| product\_id | product\_name | stock | price |

|------------|--------------|-------|-------|

| 6 | Trek Neko+ | -1 | 2799 |

* Surround the constraint with a TRY block.
* Add the constraint to the products table.
* Surround the error message with a CATCH block.

-- Set up the TRY block

BEGIN TRY

-- Add the constraint

ALTER TABLE products

ADD CONSTRAINT CHK\_Stock CHECK (stock >= 0);

END TRY

-- Set up the CATCH block

BEGIN CATCH

SELECT 'An error occurred!';

END CATCH

* Surround the INSERT INTO buyers statement with a TRY block.
* Surround the error handling with a CATCH block.
* Surround the INSERT INTO errors statement with another TRY block.
* Surround the nested error handling with another CATCH block.

-- Set up the first TRY block

BEGIN TRY

INSERT INTO buyers (first\_name, last\_name, email, phone)

VALUES ('Peter', 'Thompson', 'peterthomson@mail.com', '555000100');

END TRY

-- Set up the first CATCH block

BEGIN CATCH

SELECT 'An error occurred inserting the buyer! You are in the first CATCH block';

-- Set up the nested TRY block

BEGIN TRY

INSERT INTO errors

VALUES ('Error inserting a buyer');

SELECT 'Error inserted correctly!';

END TRY

-- Set up the nested CATCH block

BEGIN CATCH

SELECT 'An error occurred inserting the error! You are in the nested CATCH block';

END CATCH

END CATCH

Note: Error messages in DataCamp have different anatomy than in SQL Server, but as they show the error message, you won't have any problem.

* Run the code to verify there are compilation errors.
* Correct every compilation error.
* Run the code to get the final output: An error occurred inserting the product!

BEGIN TRY

INSERT INTO products (product\_name, stock, price)

VALUES ('Sun Bicycles ElectroLite - 2017', 10, 1559.99);

END TRY

BEGIN CATCH

SELECT 'An error occurred inserting the product!';

BEGIN TRY

INSERT INTO errors

VALUES ('Error inserting a product');

END TRY

BEGIN CATCH

SELECT 'An error occurred inserting the error!';

END CATCH

END CATCH

* Surround the operation with a TRY block.
* Surround the functions with a CATCH block.
* Select the error information.

-- Set up the TRY block

BEGIN TRY

SELECT 'Total: ' + SUM(price \* quantity) AS total

FROM orders

END TRY

-- Set up the CATCH block

BEGIN CATCH

-- Show error information.

SELECT ERROR\_NUMBER() AS number,

ERROR\_SEVERITY() AS severity\_level,

ERROR\_STATE() AS state,

ERROR\_LINE() AS line,

ERROR\_MESSAGE() AS message;

END CATCH

* Surround the error handling with a CATCH block.
* Insert 'Error inserting a product' in the errors table and surround this insertion with another TRY block.
* Surround the nested error handling with another CATCH block.
* Select the error line and the error message in the inner CATCH block.

BEGIN TRY

INSERT INTO products (product\_name, stock, price)

VALUES ('Trek Powerfly 5 - 2018', 2, 3499.99),

('New Power K- 2018', 3, 1999.99)

END TRY

-- Set up the outer CATCH block

BEGIN CATCH

SELECT 'An error occurred inserting the product!';

-- Set up the inner TRY block

BEGIN TRY

-- Insert the error

INSERT INTO errors

VALUES ('Error inserting a product');

END TRY

-- Set up the inner CATCH block

BEGIN CATCH

-- Show number and message error

SELECT

ERROR\_LINE() AS line,

ERROR\_MESSAGE() AS message;

END CATCH

END CATCH

* Set @product\_id to 5.
* Use the RAISERROR statement with a severity of 11, a state of 1 and the given @product\_id.

Catch the error generated by the RAISERROR statement you coded.

Select the error message using the appropriate function.

BEGIN TRY

DECLARE @product\_id INT = 5;

IF NOT EXISTS (SELECT \* FROM products WHERE product\_id = @product\_id)

RAISERROR('No product with id %d.', 11, 1, @product\_id);

ELSE

SELECT \* FROM products WHERE product\_id = @product\_id;

END TRY

-- Catch the error

BEGIN CATCH

-- Select the error message

SELECT 'You are in a catch block' as message;

END CATCH

* Surround the error handling with a CATCH block.
* Insert the error in the errors table.
* End the insert statement with a semicolon (;).
* Re-throw the original error.

CREATE PROCEDURE insert\_product

@product\_name VARCHAR(50),

@stock INT,

@price DECIMAL

AS

BEGIN TRY

INSERT INTO products (product\_name, stock, price)

VALUES (@product\_name, @stock, @price);

END TRY

-- Set up the CATCH block

BEGIN CATCH

-- Insert the error and end the statement with a semicolon

INSERT INTO errors VALUES ('Error inserting a product');

-- Re-throw the error

THROW;

END CATCH

* Execute the stored procedure called insert\_product.
* Set the appropriate values for the parameters of the stored procedure.
* Surround the error handling with a CATCH block.
* Select the error message.

BEGIN TRY

-- Execute the stored procedure

EXEC insert\_product

-- Set the values for the parameters

@product\_name = 'Super bike',

@stock = 3,

@price = $499.99;

END TRY

-- Set up the CATCH block

BEGIN CATCH

-- Select the error message

SELECT ERROR\_MESSAGE();

END CATCH

* Set @staff\_id to 4.
* Use the THROW statement, with 50001 as the error number, *'No staff member with such id'* as the message text, and 1 as the state.

-- Set @staff\_id to 4

DECLARE @staff\_id INT = 4;

IF NOT EXISTS (SELECT \* FROM staff WHERE staff\_id = @staff\_id)

-- Invoke the THROW statement with parameters

THROW 50001, 'No staff member with such id', 1;

ELSE

SELECT \* FROM staff WHERE staff\_id = @staff\_id

* Set the @first\_name variable to 'Pedro'.
* Assign to the @my\_message variable the concatenation of the text *'There is no staff member with '*, with the value of the @first\_name variable and with the text *' as the first name.'*.
* Use the THROW statement with 50000 as the error number, @my\_message variable as the message parameter, and 1 as the state.

-- Set @first\_name to 'Pedro'

DECLARE @first\_name NVARCHAR(20) = 'Pedro';

-- Concat the message

DECLARE @my\_message NVARCHAR(500) =

CONCAT('There is no staff member with ', @first\_name, ' as the first name.');

IF NOT EXISTS (SELECT \* FROM staff WHERE first\_name = @first\_name)

-- Throw the error

THROW 50000, @my\_message, 1;

* Save into the @current\_stock variable the value of the stock of the product.
* Use the FORMATMESSAGE function with **parameter placeholders** (%s, %d, ... ) to customize the error message. The message has to be *'There are not enough (the given product name) bikes. You only have (the stock of the product) in stock.'*
* Pass to the THROW statement the variable of the custom message.

DECLARE @product\_name AS NVARCHAR(50) = 'Trek CrossRip+ - 2018';

DECLARE @number\_of\_sold\_bikes AS INT = 10;

DECLARE @current\_stock INT;

-- Select the current stock

SELECT FORMATMESSAGE = stock FROM products WHERE product\_name = @product\_name;

DECLARE @my\_message NVARCHAR(500) =

-- Customize the message

CONCAT('There are not enough %s bikes. You only have %d in stock.', @number\_of\_sold\_bikes, @current\_stock);

IF (@current\_stock - @number\_of\_sold\_bikes < 0)

-- Throw the error

THROW 50000, @my\_message, 1;

* Pass to the sp\_addmessage stored procedure 50002 as the message id, 16 as the severity, and *'There are not enough %s bikes. You only have %d in stock.'* as the message text.
* Use FORMATMESSAGE, setting the first parameter (message number) to be 50002. Complete the second and the third parameters to replace the parameter placeholders of the message (%s and %d) with the appropriate variables.
* Pass to the THROW statement the custom message.

