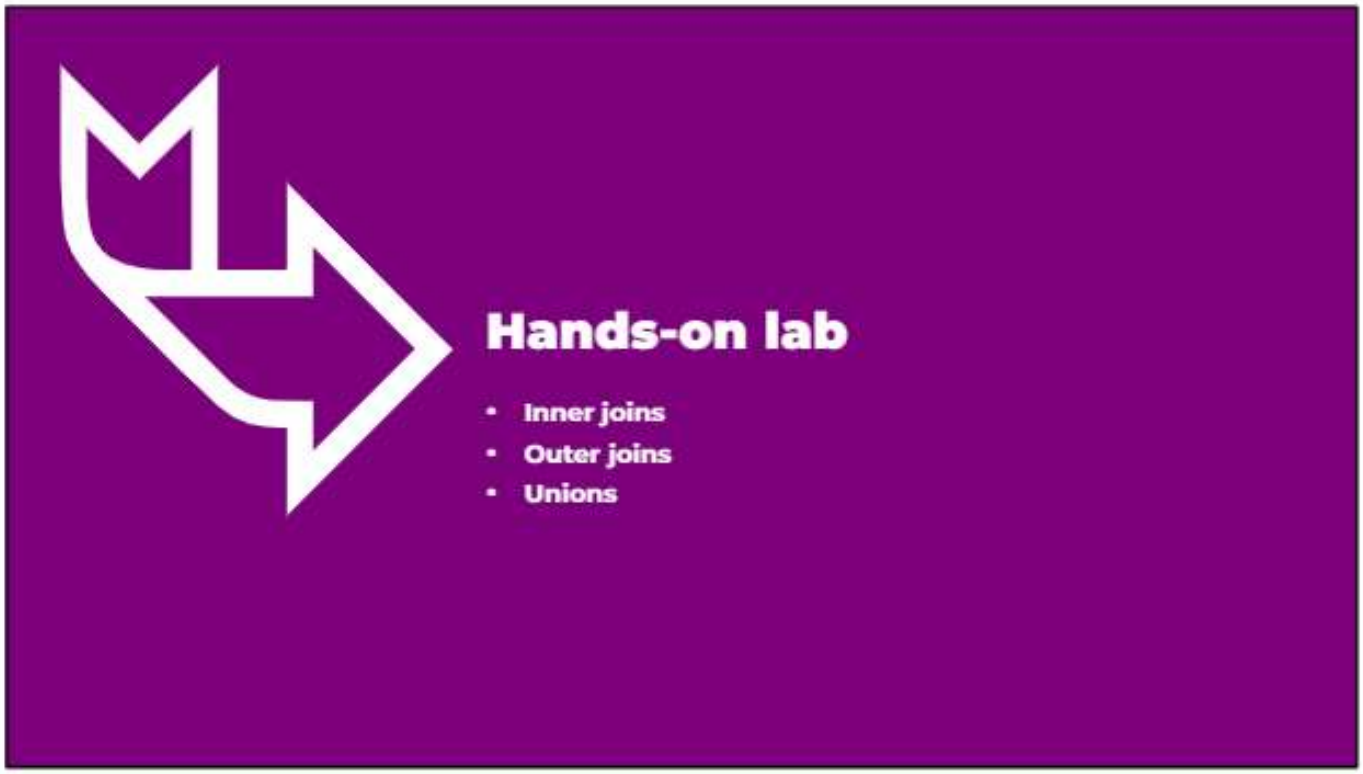
The query below returns the ProductID of all products that with a list price greater than 100 that *have* sold more than 700 units:

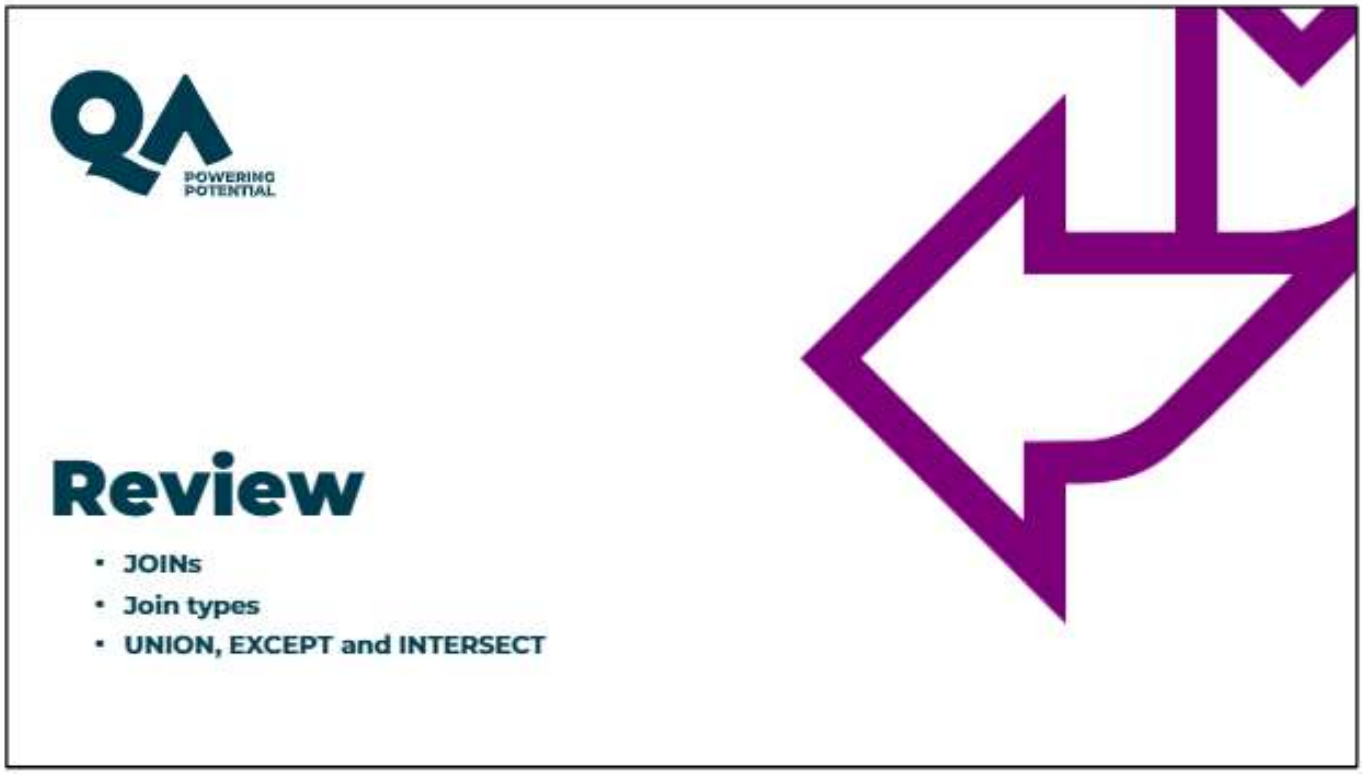
```
SELECT ProductID
FROM dbo.Products
WHERE UnitPrice > 100
INTERSECT
SELECT          ProductID
FROM dbo.[Order Details]
GROUP BY ProductID
HAVING SUM(Quantity) > 700
```

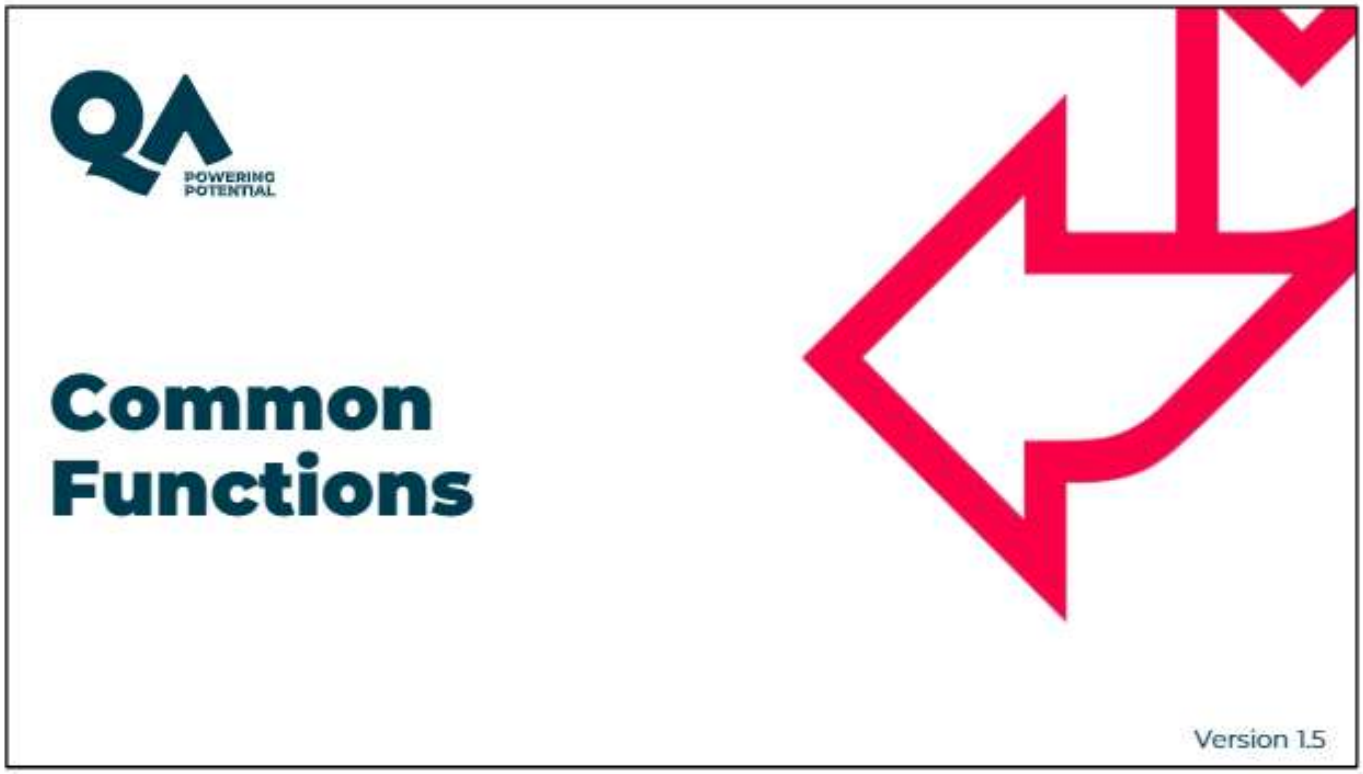
This query should return 1 row – ProductID 29. It could be rewritten using joins, which would also enable us to get more columns back in the results:

```
SELECT p.ProductID, p.UnitPrice, SUM(Quantity) AS TotalSold
FROM dbo.Products AS p
```

```
JOIN dbo.[Order Details] AS od
ON p.ProductID = od.ProductID
WHERE p.UnitPrice > 100
GROUP BY p.ProductID, p.UnitPrice
HAVING SUM(Quantity) > 700
```


