DATA SOCIETY®

Introduction to SQL - Part 4

"One should look for what is and not what he thinks should be."
-Albert Einstein.

Warm up

Before we start, check out this article about common mistakes with SQL you would want to avoid:
https://towardsdatascience.com/dont-repeat-these-5-mistakes-with-sql-9f61d6f5324f

Welcome back!

In the last class we saw how joins and implemented various functions

In this class we will:

- Implement SQL subqueries
- Work with views and indexes
- Learn about stored procedures and SQL transactions

Module Completion Checklist

Objective	Complete
Implement SQL subqueries	
Create table views	
Create and edit table indexes	
Apply SQL transactions to data	
Define and identify metadata in SQL	
Create and apply stored procedures	

What is a subquery?

- A **subquery** is a query contained inside another SQL query
  - It's also called an **inner query**, since it's placed as part of another query called the **outer query**
  - Or, it can be called a **nested subquery**, since it is contained inside another query
- Subqueries are always enclosed inside parentheses and executed prior to the outer query
- The result of the subquery is passed to the outer query

Where can subqueries occur?

- A subquery may occur in a:
  - **SELECT** clause
  - **FROM** clause
  - **WHERE** clause
- A subquery can be nested inside a:
  - **SELECT** statement
  - **INSERT** statement
  - **UPDATE** statement
  - **SET** statement

Note: Subqueries are one way of writing a query. Sometimes they become more complex and can be avoided by writing simpler queries to perform the same operations.

Correlated and non-correlated subqueries

- A subquery can contain a reference to an object in the outer query – this is called an **outer reference**
- A **non-correlated subquery** does not contain an outer reference
  - It is executed once for the entire outer query
  - It can be executed independent of the outer query
  - It makes use of **IN, NOT IN** and **ALL, SOME, ANY** operators
- A **correlated subquery** contains an outer reference
  - It is executed for each row of the outer query
  - It cannot be executed independent of the outer query
  - It makes use of **EXISTS, NOT EXISTS** operators

IN - non-correlated subquery

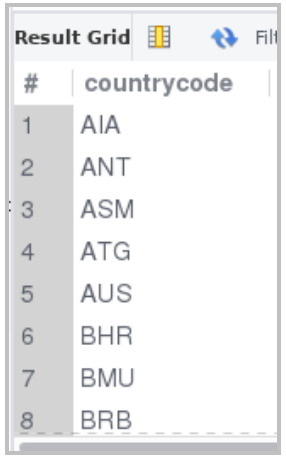
- Use the **IN** operator for checking the matching items in both subsets

```
-- Find all the countries that speak both English and Spanish.
SELECT DISTINCT countrycode FROM countrylanguage -- outer query attribute select
WHERE Language = 'English'                       -- outer query condition
AND                                              -- combine outer and inner query using and
countrycode IN                                   -- check set membership using IN
(SELECT countrycode FROM countrylanguage         -- inner query attribute select
WHERE Language = 'Spanish');                     -- inner query condition
```

NOT IN - non-correlated subquery

- Use the **NOT IN** operator for checking the items present in one subset but not in another

```
-- Find all the countries that speak English, but not Spanish.
SELECT DISTINCT countrycode FROM countrylanguage -- outer query attribute select
WHERE Language = 'English'                       -- outer query condition
AND                                              -- combine outer and inner query using and
countrycode NOT IN                               -- check set membership using NOT IN
(SELECT countrycode FROM countrylanguage         -- inner query attribute select
WHERE Language = 'Spanish');                     -- inner query condition
```



Set comparison - non-correlated subquery

- We can use a subquery after a comparison operator
- We can use all the comparison operators `<`, `<=`, `>`, `>=`, `<>`, `=` for comparing
- Set comparison uses **ALL** and **ANY** or **SOME** operators
- These operators compare value to every value returned by a subquery
- **ALL**
  - **ALL** returns TRUE only if the comparison is **TRUE for ALL** the values in the column that a subquery returns
- **ANY or SOME**
  - **SOME** is the alias of **ANY**
  - **ANY** returns **TRUE** only if the comparison is **TRUE for ANY or SOME** of the values in the column that a subquery returns

ALL - non-correlated subquery

```
-- Find the names of all cities whose population is greater than
-- that of all cities of the USA.
SELECT name FROM city WHERE                                  -- select attributes of outer query
population > ALL                                             -- comparison
(SELECT population FROM city WHERE countrycode = 'USA');     -- inner query selection
```