-- Pass the variables to the stored procedure

EXEC sp\_addmessage @msgnum = 50002, @severity = 16, @msgtext = 'There are not enough %s bikes. You only have %d in stock.', @lang = N'us\_english';

DECLARE @product\_name AS NVARCHAR(50) = 'Trek CrossRip+ - 2018';

DECLARE @number\_of\_sold\_bikes AS INT = 10;

DECLARE @current\_stock INT;

SELECT @current\_stock = stock FROM products WHERE product\_name = @product\_name;

DECLARE @my\_message NVARCHAR(500) =

-- Prepare the error message

FORMATMESSAGE(50002, @product\_name, @current\_stock);

IF (@current\_stock - @number\_of\_sold\_bikes < 0)

-- Throw the error

THROW 50000, @my\_message, 1;

* Run the code to verify there are errors.
* Correct every error.

BEGIN TRY

BEGIN TRAN;

UPDATE accounts SET current\_balance = current\_balance - 100 WHERE account\_id = 1;

INSERT INTO transactions VALUES (1, -100, GETDATE());

UPDATE accounts SET current\_balance = current\_balance + 100 WHERE account\_id = 5;

INSERT INTO transactions VALUES (5, 100, GETDATE());

COMMIT TRAN;

END TRY

BEGIN CATCH

ROLLBACK TRAN;

END CATCH

* Begin the transaction.
* Correct the mistake in the operation.
* Commit the transaction if there are no errors.
* Inside the CATCH block, roll back the transaction.

BEGIN TRY

-- Begin the transaction

BEGIN TRAN;

UPDATE accounts SET current\_balance = current\_balance - 100 WHERE account\_id = 1;

INSERT INTO transactions VALUES (1, -100, GETDATE());

UPDATE accounts SET current\_balance = current\_balance + 100 WHERE account\_id = 5;

-- Correct it

INSERT INTO transactions VALUES (500, 100, GETDATE());

-- Commit the transaction

COMMIT TRAN;

END TRY

BEGIN CATCH

SELECT 'Rolling back the transaction';

-- Rollback the transaction

ROLLBACK TRAN;

END CATCH

* Begin the transaction.
* Check if the number of affected rows is bigger than 200.
* Rollback the transaction if the number of affected rows is more than 200.
* Commit the transaction if the number of affected rows is less than or equal to 200.

-- Begin the transaction

BEGIN TRAN;

UPDATE accounts set current\_balance = current\_balance + 100

WHERE current\_balance < 5000;

-- Check number of affected rows

IF @@ROWCOUNT > 200

BEGIN

-- Rollback the transaction

ROLLBACK TRAN;

SELECT 'More accounts than expected. Rolling back';

END

ELSE

BEGIN

-- Commit the transaction

COMMIT TRAN;

SELECT 'Updates commited';

END

* Begin the transaction.
* Correct the mistake in the operation.
* Inside the TRY block, check if there is a transaction and commit it.
* Inside the CATCH block, check if there is a transaction and roll it back.

BEGIN TRY

-- Begin the transaction

BEGIN TRAN;

-- Correct the mistake

UPDATE accounts SET current\_balance = current\_balance + 200

WHERE account\_id = 10;

-- Check if there is a transaction

IF @@TRANCOUNT > 0

-- Commit the transaction

COMMIT TRAN;

SELECT \* FROM accounts

WHERE account\_id = 10;

END TRY

BEGIN CATCH

SELECT 'Rolling back the transaction';

-- Check if there is a transaction

IF @@TRANCOUNT > 0

-- Rollback the transaction

ROLLBACK TRAN;

END CATCH

* Run the code to verify there are errors.
* Correct every error.

BEGIN TRAN;

-- Mark savepoint1

SAVE TRAN savepoint1;

INSERT INTO customers VALUES ('Mark', 'Davis', 'markdavis@mail.com', '555909090');

-- Mark savepoint2

SAVE TRAN savepoint2;

INSERT INTO customers VALUES ('Zack', 'Roberts', 'zackroberts@mail.com', '555919191');

-- Rollback savepoint2

ROLLBACK TRAN savepoint2;

-- Rollback savepoint1

ROLLBACK TRAN savepoint1;

-- Mark savepoint3

SAVE TRAN savepoint3;

INSERT INTO customers VALUES ('Jeremy', 'Johnsson', 'jeremyjohnsson@mail.com', '555929292');

-- Commit the transaction

COMMIT TRAN;

* Use the appropriate setting of XACT\_ABORT.
* Begin the transaction.
* If the number of affected rows is less than or equal to 10, throw the error using the THROW statement, with a number of 55000.
* Commit the transaction if the number of affected rows is more than 10.

-- Use the appropriate setting

SET XACT\_ABORT ON;

-- Begin the transaction

BEGIN TRAN;

UPDATE accounts set current\_balance = current\_balance - current\_balance \* 0.01 / 100

WHERE current\_balance > 5000000;

IF @@ROWCOUNT <= 10

-- Throw the error

THROW 55000, 'Not enough wealthy customers!', 1;

ELSE

-- Commit the transaction

COMMIT TRAN;

* Use the appropriate setting of XACT\_ABORT.
* Check if there is an open transaction.
* Rollback the transaction.
* Select the error message.

-- Use the appropriate setting

SET XACT\_ABORT ON;

BEGIN TRY

BEGIN TRAN;

INSERT INTO customers VALUES ('Mark', 'Davis', 'markdavis@mail.com', '555909090');

INSERT INTO customers VALUES ('Dylan', 'Smith', 'dylansmith@mail.com', '555888999');

COMMIT TRAN;

END TRY

BEGIN CATCH

-- Check if there is an open transaction

IF XACT\_STATE () <> 0

-- Rollback the transaction

ROLLBACK TRAN;

-- Select the message of the error

SELECT ERROR\_MESSAGE() AS Error\_message;

END CATCH

* Set the READ UNCOMMITTED isolation level.
* Select first\_name, last\_name, email and phone from customers table.

-- Set the appropriate isolation level

SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED

-- Select first\_name, last\_name, email and phone

SELECT

first\_name,

last\_name,

email,

phone

FROM customers;

As the number of accounts is an important result, you don't want to read data modified by other transactions that haven't committed or rolled back yet. In doing this, you prevent dirty reads. However, you don't need to consider having non-repeatable or phantom reads.

* Set the appropriate isolation level to prevent dirty reads.
* Select the count of accounts that match the criteria.

-- Set the appropriate isolation level

SET TRANSACTION ISOLATION LEVEL READ COMMITTED

-- Count the accounts

SELECT count(\*) AS number\_of\_accounts

FROM accounts

WHERE current\_balance >= 50000;

After that, you want to select the same data again, ensuring nothing has changed.

As this is critical, you think it is better if nobody can change anything in the customers table until you finish your analysis. In doing this, you prevent non-repeatable reads.

* Set the appropriate isolation level to prevent non-repeatable reads.
* Begin a transaction.
* Commit the transaction.

-- Set the appropriate isolation level

SET TRANSACTION ISOLATION LEVEL REPEATABLE READ

-- Begin a transaction

BEGIN TRAN

SELECT \* FROM customers;

-- some mathematical operations, don't care about them...