Hands-on lab
Inner joins
Outer joins
Unions


POWERING
POTENTIAL
Review
• JOINs
• Join types
• UNION, EXCEPT and INTERSECT


POWERING
POTENTIAL
Common
Functions
Version 1.5

Overview

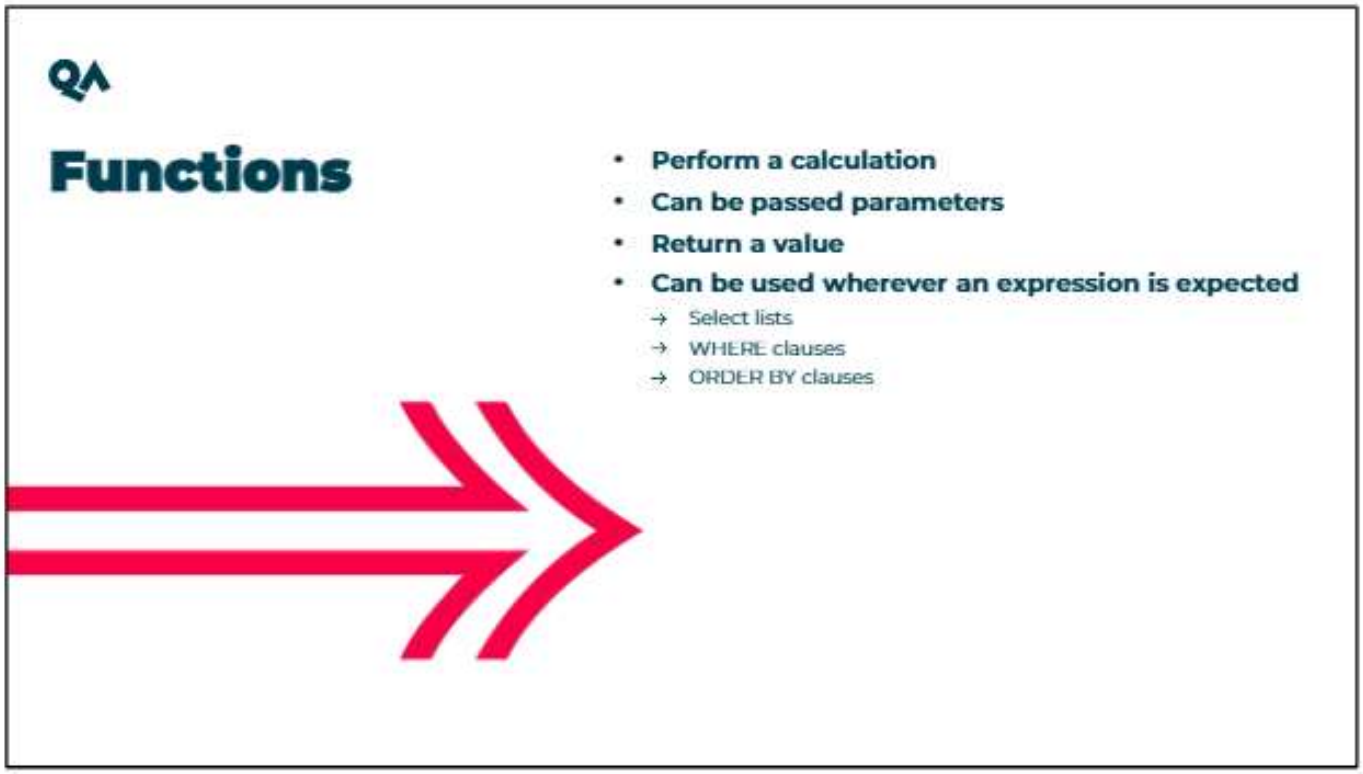
- Functions
- Text functions
- Date functions
- Working with nulls
- Data conversion functions
- Hands-on lab

OBJECTIVES

At the end of this module you will be able to:

- use text manipulation functions
- use date manipulation functions
- use the ISNULL function
- format dates


Functions



Functions are stored sets of SQL statements that return a result.

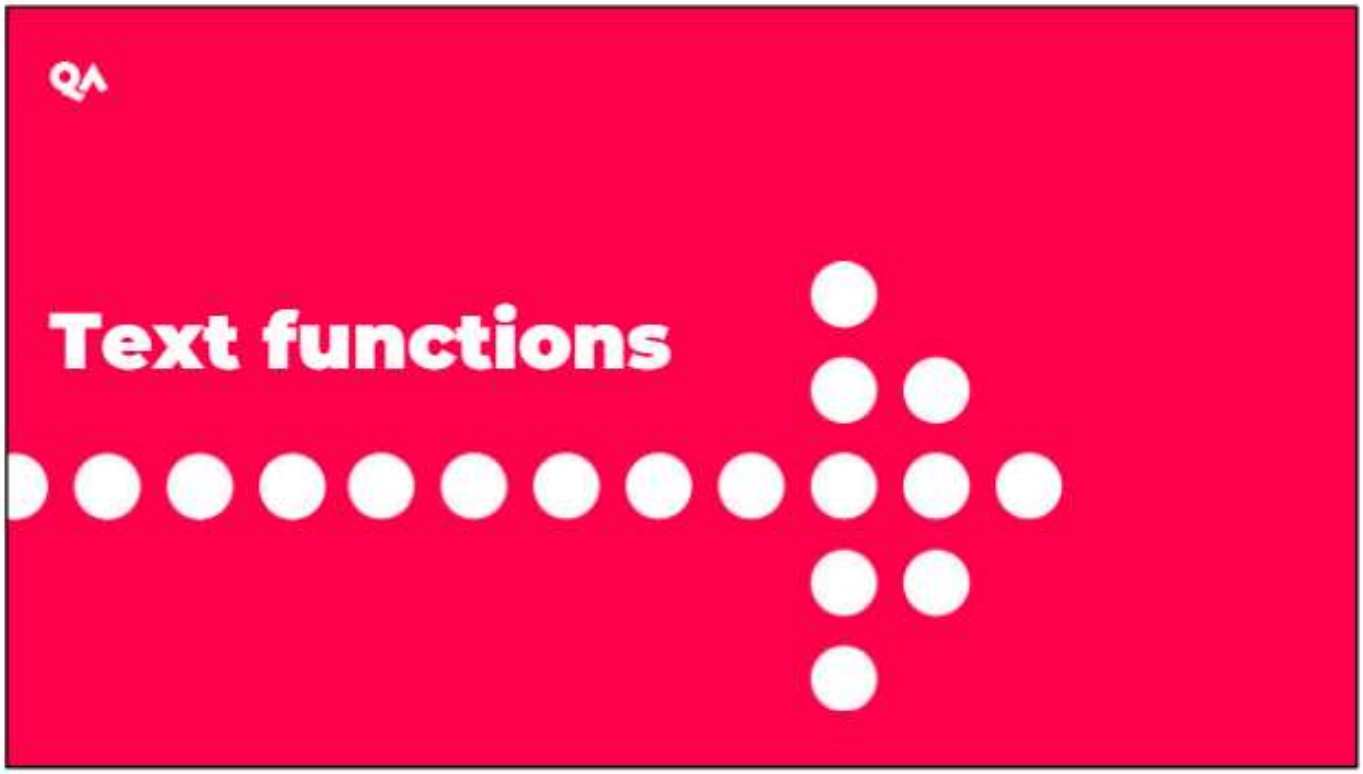
Most functions are referred to as "scalar" functions and return a single value but some functions, known as "table-valued functions" (TVFs) return tables of data and work like parameterised views. This chapter focuses on scalar functions. TVFs are beyond the scope of this course.

Functions usually require parameters to be passed to them, which normally include the name of the column that we want to perform the calculation against and possibly some other information.

We can use them anywhere that SQL is expecting an expression, that is to say in select lists, WHERE clauses, ORDER BY clauses, etc.

