SQL

with practice exercises

Learn SQL Fast

on free software

D Armstrong

Table of Contents

Learn SQL Fast

A query writing course you can read on the train, based on software you can download for free

Copyright

Disclaimer

While the advice and information in this book are believed to be true and accurate at the date of publication, neither the author nor the publisher can accept any legal responsibility for any errors or omissions that may be made. The publisher and author makes no warranty, express or implied, with respect to the material contained herein.

Introduction – SQL - Why you should learn it

SQL is widely used

SQL – Structured Query Language – is a widely used language for working with data. All organizations have data, and they typically store at least some of it in a database.

SQL is the language of databases: learning it allows you to use them effectively.

SQL can do many things

SQL allows you to view data, analyse it, and perform calculations with it. With SQL, you can change the data stored in a database, or change the way the database itself works. SQL also lets you make rules about how data may be changed and who by.

This book covers the most common use of SQL: getting data out of databases. It shows you how you to view, analyse, and report on data, which is the first step to becoming a confident SQL user, because it is the skill that lets you see what you are working with.

SQL skills are transferable

There are many database technologies: Microsoft SQL Server, MySQL, Oracle, Microsoft Access, and so on. Each has its own version of SQL. However, unlike programming languages, they are all very similar.

SQL skills in one technology, such Microsoft SQL Server, can be quickly and easily applied to another, e.g. MySQL. Learning SQL once allows you to use it many times.

Learning SQL is simple

This book is designed to make learning SQL as straightforward as possible. It is a concise, step-by-step course, written in plain English, with a focus on using what you learn. It gives examples of SQL (known as “queries”) which you can run on your home computer, and explains how they work. After that, this book gives practice exercises, so that you can try out what you learn, before moving on. Later chapters show how to combine the basics to write more advanced queries.

When a new technical term is introduced, this course will give a definition, or plain-English equivalent, at that point.

The software needed to practice SQL will run on a home PC or laptop and is freely available online. This book will guide you through downloading and installing the correct version for your machine.

Let’s get started.

Microsoft SSMS – How to get free SQL software

The big picture

The examples and exercises in this book use Microsoft technologies, although as mentioned earlier, once you have learned SQL with one technology it is easily transferable to another. The book also assumes your computer has Microsoft Windows already installed.

You will need two pieces of software: SQL Server itself and SQL Server Management Studio (SMSS). Microsoft provides both online for free, as a single package to download and install.

SQL Server is the software that handles any SQL queries you run, but it doesn't actually display the results of queries, or appear on screen at all. SMSS is the software you will actually see: the user-interface. This is where you type any queries you want to try out, and it is what displays the results.

What to download

Microsoft offers several different “editions” of SQL Server online. SQL Server **Express** is the free one, and the one this book is based on.

Within the Express edition, Microsoft also offers several different download “packages”, e.g. advanced, with tools, etc. SQL Server Express **with Advanced Services** includes SMSS, which you will need, and is the one this book is based on.

Where to download it from

You can download a free version of the above from Microsoft's website, by clicking the download link on the web page below (if you Google "download SQL Server", this page should appear near the top of your results):

https://www.microsoft.com/en-us/server-cloud/products/sql-server-editions/sql-server-express.aspx

When you click the link, Microsoft's site will prompt you to sign-in or register for an account. Once you have done this, it will provide a choice of SQL Server Express packages to download.

If you sign up for a new Microsoft account and it doesn't take you directly to the download options afterwards, go back to the web page above, click the download link again, and sign in with your newly registered account details. You will then be on the correct page to choose you download package.

Choosing between 32-bit and 64-bit

You will notice that the name of the package you need (SQL Server Express **with Advanced Services**) appears twice, once for 64-bit systems and again for 32-bit systems. You need to download the one which matches your system. To check whether your system is 64-bit or 32-bit, you can use any of the methods below. They are listed in order: fastest first. If your computer does not support one method, you can simply use the next one.

Method 1

Go to “search” on your computer. Type “system”. Press return. The system window will appear. In the system window, look down the listed information to find the system type. This will state if the computer is 64-bit or 32-bit. If you see “86”, e.g. “x86 based processor”, then this means the processor is 32-bit.

Method 2

Open Windows Explorer, i.e. open any folder, such as “My Documents”, on your computer. In the folder window which appears, on the left, you should see an option called “My Computer”, or “This PC”, or something very similar. Right-click on this option and a menu will appear. Click the “Properties” option on this menu and a new window will appear. In this window, look down the listed information to find the system type. This will state if the computer is 64-bit or 32-bit. If you see “86”, e.g. “x86 based processor”, then this means the processor is 32-bit.

Method 3

Go to “search” on your computer. Type “cmd” and press return. The “command prompt” (a black window) will appear. In the command prompt, type “systeminfo”, and press return. This will fill the command window with more information than fits on the screen. Scroll back up to top of the command window and read down until you find a mention of “64-bit” or “x64”, this will be recorded next to “system type:” or similar. If you don’t see that, do the same checks, but look for “32” or “86” in place of “64”, i.e. look for “x86” and “32-bit”. If you find any of these, then your computer is 32-bit.

Completing the download

Once you have selected a package, click on the “Continue” button at the bottom of the page. This will take you to a page which starts the download. The download is a folder, not just one file.

If your browser gives you the option to save the download, save it anywhere you like, as long as you know where it is. For example, save it to your desktop for easy access to the download.

If you don’t get the choice of where to save, it will almost certainly be downloaded to your computer’s downloads folder, typically “C:\Users\(your username here)\Downloads”.

The download may take some time, but it will continue by itself, so at this point you can leave the process to finish itself.

Installing

Open the folder that you just downloaded, and you will see a file called “SETUP”. Double-click on that file to run it, and it will take you through the install process. This will be the usual process, i.e. an install wizard.

The first window that will open is titled “SQL Server Installation Center Options”. This will give two options: “New” and “Upgrade”. Click on “New”. This will open a second window “SQL Server 2014 Setup”.

At this point, you can just keep clicking the “next” button until the install process starts. The wizard will ask you things like whether you want to install updates automatically, and which type of Authentication to use for SQL Server, but none of this is likely to affect your use of the software for the purposes of this course.

Like the download, the install takes some time, but, also like the download, it will continue by itself, so at this point you can leave the process and come back to it later.

When the install process is complete, go to “search” on your computer and type “SMSS”. The results will show you SQL Server Management Studio. Click to open it. It may take a few minutes to load up the first time you do this. When does open, a dialog box (small window) will appear. Click the “Connect” button on this box. Your software is now ready to use, you just need some data to use it on. The next chapter will cover that.

Northwind – How to get a free database to practice on

The big picture

You have now set up SSMS, which allows you to enter queries and see their results, and SQL server, which actually runs those queries on databases. What you need now is a database to run those queries on. You also need that database to contain some test data for those queries to find.

Microsoft provides a free database online, complete with data. This chapter will show you how to get that database and set it up for use with SQL Server.

Download your database

The instructions below are based on using Google Chrome, because it lets you run the installer directly from the browser, which simplifies the process a little. If you are limited to, or prefer to use, a browser which doesn't let you do this, the next section has instructions on how to find the installer file in your Downloads folder and run it from there.

You can go directly to the page that instructs you on how to download Northwind (or rather, the Northwind installer), run the installer and then connect the database in SSMS, at:

https://msdn.microsoft.com/library/8b6y4c7s.aspx

However, on Google, if you just start typing "Download Northwind …" it should auto-suggest "… for SQL server 2014". If you search for that, it will, at the time of writing, give you the page above in your search results.

The page above contains a section titled "To install the Northwind and Pubs sample databases for SQL Server". In that section, step one has a link to the "Northwind and Pubs Sample Databases Website". This link takes you to a page with the address below:

https://www.microsoft.com/en-us/download/details.aspx?id=23654

This is the page you actually download the database from. Ignore the fact that the webpage says "for SQL Server 2000": the Northwind database was created a while ago, but newer versions of SQL Server can still use it. The webpage has a big red "Download" button on it. Once you download the file, it will appear as a tab in the bottom left of your browser (at least, if you are using Chrome). The file name is "SQL2000SampleDb.msi". Clicking on this file will give you the option to run the installer straightaway.

Install your database

Preferably, run the installer from your browser as described above.

If for any reason you don’t get, or don’t use, the option to run the installer from your browser when you download it, you can still proceed. Go to your “downloads” folder. Typically this will be at: “C:\Users\(your username here)\Downloads”. Double-click the installer file to run it.

Connect to your database in SMSS

The first of the two web pages above does include instructions for getting set up with and connecting to your database, but I think the method below is easier to follow, so I recommend this way instead:

Open SSMS. From the menu bar in the top left, click on “File”, “Open” and “File”. A window will appear letting you browse through folders to find the file you need to install. By default, the database files are automatically created in a “SQL Server 2000 Sample Databases” folder on the C: Drive.

You can get to the C: drive by clicking the “computer”, or “my computer”, option on the left of your window, then double-clicking on the C: drive icon. From there, open the “SQL Server 2000 Sample Databases” folder.

From that folder, open the file “instnwnd.sql” (it may just appear as “instnwnd”). This is a file containing a SQL query which will set up the database (create it and fill it with data). Opening the file will display the text of the query, brightly color-coded.

Click the red exclamation mark “!” (called “execute”) at the top of the SSMS window. This will run the query and make the database. It may give you a couple of errors message about not being able to find stored procedures. Ignore them.

If you now go to the “Object Explorer” panel on the left side of SSMS and right-click its “Databases” item, this will open a menu. Click the “refresh” option at the bottom of that menu. Now click the plus sign to the left of “Databases” to make it expand, showing you the Northwind database. You are ready to begin!

Tables – What's in a database

Introduction

Before running any queries, you need to know what data there is to be queried. Data is stored in tables. This chapter will cover how to find the tables in your database, and how to see what they contain.

Finding tables

Open SQL Server Management Studio on your computer. If you can't find it, then typing "sql" or "ssms" in your computer's search field will bring it up for you.

If you look in the object explorer, you will see a cylinder-shaped server icon. There will be a green circle with a white triangle, like a "play" button, on it, and it will have a small white box to its left, containing a minus sign. Click the box to hide the items grouped under the icon (databases, security, etc). The box's sign will also change to a "plus". Click it again to make the hidden items reappear.

Now click on the sign next to "Databases" to make items under that visible. Repeat these steps on "Northwind" and then "Tables". This will show you a list of all the tables in the database.

Looking at tables

Right-click on the “dbo.Products” table and a menu will appear. On the menu, click the option for “Select Top 1000 Rows”. This will open a tab in the main part of your SSMS window, and the bottom half of the tab will display the contents of the table. This table doesn’t have a thousand rows, so the tab just shows you the rows it does have. If the table had more than a thousand rows, you would see the first thousand.

You will see that the table is a list of products, one product per row. You will see that each column holds a specific piece of information about each product: its ID number, name, price, etc.

Use “Select Top 1000 Rows” to look at the Categories table (“dbo.Categories”). You will see a list of product categories, one per row, with specific information about those categories in each column.

You can have multiple open tabs at a time, however, once they fill the window, SMSS will start to move tabs into a drop-down menu, and they become less convenient to work with. Having two or three tabs open at once can be useful though. You can use one or two tabs as a reference, to see what tables contain, and another to write a query on those tables.

But first, now that you have some examples on screen, we’ll cover what database tables actually are.

What tables are

Tables are where databases store data. In Northwind, for example, we have tables of employees regions, products, and so on. Each of these tables stores data about a particular **type** of person, place or thing. The technical term for this is “entity”. Employees are one type of entity, regions are another, and products a third. Entities can also be more abstract, non-physical things, such as product categories. They can be types of events or actions, such as customer orders. All of the above would have different data recorded about them, so each of them would need their own table.