SELECT \* FROM customers;

-- Commit the transaction

COMMIT TRAN

You prepare a script to select that information, and with the result of this selection, you need to process some mathematical operations. (We won't focus on these operations for this exercise.) After that, you want to select the same data again, ensuring nothing has changed.

* Set the appropriate isolation level to prevent phantom reads.
* Begin the transaction.
* Commit the transaction.

-- Set the appropriate isolation level

SET TRANSACTION ISOLATION LEVEL SERIALIZABLE

-- Begin a transaction

BEGIN TRAN

SELECT \* FROM customers;

-- After some mathematical operations, we selected information from the customers table.

SELECT \* FROM customers;

-- Commit the transaction

COMMIT TRAN

You need to select the customers and execute some mathematical operations again. (We won't focus either on these operations for this exercise.) After that, you want to select the customers with the customer\_id between 1 and 10 again, ensuring nothing has changed.

* Set the appropriate isolation level to prevent phantom reads.
* Begin a transaction.
* Select those customers you want to lock.
* Commit the transaction.

-- Set the appropriate isolation level

SET TRANSACTION ISOLATION LEVEL SERIALIZABLE

-- Begin a transaction

BEGIN TRAN

-- Select customer\_id between 1 and 10

SELECT \*

FROM customers

WHERE customer\_id BETWEEN 1 AND 10;

-- After completing some mathematical operation, select customer\_id between 1 and 10

SELECT \*

FROM customers

WHERE customer\_id BETWEEN 1 AND 10;

-- Commit the transaction

COMMIT TRAN

Now, you want every script you already made to be run with the READ COMMITTED SNAPSHOT option set to ON. In doing this, each statement under the READ COMMITTED isolation level will see the committed changes that occur before the start of each statement.

Which options do you need to set in your database?

ALTER DATABASE myDatabaseName SET ALLOW\_SNAPSHOT\_ISOLATION ON and ALTER DATABASE myDatabaseName SET READ\_COMMITTED\_SNAPSHOT ON.

the main difference between READ COMMITTED SNAPSHOT and the SNAPSHOT isolation level is that with the SNAPSHOT isolation level you can only see the committed changes that occur before the start of a transaction and the changes made by that transaction.

read some uncommitted data within a transaction. He has to decide whether to use WITH (NOLOCK) option or the READ UNCOMMITTED isolation level, but he is not sure about the differences between both options.

The WITH (NOLOCK) option behaves like the READ UNCOMMITTED isolation level. But whereas the isolation level applies for the entire connection, WITH NOLOCK applies to a specific table.

You figured it out! Remember that both WITH (NOLOCK) and READ UNCOMMITTED can read dirty reads.

change your select query to get the information right now without changing the isolation level? In doing this you can read the uncommitted data from the transactions table.

Change your script to avoid being blocked.

SELECT \*

-- Avoid being blocked

FROM transactions WITH (NOLOCK)

WHERE account\_id = 1

Improving Query Performance in SQL

* Change all SQL syntax (clauses, operators, and functions) to UPPERCASE.
* Make sure all SQL syntax begins on a new line.
* Add an indent to the calculated BMI column and OR statement.

SELECT PlayerName, Country,

ROUND(Weight\_kg/SQUARE(Height\_cm/100),2) BMI

FROM Players

WHERE Country = 'USA'

OR Country = 'Canada'

ORDER BY BMI;

* Create a comment block on lines 1 and 4.
* Add the above comment to the block.
* Comment out the ORDER BY statement and add Order by not required comment on the same line.
* Add ; directly after 'Canada' to indicate the new ending of the query.

--Select the player Name & Country

SELECT PlayerName, Country,

ROUND(Weight\_kg/SQUARE(Height\_cm/100),2) BMI

FROM Players

--where country should be USA or CA

WHERE Country = 'USA'

OR Country = 'Canada';

--ORDER BY BMI;

* Add the line comment First attempt, contains errors and inconsistent formatting on line 2.
* Block comment out your friend's query on lines 3 and 11.
* Add the line comment Second attempt - errors corrected and formatting fixed on line 14.
* Remove the block comment syntax from your query on lines 15 and 23.

-- Your friend's query

--First attempt, contains errors and inconsistent formatting

SELECT p.PlayerName, p.Country,

SUM(ps.TotalPoints)

AS TotalPoints

FROM PlayerStats ps inner join Players p

On ps.PlayerName = p.PlayerName

WHERE p.Country = 'New Zealand'

Group

BY p.PlayerName, Country

ORDER BY p.Country;

-- Your query

SELECT p.PlayerName, p.Country,

SUM(ps.TotalPoints) AS TotalPoints

FROM PlayerStats ps

INNER JOIN Players p

ON ps.PlayerName = p.PlayerName

WHERE p.Country = 'New Zealand'

GROUP BY p.PlayerName, p.Country;

* Alias the new average BMI column as AvgTeamBMI.
* Alias the PlayerStats table as ps.
* Alias the ***sub-query*** as p.
* The PlayerStats table and ***sub-query*** are joining on the column PlayerName. Add the aliases to the joining PlayerName columns.

SELECT Team,

ROUND(AVG(BMI),2) AS AvgTeamBMI

-- Alias the new column

FROM PlayerStats AS ps

-- Alias PlayerStats table

INNER JOIN

(SELECT PlayerName, Country,

Weight\_kg/SQUARE(Height\_cm/100) BMI

FROM Players) AS p

-- Alias the sub-query

-- Alias the joining columns

ON ps.PlayerName = p.PlayerName

GROUP BY Team

HAVING AVG(BMI) >= 25;

* Complete the required query using FROM, WHERE, SELECT and ORDER BY.
* Rearrange the query so that the syntax is in the order that it will run without error.

/\*

Returns earthquakes in New Zealand with a magnitude of 7.5 or more

\*/

SELECT Date, Place, NearestPop, Magnitude

FROM Earthquakes

WHERE Country = 'NZ'

AND Magnitude >= 7.5

ORDER BY Magnitude DESC;

* Rearrange the query with the correct syntax order in the format provided.

-- Your query

SELECT Date,

Place,

NearestPop,

Magnitude

FROM Earthquakes

WHERE Country = 'JP'

AND Magnitude >= 8

ORDER BY Magnitude DESC;

* Solve the query with the correct syntax order in the format provided.

/\*

Location of the epicenter of earthquakes with a 9+ magnitude

\*/

-- Replace City with the correct column name

SELECT n.CountryName AS Country

,e.NearestPop AS ClosestCity

,e.Date

,e.Magnitude

FROM Nations AS n

INNER JOIN Earthquakes AS e

ON n.Code2 = e.Country

WHERE e.Magnitude >= 9

ORDER BY e.Magnitude DESC;

Try to understand what the error is telling you when you run the first query, then comment out the query block on lines 2 and 9.-- First query

/\*

SELECT PlayerName,

Team,

Position,

((DRebound+ORebound)/CAST(GamesPlayed AS numeric)) AS AvgRebounds

FROM PlayerStats

WHERE AvgRebounds >= 12;

\*/

* In the sub-query calculate average total rebounds in a new column, AvgRebounds.
* Add the new column to the SELECT statement.
* Apply a filter condition for 12 or more average total rebounds.

-- Second query

-- Add the new column to the select statement

SELECT PlayerName,

Team,

Position,

AvgRebounds -- Add the new column

FROM

-- Sub-query starts here

(SELECT

PlayerName,

Team,

Position,

-- Calculate average total rebounds

(ORebound+DRebound)/CAST(GamesPlayed AS numeric) AS AvgRebounds

FROM PlayerStats) tr

WHERE AvgRebounds >= 12; -- Filter rows

* Select **Run Code** to see what your friend's query returns.
* Think about why his query is not giving you exactly what you require then comment out his filter on line 7.
* Add a new wildcard filter condition - Louisiana%

SELECT PlayerName,

Country,

College,

DraftYear,

DraftNumber

FROM Players

--WHERE UPPER(LEFT(College,5)) LIKE 'LOU%';

WHERE College LIKE 'Louisiana%' -- Add the new wildcard filter

* Why should HAVING *not* be used as a filter condition in this query?