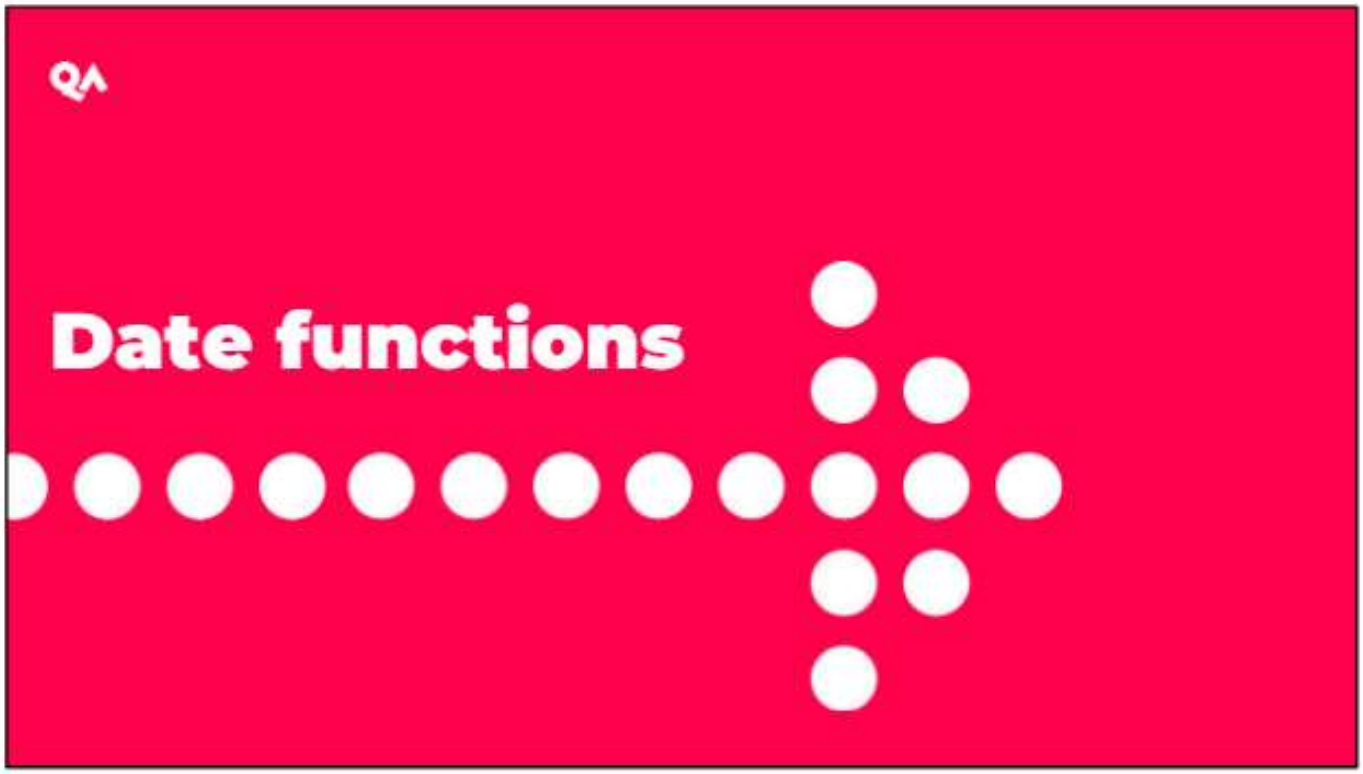
Unlike stored procedures, we don't need to tell SQL to EXECUTE a function but SQL does need to be able to work out that we are calling one, so the function's name is always followed by an open parenthesis "(", then the parameters we're passing, then a close parenthesis ")", *even if we don't have any parameters to pass to it*. The brackets are **not** optional.

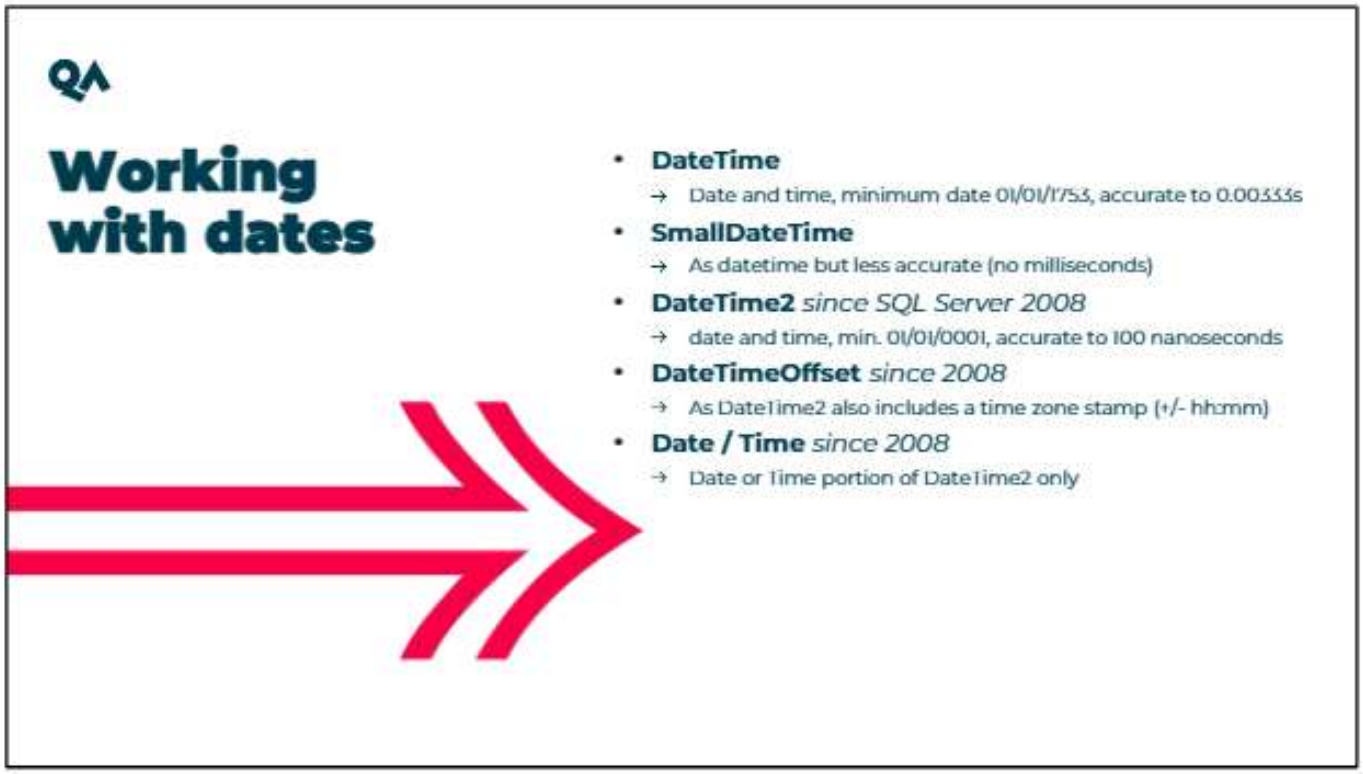
The following few pages list and discuss some of the more commonly-used functions. There are many, many more available in SQL, and more are added with each release.


Text functions

Text functions

- **Left / Right *(expr, n)***
  - → return the left / right -most *n* characters of *expr*
- **Upper / Lower *(expr)***
  - → Convert *expr* to all UPPERCASE or all lowercase letters
- **SubString *(expr, start, length)***
  - → Take *length* letters from *expr*, starting from *start*
- **CharIndex / PatIndex *(look-for, expr, start)***
  - → CharIndex looks for an exact match on *look-for* in *expr*, optionally starting at *start*
  - → PatIndex performs pattern matching similar to LIKE


Date functions



Before SQL Server 2008, we only had one kind of date-based data type in SQL, the datetime (and smalldatetime). Since SQL Server 2008, Microsoft added some new date data types:

Type	Purpose and Range
DateTime	date and time, min. 01/01/1753, accurate to 0.003 seconds
SmallDateTime	date and time, min. 01/01/1900, no milliseconds
DateTime2	date / time, min. 01/01/0001, accurate to 100 nanoseconds
DateTimeOffset	date / time, with a time zone stamp (+/- hours)
Date	date only, minimum 01/01/0001
Time	time only, zero to 23:59:59.9999999

A discussion of dates in SQL would not be complete without a mention of formats – SQL Server is (in fact, most database products are) American-built, which means its default date format is month / day / year, which catches everyone out most of the time. It is possible to change the default date format, but the best habit to get into is to specify dates in YYYYMMDD, which should be unambiguous:

```
WHERE Employees.OrderDate = '19230823'
```

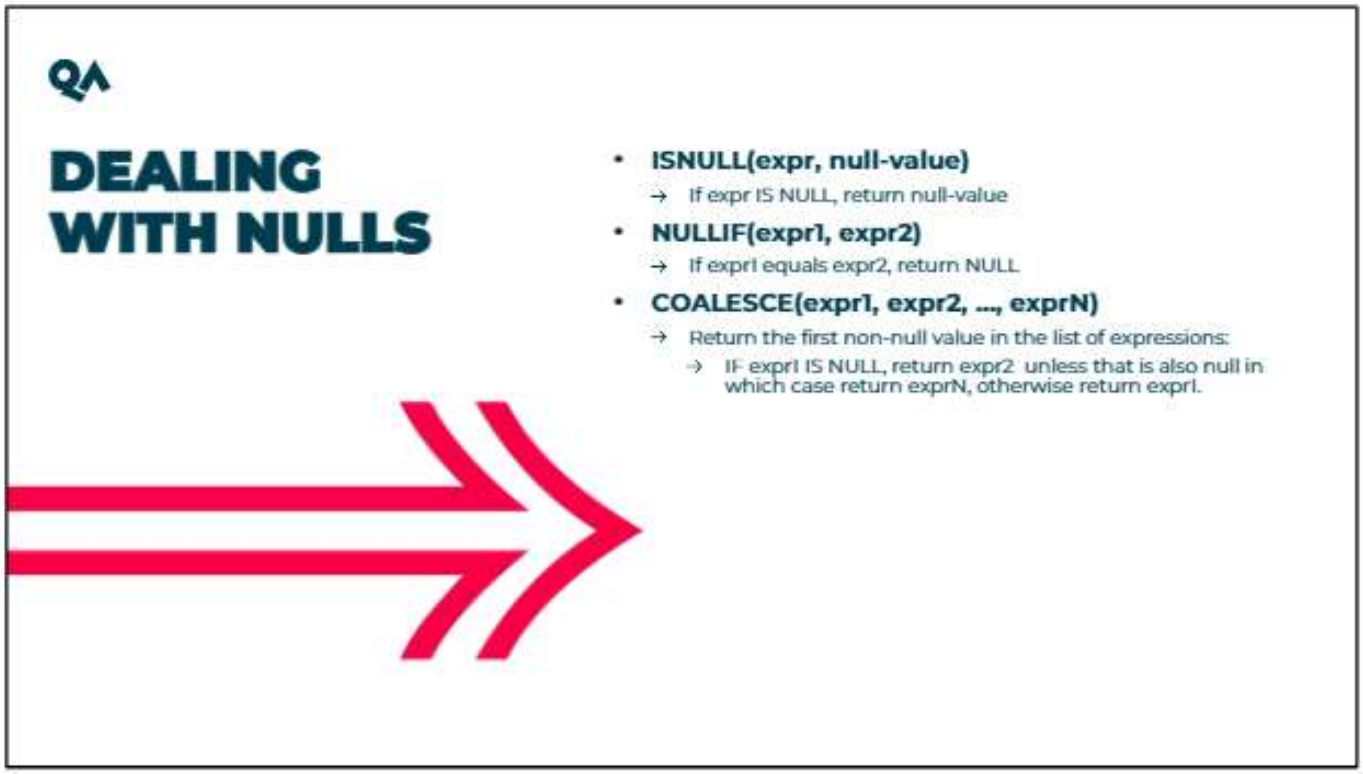
The *GetDate* function is often used in conjunction with the other date functions, for example *DateDiff* is used to find the difference between two dates and is often used to find the difference between a date column and today's date:

```
DateDiff(day, Orders.DueDate, GetDate())

DateDiff(year, '19230823', GetDate())
```