Result Grid

#	name
1	São Paulo
2	Jakarta
3	Mumbai (Bo…
4	Shanghai
5	Seoul
6	Ciudad de M…
7	Karachi
8	Istanbul

SOME or ANY - non-correlated subquery

```
-- Find the official languages of the country.
-- Use subquery approach to write the query.
SELECT Language, countrycode FROM countrylanguage                -- outer query
WHERE language = ANY                                             -- set comparison
(SELECT Language FROM countrylanguage WHERE IsOfficial = 'T');   -- inner query
```

Result Grid Filter Rows:

#	Language	countrycode
1	Dutch	ABW
2	English	ABW
3	Papiamento	ABW
4	Spanish	ABW
5	Dari	AFG
6	Pashto	AFG
7	Turkmenian	AFG
8	Uzbek	AFG

EXISTS - correlated subquery

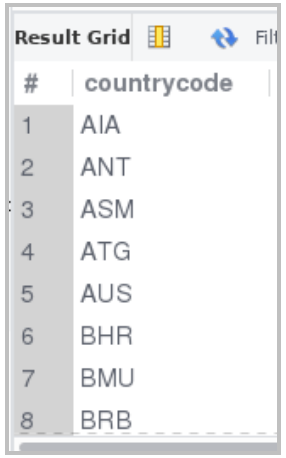
- The **EXISTS** construct returns **TRUE** if the argument subquery is not empty

```
-- Find all the countries that speak both English and Spanish.
SELECT countrycode FROM countrylanguage AS A  -- select countrycode in outer query
WHERE language = 'English'                    -- where the language is English
AND EXISTS                                    -- correlate the subquery using exists
(SELECT * FROM countrylanguage AS B           -- inner query selects all attributes
WHERE language = 'Spanish' AND                -- where language is Spanish
A.countrycode = B.countrycode);               -- combine using the join
```

NOT EXISTS - correlated subquery

- The **NOT EXISTS** construct returns **TRUE** if the argument subquery is empty

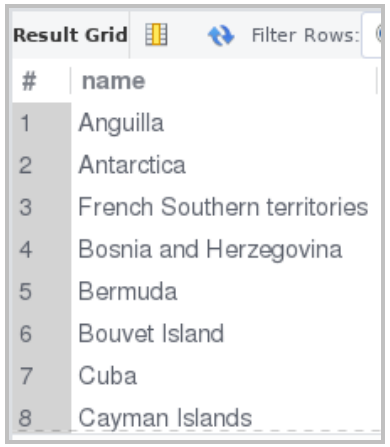
```
-- Find all the countries that speak English, but not Spanish.
SELECT countrycode FROM countrylanguage AS A  -- select countrycode in outer query
WHERE language = 'English'                    -- where the language is English
AND NOT EXISTS                                -- correlate the subquery using not exist
(SELECT * FROM countrylanguage AS B           -- inner query selects all attributes
WHERE language = 'Spanish' AND                -- where language is Spanish
A.countrycode = B.countrycode);               -- combine using the join
```



Other types of correlated subquery

- Correlated subqueries can also be written without making use of the **EXISTS / NOT EXISTS** operators

```
-- Find all the countries that speak fewer than two languages.
SELECT c.name FROM country AS c WHERE                    -- outer query
(SELECT COUNT(cl.language) FROM countrylanguage AS cl    -- inner query
WHERE c.code = cl.countrycode) < 2;                      -- correlate the query using join
```



Result Grid | Filter Rows:

#	name
1	Anguilla
2	Antarctica
3	French Southern territories
4	Bosnia and Herzegovina
5	Bermuda
6	Bouvet Island
7	Cuba
8	Cayman Islands

Subquery in SELECT clause

- Subqueries can be written in the **SELECT** clause as well

```
-- Find the difference in a city's population from the maximum populated city of the city table.
-- Arrange by the population difference.
SELECT Name, population,                              -- select attributes
((SELECT MAX(population) FROM city) - population)     -- subquery in select clause
AS population_difference FROM city                    -- column alias
order by population_difference;                       -- order by population difference
```

