DATA SOCIETY®

Introduction to SQL - day 3

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

Module completion checklist

Objective	Complete
Demonstrate working with temporal data	
Apply aggregate functions	
Generating groups	
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	

'World' database introduction

In this module, we will be using the world database. The world.sql file contains data for a world database. This file came from the MYSQL website. Today we will be using various functions to understand our data in more in-depth ways.

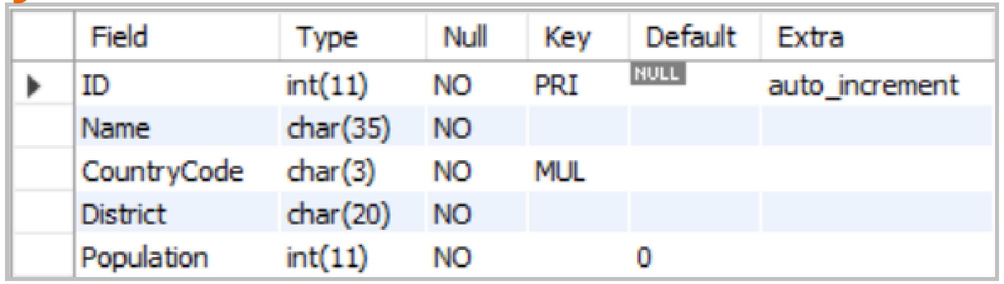
- In this module, we will be creating a database called world
- The `world.sql` file contains data for a world database
- This file came from the MYSQL website
- Your manager would now like for you to pull particular dates to organize time-sensitive data

'World' database schema

- We will use the same **world** database in this module to study MySQL

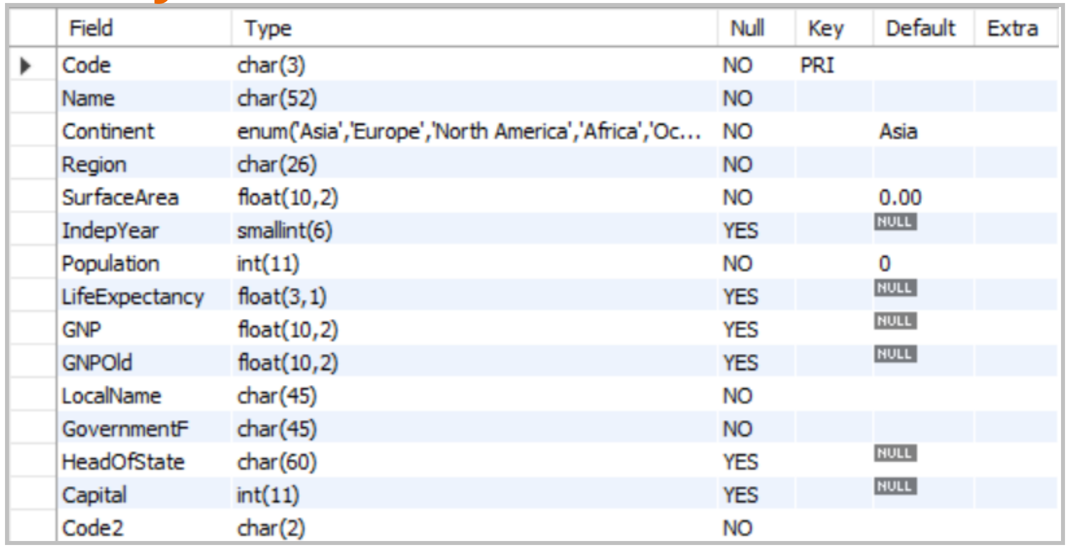
```
USE world;
```

city table



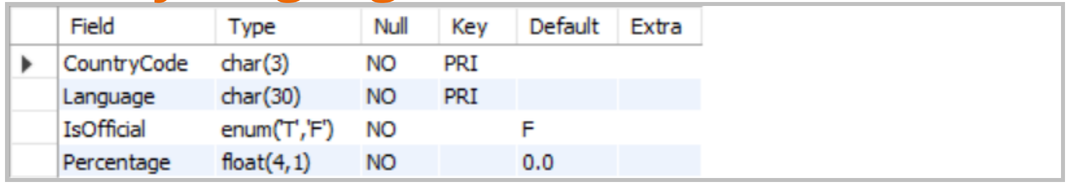
	Field	Type	Null	Key	Default	Extra
▶	ID	int(11)	NO	PRI	NULL	auto_increment
	Name	char(35)	NO			
	CountryCode	char(3)	NO	MUL		
	District	char(20)	NO			
	Population	int(11)	NO		0	