Rows

Each table row, or “record”, holds data about one **example**, or “instance”, of an entity. So an employee would get one row in the Employees table. Each new employee would be on a new row. The number of rows in a table can be anything from zero upwards, and can change over time as the database users add or delete records.

Columns

A column, or “field”, holds specific data about one **aspect** of the entities in its table. So one column might hold the first names of the employees, the next might hold the last names. The number of columns is fixed, and doesn’t change unless the design of the database is changed, e.g. by a developer.

Relationships

The records in the Northwind tables are related. For example, a customer may have several orders, so one row in the Customers table would be associated with several rows in the Orders table. To see another example, return to the Products table “dbo.Products”. The first record in the table is for Chai, and its CategoryID is 1. If you look in the Categories table, you will see that this CategoryID number is for beverages. So, if you didn’t already, you now know that Chai is a beverage. The tables are related by their CategoryID fields. Later in the book, we’ll see how SQL can use this feature to gather related data from several tables at once.

Technical terms

There are quite a few technical terms to do with data and databases. Whenever this book introduces a new term, it will usually be in quotes, like this:

> Columns in database tables are also known as “fields”.

However, when there is a plain English equivalent, the term may simply be given after it, in brackets, like this:

> Columns (fields) store specific types of data, e.g. one field may store product names and another may store their prices.

The next step

The next chapter will show you how to get data from a table. After that, you'll see how to specify which fields you want to view for each record. Once you know how to get the data in those, we'll move on to doing things with it.

SELECT – How to query (get data from) a database table

Making New Queries

In the top left of the window, under the menu options for “file”, “edit”, “view”, and the others, is the “New Query” button. Click on that button. This will make a blank tab appear. Also, the white drop-down box below the “New Query” button should now say “Northwind”. If it doesn’t, e.g. if it says “master”, click on it and choose the Northwind option from the list. You may have to do this with other new queries in future, as it can default back to “master”. If you run a query and get an error message saying that the things you are querying can’t be found or don’t exist, this is something worth checking.

Now, in the blank tab you just opened, type the text below for your first SQL query.

The Query

Type:

```
SELECT *
FROM Pzyxroducts
```

You don’t need to type the “dbo.” part of the table name, as Microsoft’s software defaults to that anyway. Also, you don’t need to use capitals in your query if you don’t want to (and it’s faster to type if you don’t). This book uses capitals purely to make clear which words are SQL keywords, e.g. SELECT, and which are not, e.g. Products. Finally, it doesn’t make any difference how many lines you spread your query across. If you type the query above all on one line, it works exactly the same.

Click the “Execute” button on the toolbar above your query tab, or just press the F5 key on your keyboard. This will run your query. You will see the results of your query appear in the bottom half of the tab.

The explanation

This query gets all the rows and columns, and therefore all the data, from the Products table. The "SELECT" keyword tells the software which columns to retrieve. The "*" symbol (asterisk) tells it to get (return) all columns.

The "FROM" keyword says which table to take data from. Followed by the Products table name, it says to take data from Products. By default, SQL returns all rows from the tables it queries. In later chapters, we'll explore ways to return only the rows you need.

To summarise, the above query says "take the data from all columns of the Products table".

If you just want to see the contents of a table, this type of query is useful. In fact, this type of query is especially useful to SQL users, i.e. you. You can use it to quickly see the contents of a table before taking the time to write more advanced queries. Once you have seen the structure and contents of a table, it is easier to query it in more specific ways, as we will do later.

Exercise – 1.1

(answer)

Get all data from the Categories table. Write and run the query to do this in SSMS.

Linked answers

When you have done this exercise, you can click on and follow the “(answer - 1.1)” link to compare your query to the one in the answer. All the exercises have hyperlinked answers, which you can refer to if you get stuck or want to check that your answer is correct.

When you check an answer, you can use your e-reader’s back button to return to the main part of this course. However, just in case you don’t have a back button, or it’s broken, or you can’t use it for some reason, each answer also includes a link back to its corresponding exercise.

How to get specific columns

Now replace the asterisk in your query with the names of two columns, i.e.

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

Run the query (Execute / F5) and see how it only shows two columns. Note that, with more than one column in the query, you have to separate column names with commas.

Exercise – 1.2

(answer)

For each product, display its name, price and the number of units in stock.

A note on how this book formats queries

This book puts each part of a query on a new line, to make them easier to understand. It also helps queries fit neatly onto the screen of your e-reader. However, as screen sizes and query line lengths vary, your e-reader may sometimes split one line from the book across several lines on the screen. These split lines will appear closer together, as below:

> This is one very long line of text which is not SQL but does show how close together lines of text appear if they are separated automatically by an e-reader to make sure they fit on a screen.

On the other hand, text which is ***put*** on two lines appears with a bigger gap in the middle, like this:

> Line 1
>
> Line 2

The next chapter has some longer "SELECT" statements, so you may notice this effect there. This note is included now to prevent any confusion about the layout of such queries later in the book. Ultimately however, SSMS treats line-breaks like spaces, which means that when you type in your queries, you can format them across however many lines you like.

SELECT - How to do calculations within rows

Introduction

This chapter covers common tasks in SQL, such as performing basic math operations and handling fields without values in. It demonstrates these with “SELECT”, as this provides a way to display the results on-screen. However, apart from the use of “AS” to rename columns in query results, all of the things in this chapter can be used in other ways. You will see some examples of this with other SQL keywords in later chapters.

An example

Enter the query below:

```
SELECT ProductName, UnitPrice, UnitsInStock, UnitPrice * UnitsInStock
FROM Products
```

When you run this, you will see, in the last column, the total value of the stock the company holds for each product type. Putting the “*” character between the two fields (“UnitPrice” and “UnitsInStock”) tells the computer to multiply, on each row, the numbers they contain.

Keeping things organised

You will notice the fourth column has no useful title. To fix that, add “AS StockValue”, to the query, as shown below.

```
SELECT ProductName, UnitPrice, UnitsInStock, UnitPrice * UnitsInStock AS StockValue

FROM Products
```

Run the query now, and you will see that the title “StockValue” has appeared. You can use the “AS” keyword to give a title to any column that appears in your query result.

Exercise – 2.1

(answer)

Change the query above so that “ProductName” appears as “Product”.

A note on units of currency

Once you become familiar with the database, it will become clear that it represents a US company that trades internationally, which also has a UK branch.

Database fields containing currency values, such as “UnitPrice”, don’t include units. They say “10.00” rather than “$10.00”, or “£10.00”, or similar.

The currency type isn’t important for this course. Currency values are referred to in the same way as other numbers, e.g. “this query lists products worth more than 3.50 per unit”, and you can think of them as whichever type of currency you prefer.

More math - operators

To do other calculations, simply swap the “*” for “/” to divide, “-” to subtract, and “+” to add. These characters are called operators, e.g. “/” is the division operator and “10 / 2” is a division operation.

Exercise – 2.2

(answer)

Get the name and total value of items on order for each product.

You can specify names for your results columns if you like, but from here on the answers won’t include renaming unless the exercise specifically asks for it.

Exercise – 2.3

(answer)

Assume all the products on order have just arrived. Show how many of each product you now have.

Constants

Sometimes, you may need to use a fixed value, or “constant”, in your query calculations. Suppose Northwind’s quality control department had signed out three units of each product for testing and now wants to return them. You want to see how many units you will have when they do. SQL can show you this easily.

```
SELECT ProductName, UnitsInStock + 3
FROM Products
```

You can use decimals the same way. For example, to calculate a 20% sales tax on unit price, see below:

```
SELECT ProductName, UnitPrice * 0.2
FROM Products
```

Exercise – 2.4

(answer)

Find a way to show the value of the 20% sales tax on each product, using division, instead of by multiplying.

SELECT without FROM

Before going any further, it is worth noting that you can also run a query without referring to any table at all. You can use “SELECT” without “FROM”. For example, try running:

```
SELECT 8
```

You will see a single “8” as a result.

The fact that “SELECT” works on its own is useful to know, especially when starting out with SQL. It provides a simple way to try out calculations in SQL. For example, to divide 10 by 2, use:

```
SELECT 10 / 2
```

Exercise – 2.5

(answer)

Find the answer to “seven times eight”.

Modulus

There is another math operator: “%”. This is called the modulus operator. It works out the remainder of a division. For example, 14 % 5 is 4, because 5 goes into 14 twice, with 4 left over; whereas 21 % 7 is 0, because 7 goes into 21 three times exactly, so nothing is left over. Try this out as below.

```
SELECT 14 % 5
```

And:

```
SELECT 21 % 7
```

Unlike the other math operators, you may never need to use the modulus operator. If you do want to practise working with it, try the next exercise.

Exercise – 2.6

(answer)

Look at the Products table again. If you store all units (currently in stock) for each product in groups of twelve, show how many units of each product will be left over.

Brackets

To do more complex calculations, you need brackets. To see why, run the examples below.

```
SELECT (2 + 2) * 5
```

And:

```
SELECT 2 + (2 * 5)
```

The results will be different: 20 and 12.

In the calculations above, the brackets change the order of the operations. This changes the results. Calculations inside brackets happen first, so the first example is 2 + 2 = 4 followed by 4 * 5 = 20, whereas the second example is 2 * 5 = 10 followed by 2 + 10 = 12.

Exercise – 2.7

(answer)

Get the combined (in stock and on order) total value of each product, e.g. if there are 10 goods in stock, and 10 on order, with a unit price of 5, then your query should return the value as 100.

Text Strings

The “+” sign has another use. If you have columns of text, rather than numbers, you use “+” to combine them. For example, in the Customers table, you have fields for address and city. Run the query.

```
SELECT Address + City
FROM Customers
```

It will show both values for each row, but in one column.

String literals

The query above will also display the text (string) values in each row without any formatting characters, such as spaces or commas, between them. We can fix this with a string literal, i.e. by running:

```
SELECT Address + ', ' + City
FROM Customers
```

The single quotes around the comma and space tell SQL to treat them as text, rather than as a field name or SQL keyword.

To be technical, when you combine two text strings in this way (joining one to the end of the other), you "concatenate" them. The "+" symbol is the concatenation operator in SQL.

Exercise – 2.8

(answer)

Change the query to include the customer contact name at the beginning of the result column. Follow the same format.

String literals with AS

Run the query below.

```
SELECT ‘some value’ AS whatever
```

This query works.

Now run:

```
SELECT ‘some value’ AS some title
```

This query doesn’t work. The reason is that there is a space in the title name. The computer reads the space after “some” as marking the end of the column title and then doesn’t know what to do with the seemingly random word “title” that appears at the end of the query.

If you want to have a space in your results column title, you need to work around this. The way to do that is to give the title as a string literal, e.g.

```
SELECT ‘some value’ AS ‘some title’
```

Now it works.

If you intend to re-use query results, e.g. as part of a subquery (we cover this later), it’s better to avoid creating titles with spaces in. SQL can only re-use such names if they are wrapped in square brackets, e.g. “[some title]”, and this makes the query less readable. A fairly readable alternative to spaces in multi-word titles is to capitalise the first letter of each word, e.g. “SomeTitle”.

Exercise – 2.9

(answer)

Add the title “Name and Address” to the result column of the query in the last exercise.

Functions

A function is something which returns a value. To see an example, run:

```
SELECT GetDate()
```

This will return the current date and time. Run it again and the value it returns will change to reflect the current time.

In this course, all function names will be written with a capital letter at the beginning of each word. This is to make them more readable, and help identify them as functions.

The “GetDate” function is perhaps a little unusual in that you don’t give it an input value. Generally, functions do take input values (arguments) and return output values based on them. One commonly-used function that does this is called “IsNull”. Before using it, however, we need to introduce the concept of “null”.

