# **Getting Started**

## **Course Introduction**

A database is used to organise and store data in an electronic format.

There are multiple types of databases, such as flat-file or hierarchical,

but the most widely used database is the relational database, that stores data in

tables that are related or connected to one another.

In this course, you will learn what databases and Relational Database Management

Systems (RDBMS) are, and be introduced to different database models.

We'll also cover how to use the Data Manipulation Language (DML) statements to

query data and manipulate result sets, as well as how to update and delete data,

and see how databases maintain the integrity of your records.

## **Introducing Databases**

Let's get started with just a basic introduction about databases and what they are.

And it is, in essence, just a collection of related data that's organized into a

logical structure.

Now we'll come back to the related data and logical structure in a moment.

But the data is stored electronically and can be processed to deliver meaningful

information to end users.

In other words, it's not meant to just be stored but rather to also be retrieved.

So for example, if you're storing sales information, you can certainly retrieve

just a single sales record, but that doesn't give you a lot of information.

But if you were to retrieve, let's say, thousands or even possibly millions of

sales records, then you can really start to ascertain some meaningful information.

We can see how sales are moving, which products are selling, where, during which

time frames, to which customers.

And you can gather up all kinds of information that is useful to your organization.

Now the data in a database is organized by fields, which are the vertical columns;

records, which are the horizontal rows;

and the collection of all of it put together, which creates the table.

Now the fields or the columns, in this case, have headers, which tell us what

we're looking at.

So the column headers are:

Employee code, First Name, Last Name, and Benefit package.

So that as we read down that column, that's all we're looking at.

We're seeing nothing but employee codes in the first column.

Then as you read across, each record, it's the collection of entries for each

individual column that creates each row or each record.

And then when you put all the records together, you have the table.

Now the databases themselves have a structure as well, and this is that logical

structure that was referenced earlier.

They can be what's known as flat, which is usually just a single table or possibly

multiple tables that don't have anything to do with each other.

They're just completely separate structures that are holding data.

A flat database might be what we're looking at right here.

It is simply a list of names and their benefit package.

Perhaps a little more realistically or day-to-day, if you will, an employee

directory or even a phone book.

It simply lists the people, their phone numbers, and their addresses, and that's

it. That's all there is to it. There is no other structure.

A hierarchical structure is organized into these parent-child types of

relationships, wherein, as you move down the hierarchy, you typically become more

specific.

As you move back up the hierarchy, you typically become more generalized.

So we have a Department entity, which is subdivided into IT and Sales, in this

case, which is then further subdivided into Contract employees and Permanent

employees in the IT department.

So it's not unlike a file management system, where you create a parent folder and

then have subfolders in that folder.

And then finally, you have the relational, and this is probably the most common

and it will be the focus of this course.

The relational database has multiple tables that has connected or related

information.

So in this example, we see that there is an Orders table, wherein a sales manager,

for example, might be entering in the sales that they are making.

But the finance manager is interested in entering customer information, but the

two are related.

Because, as a finance manager creates a record for a new customer, that customer

might turn around and generate an order.

While the order has to belong to somebody, and of course, it is the customer.

So if you look in the Orders table, you see that there is a reference to the

customer.

But that's all you need to do is simply say, this order belongs to that particular

customer.

You can then use that value that identifies the customer to go and look up all of

their customer information, so there is a relationship.

Customers generate orders. That is the relationship between those two tables.

Now with relational databases, you simply see it is a collection of tables, or

it's usually many tables.

And it's made up of data that is simply related to each other.

So there is always some kind of a field that's common between two or more tables.

However, what you'll typically see in almost all cases in relational databases is

that there is one column known as the primary key, and this is to provide a unique

identifier for each row.

And in this example here, in the Customers table, it's the ID field.

This uniquely identifies each and every customer.

Because if you were to just put in something like customer name, that would

probably repeat; all kinds of times people have the same names.

So you need something that's unique that can never be confused with anything else.

And then you also see it in the Orders table, the Order Number is the unique

identifying field for any one particular order.

But the customer that's referenced in there does actually borrow the ID from the

Customers table.

So if you were to look at what an actual record might look like, let's just say,

it's the very first order for the very first customer.

Order number 001 might be generated by customer 001.

So you would see in the Customers table, there would only be a single entry for

customer 001.

The next customer would be customer 002. The next customer would be customer 003.

So you would only uniquely see each individual customer one time.

But of course, any given customer can make possibly thousands or even millions of

orders or they might not make any, it doesn't really matter.

But you would see customer 001 possibly show up many times in the Orders table.

So one customer generates many orders, but any one order must only belong to the

unique single customer referenced in the related table.

Now the data can be reorganized and grouped as well without modifying any of the

original tables.

In other words, I could query out information about the customers.

So if I was the sales manager, for example, I might have a sales territory.

I might want to see only customers that are in my sales territory.

So I can absolutely do that.

I can group them together without having to make any changes.

All that's necessary in the Customer table would be a field called sales territory.

Something along those lines that I can extract the records that are meaningful to

me, without making any kind of changes to the way the tables are structured or the

way the data is entered.

Now there are a few rules of the relational model.

All data is stored as values in the table.

So you simply enter a value into each field, and it must be accessible to users

using the table name, the primary key, and the field name without any knowledge of

the storage location.

So it doesn't matter to them where the records are stored, in which database, on

which server.

They just need to be able to know the customer information.

For example, I want to see customer ID number 1 from the Customers table.

It doesn't matter to me where it's stored, I just want to be able retrieve that

information from my application.

It must also support null values.

Not every field has an appropriate value.

For example, if you were storing the customer's fax number, well not everybody has

a fax number, so you need to be able to leave some fields blank.

It must support at least one language.

In this case, this is known as SQL or Structured Query Language – that's the

primary language of databases.

The result sets should be modifiable.

So if you do run a query that says, show me all the customers in my sales

territory, you have to be able to make changes to any one particular record.

For example, a person might move to a different territory.

You should be able to make changes to the result set that is returned by a query.

And it must support what's known as single commands to change that data.

And in the Structured Query Language, it is simply known as an update statement.

You simply make your change by modifying the original record.

But you have to be able to also prevent changes, if that change undermines an

integrity rule.

And that means that you can't make a change to a field that would effectively

mess up the rest of the database.

In other words, if you were to change a customer's ID, that would mess up all of

the orders that were entered using that person's ID.

So there are certain fields that you can't change.

So those are the basics of databases and particularly the relational model.

## **DBMS and RDBMS**

A database management system (DBMS), is in essence the software that is used to

access the databases, to create them, and ultimately to manage the databases as

well.

Now as far as the DBMS itself goes, it does allow your users to add, modify,

delete, and retrieve data.

But they typically don't work directly with the DBMS.

The DBMS works with the operating system to manage the data.

So what that means is that users, those who are entering records and retrieving

records and just doing their day-to-day operations, will likely use different

applications to do so.

Perhaps a web-based interface or a custom in-house developed application – that's

how they interact with the database.

But the DBMS itself is the software that manages who is doing what and what they

can do.

For example, security.

Certain users maybe should not have access to certain information.

In a practical example, only the HR department should have access to the employee

information table.

Well it's the job of the DBMS to implement that security.

So that a standard sales manager, for example, does not have access to that

employee information.

So the other thing it does is to control all of the concurrent users.

Because in very large environments, you can have hundreds, possibly thousands of

users, all accessing the database in different ways at the same time.

So again, it is the job of the DBMS to manage all of that, to control all of the

traffic and ensure that everybody is able to do whatever it is that they need to

do.

As far as actually using the interface of the database management system, that

generally does not fall to end users.

It is usually the administrators, the technical staff that manage the databases,

who will use the DBMS itself directly.

Now the management systems for relational databases include the following examples,

and these are only a few.

But Microsoft, their server-based component, their higher-end one, is simply

called SQL Server, you'll hear most people say.

MySQL is an open-source example of the same style of database, if you will, more

of a higher-level server-based; usually larger databases that are supporting very

high numbers of users and very high numbers of records.

So MySQL as an open-source product is actually free.

But as of the time of this recording, MySQL is generally overseen by, if you will,

by Oracle.

And Oracle is another product that falls under the server-based database component.

And finally, Microsoft Access is more of a desktop-level.

That would be something that you might see end users just creating their own

personal databases with but certainly not something that you would support, you

know, hundreds of users or thousands of users accessing millions upon millions of

records.

Microsoft Access just simply wouldn't be up to the task for that size database –

so a much more personal-level application.

Now as far as the servers go, it can be either a physical computer or a virtual

server, that's fine, that are used to store the databases.

There is high-level support for data access, so again you can have tremendous

numbers of users all accessing the data at the same time.

And they simply contain the DBMS.

So you go and you sit yourself down in front of a server and you install the

software for the database management system.

So again, if you're talking about Microsoft SQL, for example, you would install

that on one of your Windows Server 2012 systems or whatever version you're

currently running.

You can then have one default instance and multiple named instances.

Now what this is, in essence, is the ability to install a product multiple times.

It allows you to create different configurations, and this actually was not

available in some of the older versions of the product.

It's been around for a while now, but if you go back quite a ways, you could

install a product only one time and configure it in a particular way.

You didn't have any limits on the number of databases you could create; that was

still unlimited really.

But it was the configuration, particularly with things like the security of the

database and the language support and what's known as the collation, how data is

treated in terms of alphabetical sorting and character recognition and things like

that.

So now you can have multiple instances, and each one can be configured differently.

So that I can have security configuration A on instance A, and security

configuration B on instance B.

And then I can have language A on instance C, and language B on instance D.

So you can have these completely separate configurations.

Those are your instances.

And the default is simply the one that you get when you install the product for

the first time.

After that, if you want additional instances, you can call them whatever you want.

And you can, in fact, also rename the default one if you want to.

Microsoft SQL Server is an instance, if you will, of the product running on

Windows as a Service.

So you see that there will actually be an executable file called sqlservice.exe.

So if you just go to your Windows Services console on a system that's running SQL

Server, you will see that file running.

That indicates that SQL has been installed on that system and it is ready to

service the database application.

Multiple servers can be used to ensure availability and performance.

This is a feature known as fault tolerance or clustering, wherein you can have

several servers maintaining copies of a single database.

So that if one of them fails, you have others ready to assume the workload of that

database.

And as far as your end users are concerned, they don't see any disruption in

service.

So this makes it very available and ensures high performance levels.

As far as database servers go in SQL Server, the databases themselves are stored

in a couple of different file types.

So it is still just a file like any other application to store the information,

but there are three types of files.

The first one is known as a primary database file and has the extension mdf,

that's Microsoft data file. That's basically all it is.

It's just a file that contains information, very much like an Excel file contains

spreadsheets.

So that in essence is your data.

Now you can have secondary files if you want to, extension ndf, that's up to you.

They aren't required but they can help you organize the database a little more

effectively, particularly if it's very large.

That's where you see a lot of secondary files come into play.

Because if it is an exceptionally large database, when you need to do

administrative tasks such as backing up the database, if it's all in a single

primary file you have to back up the whole database.

But there may not be very much of it that has changed since the last backup.

So it might take a very long time, even though there's very few changes.

Well secondary files, you can say, will only back up that particular ndf file

because that's the one that all the changes have occurred to.

These other ndfs have hardly changed it all.

So you can do a very specialized or particular backup.

And finally, an ldf is what's known as a transaction log, often referred to as

TLOG for short.

And this is what stores all of the changes that have occurred.

So as new records come in, as things get edited and deleted, the transaction

log – the transaction literally means something has changed – it logs all of

those entries.

So it's very useful for recovery purposes.

In the event that the database fails, you can use the transaction log to restore

any kind of changes that have not yet been backed up.

So those are the three types of files that you'll find in just about every

database instance, but again the secondary files aren't required.

So you may not see them, but quite commonly they are used.

## **Introducing SQL Server Management Studio (SSMS)**

We're going to take a look now at the SQL Server Management Studio, which is the

administrative interface for Microsoft's SQL Server.

So when you first launch the product, you are asked to connect to the particular

server on which the product is installed, or you can in fact also connect to any

other remote server that is accessible.

You can use a single management interface to connect to any and all of your

database servers.

It really does not matter where they are as long as you have physical connectivity.

So in this case, this is the name of the server, and here you see it will ask you

for the authentication method.

And if you go with the fairly standard default installation in a Microsoft

environment, it will go with what's known as Windows Authentication, which simply

means you are using the same set of credentials that you used to sign in to

Windows.

So you can see, with that, I don't even have a choice, in terms of changing the

User name and Password. It simply picks up on my existing identification.

Now the other choice is that you can be authenticated by the SQL Server itself.

Now we'll be talking about the security a little bit later on.

But those are the only two options when it comes to Authentication.

You either use your Windows identity, or you can use one that is stored directly

in the SQL Server itself.

And this would be for people who do not have a Windows identity.

So I do, and there are my credentials there.

So all I have to do is choose Connect.

And it makes the connection, and then I see all of my objects over here on the

left-hand side, which is under my Object Explorer.

So the main thing that you're interested in, in most cases, of course, are the

databases themselves.

So here we see all of the databases that are installed on this server.

And there really isn't any limit to the number of databases.

It really just comes down to the storage and the resources that are available on

that server.

So we can expand this, and you will see that there are a few databases that are

here already.

And these were some examples from previous courses and things like that, while

some of them are sample databases that you can obtain from Microsoft.

And in fact, that's what these two right here are, this AdventureWorks2012 and

AdventureWorks Data Warehouse.

These are freely available databases that you can just download from Microsoft

and use for your own purposes purely to help learn the product.

So a lot of SQL Servers might have these databases on them.

Now the other thing that you will always find is this one right here, the System

Databases.

And these are databases that are always present on all SQL Servers because they

help control and maintain the configuration of the other databases that you have.

So for example, if I expand this here, you'll find that there is one called master.

And master has various entries for every other database that you define.

So in essence, there are two categories.

The System Databases are maintained by SQL Server itself.

They help to control the system.

All the other databases are what's known as user-defined; you create them.

But every time you create a new database, an entry gets created in the master

database that says, you just created this particular database, and it has

properties like the time that you created it, the size, the storage location of

the files.

All of those sorts of things are maintained in these system databases.

Now we'll go through those a little bit later on as well, but those are your two

primary categories of databases – System and user-defined.

Now there is a lot more here than just the databases.

If you continue to scroll down, you see that there is Security.

And this is where you control the users that are accessing the environment.

So Logins and user accounts are managed within the Security.