country table



	Field	Type	Null	Key	Default	Extra
▶	Code	char(3)	NO	PRI		
	Name	char(52)	NO			
	Continent	enum('Asia','Europe','North America','Africa','Oc...	NO		Asia	
	Region	char(26)	NO			
	SurfaceArea	float(10,2)	NO		0.00	
	IndepYear	smallint(6)	YES		NULL	
	Population	int(11)	NO		0	
	LifeExpectancy	float(3,1)	YES		NULL	
	GNP	float(10,2)	YES		NULL	
	GNPOld	float(10,2)	YES		NULL	
	LocalName	char(45)	NO			
	GovernmentF	char(45)	NO			
	HeadOfState	char(60)	YES		NULL	
	Capital	int(11)	YES		NULL	
	Code2	char(2)	NO			

country language table



	Field	Type	Null	Key	Default	Extra
▶	CountryCode	char(3)	NO	PRI		
	Language	char(30)	NO	PRI		
	IsOfficial	enum('T','F')	NO		F	
	Percentage	float(4,1)	NO		0.0	

Date format

Format	Description
`%M`	month name (January to December)
`%m`	month numeric (0 to 12)
`%d`	day numeric (01 to 31)
`%j`	day of the year (001 to 366)
`%W`	weekday name (Sunday to Saturday)
`%Y`	year (4 digit numeric)
`%y`	year (two digit numeric)
`%H`	hour (00 to 23)
`%h`	hour (00 to 12)
`%i`	minutes (00 to 59)
`%s`	seconds (00 to 59)
`%f`	microseconds (000000 to 999999)
`%p`	A.M. or P.M.

Date functions

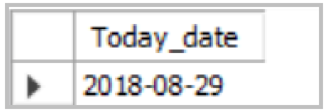
- There are built-in functions in MySQL for date manipulation
- Date functions: ***https://dev.mysql.com/doc/refman/8.0/en/date-and-time-functions.html***
- Here are some of the most commonly used functions:

Function	Description
CURDATE	Returns the current date
ADDTIME	Returns the time after a certain time interval is added
DATEDIFF	Returns the difference in days between the two date values
LASTDAY	Returns the last day of the month
DAYNAME	Returns weekday name for the date
EXTRACT	Extracts parts from the date

CURDATE & ADDTIME

- To find today's date, use the **CURDATE** function

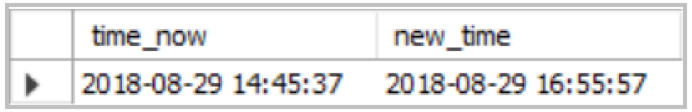
```sql
-- Get current date and display
-- as Today_date.
SELECT CURDATE() AS Today_date;
```



	Today_date
▶	2018-08-29

- To add time to the existing time, use the **ADDTIME** function
- `ADDTIME(start_value, time_to_add)`

```sql
-- Add 2 hours, 10 minutes and 20 seconds
-- to the current time.
SELECT CURRENT_TIMESTAMP()
AS time_now,
ADDTIME(CURRENT_TIMESTAMP(), "2:10:20")
AS new_time;
```

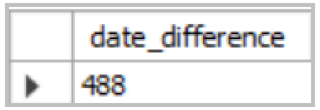


	time_now	new_time
▶	2018-08-29 14:45:37	2018-08-29 16:55:57

DATEDIFF & LASTDAY

- Use the **DATEDIFF** function to find the number of days between the two given dates

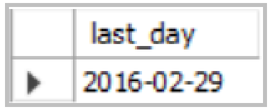
```
-- Find number of days between September 12th
-- 2018 and May 12th 2017.
SELECT DATEDIFF("2018-09-12", "2017-05-12")
AS date_difference;
```



	date_difference
▶	488

- Use the **LASTDAY** function to get the last day of the month

```
-- Find the last day of February in 2016.
SELECT LAST_DAY("2016-02-06") AS last_day;
```

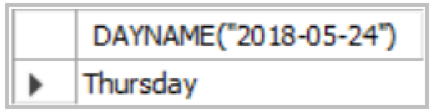


	last_day
▶	2016-02-29

DAYNAME & EXTRACT

- Use the **DAYNAME** function to return the name of the day of the week

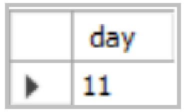
```
-- Find the day name of May 24th 2018.
SELECT DAYNAME("2018-05-24");
```



DAYNAME("2018-05-24")
Thursday

- Use the **EXTRACT** function to extract parts of the date

```
-- Extract the day part from
-- '2018-05-11 22:12:29'.
SELECT EXTRACT(DAY FROM '2018-05-11 22:12:29')
AS day;
```