Null

A null value is the computer equivalent of a blank space on a paper form. It's like a field that hasn't been filled in yet, a value which doesn't exist. Run:

```
SELECT *
FROM Orders
```

Use the scrollbar in the bottom of the results window to look through the columns until you reach the "ShipRegion" field. Some of the values in this column are null. It appears that some countries which Northwind ships goods to don't use a region as part of the destination address. In this case, it is correct for the column to have a null value.

Northwind doesn't have many null values, but Northwind is a model database. That is, it has been simplified. In real-world databases, null values tend to be much more common.

The problem with null values is that they are not values. For example, what is 2 + null? Find out with the query below:

```
SELECT 2 + null
```

The answer is null. Two more than an unknown value is still an unknown value. This is correct but unhelpful. There are many cases where we will want to replace null with something more useful in our results. For example, if we wanted to combine our "ShipRegion" column with another string, we wouldn't want to include the null. In these cases, we use the IsNull function.

The IsNull Function

Type the query below.

```
SELECT OrderID, ShipCity, IsNull(ShipRegion,”) AS ShipRegion, ShipCountry
FROM Orders
```

The function takes two arguments, which have to be entered between its brackets, with a comma between them. The first is the name of the field, in this case “ShipRegion”. The function will check if this value is null. The second argument can be any value, in this case we used two single quotes, i.e. a string literal with no contents, known as an empty, or zero length, string. Run the query and you will see that the “ShipRegion” column now shows blank spaces where it used to say “null”. For each row, the function checks if the first argument’s value is null. If the value is not null, the function returns that value. If the value is null, the function returns the second argument’s value instead.

To see a really simple example, run:

```
SELECT IsNull(null, 0)
```

And:

```
SELECT IsNull(1, 0)
```

In the first query the function is given a null value, and outputs the specified alternative: zero. In the second query, the function is given a non-null value “1”, and so outputs that value.

Exercise – 2.10

(answer)

Modify the query of the Orders table above so that it returns two columns: OrderID and the other three values formatted as part of an address, i.e. with commas and spaces between. Name the address column “Order Address”.

Exercise – 2.11

(answer)

If you run the query in the answer to Exercise 2.10 above, you may notice that when there is no region for a row, the result shows two “comma plus space” separators together. This is because the query selects both string constants every time. It is possible to make the query put this formatting only between non-blank fields. You can do this by re-arranging how it selects text. See if you can work out how to do this.

Exercise – 2.12

(answer)

Modify the query in Exercise 2.11 above so that it doesn’t use the IsNull function around ShipRegion. What does the query return now?

This should show the usefulness of the IsNull function.

Other functions

There are many other functions for handling all kinds of data. There are functions for getting parts of text strings, or converting them between upper and lower case. There are functions for rounding numbers up and down and to specified numbers of decimal places. There many less well-known functions too. Even then, all of the functions mentioned so far in this book are only the ones used to do calculations within each row, the “in-line”, or “scalar” functions. There are other types of functions which will be covered later. It is useful to know how to use functions, and do calculations, particularly the ones we have covered here. However, SQL isn’t about processing one list of values into another list of values. It can do that, but as this course will show, SQL can do, and is designed to do, a lot more.

The next step

A major strength of SQL is in specifying **which** data to get. This chapter has covered selecting columns from tables, and processing their values. The next will cover selecting rows. Do you need a list of only your customers based in Germany? You need your query to return only certain rows. Do you want to see which orders shipped yesterday? That's a set of rows too. Read on to see how to get them.

WHERE - How to get the rows you want

Matching values

Did I hear that you need the records of only your customers based in Germany? You do? Then run this query:

```
SELECT *
FROM Customers
WHERE Country = 'Germany'
```

If you use the scrollbar at the bottom of your query results, you should be able to confirm that all the rows selected have the word "Germany" in the country column.

The "WHERE" keyword is followed by a condition, e.g. "Country = 'Germany'". The query checks the condition, e.g. compares the country field to the text string "Germany" for each row. If they are equal, the comparison returns true, i.e. confirms that the condition is met, and the row is included in the query result. If they are not equal, the comparison returns false, and the row is filtered out of the results.

Exercise – 3.1

(answer)

Get a list of customers in Berlin.

Exercise – 3.2

(answer)

Get only the contact names and phone numbers of customers in Berlin.

Comments

As the course progresses to more complex queries, and the exercises become more challenging, you may find that some of your queries don't return the result you expected, or maybe even return an error message instead. If the query is very long and complex, you may have trouble finding out where the problem is.

When this happens, there is a simple way to move, systematically, towards writing the query you need. It is known as "commenting out". Here is a (rather oversimplified) example to show how it works. Suppose you wanted to correct the query below:

```
SELECT *
FROM Customers
WHERE Country = 'Germamy'
```

If you run this query, you get just the column headers, but no results.

Now type two dashes "—" in front of the last line and run it again:

```
SELECT *
FROM Customers
—WHERE Country = 'Germamy'
```

The "—" at the beginning of the last line marks it as a comment. Comments are a way for people designing queries to add notes to them in plain English. These notes are ignored by the computer when it runs the query, so the last line is taken as a comment and ignored, but the rest of the query runs, and all the rows from the Customers table are returned.

Changing the query like this allows you to test parts of a query to see if they work as expected. It is much quicker than deleting lines and re-typing them later.

In this case, we can see that the "SELECT" and "FROM" lines are working correctly, so the problem must be with the "WHERE" line. Then we can look at that one line and find issue.

Whilst the problem in this line was obvious, the problem could be more subtle. The line could have read:

```
WHERE Country = ' Germany'
```

The space before the country name could be hard to spot, especially in a long query.

Queries can have any number of lines. How many lines a query contains can vary dramatically depending on the database. Several hundred lines is not an unrealistic amount to expect, especially if the query uses unions (covered later). In such cases, you can break down the query into individual parts to be tested, until you find the part that needs to be changed.

Case

When SQL Server matches text, it ignores the case of the letters. If you put “berlin” in your WHERE clause instead of “Berlin”, you will still get the same results.

Clauses

SQL has certain keywords it uses again and again: SELECT, FROM, WHERE, and other which are covered as the course continues. These keywords, together with the text that follows them, are known as clauses. That is, you have a SELECT clause, a FROM clause, and so on.

Let's use these clauses to query a different table.

Exercise – 3.3

(answer)

Get a list of the cities in Japan where you have suppliers.

Exercise – 3.4

(answer)

Try just once to get a list of the cities in India where you have suppliers, then check the answer section.

Matching Values Inexactly

Not all values will be an exact match. Often, you may want to see all rows where the values are within a certain range. Suppose you want to see a list of products with less than ten units in stock, so you can re-order. Run this query:

```
SELECT ProductName, UnitsInStock
FROM Products
WHERE UnitsInStock < 10
```

There are two things to note about this query. One is that we replaced "=" with a "<", the less-than sign, to compare the values differently. The other is that the "10", being a number, is not enclosed in single-quotes.

There are other comparison operators we could use, as follows:

Greater than: >

Greater than or equal to: >=

Less than or equal to: <=

Not equal to: <>

Exercise – 3.5

(answer)

Change the query above to show products with ten or more units in stock.

Exercise – 3.6

(answer)

Now show the names and prices of all products with a price above 21.35.

Matching on calculations

When calculations were introduced earlier, they were used with “SELECT”, but as mentioned, they don’t have to be. You can, for example, use calculated values with “WHERE”, for example:

```
SELECT *
FROM Products
WHERE UnitsInStock + UnitsOnOrder > 100
```

This returns the twelve products where the result of the sum above is greater than one-hundred.

Exercise – 3.7

(answer)

Show a list of products with current stock worth less than one-hundred.

Matching text values inexactly with LIKE

You may also need an inexact match for a text field. For example, if you want to find the types of coffee you sell you could run:

```
SELECT *
FROM Products
WHERE ProductName LIKE '%coffee%'
```

This gives you any rows where the ProductName field contains the word "coffee", even if there is other text before or after it. The "%" symbol, when used after "LIKE", in a text string, will match any combination of letters (including no letters at all). It is known as a "wildcard". You need to use it before and after the word you want to match, as there may be text in either of those places.

Exercise – 3.8

(answer)

Use a query to show a list of all product names beginning with "P".

Colour-coding

As stated before, it doesn't matter to the computer whether SQL is typed in capitals or not: it works either way. In this book, the SQL keywords, like "WHERE", which form clauses, are written in capitals, to help make clear what they are. If you look at your queries in SSMS, you may notice that these words are colour-coded blue.

However, the word "LIKE" is grey, not blue, indicating that it is different. If you look at the "=" in your query, you will see that this symbol is grey too, indicating that it is categorised together with "LIKE". This means that "LIKE" is an operator. It is a comparison operator, along with "=", "<>", ">", etc. In this book, operators are also shown in capitals, but their colour in SSMS identifies them as part of another clause, rather than the start of a new one.

Date values and literals

Before doing any more comparisons, there is another type of data, or "data type", that we need to cover. We have covered text strings and numbers. We will now cover dates. Dates in SQL are formatted like strings and treated like numbers (although SQL uses special functions to do calculations with them, rather than the usual math operators). For example, if you want to see all of the orders from February 1998 and before (yes, the Northwind sample database was made a long time ago), you can write a query as follows:

```
SELECT *
FROM Orders
WHERE OrderDate < '1998-03-01'
```

Note that although you enclose the date in quotes as you would with text, it is still used as a number for the comparison. SQL simply translates the "<" sign to "before" rather than "less-than" in this query.

Also, although the format is Year-Month-Day, or YYYY-MM-DD, it is possible that you will come across a database where the dates are stored in a different format, e.g. YYYY-DD-MM. If you are using, say SQL server, you can check this by looking inside the tables. However, in some systems, such as Microsoft Access, columns in tables can, confusingly, be set to display dates in different formats to how they are stored. This can lead to some easily overlooked types of error.

Exercise – 3.9

(answer)

List the records of all employees hired from 1993 onwards.

Dates and times

In Northwind, the database columns such as order date store nine zeros, i.e. “00:00:00.000”, after the end of the dates. These are time values. They are stored in the format Hours-Minutes-Seconds-Milliseconds. In this case, times are clearly not being recorded, and the database is storing the time part of each date as zero. Effectively, each date-time value recorded is for the beginning of the date specified, i.e. midnight.

Some database fields are set up with dates only, but many record times. This affects the way they should be queried, as described below.

Time values provide a good reason not to use “=” with dates. Suppose you write a query’s “WHERE” clause as:

```
WHERE SomeDateField = ‘2016-01-01’
```

If your field contains a time value too, you will only see records where the time value is exactly midnight. You may have a thousand records that day, between 9AM and 5PM, but you will see none of them in your query result.

What you need is a way to see results across the whole range of times within the day. The next chapter will introduce some SQL keywords which will allow you to do just that, and more besides.

AND / OR / NOT – How to get the rows you want more precisely

AND - Ranges

So far, all of the queries have selected rows according to whether they met one condition. This chapter will show you how to combine multiple conditions to focus your query on the rows you need. In doing so, it will also provide a way to deal with dates.

In the last chapter, in became clear that we couldn't reliably select dates using the "=" operator. Any date with a non-zero time-value will never match a date in the format: "YYYY-MM-DD", as its value will always be between two such dates.

What we need, then, is a "BETWEEN" operator.

```
SELECT *
FROM Orders
WHERE OrderDate BETWEEN '1997-01-01' AND '1997-01-02'
```

Run this query. You get a list of orders in-between the two dates, but you also get a record with an order on the query's second date. This is because the "BETWEEN" operator is "inclusive" of the two date values.

However, in some versions of SQL, "BETWEEN" may not include the dates. It may be "exclusive" and therefore, in this case, with our time-free date fields, it may return no records at all.

