Course Overview

Course Overview

Hi everyone. My name is Xavier Morera, and welcome to my course, T-SQL Data Manipulation Playbook. Knowing how to manipulate data using T-SQL statements is one of the most basic skills required to work with a Microsoft SQL Server database. Just think about it. Querying data might be very common, but you will get nowhere without knowing how to insert, update, and delete records. And that's why we're here. Some of the major topics that we will cover include understanding the statements from T-SQL's data manipulation language, or DML, about how to add data using INSERT be one record at a time or bulk INSERT, how to modify data with UDPATE, then how to remove data from the database using DELETE and TRUNCATE, how to create and work with transactions, which are integral part of T- SQL or Transact-SQL, and finally, some other related and more advanced topics, which include OUPUT, MERGE, and using DML statements with sub queries. By the end of this course, you will know and understand how to add, modify, and delete data using T-SQL statements. Before beginning the course, you should be familiar with the basics of Microsoft SQL Server and working knowledge of SQL. I hope you'll join me on this journey to learn how to work with data using T-SQL's data manipulation language in the T-SQL Data Manipulation Playbook, at Pluralsight.

Working with the T-SQL Data Manipulation Language

Version Check

Working with the T-SQL Data Manipulation Language

Hi. My name is Xavier Morera, and welcome to this course, the one that helps you work with the T-SQL Data Manipulation Language. We humans have always had a need to count and organize our things. We have kept records using everything from simple lines in a cave wall, for example, to know how many animals we had. As time went by, we evolved to using somewhat more sophisticated tools like an abacus, which was a groundbreaking tool at the time. It allowed to count quickly and perform operations like never before, but sometimes it was also necessary to log that information, which required a somewhat more complex skill, writing. It all worked quite well as long as you managed a small amount of data and had the time to perform all necessary calculations. But as the amount of information grew, it was no longer possible to manage all of this data using conventional methods, and eventually we ended up with this little thing, or maybe not so little, that we call databases. And jumping quite a bit ahead on some important breakthroughs in history, I just want to point out a few important events that are relevant for us and this training. The invention of the concept of relational databases can be traced back to 1970 when an English computer scientist named Edgar Frank Codd, or E.F. Codd, published a paper, A Relational Model of Data for Large Shared Data Banks, while working at IBM. It is worth mentioning though that this is not the invention of the database. It is the invention of relational databases, which go a bit beyond than just storing and counting data. His work was brilliant. It basically changed the course of history. Fun fact. In the 1960's, American Airlines already used a database to help its reservation system. I truly hope that this code is not still in production though. Anyway, fast forward a couple of years, and we get to the creation of Microsoft in 1975, and then a few more years and SQL, or S-Q-L as some people say, became an ANSI standard. It is the most widely used language, but working with data. Knowing SQL is really, really important even beyond relational databases, but it was until 1988 when Microsoft SQL Server was created. Here's an excerpt from the press release on January 16, 1988. Microsoft and Ashton- Tate announce Microsoft SQL Server, a relational database server software product for local area networks, LANs, based on a relational database management system licensed from Sybase. And that's how we got T-SQL, which is Microsoft SQL Server implementation. It is its own dialect of SQL. It is ANSI SQL plus a few extensions. Sybase was also involved. A few years later, Microsoft released another product, Microsoft Access, which is another database management system that could be used to develop applications. And so many new versions have been released over time, starting with version 1 and with new releases every couple of years. And for this training, we will use SQL Server 2017 focusing on one specific sub language of T-SQL, the DML.

Understanding and Differentiating the T-SQL Sub-languages

The first thing that you need to take into account is that there is an official specification of the SQL language known as ANSI SQL, but that we, as users of Microsoft SQL Server, use a particular implementation, T-SQL. There are many other implementations, but let's not get into those details. We're going to use T-SQL, which in turn is made up of multiple sub languages, which are the different sub languages of T-SQL. And let me be clear that this classification is not written in stone. Some people may disagree on a few points, but a high level, we have DML, which stands for Data Manipulation Language. As the name implies, it helps load, modify, and delete data, manipulate data. We will also cover transactions, which some classify under TCL, or Transactional Control Language, but I will cover them with DML as well. We will get into the details shortly. It is the main topic of this course. But before, I just want to make sure that you know what are the other T-SQL sub languages that you will run into. The next one is very special, DQL, or Data Query Language. I say it is special as it has only one statement, SELECT, which is the command that will be used the most when working with a database, and it's used to retrieve data. I have included it in the DQL sub language, although some include SELECT within DML, but we won't. We will treat it as its own sub language, and hence, I will cover briefly only what you need to know about SELECT to work with DML. Then we have DDL or Data Definition Language. Sample DDL statements include CREATE, ALTER, DROP. We use them to do things like create a table in a database or to remove a table. And finally, DCL, or Data Control Language. It is used to do things like create roles or provide permissions in a database. Some statements include GRANT and REVOKE. And now it is time to dig deeper into DML.

What Is the T-SQL Data Manipulation Language?

When working with T-SQL's Data Manipulation Language, there are three statements that you will primarily use. First, INSERT, which as the name implies it is used to add records to a table. You can insert values into all columns within your table. Well, there are a few edge scenarios, but they are quite advanced, so we won't talk about them now. Or you can provide only a subset of columns, and based on your table's definition, a null value may be added or a default value can be used. Also there are a few other scenarios. For example, when you have required values or when there is a constraint. Then UPDATE, which it clearly states, it helps modify existing records in a table. Just like INSERT, there may be some constraints and limitations on what you can and cannot do with UPDATE. And the third one, DELETE, which helps you remove records from a database. There's another command that we will also see, TRUNCATE, which serves a similar function. All of these commands will be covered in detail in the upcoming modules. And now it is time to prepare the database that we need for this training and refresh our memory on a couple of statements from other sub languages that we need to know to properly work with when using the Data Manipulation Language.

The Database for This Training: StackOverflow or StackExchange

For this training, I have selected data that is quite relevant to many of us in our day to day as a developer. I am talking about Stack Overflow, the Q&A site for developers by developers, the place where we go to or end up in most of the time that we're looking for a solution to one of our problems. So, are we going to use all of the data from Stack Overflow, and how do we get it? First of all, no. Stack Overflow has a lot of data, which would require a very large database, but the good news is that we can use a smaller stack exchange site, which has the same schema. And second, to get the data, we can get one of their data dumps, which are publically provided in https:// archive.org /details/stackexchange. For this training, I will use the machine learning stack exchange data dump. Now, let me show you a demo of how to get your database up and running. There are many ways of creating the database for this training. It all starts by downloading the data. It is currently provided as a set of XML files, so some work needs to be done to get it into something that we can load into SQL Server. Well, I did that already and created a database of reasonable size for us to work with, and I provide two ways for you to get this database. The first is via SQL script and the second is by restoring a backup. Let me begin by going to my machine and looking for SQL Server Management Studio, which I assume you already know it is the integrated environment for managing SQL infrastructure be it locally at Datacenter or in Microsoft Azure. I have SQL Server 2017 installed locally, so I will just connect. Here is my database server. Let me expand to confirm that I did not yet have the database for this training created. Indeed, I don't. I will close the Object Explorer, and for menu, I will select File, Open, File. In the exercise files that I provide as part of this course, within the database backup folder, you will find a zip and a backup file. If you expand the zip file, you will find a SQL script. It is about 230 megabytes, so if you're a little bit constrained in terms of your machine resources, I recommend to wait about 2 minutes to show you the other way of creating the database. It is faster and less resource intensive. Else, open the script file. Here are the SQL statements that we will execute to create this database right here with its corresponding log file. Please make sure this path exists including the drive and the folder. Now I will rearrange this toolbar at the top and click on Execute. One by one, statements are executed. It will take a while to finish. It all depends on your machine. I will speed it up a bit until I get the message that I was waiting for, Query executed successfully. I can now close this script, then go back to the Object Explorer, expand Databases, and I can see my tables right there. I'll check the first 1, 000 rows. And we have data nicely organized into rows and columns. There's data of different types. Some columns are required and some have no values. Let's count to see how many records this table has. I can do this using a simple statement, SELECT count Id FROM tsql-dml.dbo .Posts, a tad over 35, 000 records. Good, the database is ready, but that was only method number one. There is another way, which is quicker and probably easier. Why didn't I start with this one? Well, because I wanted to execute some SQL statements to practice a little bit. Let me just delete the previous database. This will only take a few seconds. Make sure that you close existing connections. And now, it's gone. Now, right-click on Databases, select Restore Database, and as Source, we want to select the Device. Click on the ellipsis, the Backup media type is File, and select Add. Navigate to the folder where you have the tsql- dml.bak file, and click on OK. As you can see, it is a Full database backup. Click on OK, and in just a few seconds, we get a lovely message, Database 'tsql-dml' restored successfully. I will click on OK, and from the Object Explorer, I can expand this database, expand the tables, and again Select Top 1000 Rows. We have our data. I can again count and confirm that I have the same number of rows. We have successfully restored a database from a backup. And either method you pick, you will end up with the same database. Now we are ready to refresh our memory on topics that are prerequisites to work with the DML. Let's continue.

What You Need to Know About SELECT for DML

The first statement that I want to make sure we know quite well, as we need it for DML, is the SELECT statement, which as mentioned, it is so important that it has a category of its own, the DQL, or Data Query Language. As stated in docs.microsoft .com, the SELECT statement retrieves rows from the database and enables the selection of one or many rows or columns from one or many tables in SQL Server. Let's explore its syntax. A complete SELECT statement could be divided into several clauses. The most commonly used being SELECT select_list INTO new_table FROM table_source WHERE search_condition GROUP BY group_by_expression HAVING search_condition ORDER BY order_expression. There are others, but for now, let me expand on each one of these. SELECT select_list is used to specify which columns will be retrieved by the query. If you use a star, all columns are returned. Then, INTO new_table is used to insert the resulting rows of a query into a new table. At least for me, it is not something that I use on a daily basis. Then, FROM table_source, which specifies which table or tables are going to be used in the SELECT statement to retrieve the data, it is almost always required except for when no table columns are listed and instead you have literals, variables, or arithmetic expressions. Next, the WHERE search_condition, which is used to filter which rows should be returned by the query. WHERE is really important, and so I will cover it also in the next clip. Moving on, we have GROUP BY with an expression, which divides the query results into groups of rows, for example, to perform an aggregation. Then, HAVING in a search_condition, which is typically used with GROUP BY to add a condition to filter a group or aggregate. It is like a WHERE, but instead of rows, it is used for filtering groups. Then ORDER BY an expression, which is used to sort data over one or many columns or expressions. You can sort in ascending or descending order. Also, please bear in mind that UNION, EXCEPT, and INTERSECT operators can be used between queries to combine or prepare results into one result set, and you can also limit the number of results using TOP. SELECT, FROM, and WHERE are probably the three more common clauses that you will use. Now, let me show you with a demo. Using the SELECT statement. From SQL Server Management Studio, please click on New Query. If you remember, in the previous demo, the T-SQL DML database was already selected as we initiated our query from the Object Explorer, but in this case, as we just clicked on the button from the toolbar, the default database master was selected. We need to make sure that T-SQL DML is specified as the database where we will run our queries. Good, we are ready. Now type the following query, SELECT * FROM Posts. You can type the entire text or you can take advantage of IntelliSense. Now, click on the Execute button. You can also use the shortcut key, F5. Depending on your system and a few other variables, especially if you just restored the database, execution may take some time. If that is the case, you can let it run for a little bit, and if you wish, you can stop execution by clicking on cancel executing query. For our intents and purposes, this works fine as we just want to output a few rows. Just a side note, you may have noticed that I do not use a semicolon at the end of the statement. Well, in other SQL implementations, you use the semicolon. With T-SQL, it is not generally required, but it can be a good practice, especially when you have a script with multiple statements. Back to the query. In my case, it loaded all 35, 000 records. This query works well, but it is less than ideal as it will return all rows and all columns. Let's try now modifying the query. SELECT TOP 3 Id, Score, Title, Tags FROM Posts, and execute. And there we have it. We have selected a set of specific columns from a particular table while limiting the number of results. I still have included the entire query in a single line, which works fine for smaller queries, but may be a bit confusing with longer queries. Let me show you how it can be improved while also showing how we can execute more than one statement at a time. As a good practice, you can add a semicolon, which may be required in other implementations, although not here. And you can also use GO, which is a command recognized by SQL Server Management Studio to indicate that it should send the current batch of Transact-SQL statements. GO is not part of the SQL language. So now we can break our query into longer lines, which helps for readability. Indentation is not required, but again, it helps you understand and analyze the query. SELECT TOP 10, then end your line and indentation, Id, Score, Title, Tags, end your line, FROM Posts;. Now in the Results, we can see the output for both queries, which is helpful. And talking about results, this is the great result, but if we wanted to, we could output to text by using the toolbar. Now, our results are in text and aligned using spaces. The other option is to output to a file, which can help you share the results, but that's a little bit beyond the scope of this course, although it may come in handy to share or save some results. And you can keep writing more complex queries like this one, which selects columns from different tables. It uses aliases. It uses a function, count, a join or two, groups some of the results, and filters them, and finally, sorts the results in descending order. All this to select the posts with more comments and displaying its owner with the title. And yes, that was quite a few clauses, which reminds me something that's important, which is the order in which SQL Server executes the statements. Let me cover that now. I am not going to go into too many details as there is a whole training dedicated to SELECT and the Data Query Language, but it is important to note that whenever you perform a slightly more complete SELECT statement that the clauses are processed in a particular logical order, which determines how objects defined in one step are made available to the clauses in the subsequent steps. It starts with FROM, which specifies which table, view, or table variable source is used to get the data from. Then ON, which is used to specify a join condition. Next, the JOIN, which helps retrieve data from two or more tables based on the relationship between these tables followed by a WHERE, which is used to filter rows. Having WHERE being early in a query is quite important because the fastest data that you process is the data that you do not load and process to begin with. Next, the GROUP BY, which is used to aggregate data. Then, WITH CUBE or WITH ROLLUP that is used to work with multiple grouping sets. HAVING is used to filter the aggregated data, then SELECT to specify the columns that should be retrieved with DISTINCT used to return only different values, ORDER BY to sort the final data, and TOP to limit the amounts of records returned. In this training, we will be using SELECT for things like creating sub statements to insert data, as well as other scenarios. With the quick demo that I just showed you, it should be enough to work with the DML. There is just one more thing that I need to expand, the WHERE.