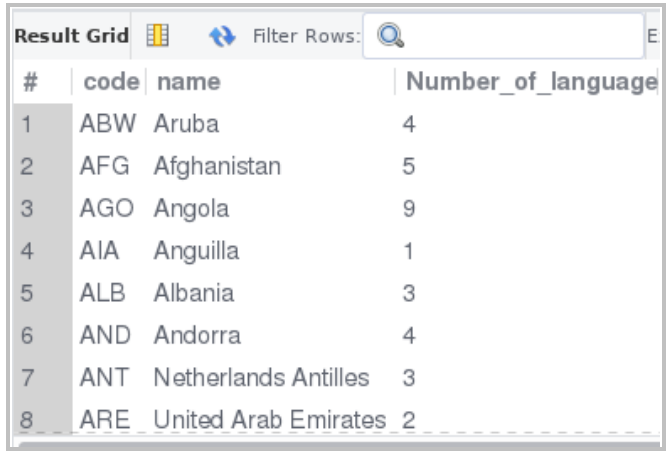
Result Grid | Filter Rows: | Export: | Wrap Cell

#	Name	population	population_differen
1	Mumbai (Bombay)	10500000	0
2	Seoul	9981619	518381
3	São Paulo	9968485	531515
4	Shanghai	9696300	803700
5	Jakarta	9604900	895100
6	Karachi	9269265	1230735
7	Istanbul	8787958	1712042
8	Ciudad de México	8591309	1908691

Subquery in FROM clause

- Subqueries can be written in the **FROM** clause as well

```
-- Find the number of languages spoken in each country.
SELECT c.code, c.name, cl.Number_of_languages        -- select attributes
FROM country AS c                                    -- outerquery table alias
INNER JOIN                                           -- join the tables
(SELECT countrycode, COUNT(*) AS Number_of_languages -- inner query
FROM countrylanguage                                 -- inner query table
GROUP BY countrycode)                                -- group by countrycode
AS cl                                                -- inner query table alias for referencing
ON c.code = cl.countrycode;                          -- join on country code
```



Result Grid | Filter Rows:

#	code	name	Number_of_language
1	ABW	Aruba	4
2	AFG	Afghanistan	5
3	AGO	Angola	9
4	AIA	Anguilla	1
5	ALB	Albania	3
6	AND	Andorra	4
7	ANT	Netherlands Antilles	3
8	ARE	United Arab Emirates	2

When to use subquery?

- Nested subqueries can often be confusing and complex
- Use subqueries only:
  - to replace complex join or union statements
  - if you want to apply the result of inner query to multiple outer queries
  - to structure the complex query in multiple logical parts for code maintenance

Note: Always remember that your database needs to perform additional steps on the background to execute a subquery, which is why the run time significantly increases if we use many nested subqueries!

Knowledge Check 1

Exercise 1

Module Completion Checklist

Objective	Complete
Implement SQL subqueries	✔
Create table views	
Create and edit table indexes	
Apply SQL transactions to data	
Define and identify metadata in SQL	
Create and apply stored procedures	

Views

- Views are **virtual data tables**; they are nothing but a SQL statement stored in the database with an associated name
- A view has columns and rows like a table. It can be dervied from one or more tables
- The tables from which the views are defined are called the **base tables**
- Views are useful when:
  - we do not want the user to view all the data in the database
  - we might want to take a specific set of columns from multiple tables and make it visible to specific users
  - we want to summarize data from various tables to generate reports

Use Case: You might want to alter the view of your database for a client, so that they can see all the production data and no sales data

CREATE OR REPLACE VIEW

- **Create a view:**

```
CREATE VIEW view_name(view_attributes) AS select_statement;
```

```
-- Create a view of country name, surface area, population, and official language.

CREATE VIEW Independent_Country_Details(country_name, country_area, -- name the view
country_population,  country_official_language) AS                  -- select attributes
SELECT c.Name, c.surfaceArea, c.population, cl.language             -- select values from base table
FROM country AS c, countrylanguage AS cl                            -- base tables alias
WHERE cl.isOfficial = 'T' and cl.countrycode = c.code;              -- base table condition
```

- **Alter a view:**

```
CREATE OR REPLACE VIEW view_name(view_attributes) AS select_statement;
```

```
-- Update the independent_country_details by removing surface area, and population.
-- Add a single column called population per area.

CREATE OR REPLACE VIEW Independent_Country_Details                          -- view name
(country_name, country_population_per_area, country_official_language) AS  -- view attributes
SELECT c.Name, (c.population/c.surfaceArea), cl.language                    -- select attributes
FROM country AS c, countrylanguage AS cl                                    -- base table alias
WHERE cl.isOfficial = 'T' and cl.countrycode = c.code;                      -- base table condition
```