Ideally, we want a query that will work as we expect, in any version of SQL, whether or not the date-time field contains time values. In this case, we want a result that includes all the records we got in the last query, except those with an order date of exactly '1997-01-02'. Try the query below:

```
SELECT *
FROM Orders
WHERE OrderDate >= '1997-01-01'
AND OrderDate < '1997-01-02'
```

This gives us only the records on or after midnight of the first day, and before, but not on, midnight of the next. It is also very clear, from reading the query, how it works.

This query uses the "AND" operator to extend the "WHERE" clause (the SSMS colour-coding explained earlier should make this easier to see). Each date comparison for each row now returns a value of true or false, this time to the "AND", which only returns true to the "WHERE" clause as a whole if it gets two true inputs, i.e. condition 1 is true AND condition 2 is true.

Operators like "AND", which process values of true/false, are known as logical, or Boolean, operators, as opposed to, for example, the math, or arithmetic operators, like "-"

and “/”.

The use of “AND” here allows you to get values in a date range. However, its use isn’t limited to working with dates. It can be used with any combination of conditions.

Exercise – 4.1

(answer)

Get a list of products with at least ten but less than twenty units in stock. Show the stock levels in your results.

Filtering results on a range is just one use for “AND”. You can also use it to filter results on separate fields, as in the next exercises.

AND - Multiple conditions

Exercise – 4.2

(answer)

Get the records for orders shipped to Brazil from 1997 onwards.

Exercise – 4.3

(answer)

Get the orders shipped to Brazil in 1997 (you can use “AND” as many times as you like, just keep adding the operator followed by its condition to the end of your “WHERE” clause).

OR – Being flexible

What if you want to get results that meet **either** of two conditions? For example, you may want a list of orders shipped to the US and Canada. You can't use the "AND" operator for this: it will only return orders that went to both countries, i.e. no orders. You need an "OR" operator.

```
SELECT *
FROM Orders
WHERE ShipCountry = 'USA'
OR ShipCountry = 'Canada'
```

The "OR" takes the results of the two comparisons and, if either one is true, returns a result of true. The row is then included in the results.

Let's apply this to another table.

Exercise – 4.4

(answer)

The UK sales team are visiting the Seattle office: list the records of any employees you would now expect to be in Seattle.

Brackets

If you think back to the chapter on “SELECT” and math, when brackets were introduced, you may remember that operations inside brackets happen before those outside of brackets

Try the following query:

```
SELECT *
FROM Customers
WHERE ContactTitle = ‘Owner’
AND (Country = ‘USA’
OR Country = ‘Mexico’)
```

This query returns a list of customers where your contact is the business owner and which are based in Mexico or the US. The query compares the country field first, and if either of the comparisons is true, it also checks the “ContactTitle”. If this is “Owner” then it includes the row in the result set.

Now run the query below.

```
SELECT *
FROM Customers
WHERE (ContactTitle = ‘Owner’
AND Country = ‘USA’)
OR Country = ‘Mexico’
```

This gives a different result. This time you still get all the US customers where your contact is the owner, but your get all the Mexican customers regardless of what is in their “ContactTitle” field. This is because the “AND” operation between the top two comparisons now happens first.

All that has changed in the above query is the position of the brackets, but this affects what you are asking the query to do, which changes the result.

The next few exercises are about putting together a similar query, one step at a time.

Exercise – 4.5

(answer)

Get a list of customers outside the US (the “not equal to” operator is “<>”).

Exercise – 4.6

(answer)

Get a list of customers where your contact has a title beginning with “Sales” or “Marketing”.

Exercise – 4.7

(answer)

Combine the two queries above - get a list of the names and phone numbers of your non-US contacts with those “Sales” and “Marketing” titles. Include their titles and countries in the results too, so you can check easily whether the query is working correctly or not.

NOT – Saying what you don't want

Suppose you have worked through your list of US and Mexico-based owners. Now you want a list of your other US and Mexico-based contacts. You could use the query below.

```
SELECT *
FROM Customers
WHERE ContactTitle <> 'Owner'
AND (Country = 'USA'
OR Country = 'Mexico')
```

You could also use this:

```
SELECT *
FROM Customers
WHERE NOT ContactTitle = 'Owner'
AND (Country = 'USA'
OR Country = 'Mexico')
```

There is no difference. The "NOT" operator returns the opposite of a comparison result: true for false and false for true. Using it above with "=" is the same as using "<>".

However, the advantage of the "NOT" operator is that it can be applied to multiple comparisons at once, i.e. to those in brackets. For example:

```
SELECT *
FROM Customers
WHERE ContactTitle = 'Owner'
AND NOT (Country = 'USA'
OR Country = 'Mexico')
```

This query retrieves the records for owners based outside the US and Mexico.

Exercise – 4.8

(answer)

Get a list of the names and numbers of your non-US contacts whose titles don't begin with "Sales" or "Marketing".

Exercise – 4.9

(answer)

Get a list of the names and numbers of your contacts outside of the US and Mexico, whose titles don't begin with "Sales" or "Marketing".

DISTINCT - How to remove duplicates

Run the query:

```
SELECT DISTINCT Title
FROM Employees
```

The “DISTINCT” keyword makes a query remove duplicate values. In the example, you may have several employees with the same job title, but this query will return each title only once. It gives you a list of the job titles that exist in the company.

Exercise – 5.1

(answer)

List the countries in which your suppliers are based.

ORDER BY - How to sort your rows

If you run a query like this:

```
SELECT *
FROM Products
```

You will get the records in the order they are stored, i.e. those with lower ProductID values will be nearer the top of the results. However, you may want them ordered differently.

If you want to easily compare how many of each item you have in stock, you could sort your query according to the values in the field “UnitsInStock”, as below.

```
SELECT *
FROM Products
ORDER BY UnitsInStock
```

This gives you a list with the least-stocked items at the top. However, it would probably be more useful to have the most-stocked items at the top, as these are likely to be more important to the company. Change the query to:

```
SELECT *
FROM Products
ORDER BY UnitsInStock DESC
```

This will sort the rows in descending order of the “UnitsInStock” values. The default order is ascending, so this query gives you the reverse order of the one above.

Next, get the top thousand rows of the table (as you did near the start of this book) and notice how, towards the bottom, the supplier IDs are not grouped together. Suppose you want to see products from each supplier together. Run this query:

```
SELECT *
FROM Products
ORDER BY SupplierID
```

Notice how rows with the same supplier ID are now always next to each other.

If you want to order your query results by more than one field, you can use commas, i.e. run:

```
SELECT *
FROM Products
ORDER BY SupplierID, UnitsInStock DESC
```

Notice how this query sorts rows by the leftmost field, “SupplierID”, first, so that records with the same Supplier ID form blocks of records, and then sorts rows within each block

by their “UnitsInStock” value. Also note that the “DESC” keyword has been applied to the ordering of “UnitsInStock” only, i.e. it has been applied only to the field it follows, not the whole “ORDER BY” clause. The “SupplierID” column is still sorted in ascending order.

Exercise – 6.1

(answer)

Group the products by category, then in ascending order of price per unit. Show the product name, category ID and unit price in your results.

TOP - How to take a small sample of rows

At the start of this book, you found the Products table, and selected the top thousand rows. This is a useful built-in option in SQL Server Management Studio. However, the SQL to do the same thing is very simple, and worth knowing, since you may not always be using SSMS and its built-in options. Run the query below:

```
SELECT TOP 1000 *
FROM Products
```

This will return the first thousand rows selected from the Products table. If there are fewer than one-thousand rows, it will return the rows that are there. If you don't want a thousand rows, just change the "1000" to the number of rows you do want.

One advantage of using "TOP" to select a small number of rows is that selecting more rows takes more time. The computer takes time to load them all. Why wait?

It is worth knowing that "TOP" gets you the first rows selected, rather than the first thousand rows stored in the table. It allows you, should you wish, to run queries like this:

```
SELECT TOP 10 *
FROM Products
ORDER BY ProductName
```

You can then view a different sample of records, in this case the top ten products, ordered alphabetically by name. The rows are put in order before the top ten are taken.

Exercise – 7.1

(answer)

List the names and prices of the ten cheapest products.

Bottom

As there is a “TOP”, you might expect there to be a “BOTTOM”. SQL Server doesn’t have this, but you can often still get the last rows of a table. The Products table has a ProductID field, containing an ID number which increases, by a value of one, for each new record added, to ensure the ID field is unique for each record. Knowing this, to get the bottom ten rows, you can use:

```
SELECT TOP 10 *
FROM Products
ORDER BY ProductID DESC
```

This will sort the data in the reverse order to how it is stored. Effectively, this makes an upside-down version of the table, and returns the top ten rows, which were at the bottom of the original table, from that.

GROUP BY - How to summarise row data

We have already covered how to do calculations within a single record. You could now, for example, calculate the stock value held for each type of product by running:

```
SELECT UnitsInStock * UnitPrice AS ValueInStock
FROM Products
```

However, if you want to know the total value of stock in the warehouse, this query still leaves you with a lot of adding up to do. What you need is a way to do calculations using whole **columns** as input. Let's look at how to do that.

Sum

Run the following query.

```
SELECT Sum(UnitsInStock)
FROM Products
```

The “Sum” function adds all the values in the named column and gives a total.

To get the total value of the stock held, run the query:

```
SELECT Sum(UnitsInStock * UnitPrice)
FROM Products
```

Operations inside function brackets are no different from operations inside any other brackets, in that they are completed first. Here, this means that, for each row, the values in the two fields are multiplied, and all of the results are then added together by the “Sum” function.

Aggregate functions

The “Sum” function is known as an aggregate function, because it brings together (aggregates) all of the input field’s values into one value, whereas functions like “IsNull” are in-line, or “scalar” functions, because they work within rows, rather than across them. There are other aggregate functions than “Sum”, such as the one in the query below:

```
SELECT Count(*)
FROM Products
```

This simply returns the number of records in the Products table.

To get the highest (maximum) value in a column, you can use a query such as:

```
SELECT Max(UnitPrice)
FROM Products
```

To get the lowest (minimum) value, use:

```
SELECT Min(UnitPrice)
FROM Products
```

Of course, the query above doesn’t tell you what the lowest-priced product actually is. To get the product name as well as the lowest price, we could use a subquery. We’ll cover subqueries later; they have their own chapter.

You can make your query filter records, before aggregating them. This allows you to work with parts of a table, such as the records of products in a particular category, or of product types which you currently have in stock.

The query below does the latter:

```
SELECT Count(*)
FROM Products
WHERE UnitsInStock > 0
```

The “WHERE” clause is applied to the query before the results are counted.

Exercise – 8.1

(answer)

Find the total number of units you have in stock for discontinued products (these have a “1”, meaning “true”, in their “Discontinued” field).

Exercise – 8.2

(answer)

Find the total number of units you have in stock for products in beverages (CategoryID = 1).

Other Aggregate Functions

There are other aggregate functions in SQL. For example, “Avg” calculates the average value of a group of results from a number column. However, they are all used in the same way within queries, so rather than just cover more aggregate functions, this course will now focus on more advanced ways of using aggregation.

Grouping

The last exercise returned one figure for one category. However, if you wanted to see the totals for every category, you would not want to have to run a query once for each of them. This is where grouping is useful. Run the query below.

```
SELECT CategoryID, Sum(UnitsInStock) AS UnitsInStock
FROM Products
GROUP BY CategoryID
```

The "GROUP BY" clause tells the query to apply the aggregate function, in this case "Sum", to rows with the same values in their category fields, so the query returns a result for each category.

To return fields in a query with grouping, those fields must have only one possible value for each group. In this case, the "CategoryID" field has only one possible value per row, because it's in the "GROUP BY" clause. That tells the query to put each category ID in its own group. The "Sum" field has one value per group, which it gets by adding up all the "UnitsInStock" values in the category.