What You Need to Know About WHERE for DML

The other clause that is very important for us to know, in fact, for the DML, I think is even more important than SELECT is the WHERE clause, which specifies the search condition for the rows that are returned by a particular query. As I mentioned earlier, a missing WHERE can have serious and unintended consequences. Like what, you may ask? Forget a WHERE on a DELETE and oops, table gone. Or like a friend of mine experienced, you run an update without selecting the WHERE, and he initiated a download and processing of about 10 terabytes of data. We found out when all alarms and dashboards started to light up like a Christmas tree. Let me tell you, it was not a fun day at work. Back to the WHERE. What do you include in the search condition? Well, you can provide one or more predicates. Which ones? At a high level, we can use the basic ones like EQUAL, UNEQUAL, and a Range. We use a comparison operator. We can also use LIKE, which can be used to find rows that contain a value as part of a string. It is not a replacement for a search engine, but it works, EXISTS to specify a sub query to test for existence of rows, IN to determine whether a specified value matches any value in a sub query or list, or BETWEEN to specify a range to test, ANY, SOME, and ALL to compare a scalar value with a single column set of values. In essence, WHERE is extremely important as it narrows down which records are affected. Don't forget your WHERE. Let me show you with a demo using the WHERE clause. Now I will show you a demo where I am hypothetically looking for a small subset of posts. All those with a higher score that are related to machine learning that WHERE created in a subset of years with at least one comment and a specific number of favorite counts. Let's see how we'll find this subset using WHERE. I am here in SQL Server Management Studio. And as usual, let's open a new query and make sure that our database, tsql-dml is selected. Now let's just get all the posts. SELECT * FROM Posts. It is a little bit hard to notice, but I get back 35, 379 records. It is right here at the bottom. But to make it easier, I'll put that number right here on the top right for you to see how the WHERE affects how many records are retrieved. I will add the WHERE and using one line per predicate, I will add the first condition, get all posts where the score is higher than 30. I execute and the number of results has decreased to 148. Now I use LIKE to filter on only those records that have a tag like machine-learning. When I execute, I narrow down my results to 32. Next, I will narrow down by those posts that were created in a specific year. Given that the field is a date, I need to use a function to extract only the year. Again, I leverage IntelliSense and I use IN to filter by the years where the values appear in this particular list. Now I am down to 18 rows. Something that is important to take into account is your Boolean operators, namely AND, OR, and NOT. If you make a mistake or use them incorrectly, your results will most likely be wrong. Notice how I get now 16, 000 rows by using an OR, so let's switch it back to an AND. Another condition. In this case, EXISTS, but in combination with a sub query. Notice how I am using a SELECT statement to create this sub query to make sure that these posts have at least one comment. We are now down to 13 rows. One more condition. I want the FavoriteCount to be between 40 and 100. Please notice how AND is used for both as a Boolean operator for the entire clause and to establish the range for BETWEEN. We are now down to five rows, which are the ones that we were looking for. And that, at a high level, is everything that we need to know about the WHERE. Well, almost. I'll just add one more thing, which while it does not apply specifically to WHERE, it can be the culprit of a big mistake. It is what happened to my friend. We have five rows now as the result of a query, but if we select only a portion of the query and execute, then SQL Server Management Studio executes only whatever is selected, which is good as you don't have to comment out what you don't want to execute. But, on the other hand, you may execute an incomplete query and run into an issue or two as my friend did. Anyway, let's now move on into the takeaway for this module so that we can start to go deeper into the DML.

Takeaway

In this module, we learned about the four different T-SQL sub languages, the DQL, the Data Query Language, DDL, the Data Definition Language, DCL, the Data Control Language, and the star of our course, the DML or Data Manipulation Language, which we use to INSERT, UPDATE, and DELETE data. The next few modules will cover in depth the DML. And finally, we took a few minutes to learn what we need for the DML of SELECT and WHERE. Let's now learn how we can add and modify data in the next module, so please come and join me.

Adding Data Using T-SQL Statements

Adding Data Using T-SQL Statements

Whenever you think of working with a database, what is the first thing that comes to your mind? Is it adding and modifying data? Well, not quite. At least for me, but I think I am not alone, whenever I have a table, I commonly query data more often than anything else, but to query, you first need data. And even before that, you need to create a table to hold your data. And now you can add your data and modify it as you see fit, which means that you can finally query your data. But querying is not why we're here. We are here to add, modify, and even remove data. And these operations are performed with these three statements, INSERT, UPDATE, and DELETE. And now in this module, we're going to focus on adding data, which we do using INSERT, which can be used to add one row at a time or even multiple records with BULK INSERT. The other DML statements will be covered in a future module.

Adding New Records with the INSERT Statement

INSERT, as the name clearly implies, is used to add one or more rows to a table or view in SQL Server. As we can see here in the syntax diagram, there are many different arguments that can be used with INSERT to accomplish the objective of adding data, but in slightly different ways. It may look at little bit complex to understand, at least for me at first glance, but it really isn't. But here's a tip. To read a syntax diagram, you need to understand the conventions used, which luckily for us, they are linked right here. At a high level in a syntax diagram, the Transact-SQL keywords are written in uppercase, but this is just a convention when writing your queries. If you use lowercase, it is fine. If you see a word in italic, it means that it is a user-supplied parameter. If you see brackets, it means that an argument is optional, and so on and so forth. I am not going to go one by one on all of the conventions. You can find the link in the exercise files provided for this module if you want to understand each and every convention used. For now, I wanted to show you where to find the conventions and what all these symbols meant. And indeed, there are some arguments that you will use almost every time while others will be kind of rarely used. So let me start with the basic syntax and then move on into the more specialized arguments. The most common statement usually is INSERT INTO table name, then column_one, column_two, up to column n, VALUES, and then you provide an expression, but also null values can be inserted or a default value. With INSERT INTO used to specify in which table the record or records will be added, then the column_one, column_two, up to column n part is used to specify which columns are going to be inserted, and VALUES provides the raw data that will be inserted into the columns provided in the previous argument. Now I will show you a few demos of how to use INSERT in different scenarios.

An Insert Statement with Columns and Values

The first scenario that I want to show you is when you execute an INSERT statement where you specify the columns and their corresponding values. In this scenario, as we learned when we saw the syntax a few moments ago, you provide a list of which columns you want to insert data to, for example, Reputation, CreationDate, DisplayName, and so on and so forth. And then you provide the values for each one, being mindful of the type. It could be an int, a DateTime, a varchar, a NULL, and more. You provide the columns that you need and in the order that you want for each particular value. Let's see a demo of an INSERT statement with columns and values. Let me start in SQL Server Management Studio by expanding the database, then tables, users, and finally, columns. I know it may be a bit small to see, so just let me show you this particular section. Okay, this is better. Now, what's important here is that we can see the columns that are defined in this table with their types. Now I will open a new query window and count how many records this table has just to make sure that I am adding records to it. SELECT count* as Total FROM Users. Then I click on Execute query, and I get this number back, 61838. I just need to remember this number for later on, or I'll make it easier. I'll just leave it here quietly in a little corner, then I will close this tab. Now, again from the Object Explorer, right-click on the users table and click on Script Table as, and here you have multiple options depending on the type of SQL statement that you want to execute. We are interested in INSERT To and then New Query Window. This creates a basic template for the INSERT INTO DML statement. Let me close the Object Explorer to focus on this command. I will add the values starting with an int, a date for which I use a function to cast to DateTime. My name, which is varchar, and a few more values, including a NULL, and at last, the account ID. And maybe you noticed or maybe you didn't, but I am not adding a value for the primary key ID explicitly. The reason is ID, besides being a primary key, it is an identity column, that is it provides an auto incrementing number on each INSERT. Primary key and identity are different, but they are used together quite often when no natural key occurs. When ready, I click on Execute query, and I get the confirmation message, 1 row affected. With this, we have added one row using INSERT with columns and their corresponding values. Of course, let's confirm. I will open a new query window and SELECT count * as Total FROM Users. How many should I get? If you guessed one more than the number right here, then you are right, 61839. The number of records in our table has increased by one, which means it worked well. Let's just take a look at the record, Select * FROM Users WHERE_AccountId =, and the one we provided when I executed the INSERT INTO. And I can see my record in the results window. There it is. I have inserted one record into the user's table using the INSERT statement from the DML, but this is just one way of using INSERT. Let me show you a few more ways.

An Insert Statement with All Values

In the previous scenario, we specified a list of columns and then each corresponding value. Actually, I provided all columns with a value even when it was a null or the default; however, providing all columns with all values is optional. There are some cases when you do not need to specify a value on INSERT and the column will take either a default value or a NULL. And even going beyond, providing the columns argument is optional. You do not need to include it if you provide all values and in the order in which they were defined in the table. Let me show you with a demo. An INSERT statement with all values. This is the INSERT statement that we used in the previous demo. It has a list of which columns we are going to be inserting data into and the values. Well, I just modified the name. It is now steve-morera, and I incremented the account ID by 1. We are providing values for all columns, and they are in the order in which the columns have been defined. This is much simpler to confirm if I use the Script Table as option from the Object Explorer. Anyway, this means I can come and remove the list of columns and leave in the values. Then I click on Execute query, and I get the message I am looking for, 1 row affected. Just to confirm, I can take the account ID and copy it. Now I will click on New Query, and then I'll write SELECT * FROM Users WHERE_AccountId and the account ID that I just inserted. Now, I will click on Execute query, and there it is. Steve-morera is now a user. And that is how you use the INSERT statement, but without providing the columns argument. Let's keep moving forward so that I can show you the other ways to use INSERT.

An Insert Statement with NULL and DEFAULT

In the previous demo, we provided values for all columns. This included those columns that are defined as NOT NULL, that is they must have a value that is different to NULL. Some of them are quite important, for example, Id, which also needs to be unique or CreationDate, which can also be important to track. Others require a value like Views because if you try to add up a number to a NULL, it may not work as expected. And then we have the other columns that support NULL like the WebsiteUrl and other pieces of information that may not be that critical to our application, and hence, they can be left as NULL. Now, let's see what we can do. Demo. An INSERT statement with some NULL and default values. Here is an INSERT statement that has all columns and all values. It is pretty much the same one I used in the first demo for this module. I just modified the DisplayName and incremented the AccountId by 1. I am going to remove these three columns, WebsiteUrl, Location, and AboutMe. These three, if you remember, are columns that accept NULL. I will also remove their corresponding values. In this particular statement, we were inserting two as NULL, but location was set to Costa Rica. Yeah, the place that you think about for vacation, but that I call home. I will do the same for the other NULL column, ProfileImageUrl, including the value. I am ready to insert this record. So I click on Execute and I get the 1 row affected message. All's good. Let's take the AccountId and run a query to retrieve this record, SELECT * FROM Users WHERE AccountId = the one I just inserted. I'll execute, and I'll get the record. As we can see, a NULL has been inserted for those columns that we did not provide a value. That was quite straightforward, right? But now let me get back to the query. I'll also make the Messages pane slightly smaller to make up some room. I let you remove from this query the Views column and its value. Why is this important? Well, because Views is a NOT NULL column. This means that it must have a value. So if I do not specify a value for a column that must have value, then I will get an error. Is that correct? Well, let's test and see. I just need to increment the AccountId by 1. I leave the DisplayName as is and then click on Execute query. And here's the question. Did I get an error? Well, not quite. One row affected. Hmmm. Let me go back to

SELECT query, change the AccountId and Execute again. It's probably a bit hard to notice, but right down here, you can see how the AccountId changes. So, what happened? The answer is quite easy. Let me open the Object Explorer. And from my Users table, I expand Constraints. And there is that DF_Users_Views constraint. That, if I Script Constraint as, CREATE To, New Query Editor Window, I can see that there is a DEFAULT value FOR Views, which is set to 0. This means that I do not have to worry when I am executing an INSERT if I do not include a value because a default value will be used. In this case, it will be a 0 FOR Views. This makes my life a lot easier, right? Of course. But a default value is not automatic. Default values need to be specified when creating the table, but this is part of the DDL or Data Definition Language course, so you need to take it into account there. Now, let me show you something interesting.