SELECT Country, COUNT(\*) CountOfPlayers

FROM Players

GROUP BY Country

HAVING Country

IN ('Argentina','Brazil','Dominican Republic'

,'Puerto Rico');

The filter is on individual rows. Using HAVING here, for filtering, could increase the time a query takes to run.

 Add the WHERE filter condition.

 Fill in the missing two Latin American countries in the IN statement.

SELECT Country, COUNT(\*) CountOfPlayers

FROM Players

-- Add the filter condition

WHERE Country

-- Fill in the missing countries

IN ('Argentina','Brazil','Dominican Republic'

,'Puerto Rico')

GROUP BY Country;

* Apply a filter condition for only rows where position equals Power Forward (PF).
* Apply a *grouped row* filter for total points greater than 3000.

SELECT Team,

SUM(TotalPoints) AS TotalPFPoints

FROM PlayerStats

-- Filter for only rows with power forwards

WHERE Position = 'PF'

GROUP BY Team

-- Filter for total points greater than 3000

HAVING SUM(TotalPoints) > 3000;

SELECT Team,

SUM(TotalPoints) AS TotalCPoints

FROM PlayerStats

WHERE Position = 'C'

GROUP BY Team

HAVING SUM(TotalPoints) > 2500;

When using HAVING in a query which one of the following statements is **FALSE**?

HAVING and WHERE produce the same output, so it doesn't matter which one you use.

* SELECT all rows and columns from the Earthquakes table.
* Look at the results of the query to determine which other columns to select.

Complete the query to select only the required columns and filter for only the requested countries.

SELECT \* -- Select all rows and columns

FROM Earthquakes;

SELECT latitude, -- Y location coordinate column

longitude, -- X location coordinate column

magnitude , -- Earthquake strength column

depth, -- Earthquake depth column

NearestPop -- Nearest city column

FROM Earthquakes

WHERE Country = 'PG' -- Papua New Guinea country code

OR Country = 'ID'; -- Indonesia country code

* Limit the number of rows to ten.
* Order the results from shallowest to deepest.

SELECT TOP 10 -- Limit the number of rows to ten

Latitude,

Longitude,

Magnitude,

Depth,

NearestPop

FROM Earthquakes

WHERE Country = 'PG'

OR Country = 'ID'

ORDER BY Depth; -- Order results from shallowest to deepest

* Limit rows to the ***upper quartile***.
* Order the results from strongest to weakest earthquake.

SELECT TOP 25 PERCENT -- Limit rows to the upper quartile

Latitude,

Longitude,

Magnitude,

Depth,

NearestPop

FROM Earthquakes

WHERE Country = 'PG'

OR Country = 'ID'

ORDER BY Magnitude DESC; -- Order the results

Add the closest city and view the output of the query to confirm duplicated rows.

SELECT NearestPop, -- Add the closest city

Country

FROM Earthquakes

WHERE Magnitude >= 8

AND NearestPop IS NOT NULL

ORDER BY NearestPop;

* Add DISTINCT() to the column representing the closest city to remove duplicates.
* Add the filtering condition for earthquakes with a magnitude of 8 or more.

SELECT DISTINCT(NearestPop),-- Remove duplicate city

Country

FROM Earthquakes

WHERE magnitude >= 8 -- Add filter condition

AND NearestPop IS NOT NULL

ORDER BY NearestPop;

* Get the number of cities near earthquakes of magnitude 8 or more.
* Add column groupings.

SELECT NearestPop,

Country,

COUNT(NearestPop) NumEarthquakes -- Number of cities

FROM Earthquakes

WHERE Magnitude >= 8

AND Country IS NOT NULL

GROUP BY NearestPop, Country -- Group columns

ORDER BY NumEarthquakes DESC;

* Add the city column from the Cities table to the first query.
* Append queries using UNION
* Add the column for the Nation capital to the second query.
* Check how many rows were returned.

SELECT CityName AS NearCityName, -- City name column

CountryCode

FROM Cities

UNION -- Append queries

SELECT Capital AS NearCityName, -- Nation capital column

Code2 AS CountryCode

FROM Nations;

* Now append the same queries using UNION ALL.
* Add the column for the country code to the second query.

SELECT CityName AS NearCityName,

CountryCode

FROM Cities

UNION ALL -- Append queries

SELECT Capital AS NearCityName,

Code2 AS CountryCode -- Country code column

FROM Nations;

UNION or DISTINCT()?

When deciding whether to use DISTINCT() or UNION in a query to remove duplicate rows, which of the following questions would you **NOT** ask yourself?

Should I be thinking about duplicate rows because my queries never produce duplicate rows?

* Add the country code column to the outer query.
* Add the country code column to the sub-query.
* Filter for a depth of 400km or more.

SELECT UNStatisticalRegion,

CountryName

FROM Nations

WHERE Code2 -- Country code for outer query

IN (SELECT Country -- Country code for sub-query

FROM Earthquakes

WHERE depth >= 400 ) -- Depth filter

ORDER BY UNStatisticalRegion;

Why is the query from **Step 1** an example of an **uncorrelated sub-query**?

The sub-query does not reference the outer query.

* Add the average magnitude column in the sub-query.
* Add the Nations country code column reference in the sub-query.

SELECT UNContinentRegion,

CountryName,

(SELECT AVG(magnitude) -- Add average magnitude

FROM Earthquakes e

-- Add country code reference

WHERE n.Code2 = e.Country) AS AverageMagnitude

FROM Nations n

ORDER BY UNContinentRegion DESC,

AverageMagnitude DESC;

Why is the query from **Step 1** an example of a **correlated sub-query**?

The sub-query references the outer query.

* Add the 2017 population column from the Cities table.
* Add the outer query country code column to the sub-query.
* Add the outer query table.

SELECT

n.CountryName,

(SELECT MAX(c.Pop2017) -- Add 2017 population column

FROM Cities AS c

-- Outer query country code column

WHERE c.CountryCode = n.Code2) AS BiggestCity

FROM Nations AS n; -- Outer query table

* Join the Nations table to the sub-query.
* Add the joining country code columns from the Nations table and sub-query.

SELECT n.CountryName,

c.BiggestCity

FROM Nations AS n

INNER JOIN -- Join the Nations table and sub-query

(SELECT CountryCode,

MAX(Pop2017) AS BiggestCity

FROM Cities

GROUP BY CountryCode) AS c

ON n.Code2 = c.CountryCode; -- Add the joining columns

* Add the table with country capital cities to the left query.
* Add the operator that compares the two queries.
* Add the city name column from the Earthquakes table.

SELECT Capital

FROM Nations -- Table with capital cities

INTERSECT -- Add the operator to compare the two queries

SELECT NearestPop -- Add the city name column

FROM Earthquakes;

* Add the country code column from the Nations table.
* Add the operator that compares the two queries.
* Add the table with country codes to the right query.

SELECT Code2 -- Add the country code column

FROM Nations

EXCEPT -- Add the operator to compare the two queries

SELECT Country

FROM Earthquakes; -- Table with country codes

INTERSECT CountryName from a table in the Earthquakes database and Country from a table in the NBA Season 2017-2018 database.