More date/time gotchas: the DateDiff function is not that clever when it comes to years and months; it simply looks at the year (or month) part of the date and subtracts one from the other. The year examples above will only correctly calculate the age after the 22nd August.


Working with
nulls



The *IsNull* function is used to give a value to null expressions. There are two main reason for this – either we want nulls to be treated as (say) 0 for the purposes of an aggregate function:

AVG(SomeNullableColumn) *versus*
AVG(ISNULL(SomeNullableColumn, 0))

Or we want to display some text for a null value:

SELECT ContactName, Phone, IsNull(Fax, 'n/a') AS Fax FROM Customers

The *NullIf* function is used for the opposite purpose, perhaps we want to ignore zeros in an aggregate:

AVG(SomeColumnWithZeros) *versus*
AVG(NullIf(SomeColumnWithZeros, 0))

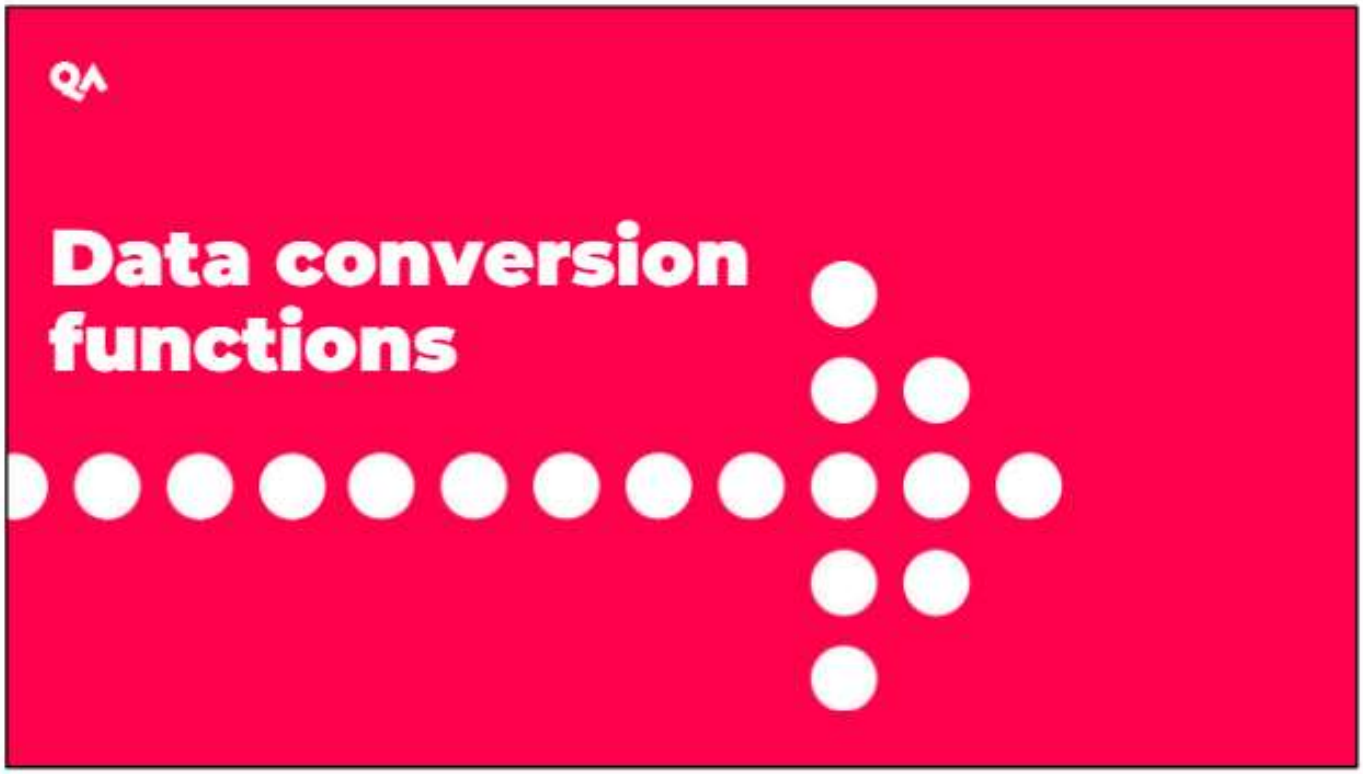
Or we want a default value to display as null

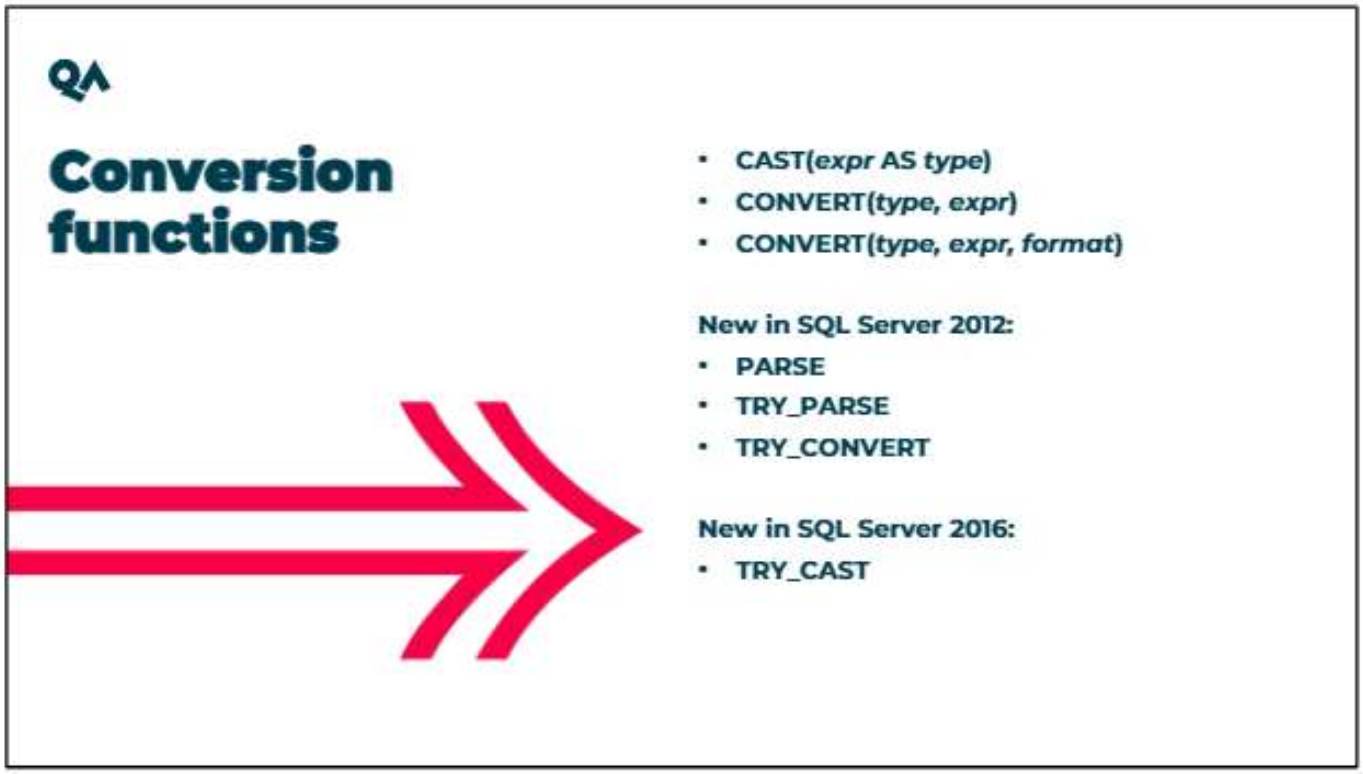
SELECT FirstName, NullIf(MiddleName, ''), LastName FROM Contacts

Finally, *Coalesce* is used to select the first non-null value from a list of possibly-null values. A canonical example would be a list of employees, some of whom have salaries and some of whom are paid hourly:

```
SELECT EmployeeID, Coalesce(Salary / 12, HoursWorked * HourlyRate, 0) AS Pay
FROM EmployeeWages
```

The assumption being that the Salary column is nullable – if it contains null then salary divided by 12 would be null so the Coalesce would ignore that and check the HoursWorked and HourlyRate columns, which hopefully both contain non-null values, otherwise the employee gets paid nothing!