An Insert Statement with an Identity

Something that we have not done so far is to include the ID, the primary key, in our INSERT statements. Why? Because it is an identity column, that is an auto incremented value is assigned automatically on INSERT; however, this does not mean that you can't specify the identity explicitly. There are cases where you need to do this, but if you must, you need to set IDENTITY_INSERT to on to allow this because if you don't, an error is raised. Let me show you. INSERT with identity. Let's learn how we can INSERT with identity. I am here in a new query window. I'll run a simple query. SELECT Id, DisplayName, AcccountId FROM Users WHERE AccountId greater than or equals to, and I will use the first AccountId that I inserted in the demos for this module. I get back four records. The first thing I notice is how the ID increments by one at a time. The IDs are sequential. And let me just make sure that these are the last IDs that have been inserted by using the MAX function. I now run this query, SELECT max AccountId FROM Users. Execute query and indeed, that is the max record, which means that I can insert a new record and explicitly set the ID. And here I have the query that I have been using to INSERT, but with a couple of placeholders. Those are the X and the Ys. I will add the DisplayName, xavier-identity, and change the AccountId to increment by 1. It ends up in 93 now, and then I will get the max value for ID. I replace AccountId by Id, and I get, which is the latest one. I have almost all the information that I need except the ID, so I add the Id column, and then I get the highest ID, which is the one that I'm going to have to increment by 1 and include in the values. Now, I execute and I get an error because you can't explicitly INSERT an identity unless you tell SQL Server that you are going to be doing it. The full error message is, Cannot insert explicit value for identity column in table 'Users' when IDENTITY_INSERT is set to OFF. Well, lucky for us, it hints us in the right direction. So I move to the top and add SET IDENTITY_INSERT Users ON, and execute it. And now I can safely run the INSERT statement, and I set the identity explicitly. I get the confirmation message. To confirm, I will move to the other tab, add a GO, and execute both statements. And now we can clearly see in the result how a new record has been inserted explicitly setting the identity. But word of advice, unless you really need it, setting IDENTITY_INSERT to OFF is much better as SQL Server is the one in charge of auto incrementing the values. Now, let's see what else we have.

An Insert Statement with Constraints

Here's another important topic to take into account when adding new records. SQL constraints are used to specify rules for the data in a table. They help on scenarios where it is required to limit the type or value of the data. And these rules are enforced by aborting an operation when a constraint is violated. Constraints can be at a table level or column level. Some of the typical constraints include specifying that a particular column's value cannot be NULL or that it must be unique or that a column is a primary key, which means that it can't be NULL and it needs to be unique. Also, foreign key, which uniquely identifies a row in another table or to set a default value as we used in a previous demo. And that's just to name a few. And, of course, this applies directly when using INSERT. Let me show you. Demo, INSERT with constraints. I will start in SQL Server Management Studio where as usual, I will open a new query window, and I will run this query, SELECT TOP 10 * FROM Comments and execute query. This is the table where I will be adding a new comment. Something important is that each record in this column that is a comment is related to a particular post, that is each post can have 0, 1, or many comments. It is a one-to-many relationship. And how are they related? Well, each comment contains the reference to key from the post in the PostId column. It is a foreign key constraint. And this also applies for the UserId, which maps to the ID in the users table. Let me check at least one of them. To demonstrate this, I will take this PostId 5 and run a query to get the post that I am referring to. But before, let me just save this text right here to confirm. This is a super theoretical AI question, an interesting and ellipses. That should be enough. Now, the query, SELECT * FROM Posts WHERE Id = 5, and I execute it. I get this record, which was created in 2014, and if I check the title, I can save it here. How can I do simple machine learning without hard-coding behavior? I'll use this to check something soon. Just give me a second because if I check datascience.stackexchange .com, I can see that I have post 5, and the title matches, How can I do simple machine learning without hard-coding behavior? But right here below, I can see the comment, this is a super theoretical AI question. Again, this is a match. And that was quite an interesting way of confirming how a database is used to run this Q&A site. But now let's go in and insert a comment. I have used the trick of Script Table as, INSERT To, New Query Editor Window to get this query. Now, I'm just going to fill in the blanks. You already know quite well how to do this. I will use 5 for the PostId, 0 for Score, then provide the CreationDate, me as UserDisplayName, and for the UserId, let me just run a quick query, SELECT * FROM Users WHERE DisplayName = xavier-morera, execute query, and that's my ID. I will copy it, and I will add it as an argument in my INSERT query. And this would run fine, but what if I add a few 0s to the UserId, which means that I am now violating a constraint because there is no UserId with this number. So if I run, this is what I was expecting. I get a message telling me that I can't insert a value if I am violating a constraint. The precise message is a bit longer. It reads, The INSERT statement conflicted with the FOREIGN KEY constraint "FK_USERID_COMMENTS", and it also tells me where it occurred. Finally, something very important. The statement has been terminated. This means that my INSERT failed. But you might be wondering, how do I know which are the constraints in my table? Well, it's easy. Open your database, navigate to your table, which is Comments in this case, open Keys, and here you will see the Primary Key, which starts with PK and then the foreign keys, which start with FK. There are two, one for the post and one for the users. If you click on one, it will show you the foreign key relationships with more information, which includes what should be checked on creation, if it should enforce replication, and what are the rules to apply when INSERT and UPDATE are executed? In a future demo, we will talk about triggers to get to know more about constraints and cascade actions. Also if you Script Key as, then you will be able to see the constraints, but this is more of a topic for the DDL course. For now, you just need to know that if you insert data with the incorrect values for a constraint, your statement will be terminated, that is your statement will fail, but if you use the correct value, and let me just fix this one, then the INSERT will succeed, and you will add the record. Let me just confirm this by running a query. If I run this SELECT, I can see my new comment right here. It has been inserted successfully.

Adding Multiple Records from a Data File with BULK INSERT

So far, we have been adding records one at a time, which works well, but sometimes we want to INSERT more than one record at a time, like when? Well, suppose we have a pretty large file with structured information like a CSV file. That's a comma-separated file. In a scenario like this one, it is much better to import all the records in the file at once instead of importing one row at a time be it with an application, a script, or running multiple SQL statements. SQL Server can do this for you using the command BULK INSERT, which imports the data file into a database table or view in a user-specified format that SQL Server understands. In this demo, I will show you how to import a CSV that contains multiple comments into a table. There are other scenarios with more specialized formats, which we will cover in a future module with some advanced techniques. And just at a high level, these are some of the parameters that can be used with BULK INSERT. I won't spend a lot of time just repeating to you what they do. The explanation is included in docs.microsoft .com, but I will mention some of the most important ones. First, BATCHSIZE, which specifies the number of rows that should be processed in a batch. A batch is just a group of records. The whole point is breaking up the job into smaller chunks. This is important for when you're processing large files. Another one, ERRORFILE, which specifies the file used to collect rows that have formatting errors and cannot be converted to an OLEDB rowset. In summary, these are the rows that you need to manually review. Then FIRSTROW, which specifies the number of the first row to load. This is useful when the data file includes headers because you don't want to import your header as a row. Another one, KEEPIDENTITY, which is used to set the identity explicitly. KEEPNULLS, which is important to keep the NULLS instead of using default values. MAXERRORS can help set a threshold of how many errors are acceptable until the operation is cancelled, and there are a few other important ones like FIELDQUOTEs to set which character to use for quotes and FORMATFILE, which helps you specify the format that you want to use. There are other fields to specify termintators like FIELDTERMINATOR and ROWTERMINATOR. And these together to format FIELDQUOTE, FORMATFILE, FIELDTERMINATOR, and ROWTERMINATOR are also called Input File Format options. As mentioned, many other options are explained in more detail in the official Microsoft documentation, but for now, let me show you how to use BULK INSERT. Demo, adding multiple rows with BULK INSERT. This is the file that we will be inserting, comments.csv. You can find it in the exercise files for this training. If I open the file with Notepad++, I can see how the first row contains the columns for the comments table while the next few rows contain the values for each row in comma-separated value format. Also, please notice that they are in order. All of these three comments have myself as the UserDisplayName and with my UserId. Let me check how many comments I have added up until now. SELECT * FROM Comments WHERE UserDisplayName = 'xavier-morera'. Execute query, and I can see that I have added exactly one comment. So my intention now is to get these three comments that I have in this CSV file into my comments table. To achieve this, I will open a new query window and run this statement, BULK INSERT Comments. This is how I specify that I will insert multiple records into the comments table. Notice there is no INTO keyword, just BULK INSERT and a table, then the location of the file, which in my case is 'F:\tsql\comments.csv ' WITH, and now I open a parenthesis to provide the arguments. My CSV is straightforward, so I just need to specify FIRSTROW = 2 because the file has a header row, FORMAT='CSV'. This is pretty standard. And for this example, that's it. I can now just execute the query, and I get the message, 3 rows affected, letting me know that multiple records have been inserted into this table. Now if I go back to my original query, which had only one row, if I reexecute now, I will get four rows. It worked. I have inserted multiple rows with BULK INSERT. As I mentioned earlier, I will cover other more complex formats in a future module. For now, let's do the takeaway for this module so that we can move forward into the next topic, how to modify data using UPDATE.

Takeaway

In this module, we started by stating how SELECT is potentially the most common statement that you will run in your database, but then we thought twice about it and realized that to be able to query data, you first need data. And to add data, you use the INSERT statement from the DML. Also, we learned that there are multiple scenarios to consider. We can INSERT with all columns and all values or just by providing all values, then we learned how to INSERT with NULL and default values and how to set explicitly an identity. We also learned how to INSERT taking into account constraints like a foreign key, and finally, how to insert multiple rows at once with BULK INSERT. And now let's move on into the next module to learn how to modify the data that we just inserted using UPDATE.

Modifying Data Using T-SQL Statements

Modifying Data Using T-SQL Statements

Now that we have added data to our database, there is a new need that we may have, to modify our data using T-SQL statements. And for this purpose, we use one of the available statements from the DML, the UPDATE statement. In this module, I explain how to update your data be it one row at a time or many rows at once, which is a scenario where you need to be extremely careful. It is quite easy to make a mistake and update way more than you intended. And given that a database is commonly how many applications stored their state, you could run an update that may have unintended consequences with potentially catastrophic outcomes. So stay with me while I teach the practical aspects of how to use UPDATE both with the more commonly used scenarios, as well with a few ones that are less common.

Modifying Records with the UPDATE Statement

UPDATE, as the name clearly implies, is used to modify one or more rows to a table or a view in SQL Server. And as we can see here in this syntax diagram, there are many different arguments that can be used with UPDATE to accomplish the objective of modifying data, but in slightly different ways. Just like with other DML statements, it may look a little bit complex to understand at first glance, but because you already know how to read a syntax diagram, I briefly covered this in the previous module, you now understand the conventions used, and it all looks clear. UPDATE has several potential arguments, some that you will use almost every time while others will be kind of rarely used. Let's start with the most commonly used and basic structure for UPDATE. It goes something like this, UPDATE table_name SET column_1=value_1, column_2=value_2, and so on and so forth until you get to column_n=value_n. And then the WHERE where you provide a condition. The UPDATE table_name specifies which is a table where 0, 1, or many records will be updated, then SET and a list of which columns with which values should be updated. This is slightly different from INSERT where you first had all the columns and then all values or simply all values in order. In any case, this is quite clear. Also, you can update as many columns as you want, be it one or all as long as you respect constraints and the types. And then the WHERE clause, which limits the scope of which records will be updated. It is not mandatory, but most of the time it is used, else you will be performing an UPDATE on all records in the table, something that is not usually what's intended. Something that is quite important to mention even though it falls a little bit out of the scope of what I am focusing on today is that if you are building an application that performs updates, that you should always validate the input to avoid SQL injection, that is making sure that a malicious user is not sending statements that can perform massive updates or deletes. Now let me show you how to use UPDATE with a series of demos.

Using UPDATE to Modify Data in One Row

The first scenario we're going to cover, which is one of the most basic scenarios is how to modify data in one row be it one or many columns using UPDATE. Something that is quite important is that when we're using UPDATE that we narrow down the scope of which rows we want to update using WHERE. That's usually you want to modify a subset of records and not the entire table. And aside from this, you have to be careful because you can't just update any value in any row. There are columns or fields that you can't just update, for example, autoincremented values and constraints. Let me show you with a demo using UPDATE to modify data in one row. Let's begin by opening a new query window from SQL Server Management Studio and running a simple query, SELECT * FROM Users WHERE DisplayName = 'Xavier'. Let me execute and let me direct your attention to here, the UpVotes and DownVotes, which are set to 0. Let's imagine for a second that the task at hand is to change these values. So let me get ready to update them. I am going to start with UPDATE Users. This is how I specify the table, then SET, and I provide which are the columns with the new values. UpVotes will now be 10 and DownVotes will be 5. And here's where you should be careful. If you just execute, then you will update all values in the table, which is usually not what you want, so you narrow down which columns should be modified using a WHERE. In this case, WHERE DisplayName = 'Xavier'. It has the same WHERE clause that I used above. I will select the UPDATE statement and execute, 1 row affected. Let's reexecute the SELECT, and indeed, if we look at the UpVotes and DownVotes, they have been updated accordingly. And that's how you update one or many fields in one row.