You also see that there are various roles, that you can assign people to, so that

they have certain administrative abilities and certainly various other features

with respect to security.

That again, we'll also cover later on, but for the time being we just want to get

familiar with the interface.

Then you see certain other features that are fairly common such as Replication,

which is to maintain additional copies of the databases.

And AlwaysOn High Availability, and this is where you maintain redundant copies of

databases, so that if one fails, there is always another copy of the same database

available to assume the services.

So that your clients don't notice any disruption of services in the event of a

failure.

Under Management, you'll find fairly standard day-to-day tasks that you might have

to do such as automated jobs and things along those lines.

You see here, Data Collection and Resource Governor, as well, for performance

management and just keeping an eye on how the server is performing.

Maintenance Plans to do things like backups and cleaning up of the databases.

Mail that you can configure, so that you can receive alerts.

In the event that you do configure an automated backup, for example, you can

receive an e-mail that informs you that the backup succeeded or failed.

So there are a number of components that we'll certainly be going through.

But those are some of your basic objects that are available within the SQL Server

environment.

And again, this is what you see, what's called the Object Explorer so that you can

see all of these.

Now once you do start to get in and start working with a database, you can just

select it and expand that.

And you'll see all of the objects that are particular to just that specific

database.

So here we see are the Tables.

So this is where we would see the actual data storage for those tables within that

specific database.

So we'll certainly be doing some work in there.

And then we'll also be looking at some Views, which are affectively saved queries.

This is how you start retrieving data out of the database to give you some

meaningful information.

So we can expand this as well, and you see that there's already several Views that

have been defined in there.

And they help to retrieve the data, usually for purposes such as reporting.

So that I can simply create things like sales totals and present them to my users.

So that's just some of the basics, and we'll certainly be doing a lot more work in

the SQL Server Management Studio as we move through the course.

### **Lesson Test**

Q1: You need to do some database maintenance and create a backup:

Object Explorer -> Management

Q2: What do you call the unique identifier that links data record sets between

different tables?

Primary Key

Q3: In a SQL Server, databases are stored in three files.

What are the extensions of these files?

.mdf, .ndf, .ldf

-----------------------------------------------------------------------------------------

# **Database Design**

## **Data Types**

When you store information in a table in a database, you need to inform the

database what kind of information you are supplying.

And this is what's known as the data type.

And every field that you define in every table has a data type.

So it's an attribute, a characteristic that specifies how this information is

stored and how it's being used.

Now in essence, you use the data type that provides the most precision or the

greatest value range depending on what it is you're storing, and we'll see some

examples of that in a minute.

But it is possible to convert from one data type to another in a couple of

different instances.

And it can be done either automatically by, for example, a SQL Server, and this is

known as an implicit conversion, which simply will convert from one type to

another.

Or you can do it manually yourself using various statements such as CAST or

CONVERT in a command in SQL Server, which means that you simply want to explicitly

control yourself how data types are converted.

And common examples of this include when you're doing calculations, you might be

taking field 1 and doing a mathematical calculation with field 2.

Now normally with mathematical calculations, you need to deal with numbers.

But you don't always have numbers in the two fields, so SQL Server is able to

recognize this and, in some cases, do an implicit conversion.

For example, a text-based field does allow you to store numeric information.

So you might have any kind of numeric value in a field that actually has a text

type of data.

So with that, if you were to, let's say, just multiply it against another field

that is actually a numeric data type, SQL Server will pick up on that, and it will

complete the conversion automatically and give you an answer.

But if you know that you're dealing with a non-numeric format, you want to change

it, this is where you can explicitly do so in a SQL statement that you create, and

those are the explicit conversions.

Now some of the categories – touched on them already – but exact numbers and/or

approximate numbers; these are numeric categories.

But in some cases, you don't know the exact value.

You know, you might need to just enter an approximate value.

And a very good example of an approximate number is the value for pi.

You know, you cannot assign an exact value to the number pi.

So sometimes you just have to put in estimate.

Exact numbers are exactly that.

If you have a quantity and somebody orders five of something, that is an exact

number.

Date and times can be stored.

Character strings are basically just textual information.

Unicode character strings are for multilingual support, usually.

Binary strings, CLR is Common Language Runtime for developers usually and spatial

data types for geographic type of information such as GPS systems.

In SQL Server, the specific data types include, and this is only a small subset,

but money for when you're entering currency values, datetime for any kind of date

and/or time information, int is short for integer or a numeric value, char and

varchar are both for string types; character is what that stands for.

The char is used when you know exactly how many characters are going to be entered

such as an abbreviation.

Common example here, if you look at the postal information for a state or a

province, there is usually a two-character abbreviation.

So you can always specify only two characters.

But for a person's last name, for example, you have no idea how many characters

that are going to be, so that is a variable character field.

It allows for flexibility.

Boolean is a yes/no, on/off, true/false, always just one of two possibilities.

Float is a numeric type that usually has very high decimal places associated with

it; scientific notation.

And bit is a one or a zero.

So again those are just a couple of examples of some of the data types.

But every time you define a new field, you have to specify what kind of data is

being stored in that field and that's your data type.

## **Data Integrity and Constraints**

Data integrity is an approach whereby you ensure the information in your database

is correct and accurate.

There are four different categories:

1. Entity integrity typically refers to the uniqueness of a row, for example,

the primary key.

This ensures that you only ever have one unique customer number one.

You don't ever want to see two customers with the same ID.

2. Domain integrity refers to all of the collective values in a particular column.

Now this might seem similar to entity integrity if you're talking about a primary

key, for example.

If you looked at all of the IDs, you would want to make sure that there are no

duplicates.

But it doesn't just have to be for a primary key.

You might, for example, be storing a person's credit card number; that would also

be something that you would want to ensure is unique.

So if you looked all the way down to the column for every credit card entry, you'd

want to ensure that there were no duplicate values.

Now there's other examples of domain integrity – that's just one.

Some other ones might be something as simple as a not null.

You might want to enforce some kind of entry for any particular column;

that's another example of domain integrity.

So if I look down the credit card type of field, not only can there be no

duplicates, but there must be a value of some kind.

This is another example of domain integrity.

It might also just be a range of acceptable values such as date entries, for

example, date of birth must be either today or back.

You can't enter a future date for date of birth.

So those are several examples of domain integrity.

3. Referential integrity refers to a related record from one table connecting to a

related record in another table.

So this is typically where you see a primary key also used.

But in this example, you see in the graphic, a customer ID uniquely identifies the

customer.

The Order Number uniquely identifies the order.

Those are the two primary keys.

But in the Orders table, you see a reference to that customer who generated that

order.

That is what's known as the foreign key, and we'll talk about in a moment.

But the customer value that you enter in the Orders table is the ID of the

customer that comes from the customer table.

That's referential integrity, and it ensures that you can only create orders for

customers who exist in the Customers table.

So in this simple example, we see there is only three records for the Customers

table.

So let's just assume they are customer 1, 2, and 3.

Hence, in the Orders table, you could not put an order in for customer number 4.

They simply don't exist.

4. User-defined integrity is any other role that you defined as per your business

needs or the logic that you want to implement in your database, and it really can

be just about anything.

Now constraints are methods by which you enforce the integrity, and we've already

talked about a couple of them.

Primary key is by nature a constraint.

If you assign the primary key attribute to a field, it will automatically not

allow you to leave it empty, nor will it allow you to enter duplicate values.

So you are constrained in the values you can create.

The foreign key, again, references the primary key of another table to maintain

the connection, the relationship between the data.

Unique keys are used to maintain uniqueness not, just for a primary key, but again

that credit card example.

You want to make sure that those are kept unique so it will not allow duplicates.

Indexes helps to search, speed up searches and sorts, but in fact, it is indexes

that maintain the uniqueness of fields as well.

But we'll talk about indexes in a lot greater detail later on, but they do, in

fact, help the performance of your database as well.

And triggers can be used to execute code automatically in response to some kind of

a modification, so that if you change a particular value, it can trigger

additional changes elsewhere in the database.

Now specifically the primary key, it is again the unique identifier for a row, and

only one primary key can exist in a table.

And the columns that are using the primary key cannot be left empty.

Not null, is what that's referred to, but those are some of the inherent

characteristics of a primary key field.

A foreign key is used to link the tables.

And a foreign key again is always the primary key from some other table.

So columns using the primary key are also used to create the foreign key.

And again in this example, customers' ID number is their primary key, but in the

Orders table, the reference to customer uses the ID field from the Customers table.

Hence it's the primary key in the Customers table, but it's the foreign key in the

Orders table.

So it simply creates a unique constraint in that same or any other table.

You must reference the unique customer that owns this particular order.

But the columns that are using the foreign key can be left null, and it can also

repeat.

In other words, customer number one can generate many orders.

So if you look at the Orders table, and again in this example, there is only three

simple records here, but you could imagine that customer number one generated all

three orders.

So even though order number one is unique and order number two is unique and

order number three is unique, the customer ID for all three of those might be

customer number one.

So it can repeat. It could also be left null.

Now that's generally not desirable because what it would look like would be order

number one was purchased by nobody.

That's generally not desirable, as mentioned, but customer number one might not be

in your database forever.

You might delete customer number one.

In that event, the order that they made could be removed, or you could just remove

the ID.

Now again, generally you try to protect against that, but there are a situations

where it might arise where foreign keys might be left null.

But you need to make sure, if you're going to allow that, that there is no

verification for null values.

So in other words, you need to make sure that the column is allowed to accept

nulls, but that's up to you.

Some additional constraints.

Unique constraints to simply force uniqueness in any kind of non-primary key

column, and again the credit card was the example there.

A unique index is also still unique, but it indexes the row as well, which again

we will talk about.

But it effectively sorts the row in some kind of fashion, so that it's very easy

to retrieve values a lot more rapidly.

And finally, triggers can be used.

Again, that can simply respond automatically to some kind of table modifications.

So you may change A that triggers change B and so on and so on.

So all those constraints can be used to help maintain the integrity of your

database.

## **Table Design**

We're going to take a look now at some of the fundamentals of table design.

But before we can get to that, you do need a database in which you can create

that table.

But that's very simple in SQL Server Management Studio.

You can simply right-click the Databases folder and choose New Database.

Now we won't worry about all these properties at this point.

We're just looking at table construction.

So all you need to do is provide a name and that's good enough to get started.

So let's just call this Sample\_DB.

And as long as you have a name, that's all you have to do.

Click on OK.

That will create a database for you and you're ready to get started.

Now as far as table design goes, this is something that requires extensive thought

and planning and you would really want to do this on pen and paper first before

you just jump in and start creating tables.

But for some of the fundamental concepts, we can just go ahead and create a table

by expanding our Sample\_DB, and there you see, Tables.

So we can just right-click, choose New - Table, and we're ready to get started.

So we're just going to cover off some of the basics when it comes to table design.

Now the first thing you see here, is it obviously wants a Column Name that you can

pretty much use anything, but let's assume this is a customer type of database.

So the first thing we want to do is to ensure that there are no duplicate values

anywhere in the table.

This is typically accomplished by the use of a primary key.

So a very common approach is to create some sort of identification field first.

So I'm going to go with Customer\_ID.

Now very quickly, the use of the underscore here.

It's not required, but it is a very common standard implementation in databases

that you not use spaces.

All of the administrators and developers out there will expect to see either all

one word or some kind of character that separates different words within a name

such as the underscore.

You can see here, in fact, the names of the other databases have no spaces, and

that's usually because developers always want to avoid spaces.

Some people might use them and some others would not.

You will save yourself a lot of headache if you just avoid spaces.

So there is the name.

The Data Type, if you hit your Tab key, it will automatically select a Data Type,

and the nchar is a character-based Data Type, in this case, of 10 characters

maximum.

Now n simply stands for national, and it refers to a unicode-based character set,

which is supported by multiple languages.

We won't worry too much about that.

But what I'm going to do in my case is just set an integer-based value.

So I can scroll down and I can find int, scroll up, for integer.

And that allows me to just put in a simple number.

Now we're going to come to some other properties of the columns in an upcoming

demonstration, but for the time being, I'll leave it as an integer value, but look

at the null column here – Allow Nulls – says Yes.

In other words, I am allowed to leave it blank.

Now you don't want that to be done either when it comes to the identity field.

So you can accomplish two goals here.

You can ensure that there are no duplicates and no nulls by the use of a primary

key.

And there you see is the option to 'Set Primary Key' in the toolbar.

So you simply have to click on that and watch what will happen to the Null option

here.

It automatically turns off.

And you see down here – Allow Nulls – now says No.

You are not allowed to implement a null value when you are dealing with a primary

key. There must be an entry.

So there's two, as I mentioned, two goals accomplished.

We will have no duplicate values and no ID field will be allowed to be left blank.

Now as for the remaining columns, it is certainly up to you.

But there are number of different approaches that you'll find when it comes to

table design.

But one of the things that you often want to implement as well is to take any

particular field that can be broken down into smaller units and do so.

In other words, don't do something like this:

FullName and then enter in their first and last name in the same value.

That way, if you do so, you cannot search or sort by just the person's last name,

for example, and you might often want to sort alphabetically by a last name.

Well this, kind of, prevents that.

So what you want to do is to break them down as small as possible.

So we want First\_Name and Last\_Name.

Now as far as the Data Types go here, again, we'll talk about this in greater

detail, but since you don't know how many characters are going to be in somebody's

name, then a good candidate Data Type there is the varchar – that's variable

characters.

So I'm going to set that for both of them.

But I probably don't need 50 characters; that's what this value is here.

So it is variable but only up to a maximum of 50.

So what I can do is just make this a more reasonable value.

First\_Name, 20 should be okay and Last\_Name, maybe, 30.

And this will help to save some space in the overall size of your database.

Now if it's a very small database, you don't have to be as concerned with this.

When you start getting up into the server-class databases that have millions upon

millions of records, a savings of 20 to 30 characters per entry adds up to be

quite a lot.

Now you might also see the same thing with something like Address.

You probably would not want to put their entire address into one field because

then you could not search or sort by, let's say, their state or province.

So you would probably want to break this up again into something like

Street\_Address.

And I won't worry about the Data Type from this point on.

Then you might see something like City, State or Province, and something like

Zip or Postal Code.

So those are some of the basic fundamentals of designing tables.

You definitely want your primary key on there to ensure uniqueness and ensure that

there are no nulls.

You generally want to try to break things down into their smallest units possible,

and of course, you do want to set the appropriate Data Types and then determine

whether or not you're going to allow null values.