SELECT - Display View Data

- **View the data**

```
-- Select from a view.
SELECT * FROM Independent_Country_Details;
```

	country_name	country_population_per_area	country_official_language
▶	Aruba	533.678756	Dutch
	Afghanistan	34.841816	Dari
	Afghanistan	34.841816	Pashto
	Anguilla	83.333333	English
	Albania	118.310839	Albaniana
	Andorra	166.666667	Catalan
	Netherlands Antilles	271.250000	Dutch

Updating a View

- Updating a view has certain limitations since views are virtual tables
- Every time a view gets updated, **the base table also gets updated**
- The restrictions are:
  - No aggregate functions are used
  - No **GROUP BY** or **HAVING** clause should be used
  - No subqueries in the **SELECT** or **FROM** clause
  - The subqueries in the **WHERE** clause do not refer to the table in the **FROM** clause
  - No utilization of **UNION, UNION ALL or DISTINCT**
  - The FROM clause contains at least one table or updatable view
  - The FROM clause uses only **INNER JOIN** if there is more than one table or view

UPDATE & DROP

- **Update a View**

```
-- Update the language of Albania to English/Albanian.

UPDATE Independent_Country_Details                    -- update the view
SET country_official_language =  'English/Albanian'   -- set the value
WHERE country_name = 'Albania';                       -- condition for set
```

	country_name	country_population_per_area	country_official_language
▶	Aruba	533.678756	Dutch
	Afghanistan	34.841816	Dari
	Afghanistan	34.841816	Pashto
	Anguilla	83.333333	English
	Albania	118.310839	English/Albanian
	Andorra	166.666667	Catalan
	Netherlands Antilles	271.250000	Dutch
	Netherlands Antilles	271.250000	Papiamento
	United Arab Emirates	29.198565	Arabic
	Argentina	13.318947	Spanish

- **Delete a View**

```
-- Delete the view Independent_Country_Details.

DROP VIEW Independent_Country_Details;
```

Index

- Indexes are used to retrieve data from the database **faster**
- When we insert data into a table, it does not get inserted in any particular order
- For example, if we need to search a particular `department_name` in the `department` table, it goes through each and every row to fetch the data
- Creating an index on `department_name` makes the query execute faster
- The user **cannot see** the index, but they **speed up the search**

Index Clauses

- Index clauses:
  - **ADD INDEX** clause creates an index on a table
  - **SHOW INDEX** clause shows an index on a table
  - **DROP INDEX** clause deletes an index on a table

CREATE, SHOW, and DROP INDEX

- **Create an index**

```
-- Add country name as an index to the country table.
ALTER TABLE country ADD INDEX country_name_idx(name); -- add an index
```

- **Show an index**

```
-- Display the index.
SHOW INDEX FROM country;  -- show the index
```

	Table	Non_unique	Key_name	Seq_in_index	Column_name	Collation	Cardinality	Sub_part	Packed	Null	Index_type	Comment	Index_comment	Visible
▶	country	0	PRIMARY	1	Code	A	239	NULL	NULL		BTREE			YES
	country	1	country_name_idx	1	Name	A	239	NULL	NULL		BTREE			YES

- **Delete an index**

```
-- Drop the index on the country table.
ALTER TABLE country DROP INDEX country_name_idx; -- drop the index
```

Transaction

- A **transaction** is a discrete unit of work that must be completely processed or not processed at all
- Until now, we saw only one user operating on the database
- What if multiple users operate on the same data at the same time?
- To avoid these types of issues, we use transactions in order to **maintain data integrity**
- Any transaction should maintain four properties, a.k.a **ACID** properties:
  - **A**tomicity - either all operations happen or none happen at all
  - **C**onsistency - makes sure that data integrity is maintained
  - **I**solation - enables transactions to take place independently of each other
  - **D**urability - ensures that the result of a committed transaction persists in case of system failure

Transaction Control Commands

- `START TRANSACTION;`
  - start a transaction
- `SAVEPOINT;`
  - the point in a transaction when you can roll the transaction back to a certain point without rolling back the entire transaction
- `COMMIT;`
  - used to save changes invoked by a transaction to the database
- `ROLLBACK;`
  - used to undo transactions that have not already been saved to the database
- `RELEASE SAVEPOINT;`
  - used to release the existing savepoints; once the savepoint is released, you cannot use `ROLLBACK` to undo transactions performed since last savepoint