Using UPDATE to Modify Data in Multiple Rows

The next scenario involves using UPDATE to modify data in many rows be it all at once, which can either be done by setting a new value or using a calculated value. And when I say many rows, I usually mean a subset of records from the table using WHERE, but it can even mean the entire table. Let me show you. Demo, using UPDATE to modify data in multiple rows. Let me begin by opening a new query window from SQL Server Management Studio and typing in the following query, SELECT * FROM Users WHERE DisplayName LIKE, and then a new string that has xavier in the middle. I execute and I get eight rows back. If I scroll to the right, I can see how they have a different number of views. Some have a 0 and other users have a 1 and 9 views respectively. And what I have been requested to do is to increase the views for all these users to 100. By increase, I mean update this value. So to do this, I will run this query, UPDATE Users, I specify the table, SET, then Views = 100. This is which column to which value, and then I limit which are the records that I will be modifying, WHERE DisplayName LIKE '%xavier%'. I select this query and execute. Good. I get back eight rows. Now to confirm, I will run the SELECT statement, which if I scroll to the right, I can see that the value of Views has been updated to 100 for these eight records, and this is how you use UPDATE to modify data in multiple rows. And needless to say, if I don't specify a WHERE, you will update the entire table. Now, let me show you something else that will prove quite useful.

Retrieving Number of Affected Rows with @@Rowcount

Now this is something that may prove quite useful, especially when you're developing applications. Currently, we're working and learning using SQL Server Management Studio, which tells us how many rows are affected on each query, that is you run the query and it immediately lets you know if your update modified one or eight rows. But an application does not use SQL Server Management Studio. It will most likely run a query by connecting to the database, and it needs a programmatic way to know how many rows were affected, and for this purpose, we use @@ROWCOUNT. Let me show you. Demo, using @@ROWCOUNT to obtain number of rows affected by UPDATE. For this demo, I will pick it up from the previous UPDATE demo, which I already know that it updates eight rows. I will delete the SELECT statement from the top and add a 0 to the number of views to make sure that multiple rows are modified with a different value, and I will add the following statement, IF @@ROWCOUNT is greater than 0, which we know it's true, then PRINT 'Multiple rows have been updated'. And then one more query, SELECT Id, DisplayName, Views FROM Users WHERE DisplayName LIKE, and then '%xavier%' with the percent signs on both sides, which now I execute, and I get back all eight rows in the results window, but let me move to the Messages window where I can see that the message that I added in the IF, this IF right here, was executed. Now I will change the value of Views to 1001, just a different value to test, and I will declare a variable called @rowsAffected of type INT. Then I will set this value to @@ROWCOUNT and change any other occurrence of @@ROWCOUNT for my variable. Then I will concatenate the message, that is the rowsAffected message, with my number so that the message tells me how many rows were affected. Of course, I have to cast my int to a varchar to be able to concatenate to the string. I reexecute and now I can see the new message, which contains the number of rows, eight, and that is how you know how many rows were affected on a given operation with @@ROWCOUNT. Let's keep moving forward and see what else we can do with UPDATE.

Modifying Data Using UPDATE with Constraints

Let's talk now about what happens when you try to modify data using UPDATE when there are constraints. In a nutshell, we just modified some values in your tables. For this, we did an update and we specified what is the value for a particular field for one or many rows. It was pretty straightforward, right? Well, unless there is a constraint for that particular field that you're trying to modify. One example of a constraint being a foreign key in a case like this when an error is raised if the constraint is violated. Let me show you. Demo, using UPDATE with constraints. Here I have a statement from the DDL, the Data Definition Language, but this is a DML training, so I won't get into too many details of what it does except for this part right here, ADD CONSTRAINT FK_USERID_COMMENTS FOREIGN KEY UserId REFERENCES dbo.Users Id. What it basically means is that there is a relationship between comments where each comment was made by a particular user and that the UserID should match an ID in the users table. In layman terms, it's just a simple foreign key. Now, let's visualize this relationship by looking at the data. I will open a new query window and we'll look for all the comments that are related to PostId = 5. I execute, and we have two, but let's focus on the first comment. You might recall this one from a previous demo. It is this one right here, the one that starts with this is a super theoretical demo. This is the comment that is related to this post and that we're going to modify the data and violate a constraint to see what happens. Back to the result, the idea of this comment is 5, so comment Id 5 references PostId 5. And this comment was written by UserId 34, which is the last column in this result. Now from a new query window, let's write this UPDATE statement, UPDATE Comments SET UserId = 34. This is the current user and a valid one, but now let's add a few 0s. This way we're going to be referencing a user that does not exist, and finally, WHERE PostId = 5. Now what do you think should happen if I run this? Well, first there is something that I need to check. I'm just going to confirm quickly that there is no user 34 and all these 0s. If there was a user with this ID, then the UPDATE would work fine. So I check using this SELECT statement, and indeed, there are no users with this UserId. I'll delete the query and execute. And I get the expected result, The UPDATE statement conflicted with FOREIGN KEY constraint "FK_USERID_COMMENTS". This is the constraint that I showed you at the beginning of the demo. And then, I get a few more details telling me which are the columns. And that's it. With this demo, I am able to show you what happens when you try to update a value that has a constraint, so please check your data before running an update to make sure that you are providing appropriate values that do not violate any constraints. Now let's see what else we have.

UPDATE Using Variables

So far, we have been assigning values to columns, that is our values would be a particular number or a string or something similar, but there is another scenario where you can assign values to variables to perform one or more operations and then use these variables in the UPDATE. This has clear advantages as you do not need to know upfront which is the value that you will be updating, but instead, you can calculate or create new values based on existing data. This is extremely useful for multiple rows. The variable assignment is performed on each row in the output. Let me show you. Demo, using UPDATE with variables. Let me first run this query, SELECT Id, DisplayName, Reputation, CreationDate FROM Users, and then I execute. I get 60, 000 something users. That is quite a bit, right? But now for this exercise, I think it would be nicer if I get those that have the highest reputation, so I will add an ORDER BY Reputation in descending order and execute. And how would this work? Okay, let's remember these values. For UserId, 836. The Reputation is 17699, and this user created the account in 2014, which is about 5 or so years ago. Why is this important? Because the task that I have been requested to do is to update the reputation of all users by increasing it with the number of years that they have been in this site. If I am not mistaken, this user will end up with 17704 as the reputation after my update. Let me leave these values here and see if it works. So I will declare a variable. Remember, a variable in SQL Server is not that much different in essence to variables in other languages. It can hold a data value of a specific type, but with a slight difference that when they are using a batch, that is multiple rows are returned, then it is used once per row. It can be used as a counter or to save data that can be used to perform operations. And then use the variable to update one field or many. My variable will be YearsActive, which is an int, and I will initialize it to 0. I will use this variable to perform a calculation and save the result. So I start my UPDATE statement, just a well-known UPDATE Users SET, and then YearsActive, I set it to get the current YEAR as a number using the function GETDATE with this abstraction of the Users.CreationDate. YearsActive should now be set to 5 for user 836, but this calculation will be performed on each row that is returned. Then I will increase the reputation by YearsActive using the += sign. Now I will select the UPDATE and the DECLARE statements and execute. I can see the message that tells me that 61, 000 rows have been affected. As a note, you can also use TOP to perform an update on only a subset of records, but I'm doing it on all records, so now let me confirm by running the SELECT statement again. And indeed, for user 836, I can see that the reputation has increased to 17704, which is the number I predicted a bit earlier, and that is how you perform an UPDATE using a variable. Of course, you can use more than one variable. Now let's see what else we have.

FROM and JOIN within an UPDATE

Here's another scenario that can be quite useful from time to time. So far, we have been updating columns using values, and as we just learned, using variables as well. But what if you want to update a column using data that is in a different table? Well, here's where you can use a FROM and a JOIN. Let me show you. Demo, using FROM and JOIN within an UPDATE statement. For this demo, I'll start by running this Title FROM Posts WHERE PostTypeId = 1. This is to filter by questions only. And how do I know this? Well, it's in the README of the stack exchange dump. I execute, and these are my users. Just like in the previous demo, I will remember the top post, which has a score of 176. This post has four comments. Go ahead and check it yourself if you want to, which means that the new score for this post will be 216 after the update. Now I will write my UPDATE statement, UPDATE Posts SET Posts.Score. And please notice how I am including the name of the table first. It's important to avoid situations where two tables have the same column name. And I increment its value by the score in the comments times 10. In this case, the comments score will be added up for every comment, even if we specified only one addition. That is, it evaluates this operation for every comment where it matches the criteria that I am about to specify, FROM Posts INNER JOIN Comments, those are the two tables that I will use on this statement and then the clause that is used to know which rows should be used, that is Posts.Id = Comments.PostId. So all comments for a particular post will be used for the operation indicated above. And just to make sure that I use the same WHERE as above, I add it here, WHERE Posts.PostTypeId = 1. Now I execute, and over 7, 000 rows are affected. Now if I check by running the SELECT statement again, I can see that I get 216 as score, which is the one that I predicted earlier. There were four comments with a score of 1, so the score was increased by 40. This type of update is quite useful as you're modifying your data based on data that's already available in your database even if in other tables. And it's especially useful for BULK UPDATES, but also effective for smaller ones. Now let's talk a bit more about what you can do with more advanced classes with the UPDATE.

A Few More Details on UPDATE

So far, we have covered some of the main scenarios that you will run into when working with the UPDATE statement from the Data Manipulation Language. We could spend quite a bit of time covering all possible scenarios, but that would take a really long time. Luckily, unless you have a really specific use case, you are ready to tackle most business cases for modifying data in your table. So let's spend just one more minute or two reviewing what else you can do with UPDATE. Well, at a high level, you can use a common table expression, which as you know, it is a temporary result set that you can use for the update. Then, you can specify where to apply the UPDATE, which can be applied only to a subset of rows, which can be specified with TOP. Then you can set on columns, which is a normal use case, but you can also set properties, fields, or methods on user- defined types, as well as being able to set variables on which you can apply operations. Additionally, just like with INSERT, you can use the OUTPUT clause to return data or expressions after the UPDATE has been performed. Then the FROM, which we just saw, which allows to specify other tables or views that can be used in the UPDATE. Then the WHERE to narrow down the scope of the UPDATE. For this, you provide a search condition, and you can also use with cursors. And finally, the ability to use OPTION with query hints. As mentioned, there are many possibilities on how to use UPDATE, but we covered the most common scenarios that will help you modify data with UPDATE. Let's now do the takeaway for this module so that we can continue with the next DML statement, DELETE.

Takeaway

In this module, we learned how to modify data in our database using UPDATE, which has many different available parameters, but that the main scenarios involved updating in a destination table a set of columns with some values on a specific set of rows. We then learned how to UPDATE data in one row using this basic UPDATE. And we went a bit further and modified multiple rows. But be careful as multiple rows can also be all rows, which if not done on purpose, can be an issue. Then we learned how to use @@ROWCOUNT to programmatically get the number of rows affected by an UPDATE, which is quite useful when you're accessing the database from an application. And just like with INSERT, you could also use OUTPUT to return what has been modified. Also, you can use variables to perform operations and assign the result of an expression through your columns, which works on one or many rows, that is the variable is evaluated for every row that's returned for the UPDATE. Then you can use FROM to specify other tables or views that can be used in the UPDATE. Let's now get on into the next DML statement, DELETE.

Removing Data Using T-SQL Statements

Removing Data Using T-SQL Statements

Now that we have added and modified data in our database, it is time to take the next step, learning how to remove data from our database using one of the available statements from the DML, DELETE, although we will also talk about TRUNCATE. But there is one word of advice. Something that's important to mention is that whenever you're using DELETE, make sure that you indeed do not need the data because when you delete it, it is gone. Bye-bye. Hasta la vista data. Something that I've seen and done multiple times that may be helpful that is instead of actually deleting a row that you mark it as deleted. That way you still have the data in case you need it for analysis, counting, grouping, reviewing, or making it available again. I am not saying you should not delete, but just telling you that there are other options that you should consider. And, of course, you should also take into account backups. Having said that, let me now show you how to remove data from the database using DELETE.

Removing Records with the DELETE Statement

The syntax for the DELETE statement is pretty straightforward as it has two primary components. It can be as simple as DELETE FROM table_name, WHERE, and a condition. The first component, that is the DELETE FROM table_name, is where you specify the table where records are going to be removed. You specify one table. And then the WHERE condition is what you use to narrow down which records are going to be deleted be it one, many, or if there is no WHERE clause, then all records in the table are deleted, and that's the basic DELETE, although you can use some other arguments that we talked about in the previous modules, namely rowset_function_limited, which inserts data into another linked server using open query or open rowset functions. Then WITH that specifies one or more table hints that are allowed for a target table and the OUTPUT clause, which returns the rows that were deleted by the statement. This is very useful because sometimes when an application is interacting with a database, it requires the ID of recently deleted records. Now, let's see the DELETE in action.

Removing a Specific Record with DELETE

Removing a specific record with DELETE. This first scenario that we're going to cover is the most basic one, but probably the most common one as well. It's basically when you need to remove data, namely removing one specific record from a table, which is done, as mentioned earlier, using the DELETE statement in combination with a WHERE that narrows it down to a single record. Now let me show you with a quick demo how it works. Demo, removing a specific record with DELETE. I'm going to start by running this query, SELECT * FROM Comments WHERE PostId = 5. I'll execute, and I get back two records. These are the two comments that are related to this post in particular. Let's say that I want to delete one of these comments, the one with Id 5, so I quickly write the query, DELETE From Comments WHERE PostId = 5. Oh, wait a second. I was about to make a mistake here. Both the Id and the PostId for the first comment are 5, so if I run this query, I will delete all comments, not just this one in particular. So I am not going to execute this query. Why did I show you this? Well, I did this intentionally just to remind you that deleting data can be a big deal, so please be very careful whenever you're going to run a DELETE statement. So let me go ahead and write the correct query, DELETE FROM Comments Where Id =, and let me copy paste the Id 58867, and then I execute, 1 row affected. This is important because I'm only trying to delete 1 record. So now let's confirm, and for this, I will reexecute the SELECT statement. And as we can see, the comment with Id 58867 has been deleted from the database, and that's how you remove one specific record from the database using the DELETE statement from the DML.