SELECT CountryName

FROM Nations -- Table from Earthquakes database

INTERSECT -- Operator for the intersect between tables

SELECT Country

FROM Players; -- Table from NBA Season 2017-2018 database

With one exception, all NBA teams are USA based, so why does USA not appear in the results? Are there no Americans playing in the NBA?

To help get your answer, use the two queries below;

1. Delete the query in the query console.
2. Copy and paste one of the queries into the query console.
3. Select **Run Code** to view the results.
4. Repeat steps 1 to 4 for the other query.

SELECT \*

FROM Nations

WHERE CountryName LIKE 'U%'

SELECT \*

FROM Players

WHERE Country LIKE 'U%'

The values do not match. In the Nations table, the value for country name is stored as United States of America, and in the Players table, the value is stored as USA.

INTERSECT is being used correctly. However, although both tables contain names of countries, the way the values are stored is different. In the Nations table, the values are stored as United States of America and in the Players table, as USA. Therefore, there is no match, which is a good reason to perform a thorough data interrogation on all data sets before working with databases.

* Add the 2017 country population and capital city name columns to the outer query.
* Add the operator to compare the outer query with the sub-query.

-- First attempt

SELECT CountryName,

Pop2017, -- 2017 country population

Capital, -- Capital city

WorldBankRegion

FROM Nations

WHERE Capital IN -- Add the operator to compare queries

(SELECT NearestPop

FROM Earthquakes);

* Update the query with the 2016 population instead of the 2017 population.
* Add the operator to compare the outer query with the sub-query.
* Add the two city name columns, being compared, in the sub-query.

-- Second attempt

SELECT CountryName,

Capital,

Pop2016, -- 2016 country population

WorldBankRegion

FROM Nations AS n

WHERE EXISTS -- Add the operator to compare queries

(SELECT 1

FROM Earthquakes AS e

WHERE n.Capital = e.NearestPop); -- Columns being compared

* Add the operator to compare the outer query with the sub-query.
* Add the country code column to the sub-query.

SELECT WorldBankRegion,

CountryName

FROM Nations

WHERE Code2 NOT IN -- Add the operator to compare queries

(SELECT CountryCode -- Country code column

FROM Cities);

* Add the country capital column to the outer query.
* Add the operator to compare the outer query with the sub-query.
* Add the two country code columns being compared in the sub-query.

SELECT WorldBankRegion,

CountryName,

Code2,

Capital, -- Country capital column

Pop2017

FROM Nations AS n

WHERE NOT EXISTS -- Add the operator to compare queries

(SELECT 1

FROM Cities AS c

WHERE n.Code2 = c.CountryCode); -- Columns being compared

* Add the country capital name column to the outer query.
* Add the city name column to the sub-query.
* Check how many rows the query returns. Does this mean that earthquakes have been recorded near every capital city in the world?

SELECT WorldBankRegion,

CountryName,

Capital -- Capital city name column

FROM Nations

WHERE Capital NOT IN

(SELECT NearestPop -- City name column

FROM Earthquakes);

The column in the SELECT statement of the sub-query contains *NULL* values and will require a filter to remove the *NULL* values from the query.

* Add the WHERE filter condition to the sub-query to get the query working correctly.

SELECT WorldBankRegion,

CountryName,

Capital

FROM Nations

WHERE Capital NOT IN

(SELECT NearestPop

FROM Earthquakes

WHERE NearestPop IS NOT NULL); -- filter condition

* Add the table for the outer query.
* Add the operator to compare the outer query with the sub-query.
* Add the table for the sub-query.
* Check the results. Only columns from the Teams table are returned.

-- Initial query

SELECT TeamName,

TeamCode,

City

FROM Teams AS t -- Add table

WHERE EXISTS -- Operator to compare queries

(SELECT 1

FROM Earthquakes AS e -- Add table

WHERE t.City = e.NearestPop);

Something doesn't look right. You'll need columns from the Earthquakes *and* Teams tables to makes sense of the results.

* Add the place description and country code where the earthquake occurred.
* Add the operator to compare the tables.

-- Second query

SELECT t.TeamName,

t.TeamCode,

t.City,

e.Date,

e.place, -- Place description

e.Country -- Country code

FROM Teams AS t

INNER JOIN Earthquakes AS e -- Operator to compare tables

ON t.City = e.NearestPop

 The two earthquakes returned in the second query occurred in San Antonio, Chile, nowhere near the home of the San Antonio Spurs. We were able to determine this because results from an INNER JOIN can contain any column, from all joined queries.  
  
An insurance company would probably use a more comprehensive spatial database for this type of assessment instead of our simple databases used in this course.

* Add the joining operator between the Customers and Orders tables.
* Add the joining columns from the Customers and Orders tables.

-- First attempt

SELECT c.CustomerID,

c.CompanyName,

c.ContactName,

c.ContactTitle,

c.Phone

FROM Customers c

LEFT OUTER JOIN Orders o -- Joining operator

ON c.CustomerID = o.CustomerID -- Joining columns

WHERE c.Country = 'France';

Add the filter condition to turn the query into an exclusive LEFT OUTER JOIN

-- Second attempt

SELECT c.CustomerID,

c.CompanyName,

c.ContactName,

c.ContactTitle,

c.Phone

FROM Customers c

LEFT OUTER JOIN Orders o

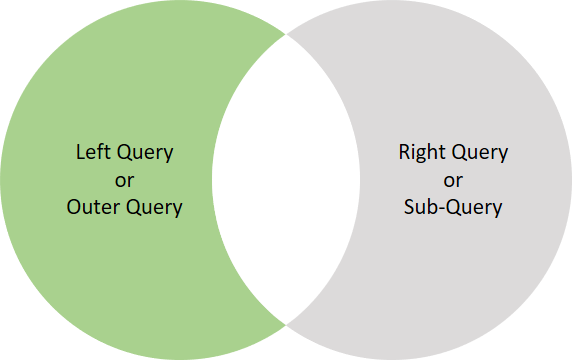
ON c.CustomerID = o.CustomerID

WHERE c.Country = 'France'

AND o.ShipCountry IS NULL; -- Filter condition

The Venn diagram below describes which method used to check whether the data in one table is present, or absent, in a related table?

The Earthquakes database is available for you to test scenarios in the **query console**.



exclusive LEFT OUTER JOIN

LEFT OUTER JOIN checks for data in one table that is absent in a related table. It does this by using IS NULL in a WHERE filter condition of the right query to restrict it to rows that do not match in the left query.

* Turn on STATISTICS TIME.

SET STATISTICS TIME ON -- Turn the time command on

* Add the filter operator for each sub-query.
* Add the table from the Earthquakes database to the *first* query.

-- Query 1

SELECT \*

FROM Teams

-- Sub-query 1

WHERE City IN -- Sub-query filter operator

(SELECT CityName

FROM Cities) -- Table from Earthquakes database

-- Sub-query 2

AND City IN -- Sub-query filter operator

(SELECT CityName

FROM Cities

WHERE CountryCode IN ('US','CA'))

-- Sub-query 3

AND City IN -- Sub-query filter operator

(SELECT CityName

FROM Cities

WHERE Pop2017 >2000000);

* Add the filter operator for the sub-query.
* Add the two city name columns being compared to the sub-query.

-- Query 2

SELECT \*

FROM Teams AS t

WHERE EXISTS -- Sub-query filter operator

(SELECT 1

FROM Cities AS c

WHERE t.City = c.CityName -- Columns being compared

AND c.CountryCode IN ('US','CA')

AND c.Pop2017 > 2000000);

* Turn off STATISTICS TIME.

SET STATISTICS TIME OFF -- Turn the time command off

* Turn on STATISTICS IO.