However, if we wanted to include "ProductName" in the "SELECT" clause of this query we couldn't do it, because there are many product names for each category. With many possible values, the query cannot return any of them.

Exercise – 8.3

(answer)

Get the number of units on order from each supplier. Show the supplier IDs in your results.

Using ID fields limits the usefulness of the results to anyone wishing to read them. However, you can get the names associated with those IDs in your results. We'll cover how to do that in the chapter on inner joins.

Exercise – 8.4

(answer)

Get the total value of those units, for each supplier.

Exercise – 8.5

(answer)

Get the price of the most expensive item in each category.

Exercise – 8.6

(answer)

Get the price of the cheapest item from each supplier.

Grouping by multiple fields

It is also possible to group by more than one field, as below.

```
SELECT SupplierID, CategoryID, Count(*) AS ProductCount
FROM Products
GROUP BY SupplierID, CategoryID
ORDER BY SupplierID, CategoryID
```

This query shows how many products each supplier provides in each category.

HAVING - How to get the rows you want, after summarising

Suppose we want a list of categories that contain less than ten products. To see how many items each category contains, we could do this:

```
SELECT CategoryID, Count(*)
FROM Products
GROUP BY CategoryID
```

At this point, it looks like the results could be filtered by applying a “WHERE” clause to the “Count(*)” field. However, the “WHERE” clause is applied to the fields of the table in the “FROM” clause **before** the rows are counted. It can’t filter by a count that doesn’t exist yet.

To filter by aggregate values, such as those in “Count(*)” above, you can use “HAVING”. It works like “WHERE”, but for aggregated values. Run the query below.

```
SELECT CategoryID, Count(*)
FROM Products
GROUP BY CategoryID
HAVING Count(*) < 10
```

Now you should see the results you need.

The count is included in the “SELECT” clause above because it helps confirm that the results are correct. However, if you wanted just the list of categories, you could leave out that part of the query, as below.

```
SELECT CategoryID
FROM Products
GROUP BY CategoryID
HAVING Count(*) < 10
```

The “SELECT” clause specifies data to display. The “HAVING” clause specifies criteria for excluding data. You don’t have to display data in your results in order to filter them by it: you only need the aggregate calculation in the “HAVING” clause. For the exercises below, however, you may wish to include the aggregate columns in the results you display, so you can more easily see whether your queries work as intended.

Exercise – 9.1

(answer)

List the IDs of categories with more than 100 units in stock.

Exercise – 9.2

(answer)

List the IDs of suppliers which have only one product.

Exercise – 9.3

(answer)

List the categories which don’t have units on order.

Applying this to other tables

Take a look at the “Orders” table to see what data is specific to each order: it has one row per order, and therefore, a unique order ID for each row. Now take a look at the “Order Details” table to see data specific to each product in each order, it has one row per product per order. In one order, there can be many products, so in the “Order Details” table, there can be many rows with the same order ID. This is known as a one-to-many relationship between the tables, or rather, the records in them.

Grouping and aggregation are useful for dealing with data organised this way. To complete the following exercises, group and aggregate the data in the Order Details table.

Note that when referring to the table, you will need to use the format:

```
FROM [Order Details]
```

The square brackets “[]” tell SQL that the text within them is all one name, despite the space in the middle.

Exercise – 9.4

(answer)

List the orders (by ID) containing less than fifty units.

Exercise – 9.5

(answer)

Get a list of orders in which any of the products have been discounted by more than 20% (0.2). Show one of these higher-than-20% discounts for each order.

Multiple aggregate functions

You can use more than one aggregate function at a time, e.g.

```
SELECT OrderID, Count(*), Sum(Quantity)
FROM [Order Details]
GROUP BY OrderID
```

This (slightly contrived) example tells you both the total number of products and the total number of units, in each order.

Exercise – 9.6

(answer)

Get a list of orders, their cash values (before any discount) and the number of units they contain. Add your own column titles to the query result.

Alias - How to query with less typing

Column Alias

An alias is an alternative name, of your choice, for a table or column. You have already used aliases several times to rename the column headings as they appeared in your query results, e.g.

```
SELECT ContactName AS Contact
FROM Customers
```

You don’t have to use the “AS” keyword at all. Remove the “AS” and run the query again.

```
SELECT ContactName Contact
FROM Customers
```

This gives exactly the same result. However, the query with “AS” in it is clearer to read. The extra word makes the fact that you are using an alias much more obvious.

Table Alias

In the coming chapters covering joins, we will run queries on multiple tables at once. This creates a problem for SQL in that, if two tables have the same field name, and you ask your query to give you that field, SQL doesn't know which table you want to take the field from. If you ever try to run a query where this is an issue, you will get an error message telling you that the field name you requested is "ambiguous". SQL solves this problem by letting you specify the table name of the field you need as well. We can see how this works, even with one table.

```
SELECT Customers.CustomerID, Customers.CompanyName
FROM Customers
WHERE Customers.CustomerID LIKE 'A%'
```

Although we don't need to specify a table when there is only one, this example does show how a table is specified (before the field name, separated from it by a "."). However, it also shows that this way of specifying tables is inconveniently long to type out, or even to read through easily. The query might contain twenty fields, in the "SELECT" clause, five more in the "WHERE" clause, and take data from five different tables. Such a query quickly becomes unreadable and tedious to type, even with copy and paste. This makes it harder to understand, or to re-use by changing its code. SQL solves this problem by allowing the use of aliases on table names, to shorten them, as below.

```
SELECT c.CustomerID, c.CompanyName
FROM Customers c
WHERE c.CustomerID LIKE 'A%'
```

This query returns any records in the "Customers" table which have a "CustomerID" field beginning with "A". The "c" is the alias for the customer table. The table alias is made by putting the alias name ("c") after the real table name ("Customers") in the "FROM" clause. If you take this part away, as below, the query fails to run.

```
SELECT *
FROM Customers
WHERE c.CustomerID LIKE 'A%'
```

In queries using a subquery, join, or both, the use of table aliases can save time and make it easier to read through a query you just wrote.

INNER JOIN - How to combine related records from different tables

Inner joins return a list of matching records. Typically, they are made by joining two tables on the unique ID field of one, such as the ID field of the Region table, and a reference to that field in the other table.

As an example, run the query below:

```
SELECT *
FROM Region
INNER JOIN Territories
ON Region.RegionID = Territories.RegionID
```

Note that the result of this query is fifty-three rows, the same as the number of territories. Each territory has only one region (the region it is in) and that is the region that appears next to it in the results.

We can shorten the query by using table aliases, as below:

```
SELECT *
FROM Region r
INNER JOIN Territories t
ON r.RegionID = t.RegionID
```

So far, we have used "*" to get all of the rows from both tables. This is just for simplicity, however. If we want to limit the number of columns returned, we can. Try changing the "SELECT" clause of the shortened query to each of those below, and then running it again.

```
SELECT t.TerritoryDescription, r.RegionDescription
```

This gives us the specified columns, one from each table. Note that we have also used the aliases "r" and "t" to specify which tables to take the columns from. We don't always need to do this, but it is good practice. If any of the column names we wanted to select appeared in both tables, and we didn't use the alias, then SQL would not know which one to pick, and would return an error message.

Now try running the query below:

```
SELECT t.*, r.RegionDescription
```

And:

```
SELECT t.TerritoryDescription, r.*
```

You will see that the first version above returns two columns, the next returns all the columns from Territories (and one from Region) and the last returns all columns from Region (and one from Territories).

Exercise – 11.1

(answer)

Return a list of products and their category names. Include all columns from the Products table and only the category name column from the Categories table.

Exercise – 11.2

(answer)

Return a list of orders and the names of the companies which placed them.

Exercise – 11.3

(answer)

Earlier, in the chapter on the grouping, there was the exercise (8.3) below:

> Get the number of units on order from each supplier. Show the supplier IDs in your results.

Only the supplier ID was shown in the results. Here is the query that answers that exercise.

```
SELECT SupplierID, Sum(UnitsOnOrder)
FROM Products
GROUP BY SupplierID
```

Now that we have covered joins, you know enough to replace the supplier ID with the supplier name. Try it. If you can't work out how to do this, then just use the answer to this exercise as an example, and the next exercise will give you another chance to put this technique into practice.

Exercise – 11.4

(answer)

Also in the chapter on the grouping, there was the exercise (8.5) below:

> Get the price of the most expensive item in each category.

The query was:

```
SELECT CategoryID, Max(UnitPrice)
FROM Products
GROUP BY CategoryID
```

Can you make it return the results with the category names, instead of the IDs?

Other ways to join

You can join tables in more ways than the one shown above. You can use an “AND” in your “ON” condition, to only join tables with two matching sets of fields. You can use the multiple “JOIN … ON …” clauses, one after the other, to join multiple tables together. You can join tables on other conditions than a field in one matching a field in the other.

Although joins are usually made on condition of a field from each table having an equal value to the other, you can join tables however you like. You can join records on the basis that the values are not equal, “<>”, if you want. However, you are more likely to join on something like a date field, where the dates are the same, but the times are not. For example, if you look in the Orders table, you will see that the dates do not include time values, apart from “00:00:00.000” i.e. midnight. The time the orders were placed is in fact not being recorded. If you were trying to join this table to a table which had dates with the times recorded too, you could use an operator like “BETWEEN”, a ”>=” condition joined by “AND” to a “<” condition to simulate between, or some way of rounding the field with the time value. The “ON” clause, like the “WHERE” clause, is a filter. It is a small step from using “WHERE” to using “ON”.

Outer Joins – How to get unrelated records

The problem with inner joins

Before covering outer joins, we need an example to highlight the limitations of inner joins. We'll take our example from the results of the query below, so run it before reading on.

```
SELECT *
FROM Customers c
INNER JOIN Orders o
ON c.CustomerId = o.CustomerId
```

Your results table will include a list of all the orders in the database, alongside the records of the customers which made them. Note that the customer record part of each row is duplicated down the results table. Each customer may make several orders, so one customer record will be matched to several order records, and duplicated for each of them. Each order has only one customer, however, so only appears once in the query result. You can confirm this by checking the order table contents: it will have the same number of rows as your query result.

All orders are accounted for: so far, so good.

Suppose you are now looking through the list for a particular customer and you can't find them. You know they are in the Customers table. Has something gone wrong with your query?

In the next exercise, check that all of the customers are on the list too. You'll need a way to get a list of distinct customer names from the result, so you can compare it to the records in your customer table. Fortunately, we already covered the keyword ("DISTINCT") that lets you do that.

Exercise – 12.1

(answer)

Get a list of the company names present in your query result (no duplicates).

What it all means

If you compare the list to your Customers table, even just by the number of rows in each, you will see that the list is missing a couple of records. If you were to take the customer IDs of those records, and search for them in the Orders table, you wouldn't find them. They are not there. Those customers have no orders.

This is also why they don't appear in the query result. The join depends on customers and orders having matching customer IDs. If there are no orders for a particular customer, the customer record cannot match anything and is never included in the results at all.

In this sample database, these kinds of situations are hard to find. In the real world, they are common. Here we have two customers who have never placed orders. Maybe they are new customers.

Another common, real-world example of one table having unmatched records in another would be in tables of sales or orders, and refund tables. If a sale is refunded it would have a matching record in the refunds table; if not refunded, there would be no refund and no record. Therefore, doing an inner join on the two tables would result in a list of refunds, and their sales, but not the un-refunded sales.