Removing Multiple Records with DELETE

Now let me talk about removing multiple records with DELETE, which is another commonly used scenario, but one where you need to be a little bit extra careful as it may be easier to make a mistake that may end up with unintended consequences. Why? Because for our particular scenario, you need to filter by a set of records that you want to delete. And given that it is potentially multiple records, it is possible that you may end up deleting more than intended. In fewer words, make sure that you correctly filter by those records you want to delete. Let me show you. Demo, removing multiple records with DELETE. Let's see this next scenario. SELECT * FROM Comments WHERE UserDisplayName = 'xavier-morera'. Earlier, I inserted a few comments using BULK INSERT. You remember this, right? Well, now the task at hand is to delete these comments all at once. So let me work in this. Here is my statement, DELETE FROM Comments, and let me copy the WHERE with the search condition above, WHERE UserDisplayName 'xavier-morera'. Then I select only this statement, execute, and 3 rows affected, which gladly matches the number of rows that I wanted to delete. But, of course, it is always good practice to check, so I run the SELECT statement above, and indeed, 0 rows have been returned. That's good. That is how you use the DELETE statement with multiple rows, although for this one, I deleted all rows where that field matched. I can also use a search condition where I can use operators and only a subset of certain rows will match, like this one. Let me now open a new query window and type this query, SELECT PostId, Count(PostId) as PostCount. Aliases are always a good idea for clarity and then deaggregation, GROUP BY PostId and ORDER BY PostCount DESC. Why am I doing this? Because I want to look for the post that has the highest number of comments and delete a few of those, and PostId 25208 has 24 comments. This one is going to work for my intents and purposes. Let me return all comments from this PostId, and there it is. Here are the 24 comments. Okay, so now the task at hand is that I have been requested to delete only a subset of records. And for this, let me show you the search condition. I was requested to delete all comments that are newer than a specific date, the first of December of 2017. So I do DELETE

FROM Comments WHERE PostId = and then the post with the higher number of comments. Here all comments will be deleted for this post, but I use then an AND and provide another condition. Just like before, you can use multiple conditions with AND or OR and even you can use NOT. In my case, it is CreationDate greater than 2017-12-01. And please notice that I am providing a string. SQL Server is really good about this. In some cases, it's not necessary to be strict and do a cast. Now, I execute, and 20 rows affected. This means that I am left with four rows. Let me just confirm that this is the case, and indeed, by running the SELECT, I get back four rows, and that is how you use DELETE to remove multiple records from your database. Let's keep moving forward.

Deleting Data Using JOINs

In a great deal of cases, you delete rows based on a known condition. And by known condition, I mean because you know the ID of the rows or IDs if it is multiple rows or you delete by values in a column like a date range or something similar. The point being that you are quite specific on what you will be deleting. But there's another scenario that's quite useful in large part because you know what you want to delete, but you perform the DELETE based on a condition that uses data from another table. What I mean is that you use data from another table to create a condition based on the relationship between these two tables. For this, we use a JOIN. Let me show you. Demo, removing rows using a condition based on data from another table, which, of course, in fewer words means that we are going to be using a JOIN. Let me type in this query, SELECT Count(Id) as CommentCount FROM Comments. And then this other one, SELECT Count(Id) as NegativeScorePosts FROM Posts WHERE Score is less than 0. In total, we have 37, 000 comments, which is a number that I will leave right here on the right as I will use it later. Then there's a few hundred NegativeScorePosts. Yes, if you are down voted, then your score will be less than 0. Not nice, but useful to the community to know which posts are not the best. Next, I'm going to write this DELETE statement, which starts in the same way as before, DELETE FROM Comments. That's pretty standard. And then here's something new, FROM Comments, again. Yes, this is not an error, then JOIN Posts, this is the other table that we will use for specifying a condition ON Comments.PostId = Posts.Id. This is how we specify how the rows and comments are related to the other rows in posts, that is the relationship between comments and posts. And then a WHERE, which is where we specified that the score is less than 0. I execute, and 755 rows have been deleted. Let me put it here on the left. We now reexecute the first query. We get back 36771. If we do this subtraction, we get that this is correct. We have deleted multiple records in one table based on a condition that took into account data from another table. That is quite nice as we do not need to know in advance which records we want to delete, but instead, we delete records based on rules that we can specify using the expressions in SQL. That's quite nice. Now let's see what else we have.

Removing Data with Constraints and Cascade Deleting of Data

Another scenario that I want to cover is how DELETE is not always possible, at least at first, that is you run the statement and sometimes you may get an error. The question is why? Well, maybe you had a constraint like a foreign key, that is the row that you are trying to delete is being referenced by rows in another table. For cases like this, you can use something that is called DELETE cascade of data, that is you remove the row being referenced and then the rows that are referencing this row are automatically deleted. Of course, you've got to be very careful in this scenario because you're explicitly deleting one row, but implicitly you're potentially deleting many more rows. And before getting into the demo, I just want to mention that for DELETE cascade to work, you need to specify when you're either creating the table or by performing an alter table, that is using the Data Definition Language, or DDL. It goes a little bit beyond the DML. Now, let me show you the demo where I will cover some DDL statements before covering the DELETE statement from the DML with cascade deleting of data. Demo, removing data with constraints and cascade deleting of data. Okay, so for this one I will start in a similar way as with a few of the previous demos where I created a couple of extra tables. In this case, I will create a copy of the Posts table. I'll call it PostsCopy and then the related table, CommentsCopy. There is just one difference on this statement, which if I'm allowed, let me direct your attention right here. In the constraint, there is something extra, ON DELETE CASCADE, which tells SQL Server that if a record is deleted where it is a foreign key constraint on another table that it should then delete first the records in the other table that reference this one before deleting the record itself. And this is what I'm going to show you now. I execute, and I have created both tables. All good. This is something that I can confirm by looking at the Object Explorer. I can open it in the usual way via the View menu, and you might have to refresh, but here are both tables. Anyway, let me close this. The next step is to insert first the posts. We know quite well how to do this. I am executing an INSERT with a SELECT. Please notice that IDENTITY_INSERT is set to ON. As I want to preserve the primary key, this is necessary for the foreign key in the comments to reference the correct post. Then set the identity to OFF once the data has been added, and finally, copy all the comments. And I will execute in a second as I just want to show you something that I have not used in this training yet. Next to the Execute button, there's a checkmark that is used to parse the command, that is to make sure that it has the correct syntax. It does not check the execution, only the syntax, but it might come in handy sometimes. Click it, and it displays a message saying Commands completed successfully, which means that the syntax is correct. But then I execute, and I have the data that I need to show you this demo. Then off to the next tab where I have this query that retrieves the top post with the highest number of comments. It is PostId 30430 with 23 comments. Now if I take this PostId and use it for this next query to get back all comments, yes, there they are, and now my task is the following, DELETE the post with Id 30430, yes, this one that has 23 comments. And as expected, this won't work because I would be violating a constraint. The full message is The DELETE statement conflicted with the REFERENCE constraint "FK_POSTID_COMMENTS", and so on and so forth. Here, notice that I am performing this statement against Posts, the original table, not the one that I just created. I can't delete this post because it does not have the DELETE cascade. If I was able to delete this post, all these comments would not have a parent post, and that violates the constraint. So what can I do? Well, there are several things that are possible. For example, I could first delete all comments that reference this post and then I can safely delete the post, but that operation involves two statements, so if something happened, we might delete the comments, but not the posts. Something we will learn how to deal with when we get to transactions, but for now, I'll show you a different way with one statement. Here are my comments and the posts for 30430. And note this that these are the copy tables. This is the before. Then I delete the post only, and I get that 1 row affected message. Yes, one row has been deleted from Posts. One thing that's important is that I'm not getting a notification for comments. There is no additional message of 23 rows have been deleted from comments, so this is something that you need to watch out for. Now if I reexecute the two statements above, I can confirm that both the posts and the comments from the copy tables have been deleted. That DELETE cascade worked as expected. When I delete a post, all comments that reference this post are deleted as well. That's great. Let's keep moving forward and see what else we have to remove data from a database.

Clearing Tables Using TRUNCATE and How It Differs from DELETE

Next, I am going to cover another statement that is used to remove data, which is similar to DELETE, but with a few differences that are worth noting, the TRUNCATE statement, which goes something like this, TRUNCATE TABLE table_name WITH (PARTITIONS), and then the (partition_number_expression). I said similar to DELETE as it removes rows. Well, it removes all rows, but in reality, a TRUNCATE statement is more related to table partitions, but we are not covering partitions here, so I want to focus on the functionality of TRUNCATE that works like a DELETE without a WHERE, removing all data from a table, but with the advantage that it's faster than DELETE and uses fewer system and transaction log resources. So, it is TRUNCATE TABLE and then the name of your table, and that should be enough to remove all records from a table quickly, efficiently, and resetting the identity columns, something that I will show you in a few minutes. And then the second part, WITH PARTITIONS, which is used to remove rows from a specific partition, although you can also provide, as argument, multiple partitions. It is worth mentioning that TRUNCATE TABLE removes all rows from a table, but that the table structure and its columns, constraints, indexes, and so on remain as is. To remove the table definition in addition to its data, you should use the DROP TABLE statement from the Data Definition Language, the DDL. And just to compare DELETE verses TRUNCATE, here are a few differences that I want to mention. First of all, TRUNCATE has better performance because it does not log any deletion. DELETE is slower because it has to log every deleted row, but doing this also allows DELETE to roll back in a transaction while TRUNCATE can't. Another difference is the fact that TRUNCATE does not support the use of WHERE clause while DELETE does. Then after performing a deletion, some triggers can be fired, but removing data using TRUNCATE won't trigger anything. TRUNCATE should be used when we want to completely clear a table and reset every identity column while performing any number of DELETE queries keeps the identity column untouched. And finally, while DELETE is a DML statement, what we focused on this course, TRUNCATE is a DDL statement. But even though it is from the DDL, the Data Definition Language, I wanted to cover it as it can be used sometimes instead of a DELETE. It's also worth noting that TRUNCATE is not possible when a table is referenced by a foreign key or tables used in replication or with index views or when it participates in an index materialized view or was published by using transactional merge replication. But having said that, now it is time for a demo. Demo, clearing tables using TRUNCATE and how it differs from DELETE. Okay, now for this demo, we are going to start by removing both CommentsCopy and PostsCopy from our database. For this I use the DDL command DROP. Why am I doing this? Because I want to show you a demo where I start with freshly created tables. Okay, after removing the tables, I will create them both and then populate them. I'll click on Execute and 30, 000 in changed comments and posts are inserted into my database. Next, I will check how many comments are in the CommentsCopy table, as well as what is the MaxIdentity. If I execute, I can see that both numbers match. This is why I created the new tables. It is easier to remember. I'll leave this number right here on the right, 36771. And then the all too powerful query, DELETE FROM CommentsCopy, which if I execute, all records are removed from this table, 36771 rows affected. Again, be careful. It's not likely you can press Ctrl+Z and undo. I quickly run a SELECT * FROM CommentsCopy, and indeed, 0 rows are returned. I have an empty CommentsCopy table. So now I go into the next tab where I have an INSERT statement ready. I am going to add one record, and as a reminder, I have IDENTITY_INSERT set to OFF. And nowhere in this INSERT I am providing the ID. SQL Server provides it automatically. So I execute, and I get that 1 row affected message. Then I change back to the previous tab, and I run the Select * FROM CommentsCopy query. I can see that this table has only one record, the one that I just inserted, which if I look right here, I can see that the Id is 36772. That's one more than the MaxIdentity that the table had earlier. This is because when you run a DELETE on the entire table, the identity column will use the next value and INSERT. In layman terms, the counter used for the identity column is left as is and incremented the next time it is required. But, and let me speed this up as you already know what I'm about to do, let me go back to the first tab, drop the copy tables, create them again, and populate them. Then I go to the second tab and execute both SELECTS, the Count and MaxId match again. But instead of using DELETE, I will use this, TRUNCATE TABLE, and notice that it's not necessary to have the FROM. I execute, and I get a good message, Commands completed successfully. I check and indeed, the table is empty. I move to the next tab and insert a record. This is exactly the same process that we did before with the delete of all records from the table. Now I go back to the previous tab, and I execute the SELECT. And this is what I wanted to show you. The identity column value has been reset, so the new Id is set to 1. This is one of the most noticeable differences when you're using TRUNCATE TABLE verses DELETE FROM. On a larger table, you would also notice the performance, as well as the impact on the logs. And now let's do the takeaway so that we can move on into other interesting topics.

Takeaway

In this module, we learned how we can remove records using the DELETE statement, although I'd recommend that you are careful when deleting data. In some cases, you could do a soft delete, that is mark a record as deleted instead or removing the data right away. Although if you prefer to delete it, go ahead. Then we went right into the DELETE statement where we learned how to delete one record or even multiple records or even delete records based on a condition that is relative to data from another table, which meant we had to use a JOIN. Then we moved on into how to remove data with constraints, for example, when rows in another table reference a key in the table where we wanted to remove the data on that particular row, namely a foreign key. In this case, we learned how cascade DELETE worked, although for cascade DELETE, we needed some help from the DDL, the Data Definition Language, to specify where the cascade operation should take place. And finally, we talked about TRUNCATE, which is not a DML statement. It is a DDL statement, but I wanted to mention it given that it can be used to remove all records from a table like a DELETE, but with a slight difference that it resets the identity column. And now let's move on into the next module where we will cover a topic that is extremely useful, transactions.