SET STATISTICS IO ON -- Turn the IO command on

* Add the table used to count the number of orders.
* Add the filter operator for the second sub-query.

-- Example 1

SELECT CustomerID,

CompanyName,

(SELECT COUNT(\*)

FROM Orders AS o -- Add table

WHERE c.CustomerID = o.CustomerID) CountOrders

FROM Customers AS c

WHERE CustomerID IN -- Add filter operator

(SELECT CustomerID

FROM Orders

WHERE ShipCity IN

('Berlin','Bern','Bruxelles','Helsinki',

'Lisboa','Madrid','Paris','London'));

From the STATISTICS IO output below, how many data pages from the Orders table were read from memory to complete the query?

Table 'Customers'. Scan count 1, logical reads 2, physical reads 0,...

Table 'Orders'. Scan count 2, logical reads 32, physical reads 0,...

32

* Add the join operator.
* Add the shipping destination city column in the filter condition.

-- Example 2

SELECT c.CustomerID,

c.CompanyName,

COUNT(o.CustomerID)

FROM Customers AS c

INNER JOIN Orders AS o -- Join operator

ON c.CustomerID = o.CustomerID

WHERE o.ShipCity IN -- Shipping destination column

('Berlin','Bern','Bruxelles','Helsinki',

'Lisboa','Madrid','Paris','London')

GROUP BY c.CustomerID,

c.CompanyName;

* Turn off STATISTICS IO.

SET STATISTICS IO OFF -- Turn the IO command off

* Add the two country codes to the filter condition for **Query 1**.

-- Query 1

SELECT \*

FROM Cities

WHERE CountryCode = 'RU' -- Country code

OR CountryCode = 'CN' -- Country code

* Add the two country codes to the filter condition for **Query 2**.

-- Query 2

SELECT \*

FROM Cities

WHERE CountryCode IN ('JM','NZ') -- Country codes

For these two queries, what conclusion could you make using the following output from the STATISTICS IO command?

**Query 1**

4694 results returned

Table 'Cities'. ..., logical reads 274, ... ,

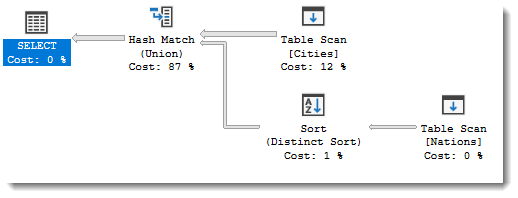
**Query 2**

212 results returned

Table 'Cities'. ..., logical reads 10, ... ,

**Query 2** accesses a clustered index because logical reads indicates fewer data pages were accessed compared to **Query 1**

You are given an execution plan of a query that returns all cities listed in the Earthquakes database. The query appends queries from the Nations and Cities tables. Use the following execution plan to determine if the appending operator used is UNION or UNION ALL



Add the operator that the execution plan indicates was used to append the queries.

SELECT CityName AS NearCityName,

CountryCode

FROM Cities

UNION -- Append queries

SELECT Capital AS NearCityName,

Code2 AS CountryCode

FROM Nations;

### Building and Optimizing Triggers in SQL Server

* Create a new trigger on the Discounts table.
* Use the trigger to *prevent* DELETE statements.

-- Create a new trigger that fires when deleting data

CREATE TRIGGER PreventDiscountsDelete

ON Discounts

-- The trigger should fire instead of DELETE

INSTEAD OF DELETE

AS

PRINT 'You are not allowed to delete data from the Discounts table.';

* Create the new trigger for the Orders table.
* Set the trigger to be fired only after UPDATE statements.

-- Set up a new trigger

CREATE TRIGGER OrdersUpdatedRows

ON Orders

-- The trigger should fire after UPDATE statements

AFTER UPDATE

-- Add the AS keyword before the trigger body

AS

-- Insert details about the changes to a dedicated table

INSERT INTO OrdersUpdate(OrderID, OrderDate, ModifyDate)

SELECT OrderID, OrderDate, GETDATE()

FROM inserted;

* Create the ProductsNewItems trigger on the Products table.
* Set the trigger to fire when data is inserted into the table.

-- Create a new trigger

CREATE TRIGGER ProductsNewItems

ON Products

AFTER INSERT

AS

-- Add details to the history table

INSERT INTO ProductsHistory(Product, Price, Currency, FirstAdded)

SELECT Product, Price, Currency, GETDATE()

FROM inserted;

* Run an update on the Discounts table (this will fire the CustomerDiscountHistory trigger).
* Get all the rows from DiscountsHistory to verify the outcome

-- Run an update for some of the discounts

UPDATE Discounts

SET Discount = Discount + 1

WHERE Discount <= 5;

-- Verify the trigger ran successfully

SELECT \* FROM DiscountsHistory;

Execute the MonthlyOrders regular stored procedure, using EXECUTE MonthlyOrders.

Then, try to execute the CustomerDiscountHistory trigger using

UPDATE Discounts SET Discount = Discount + 1 WHERE Discount <= 5

What conclusions can be drawn from the execution results and the step performed earlier?

Triggers can only be fired by the corresponding event, while regular stored procedures can be executed explicitly when needed.

Insert new data into SalesWithPrice and then run a SELECT from the same table to verify the outcome.

-- Add the following rows to the table

INSERT INTO SalesWithPrice (Customer, Product, Price, Currency, Quantity)

VALUES ('Fruit Mag', 'Pomelo', 1.12, 'USD', 200),

('VitaFruit', 'Avocado', 2.67, 'USD', 400),

('Tasty Fruits', 'Blackcurrant', 2.32, 'USD', 1100),

('Health Mag', 'Kiwi', 1.42, 'USD', 100),

('eShop', 'Plum', 1.1, 'USD', 500);

-- Verify the results after adding the new rows

SELECT \* FROM SalesWithPrice;

Insert new data into SalesWithoutPrice and then run a SELECT from the same table to verify the outcome.

-- Add the following rows to the table

INSERT INTO SalesWithoutPrice (Customer, Product, Currency, Quantity)

VALUES ('Fruit Mag', 'Pomelo', 'USD', 200),

('VitaFruit', 'Avocado', 'USD', 400),

('Tasty Fruits', 'Blackcurrant', 'USD', 1100),

('Health Mag', 'Kiwi', 'USD', 100),

('eShop', 'Plum', 'USD', 500);

-- Verify the results after the INSERT

SELECT \* FROM SalesWithoutPrice;

The previous step used both a computed column and a trigger to calculate the TotalAmount value automatically. From a user perspective, there was no difference, but from a technical perspective, there is one.

What is the major limitation of computed columns that can be easily overcome with the use of triggers?

* A computed column cannot use columns from other tables for the calculation.
* Create the TrackRetiredProducts trigger on the Products table.
* Set the trigger to fire after rows are deleted from the table.

-- Create the trigger

CREATE TRIGGER TrackRetiredProducts

ON Products

AFTER DELETE

AS

INSERT INTO RetiredProducts (Product, Measure)

SELECT Product, Measure

FROM deleted;

Remove retired items from the Products table and check the output from the RetiredProducts table.

-- Remove the products that will be retired

DELETE FROM Products

WHERE Product IN ('Cloudberry', 'Guava', 'Nance', 'Yuzu');

-- Verify the output of the history table

SELECT \* FROM RetiredProducts;

* Keep track of canceled orders (rows deleted from the Orders table). Their details will be kept in the table CanceledOrders upon removal.
* Keep track of discount changes in the table Discounts. Both the old and the new values will be copied to the DiscountsHistory table.
* Send an email to the Sales team via the SendEmailtoSales stored procedure when a new order is placed.