So in this case, you see all of them are allowing nulls.

It's up to you, but you generally want to then selectively go through and say,

well, you know what, I want to ensure that they must enter at least their first

and last name, for example, in this case.

Now let's also put in one more field.

Let's juts put in Phone\_Number or perhaps an e-mail address or anything along

those lines.

So again you know, I would probably go back after the fact and clean up the Data

Types here and set something a little more specific, but we'll talk about those

later on.

For the time being, we just want to see how you can get in there, create a table,

and follow some of those basic rules of primary key for uniqueness and null

ability and breaking down fields into their smallest constituent component.

So with that done, I'm going to close this table.

I'm going to say Yes to save it.

We'll make some changes, of course, as we go through here.

And let's now just call this Customers\_Table. Click on OK.

And there we have a simple but functional table.

## **Column Design**

Now that we have a basic table design, we're going to take a look at some

fundamentals of column design within that table.

Now you can access the properties by right-clicking the table you just created

and simply choosing Design, and that brings you straight back into the designer

where we created it in the first place.

Now with respect to the properties of any given column, certainly one of the

things that you want to set straight away is the appropriate Data Type, and we're

still going to get to that coming up.

But there are some other properties that are particularly useful, given a

particular type of data.

And the first one we're going to look at is what's known as an identity in SQL

Server.

Now as far as a field such as Customer\_ID goes, that, sort of, indicates that this

is something that's used to identify the customer and it absolutely is.

And that again is accomplished by the use of primary key, which ensures that there

are no duplicate values and no null values.

But as it stands right now, we have to manually enter an integer value for the

customer.

Now that could be fine. Maybe the Customer\_ID is 24361.

In that case, an integer value is fine.

But you might also want to have the database enter a value for you automatically.

And in SQL Server, that's what's known as an identity property.

In some other databases, you might find something like an auto increment or

something similar to that.

But if we scroll down to the Column Properties here, you'll find that there is

one that's called Identity Specification. Currently it says No.

It is not using that property.

But if we extend this, we see right there, Is Identity is set to No.

Well you can just change this to Yes.

And watch what happens as soon as you do.

You see right now the Identity Increment and Seed are grayed out.

I can't access these fields, I can't type anything in there.

But as soon as I change this to Yes, then you see that you get these two options.

So the Seed is the starting value.

The Increment is then what it will count by.

So by enabling it in it's default state here of 1 and 1, my first customer will be

customer number one, my second one will be customer number two, customer number

three, and so on and so on until I have entered all of my customers.

Now you can change these values, if you want to start with something a little

higher.

If you don't want to say, customer number one, you can change this.

For example, I can put in customer 100 as my first value.

And then I could increment by 10 if I want to.

That will give me customer 100, 110, 120, 130, 140.

It doesn't matter; you pick the values.

For simplicity sake, I am just going to leave them at their default values of 1

and 1.

Now this can also be used for any other column.

It does not have to be a primary key.

It's anything that you feel would benefit from an automatically incrementing value.

Now the other option that you can set, of course, is the null ability and that is

entirely up to you.

But if you are going to Allow Nulls, you have to keep in mind that it will allow

the end user to leave it blank.

So if that's acceptable, then by all means go ahead and do it.

Something like a fax number, for example, if that was stored, not everybody has a

fax number.

So you would certainly want to Allow Nulls.

But if you want to force the user to enter something, then do not Allow Nulls by

turning it off. So again that's entirely up to you.

Now another thing you can do to help the integrity of any given table with respect

to the values in a column is that you can enter in an automatic value for them,

if it’s something that's very common.

Now the user is allowed to override this if they want to, but it can help to keep

the values consistent.

If it’s something that is commonly entered, so as an example, looking at State or

Province.

If this is not a particularly large business and all of my business is done within

the same state or province as the one I am in, then I can save a bit of time and

help to ensure consistency by entering what's known as a default value.

So I'm going to scroll back up here and you will see near the top, right there,

Default Value or Binding.

So the whole idea is to simply save a bit of time for the users and to help for

consistency sake.

This way, if it's always the same entry, they can just skip over it.

So let's imagine you do business in New York State, then you can simply enter in

NY as the value, and it will automatically enter that value for every new record

that you create.

Now if it's something different, the user can absolutely change it.

Default values are not permanent or not read-only – it's just something that will

save a bit of time.

So any kind of value that you have that is fairly consistently this same value,

that's a good candidate for the default value.

Now finally, before we get into too much more detail about the Data Types, one

other thing that's very common, and this is a good example.

Is if you have a value that is consistently the same size, then you can also help

to save some space and ensure the integrity of the database by specifying exactly

that size.

If you go with the post office abbreviations for every state or province, then it

is always just two characters.

That way, you never allocate any more than two and you will always help to ensure

that exactly two are entered.

So you don't get any typos, for example, somebody hitting an extra character by

mistake and putting in three or four or something along those lines.

So you know, again the whole idea is to help keep all of the data consistent

within any given column.

So the Identity Value or the Auto Increment in some other databases, the Default

Value and the Size, and, of course, the Data Type that goes along with that can

really help to build a nice consistent and user-friendly table for your end users

when entering records.

## **Table and Column Properties**

We're going to take a bit of closer look now at some of the properties of tables

and columns.

And I want to start with one that's known as Collation.

Now this is something that deals with how characters are treated, if you will,

when searched or sorted or compared against each other.

When you issue a query statement, for example, you have to specify criteria that

says, I'm looking for particular values.

Well Collation deals with how what you supplied is compared against that which is

in the database.

And for the most part, it deals with things like case sensitivity and accent

sensitivity, particularly if you are in a multilingual environment and you need to

support these different languages.

Now it's typically something that is set by whoever does the installation of the

product, but it can be set per database and even right down to the per table and

even sometimes the column level, but usually you don't need to be that specific.

Now I'll show you where you can just find it.

First of all, if you right-click on the table and go to Properties, you will find

it under Extended Properties.

Right there, it says Collation.

Now this is the default Collation that was installed with this particular product

and it generally supports languages that are Latin derivative.

The CP1 stands for Code Page1 and that's really beyond what we need to get into

at this point, but it's just how pages are written to disk, if you will.

But these last four characters here are the important ones.

CI is case insensitive.

AS is accent sensitive.

So case insensitive means that if, for example, I was to search for a person's

last name.

If I did not capitalize the first letter of their last name in my criteria, it

would still work if it was case insensitive.

But if it was case sensitive, I would have to make sure I get the case correct

for any results to come back as matching results.

Now accent sensitive means that I must get the accents correct if I'm in a lingual

environment that uses accents such as French.

But you can change these, as mentioned, but it depends on where and when you set

it.

You can see that, in here, I don't have the option to change this.

You can't just type in a new Collation here; you'd have to choose it from a

drop-down.

But this is a function of what was set at the database level.

So I'm just going to Cancel this quickly.

And if I right-click on the entire Database and go to Properties, under the

Options page there you can see a pick list of all of the available Collations.

And as you can see, there's quite a few.

So for the time being, I just wanted to show you how you can find out what it is.

I could change it here, but changing it would only change it from this point

forward.

In other words, tables and columns that were already created do not change.

I would have to manually write some code to change those existing objects.

So really, at this point, it's just a matter of being able to find out what it is,

so that you can be assured when you are writing some queries that your queries are

going to work, even if you don't match the case, and/or, of course, if you do

have to match the case.

Because if you find one, you see that here is CS, that is case sensitive.

So somebody may have created the database with that Collation.

And if that's the case, you have to make sure that all your case matches when

issuing out criteria.

So that's one of the first properties.

Now if we look at the table itself, you can right-click on it again and choose

Properties.

You see there's quite a few, and you can just find out, of course, what properties

have been set on any given table by just going through each of these pages.

Now again, at this point, I'm not really going to get into what all of these are

all about but rather just how to find the values, so you can gain better insight

into the construction of your tables.

So it's just simply a matter of right-clicking and choosing Properties and

browsing through these pages. A lot of these cannot be changed at this point.

Once you create the table with those Properties, it's pretty much done unless

you're willing to drop and re-create or use some fairly extensive code, but that's

how you can find out.

Now each individual column in the table also has properties.

And you see, if you just expand them here, it tells you some of the main

characteristics right here.

We see our Customer\_ID is the primary key and that's what the little key icon

indicates.

Its data type is integer and it will not accept nulls.

So you get some basic properties right here.

But if you do want to really find out some more information, again, you can just

right-click, choose Properties, and it will tell you everything about that

particular column, including things like the Identity and the Identity Seed and

Increment, which we set earlier.

So that's where you can again just see the values...I cannot make changes here.

But if you do want to make changes, you can certainly just go into your designer

again.

I can right-click and I can choose Modify.

That brings me right back into the table design.

I can also just right-click the table itself and choose Design. Same interface.

So one of the other properties that is certainly worth having a closer look at is,

in fact, the Data Type.

Now we've touched on a few of these already, but these can be changed at any point

in time.

So there is no restrictions, you can hit your drop-down here and simply choose to

a different data type.

But do be aware that the existing information that's already entered into any

cells of the table may not conform to the change that you make.

For example, if you decide you want to shrink the size of this.

Let's just say, for example, I want to make these 5 characters.

Well if there is a value in there that is already set to 10 characters, then

that's going to cause problems.

It's going to tell me that there is information in there that would have to be

truncated or shortened to accommodate that value.

So there is probably a lot of instances where you may not want to do that, but

it’s certainly up to you.

But with respect to the Data Types themselves, you see there is a fairly extensive

list here.

And again, we really don't have the time to go through all of them.

But the properties of the Data Types that you do need to be aware of deal with

the size.

So you're seeing a number of them that have these numeric values after them

indicating how large a value they will accept.

Now the smaller, the better in terms of the overall size of your database.

So if we can save some space here, that's definitely going to help, particularly

if it's a very large database with millions of records.

But you obviously can't go too small.

And again choosing the correct Data Type in the first place is also particularly

important.

Now what I'm going to do here in terms of giving you an idea of what these are all

about.

In fact let's just go to the web page.

You can just search for these.

I just did a search for SQL Data Types and there is the MSDN document supporting

this.

Chances are you have this installed in your SQL documentation as well, but this

just gives you an idea of the categories of the Data Types and some examples.

And then you see here that there are Exact Numerics, there are Approximate

Numerics, there's Date and Time, there is Character and String, and so on.

So you can just click on any one of these, and it will give you all of the

properties and values that are appropriate for that data type.

So it will take a little bit of extra research, but knowing the data type and

knowing the appropriate size will definitely help you when it comes to designing

effective tables that are efficient and, of course, appropriate for the

information that you want to store in your tables.

## **Understanding Relationships**

The next component that we'll discuss with respect to database fundamentals is

the concept of relationships.

Now when it comes to the tables in your database, this again is something that you

want to try to map out and plan very much ahead of actually creating the database.

So it's again very much done on pen and paper, if you will, before you ever get

into an interface.

But before we even get to that stage, you need to understand the types of

relationships that you will find.

And this is something that is typically indicated by the normalization process.

Now to help in the demonstrations here, I have downloaded another database

because, plain and simple, it's a much smaller database and it's much more simple

in its layout and its design.

And it's known as the NORTHWND database.

Now if you've ever used Microsoft Access, which is just the desktop version of a

database application, you may have already seen this database.

But it is fully supported in SQL as well, and it's freely downloadable.

So you can absolutely just go ahead and download that and mess around with this

database as well.

So let's just take a quick look at the tables here.

And you see there is not nearly as many as there is in something like

AdventureWorks, which is another sample database.

We see the tables there are quite numerous compared to what's in NORTHWND.

So as mentioned, this is a much simpler database.

But when it comes to the relationships, the concepts are still the same.

Now in our sample database that we created, we made a simple Customers table, and

we have the same thing here.

So we can expand this, and we see that the columns are fairly similar; we're just

storing the company name, their contact information, and address information, and

that's it.

And that's perfectly fine. Now this is a commercial type of organization.

So the customers generate Orders.

So right there we see a relationship.

Now to actually view this, what we can do is use the Database Diagrams feature

that is right there below the name of the database.

And AdventureWorks has this feature as well, all of them do.

There you see our diagrams, and this allows you to just map things out.

Now another option you can do is to create a view that has a bit of a diagram

application built into it.

But just for the sake of making it easy to see, I'm going to go ahead and create

a database diagram.

So I'm just going to right-click and choose New Database Diagram, and it simply

asks you to add in the tables.

So I'm going to go ahead and add in the Customers table.

And I can just hit the Ctrl key and I can select the Orders table at the same

time, and I can Add them both.

Close this and this will show you that there is a relationship between the two

tables.

And basically it is this line here.

And in fact you see, if I mouse over that line, a little balloon pops up that

tells me there is a relationship between the Foreign Key Orders, and Customers and

Orders.

So in essence, what it comes down to is that you have primary key fields.

Their job, as mentioned, is to uniquely identify every record in this table.

So this uniquely identifies my customers. That's it.

But my customers generate orders.

So over in the Orders table, we have an OrderID field and this is the primary key

of this table, which uniquely identifies every order. That's perfectly fine.

That's exactly what we want.

But one of the things we need to know about the order is which customer ordered

it.

And that is this field right here; that is the foreign key of this table.

So any time you see a primary key of some other table in this other table, that

in almost every case, indicates that there is a foreign key.

And in fact, we're seeing another one right here with EmployeeID because not only

did a customer have to generate the order, an employee had to sell it.

So that is another field – that's a foreign key relationship.

But we won't worry about that for the time being; let's just focus on the

Customers and the Orders.

Now what are you seeing here is the key icon on this side and then a little –

this is actually meant to be an infinity symbol – on this side.

Those icons mean that this is what's known as a one-to-many relationship.

Now that, in essence, means that one customer – that's the primary side –

generates many orders.

That is the standard one-to-many relationship.

And then if you work it in, kind of, a reverse fashion, if you looked at any one

of the many orders, it should be traceable back to a single customer.

In other words, a single order can not be owned by two customers.

It has to be traceable back to the unique customer that generated that order,

and that is a standard one-to-many relationship.

But that's not the only type of relationship that you might find.

There is also a many-to-many relationship.

But this is something that needs to be resolved in databases because it generally

doesn't work.

Now I'm going to zoom out a little here.

I'm just going to right-click and choose Zoom, just so we can see a few more

tables.

And I'm going to bring in a couple more.

Now what you might think about an order just logically is that if somebody is

going to order something, they need to order a particular type of product that

they are looking to buy.