Maintaining Data Integrity with Transactions

Maintaining Data Integrity with Transactions

It goes without saying how important data integrity is. At no point, at least that I can think of right now, is desirable to leave our database in an inconsistent state. So, what do we do? Well, we maintain data integrity using transactions. And what exactly is a transaction? Well, in a nutshell, a transaction is a set of statements that are performed so that they all are guaranteed to succeed or fail as a single unit, as simple as that. It is an all or nothing. And, again, why is this important? Well, let's picture this scenario. You are paying your credit card using your bank's online application. It could be their mobile app as well. And the application starts by deducting the money from your savings account to pay the credit card balance. Then the application quits unexpectedly right before paying the card, but the payment was not yet applied, so you don't have the money anymore, but you are still in debt. So the big question is where did the money go? And yes, with an interrobang. If you're not familiar, the interrobang is the question mark and exclamation marks. It's when you're really surprised. Well, this is just one scenario, but there are many more where it's very important that it all fails or succeeds at once like deducting an item from an inventory on a sale, but failing to initiate the shipping of the order. Well, if this happens, you may fail to deliver to a customer or lose an article that you may not be able to sell later on. And in here, you lose money. But aside from money, there are all kinds of problems that can arise from inconsistent data in a database, but that's not all. As mentioned, the primary benefit of transactions is data integrity and this is even of more importance when there is a high level of concurrency. But a secondary benefit of using transactions is speed. There is often an overhead associated with the process of committing data to the database. So if you're inserting a large number of rows, if you save your changes on every row, then you may end up paying a performance penalty compared to committing once after all the inserts or even updates are completed. So now let's learn in this module how we can use transactions, namely single units of work to leave our data in a consistent state every time.

Transaction Properties

Before getting into the details of how transactions work, let me talk about the properties of a transaction, which is something that is key to understanding the big picture. There are four standard properties that transactions have. First, atomicity, which ensures that all operations within the org unit are completed successfully, otherwise, the transaction is aborted when an error occurs and previous operations are rolled back to their original state. Then we move on into consistency, which ensures that database changes are guaranteed up in a successfully committed transaction, that is no transaction is left in a half-finished state. Also, any changes should comply with any constraints placed on the data, for example, on the foreign key or particular value. Then we have isolation. Each transaction has a defined boundary. They operate independently of and transparent to each other. While one transaction is running, another transaction can only see the data affected by the first transaction either in its original state or the resulting state, but not in the intermediate state of execution, and then durability, which means that the data modified by a successful transaction persists regardless of other external factors. These properties are usually referred to by the acronym ACID, and every database software that offers support for transactions enforces this for ACID properties automatically, and Microsoft SQL Server is no different. And now let's move on into understanding T- SQL transactions.

Understanding T-SQL Transactions

As I just said, a transaction is a single logical unit of work that can be composed of one or several T-SQL statements. The transaction starts when the first T-SQL statement is executed, and it ends when the changes to the database are saved or discarded. In this diagram, we can see the basic flow of a transaction. It starts by executing a couple of SQL statements, which will make one or more changes in the data stored in our database. Once execution is done, the database engine checks the integrity of the database and if the information is consistent, then it saves the changes, but if it's not consistent or an error occurred, then the changes are reverted and all changes are discarded. Also, a transaction can be divided into several steps, which is what we call savepoints. Where we create them, we then execute a few SQL statements, and if there's an issue, we can revert our changes, but up until the savepoint, that is the previous SQL statements are not discarded. And there's also the possibility of having nested transactions, and there are several transaction modes. First of all, we have autocommit transactions where each individual statement is a transaction. That's pretty much what we've been using up until now. We saw how even within a single statement if there is an error, then the changes are rolled back. Remember that a single statement can modify multiple records, so if one error occurs, then all changes are discarded. Next, we have explicit transactions where each transaction is explicitly started with a BEGIN TRANSACTION statement and explicitly ended with a COMMIT or ROLLBACK statement. And finally, implicit transactions where a new transaction is implicitly started when the prior transaction completes, but each transaction is explicitly completed with a COMMIT or ROLLBACK statement. And finally, there is just one more case, the batch-scoped transactions, which only works with something called MARS, or Multiple Active Result Sets, which is a feature that allows the execution of multiple batches on a single connection. Now let's see some of the keywords that we use to work with transactions. And here they are. In order to control the flow of the transaction, T-SQL provides the following keywords, BEGIN transaction, which is used to declare a new transaction, then SAVE, which is used to create checkpoints between multiple T-SQL statements. We can come back to a particular savepoint and continue instead of losing all the work with it, which would happen if we discarded all statements because of one error, then ROLLBACK, which reverts the changes to a specific and previously created savepoint or for the entire transaction, then COMMIT, which saves the changes to the database, and then SET transaction, which starts the execution of a transaction. Let's now create a transaction.

Creating Transactions Using BEGIN and COMMIT

Let's see how we create a transaction. Here is a syntax. There are three main components, BEGIN TRAN, or alternatively, you can use the whole word TRANSACTION, then the transaction_name or transaction_name_variable. Finally, WITH MARK, and a transaction description. The BEGIN TRAN or BEGIN TRANSACTION marks the starting point of an explicit transaction. It increments at that TRAN count by 1 with @@TRANCOUNT being a variable that keeps the number of BEGIN TRANSACTION statements that have occurred in the current connection. Then you give a name to the transaction, which should follow the rules for identifiers in SQL Server. There is a limit of 32 characters, and only use a name in the outer-most transaction. Also, it is case sensitive, always. And finally, WITH MARK, which places a transaction name in the transaction log, which can be used for recovery purposes. And that is mostly it. That's how you start a transaction, but as usual, let's see this in action to understand it better. Demo, creating transactions using BEGIN TRAN. Okay, so for this demo, let me prepare a few queries. The first one is SELECT * FROM Users WHERE DisplayName = 'xavier-morera'. Okay. I'll execute this, and here's the result. And this is the number that I'm looking for. This is my ID, 75298. Now let's see how many posts I have created by using this query, SELECT * FROM Posts WHERE OwnerUserId =, and I'll use my Id, 75298. Now, I execute, and check this out. I get 0 results back. It turns out I have created no posts at all, but since this is my database and this is a demo, I'm going to look for the user with the highest number of posts and kindly donate these posts to me. This is just a test, right? Don't do this on a production database. So the query I will use is SELECT OwnerUserId, Count(OwnerUserId) AS TotalPosts FROM Posts GROUP BY OwnerUserId ORDER BY TotalPosts DESC. I will execute, and I see that user with ID 836 has 384 posts. Now what I'm going to do is to move all of these posts to me, and then I'm going to do the same for comments. So I will create a transaction that updates both these tables. But if something goes wrong, all the changes are discarded. And that's the beauty of transactions. It's an all or nothing, but leaving the database in a consistent state. So we start the transaction with BEGIN TRANSACTION, and that's it. You can also give it a name, but it is optional. And then this statement to UPDATE Posts. I'll SET the OwnerUserId to my ID and OwnerDisplayName to 'xavier-morera'. And this is important, WHERE OwnerUserId = 836. Don't forget the WHERE, else you'll update the entire table, but we already talked about this. Then the second statement for the comments, UPDATE Comments SET UserId = 75298 UserDisplayName = 'xavier-morera' WHERE UserId = 836. Okay, those are the two statements for this transaction. Now what? And please notice that I have not yet executed these statements. Well, I'll show you next. And the now what is to save your changes by using COMMIT TRANSACTION. The syntax is COMMIT TRAN or COMMIT TRANSACTION, then, optionally, the transaction_name or transaction_name_variable and another optional argument WITH DELAYED DURABILITY ON or OFF. The first part, the COMMIT TRAN or COMMIT TRANSACTION is used to mark the end of the transaction. If @@TRANCOUNT is 1, then COMMIT TRANSACTION makes all data modifications since the start of the transaction permanent in the database. It frees the transaction resources and decrements @@TRANCOUNT to 0. When @@TRANCOUNT is greater than 1, COMMIT TRANSACTION decrements its value by 1, and the transaction stays active. Oh, and by the way, you can also do COMMIT WORK, which is equivalent to COMMIT TRANSACTION except that it does not accept a name. Transaction_name specifies a transaction name assigned by a previous BEGIN TRANSACTION and delayed durability requests for this transaction to be committed with delayed durability. Fully durable transaction commits are synchronous while delayed durable are asynchronous. Their request is ignored if the database has been altered with delayed durability equals disable or forced. It is our responsibility to issue COMMIT TRANSACTION only at a point when all data referenced by the transaction is logically correct. Now, let's see how it works. Demo, making changes permanent with COMMIT TRANSACTION. Let me do COMMIT TRANSACTION. Yes, that should do the trick. This will make the changes in the database permanent, but as mentioned earlier, I have not yet executed anything, so I execute BEGIN TRANSACTION. At this point, I have created a new transaction in the database. Then I execute the first update, which if you recall, this creates a lock on these records that are being modified. Then I'll execute this other statement. Okay, that's good. And now, take a look at this. Let me change to the other tab and execute this SELECT statement, which should retrieve the records that I just modified, those records that are involved in this transaction. And please notice how I am not getting any results back. You can see at the bottom how it's stuck in Executing Query because a lock has been placed here, even the transaction. So for now, I'll just cancel the query. And this is the reason why you should not leave your transactions open for a long time because you can affect other queries that might have to wait until your transaction has completed, which is what I'm going to do next. I am going to complete my transaction, but before going to the other tab, I'll just execute this query again and just leave it running. The moment that I commit the transaction, this query will complete immediately. I will leave it like this and change to the previous tab and will execute the COMMIT TRANSACTION. Now my work has been saved. Both comments and posts from user 836 belong to me now, which if I change to the other tab where I left the query running, I can see how the query has completed. And if I scroll to the right, I can confirm that both OwnerUserId and OwnerDisplayName reference my user. And that's how you work with a transaction in SQL Server. Just a reminder, please note that you can have nested transactions. When you start a new transaction using BEGIN TRAN, changes made to the data need to be committed for every transaction that you created, and changes are persistent in the database for everyone only when you commit the top-level transaction. Now, let's keep moving forward so that I can show you something that might be useful.

Querying Data Locked in a Transaction with NOLOCK

Now I am going to talk about something that falls a bit out of the scope of the Data Manipulation Language, the use of the NOLOCK hint or querying. And yes, while this is a DML training, I really think it is important to talk about this topic because it can be quite useful to know about it while you are working with transactions, so here it goes. In the previous demo, we saw how we can't read data that is currently locked because there is a transaction open at the moment that involves the data that I would like to query, and this is important because we want to guarantee that the results of our queries are consistent. But what if we want to be able to read the data anyway regardless of whether the transaction has not completed either successfully or has been rolled back? And for this, we use the NOLOCK hint, which retrieves records regardless of locks. This is what can be called a dirty read, and there is a slight possibility that you might be retrieving records that could be out of sync, although there is one advantage as it does not take into account locks, it is faster. You can potentially get better performance. The NOLOCK is equivalent to READUNCOMMITTED. In fewer words, you are setting the transaction isolation level. Oh, and this only works for SELECT. You can't use this hint with UPDATE, DELETE, or INSERT. Let me show you how it works. Demo, querying data locked by a transaction using NOLOCK. For this demo, we are going to reuse the statements from the previous demo. Here I have two tabs. In this one, I have a section that updates both posts and comments, and the other tab, which I'll use in a bit, has the SELECT that queries the data for this table. Step one, let's start by executing BEGIN TRANSACTION. Good. We have an ongoing transaction. Next, I'll execute the two UPDATES, and I can see it changes quite a bit of rows in both tables. At this point, there should be a lock in place, but should is not the same as is, so let's confirm that indeed there is a lock because of this transaction by running a query. So, I'll take the SELECT statement and run the query. And indeed, I don't get a result yet because the query is waiting for the transaction to finish, so I will cancel this query like this, and then I will add the hint. Let me just make this statement span a few lines and add here, WITH (NOLOCK). And now when I reexecute, I get all rows returned right away. I was able to read the data regardless of the lock that was put in place by the transaction. This is one of those steps that's really useful to know for certain cases where there are transactions running. Again, this is not strictly DML, but it is good to know for when you are working with transactions. Now go ahead and commit this transaction as we don't want to leave a transaction open. And now let's go to the next demo.

Undoing Transactions Using ROLLBACK

The ROLLBACK transaction is a statement that returns the state of the data to a previous state. And by this, I mean to the start of the current transaction. The reasons of why to roll back are several, but it can be because of an error or a specific condition. You can roll back an entire transaction or just a specific part of one, namely when you use a savepoint, with savepoints being the topic that we're going to cover right after this one. Here's the syntax, ROLLBACK TRANSACTION and transaction_name or savepoint_name. The ROLLBACK TRANSACTION just indicates to the database to initiate the rollback. And the second part is what tells what to roll back be it either the transaction_name, which is optional or the savepoint_name, which rolls back until that particular savepoint. Let me show you with a demo, undoing transactions using ROLLBACK. Let me start a new transaction. This is something that you understand and know quite well to do by now. BEGIN TRANSACTION insert_another_user. This is the name for this transaction. Now I am going to insert a user into a users table in these three columns, DisplayName, Location, and AboutMe. The values are going to be another-sql- user, Costa Rica, and this user will never exist because indeed it won't as it will roll back that transaction before committing it. Okay, so I start the transaction by executing the BEGIN TRANSACTION. Commands completed successfully. Now I insert the user. I execute, and look at this, 1 row affected. This means that one row has been inserted into the users table; however, given that right now we have not yet committed the transaction, this change is not visible to other users of the database that are querying this table. So let's say that for whatever reason I now need to roll back the transaction. So I will use this statement, ROLLBACK TRAN and the name of the transaction. Let me execute this statement, and indeed, Commands completed successfully. So now let's confirm. If I run this SELECT statement, SELECT * FROM Users WHERE DisplayName = 'another-sql-user', then I get 0 rows returned even though I had a 1 row affected message earlier. This means that the INSERT, even though it happened, it was never committed because I rolled back the transaction by using ROLLBACK. I have discarded the changes that I made earlier, and now let's move on into a related topic, savepoints.