Left Outer Joins

Coming back to our query in Northwind, if you need to see the unmatched records too, you use a different type of join: an outer join, as below:

```
SELECT *
FROM Customers c
LEFT OUTER JOIN Orders o
ON c.CustomerId = o.CustomerId
```

You may notice from the row count that this query gives you a slightly larger set of results. It now includes the two customer records you were missing before. Let's make it more specific. Add a "WHERE" clause, as below, and run it again.

```
SELECT *
FROM Customers c
LEFT OUTER JOIN Orders o
ON c.CustomerId = o.CustomerId
WHERE o.CustomerId IS null
```

This time, you will only see two records: the unmatched customer records, alongside the Orders table columns. The Orders columns will be full of null values, as there is no order data to put in them.

An **outer** join includes unmatched records in the query result. A **left** outer join includes unmatched records from the left hand table. This will be the one before the "JOIN" keyword in the query, i.e. the table name to the left of "JOIN" if the query is written all on one line. In this case, it's the Customers table.

Right Outer Joins

A **right** outer join does the reverse of the left outer join. It includes unmatched records from the right hand table i.e. the table specified after the "JOIN" in the query, in this case, the Orders table. To see how this works, edit your query to make it into the one below. The table names have been swapped, but the join type has been changed too, so the query will still return unmatched records from the Customers table. Run the query now.

```
SELECT *
FROM Orders o
RIGHT OUTER JOIN Customers c
ON c.CustomerId = o.CustomerId
WHERE o.CustomerId IS NULL
```

You will see two rows, as before. Scroll to the right of the results, and you will see the customer fields, to the right of the order fields. The results are identical, apart from the table columns being swapped left to right, and if you really wanted the columns in the original order, you could specify that with "SELECT c.*, o.*". Therefore, you don't need to use right outer joins at all if you don't want to, because a left outer join can do the all the same things.

Inner Joins vs. Outer Joins

Let's continue with left outer joins. Change the last query, so that it uses a left outer join (and no "WHERE" clause) as below:

```
SELECT *
FROM Orders o
LEFT OUTER JOIN Customers c
ON c.CustomerId = o.CustomerId
```

Run the query. This time the **Orders** table is on the left side of the left join. The query will return all the orders, including orders without customers.

Now add this line to the end of the query:

```
WHERE c.CustomerId IS NULL
```

This would return only the orders without customers matched to them. It returns no results, so we know there are no such orders. There are no orders without customers, which is to be expected as that should not be possible.

In this case, the query (with or without the "WHERE" clause) returns the same results as it would with an inner join.

Exercise – 12.2

(answer)

There is a table called "EmployeeTerritories" which shows which territories are assigned to which employees. Write a query that returns all the territories, and the ID of any employee assigned to them.

Exercise – 12.3

(answer)

Some of the territories do not have assigned employees. Change the query so that it only returns results for those territories.

Exercise – 12.4

(answer)

Change the query so that it returns only results for those territories that do have employees assigned. Make sure you check the answer for this, even if you think your results are correct.

Exercise – 12.5

(answer)

Change the query so that it shows the names of those employees alongside the territory

description. Remember that you can join more tables just by putting “… JOIN … ON …” statements one after the other, in the format:

…

FROM TableX AS x

INNER JOIN TableY AS y

ON x.FieldA = y.FieldA

INNER JOIN TableZ As z

ON y.FieldB = z.FieldB

…

Full Outer Joins

It is even possible to return unmatched records from both sides of the join at once, by using “FULL OUTER JOIN”. It is probably fair to say that these are not commonly used. Indeed, some systems, like Microsoft Access, don’t support full outer joins at all.

Self-Join - How to join a table to itself, and why you would want to

The Employees table is a bit different from the other tables: it references itself. Employees report to other employees with records in the same table. If you wanted to, for example, see a list of employees next to their managers names, you would need to join the table to itself. This is called a self-join. Of course, a table can't really be in two places at once, so what we do is join two (temporary) copies, or "instances", of the table. We use "AS" aliases to make a staff copy and a manager copy. We then join these as we would any normal pair of tables.

To see how it works, run the following query:

```
SELECT staff.FirstName + ' ' + staff.LastName AS Staff,
boss.FirstName + ' ' + boss.LastName AS Boss
FROM Employees staff
INNER JOIN Employees boss
ON staff.ReportsTo = boss.EmployeeID
```

Note that only eight of nine employees appear in the results. One employee (the head of the company) doesn't report to anyone, so isn't shown.

Exercise – 13.1

(answer)

Use an outer join to make that employee appear.

Cross Join – getting all possible combinations of table rows

Before starting on this topic, I should point out that this type of join is not commonly used, and you won't need to know about it to understand the later chapters. You could skip these few pages, and just start the next chapter: "UNION / UNION ALL". However, cross joins are easy to do, help show how joining tables in SQL actually works, and do have some practical value, so they are included here for anyone interested.

Introducing cross joins

A cross join is the simplest type of join. It is the join you make when the rows in your tables don't need to reference each other. The Northwind database doesn't have any such tables. However we can still use the tables it does have, to demonstrate how cross joins work, before introducing the situations in which you might want to actually use them.

Let's take a look at the tables to be used in the join. Look in the Region table and note its four rows. Look in the Territories table and note that it has fifty-three rows, each with a Region ID from one to four. We can conclude that Northwind operates in four regions, which are broken down into a combined total of fifty-three territories.

How cross joins work

To see how cross joins work, run this query:

```
SELECT *
FROM Region, Territories
```

This query gives you a list of two-hundred-and-twelve rows. These rows contain all the columns from both tables. If you scroll down, you will see that each of the four Region IDs has a quarter of the rows, and if you do the math, you will see that 4 x 53 = 212.

This query gave you every possible combination of rows, each of the fifty-three regions for each of the four territories. It didn't check that those regions were in those territories, because we didn't tell it to.

Joins combine two sets of rows, and return any combinations which meet one or more conditions. In the inner join we looked at earlier, that condition was that the territory has a Region ID which matches the region. In a cross join, we want all combinations, so if we were forced to use a condition, we would use one that returned true for every combination of rows. To show how this is possible, we can simply use a condition like the one in the "WHERE" clause of the query below.

```
SELECT *
FROM Region, Territories
WHERE 1 = 1
```

If you run this query, you will see the exact same result as before. One always equals one, so the condition is always met.

This query gives the same results as a cross join, but there is still one more thing we need to do to fully show how cross joins work. We need to use the actual SQL join clause to write the query. This is simple to do, just replace the comma between table names with "JOIN" and replace "WHERE" with "ON".

```
SELECT *
FROM Region
JOIN Territories
ON 1 = 1
```

Running this query gives you the exact same result as before, but this time it is very clear in your query that you are joining tables, and on what condition. All other joins use variations of the query syntax above, simply adding the type of the join before the word "JOIN", e.g. inner, left outer, right outer, full outer.

If we wanted to make this an inner join, we would simply put "INNER" before "JOIN" and change the filter in the "ON" clause. Because the "ON" clause, like the "WHERE" clause, is just a filter.

Uses of cross joins

Suppose you have a list of dates and a list of time slots, each in their own database table. Maybe they are the dates and times of performances in a theatre. You want a list of all possible date and time combinations. You need a cross join.

Suppose your theatre has twenty rows, from A to T, and twenty seats in each, from 1 to 20. You have, or can easily make, a table with a list of rows in, and another with a list of seat numbers. You can use a cross join to make all combinations of row and seat number: A1, A2 … B1, B2 … etc. You then have a list of the seats available for a performance.

Suppose you want to combine your list of seats with your list of time slots to generate a list of tickets you can sell. Again, you need a cross join.

Exercise – 14.1

(answer)

Use SQL to calculate how many combinations of rows there would be if the Shippers and Customers tables were cross-joined.

Exercise – 14.2

(answer)

Actually join the two tables.

Exercise – 14.3

(answer)

Use SQL to count how many rows the result has.

UNION / UNION ALL - How to combine rows from two tables

UNION ALL

Some tables are not related to each other in the sense that they can be cross-referenced by, for example, comparing fields containing ID numbers. Joining them may not make sense. However, they may still have things in common, and you may want to query them together because of this. For example, you may want a list of the cities, or countries, where you have trading partners, i.e. suppliers or customers. However these are listed in two different tables. It would be useful to have a way to add the tables' rows together. There is a way, as shown below:

```
SELECT CompanyName
FROM Suppliers
UNION ALL
SELECT CompanyName
FROM Customers
```

The "UNION ALL" statement above adds the results of the Customers query onto the bottom of the results of the Suppliers query. You now have all the names in one list.

However, this result doesn't show you whether the companies are customers or suppliers. Try this instead:

```
SELECT CompanyName, 'Supplier' AS CompanyType
FROM Suppliers
UNION ALL
SELECT CompanyName, 'Customer'
FROM Customers
```

The string literals now tell us which query, and therefore which table, the rows came from. We only need the "AS" part in the top query, because the column can only have one title.

Note that for "UNION" to work, each field in the top "SELECT" clause must have a field of the same type directly below it in the lower "SELECT" clause. You could not, for example, make a union of the supplier ID and the customer company name. One is a number and the other text, so SQL would return an error.

Exercise – 15.1

(answer)

Change the query to also include country and city fields.

UNION

There is also a “UNION” keyword. It works like “UNION ALL”, but it also removes duplicate rows (i.e. rows where all the values are the same as those of an earlier row in the result) from the combined set of results, whereas “UNION ALL” doesn’t.

For example, the query below uses “UNION ALL”

```
SELECT Country
FROM Suppliers
UNION ALL
SELECT Country
FROM Customers
```

If you run this, you will see some countries appear several times. If you delete the “ALL” from the query above and run it again, you will see that the list now contains distinct values, no duplicates. Change the query to the one below, and run it.

```
SELECT Country, ‘Supplier’
FROM Suppliers
UNION
SELECT Country, ‘Customer’
FROM Customers
```

Note that although you now see some countries appear twice, this is only because their rows have different values in the second column, so the rows as a whole are not duplicated, and “UNION” leaves them in.

There are other ways to combine two result sets, which will be covered next, but “UNION” and “UNION ALL” are probably the most commonly used.

INTERSECT – How to get rows that are only in both tables

Run the query:

```
SELECT Country
FROM Customers
INTERSECT
SELECT Country
FROM Suppliers
```

You will see a list of countries. These are the countries which appear in both tables. As with “UNION”, there are no duplicate rows in the results; they have been removed.

Also, as with “UNION”, when using “INTERSECT” the number and type of rows must match up. The “EXCEPT” clause is the same. We’ll cover that next.

EXCEPT – How to get rows that are only in one table

Run the query:

```
SELECT Country
FROM Customers
EXCEPT
SELECT Country
FROM Suppliers
```

You will see a list of countries. These are the countries where Northwind has customers but no suppliers. Again, duplicates have been removed.

This keyword has another use. Its lets you compare two query results and check if they are the same. If, during any exercise, you wrote a query which differed from the one in the answer, but which you still believe was correct, now is your chance to test whether you were right! First, make sure your query has the same columns, in the same order, so "EXCEPT" doesn't give you an error. After that, simply run the following SQL:

```
YourQueryHere
EXCEPT
AnswerQueryHere
```

This gives you the rows your query returns (if any) which are different from those in the correct answer. Also, run this:

```
AnswerQueryHere
EXCEPT
YourQueryHere
```

This gives you the rows in the answer which are missing from your query (again, if there are any).

If there are no results in each case, then your query result is identical to the answer query result, and therefore your query is correct (or at least, it is good enough that it doesn't produce any errors using the data provided in Northwind), so, well done!

Sys / Metadata - How to get information about your database

The Northwind database is a small database. It has a small number of tables, and those have small numbers of rows and columns. It is simple. The table and field names are self-explanatory and predictable. Databases are rarely this simple in the real world.