And if you go through the fields here, there is nothing here about the product.

So I'm just going to remove the Customer entirely for a moment.

And then I just put our Orders over here and I'm going to add in another table.

And the Products table contains the products that this organization sells.

So we're going to Add that and Close here.

So you might think, okay well, an order should contain products.

But note that there is no relationship here.

There is no foreign key for the product; there is no reference to the Order in

the Products table.

And you might think about it in terms of this one-to-many business.

Well if I look it any one single order, it is certainly very plausible that I

might order any number of products on that order.

You can imagine somebody is saying I want one of this, two of these,

five of those, and so on and so on, all on the same order.

So one order can have many products. Seems good so far.

But if you reverse it, any one product can show up on multiple orders as well and

that is a many-to-many relationship.

And there is basically no way to accommodate that in the existing structure.

One Order can have many Products; one Product can be on many Orders.

That basically doesn't work.

So what you have to do is introduce yet another table.

And in this example, it's called the Order Details table.

If we Add this one and watch what happens with the relationships, all of a sudden

they appear.

We'll Close this and make a little more sense out of this.

And you see what happens is that this is another table that sits in between the

two, and there are relationships to both.

This is often referred to as an intersection table.

And what you see there now is the OrderID primary key is also here in the Order

Details table in a one-to-many scenario, and the ProductID is also included in a

one-to-many and these two one-to-many relationships resolve the many-to-many

problem.

And then there is a few other fields here as well, but that, in essence, takes

care of the issue.

And then both of these fields together are the primary key of this table.

It requires both unique values to generate an overall unique record.

So order number one could have product number one, order number one could also

have product number two.

But when you take those together, that is a unique record and you resolve the

many-to-many issue.

### **Lesson Test**

Q1: If a primary key is added to a table, what attributes does it provide?

Acts as a unique identifier, prevents null values

Q2: If the Identity Specification is set to "Yes", which of the following

properties can be controlled?

Identity Seed and Identity Increment

Q3: Given the Collation default SQL\_Latin\_General\_CP1\_CI\_AS, what does the end

CI\_AS stand for?

Case Insensitive\_Accent Sensitive

Q4: Match the description to the Data Integrity category:

Referential integrity: related records from one table to another table

**X** Entity integrity: uniqueness of collective items in a row

User-defined integrity: any rule defined in relation to a specific dataset

**X** Domain integrity: uniqueness of collective items in a column

Q5: Which of the following commands are used to manually handle explicit

conversions from one data type to another?

CAST and CONVERT

Q6: What is the relationship between these tables, given the symbol linking them?

Customers{PK:CustomerID} PK-INF Orders{PK:OrderID}

One-to-many

-----------------------------------------------------------------------------------------

# **DML Statements**

## **SQL, DDL, and DML**

SQL or as a lot more frequently referred to as sequel, again, is the Structured Query Language. And this is used, as the name indicates, to query the databases.

Now we're going to look at a couple of different categories of SQL statements.

But before we get to that, SQL in and of itself, does have a few different varieties, if you will. Each different vendor of, particularly, server-class databases found that SQL in its native format, if you will, wasn't always appropriate or at least able to accommodate all of the requirements.

So what you find is that there are a few dialects, if you will, of SQL.

Microsoft implements it as Transact SQL.

Oracle implements it as Procedural SQL.

So you do find that there are a few different, you know, as mentioned, flavors I guess, if you will, of the SQL language, but most of its core capabilities are consistent.

Now DDL and DML that you see here in the title and a couple of others are categories of the SQL statements. And we'll come to those in a moment.

So they're used for the following functions to access the information in a database management system:

We can define the data with SQL statements, we can then retrieve the records, we can manipulate the information, we can control who has access to it in terms of permissions and security, we can control what information can be shared, and we can control the integrity of the data, making sure it's always consistent and correct.

Now for some of those subcategories of the SQL language, we find the Data Manipulation Language, or DML, statements, which are used for, as the name indicates, manipulating the data, making changes, editing or updating records, those are manipulations that we perform.

DDL stands for Data Definition Language, and this is what we use to create objects in the database such as a table or a column in a table. You have to define those, using DDL statements.

And the Data Control Language, or DCL, statements, allows you to determine the access, who is able to access this information, so your security is typically maintained with DCL statements. Now some common ML statements, Manipulation Language.

These include SELECT, INSERT, UPDATE, and DELETE, and in some cases what's known as a BULK INSERT.

Now SELECT really doesn't manipulate per se. In other words, you don't change anything with the SELECT statement. That is simply the retrieval statement – show me the records from whatever tables.

INSERT is to create new records. For example, you create a new customer, you issue an INSERT statement.

UPDATE: You certainly manipulate with this. This is where you make changes. You want to change a person's address or their phone number, that would be an UPDATE statement.

And DELETE would be to remove. This can be for either a single entry in a table or for the entire table itself or even the entire database.

Now BULK INSERT is the same idea as the INSERT statement, but it's typically used for very large operations where you have some kind of input source such as a file, maybe, someone has sent you. Here is an Excel spreadsheet with 50,000 new records.

You don't want to obviously manually type them all out. You simply take that input file and you provide it to the database. And with a BULK INSERT statement, it will extract the values from that file and place them into the table automatically. But what you also probably don't want to do with that is a lot of logging. And BULK INSERT statements support the use of not being logged because it doesn't really matter if it fails. You still have the original file, so you can try again.

And logging is, of course, the ability to recover from a failure. So bulk inserts are kind of unique, but they are still inserts, and they manipulate the records in the table.

The DDL statements include the CREATE. So when you are creating a new record in a table or a new table or even a new database.

ALTER allows you to make changes to that, which you just created, so if you want to change some properties, for example.

DROP is to remove. Again, that which was previously created.

TRUNCATE TABLE is to remove records. Typically this is something that is done after a backup operation. So once you've performed a backup of, let's say, for example, a transaction log, you can remove it, you can clear out the log.

That way, you don't occupy up all this additional space. You can do the same thing with tables as well. Maybe you've archived off a table into a different data warehouse, for example.

From that point you can truncate the records out of that table. Enabling and/or disabling triggers – you can turn them on and off. So you have this automatically executed code.

You can simply say you want those to be allowed or you don't want them to be allowed.

In some cases, you just simply want to turn them off.

If you're making extensive changes to the structure of various objects, you might end up firing off a bunch of triggers.

So just while you are doing those extensive changes, you can simply disable the triggers. So again, all of those put together, in essence, allow you to perform all of the tasks that are necessary in your day-to-day operations of a database environment.

## **Using SELECT Statements**

Being able to retrieve information from your tables is perhaps one of the most critical components of a database.

Obviously we want it to be able to store the information effectively and efficiently, but it's the data retrieval that provides us with information. So we're going to get started with some basic SELECT statements that will show us how to retrieve the information from a table.

Now there are a number of ways that you can get started with this, but we are going to take a look at just the simplest SELECT statement that there is in a new query.

And you can do that by – just for starters – selecting the appropriate database.

So again I'm going to use the NORTHWND database here just because it's a bit smaller and a bit simpler. So we can just get started by clicking on the New Query button here in our Management Studio.

And that will just pull up a blank window that we can start issuing out some code for. Now in terms of the correct database, since NORTHWND was already selected, that should be fine.

But you do see that when you select the window here, there is a drop-down list where you can choose the correct database if for any reason it's not the right one. So as long as that says the database you want to deal with, you're good to go.

Now the syntax of a SELECT statement in its simplest form is quite simple, and it is simply the word Select. And then what you need to do is to supply the names of the columns from the names of the tables that you would like to see. Now as mentioned, we'll go with a very simple example to begin with – the Customers table. Now I can expand this and I can see while those are the names of the columns that I want to see, so I can start referencing each of those column names. Or if you want to see all of them and you're not particularly concerned with being selective, then you can make things very quick and easy by simply saying select everything. And in Transact SQL syntax, that is simply the star key. So Select \* means Select everything. I'm not filtering my records, I want to see the entire result set.

Now the next keyword is from which table do you want to select. Even though we've highlighted the customer table over here, you still have to tell it – when you're writing code – which table. And the keyword is from and then the name of the table, which in this case is customers. Now the dbo component. We will talk about this a little bit later, but you don't need to reference that at this point. So I can just start typing in the word customers, and you see, in fact, that Management Studio includes IntelliSense, which picks up on what I'm typing. And this really helps to speed things up and eliminate typos. So having entered in just those few characters, it's giving me options of all of the objects that start with those characters, and there's Customers. So I can just double-click on this and that completes my syntax. And if you see any typos or anything in here – let's just temporarily remove a letter here – this will automatically pull up the red squiggly line that tells us there is something wrong with this particular piece of code, so we can correct that.

So there it is. The most basic and/or the simplest SQL statement – Select \* from Customers – click the Execute button to run this, and there are the results. So what I'm seeing here are all of the columns for all of the records in the Customers table. So I can simply generate SELECT statements like that to pull back all of the records that I want to see. Now in terms of selectively pulling out just some of the columns, you absolutely can do that because maybe you aren't concerned with seeing the ContactTitle or the ContactName or the City or the PostalCode or anything. You know, maybe you only want to see a few of their values. So you absolutely can do that by simply specifying the name of the field instead of every single field. So let's just try that. We'll get rid of the asterisk here, and we'll start typing in some of the column names. And again, with the IntelliSense, you see that as soon as you type in the first few characters, it narrows things down. So I can just continue typing and by the time I get to Cus, there is CustomerID.

So if I want to see that I can just double-click on it. Now if you want another column, you have to separate the list of fields with commas. But let's just run this for now and see what happens. We only, in this case, have specified the CustomerID. We can Execute this and there we go. So now we filtered out everything except for the CustomerID. Okay. Let's pull up another one. Let's bring in their phone number, so we can hit a comma and then just start typing in phone, and there is the Phone right there. So we can just double-click and we can execute that and now we see the phone number along with it. So those are the fundamental components of a SELECT statement. You have to specify which columns you want to see separated by commas, or you can simply say, show me all of them. Then you also have to specify the table from which you are selecting. Now you absolutely can select records from multiple tables as well, but we'll come to that a bit later. For the time being, we just want to focus on the fundamentals of a SELECT statement, making sure you specify the correct columns, making sure you specified the correct table, and making sure that you are in the correct database. From that point, you will start to see your records. And I will show you one little quick other option. I'll close this entirely.

And if you do just want to see a fair number of records from the whole table, you can just right-click and right there you just need to Select the Top 1000 Rows. Now there may not be 1000 records, that's fine. It will return as many as there are, up to 1000. So I'll just click on this, and there, you see, is an automatically generated script that says SELECT TOP 1000. So that's another thing you can do. So even if there were, let's say, 2000 records in the table, you could easily just change this from 1000 to 2000, re-execute, and you'd get more records back. So there you see some of the syntax in a little more detail, but it's still selecting each column name by name, separated with commas. In this case, it is placing them inside a square bracket. That's not particularly necessary, but it won't hurt anything. And then in the FROM statement, it actually qualified the entire object name. So it's not just the Customers table, but from the dbo – this is called a schema – and again we'll come to that later, but from the NORTHWND database.

So this just clarifies it, if you will, if there by chance happen to be two tables or two columns, is more likely with the same name, this ensures that you qualify entirely so that there's no ambiguity. It's not going to come back and say I'm not sure which column you're talking about because there's two with the same name. Now you should avoid that anyway, but that's why it puts in the entire name here. Now as mentioned, we'll come to more of that as we go. But for that simple SELECT statement: select the columns, separated by commas, FROM the table, and your results will come back.

## **Filtering Results**

Now let's take a look at how you can filter the result sets of a SELECT statement by using a couple of different keywords that are designed to filter the results because there's not all that many instances where you simply want to see every record in the table.

So a couple of quick things before we get started. Multiple SELECT statements issued here, and when you are using SQL Server Management Studio, you absolutely can run more than one statement at any one given time.

But you can also just select the statement in terms of just highlighting it. So you can see you can just highlight the code that you want to execute and you can then execute just that piece of code.

The other thing you can do is you can document what the code is for. And you can see that if you just put in two dashes, then everything that follows those two dashes turns green. These are what's known as comments. And they're simply meant to inform the reader what this code is for and not necessarily any other reader but even yourself, so that you can simply document, this is what I'm doing with this code.

So you see the comment simply says, I want to Select all Customers. Then there is the statement.

So this has already been executed, and there is my entire result set: 91 rows come back when you specify Select every Customer. That's all well and good.

But the first filter we'll look at here is to implement something by, in this case, the Country. Some kind of filter that says, well I don't want to see every customer, I only want to see those that I look after. Maybe they're in my sales territory or sales region or anything along those lines.

So there you see it's still the same first portion: Select everything in terms of the fields, that's show me every column. But I'm only concerned with those who are in the United States. So there you see is what's known as a Where clause.

Where Country, which is the name of the column, right there, is equal to and then in single quotation marks. Because this is textual information, the single quotes help to inform SQL Server that this is not part of the code, if you will, rather just a value that I'm supplying for it to use as a comparison.

So by specifying an example of what it is I want to see, I can execute this and I will only see customers that are in that country. So let's highlight this, Execute, and there they are.

And you can see now there are only 13 rows. Now if we scroll over and we find the Country field, we should see nothing but the USA. There they are.

There are 13 customers in that Country. And I'm still seeing every column, but I've obviously filtered the result set quite significantly. The other 78 customers do not show up. Now you can filter it by more than one column. You can filter by as many as you want. So with this, you see the next statement filters with an "and" Clause.

So now what has to happen is that I've got the same thing, selecting everything from customers, where Country = USA, but I also want to see within those USA customers only those in this particular sales region.

So I can see that we have the Region right there, and I'm seeing every customer in the USA.

So now I want to be even more specific, and I only want to focus on the WA region. So this, you might assume, is my sales territory. So now I can select this code and run that, and there you see now I'm down to only three rows. There are my three customers, for example.

So the implementation of an "and" clause requires both pieces of criteria to be satisfied, not just one of them.

Now I should mention that you could also use an OR statement. And if you do that, then either one of those can be met; they will all come back.

So if the Country was USA or the Region was WA, then the records would be returned with either of them. So let's see what happens with that one. Now this is probably to show me all of the USA again because there's only a single region of WA within that. And I'm just going to see all of the other ones. So I'm back to 13 rows because they came back because the USA was satisfied.