Transaction Control Example

```
-- Start a transaction and update Latin as one of the languages in USA with 0.1%.
-- Update Greek as another language spoken with 0.02%.
-- Now we found that Greek language information is false, so rollback to the previous change.

SET AUTOCOMMIT = 0;                                                    -- set the autocommit to 0
START TRANSACTION;                                                     -- begin a transaction
SAVEPOINT my_savepoint;                                                -- check a savepoint
INSERT INTO countrylanguage VALUES('USA', 'Latin', 'F', 0.1); -- insert a row into countrylanguage
SAVEPOINT after_latin_addition_savepoint;                              -- check a savepoint
INSERT INTO countrylanguage VALUES('USA', 'Greek', 'F', 0.02);-- insert another row
ROLLBACK TO SAVEPOINT after_latin_addition_savepoint;                  -- rollback to the previous savepoint
COMMIT;                                                                -- commit the work
```

Transaction Control Example

- **Before transaction**

CountryCode	Language	IsOfficial	Percentage
USA	Chinese	F	0.6
USA	English	T	86.2
USA	French	F	0.7
USA	German	F	0.7
USA	Italian	F	0.6
USA	Japanese	F	0.2
USA	Korean	F	0.3
USA	Polish	F	0.3
USA	Portuguese	F	0.2
USA	Spanish	F	7.5
USA	Tagalog	F	0.4
USA	Vietnamese	F	0.2
NULL	NULL	NULL	NULL

- **After transaction**

CountryCode	Language	IsOfficial	Percentage
USA	Chinese	F	0.6
USA	English	T	86.2
USA	French	F	0.7
USA	German	F	0.7
USA	Italian	F	0.6
USA	Japanese	F	0.2
USA	Korean	F	0.3
USA	Latin	F	0.1
USA	Polish	F	0.3
USA	Portuguese	F	0.2
USA	Spanish	F	7.5
USA	Tagalog	F	0.4
USA	Vietnamese	F	0.2
NULL	NULL	NULL	NULL

Knowledge Check 2

Exercise 2

Module Completion Checklist

Objective	Complete
Implement SQL subqueries	✔
Create table views	✔
Create and edit table indexes	✔
Apply SQL transactions to data	✔
Define and identify metadata in SQL	
Create and apply stored procedures	

Metadata

- Metadata is information given to you about your dataset
- Every time we create a database object, the database server needs to record various pieces of information about that object
- All metadata is collectively called **data dictionary** or **system catalog**
- MySQL stores such information in a special database called **INFORMATION_SCHEMA**
- `INFORMATION_SCHEMA` stores all the information related to:
  - Table name
  - Column name
  - Column datatype
  - Default column values
  - NOT NULL column constraints
  - Primary key columns
  - Index names
  - Indexed columns
  - Foreign key details

SHOW DATABASES

- **View all the databases in MySQL**

```
-- View all databases.
SHOW DATABASES;
```

Database
company
employee
information_schema
mysql
performance_schema
sys
transaction
university
world

SHOW TABLES & COLUMNS

- **View all tables from the world database**

```
-- Show tables from a specific database.
SHOW TABLES FROM world;
```

	Tables_in_world
▶	city
	country
	countrylanguage

- **View all columns from the city table**

```
-- Show columns from a table.
SHOW COLUMNS FROM city;
```

	Field	Type	Null	Key	Default	Extra
▶	ID	int(11)	NO	PRI	NULL	auto_increment
	Name	char(35)	NO			
	CountryCode	char(3)	NO	MUL		
	District	char(20)	NO			
	Population	int(11)	NO		0	

Information_schema Database

- `INFORMATION_SCHEMA` is like another database which stores all metadata information in individual tables

```
-- Show tables from a database.
SHOW TABLES FROM INFORMATION_SCHEMA;
```

Tables_in_information_schema
ST_SPATIAL_REFERENCE_SYSTEMS
STATISTICS
TABLE_CONSTRAINTS
TABLE_PRIVILEGES
TABLES
TABLESPACES
TRIGGERS
USER_PRIVILEGES
VIEWS

- View all columns from `tables` table of `INFORMATION_SCHEMA` database

```
-- Show columns from a table.
SHOW COLUMNS FROM INFORMATION_SCHEMA.tables;
```