Data conversion functions



SQL is generally quite good at working out how to convert between data types but sometimes we need to tell it exactly how we want to change a type. The most obvious example is in dealing with the various different date data types – *GetDate*() returns a datetime but we might only be interested in the date portion:

CAST(GETDATE() AS Date) or
CONVERT(Date, GETDATE())

Or when truncating long text – the Notes column contains up to 2GB of text and we only want the first 25 characters:

CAST(Notes AS varchar(25)) or
CONVERT(varchar(25), Notes)

Cast is preferred by many people over *Convert*, but they both do the same job in most cases.

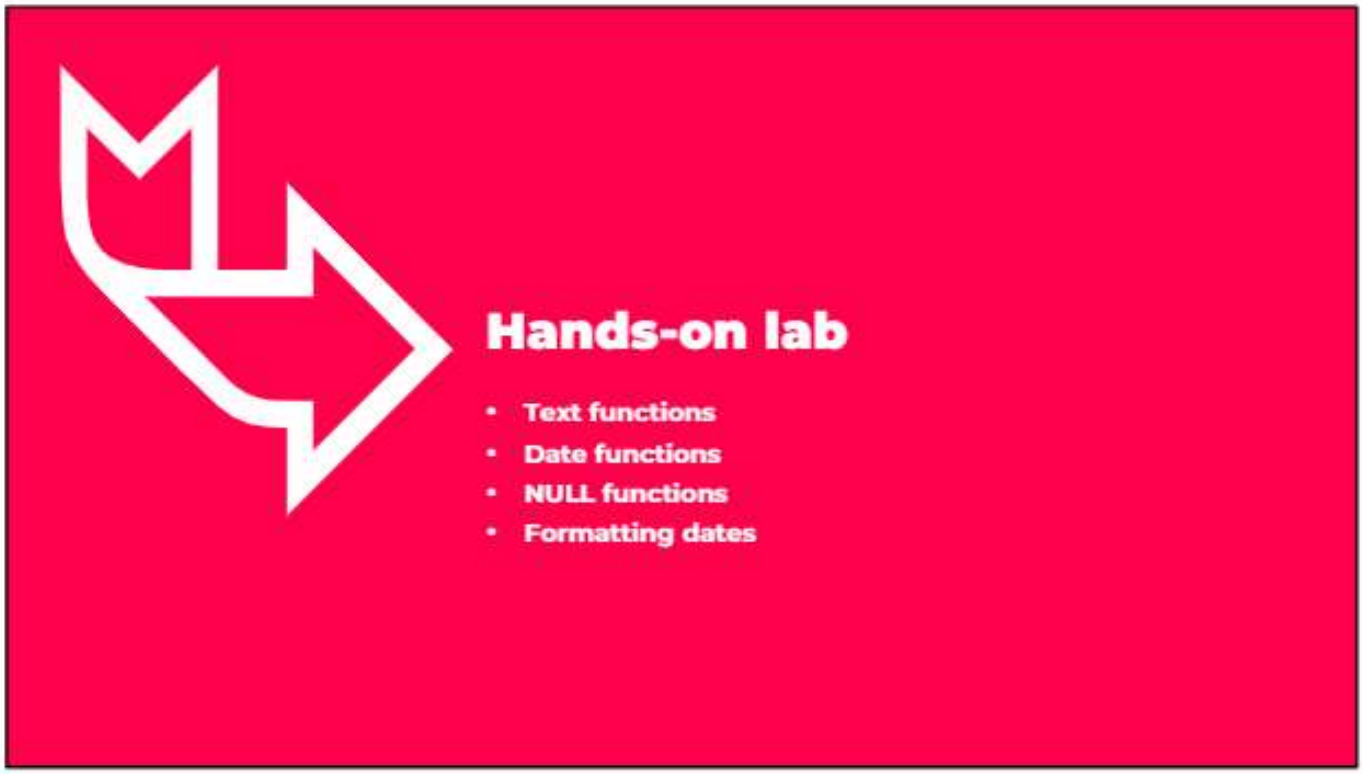
Where Convert really comes into its own is when we use the optional, third, *format code* parameter which enables us to ask for an alternative format, especially useful for dates:

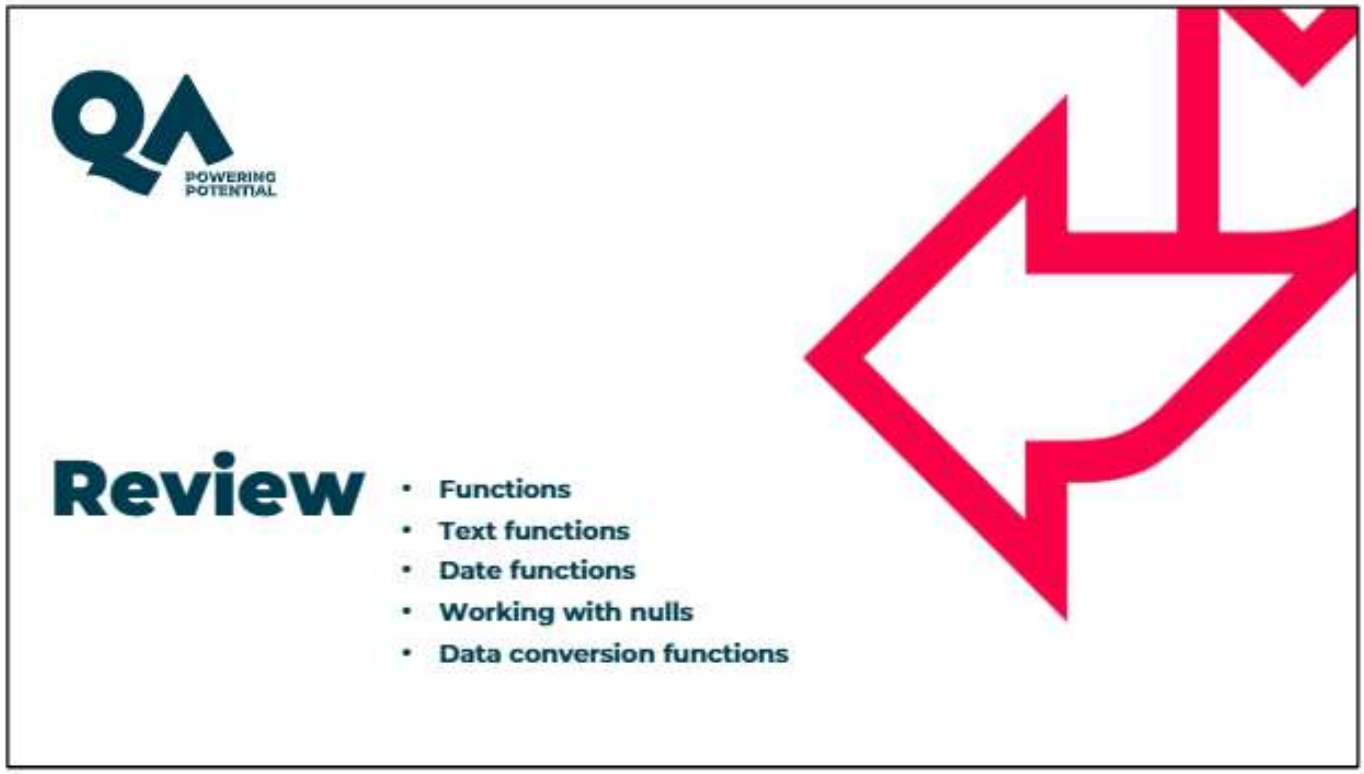
CONVERT(varchar(20), OrderDate, 103) - - UK Date format
CONVERT(varchar(20), OrderDate, 106) - - dd MON yy format

For a complete list of format codes, search for "convert function" in Books Online or on the Microsoft website.

The newly-added PARSE / TRY_PARSE functions provide improved support for internationalisation.

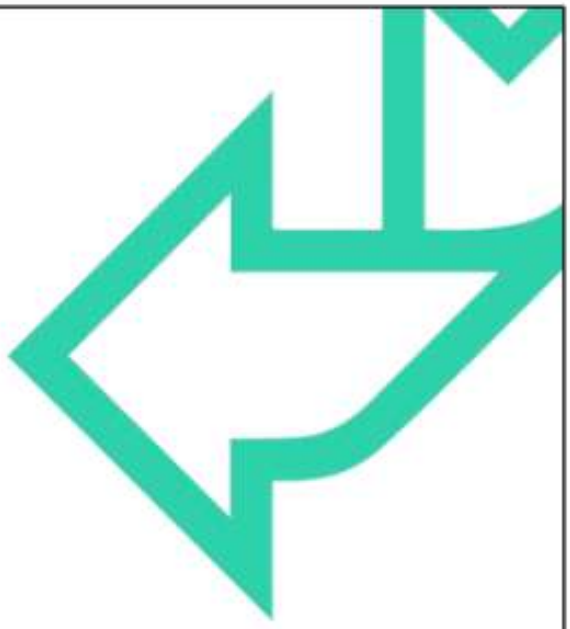
If CAST, CONVERT or PARSE are used and their operation cannot be performed – for example if an attempt is made to change a text column into an integer, or a PARSE uses an incompatible date or currency format for the provided culture an error will be returned causing the batch to halt. If this behaviour is undesirable the TRY_ variants of the commands can be used instead. Should an error be encountered when one of these is used a NULL value will be returned for any failing rows rather than the whole batch failing with an error.