So even though I specified the WA as an "or" clause, all of the other ones came back because it said, well, either one of them can be met. So you definitely need to be careful with ands versus or. And will almost always reduce the number of records; or will almost always increase the number of records because we are saying "meet both pieces of criteria" versus "meet either one of them."

So I'll just change this back to an "and" for the time being and we'll see more examples of that. Now the other thing you can do is to just sort of reverse the logic of this and use a NOT statement. So this exclude results by, as mentioned, sort of negating what you specified earlier. So there is the same statement of where Country = USA. So to negate that, you can, sort of, isolate this in parentheses. So there you see a set of parentheses, and then you can place the word NOT in front of that. And it says, okay, show me every customer whose country is not the USA. So let's Execute this.

Now in this case, I didn't say, show me everything. I just want to be a little more focus I'm only showing the CustomerID and the Country. So we'll Execute here, and what we should not see in the result set is USA. And there we are, 78 rows – the 13 USA customers have been excluded. Now I can scroll through and I'm not going to see anyone in USA. So again, you can just, sort of, flip-flop the results by using a NOT. You do have to remember to take your existing criteria statement and enclose it in the parenthesis, then negate it with the NOT. The other filter you can implement is a Range. And this allows you to specify a minimum and a maximum value. And in this case, I have gone to the Order Details table because it has Quantity field. This is just very commonly done with numeric values or maybe date values. So in this case, you see, I'm looking for the OrderID and the Quantity from the Order Details table, where the quantity is between 10 and 20. So that gives me the range, and this is inclusive of 10 and 20. So I should see values that start with 10 and end with 20. So let's highlight this and Execute that. And there we see are the Quantity values, falling between that range. So there is 10s and there is 20s. So again, they are inclusive.

Now you could achieve the exact same results if you did something like this: greater than or equal to and less than or equal to. Okay. That would effectively be the same thing. But you see, I've got a syntax error because I'd actually have to say, where quantity is greater than equal to 10 and quantity again would have to be specified, so that's a little bit redundant. So I like the between, because it's obviously a little simpler, and I don't have to type out the word quantity twice. But this should return the exact same results. So clearly, between is a little simpler, but you absolutely can use those mathematical comparison operators of greater than, less than, equal to, and so on and so on. But between 10 and 20, a very simple option for Range filtering. And that's how you can just reduce the number of records that come back, so you can focus in on the values that you are interested in and not the entire result set.

## **Ordering Result Sets**

Now let's take a look at how we can sort the results of a query, and this is very common for things like the highest sales or the best-selling product, things along those lines. And it's still the basic SELECT statement, and you still, of course, have to just determine which table, which columns, and then of course, which field, which column you want to order the result sets by. And in fact, Order By is the term that you use. So here is the basic SELECT statement. Select, in this case, my OrderID and the Quantity from the Order Details table. So I have already executed that and there are my results. But you can see here that, currently, it appears to be sorted by the OrderID, where maybe I'm looking for the Quantity to be the sorted field. So you can see that these are all over the place in terms of their sort order. So that's perfectly fine. We can just add in an Order By clause. And in this case, you simply specify which field do you want to Order By. So I want Quantity, and again, you can just type in the first few characters and there is your IntelliSense. So if it's now the first one that I want here, in fact, all I have to do is hit Tab and that will complete that, and I can execute and now I see that the result sorted by the Quantity field. And I can just go through these, and these, of course, are sorted in ascending order.

Now that's the default to go from one up in ascending order. But you can also reverse that if you want to by explicitly saying descending, and that's all you have to do is just type in the first four characters and that will simply reverse that. So I can Execute now, and now I'm seeing it from highest to lowest. And again that is definitely very common when looking for the top sales that, you know, that person who has spent the most money. I would want to see it in descending order. Now it does support ascending as an argument here. So I could just change this to asc for ascending and that would reverse it yet again, but ascending in fact, is the default. So if you just leave it blank, you see the same results. So it's up to you. Some people like to explicitly see in the code what the sort order is, but ascending is the default, so that's up to you. Now it doesn't just work on numeric fields. You could do it by a person's name, for example, as well. So if we look at something like the employees, we see under the Columns there, there is a FirstName and LastName. So let's try a SELECT statement on that. Let's Select \* from Employees and we'll just execute only this part and we can see there results. So there we see are the LastNames, and they are not currently in order.

So let's order by LastName, and again we'll Execute only that. And now we should see them sorted by the LastName. And then the same thing if I want to change the order, I can do descending because you see it's currently A to Z. But if I do it this way, it will be Z to A. So there we go. Everything sorts appropriately, and you know, you can do that a pretty much in any kind of field that you feel would be appropriate in terms of sorting. But your Order By clause is how you implement that sort order. Now you can sort by more than one field as well, but I don't have a very good example here. You could sort by LastName and FirstName, but what happens then is that it will only sort by the FirstName when there are duplicates in the LastName. So if for example, let's just say, we had a very common LastName such as Smith. If there were three Smiths, then you could sort by the LastName and all the three Smiths would be one on top of the other. But then it would move to the next column to then ensure that if there was, let's say, an Ann Smith - that starts with A - that would come before a Peter Smith - that starts with P. So as mentioned, you can absolutely sort by more than one column, but it will never move to the second column unless there are duplicates in the first. And that's the use of your Order By clause.

## **Using Aggregate Functions**

Sometimes you need to ascertain information from your record sets that isn't overly attainable just by looking at the values. Now what I mean by that is let's just put in a Select \* statement ahead of this and we'll pull up everything from Order Details. Now we'll Execute just that, and there's all of my records. What I was referring to in terms of not being able to ascertain, just by looking at the results, are things like what's known as these aggregate functions. They refer to the functions that you might be familiar with if you've used anything like Excel - standard numerical functions that operate on some kind of value to return some kind of value to you. So they are fairly self-explanatory - min is the minimum value, max is the maximum value, count is simply how many entries there are, and avg is the average of all of the values.

Now these are particularly useful when you're dealing with numeric values. So for example, I'm seeing here the UnitPrice, I'm seeing the Quantity - things like that.

So I might want to know, well, you know, what was the highest amount of Quantity entered for any particular order? What was the lowest? How many entries are there? And what was the average? So I can do that for really any field I want, but I'll use Quantity for the time being, just as a simple example. So in terms of the SELECT statement, let's just comment out the first part here. There you see is the field on which you want to apply the functions - so in this case, the Quantity field. So I have two choices in determining the minimum and the maximum. I can scroll through the results until I find what appears to be the minimum; now there is 1. So I can assume that that's the minimum, but maybe there's a 0. For some reason, maybe, it's just a typo. You know, who would order nothing of something? But I could use it to find typos. But I would have to scroll through and take note of just about every value to determine what the maximum is. So obviously that's not very efficient because I can see down below here that there are 2,155 rows in this results set. So I don't want to go through all those looking for the highest value.

So our aggregate functions will tell me. So when you do want to apply an aggregate function, you do have to qualify the name of the field in parentheses; otherwise it will not work. So what you do is you specify the name of the function first and then you apply it to the name of the field in parentheses. So let's find the minimum value first, and we know that that's going to be one. But we'll still execute the statement as is, and I should expect to simply see the number one. And there it is. Now it says No column name here. I will tell you that you can give a name to this column if you want to, but that's something we'll address later. For the time being, I know what I'm looking for. So I don't need to worry about this making sense to anyone else. So to find the highest value, you might imagine - and quite correctly - we simply have to change this to max, re-execute the same statement, and there it is: 130 is the highest quantity that was ordered. So I could go back and run my SELECT statement here, but I would not find anything that was higher than 130.

And if I want a count, that's not particularly useful in this particular case. Because it's simply going to tell me how many entries there are, which I would expect to see the same as when I ran the SELECT statement saying show me everything. But when you do a count, it does not include nulls. So in this case, I'm expecting to see the same amount - 2155. And there it is - 2155. So that's exactly what we saw when we ran the SELECT statement. But that's useful as well because that tells me that there are no nulls. And you wouldn't want to see a null value in a field like Quantity because again that would indicate somebody just probably made a mistake. But entered in a null value for the amount that somebody had ordered. So you know, that's useful as well.

And finally, the average will tell me the average quantity for all of the orders. So I'll Execute that and find out what the average is. And it's 23. So again, that's simply the aggregate functions that you can use. Now you can apply to a different field as well, no problem. Maybe, you want to find out what the average UnitPrice is instead. So you absolutely can change that. Instead of doing it on quantity, simply do it on UnitPrice. And that will tell us the average unit price, in this case as well, $26. I can do the exact same things with my mins and max. I can find the most expensive, and I can find the least expensive with the min and max. So 263.50 is my highest selling unit, and $2 is my minimum. So again, those are your aggregate functions, and the key things are to ensure that you specify the name of the field in parentheses and then the function prefacing the name of that field. And then simply just specify which aggregate function you want to apply, and that will return those values to you. So you can determine that information without having to read through every single record of a Results set.

### **Lesson Test**

Q1: Given the Order Details table, what is the SELECT statement to find the average UnitPrice?

Select avg(UnitPrice) from [OrderDetails]

Q2: Given Select \* from tbl\_USA, which SQL statement will select all records except those that have Maine in the State column?

Where NOT(State = ‘Maine’)

Q3: Given the results on LastName, which of the following are the ORDER BY clauses to sort LastName in ascending order?

Order By LastName, Order By LastName asc

Q4: Given a table named Customers, which of the following is the SELECT statement for the result shown?

Select CustomerID, Phone from Customers

Q5: Match the sub-language to the appropriate statement, where each sub-language matches three statements.

DDL (Data Definition Language): CREATE, ALTER, ENABLE TRIGGER

DML (Data Manipulation Language): BULK INSERT, UPDATE, DELETE

-----------------------------------------------------------------------------------------

# **Multi-Table Queries**

## **Using Inner Joins**

We are going to take a look now at how you can start to retrieve data from more than one table. And the way that I'm going to approach this is, rather than just typing in the code, I'm going to use the view builder, which gives you a graphic environment, so you can visualize what's going on a little more easily. So I'm going to use the NORTHWND database here again. And if you expand that, you'll find that there is an option for Views right there. And Views are essentially a saved query. It saves the SQL statement so that you can execute it over and over again and, of course, get the most current records that are in the database. So if we expand Views, we'll see that there are already some of these present. And again, that's just because this is the sample database. But we'll start with a fresh one. So you can just right-click and choose New View. And all you have to do is select the tables from which you want to retrieve records. Now I'm going to start with just one. I will just go with the Customers table and choose Add and I'll Close this window here. And you can see what happens is that this already puts in the code automatically as you start selecting fields. So just to quickly see what happens here, I'm just going to select the CustomerID and that's it. I'm not too concerned with the Columns at this point. And you can see that the statement is automatically generated -SELECT CustomerID FROM the Customers table.

And we can execute that by clicking on the exclamation mark here, and we see the results. So this is showing me every customer from the customer table. But as mentioned, we want to see records from two tables; so we can do this.

Let's just clear this out here, and what we can do is add in a second table. So you can just right-click anywhere here inside the window or you can click this button here to add in another table. So we'll add in the Orders table and we'll choose Close. Now as soon as this table goes in, you'll see that there's a change to the SQL statement that was in there. So let's just move this out of the way for a minute, and you can see it's currently saying SELECT FROM Customers.

But as soon as we add this extra table, because of the fact that there is already a relationship between these two, we will see that the statement has updated automatically and, in fact, even see that here the query has changed. Close and now look at what happens.

We have a significant amount of additional code here. But the key aspect of this statement is this right here - what's known as the INNER JOIN. This is referring to the relationship here. Now I'm just going to move this table down a bit further. And you can see that here is the primary key of CustomerID relating to the foreign key CustomerID in the Orders table. And that again is how you relate the records. The order needs to know which customer owns it; so that's the relationship. So the statement here says, in essence, this is how the two tables are related.

So this is what's known as an INNER JOIN and it simply means that the two tables are joined together by this statement here. Now again, don't concern yourself too much with the dbo portion of this here. We will come to that later. So you see here it's Orders table is being joined to the statements.

So we're saying JOIN the Orders table ON the field that creates the relationship. And that's: Customers.CustomerID is equal to Orders.CustomerID.

Because it's the same field in both tables; it's just a primary key here, a foreign key here. So the reason it's prefacing the name of the field with the table is because it's the same field name. It's CustomerID in both tables. But this qualifies it.

This one says that the value in the Customers table should be equal to the value in the Orders table. In other words, if you look at an order, any particular order, and let's use a simple example, saying order number one, you should see a value for the CustomerID.

And that value should be equal to a value that exists in the Customers table. So that's how you create a join statement and connect these two tables. You are basically saying I want to see records where there is a CustomerID in the Orders table that is equal to a CustomerID in the Customers table. So what this will result in is that, whatever fields I select, I'll see the information for those records. But to further clarify this, what you will see are the customers who have generated orders. So we can select just those two fields, CustomerID and OrderID, and I will see a CustomerID value for any particular order that they have generated. So we can execute this statement and see the results, and we're seeing that this customer, ALFKI, generated that order. The same customer generated that order, and the same customer generated that order. So again, that's the standard one-to-many relationship.

One customer can absolutely have many orders, but if you look at any single order, it is traceable back to the single customer who purchased it.

So again, we can include the CustomerID field from this table as well, but it would simply, redundantly, show the customer. But that's exactly what we asked for - we want to see value that is equal, the same value in both tables. So that again is what's known as the INNER JOIN statement. And it's how you specify the relationship between those tables, and it's how you can retrieve records from more than one table at once. That's the inner join.

Now that is the default type of join, but it's not the only type. We will talk about some others later on. But if you just add in an extra table where a relationship exists, the default is what's known as this INNER JOIN. And it, kind of, refers to the fact if you look at this visually: this table is inside of that one, if you will.

As we read from left to right, the Orders table is inside of this visually. But basically, that's what you do when you need to select records from multiple tables at the same time - you have to specify the join, which determines the relationship between those two tables and retrieves all of the records where there are matching values in both tables.

## **Using Outer Joins**

We're going to have a look now at a different type of join statement, and again I'm just going to use the graphical View Designer to demonstrate this.

So I'm going to right-click and choose New View. And I'm going to add in the same tables, the Customers table and the Orders table. And we see that this puts in our statement automatically, and again there is our default INNER JOIN.