Field	Type	Null	Key	Default	Extra
TABLE_CATALOG	varchar(64)	YES		NULL	
TABLE_SCHEMA	varchar(64)	YES		NULL	
TABLE_NAME	varchar(64)	YES		NULL	
TABLE_TYPE	enum('BASE TABLE','VIEW','SYSTEM VIEW')	NO		NULL	
ENGINE	varchar(64)	YES		NULL	
VERSION	int(2)	YES		NULL	
ROW_FORMAT	enum('Fixed','Dynamic','Compressed','Redundan...	YES		NULL	
TABLE_ROWS	bigint(21) unsigned	YES		NULL	
AVG_ROW_LENGTH	bigint(21) unsigned	YES		NULL	

Information_schema Database

- Let's try to view the tables about our `world` database from the `INFORMATION_SCHEMA`

```
-- Show information about tables in world database.
SELECT table_name, table_type      -- select table name and type
FROM INFORMATION_SCHEMA.tables     -- from the `tables` table of information schema
WHERE table_schema = 'world';      -- from the world database
```

TABLE_NAME	TABLE_TYPE
city	BASE TABLE
country	BASE TABLE
countrylanguage	BASE TABLE

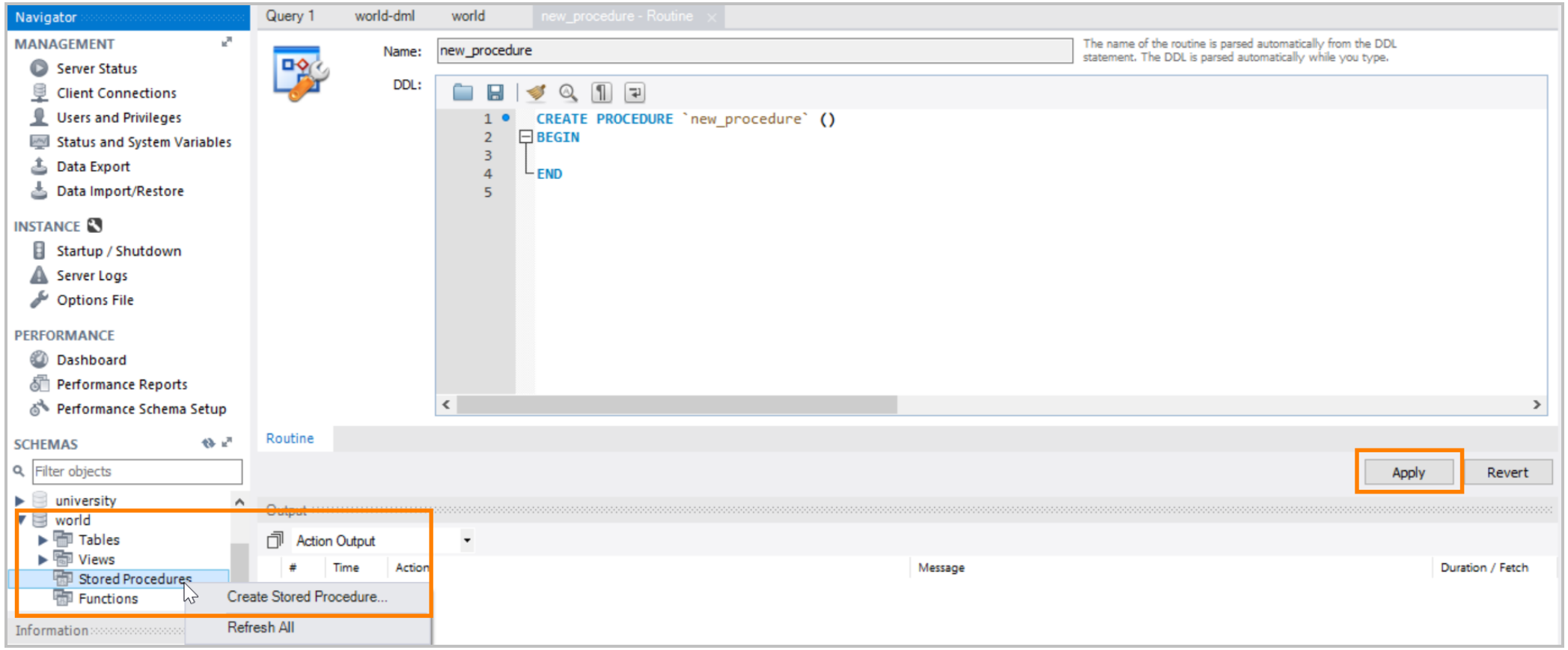
Stored procedures

- Stored procedures are SQL statements that can be saved and reused
- If we want to perform the same operation very frequently, we can use a **stored procedure**