Hands-on lab
Text functions
Date functions
NULL functions
Formatting dates


POWERING
POTENTIAL
Review
Functions
Text functions
Date functions
Working with nulls
Data conversion functions

QA
POWERING POTENTIAL

Views and Stored Procedures

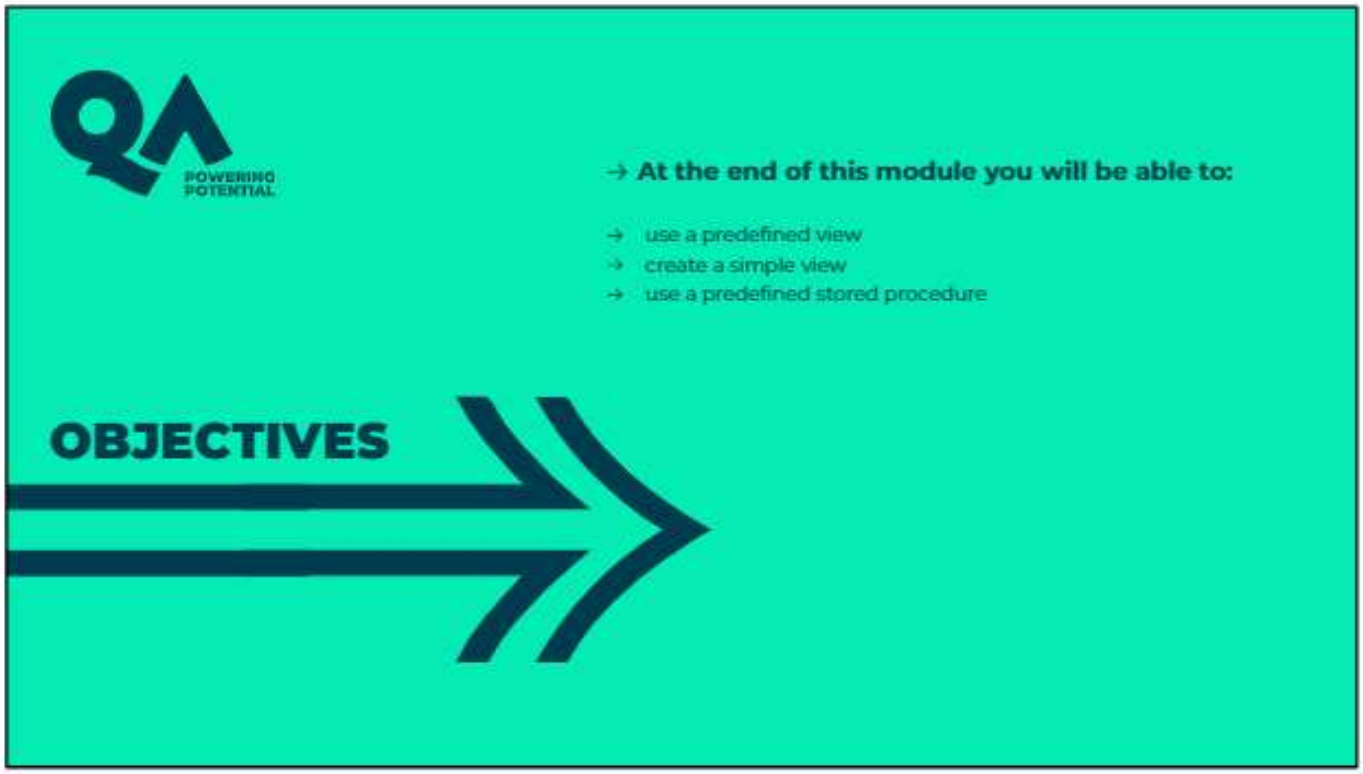


Version 1.5

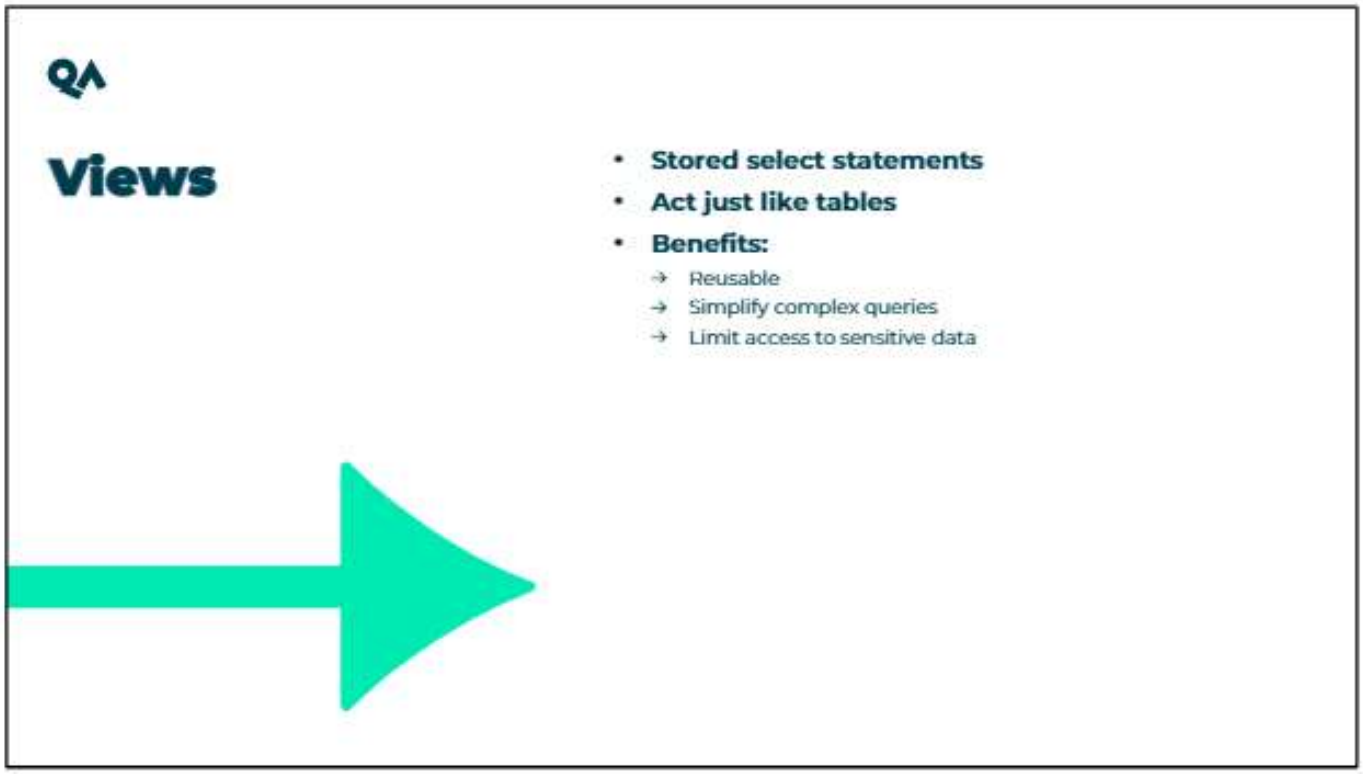
POWERING POTENTIAL

Overview

- Views
- Using Views
- Stored Procedures
- Using Stored Procedures


POWERING
POTENTIAL
→ At the end of this module you will be able to:
→ use a predefined view
→ create a simple view
→ use a predefined stored procedure
OBJECTIVES


Views



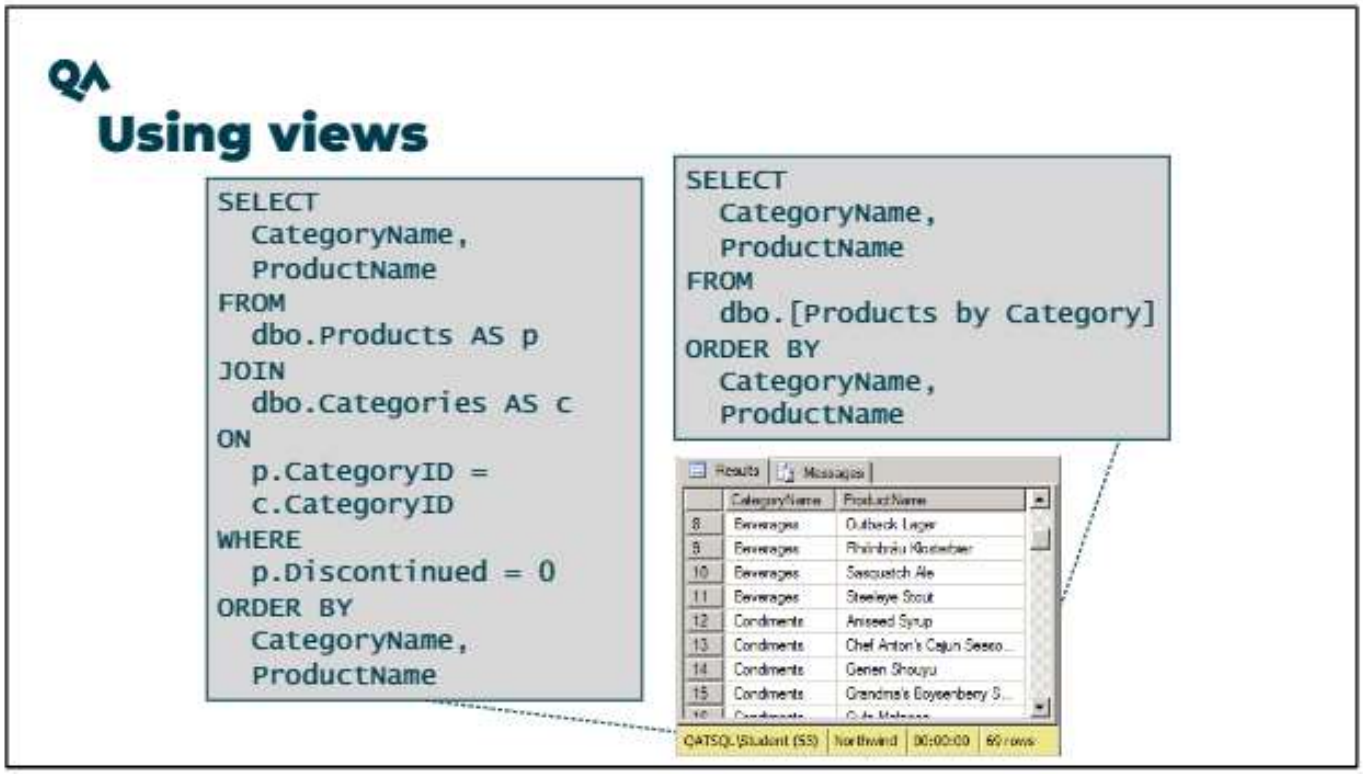
As you might have realised by now, queries can quickly become extremely complex and hard to follow once we start joining tables together. To simplify matters, select statements can be stored in the database as views which we can then query as though they were tables.