So I can just quickly demonstrate – pull in CustomerID and the OrderID, and again we see the relationship between those two tables. And if I execute this, I see that there are 830 rows. So I see the customer, and I see the individual orders that that particular customer has generated. That again is our INNER JOIN statement. But in essence, this means to retrieve all of the records where there is a matching value on the CustomerID field. But that's not always the case. There may be mismatches on the CustomerID field in situations where the CustomerID here does not match the value that is over here.

Now these can be located by changing the join type. This is what we see as the INNER JOIN. And if, in fact, you take your mouse and you just hover over this little diamond in between the two tables, it tells you INNER JOIN, looking for equal values on the CustomerID field.

But you can right-click on this little diamond, and you see two choices here – Select All Rows from Customers or Select All Rows from Orders. Now before I get to what that really means, watch what happens with the icon itself. We see this little diamond. I'm going to right-click, and I'm going to say Select All Rows from Customers.

Look what happened – it squared off on the left-hand side. Because again, just visually, this is the table that's on the left. And look at the statement. It now says LEFT OUTER JOIN, as opposed to an INNER JOIN.

Now I can just quickly remove this and do the other options: Select All Rows from Orders. And you see, what happens is it squares off on the right-hand side, and now it says RIGHT OUTER JOIN. So what does that mean? I'm just going to take this off for a minute; so it's back to the way that it was – the inner join. Again, to quickly refresh, the inner join says to look for matching values on the CustomerID field from both tables. But there is not always a match.

And what I'm referring to is the fact that you could have a customer who has not yet generated any orders. So what's going to happen, in this case, is that there would be null values.

So in other words, they simply haven't purchased anything yet. So this is what you can do to locate those particular customers. You can use the OUTER JOIN. So that's what it means to show me all customers. So if we want to, sort of, translate this into English, it says select whatever customer information, regardless of whether or not there is associated or related order information. So let's look at our total again.

We ran this query and got 830 on the INNER JOIN. Now we'll say show me every customer regardless of whether or not there are orders. And let's find out how many we get back.

Lo and behold, 832.

So we have two customers who have yet to purchase anything. Now how do we sift through them? How do we find those customers who have yet to generate an order?

Well it simply means that there would not be any order at all placed for that particular customer. So what we can do is use the null criteria – we can look for null values.

So what we should find if we're looking for every customer regardless of whether or not they have purchased anything, then there should be some nulls in the OrderID field. Now we could scroll through them all.

We'd find them eventually, but we can make it a lot easier on ourselves by simply using a Filter and looking for null. So on the OrderID field, because it would not have an order, we can go to the Filter. And the syntax for this is simply "is null." Hit Enter.

That will change that automatically to the correct syntax. And there you see – WHERE OrderID IS NULL. And if I re-execute this now, there they are.

These two customers are the ones who have yet to generate any orders. So this is very useful –not only do I want to see the customers and the orders they've made, I want to know which customers aren't purchasing anything.

So we can contact them and, of course, try to boost the sales to those customers. So that's one use of the OUTER JOIN – finding customers with no orders.

Now the other side of that – let's just take this off and set it back to the INNER JOIN. The other side of that is to find orders that have no customer. Now this ideally should not happen. **Referential integrity requires that if you are going to create an order, you must enter a customer.** But you never know.

Over time, through standard maintenance or changes, it's possible that somebody – one way or another – violated referential integrity and somehow managed to get a customer in here who does not exist.

Or it could be the more likely case that the customer was deleted. Now again, referential integrity can protect against those kinds of things but you can turn that off temporarily.

So again, after a fairly decent amount of time that your database has been in use, it is possible to find violations of referential integrity. You somehow might see a blank entry for the CustomerID of an order, or it might reference an incorrect customer.

So we can basically just flip-flop this. We'll just right-click on our icon here and say Select All Rows from Orders.

And now we'll just do the same thing – instead of looking for nulls on the OrderID, we want to look for nulls on the CustomerID. So I'm just going to cut this out of here, paste it in here, and I've just reversed the query.

Now I'm saying show me every order regardless of whether or not it has a customer that owns it. So I can execute this, and we get no results back. Now this did not fail. This is fine.

This means that all of my orders are in good shape. They all reference valid and existing customerIDs. But if a record was to come back here with, let's say, a null value in the CustomerID, it means that you have what is sometimes referred to as an **orphan record**. It simply has no owner, and they do pop up from time to time.

So finding none is a good thing in this case. But from time to time, you might want to do that to see if you do have any orphan records. And in either case, you're simply looking for those records with no matching values on the primary-foreign key relationship, and you do that by implementing your outer joins.

## **Using Cross Joins**

We're going to take a look now at another join type known as a CROSS JOIN.

Now I've gone back to the standard code window for this example, most notably because the View Designer, for whatever reason, does not support the use of cross joins. Now they are fairly specific in their use, and there are not a whole lot of instances where you will find them useful, simply because of the number of records that are returned by CROSS JOINS.

But they are useful in a lot of cases for what-if scenarios, and that's typically where you'll find them. Now just to give you an idea on the numbers that are returned by cross joins – and they are almost always very high – the first thing I've done here is to select out the OrderDate of a particular product.

So I want to simply find on which days has this particular product been purchased. So this is a standard INNER JOIN and it shows me 2,155 rows. So I'm seeing this product was ordered on that particular day. Now it's by default being sorted by the date. I could certainly sort it by the product; doesn't really matter. All we are concerned about here is the number of records. So 2,155 of that product being sold on whatever particular day.

But you could imagine that there are a lot of days where any particular product is never sold. So this is where you might implement a CROSS JOIN, not obviously to find instances where it did not sell. There are going to be a lot of days where any particular product did not get purchased. But it's basically to give you this what-if. Well what if we sold every product every day? What kind of numbers will we be generating in that case?

So that certainly can be useful, but you definitely need to be careful with cross joins because what it does is it **takes every possible combination of records and returns them**. So let's just use some easy math for a minute here. Let's imagine we only had 10 products and that we've only been in business for 10 days.

If you were to create a CROSS JOIN saying show me every possible combination of product in days, **it simply takes all 10 products and multiplies them by all 10 days, and you would get 100 records back**. Now that's obviously very small. There are a lot more than ten products, and we've been in business in the NORTHWND database for a lot more than ten days.

So I've created a cross join here, and there you see what happens. You simply CROSS JOIN in the other table. You don't specify any kind of JOIN statement or any one field equals any other field.

Cross joins don't use those because it simply says show me the possible combinations of all columns from all tables that you reference. So it's a pretty simple statement. I'm still looking for just the OrderDate, right there, and the ProductID. But I'm cross joining in the Order Details table where the relationship exists, and look at the total number of records: **1.78 million rows came back**, and what we're seeing is a simple what-if. Well there they are. I could have possibly sold product number one on this day, this day, this day, this day, this day, this day, this day.

So I am just seeing complete redundancies of every product on every possible day. So you clearly can see the number of records here is enormous. Then we move to the next day, and we see the same products again. So basically this is just a complete...well this is what it would look like if you sold every product on every day.

Now again, just to give you an idea in terms of the math, here is another CROSS JOIN.

And this one deals with Sales Reps and Territories in the AdventureWorks database. And this is just really for the math because what I've done here is to select out simply the salespeople. Show me how many salespeople I have. So I'm pulling out what's known as their BusinessEntityID. This "as" statement – we will discuss this later – but this is simply an alias, so I can name the column something else. So I'm going to Execute this, and we see that there are 17 rows. So I've already noted that in the comments section here. So I have 17 Sales Reps. Now I am going to pull out just the sales territories from the territories table. So we can Execute that, and we take note here that there are 10 rows. And I have noted that here as well. So if I implement a cross join, it would say well what if we had every Sales Territory represented by every Sales Rep?

Now in practical terms, of course, you wouldn't do that. But again, it would just give you the idea of the math here. So cross joining – it simply takes the 10 territories, multiplies it by the 17 Reps, and we should see 170 rows. So let's highlight this code and Execute; sure enough, 170 rows. So again, you know, that's not a very good practical example – you would not assign every sales rep to every territory. But it does demonstrate the math, and it is good in some cases for those what-if scenarios. What if we were able to sell every product in every territory or every product on every day? What kind of numbers would we be seeing?

So this certainly can be useful, but you definitely need to be careful in terms of the results because if you have thousands of records in one table and possibly even hundreds of thousands in another table, your results set is going to be enormous. And it could take hours to run possibly.

So you definitely want to throw in some criteria, you know, filter the results – only focus on a particular product or a particular sales territory. You know, what would happen if we sold just this one product in every territory? Maybe you could get a better idea of some useful results there. But cross joins always simply take all of the records from one table and **multiply** them by all of the records in the other table, and that's your results set.

## **Combining Result Sets**

Another way that you can combine the result sets from multiple tables is to use what's known as a UNION operator. Now this is not a JOIN type but still deals with any other type of JOIN.

But it is simply to take one result set of any given SELECT statement and, in essence, paste them together, if you will, with the result sets from another statement. Now I'm going to use both the graphic viewer and the code here to demonstrate this because the code itself does get a little bit complex.

But in terms of creating it, it's very simple to do just in the graphic editor here. So you can see what I've done is to just use my NORTHWND database, and I've pulled in Orders, Order Details, and Products.

And I've only selected the OrderDate, the ProductID, and a Quantity. And that was just to demonstrate that you could have additional fields if you want. I wasn't overly concerned with the Quantity.

But I simply checked those off there, and that made my statement automatically. But what I then wanted to do was to focus on just a particular sales year. So on the OrderDate column, I put in a filter – BETWEEN January 1, 1997, and December 31, 1997. So this gives me all of those products' sales for the entire year.

Now that had to do some conversions on the DateTime column – we won't really get into that at this point – but it did it automatically.

All I did was type in this, and the code was entered automatically. So there's the statement. We can see there are 1,059 records returned by this statement. I simply copied this entire statement and pasted it into a code window here.

So this is exactly the same, but I cleaned it up a little bit, just getting rid of all those dbo references before each of the tables. So it's a little easier to read, but that's all I did – pasted, got rid of all the dbos, re-executed. Same result set: 1,059.

Now the UNION statement would in essence retrieve another set of results, but I want them to come back in the same result set. So an example of when you might see this is simply when you've already determined the result set from one statement, then you've determined a result set from the other statement, and you just want to combine the results. So what I'm going to do is simply highlight all of this, copy this, and then I am just going to go down to my next line here. And I'm going to just Enter and hit a blank space and paste in the identical code.

So I've got two statements here. At this point, it doesn't make much sense to run them both, but what I'm going to do now is change the year. So I want to see the exact same information but for the next sales year. So I'm going to increase this to 1998 in both cases.

Now I can highlight just this portion, and I can execute that. So we'll do exactly that, and we'll Execute. And we'll see that there are 691 rows for that year. So we didn't do as well that year, but that's fine.

So now I might think, well you know what, let's just put all of these together and have a two-year total. So that is exactly what the UNION statement does. So on my blank line here, all I need to do is add in the word UNION.

So we got 1059 the first time. We got 691 the second time. So we should expect to see somewhere in the neighborhood of 1,700 rows when we UNION the two of them together. So let's test that out. Let's Execute, and there we see 1,748 rows. So it is similar in some respects to the CROSS JOIN.

But the fundamental difference is that the cross join takes all records from one table or at least all records from one result set and multiplies them by every record in another result set. **The UNION statement takes all records in the result set and adds them to all records in another result set.** So it does not multiply and give you every possible combination.

It simply says here's result set one stuck together with result set two. Now when you use the UNION statement, typically you want to use them on more or less identical statements – so the same columns. It doesn't absolutely have to be but it does need to be the same number.

You can't take a statement that has three columns and UNION it together with a statement that produces four. But they really should be, for the sake of, you know, making sensible queries, they usually should be something along these lines. Now maybe a more applicable real-world example might be selecting out each individual quarter – quarter 1, quarter 2, quarter 3 – and then to keep a running total of all of those quarters. You could UNION one and two together and produce a semi-annual total.

Then you could do the same thing at the end of year – UNION all four together and have an annual total. So that's just another example of combining results. That's...your UNION statement simply says, well, here is a running total – year-to-date sales; a few things along those lines. But at the end of the day, it simply pastes together result set one with result set two.

## **Manipulating Result Sets**

Now let's take a look at an example of ways that you can break down result sets in terms of comparisons and see how particular record sets are divided up, if you will. Now for this particular example, we're going to focus on two key words known as the INTERSECT and the EXCEPT statements.

And they are very similar to the results that you might be able to achieve using JOIN, particularly OUTER JOIN, because you're looking for records that don't necessarily have a matching value in their relationships. But it's a bit of a simpler statement.

Now I've gone back to the AdventureWorks database for this example because there is a good practical implementation of those concepts. So we're going to begin with just simply pulling out the products that we have from our Production table. And you can see there is the Production.Product table. Now I'll tell you right now that further down, we have a related table called WorkOrder. And you can imagine that on any particular work order, a customer would be wanting to see products.

So somebody creates a work order; there typically needs to be products involved with that work order. But of course, not every product is going to be on every work order. So we can see how they are divided up, how they compare, how they are broken down, if you will.

So let's try to clarify that by, first of all, just pulling out the products. So we're selecting only the ProductID from the Production table, and we can Execute that. And we see that there are 504 records. So I've taken note of that right here – 504 rows.

So I've already executed these, just to confirm. So now what we're going to do is break that down. I, for starters, want to know only the products that have work orders for them. And you can see I've already executed this and taken note of the total – there is 238.

But we will still run them. But we see the statement here. Instead of having to create a JOIN statement, all I have to do is SELECT that product from the Production table and INTERSECT it with the ProductID FROM the WorkOrders table. So I do need to know that there is a relationship there. The ProductID field exists in both tables.

And I can expand the Columns here just quickly. Let's have a look.

Under Columns, we see there is the primary key of ProductID in the Product table. But if I go down to the WorkOrder table, that's not going to be the primary key – it's the foreign key. So there we see FK.

WorkOrderID is the primary key of this table. So it's still a standard relationship. Rather than having to specify all of that JOIN statement, I can simplify quite a lot by simply specifying the keyword of INTERSECT. So what I see here is the Production table and the associated products, then those same products that are in the WorkOrders table.

So again, this translates into selecting only products that have work orders for them. So it's looking for those matching values on the ProductID. So we can run this, and we see that there are 238 rows. So these products have active work orders for them.

Then I can see – well let's see the flip side of that – select only the products that have no work orders for them. So in this case, rather than an INTERSECT, I specify an EXCEPT statement.