Partially Undoing Transactions Using Savepoints

The next topic that I want to cover is how to partially undo transactions by using savepoints. A savepoint lets you roll back a transaction, but only up to a specific point within that transaction instead of rolling back the full transaction. It is quite useful when there is a certain possibility of error in a transaction at some point, but where the steps that are previous to this particular statement are too costly to simply rerun. The statement that we use is SAVE TRANSACTION. And this is the syntax, SAVE TRAN or SAVE TRANSACTION and the savepoint_name or savepoint_variable. Where the first argument simply creates the savepoint and the second one is used to specify the savepoint name or the variable you use. Let me show you with a demo, partially undoing transactions using savepoints. Here is the scenario for this demo. And, in this case, I want to explain it as it has a few more steps than usual, including one where we discard an intermediate step. The task at hand is that I have to delete all comments that are tagged to a particular technology. And given that deleting data is an operation where I need to be careful, I am going to use a transaction, so BEGIN TRAN. Then I DELETE all Python comments, and now I create a savepoint, which I do using SAVE TRAN, and then I give it a name, del_python. This is my first savepoint. Next, I will delete the Java-related comments and then create another savepoint, which I'll call del_java, but here, I realized that I needed to delete a different set of comments, namely those tagged with C++. So I roll back to the first savepoint, del_python. Please notice that even though I had newer savepoints, I rolled back to this specific one, discarding the changes in the database related to the deletion of Java comments. Now I can delete the C++ comments and finally commit the transaction. The end result is that I have deleted Python and C++ comments and that the deletion of Java comments was discarded. Now, let me show you how it works. So I start a transaction using BEGIN TRAN and execute. Commands completed successfully, then DELETE FROM Comments WHERE Comments.Text LIKE '%python%'. I select the statement and execute it, 878 rows affected. I have removed all comments that are related to Python. Next, I create a savepoint by using SAVE TRAN, giving it a name, del_python. This is important, as I mentioned, because if we have multiple savepoints, we will be able to tell them apart, which is what I will do in this demo. Next, I copy the part of the DELETE statement from above just that I will delete comments that are related to Java like this. Now, I'll execute the statement, and I get 71 rows affected. Next, I will create another savepoint, del_java. In this demo, I won't use the savepoint, however, I wanted to include multiple savepoints because I wanted to make it clear that this is possible and that you can roll back to another savepoint and not just the last one. Now I will run another query, SELECT * FROM Comments WHERE Comments.Text LIKE '%python%' or Comments.Text LIKE '%java%'. Indeed, when I execute, there are no rows returned, but let's imagine that at this point I realize that I needed to delete Python and C++ instead of Java. By replacing Java with C++, I can see that there are quite a few records that reference C++, 22 in total to be precise. So, and let me make up a little bit of room in my query editor, now I will execute ROLLBACK TRAN del_python. Notice that I'm specifying the savepoint. I am discarding all changes up until that savepoint. This means that those Java records that I deleted earlier are now available again. I have discarded one DELETE operation, so now I can copy a part of the DELETE statement above just that now I will delete all comments where the text is LIKE C++. I execute, and I get 24 records affected. Now if I copy this statement above that checks for comments around Python and C++, when I execute it, I can see that there are no records returned because I deleted all of them. Also, I don't know if you have noticed, but if I run SELECT statements within the transaction, it does show me changes made to the database. The changes that I make in this transaction are isolated to my current transaction. But to make them permanent, I need to COMMIT TRAN. That's good. And now let me check to see if I still have the Java comments using this SELECT statement. And indeed, they are still there. But as I showed you earlier, and let me do this after the COMMIT TRAN, indeed, all Java and C++ comments have been removed, and that is how savepoints are used to partially undo transactions. Let's now keep moving forward.

Distributed Transactions at a High Level

A distributed transaction is an operation that can be run from multiple different SQL Servers, that is multiple databases in the network where these operations retain the ACID properties of transaction processing. Data is retained in a consistent state even in multiple different machines. A distributed transaction is managed by the Microsoft Distributed Transaction Coordinator, also known as MS DTC. And even if there aren't multiple servers, the transaction always starts in one server, and this server is called the transaction originator. It controls the completion of the transaction. Any subsequent COMMIT or ROLLBACK statements are sent to the controlling instance, which requests that MS DTC manages the completion of the distributed transaction across all of the instances involved. The syntax is quite similar to a regular transaction, BEGIN DISTRIBUTED TRANSACTION or TRAN and the transaction_name or variable. The first part is used to specify the starting point of a distributed transaction and the second one for the transaction_name or variable. Distributed transactions is a topic that requires multiple different databases to be configured to work together, and thus, I only mentioned this topic at a high level. And now let's do the takeaway for this module to move on into more advanced topics.

Takeaway

Let's summarize what we learned in this module. We covered transactions where we have multiple statements that they all either succeed or fail, but as a single unit, which is extremely important to maintain data integrity while it also can help with speed. We also learned how transactions have four properties, atomicity, consistency, isolation, and durability, which we can remember as the ACID properties. Then we learned that there are several types of transactions, namely, autocommit, explicit, and implicit with explicit transactions being what we covered in detail in this module. We learned about the multiple statements that we use for transactions, which are BEGIN to start a transaction, COMMIT to save the changes and make them available for everyone, ROLLBACK to discard changes, which can be all changes or only a specific set of changes since a particular point, that is if we created a savepoint, which is done using the SAVE statement. We also briefly mentioned that it is possible to have transactions work across multiple databases, which is what we know as distributed transactions. And now let's move on into the next module where we will cover some more advanced topics.

Using Advanced T-SQL Techniques

Using Advanced T-SQL Techniques

Let's do a quick recap. So far, we learned what the Data Manipulation Language is, learning how to add data, modify it, and finally how to remove it from the database. I covered the most basic scenarios for INSERT, UPDATE, and DELETE, which if you take into account are a little slow and ae adapted to SQL, it means that about 80% of what you need to do can be done with about 20% of the statement. And now in this module, I will talk about a few scenarios that are a bit more advanced while still being reasonably common. Oh, and by advanced, I don't mean complex. What I mean is that they fall out a bit of the main scenarios that you use. Let's begin.

Using INSERT with Multiple Values

Here's a scenario that can be quite useful from time to time. In the previous module, we learned how to INSERT records providing a list of columns, which is optional, hence the square brackets that I'm using here, and then the values that I want to insert. But we did this for only one record; however, there are times when you want to INSERT more than one record, and you want to do it in a single statement because you can always use multiple statements and execute all of them within the same batch. But to INSERT multiple records in a single statement, you provide multiple values separated by a comma. Let me show you. Demo, using INSERT with multiple values. Let me run this query, SELECT * FROM Comments WHERE PostId =10, with 10 being a particular post that I have selected because it has one comment, which is good for this particular demo. So let me go to the next tab, and in here, I have an INSERT statement that I could use to INSERT one record. As I said, one, which would have as text this one comment, and all the rest of the values are enclosed in this list right here. But what if instead of adding a single comment, I wanted to add multiple? Well, here's how it works. Let me add a comma, and now I can copy this first list of values and paste. I will now modify the text so that it says this is a second comment. Then I can paste a third list of values, and the text will be even a third. So far, I have three lists of values, but I only have one list of columns, and now I have these three records ready to be inserted. I will click on Execute, and I get the message that I'm expecting, 3 rows affected. I have added three records using a single statement. Let me just confirm. For this, I can go to the previous tab, which still has the previous result set with one record. I will click on Execute, and now I can see that three new records have been inserted, and that's how you insert multiple records using a single statement. Let's see what else we have.

SELECT within an INSERT Statement

It is quite typical that we provide those values that we want to insert in one or many records, correct? But there's one tiny detail, which is that we need to know up front the values, which is not always possible or sometimes even practical; however, you can use a SELECT statement to retrieve values from a table and use them in the statement. A good case would be to populate the tables, that is insert records from data already available in the database, which is what I will show you next. So that you know, what I'm about to show you can also be used with UPDATE and DELETE. Demo, using SELECT within an INSERT statement. So here's the scenario for this demo. My job is to find those users of Stack Overflow that have a high reputation, let's say more than 5, 000, and hire them as employees of a company. Getting one by one all users is probably going to take a while, so what I will do is create a new table called Employees and use a SELECT to insert all employees into this new table with a single statement. So, let me then create a query that will return those high-reputation users, SELECT Id, Reputation, CreationDate, DisplayName FROM Users WHERE Users.Reputation is higher than 5000. I execute, and these are my users that I am about to convert to employees. I still don't have the Employees table, something that I can confirm by running this query, SELECT Distinct TABLE_NAME FROM information_schema.TABLES, and yes, indeed, there is no Employees table, so I create it using this DDL statement. Let me execute it. Commands completed successfully. Now I check again. For this, let me execute this SELECT, and there it is, the Employees table. Now I will move on to the next step where I have ready an INSERT statement to add records to the Employees table. It has the list of columns, but it does not have the values. Instead, I left an ellipsis here as a placeholder to know that here is where I would place the values that I want to insert, which, of course, you could add the values one by one, copying them from the other table, although manually inserting each record would be like printing a Word document, giving it to the person sitting next to you for them to type it in. And I kid you not, I once saw this in college, but instead, let me copy this full query and change Tab, and now I will select the ellipsis and paste the SELECT statement. I'll just adjust the indentation to make it look a little bit nicer. And there is one column in the Employees table that is not available in the Users table, the salary, which at the moment I will add a scalar value, 0. And I did this to show you that you can include columns, scalars, or expressions. Now I will execute the IDENTITY_INSERT for the Employees table as I want the employees to have the same ID. This is a design choice. You could also have used a foreign key. And there you go, Commands completed successfully. Then the INSERT with 10 rows affected, and finally, SET IDENTITY_INSERT Employees OFF. Commands completed successfully once more. Now onto the last tab, and SELECT * FROM Employees, and I get back all users with a high reputation whom are now part of the Employees table. Nice. We've inserted multiple records into a table using a SELECT statement, which saves us a lot of work on data entry. Let's now keep moving forward.

Using SELECT to Retrieve a Specific Field in an INSERT

It is quite common that when you add data to your database that you INSERT multiple fields. In such scenario you can provide all field values, you know, you can type them all in, but there's another way. You can retrieve one field based on another one. How? Well, you can use a SELECT statement just like we did in the previous demo; however, in the previous demo, we got a list of fields that we used to insert into the new table, but in this case, we do so for a single field. Why? Well, first of all, it has the advantage that it's less error prone, and second, it's quite convenient. Let me show you with a demo, using SELECT to retrieve a specific field in an INSERT. Let me run this query, SELECT Id FROM Posts WHERE Title = 'Parallel and distributed computing'. I execute and I get back the Id for this post, 81. Now, I'll find the ID for my user with this statement, SELECT Id FROM Users WHERE DisplayName = 'xavier-morera'. And there it is, 75298. Let me now make up a little bit of room at the top, and now this, INSERT INTO Comments. The columns will be CreationDate, PostId, Score, Text, UserDisplayName, and UserId. And the values are GETDATE or CREATIONDATE, and now I want to add PostId, which I still have in my results, but if I didn't or if it was an easy-to-miss type value, then I could just copy the SELECT statement and include it as this column value. And please notice the parenthesis. Then I will add the Score. A 0 would do. And the Text, which is I added a comment using a SELECT, and the UserDisplayName, which is my user. And I will reuse this SELECT statement that I have below, which is what I can use to get the UserId. And talking about UserId, please observe how I have my UserDisplayName in both of these values. If you had a batch or a stored procedure, you could use a variable, and that way you can get right both values from a single param or variable. In this case, assume a UserDisplayName is unique, which can be enforced, and, of course, don't forget to close the final parenthesis, then execute, 1 row affected. And just to make sure, I will run this query, SELECT * FROM Comments WHERE UserDisplayName = 'xavier-morera', and then as it was the last comment, ORDER By CreationDate DESC. And there it is, my latest comment, the one that I just inserted with a matching UserId. And that's how you insert records and provide values for one field using a query based on another field. Good. Let's move on into the next demo.