Some points to note about views:

- They do not take up space – when we use a view in the FROM clause of a SELECT statement, SQL looks up the view definition and uses that as the basis for the selection.
- There is very little overhead to using views, just the time taken to look up the view's definition. SQL does not go and get all of the rows and columns selected by the view and then apply any WHERE clauses. The final query that is executed is a combination of the view's definition and our select statement.
- Views can also be used to control access to sensitive data. Whilst it is possible for database administrators ("DBAs") to restrict access on a column-by-column and user-by-user basis, it is easier (both to set up and to maintain) to restrict access to tables and allow access to views which don't include the sensitive data.

A full discussion of database security is beyond the scope of this course.


Using Views



In the example on the slide, we are using a view that already exists in the Northwind database, called "Products by Category", that joins the Products and Categories tables and includes a WHERE clause to get rid of discontinued products. Our second FROM clause is much simpler but we still need an ORDER BY clause as view definitions can not include sorting*. Remember sorting is the last thing to occur, so we have to know what records to get before doing the sort. If the view definition had the ORDER BY, but the user's SELECT statement had the WHERE clause, SQL could potentially be doing a whole lot of sorting before realising that it didn't need to return any records!

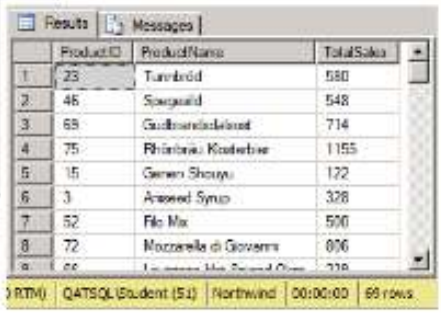
The second query is a lot simpler, and would be even more so if the view being referenced didn't have spaces in its name.

One thing that we can't do in a view is ask for any columns *not* in the view's select list. For example the "Products by Category" view doesn't include ProductIDs or CategoryIDs, so we can't ask for them in the second query but we could have in the first one.

*Strictly speaking, it is possible under some circumstances to include ORDER BY in a view. But definitely not in this case!

Using views (2)

```
SELECT
  p.ProductID,
  p.ProductName,
  SUM(Quantity) AS TotalSales
FROM
  dbo.[Current Product List] AS p
JOIN
  dbo.[Order Details] AS od
ON
  p.ProductID = od.ProductID
GROUP BY
  p.ProductID, p.ProductName
```

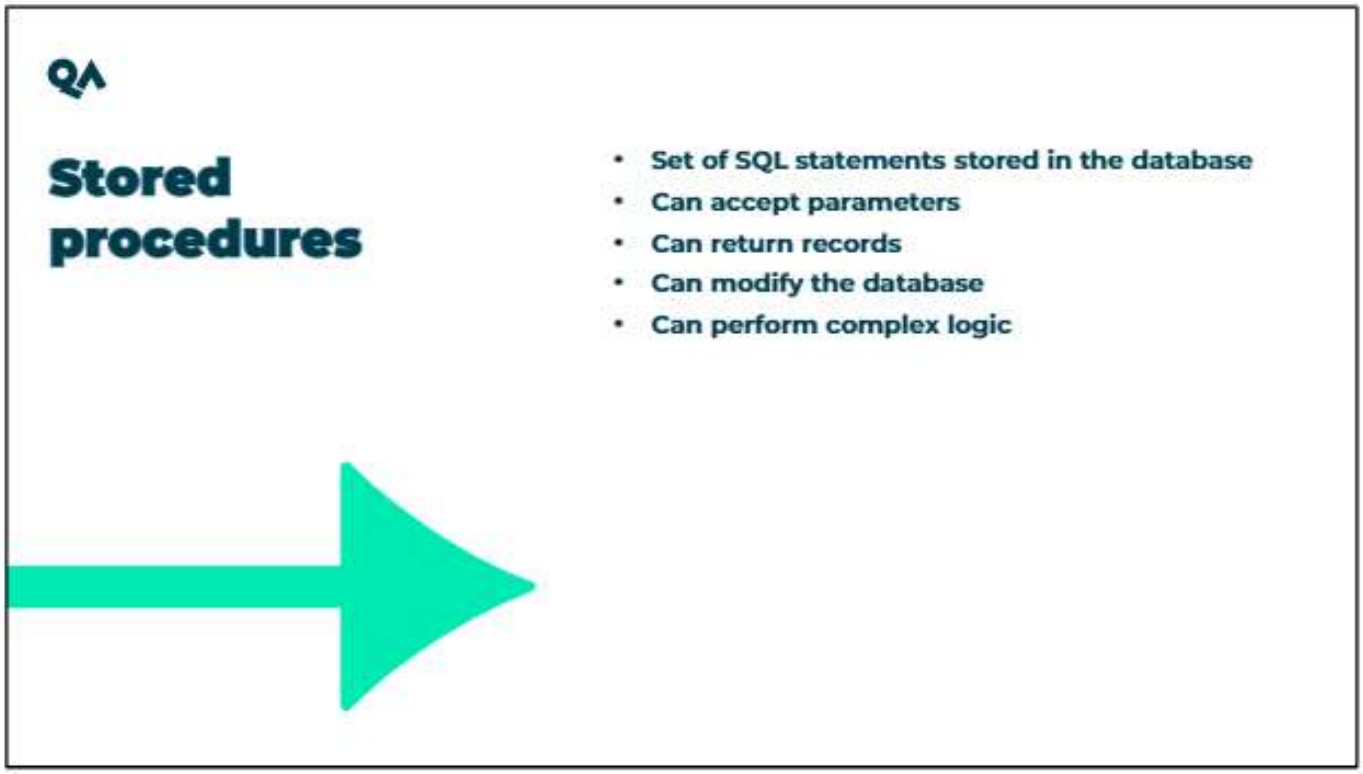


The example on the slide JOINs the Order Details table to a view called "Current Product List", which is saving us a single SQL clause, namely:

WHERE Products.Discontinued = 0

It serves to illustrate that you can JOIN tables to views, or views to views, or views to tables as well.


Stored Procedures



Views are always and only stored SELECT statements.

Stored procedures, also known as “stored procs” (pronounced to rhyme with “frocks”) or “SPs” are any stored set of SQL statements, which could be a query or could be some complex database update code.

SPs can accept parameters, which for a proc that returns records could be used to adjust the WHERE clause or for a proc that adds a new record to a table could be a list of the values to add.

Since SQL 2005 stored procedures can also be created using managed code such as C#

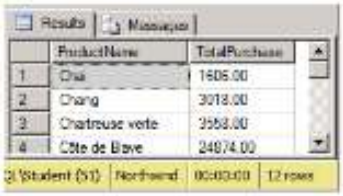

Using Stored
Procedures

Using stored procedures

- **EXECUTE** *<<procedure-name>>*
- **EXEC** *<<procedure-name>> param1, ..., paramN*

```
EXEC dbo.[Ten Most Expensive Products]
```

```
EXEC dbo.SalesByCategory
  'Beverages', 1996
```



We use the *EXECUTE* statement to call a stored procedure. The *EXECUTE* can be and often is shortened to *EXEC*. This is then followed by the name of the SP and then any parameters that need to be passed to it.
In the first example on the slide, we are executing one of the procs in the Northwind database that effectively runs the following query:

SELECT TOP 10 ProductName, UnitPrice FROM Products ORDER BY UnitPrice DESC

In the second example on the slide, we are executing another of the sample procs. This one calculates the annual sales for products in a certain category, so we need to tell it which category we are interested in and also which year. We could get the same results by executing the proc in the following way:

EXEC dbo.SalesByCategory @CategoryName = 'Beverages', @OrdYear = 1996

The author of the proc has also defined a default value for the "OrdYear" parameter of 1998, so we could execute it:

EXEC dbo.SalesByCategory @CategoryName = 'Beverages'

and get the sales for the beverages category for the year 1998

HANDS-ON LAB

- Using views
- Creating views
- Using stored procedures


POWERING
POTENTIAL
Review
→ Views
→ Using Views
→ Stored Procedures
→ Using Stored Procedures


POWERING
POTENTIAL
Modifying Data
Version 1.5


Overview
Adding new rows
INSERT
Modifying existing rows
UPDATE
Removing rows
DELETE


INSERT

QA

INSERT

- **INSERT [INTO]** *table*
  (*col1*, ... *colN***)**
  VALUES (*val1*, ..., *valN***)**
- **INSERT [INTO]** *table*
  SELECT ...