Use case: We can create a stored procedure to find the top 5 most populated countries and use it compare their GNPs

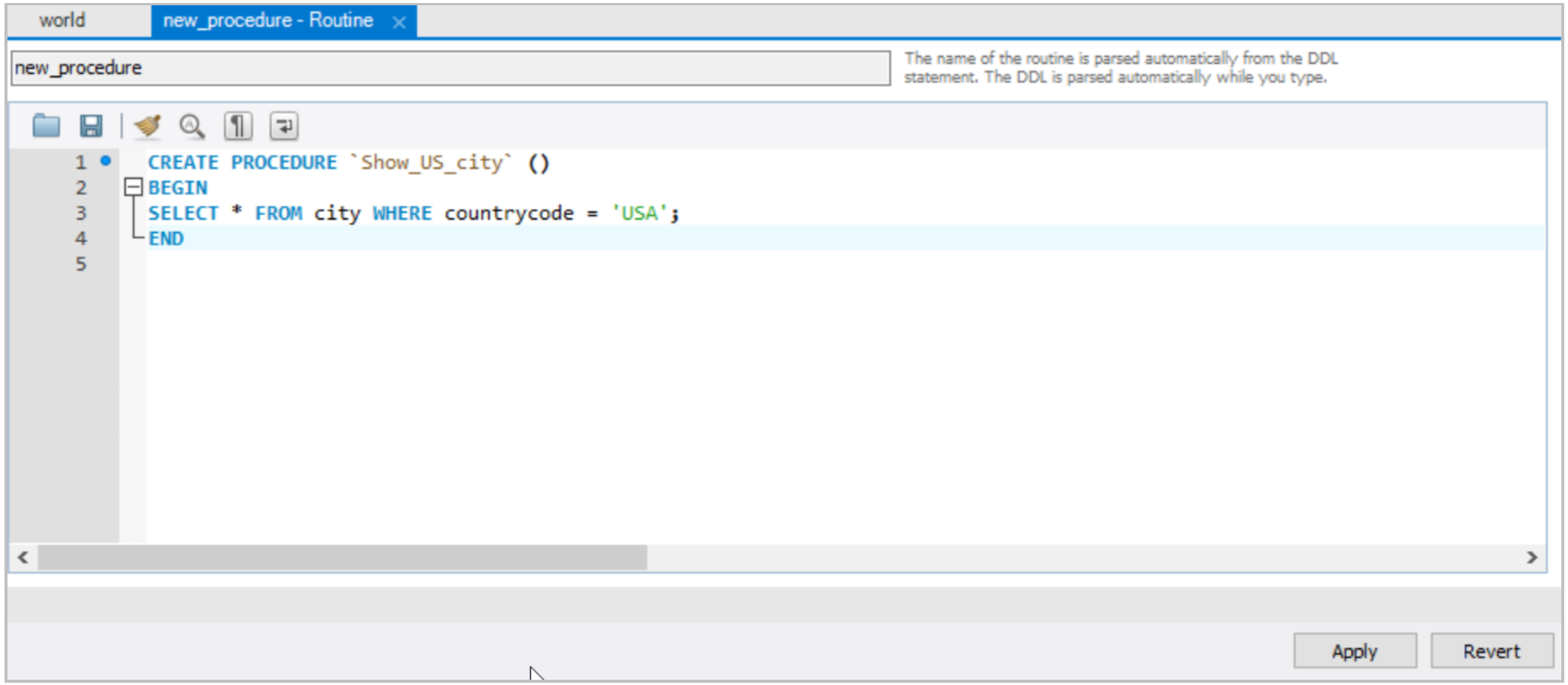
Stored procedures

- Let's create a procedure to display all the details of the US cities
- In the world schema, go to **Stored Procedures** and right click it
- Select `create Stored Procedures`