You may find yourself querying a system containing hundreds of tables. If you want to find a table, and only have a rough idea of what its name is, or should be, you may have to scan a list of hundreds of names. In this situation, knowing just a little about how to query the system, or metadata, of a database, can be a huge time-saver. Try out the following query:

```
SELECT Name
FROM Sys.Tables
WHERE Name LIKE '%Demo%'
```

This returns a list of tables containing the string "Demo" in the name. If you right-click on the names, you can copy them for use in new queries, e.g. you could select the top ten rows of a view named in the results of the query above.

Learning from existing views

Databases may also have many, maybe hundreds, of stored queries, known as “views”, already made. If many people run queries from your database, one of them may already have made queries you could use that you don’t know about yet.

The views are listed below tables in the object explorer (in the left hand panel). If you want to see the queries they contain, right-click on them and choose the options “Script View as” then “CREATE to”. This will show you the SQL the view uses. There will be some extra lines of SQL at the top, to do with saving the query as a view, but further down you will see the familiar SQL for getting data. If you want to see some results for a view, you can use the “Select Top 1000 Rows” menu option on views too.

Northwind doesn’t have hundreds of views, but we can still use it to show how to search large numbers of views efficiently, as in the exercise below.

Exercise – 18.1

(answer)

Can you change the query above to display a list of views containing the word “Product”? Hint: take the query above and change the words “Tables” and “Demo”.

Subqueries - How to make one query use a result from another

Subqueries in the FROM clause

This first use of a subquery is one of the simplest to grasp. Suppose you have just run a query and would now like to perform a further query on the results table it gave you. Maybe the first query is very long and complex, full of joins and calculations. Maybe it was written by someone else and you don't want to edit it. You just want to pull out the data you need from the results table. One way to do this would be to create a view (effectively a saved query) in the database. Like the views in the last chapter, this would have a name, and you could refer to that name the same way you refer to a table name: in the "FROM" clause of any other query. Another way, which saves you from having to create a view, is to query your query result directly.

We don't actually need to make a long complex query like the one imagined above. We can see the principle easily in a much smaller example. Run the code below as you first query.

```
SELECT ProductName, UnitsInStock * UnitPrice AS StockValue
FROM Products
```

Now suppose you want to list only those products with a stock value in the thousands. The following query lets you do so very simply, by adding a couple of lines to each end of the original.

```
SELECT *
FROM (
    SELECT ProductName, UnitsInStock * UnitPrice AS StockValue
    FROM Products
) sq
WHERE sq.StockValue >= 1000
```

Note that the brackets around the original, or "inner", query mean that it will be completed first. Its results will be used in the outer query, just like a table would be. In this example, we have the outer query give the inner query's result the alias "sq", for "subquery", so that we can refer to it. The outer query takes only the rows from the results table which meet the condition in the "WHERE" clause, and it does so using the "StockValue" field, which the subquery generated.

This saves some thinking compared to modifying the original query. To filter it the same way, we would have needed to add the following clause to the original query:

```
WHERE UnitsInStock * UnitPrice >= 1000
```

With the subquery approach, on the other hand, you can run the first query without

reading its code and design your outer query simply by looking at the results table and deciding which bits you need.

Exercise – 19.1

(answer)

Without changing its subquery, can you make the query above return the combined stock value of all the products you stock?

Exercise – 19.2

(answer)

Another use for subqueries in the FROM clause is to perform grouping or sorting on the results of a union. Use the SQL below as a subquery. Have your main query return a list of countries and cities, making sure any rows with the same country or city stay together.

```
SELECT CompanyName, 'Supplier' AS CompanyType, City, Country
FROM Suppliers
UNION ALL
SELECT CompanyName, 'Customer', City, Country
FROM Customers
```

The IN operator

The “IN” operator is commonly used in subqueries, particularly subqueries inside “WHERE” clauses, but you don’t need a subquery to use “IN”. You can see its effect clearly by running the example below:

```
SELECT SupplierID, CompanyName, Country
FROM Suppliers
WHERE Country IN (‘UK’, ‘USA’)
```

Using “IN” let’s you check if one value (e.g. the value in a record’s “Country” field) matches any of a list of values.

To see a list of suppliers from other countries, run this query:

```
SELECT SupplierID, CompanyName, Country
FROM Suppliers
WHERE Country NOT IN (‘UK’, ‘USA’)
```

The “NOT” makes the query check for countries which are not on the list.

Subqueries in the WHERE clause – using the IN operator

Suppose you need a list of all the products supplied from a particular country. The Products table doesn't say which country supplies a product. What it does say is the Supplier ID, which lets you find the record in the supplier table and from that, the supplier's, and therefore the product's country.

Still, you suspect that you will have to produce similar lists in future, for other countries. You want a query that you can re-use to get those results, just by changing the country name in your "WHERE" clause. The subquery below is one way to do this.

```
SELECT ProductName
FROM Products
WHERE SupplierID IN
(
    SELECT SupplierID
    FROM Suppliers
    WHERE Country = 'UK'
)
```

This should return a list of British sounding foods, and some more exotic sounding liquids, presumably imports to the UK which the supplier then ships on. If you change "UK" to another country name, you can get a similar list for a different country

The subquery returns a list of UK supplier IDs, and the main query, which returns the results you actually see, gives you all the products with a supplier ID on that list.

Note the use of "IN" above. This tells the "WHERE" clause to check if **any** of the values in the subquery match the product's supplier ID. If we had used the "=" operator, then we would be limited to subqueries which always return a single value, i.e. a single Supplier ID cannot be equal to a list of Supplier IDs, and this situation would cause the query to return an error anyway. Using "IN" lets us match a list against a single value.

Exercise – 19.3

(answer)

Return a list of products supplied from the UK or USA.

Exercise – 19.4

(answer)

Make a query which returns a list of products in the seafood category.

Exercise – 19.5

(answer)

Make a query which returns a list of products from only the categories which you are confident will be suitable for vegetarians. Assume these particular vegetarians don’t eat fish.

Subqueries in the WHERE clause – the ALL operator

Like “IN”, the “ALL” operator allows you to compare a list of values to a single value. However, it allows you to use operators like “>” to do so. The example below shows how.

```
SELECT *
FROM Products
WHERE UnitPrice >= ALL
(
    SELECT UnitPrice
    FROM Products
)
```

The subquery returns a list of unit prices. The main query returns a list of products with a unit price equal to or greater than each value on the list i.e. equal to the highest value on the list. This gives you the record for the most expensive product.

At this point, you might think that you could just use an aggregate function without a subquery, and get the maximum unit price. You could, but you couldn’t get the rest of the product record that way (try it and see). You could use an aggregate function inside the “WHERE” clause, as below:

```
WHERE UnitPrice =
(
    SELECT Max(UnitPrice)
    FROM Products
)
```

However, this is no simpler: you still have to use a subquery.

Exercise – 19.6

(answer)

In the chapter on grouping and aggregate functions, we mentioned this exact problem: that you can get an aggregate value, such as a minimum or maximum, but not the record(s) associated with it (in that case, the record for the cheapest product). The queries above show how to solve this. Can you alter the “ALL” query to return the record for the cheapest product?

Subqueries in the WHERE clause - the “ANY” operator

This is written into the query the same way as “ALL”, except that it returns true if even one of the values in a list meets the specified condition. Run the query below to see.

```
SELECT *
FROM Products
WHERE UnitPrice < ANY
(
    SELECT UnitPrice
    FROM Products
)
```

This query returns all the rows of the table, apart from the one for the most expensive product. The most expensive product doesn’t have a unit price cheaper than any of the other products, so it never meets the condition.

Where to go from here

There is a lot more to SQL. You can use it to change the data in a database. You can use it to make your own tables and functions or to modify other people's. It is not just a way of getting data, but the actual language used for controlling how databases work.

Most people will use SQL only to get data for reports or analysis. If that describes you, and you often find yourself exporting, or just copy-and-pasting, the results of your SQL queries into Excel for further work, you may wish to move on to a similar course I wrote called Excel: Learn Formulas Fast. It's a concise, exercise-packed tour of what Excel can do with data, which is much more than just calculations. It's available in e-book format.

Other people will go on to do database administration and development. This may involve importing new records, correcting faulty records in bulk, changing the database structure itself, and, of course, designing new queries for all the reports that users need. If that's something you'd be interested in, that can also all be practised using the SQL Server and the Northwind database.

If you are interested in database design, rather than just working with data inside databases, this is a separate topic to SQL. You can use SSMS to create a SQL Server database, but it's worth reading up on database design topics like normalisation first. If you want to give a SQL Server database a user interface, e.g. data-entry forms, you will need to use separate software to design one. For example, Microsoft Visual Studio allows you to design websites, so you could use that to make one that accesses your database. If you want to go down the non-Microsoft route for your website, you could use PHP and MySQL instead, which are also free to download online.

If you're not into web design, and just want to get started experimenting with databases, you could use Microsoft Access. It's limited, compared to the options above, but a lot of computers have it already installed, and it lets you design forms and reports without having to use other software or learning about web design or programming, so it may give you a quick way to try some things out.

Author's note

I hope you found this book useful. If so, you may wish to try some of my other books, listed over the page.

I also hope that this book saved you time, that it provided clear, concise information, and a simple way to practise using it. That's the standard I aim for: to write the kind of book that I would want to use.

At this point, I would normally ask nicely for a review on the book's Amazon page. However, from leaving reviews myself, I know it can be hard to think what to write, so I'm trying out ways to make it easier. Instead, I'm asking, if you can spare a couple of minutes, please, go to this book's Amazon page, click "leave a review", and type in any one of the reviews below that you agree with:

> This book did everything it said it would.
>
> This book was clear and to the point.
>
> I found the practice exercises helpful.
>
> I did learn SQL fast!
>
> I like the way this book is written.
>
> This book was well formatted for Kindle.

I do appreciate honest reviews. Leaving a good review helps my books do well and encourages me to write more of them. One of those books may then be there to save you time when you next need to learn a new skill in a hurry. That's what I can offer, honestly, in return for the support of a good review.

Finally, if you feel that anything about this book falls short of what you hoped for, you can let me know at:

authordarmstong@gmail.com

To skip over the answer section, and see other books I have written, follow the link below:

Other books you may find useful

Answers

Answer – 1.1

(back)

```
SELECT *
FROM Categories
```

Answer – 1.2

(back)

```
SELECT ProductName, UnitPrice, UnitsInStock
FROM Products
```

Answer – 2.1

(back)

```
SELECT ProductName AS Product, UnitPrice, UnitsInStock, UnitPrice * UnitsInStock AS StockValue
FROM Products
```

Answer – 2.2

(back)

```
SELECT ProductName, UnitPrice * UnitsOnOrder
FROM Products
```

Answer – 2.3

(back)

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

Answer – 2.4

(back)

```
SELECT UnitPrice / 5
FROM Products
```

Answer – 2.5

(back)

```
SELECT 7 * 8
```

Answer – 2.6

(back)

```
SELECT ProductName, UnitsInStock %12
FROM Products
```

Answer – 2.7

(back)

```
SELECT ProductName, (UnitsInStock + UnitsOnOrder) * UnitPrice
FROM Products
```

Answer – 2.8

(back)

```
SELECT ContactName + ', ' + Address + ', ' + City
FROM Customers
```

Answer – 2.9

(back)

```
SELECT ContactName + ', ' + Address + ', ' + City AS 'Name and Address'
FROM Customers
```

Answer – 2.10

(back)

```
SELECT OrderID, ShipCity + ', ' + IsNull(ShipRegion,'') + ', ' + ShipCountry AS 'Order Address'
FROM Orders
```

Answer – 2.11

(back)

```
SELECT OrderID, ShipCity + ', ' + IsNull(ShipRegion + ', ','') + ShipCountry AS 'Order Address'
FROM Orders
```

This answer does become a little hard to read within the brackets of the IsNull function. However, all we have done is to move the "comma plus space" into the first argument of the function onto the end of the ShipRegion field. If this field is null, then adding the string constant won't affect it: it will still be null. So this time, the second "comma plus space" will only added to the address if the ShipRegion is not null.