Create a new trigger on the Orders table to keep track of any canceled orders.

-- Create a new trigger for canceled orders

CREATE TRIGGER KeepCanceledOrders

ON Orders

AFTER DELETE

AS

INSERT INTO Orders

SELECT \* FROM deleted;

Create a new trigger on the Discounts table to keep track of discount value changes.

-- Create a new trigger to keep track of discounts

CREATE TRIGGER CustomerDiscountHistory

ON Discounts

AFTER UPDATE

AS

-- Store old and new values into the `DiscountsHistory` table

INSERT INTO DiscountsHistory (Customer, OldDiscount, NewDiscount, ChangeDate)

SELECT i.Customer, d.Discount, i.Discount, GETDATE()

FROM inserted AS i

INNER JOIN deleted AS d ON i.Customer = d.Customer;

Create the trigger NewOrderAlert to notify the Sales team when new orders are placed.

-- Notify the Sales team of new orders

CREATE TRIGGER SendEmailtoSales

ON Orders

AFTER DELETE

AS

EXECUTE SendEmailtoSales;

* Create a new trigger on the Orders table.
* Set the trigger to prevent updates and return an error message instead.

-- Create the trigger

CREATE TRIGGER PreventOrdersUpdate

ON Orders

INSTEAD OF UPDATE

AS

RAISERROR ('Updates on "Orders" table are not permitted.

Place a new order to add new products.', 16, 1);

* Create the trigger PreventNewDiscounts on the Discounts table.
* Set the trigger to prevent any rows being added to the Discounts table.

-- Create a new trigger

CREATE TRIGGER PreventNewDiscounts

ON Discounts

INSTEAD OF INSERT

AS

RAISERROR ('You are not allowed to add discounts for existing customers.

Contact the Sales Manager for more details.', 16, 1);

Create the new trigger following the company's requirements.

-- Create the trigger to log table info

CREATE TRIGGER TrackTableChanges

ON DATABASE

FOR CREATE\_TABLE,

ALTER\_TABLE,

DROP\_TABLE

AS

INSERT INTO TablesChangeLog (EventData, ChangedBy)

VALUES (EVENTDATA(), USER);

* Create a new trigger, PreventTableDeletion, to prevent table deletion.
* The trigger should roll back the firing statement, so the deletion does not happen.

-- Add a trigger to disable the removal of tables

CREATE TRIGGER PreventTableDeletion

ON DATABASE

FOR DROP\_TABLE

AS

RAISERROR ('You are not allowed to remove tables from this database.', 16, 1);

-- Revert the statement that removes the table

ROLLBACK;

* Create the INSERT statement that is going to fill in user details in the ServerLogonLog table.
* Select only the details for the situation when the session\_id is the same as the @@SPID (ID of the current user).

-- Save user details in the audit table

INSERT INTO ServerLogonLog (LoginName, LoginDate, SessionID, SourceIPAddress)

SELECT ORIGINAL\_LOGIN(), GETDATE(), @@SPID, client\_net\_address

-- The user details can be found in SYS.DM\_EXEC\_CONNECTIONS

FROM SYS.DM\_EXEC\_CONNECTIONS WHERE session\_id = @@SPID;

Create a new trigger at the server level that fires for logon events and saves user details into ServerLogonLog table.

-- Create a trigger firing when users log on to the server

CREATE TRIGGER LogonAudit

-- Use ALL SERVER to create a server-level trigger

ON ALL SERVER WITH EXECUTE AS 'sa'

-- The trigger should fire after a logon

FOR LOGON

AS

-- Save user details in the audit table

INSERT INTO ServerLogonLog (LoginName, LoginDate, SessionID, SourceIPAddress)

SELECT ORIGINAL\_LOGIN(), GETDATE(), @@SPID, client\_net\_address

FROM SYS.DM\_EXEC\_CONNECTIONS WHERE session\_id = @@SPID;

Which characteristics can be set when creating a logon trigger?

The trigger name / The set of actions to be performed

For logon triggers, you can only choose the trigger name and the actions to be performed by it. The trigger is always created at the server level, and it's an AFTER/FOR trigger

* Start creating the triggers report by gathering information about existing database triggers from the sys.triggers table.

-- Get the column that contains the trigger name

SELECT name AS TriggerName,

parent\_class\_desc AS TriggerType,

create\_date AS CreateDate,

modify\_date AS LastModifiedDate,

is\_disabled AS Disabled,

-- Get the column that tells if it's an INSTEAD OF trigger

is\_instead\_of\_trigger AS InsteadOfTrigger

FROM sys.triggers;

Include information about existing server-level triggers from the sys.server\_triggers table and order by trigger name.

-- Gather information about database triggers

SELECT name AS TriggerName,

parent\_class\_desc AS TriggerType,

create\_date AS CreateDate,

modify\_date AS LastModifiedDate,

is\_disabled AS Disabled,

is\_instead\_of\_trigger AS InsteadOfTrigger

FROM sys.triggers

UNION ALL

SELECT name AS TriggerName,

-- Get the column that contains the trigger type

parent\_class\_desc AS TriggerType,

create\_date AS CreateDate,

modify\_date AS LastModifiedDate,

is\_disabled AS Disabled,

0 AS InsteadOfTrigger

-- Gather information about server triggers

FROM sys.server\_triggers

-- Order the results by the trigger name

ORDER BY TriggerName;

* Enhance the report by including the trigger definitions. You can get a trigger's definition using the OBJECT\_DEFINITION function.

-- Gather information about database triggers

SELECT name AS TriggerName,

parent\_class\_desc AS TriggerType,

create\_date AS CreateDate,

modify\_date AS LastModifiedDate,

is\_disabled AS Disabled,

is\_instead\_of\_trigger AS InsteadOfTrigger,

-- Get the trigger definition by using a function

OBJECT\_DEFINITION (object\_id)

FROM sys.triggers

UNION ALL

-- Gather information about server triggers

SELECT name AS TriggerName,

parent\_class\_desc AS TriggerType,

create\_date AS CreateDate,

modify\_date AS LastModifiedDate,

is\_disabled AS Disabled,

0 AS InsteadOfTrigger,

-- Get the trigger definition by using a function

OBJECT\_DEFINITION (object\_id)

FROM sys.server\_triggers

ORDER BY TriggerName;

* Create a new trigger called CopyCustomersToHistory.
* Attach the trigger to the Customers table.
* Fire the trigger when rows are added or modified.
* Extract the information from the inserted special table.

-- Create a trigger to keep row history

CREATE TRIGGER CopyCustomersToHistory

ON Customers

-- Fire the trigger for new and updated rows

AFTER INSERT, UPDATE

AS

INSERT INTO CustomersHistory (CustomerID, Customer, ContractID, ContractDate, Address, PhoneNo, Email, ChangeDate)

SELECT CustomerID, Customer, ContractID, ContractDate, Address, PhoneNo, Email, GETDATE()

-- Get info from the special table that keeps new rows

FROM inserted;

* Create a new AFTER trigger on the Orders table.
* Set the trigger to fire for INSERT, UPDATE, and DELETE statements.

-- Add a trigger that tracks table changes

CREATE TRIGGER OrdersAudit

ON Orders

AFTER INSERT, UPDATE, DELETE

AS

DECLARE @Insert BIT = 0;

DECLARE @Delete BIT = 0;

IF EXISTS (SELECT \* FROM inserted) SET @Insert = 1;

IF EXISTS (SELECT \* FROM deleted) SET @Delete = 1;

INSERT INTO TablesAudit (TableName, EventType, UserAccount, EventDate)

SELECT 'Orders' AS TableName

,CASE WHEN @Insert = 1 AND @Delete = 0 THEN 'INSERT'

WHEN @Insert = 1 AND @Delete = 1 THEN 'UPDATE'

WHEN @Insert = 0 AND @Delete = 1 THEN 'DELETE'

END AS Event

,ORIGINAL\_LOGIN() AS UserAccount

,GETDATE() AS EventDate;

Create a new trigger, PreventProductChanges, that prevents any updates to the Products table.