And that basically flips it around. This shows me products that have no work orders for them. And I get 266 rows back. I can Execute that and verify – 266. So these products have no work orders.

And if we scroll down a little further, there's the equation that I've taken note of. This is how those records are divided up.

Of the 504 total records, 238 have work orders, 266 do not. But it adds up to the same total.

That record set plus this record set gives me the total record set. So just to, sort of, double verify that, if you will, I can switch this around and select out the work orders that have no products. This is, kind of, looking for those orphan records that were referred to earlier. I want to see if there are any work orders in progress that don't have a product assigned to them.

And there really shouldn't be by the business rules of this environment – a work order must contain a product. So if I select this, I'm hoping to find zero rows. And I know I'm going to because I already got 238 plus 266 for a total of 504.

But if for some reason I only saw 265 come back, then I'd only see 503, and that would indicate that there is a record missing somewhere. So if I Execute this, I should see nothing and there we go.

So again, these are a little simpler ways to implement the outer joins and looking for those null values perhaps or for orphan records. But this is definitely a simpler statement than creating those outer joins and then specifying null criteria.

The INTERSECT, in essence, shows me only products that have work orders.

The EXCEPT shows me only products with no work orders.

And I can find those ones and maybe determine, well, how come these ones aren't being ordered? Let's kick up the advertising on those products, whatever.

But you still find those records that do not have related entries in their related tables.

### **Lesson Test**

Q1: What is the fundamental difference between CROSS JOIN and a UNION statement?

CROSS JOIN statement takes all the rows from one table and multiplies it by all the rows from another table.

UNION statement takes all the rows from one table and combines them with the rows from another table.

Q2: Given two tables, food having 7 rows, and company having 3 rows, how many rows would be listed in the result?

SELECT food.items\_name, food.item\_unit, company.company\_name, company.company\_city FROM foods CROSS JOIN company;

21

Q3: What is the INNER JOIN code for the screenshot?

Customers INNER JOIN Orders

ON Customers.CustomerID = Orders.CustomerID

Q4: What is the purpose of the statement shown?

SELECT ProductID

FROM Production.Product

INTERSECT

SELECT ProductID

FROM Production.WorkOrder;

Selects only the productID that have work orders

Q5: You can create a LEFT OUTER JOIN by selecting all the rows from the table on the left:

Right-click Diamond in View,

Select All Rows from Customers

-----------------------------------------------------------------------------------------

# **Working with Data**

## **Inserting Data**

So we've now seen a few different ways how you can retrieve data. Now we are going to take a look at how you can actually get that data into a table. So this is the use of the INSERT INTO statement.

And just before we get too far into it, you do need to know a little bit about the design of the table before you start issuing INSERT statements. In particular, you need to make sure that your data types are correct, that you're entering the right amount of information in terms of the size of any particular field, and also if there are any automatically generated yields or default values.

So I have, right here, in my comment that we are going to add a new record using the INSERT statement, but to take note that the Customer\_ID and the State\_Province field have automatically entered values. Now just to clarify that as well, in fact, it's only the ID that has an official automatic entry. The State\_Province field has a default value. So let's just go back and quickly look at the design of the table.

So I've made a few changes since we first created this table – most notably, just some larger fields. So that is one thing to take note of. You need to make sure that any given field is large enough to contain all of the information. And in fact, when I had these just at their defaults – when you assign the character data type, it defaults to only 10 characters – and my Street\_Address was still only left at 10. So that's probably not enough. That's generally not large enough to accommodate all of the characters and the person's street address. So I increased it.

Now this is probably a bit too far but it's fine for our sample. But I increased it to a larger size to accommodate that and the same thing for the City as well. So for starters, make sure you have enough space. But now our Customer\_ID field is an integer value, and if we look at its Properties down here, it is also an Identity field. So it will automatically seed an increment by one, each time a new record is entered. So in other words, since the database engine is going to insert this value automatically, you don't have to.

The other field is the State\_Province field here. We specified a default value on that one, which means you can leave it blank if you want to and you can allow the database engine to insert the value automatically. Now you can change it if you want – that is perfectly fine.

But if it is going to be the default, then you can basically skip over it, if you will. So again, just a little bit about the Properties in terms of the table design. So we'll go back now to the statement, and first thing we'll do is simply select out the record, which I've already done. You see, there are currently no records. So to create a new one, our syntax is the INSERT INTO statement. So that's the first part – INSERT INTO.

Then you have to specify the name of the table into which you are assigning this new record. And then, you see, what you've to specify is a list of each of the columns in that table. So you can save yourself a lot of time here by just, of course, expanding this and seeing the names of the columns.

But if you just start typing in the first few characters – and I can demonstrate that; I'll just start typing in here, "f." And you see that it pulls up all of the fields that start with any given character, and you can just select the one that you want. This avoids typos and helps to speed things along. So it doesn't take very long to create the statements. So definitely take advantage of that auto-complete. But you can see what I've done here. You have to enclose this in parentheses, for starters, but I did not include the ID field. I don't have to because it is automatically generated. So that's one of the things you can do as well. When you know that there is an automatically generated field, simply don't include it in the list of columns.

So I start with First\_Name. Then I went through all the remaining columns – Last\_Name, Street\_Address, City, Province, and so on. So you always have to tell it which columns you are entering values in. From that point, you use the key word, Values. Again, start with a parenthesis and then you specify the value for each associated field. So Mike equates to First\_Name, Smith equates to Last\_Name, 123 Any Street to Street\_Address, and so on and so on.

And then, finally, when we get to the State\_Province, if you want to insert the default value, you can simply use the keyword DEFAULT. Then we have our Zip\_Postal\_Code and a Phone\_Number. So these are obviously just made up values. But those are the two things you can do with the INSERT statement – simply ignore the first one that is automatically generated, which again is the ID, and then use the key word of DEFAULT if there is a default value. Now as mentioned, if you want to overwrite it, you absolutely could. You want to put a different value in – go ahead, that's fine. But this will automatically insert the default value. And do take note that each of these entries needs the single quotes around them to qualify the start and the end of the string of text. And it also helps to identify that there are spaces, so it knows to actually insert a space here as opposed to cramming it all together into one single string of text. So once you're done, you can close off your parentheses and make sure that's done on both. You see, there is a closed parenthesis there, and you're pretty much ready to go. So we can Execute this INSERT statement, and then we can see that one row has been affected. And then we can select and verify that the customer has been created, and there we go.

Now, quickly, the reason why I got Customer\_ID number 2 – this is not their ID; this is simply the first record. The ID of 2 went in simply because I tested this statement before I started the recording, and I created customer number 1. I then deleted that record and ran it again. Even though customer 1 was deleted, the next record will be customer number 2. And that is simply because unique identity values are never reused. So if I were to delete again and re-execute, this same record would go in but it will be customer number 3. So there are the values that are specified – Mike, Smith, 123 Any Street, in Some City. But there is my DEFAULT keyword inserting the default value. There is my postal code, there is my phone number; this record has been created.

Now from a practical perspective, end users are certainly not going to be issuing INSERT INTO statements in SQL Server Management Studio. They would be using a custom-designed front-end application or a web interface to create these records. But what it would be doing – it would give them all of the fields to enter the values in. And then they will click something like a Submit button. That would call what's known as a stored procedure, which, in essence, would contain this exact code. So it would simply send the values that they've entered into the procedure. It would execute this code and create the record. But if you are a database administrator, you absolutely could create new records in this fashion. But in most cases, it is a front-end interface that users are seeing. They simply enter the values, click on Submit or Save or Enter or anything along those lines. And it sends that information into a saved INSERT statement. Either way, the record gets created. That is how you enter in new records using your INSERT INTO statement.

## **Bulk Inserts**

Sometimes when you are looking to insert new records into a table, you might already have a source of the records. And one of the most common examples of this is somebody has already done some kind of an export, possibly from a completely separate application, or they have actually manually typed up some kind of a source file.

But regardless of how you obtain the information, it's already packaged up into some kind of a file. With that, you can use that entire file as a source of information to import all of the records into an existing table. And this is what's known as the bulk import because there are a lot of records. Now this is a fairly large topic, there are a number of different ways by which you can accomplish this.

So what I've done is to pull up a website here that gives you a lot more information than we can cover in a few minutes. But specifically, I also chose this one because it talks about keeping identity values, which is very common in a lot of tables. So the problem might be that the source of records that you have in your file may not include something like an ID field or you might want to simply discard what might be there because, maybe, it's incorrect as per what your column definition specifies.

So again, if my ID field, my identity values are simply 1, 2, 3, 4, 5, 6, 7, 8, 9, the records in the file that I've obtained may not use 1, 2, 3, 4, 5, 6, 7, 8, 9 or there might be duplicates or there just might be any kind of problem that would prevent those records from coming in correctly.

So keeping the identity values can be done with this KEEPIDENTITY keyword. So we'll get back to that in a moment. But what I've done here for the demonstration is to pretty much follow exactly what is on this page here.

For starters, what this will get you to do is, in fact, to create a target table that's empty so that you have some place to import the records, and then it also gives the instructions as to how to create the export file. So what it does is simply copies some records from one table and then creates a new table with no records. Then you can take those copied records and import them into the newly created target. So that sounds like a lot, but I'll go through each of the steps here so you can see, for starters, that there is some code to use to create a target table.

Now just a couple of quick things - this uses the AdventureWorks database. And if you have at least version 2012 of SQL Server, then what you will likely find is that your AdventureWorks database is called AdventureWorks2012. So just take note of that.

Now I've already done some of these preliminary steps, and here is the first one of those to actually export some of those existing records and then to create the blank table that you can use to import them. So again, I'll just quickly go through that by coming back to the web page here.

The first step creates, in essence, a duplicate table with no records. So you can see what it's doing is pulling records out of the HumanResources table and then creating one called myDepartment instead of just Department. So if you will, this is the real table - HumanResources.Department.

This is the one that we're going to make up to use as a target; so HumanResources.myDepartment. So you can just hit Copy here and this will copy that code, and you can then simply paste it into your SQL Server Management Studio. And I did exactly that.

There is the code right there, and the only change I made was to add the 2012.

Otherwise, it doesn't work because it won't find the correct database name. So there you see just some comments that tell you what is happening here - we're creating an empty table to use as the import target.

So we SELECT everything (\*) FROM the HumanResources.Department table INTO the HumanResources.myDepartment table. So that's the dummy table, if you will.

But look at the criteria: WHERE 1=0. That's just something that says let's evaluate all the records. If one equals zero, import it. Well, one never equals zero. So why do we use this? To create a blank table.

No records get imported, but what this accomplishes is it creates the exact same structure of the table. So if we look at the Tables now, we'll go to HumanResources. You see, there is Department. When this code is executed, which we can just do...sorry, my mistake; I already ran it. So there's already one here. So if we just Refresh the Tables here, we now see HumanResources.myDepartment. But it's empty. So let's SELECT the records and we see nothing. So there we go. So we'll leave that open because we'll run this again later. So again, the idea behind that 1=0 is so that no records will actually be imported.

But it creates the structure for us. So now we have a place to import records in, and we know it has the same structure as the other real Department table. Now the next part that you see here is to simply run some utilities that will export existing data out to a file. That gives us something to use to import, and that's these two statements right here. This bcp utility is what's known as the Bulk Copy Program, and all it does is it creates an output file. And then this is simply the format of that file.

So again, don't worry too much about the syntax at this point. Copy this code and then just run that from a Command Prompt, right from the root of your C Drive. So you can see, I did exactly that. I pasted it right from the root of my C Drive. Again make sure you add in the 2012 for the database name; but otherwise that code is exactly copied. I ran that. Then I ran the format one, which again just formats the file correctly. And I did the same thing. I just added in the 2012, and both of those ran; and you can see created for me - these two files (DAT & XML). So those now are the format file and the data file that I can use to import records into the new table.

So the last step is to simply run the BULK INSERT statement right there, which will keep the identity values. And that is specified with that code right there. So again, I just Copy this code, pasted it into my SQL Server Management Studio. And again, I just got rid of the USE AdventureWorks, and just specified the database here because it's AdventureWorks2012.

I have not yet run this code. So there you see - it is the BULK INSERT statement. It's going into the myDepartment table instead of the Department table. There is the source file - the data file itself. There is the KEEPIDENTITY key word, which will make sure that any identity fields are kept, and there is the FORMATFILE. And again, that's just simply these two files, right here, that we just created using bcp.

Execute this code; 16 row(s) affected.

So if we go back now to our SELECT statement where previously there was nothing; let's re-execute. There they are - 16 records just like that. And I did not have to type in any of that, and the ID fields - the identity fields - were preserved. So I see 1 through 16. And basically, that is how you can very rapidly put in possibly hundreds of thousands, maybe even millions of rows, into a table with just a few statements.

So again, there's a few steps that you might not normally do there. In a practical environment, you probably would not have to create the table. But you typically do have to have something that will create the file that you use as your input source. Once you have that, you can run your BULK INSERT; pull all of those records in at one time.

## **Updating Data**

Some of the records in your tables might get entered and might never change. They are very static in nature. But at the same time, you may have records that will change all the time – very dynamic records.

So obviously, you need a way to implement those changes. And in SQL Server, it is the use of the UPDATE statement. Now I've gone back to my NORTHWND database for this example, and I'm going to work on the Products and their UnitPrice because, of course, prices change.

So to begin, I've pulled out all of the Products and their associated UnitPrice from the Products table, and I've already executed that. And here are my results. So 77 rows, and they all have the associated price there.

But just for the sake of easy math, we look at record number 3 here – it's $10. So let's just imagine that we want to increase the price of this particular product. So the next thing I did was just to filter out that particular one – just so we can focus on it. So I'll Execute that.

And there we see, now only that product and there is its unit price. So it's a fairly simple statement to change that, although it's not quite what you might think in terms of what you need to specify in the code.

There is the UPDATE Statement. But instead of directly updating that particular value in that field, you actually specify to update the table in which it resides. So this is the first part, right here, UPDATE Products.

So you'll actually refer to the table name when you issue the statement itself. Then you further clarify it by using the SET statement. And in this case, you Set the existing field of UnitPrice equal to whatever it is you want it to be. So you can see in my comments here, I'm going to change the unit price for that product by applying a 10% increase. So that's what this is here.

It is taking the existing value of UnitPrice and simply multiplying it by 1.1 – that's a 10% increase. But here is the key, very important thing to remember. You definitely want to specify the same criteria that we did up here.