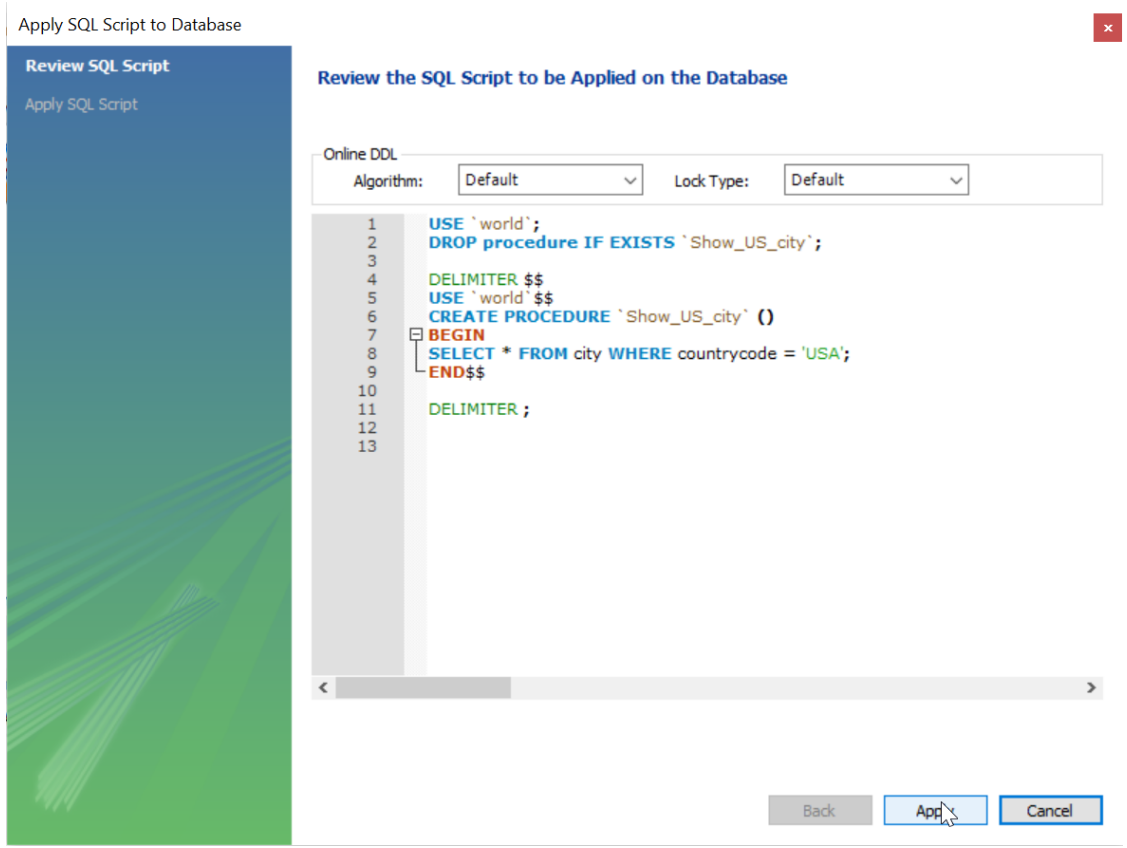
Stored procedures

- Enter the SQL statement(s) between `BEGIN` and `END` keywords



Stored procedures

- Click **apply** at the bottom of the dialogue window with generated code



Stored procedures

- Let's call the saved procedure

```
-- Call newly created stored procedure.
CALL Show_US_city;
```

ID	Name	CountryCode	District	Population
3793	New York	USA	New York	8008278
3794	Los Angeles	USA	California	3694820
3795	Chicago	USA	Illinois	2896016
3796	Houston	USA	Texas	1953631
3797	Philadelphia	USA	Pennsylvania	1517550
3798	Phoenix	USA	Arizona	1321045
3799	San Diego	USA	California	1223400
3800	Dallas	USA	Texas	1188580

Knowledge check 3

Exercise 3

Module Completion Checklist

Objective	Complete
Implement SQL subqueries	✔
Create table views	✔
Create and edit table indexes	✔
Apply SQL transactions to data	✔
Define and identify metadata in SQL	✔
Create and apply stored procedures	✔

SQL Summary

- SQL is a powerful query language that allows you to to define, manipulate and control data structures within your database
- SQL's querying ability makes data more accessible to non-programmers
- SQL allows you to answer questions about your data easily
- SQL lets you create multiple views of databases and data strictures for various users

- **What did you enjoy learning about SQL?**
- **How do you think it can impact that work you already do?**

This completes our module

Congratulations!