UPDATE

UPDATE

- **UPDATE**
  table
  SET
  col1 = val1,
  ...,
  colN = valN
  WHERE
  colX = valX

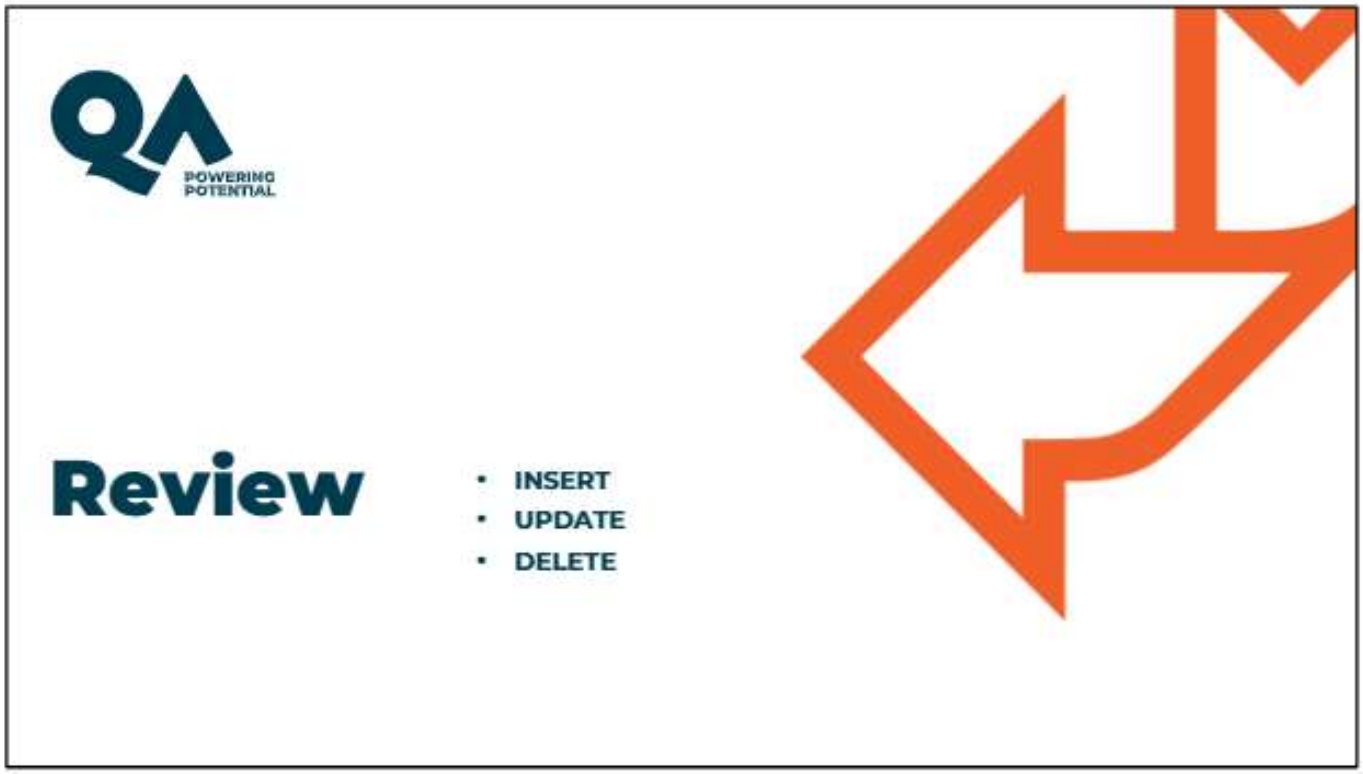

QA
DELETE

QA

DELETE

- **DELETE [FROM]** *table*
  WHERE ...

8


POWERING
POTENTIAL
Review
INSERT
UPDATE
DELETE


POWERING
POTENTIAL
Working with
Tables
Version 1.5


POWERING POTENTIAL
Overview
Data Definition Language (DDL)
CREATE
ALTER
DROP
Data Manipulation Language (DML)
INSERT
UPDATE
DELETE


CREATE

QA

CREATE

- **Use CREATE to add a table to the database with the columns defined**
- **CREATE TABLE table**
  (
  ColumnName datatype <options>
  ...
)
- **Options:**
  - → NULL / NOT NULL
  - → **DEFAULT**
  - → **CHECK**
  - → **PRIMARY KEY**

Data Types

- **Numerics**
  - BigInt, Int, SmallInt, TinyInt, Bit
  - Decimal, Numeric
  - Money, SmallMoney
  - Float, Real
- **Date / Time**
  - DateTime, DateTime2, Date, Time, SmallDateTime, DateTimeOffset
- **Character strings**
  - VarChar, Char, NVarChar, NChar
- **Binary data**
  - Binary, Image, VarBinary
- **Other**
  - XML, Geometry, Geography, HierarchyID, Cursor

Options

- **NULL / NOT NULL**
  - NULL allows blank / empty values in the column.
  - NOT NULL does not allow blank / empty values.
- **DEFAULT**
  - Allows for a default to be entered if no value is given, such as No in an OrderedFilled column, as the field will be set to Yes only when the products are sent.
- **CHECK**
  - Allows for a rule to be placed on the field such as Age < 130 AND Age>= 0 on a column holding human ages.
- **PRIMARY KEY**
  - This designates the column (or columns combined) as unique within the table, so no duplicates can be held and no nulls are allowed.
  - One primary key is allowed per table.


ALTER

QA

ALTER

- **Use ALTER TABLE to change the structure of a table:**
  - Add columns
  - Remove columns
- **ALTER TABLE tableName**
  ADD columnName datatype <options>
- **ALTER TABLE tableName**
  DROP COLUMN columnName


DROP

QA

DROP

- Use DROP TABLE to remove the table from the database
- DROP TABLE tableName


INSERT

QA

INSERT

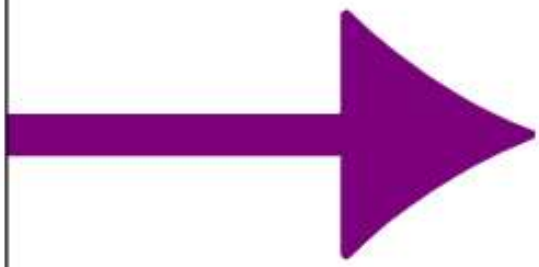
- **INSERT [INTO]** *table*
  (*col1***, ... *colN***)**
  VALUES (*val1***, ..., *valN***)**
- **INSERT [INTO]** *table*
  SELECT ...


UPDATE

QA

UPDATE

- **UPDATE**
  table
  SET
  col1 = val1,
  ...,
  colN = valN
  WHERE
  colX = valX

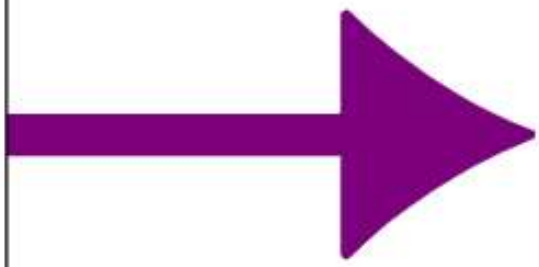


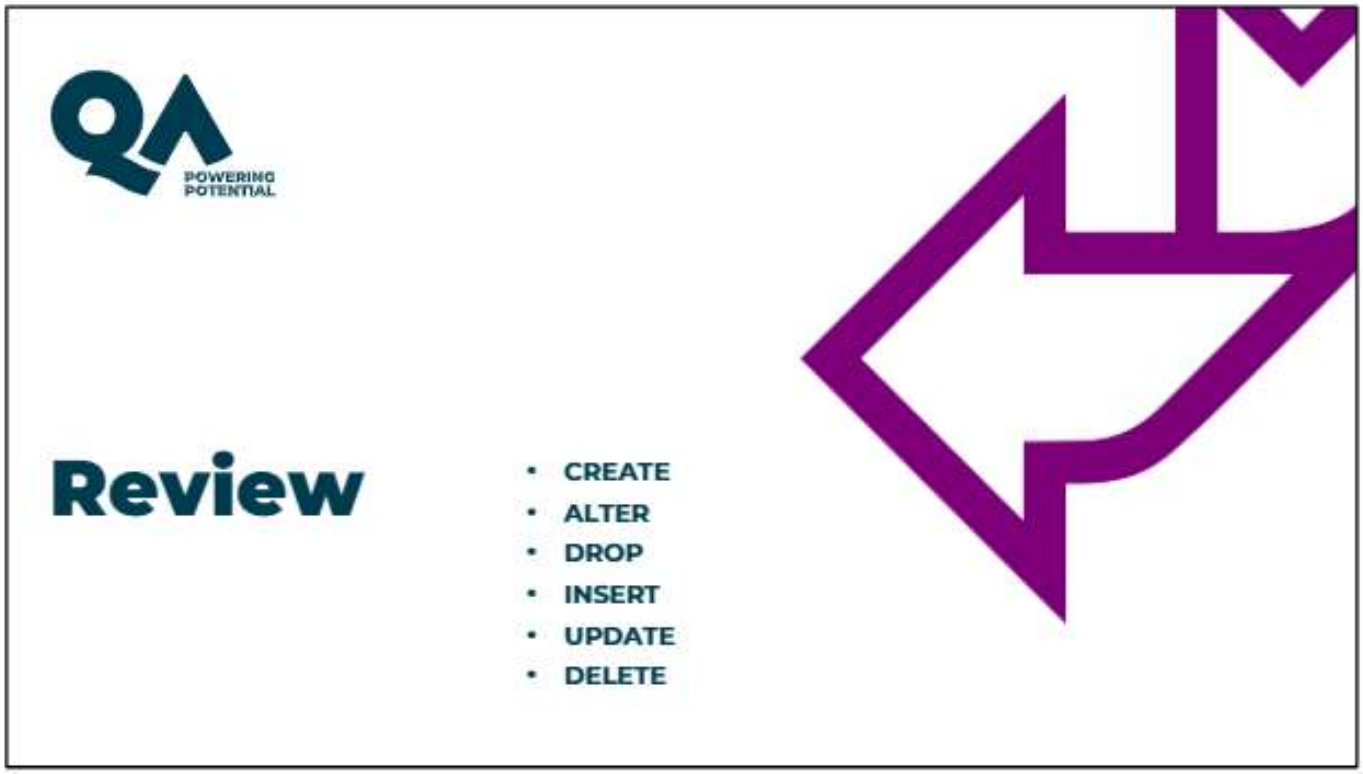

DELETE

QA

DELETE

- **DELETE [FROM]** *table*
  WHERE ...




POWERING
POTENTIAL
Review
CREATE
ALTER
DROP
INSERT
UPDATE
DELETE

FANCY A CHAT?

0345 757 3888

INFO@QA.COM

QALtd QA-Ltd QALtd QALimited

QA.COM

V1.0