If I were to leave this blank or accidentally run this code without highlighting it – let's say I only did that –then what I'm now telling this to do is to update the unit price by 10% for every product in the table.

So you definitely want to be careful when you are issuing UPDATE statements to ensure that you specify what criteria are going to control the records that are going to be updated. So it is very important to make sure you have your WHERE clause in.

Again stress that, if you left this WHERE out, you would update the price for every product in this table. So we don't want to do that.

So we'll highlight the entire block of code and apply a 10% increase to where the ProductID is equal to number 3. So let's Execute that. One row has been affected.

And now we should be able to reselect out that same value to verify, and it should now be $11 – 10% higher – let's Execute, and there it is – $11. So that ensures that you selectively change only those records that should be changed, and that is definitely something that you want to ensure.

Any kind of statement that can make a broad scope to change, you just need to be very careful. Make sure that you are selecting out only the appropriate records first. So get in the habit of zooming in or focusing in on that which you want to change first. Then apply your updates, then go back and verify that the update succeeded. And that way, you can implement the changes as per your requirements.

## **Deleting Data**

Similar to updating records, there may be instances where you need to delete records. And this is particularly common while you are testing and developing.

It should be noted that once a database is in production, deleting is actually somewhat rare. A lot of organisations like to keep the fact that the record was created at some point.

Rather than just discard it, they might set its status as something inactive, for example, as opposed to entirely removing it. But when you need to, you certainly can get rid of records from your tables. And you do so with the use of the DELETE statement.

Now similar to the UPDATE statement as well, the DELETE statement actually operates on the table level. And then you specify which records you want to remove. So again, it's a good habit to get into – just to pull all of the records back first, just so you know that you're dealing with the correct information. So there is a simple Select \* from my Customers\_Table back in my Sample\_DB, which, of course, has no meaningful data. So I added a few records. You can see now, I have a few more customers here. So we'll simply look at how you can remove one.

So there's my SELECT statement. The next one is to delete just a single record. And the key component here is to ensure that you specify a WHERE clause.

If you do not, you would delete all of the records from the table. And you can see, I do have that down here. But let's look at the first example here, to DELETE from the Customers\_Table where the Customer\_ID equals 5.

Now you could, for the sake of knowing it, specify really any piece of criteria as long as it was appropriate to that particular field and not something that would wipe out every single record. So for example, if I wanted to delete Joe Prince here, I could, in fact, say DELETE where the last name equals Prince. In this example, there was only one record that satisfies that criteria.

But clearly, in a production database, that might satisfy possibly hundreds of records. And if that was the case, it would delete everyone whose last name happened to be Prince.

The same thing would apply if I were to use the state or the province, for example, here. Because they are all NY, if I were to use that as criteria, it would delete all of them. So another very good habit to get into when deleting is to **use the unique value** as the criteria.

Now there may be instances where you do want to use something else, strictly because of the fact that it does have duplicates. Maybe you've lost a sales territory, and you want to delete all the records for that territory.

So you absolutely could use something like the city or the state and province in that case. But that's much more severe and much more difficult to recover from. So always just make sure it's an appropriate instance.

But, by and large, you would use something that uniquely specifies a particular record. So in this case, I'm deleting this particular customer number 5. So let's Execute that statement, and one row should have been affected because that's the only one whose Customer\_ID was 5.

Let's go back and view, and we should no longer see customer number 5. There we go – that customer is gone.

Now there is the syntax if you wanted to DELETE every record from the table. That would keep the table. And as mentioned earlier, this is something that is fairly common when you're testing and developing – you would probably put in some invalid or inaccurate data just to test things out.

But the table definition, its properties are fine. So you want to keep the table definition, but you want to purge out the data, then you could just issue this statement right here – DELETE from Customers\_Table. With no criteria, that will remove all records. Three rows have been affected. Now let's put our SELECT statement back in here, and we see that the records are gone.

So again, you definitely want to be careful. But if you do want to just blow away the whole table, you can specify that statement with no criteria.

But in most cases, you will specify a unique value to selectively remove just that record. It also should be noted that you can only delete the entire record. You cannot use the DELETE statement to remove a particular field from the record – that's just not possible. You would be doing an UPDATE Statement then if you did want to remove a value. Let's just say, you had a fax number and then the person no longer has a fax number. Well you would not use the DELETE statement. You would update the record to simply insert a null value where there used to be a value. So that's an update. DELETE removes the entire record and only the entire record or every record that satisfies your criteria. Always be sure that you are supplying the correct criteria before you execute your DELETE statement. Otherwise, you might have to go back to a backup to recover that information. So you definitely want to be careful with it. But when appropriate, by all means, you can issue that statement to remove your records.

## **Transactions**

We're going to take a closer look now at how referential ensures that your data remains intact and accurate and correct. Now as the name indicates, referential integrity, it refers to the integrity of how valid your data is.

Now I've gone back to my sample database for this just because it's very easy for me to add and create new tables and records and to delete them as well. So I only have a few customers here, and what I did was to create an additional table called an Orders table.

And it's very simple, the only Columns are the OrderID as a primary key for this table, and then you see a CustomerID as the foreign key. Now I will show you how this relationship was set up, but I did not create the column as a foreign key. All I did was simply call it CustomerID, make it an integer data type, and that's it. I didn't make any other changes to that field at all.

The relationship between the two was set up in what I like to use, the Database Diagram method and just because it's nice and visual. So I'm just going to collapse all this. And you see here is the option for Database Diagrams.

You can expand this, and I created one simply called Relationships. Now I'll show you how to do that in a moment. But if I just double-click on this, there you see is the relationship. The Customer\_ID is my primary key in the Customers\_Table. The OrderID is the primary key in the Orders table to uniquely identify each order, but we also need to know about that the order is which customer owns it. So this is the foreign key. Now to create this, all you have to do is drag your Customer\_ID from this table and drop it over here in the Orders table, and that pulls up the relationship creation window wherein you specify the relationship. So you see the Customers\_Table Customer\_ID is related to foreign key CustomerID. Click on OK. There is your relationship. Now there are some other properties about the foreign key relationship that we'll see in a little bit, but that's how you create the relationship.

And there are other ways to do it, but that's one of the ways. Now with respect to those other properties, this is the window here where you can modify them, but I'm going to come back to that. So I'm just going to Cancel this for the time being. And again, that's simply how you create the relationship between those two tables.

So now that I have that, I have my customers here and there's currently only 4. I tested all this out earlier. Customer\_ID 4 has already been deleted. But as I said that was just a test, and I'm going to use Customer\_ID number 5 in this current demonstration to show how we can ensure that referential integrity will keep the data intact. So again there's just my customers.

Now what you see here is some code that creates some orders. So I inserted a few orders, and I just assigned an OrderID and then the customer that owns that order. So we can see those right there. I'll just re-execute this, and there are my orders. So 1, 2, 3, 4 are the orderIDs simply to uniquely identify the order. Then the customer is the owner of that order. So here you see is an example, and you absolutely can have duplicate values in a foreign key. Customer number 1 owns both order number 1 and order number 2, Customer number 2 owns order 3, Customer 5 owns order 4. So what's happening right now is that referential integrity is protecting this customer. Why? Because they have an order.

**So what I cannot do at this point is to delete this customer because that would leave this order with no owner.** So here you see is some code that attempts to delete a customer with orders. So there is number 4, and again that's why number 4 is missing because I did this just as a test. So now I'm going to change it to 5 because Customer\_ID 5 is still right there. So note that our comment here says this would violate referential integrity. So I'm going to change this to number 5, and I'm going to try to re-execute this code, and watch what happens.

I get an error. The DELETE statement conflicted with the REFERENCE constraint. That's the foreign key Orders\_Customers\_Table that I created. Says you cannot do it because, in essence, they have existing orders and the statement, you see, has been terminated. So let's – even though I attempted to run this – let's go back and reselect the customers. Customer 5 is alive and well. So that's referential integrity at work. It is protecting the customer because they have an order. Now what I have here is a stored procedure, okay. This, you see in the comment, does not violate referential integrity. Well why is that? Here's the code.

The procedure accepts a CustomerID as an input value. So you simply call the procedure and you tell it which customer you want to delete. Now there is a transaction TRAN being declared here, and I'll come back to that in a minute as well, but let's just look at the code here first. What happens is that it deletes the Orders first where the CustomerID is equal to whatever is provided by the calling user. Once the order is removed, then it deletes the customer. This works because it removes the related records first. The related record, the order, can absolutely be deleted. It doesn't matter if the customer is still there or not. Somebody might call up and simply say I don't want that order any more, fine, you absolutely can delete the order. But you cannot delete the customer while they have active orders. Now the reason this is incompetent to transactions is because it allows me to do both units of code at the same time in a single execution. Delete From the Orders table and Delete from the Customers\_Table, all in one transaction. And the other benefit of a transaction is that should this fail, if for some reason it's still violating any kind of other integrity rule that, maybe, I don't know about, transactions must either complete entirely or not at all.

So if it succeeds, the transaction is committed to the database. It's saved. Everything will go in or come out exactly as I'm specifying here. If it fails anywhere for any reason, the entire transaction is rolled back. It's undone, it's like it never happened. So that's another benefit of transactions and integrity. You can simply protect multiple records and multiple statements can all be executed entirely as a single unit or not at all. So this deletes the order first, then the customer. So let's go back and, we'll see now, we'll execute the stored procedure and again there was my test, which deleted number 4. So now let's change to 5 and we'll re-execute. Two rows affected, not just one. It deleted the customer and their orders. Let's go back and verify. So customer 5 is still there. Let's re-execute; Customer 5 is gone. Let's go back to the Orders table. Select from the Orders – the order 4 – that customer is gone. So there we see referential integrity at work. It is absolutely fine to delete the order first, then delete the customer. But if you attempt to delete the customer while the order exists, referential integrity prevents that from happening and ensures the validity of your data.

### **Lesson Test**

Q1: What is a best practice when using the DELETE statement?

Use a SELECT statement before DELETE to ensure you have the right data set.

Use a WHERE clause to ensure you are selecting the right data set.

Q2: By default, how does the BULK INSERT command handle imported identity values?

Ignores values in the identity column on the data file being imported.

Assigns new unique values based on the seed and increment values.

Q3: What is important when adding data into a table?

The field data type matches the data type being added to a field.

The number of characters defined for a particular field is not exceeded.

Q4: When using the UPDATE statement, what is one of the most important things to keep in mind?

The specific data you want to update is defined.

Q5: From the Database Diagrams screen, how do you create a Foreign Key in the Orders table?

Drag and drop the Primary Key into the desired field in the Orders table.

-----------------------------------------------------------------------------------------

# **Practice: DML Statements**

# **Exercise: DML Statements**

Okay. Time for an exercise now. And we will ask you to determine the SQL commands that you would input into SQL Server Management Studio to:

Return all rows from an Employees table where the Joining\_Date is prior to 02/12/2015, and their Status is Contractor or Developer and their Salary is between 60,000 and 80,000.

**SELECT \* FROM Employees**

**WHERE Joining\_Date < ‘2015-12-02’**

**AND Status = ‘Contractor’**

**OR Status = ‘Developer’**

**AND Salary BETWEEN 60000 AND 80000**

Our first one was to return all rows from an Employees table where Joining\_Date is prior to 02/12/2015 and Status is Contractor or Developer and Salary is between 60,000 and 80,000. So that statement would look something like this. For starters, returning all rows means that you can simply issue a Select \* statement from the Employees table. That will return every column. But we do still want to filter the results. So we have some criteria here and the first one is the Joining\_Date being prior to 02/12/2015. So we can see the criteria there is where Joining\_date is less than, and in this case it's just less than, not less than or equal to because it actually said prior to that date, so not inclusive of 02/12/2015. So where Joining\_date is less than that date and then we see an "and" statement because it did specify that it wants the date and the Status being contractor, but then we see "or" developer. So we have our first "and" after the date in the criteria.

Then in terms of the status, you specify the Status is equal to contractor or the Status is equal to developer. So the "and" condition means that both the date must be met and at least one of those two statuses. That's the "or." If we remove the date temporarily and simply look at status, it could be either contractors or developers. Either one would satisfy the criteria, but with that, you would then have to combine the Joining\_date. It must also be prior to 02/12/2015. Then we see the Salary included and this also specifies an "and" condition. So of all of the results that came back that were either contractor or developer and hired prior to 02/12/2015, there must also be a Salary between 60,000 and 80,000. So again, we see the "and" condition, so that one must be met. So the only instance that we see "or" here is with the Status. It can be either one of those, but the hire date must be prior to 02/12/2015 and the salary must be between 60,000 and 80,000. Of those, either contractors or developers would be returned.

Then update the email of a customer from the Customers table with the CustomerID of 10 to [Rick\_Tallon65@hotmail.com](mailto:Rick_Tallon65@hotmail.com)

**UPDATE Customers**

**SET email = ‘**[**Rick\_Tallon65@hotmail.com**](mailto:Rick_Tallon65@hotmail.com)**’**

**WHERE CustomerID = 10**

Now for updating the customer's email, we were wanting to change their address to Rick\_Tallon65@hotmail.com where the CustomerID was equal to 10. So this is our update statement, and this one is fairly simple, but you have to remember here is that you update the table not directly on the particular column. We update the Customers table, but then we explicitly Set the email column equal to the desired value. And the criteria is specified where CustomerID equals 10. And you have to be careful with Update statements because if you forgot that criteria, you would update every customer with the Rick\_Tallon65 e-mail address. So you always need to make sure you are focusing in on the correct record. Then you use your Set statement to manipulate that particular column.

And then finally, to delete all rows from the Employees table with Leaving\_Date set to 12/31/2013 or prior to it.

**DELETE FROM Employees**

**WHERE Leaving\_Date <= ‘2013-12-31’**

And finally, deleting all rows from the Employees table with a Leaving\_Date set to 12/31/2013 or prior to it. Now again with the Delete statement, you're operating on the table, not any particular column even though we are referring to the column of Leaving\_Date. That is simply our criteria. So we delete from the Employees table where the Leaving\_Date is less than or equal to and that satisfies the 12/31/2013 or prior to it. In other words, that is inclusive of that date and then anything else that happens to be prior to it as well. So that's the less than or equal to that date that includes that date and all days that are prior. And again, you definitely have to be careful when issuing Delete statements and ensure that the criteria is correct because if you forgot the criteria or it was simply wrong, you could end up deleting every record from the table. So there are a couple of examples of a Select statement, an Update statement and a Delete statement.

-----------------------------------------------------------------------------------------