Retrieving Records on INSERT Using the OUTPUT Clause

By now we know quite well how to add data to the database using INSERT, but something that we also know quite well is that not all values are known when you're adding the data. We could have auto assigned values, calculated fields, or even default values. And it may be the case that we need to know them on execution right away, for example, to provide feedback to a user or to initiate some other process. So what can we do in a case like this? Well, we can use the OUTPUT clause to retrieve the values of these columns. Let me show you. Demo, retrieving records on INSERT using the OUTPUT clause. Let's start with the INSERT as we know it. INSERT INTO Posts, then the columns, PostTypeId, Body, OwnerUserId, OwnerDisplayName, and Title, then the VALUES. This is quite standard, 1, This is a new post, my OwnerUserId, my OwnerDisplayName, and the title. All normal up until now. Those values will be inserted, but what I don't know is what is the ID that will be assigned automatically by the database? That is the identity column. So, let's just do this. OUTPUT INSERTED.Id. INSERTED is to let the database know what's the type of the action, and then the column that you want retrieved, which is the autonumeric value, the primary key ID. Execute and instead of 1 row affected, SQL Server Management Studio returns the Id value, which is 53077. If you are executing this from an application, the result set would have this value as well, which you could read and use right away. And just to confirm, I will run SELECT * FROM Posts WHERE Id = 53077, then execute, and that is exactly the post that I just inserted. That's nice, right? But that is for INSERT. What about the other DML statements? Well, the good news is that OUTPUT also works with UPDATE. Demo, retrieving modified records on UPDATE using the OUTPUT clause. Here I have an UPDATE statement where I'm increasing the score of Posts by their FavoriteCount, if FavoriteCount is greater than 50. If I execute, I get 14 rows affected, but the question is which ones? Well, just add the OUTPUT clause, and what do you think goes next? Here's a small surprise. You use Inserted. Perhaps you were expecting Updated, but no, it is Inserted, and then the column name, for example, Id, but you are not limited to returning a single column. You can retrieve many. In this case, I will return the Title, then the Score, then the FavoriteCount. And finally, I'll execute, and I get back a result set that includes the new values after the UPDATE for the fields that I specified in the OUTPUT clause. Now I assume you know what I'm going to say next, right? If you guessed that it worked with DELETE, well, you were right. Demo, retrieving removed records on DELETE using the OUTPUT clause, and I will include JOIN and TOP. Indeed, instead of just showing you a demo with DELETE and an OUTPUT, I'll do a slightly more advanced DELETE. I'll first retrieve all those Users WHERE DisplayName is like '%darth%'. I'll execute the query, and there are quite a bit. I am going to delete the first three users that I find, but I want to make sure that these users do not have posts nor badges. Something like this. I don't want to violate a constraint. And the first couple of users that apply are kneelb4darth and Darthtrader. Let's just remember them for now. So I will start with DELETE TOP (3) FROM Users, and then the JOIN FROM Users LEFT JOIN Posts ON Users.Id = Posts.OwnerUserId LEFT JOIN Badges ON Users.Id = Badges.UserId WHERE DisplayName like '%darth%' and just a final couple of conditions to make sure that the OwnerUserId and UserId IS NULL. Then the OUTPUT, and what do you think goes next? Well, DELETED. *. That way we get back all fields. I will execute, and those are the three records that have been removed from the database. We knew about these first two, kneelb4darth and Darthtrader. They were included in the first query. If we now reexecute, we can no longer see them in our table. By using TOP and JOIN, we created a slightly more interesting DELETE with OUTPUT. That's good. Let's keep moving forward.

Utilizing MERGE to Perform Operations on a Target Table

Let's talk now about another command that can prove itself quite useful in certain scenarios, the MERGE command, which is used to synchronize tables. And by synchronizing tables, I don't mean like how we did with users and employees where we just copied over and inserted many rows using a SELECT statement. Instead, we synchronized tables that have a mixture of matching characteristics, that is it is required to run INSERT, UPDATE, and DELETE operations at the same time on a target table based on differences found in a source table. The MERGE statement works using two tables, the source table and target table. The target table is a table to be modified based on data contained within the source table. The two tables are compared by using a MERGE condition, which specifies how rows from the source table are matched to the target table. If you have used inner joins before, it is like the join condition used to match those rows. After performing a MERGE statement, there are three types of rows, matched, which are the rows that satisfy the matched condition. They are common to both the source and target tables. Then, not matched by target, which are the rows from the source table that didn't match any rows in the target table. And then not matched by source, which are the rows in the target table that were never matched by a source record. A MERGE statement can look something like this. You can have a MERGE target table using sourceTable ON mergeCondition and WHEN MATCHED, and then we can have a statement like THEN and an updateStatement WHEN NOT MATCHED BY TARGET, THEN you could have an insertStatement WHEN NOT MATCHED BY SOURCE THEN, and you can have another statement like a deleteStatement. But as usual, I think it's easier if I show you with a demo, synchronizing tables using the MERGE statement. This demo involves quite a few steps, so I have prepared a set of tabs to walk you through the process so that we can focus on what's important, the MERGE command. First of all, I'm going to create two tables, the CommentsSource and the CommentsTarget. Both tables are equivalent. They have the same columns. There you go. To confirm, I can open the Object Explorer and validate that they are there, CommentsSource and CommentsTarget. I will close the Object Explorer and move on into the next tab where I have an INSERT statement that takes all rows from comments and inserts them into CommentsSource, except those comments WHERE the Text is LIKE '%python%'. Please notice how I am using something new here. I am specifying which rows not to insert. I execute, and about 36, 000 rows have been inserted. Now I move on into the next tab, and in here, I will be inserting INTO CommentsTarget all rows except those that talk about Java. I execute, and about 37, 000 rows are inserted. The number is slightly different between both tables as Python tends to be a bit more popular than Java. Now move on into the next tab and execute this SELECT statement to confirm that the source table does not have any comments that mention Python. And indeed, this is correct. Then in the Target table, there should be no comments that mention Java. I execute 0 results. Again, this looks good. But if I check the Source table, I can confirm that there are Java-related comments in the Source table. This is what I was expecting, and there are Python-related comments in the Target table. So far, so good. Now I move on into the next tab, and here's my MERGE statement. It specifies which are the two tables that are going to be used on MERGE. Then the MERGE search condition, then a NOT MATCHED BY a TARGET clause specifying what to do, which in this case, it is INSERT, and what to do WHEN NOT MATCHED BY SOURCE. In this case, it is DELETE. Then I execute, and 930 rows affected. We have performed a specific synchronization scenario, but notice that this synchronization is based on a specific scenario and with certain rules. It is not a replication where both tables end up being exact copies, which I can confirm by going back to the previous step that CommentsSource should not have any Python records, but I did insert Java comments into the Target table, so I should get back plenty of records here, which is the case. Java-related comments should still be in the CommentsSource, but I deleted Python-related comments from the Target table. It all looks good. That's how you used a MERGE statement to synchronize tables by running INSERT, UPDATE, and DELETE statements on different scenarios. Let's keep moving forward.

BULK INSERT Using a Format File

There are many times when you need to copy data in large quantities. You could always create an application to do so; however, why would you want to reinvent the wheel? Instead, you should use a utility to import and export data, and not just any utility. You should use one that is already tried and tested. Perhaps one that is even installed by default with SQL

Server like the bulk copy program, which is also known as bcp, which lets you export and import data from a SQL Server database to text files and vice versa. With the bcp, you get the advantage that you can use a format file to specify how you want to import data that is available in a specific format so that you can map this data to your database. Let me show you with a demo, BULK INSERT using a format file with bcp. The documentation for the bcp utility just like the rest of the DML statements that we have been covering all along this training can be found in docs.microsoft .com. You can read it in detail if you want to explore which are all the arguments that you can use. But for now, let me show you how to use it. I am here in the command line inside the folder where I have the exercise files that I provided for this training. From here, I will execute bcp just to get a look of all possible parameters. And here they are. Now I will execute bcp. What I'm going to do now is I'm going to create a format file that I will use to specify how to import some data that I have available in another format file, which is called comments.dsv. At this moment, I'm going to specify that I'm going to create a format file with these arguments, tsql-dml, this is the database which is installed in this machine, dbo, and then the table, now the parameters, format nul, because I am exporting so I don't have to specify the path to the file. Then, -c and -f. The first one specifies that I'm using character data type and then the -f specifies the format, which is comments- format.fmt. Then -t, which is used to tell the field terminator, namely a comma. And -T, which means that it is a trusted connection, which is possible because the database server is installed in this same server. Enter, and you know what they say. No news is good news, well, at least sometimes like in this case because a format file has been created, one that looks like this. The format file has in the first column the position of a particular column in the input file. Then the second column has the data type in the input file, which, in our case, it's SQLCHAR, something we specified earlier, then prefix length and host file data length, then the terminator, server column order, and server column name. Finally, it has the collation. Once you generate a format file, which is what we just did with bcp, is what specified how the table looks in our database, you need to take a look at the file that you're trying to import and modify it so that you can provide the file that you want to import and the format file, then SQL Server will know how to import this file. When I check the file with the data, it looks like this one. As you can see, the fields are separated by a dollar sign; however, it's more common that they are separated by a comma. So what I do now is that I modify the terminator in my format file, and in general, I make any necessary changes like a reordering of columns or removing some columns. Now my format file maps directly to the data stored in comments.dsv. Then I will do BULK INSERT just like I did before with the CSV, but in this case, using the format file. So it is BULK INSERT Comments FROM 'F:\tsql\ comments.dsv '. This is the data file WITH FORMATFILE, and I specify 'F:\tsql\comments- format.fmt. SQL Server now knows how to import this data file. I execute, and 3 rows affected. So now I can confirm that these records have been inserted using the SELECT statement. Looking for comments WHERE UserDisplayName = 'xavier-morera' AND TEXT LIKE 'This is a comment%'. And indeed, there they are. And that's how you can use a format file to specify how the data that you want to import into your database is structured, letting you import in bulk with a little bit of help from the bulk copy tool, or bcp. And now let's do the takeaway for this module.

Takeaway

Let's now go over what we learned in this module. Earlier, we used the INSERT statement to add one record at a time, but we learned how it's also possible to insert multiple records by providing multiple values. Then, we learned how to populate one table based on a query, namely by using a SELECT statement, which has the advantage that it retrieves many records with multiple fields. That's quite convenient to add data in large quantities if needed. Then, we learned how we can also retrieve specific fields for an INSERT by using a query, which can be quite useful to obtain related fields. A search by ID being a frequent scenario and that this also works well with UPDATE and DELETE. Then we learned how to retrieve records affected using the OUTPUT clause, which works with INSERT and UPDATE using INSERTED., and the columns that you want to retrieve, but that there's no updated.clause, and then DELETE, which uses DELETED. and the columns. Next, we covered the MERGE clause, which is used to synchronize tables. And by this, I don't only mean copy data over, instead it uses a mix of INSERT, UPDATE, and DELETE clauses to synchronize only what's needed based on different scenarios, which includes when fields match in rows in both tables or when rows are not matched by target or when not matched by the source. And finally, we covered BULK INSERT for a scenario where the input files are not in the exact same format that our tables are where you can specify how your records are using a format file that can be generated with a bulk copy program, or bcp. And now let's do the final takeaway for this course.

Final Takeaway

Final Takeaway

It is now time to summarize what we learned in this training with the final takeaway. We began by learning that there are four different T-SQL sublanguages, the DQL, Data Query Language, which is used to retrieve data, DDL, the Data Definition Language, to define the structures in our database, and DCL, the Data Control Language to provide and control access to our database. And the star of our course, the DML, or Data Manipulation Language, which we use to INSERT, UPDATE, and DELETE data. We also took a few minutes to refresh what we needed to know to work with the DML, SELECT and WHERE, with SELECT being potentially the most common statement that you will run in your database, but to be able to query data, you first need data. And to add data, you use the INSERT statement from the DML, which we covered in detail in this training. As there are multiple scenarios to consider that include using all columns and all values or just the values, when there are NULL values and default values, and when there's a requirement to set the identity value. We also learned how to INSERT with constraints and finally, using BULK INSERT. Then in the next module, we covered how to modify data in our database using UPDATE, which has many different available parameters, but that the main scenario involved updating in a destination table a set of columns with some values on a specific set of rows to update data in one row or multiple rows. Then we used @@ROWCOUNT to programmatically get the number of rows affected by an UPDATE, covering also how to use variables and joining data from multiple tables. Then we moved on into the module where we learned how to remove records using DELETE, but being careful because removing data means hasta la vista data. We used multiple scenarios with DELETE, be it for one record, many, or even based on conditions from another table with a JOIN. We also learned how we can remove data with constraints using cascade DELETE. And finally, explored a similar statement that while it does not belong to the DML, it is DDL instead, it removes all rows from a table, so it works similar to a DELETE. I am talking about TRUNCATE. Then we talked about transactions, which are important to maintain data integrity as multiple operations take place in an all-or-nothing fashion while also helping with speed. We talked about the four properties of transactions, that's ACID, which stands for atomicity, consistency, isolation, and durability. Then we talked about the types of transactions, namely autocommit, explicit, and implicit, being implicit, the one that we covered using BEGIN, COMMIT, ROLLBACK, and SAVE. Then, we briefly talked about distributed transactions. Then we moved into the next module where we covered some more advanced topics, and by advanced, I mean a bit less common, but still useful while not being too complex like adding multiple records in a single statement or populating one table using a query and retrieving specific related fields used for INSERT. Additionally, we used the OUTPUT clause to retrieve which data has been INSERTED, UPDATED, or DELETED. Next, MERGE, which synchronized tables by using a combination of INSERT, UPDATE, and DELETE statements. Finally, we learned how to BULK INSERT data that was not in the exact same format that we were expecting with the help of BULK INSERT, the bulk copy program, known as bcp, and a format file. And with this, I want to thank you for taking the time to learn with me the T-SQL Data Manipulation Language in this Pluralsight course, the T-SQL Data Manipulation Playbook. I am Xavier Morera, and remember, what you learn is yours for life.