-- Prevent any product changes

CREATE TRIGGER PreventProductChanges

ON Products

AFTER INSERT, UPDATE

AS

RAISERROR ('Updates of products are not permitted. Contact the database administrator if a change is needed.', 16, 1);

* Add a new trigger that fires for INSERT statements and checks if the order quantity can be fulfilled by the current stock.
* Raise an error if there's insufficient stock. Otherwise, perform an INSERT by making use of the inserted special table.

-- Create a new trigger to confirm stock before ordering

CREATE TRIGGER ConfirmStock

ON Orders

INSTEAD OF DELETE

AS

IF EXISTS (SELECT \*

FROM Products AS p

INNER JOIN inserted AS i ON i.Product = p.Product

WHERE p.Quantity < i.Quantity)

BEGIN

RAISERROR ('You cannot place orders when there is no stock for the order''s product.', 16, 1);

END

ELSE

BEGIN

INSERT INTO Orders (OrderID, Customer, Product, Price, Currency, Quantity, WithDiscount, Discount, OrderDate, TotalAmount, Dispatched)

SELECT OrderID, Customer, Product, Price, Currency, Quantity, WithDiscount, Discount, OrderDate, TotalAmount, Dispatched FROM inserted;

END;

Create a DatabaseAudit trigger on the database that fires for DDL\_TABLE\_VIEW\_EVENTS.

-- Create a new trigger

CREATE TRIGGER DatabaseAudit

-- Attach the trigger at the database level

ON Database

-- Fire the trigger for all tables/ views events

FOR DDL\_TABLE\_VIEW\_EVENTS

AS

INSERT INTO DatabaseAudit (EventType, DatabaseName, SchemaName, Object, ObjectType, UserAccount, Query, EventTime)

SELECT EVENTDATA().value('(/EVENT\_INSTANCE/EventType)[1]', 'NVARCHAR(50)') AS EventType

,EVENTDATA().value('(/EVENT\_INSTANCE/DatabaseName)[1]', 'NVARCHAR(50)') AS DatabaseName

,EVENTDATA().value('(/EVENT\_INSTANCE/SchemaName)[1]', 'NVARCHAR(50)') AS SchemaName

,EVENTDATA().value('(/EVENT\_INSTANCE/ObjectName)[1]', 'NVARCHAR(100)') AS Object

,EVENTDATA().value('(/EVENT\_INSTANCE/ObjectType)[1]', 'NVARCHAR(50)') AS ObjectType

,EVENTDATA().value('(/EVENT\_INSTANCE/LoginName)[1]', 'NVARCHAR(100)') AS UserAccount

,EVENTDATA().value('(/EVENT\_INSTANCE/TSQLCommand/CommandText)[1]', 'NVARCHAR(MAX)') AS Query

,EVENTDATA().value('(/EVENT\_INSTANCE/PostTime)[1]', 'DATETIME') AS EventTime;

* Create a new trigger called PreventDatabaseDelete.
* Attach the trigger at the server level.

-- Create a trigger to prevent database deletion

CREATE TRIGGER PreventDatabaseDelete

-- Attach the trigger at the server level

ON ALL SERVER

FOR DROP\_DATABASE

AS

PRINT 'You are not allowed to remove existing databases.';

ROLLBACK;

Remove the PreventNewDiscounts trigger attached to the Discounts table.

-- Remove the trigger

DROP TRIGGER PreventNewDiscounts;

Remove the PreventTableDeletion trigger attached to the database.

-- Remove the database trigger

DROP TRIGGER PreventTableDeletion

ON Database;

Remove the DisallowLinkedServers trigger attached at the server level.

-- Remove the server trigger

DROP TRIGGER DisallowLinkedServers

ON ALL SERVER;

* Modify the trigger definition and fix the typo without dropping and recreating the trigger.
* Add the missing word to the PRINT statement.

-- Fix the typo in the trigger message

ALTER TRIGGER PreventDiscountsDelete

ON Discounts

INSTEAD OF DELETE

AS

PRINT 'You are not allowed to remove data from the Discounts table.';

* Pause the trigger execution to allow the company to make the changes.

-- Pause the trigger execution

DISABLE TRIGGER PreventOrdersUpdate

ON Orders;

* Re-enable the disabled PreventOrdersUpdate trigger attached to the Orders table.

-- Resume the trigger execution

ENABLE TRIGGER PreventOrdersUpdate

ON Orders;

* Get the name, object\_id, and parent\_class\_desc for all the disabled triggers.

-- Get the disabled triggers

SELECT name,

object\_id,

parent\_class\_desc

FROM sys.triggers

WHERE is\_disabled = 1;

* Get the unmodified server-level triggers.
* An unmodified trigger's create date is the same as the modify date.

-- Check for unchanged server triggers

SELECT \*

FROM sys.triggers

WHERE create\_date = modify\_date;

* Use sys.triggers to extract information only about database-level triggers.

-- Get the database triggers

SELECT \*

FROM sys.triggers

WHERE parent\_class\_desc = 'DATABASE';

* Modify the PreventOrdersUpdate trigger.
* Set the trigger to fire when rows are updated in the Orders table.
* Add additional details about the trigger execution into the TriggerAudit table.

-- Modify the trigger to add new functionality

ALTER TRIGGER PreventOrdersUpdate

ON Orders

-- Prevent any row changes

INSTEAD OF UPDATE

AS

-- Keep history of trigger executions

INSERT INTO TriggerAudit (TriggerName, ExecutionDate)

SELECT 'PreventOrdersUpdate',

GETDATE();

RAISERROR ('Updates on "Orders" table are not permitted.

Place a new order to add new products.', 16, 1);

* Find the ID of the Orders table by using the sys.objects system view.

-- Get the table ID

SELECT object\_id AS TableID

FROM sys.objects

WHERE name = 'Orders';

* Find all the triggers attached to the Orders table by joining the first query with sys.triggers.
* Select the trigger name column.

-- Get the trigger name

SELECT t.name AS TriggerName

FROM sys.objects AS o

-- Join with the triggers table

INNER JOIN sys.triggers AS t ON t.parent\_id = o.object\_id

WHERE o.name = 'Orders';

* Filter the triggers fired for UPDATE statements, joining the previous query with sys.trigger\_events.
* Select the triggers and their firing statements by matching the object\_id columns from sys.triggers and sys.trigger\_events.

SELECT t.name AS TriggerName

FROM sys.objects AS o

INNER JOIN sys.triggers AS t ON t.parent\_id = o.object\_id

-- Get the trigger events

INNER JOIN sys.triggers AS te ON te.object\_id = t.object\_id

WHERE o.name = 'Orders'

-- Filter for triggers reacting to new rows

AND te.type\_desc = 'UPDATE';

Include the trigger definitions in your selection with the use of a standard SQL Server function.

SELECT t.name AS TriggerName,

OBJECT\_DEFINITION(t.object\_id) AS TriggerDefinition

FROM sys.objects AS o

INNER JOIN sys.triggers AS t ON t.parent\_id = o.object\_id

INNER JOIN sys.trigger\_events AS te ON te.object\_id = t.object\_id

WHERE o.name = 'Orders'

AND te.type\_desc = 'UPDATE';