Answer – 2.12

(back)

SELECT OrderID, ShipCity + ‘, ‘ + ShipRegion + ‘, ‘ + ShipCountry AS ‘Order Address’

FROM Orders

This query now returns null values for order address whenever the region is null.

Answer – 3.1

(back)

SELECT *

FROM Customers

WHERE City = ‘Berlin’

Answer – 3.2

(back)

SELECT ContactName, Phone

FROM Customers

WHERE City = ‘Berlin’

Answer – 3.3

(back)

SELECT City

FROM Suppliers

WHERE Country = ‘Japan’

Answer – 3.4

(back)

SELECT City

FROM Suppliers

WHERE Country = ‘India’

This query returns no rows. This is the correct result. You don’t have any suppliers based in India, so the WHERE clause removes all the rows.

The opposite case here is that, if you had several suppliers based in the same city in India, that city’s name would appear several times in your results. The table records are one per supplier, not one per city, so this query can return duplicate rows. We’ll cover how to prevent such duplication in the chapter on the “DISTINCT” keyword, later on.

Answer – 3.5

(back)

```
SELECT ProductName, UnitsInStock
FROM Products
WHERE UnitsInStock >= 10
```

Answer – 3.6

(back)

```
SELECT ProductName, UnitPrice
FROM Products
WHERE UnitPrice > 21.35
```

Answer – 3.7

(back)

```
SELECT *
FROM Products
WHERE UnitsInStock * UnitsOnOrder < 100
```

Answer – 3.8

(back)

```
SELECT ProductName
FROM Products
WHERE ProductName LIKE 'P%'
```

Answer – 3.9

(back)

```
SELECT *
FROM Employees
WHERE HireDate >= '1993-01-01'
```

Answer – 4.1

(back)

```
SELECT ProductName, UnitsInStock
FROM Products
WHERE UnitsInStock >= 10
AND UnitsInStock < 20
```

Answer – 4.2

(back)

```
SELECT *
FROM Orders
WHERE OrderDate >= '1997-01-01'
AND ShipCountry = 'Brazil'
```

Answer – 4.3

(back)

```
SELECT *
FROM Orders
WHERE OrderDate >= '1997-01-01'
AND OrderDate < '1998-01-01'
AND ShipCountry = 'Brazil'
```

Answer – 4.4

(back)

```
SELECT *
FROM Employees
WHERE Country = 'UK'
OR City = 'Seattle'
```

Answer – 4.5

(back)

```
SELECT *
FROM Customers
WHERE Country <> 'USA'
```

Answer – 4.6

(back)

```
SELECT *
FROM Customers
WHERE ContactTitle LIKE 'Sales%'
OR ContactTitle LIKE 'Marketing%'
```

Answer – 4.7

(back)

```
SELECT ContactName, Phone, ContactTitle, Country
FROM Customers
WHERE Country <> 'USA'
AND (ContactTitle LIKE 'Sales%'
OR ContactTitle LIKE 'Marketing%')
```

Answer – 4.8

(back)

```
SELECT ContactName, Phone, ContactTitle, Country
FROM Customers
WHERE Country <> 'USA'
AND NOT (ContactTitle LIKE 'Sales%'
OR ContactTitle LIKE 'Marketing%')
```

Answer – 4.9

(back)

```
SELECT ContactName, Phone, ContactTitle, Country
FROM Customers
WHERE (Country <> 'USA'
AND Country <> 'Mexico')
AND NOT (ContactTitle LIKE 'Sales%'
OR ContactTitle LIKE 'Marketing%')
```

There are several queries you could use to get this data. The one below avoids using brackets.

```
SELECT ContactName, Phone, ContactTitle, Country
FROM Customers
WHERE Country <> 'USA'
AND Country <> 'Mexico'
AND ContactTitle NOT LIKE 'Sales%'
AND ContactTitle NOT LIKE 'Marketing%'
```

Answer – 5.1

(back)

```
SELECT DISTINCT Country
FROM Suppliers
```

Answer – 6.1

(back)

```
SELECT ProductName, CategoryID, UnitPrice
FROM Products
ORDER BY CategoryID, UnitPrice
```

As ascending order is the default, there is no need to use any keywords with UnitPrice or (CategoryID). There is an “ASC” keyword in SQL, the opposite of “DESC”, but we don’t need it.

Answer – 7.1

(back)

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

Answer – 8.1

(back)

```
SELECT Sum(UnitsInStock)
FROM Products
WHERE Discontinued = 1
```

Answer – 8.2

(back)

```
SELECT Sum(UnitsInStock)
FROM Products
WHERE CategoryID = 1
```

Answer – 8.3

(back)

```
SELECT SupplierID, Sum(UnitsOnOrder)
FROM Products
```

GROUP BY SupplierID

Answer – 8.4

(back)

SELECT SupplierID, Sum(UnitsOnOrder * UnitPrice)

FROM Products

GROUP BY SupplierID

Answer – 8.5

(back)

SELECT CategoryID, Max(UnitPrice)

FROM Products

GROUP BY CategoryID

Hopefully, if you didn't notice it already, this exercise made you notice the need to change the grouping field in two places: the SELECT and GROUP BY clauses. Typically, when grouping is used, the non-aggregate fields in these clauses should match.

Answer – 8.6

(back)

SELECT SupplierID, Min(UnitPrice)

FROM Products

GROUP BY SupplierID

Answer – 9.1

(back)

SELECT CategoryID, Sum(UnitsInStock)

FROM Products

GROUP BY CategoryID

HAVING Sum(UnitsInStock) > 100

Answer – 9.2

(back)

SELECT SupplierID, Count(*)

FROM Products

GROUP BY SupplierID

HAVING Count(*) = 1

Answer – 9.3

(back)

```
SELECT CategoryID, Sum(UnitsOnOrder)
FROM Products
GROUP BY CategoryID
HAVING Sum(UnitsOnOrder) = 0
```

Answer – 9.4

(back)

```
SELECT OrderID, Sum(Quantity)
FROM [Order Details]
GROUP BY OrderID
HAVING Sum(Quantity) < 50
```

Answer – 9.5

(back)

```
SELECT OrderID, Max(Discount)
FROM [Order Details]
GROUP BY OrderID
HAVING Max(Discount) > 0.2
```

Answer – 9.6

(back)

```
SELECT OrderID, Sum(Quantity * UnitPrice) As CashValue, Sum(Quantity) As Units
FROM [Order Details]
GROUP BY OrderID
```

Answer – 11.1

(back)

```
SELECT c.CategoryName, p.*
FROM Products p
INNER JOIN Categories c
ON p.CategoryID = c.CategoryID
```

Answer – 11.2

(back)

```
SELECT c.CompanyName
FROM Orders o
INNER JOIN Customers c
ON o.CustomerID = c.CustomerID
```

Answer – 11.3

(back)

```
SELECT s.CompanyName, Sum(p.UnitsOnOrder)
FROM Products p
INNER JOIN Suppliers s
ON p.SupplierID = s.SupplierID
GROUP BY s. CompanyName
```

Answer – 11.4

(back)

```
SELECT c.CategoryName, Max(p.UnitPrice)
FROM Products p
INNER JOIN Categories c
ON p.CategoryID = c.CategoryID
GROUP BY CategoryName
```

Answer – 12.1

(back)

```
SELECT DISTINCT CompanyName
FROM Customers c
INNER JOIN Orders o
ON c.CustomerId = o.CustomerId
```

Answer – 12.2

(back)

```
SELECT *
FROM Territories t
LEFT JOIN EmployeeTerritories e
```

ON t.TerritoryID = e.TerritoryID

Answer – 12.3

(back)

```
SELECT *
FROM Territories t
LEFT JOIN EmployeeTerritories e
ON t.TerritoryID = e.TerritoryID
WHERE e.EmployeeID Is Null
```

Answer – 12.4

(back)

```
SELECT *
FROM Territories t
INNER JOIN EmployeeTerritories e
ON e.TerritoryID = t.TerritoryID
```

Yes, you could keep using the query from the last exercise, with its left outer join, and just change the condition in the “WHERE” clause, but this would be inefficient. If you queried a large table this way, it would slow your query. This doesn’t matter here, but speed is an issue in bigger databases.

Answer – 12.5

(back)

```
SELECT FirstName, LastName, TerritoryDescription
FROM Employees e
INNER JOIN EmployeeTerritories et
ON e.EmployeeID = et.EmployeeID
INNER JOIN Territories t
ON et.TerritoryID = t.TerritoryID
```

This should give you the same number of results as before (49). You may have done the join in a different order, but this doesn’t matter for inner joins.

Answer – 13.1

(back)

```
SELECT staff.FirstName + ‘ ‘ + staff.LastName AS Staff,
```

```
boss.FirstName + ' ' + boss.LastName As Boss
FROM Employees staff
LEFT OUTER JOIN Employees boss
ON staff.ReportsTo = boss.EmployeeID
```

Answer – 14.1

(back)

```
SELECT (
SELECT Count(*)
FROM Shippers
) * (
SELECT Count(*)
FROM Customers
)
```

Answer – 14.2

(back)

The quick way:

```
SELECT *
FROM Shippers, Customers
```

The “join” way:

```
SELECT *
FROM Shippers
INNER JOIN Customers
ON 1 = 1
```

Answer – 14.3

(back)

The quick way:

```
SELECT Count(*)
FROM Shippers, Customers
```

The “join” way:

```
SELECT Count(*)
FROM Shippers
```

```
INNER JOIN Customers
ON 1 = 1
```

Answer – 15.1

(back)

```
    SELECT CompanyName, 'Supplier' AS CompanyType, City, Country
    FROM Suppliers
    UNION ALL
    SELECT CompanyName, 'Customer', City, Country
    FROM Customers
```

Answer – 18.1

(back)

```
SELECT Name
FROM Sys.Views
WHERE Name LIKE '%Product%'
```

Answer – 19.1

(back)

```
SELECT Sum(StockValue)
FROM (
SELECT ProductName, UnitsInStock * UnitPrice AS StockValue
FROM Products
) sq
```

Answer – 19.2

(back)

```
SELECT *
FROM
(
    SELECT CompanyName, 'Supplier' AS CompanyType, City, Country
    FROM Suppliers
    UNION ALL
    SELECT CompanyName, 'Customer', City, Country
```

```
    FROM Customers
) sq
ORDER BY sq.Country, sq.City
```

Answer – 19.3

(back)

```
SELECT *
FROM Products
WHERE SupplierID IN
(
    SELECT SupplierID
    FROM Suppliers
    WHERE Country IN ('UK', 'USA')
)
```

Answer – 19.4

(back)

```
SELECT *
FROM Products
WHERE CategoryID IN
(
    SELECT CategoryID
    FROM Categories
    WHERE CategoryName = 'Seafood'
)
```

In this case, you could also use "=" instead of "IN", as there is only one seafood category, and therefore only one ID returned by the subquery.

Answer – 19.5

(back)

```
SELECT *
FROM Products
WHERE CategoryID IN
(
```

```
    SELECT CategoryID
    FROM Categories
    WHERE CategoryName NOT IN ('Seafood', 'Meat/Poultry')
)
```

Answer – 19.6

(back)

```
    SELECT *
    FROM Products
    WHERE UnitPrice <= ALL
    (
        SELECT UnitPrice
        FROM Products
    )
```

Other books you may find useful

I have written similar *Learn…Fast* courses for:

- Command Line and Batch Script (for Windows)
- Excel VBA (function design)
- Excel (formulas)

I have also written brief guides to:

- Keyboard shortcuts
- IT Support

All of the above are available as e-books.