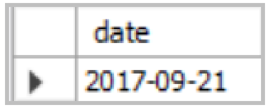


day
11

CAST or CONVERT

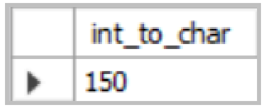
- It is possible to convert one data type to another
- We can use the function **CAST** or **CONVERT** for conversion
- Cast function: ***https://dev.mysql.com/doc/refman/8.0/en/cast-functions.html***

```sql
-- Convert the string "2017-09-21" to date type.
SELECT CAST("2017-09-21" AS DATE) AS date;
```



	date
▶	2017-09-21

```sql
-- Convert the int 150 to char type.
SELECT CONVERT(150, CHAR) AS int_to_char;
```



	int_to_char
▶	150

Module completion checklist

Objective	Complete
Demonstrate working with temporal data	✔
Apply aggregate functions	
Generating groups	
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	

Aggregate functions

- **Aggregate functions** take a collection of values as input and return one value as the output
- There are five built-in aggregate functions

Function	Definition
MAX	Returns the maximum value
MIN	Returns the minimum value
AVG	Returns the average value
SUM	Returns the sum of values
COUNT	Returns the number of observations

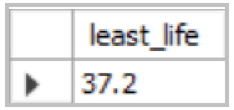
Note: These aggregate functions ignore NULL values!

- Your manager would like for you to make a report with variables that are aggregated to create a more detailed and summarized insights

MIN & MAX

- Use the **MIN** function to find the minimum in a column

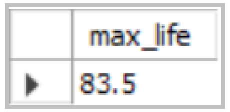
```
-- Find the minimum life expectancy
-- from the country table.
SELECT MIN(lifeexpectancy)
AS least_life  FROM country;
```



	least_life
▶	37.2

- Use the **MAX** function to find the maximum in a column

```
-- Find the maximum life expectancy
-- from the country table.
SELECT MAX(lifeexpectancy)
AS max_life  FROM country;
```

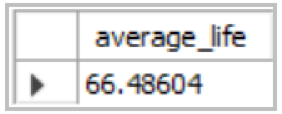


	max_life
▶	83.5

AVG & SUM

- Use the **AVG** function to find the average in a column

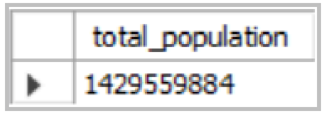
```
-- Find the average life expectancy
-- from the country table.
SELECT AVG(lifeexpectancy)
AS average_life  FROM country;
```



	average_life
▶	66.48604

- Use the **SUM** function to find the sum of a column

```
-- Find the total population
-- across all cities.
SELECT SUM(population)
AS total_population FROM city;
```

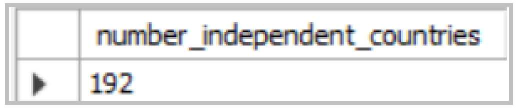


	total_population
▶	1429559884

COUNT

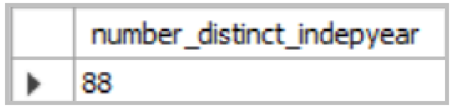
- Use the **COUNT** function to find the total number of observations in the column
- We can also use **DISTINCT** to find the total number of distinct observations
- `COUNT(*)` counts the total number of rows in the table

```
-- Find the number of countries who got their independence.
SELECT COUNT(indepyear) AS number_independent_countries FROM country;
```



	number_independent_countries
▶	192

```
-- Find  the distinct number of indepyear from the country table.
SELECT COUNT(DISTINCT indepyear) AS number_distinct_indepyear FROM country;
```

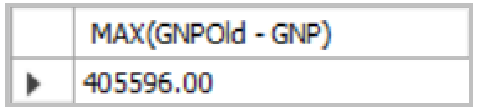


	number_distinct_indepyear
▶	88

Using expressions

- We can also build expressions to use in these aggregate functions

```
-- Find  the maximum difference between the gnpold and gnp in the country table.
SELECT MAX(GNPOld - GNP) from country;
```



	MAX(GNPOld - GNP)
▶	405596.00

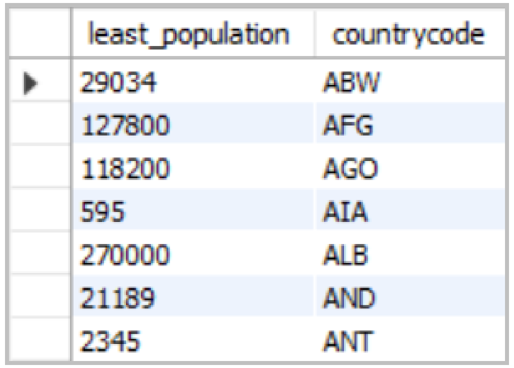
Module completion checklist

Objective	Complete
Demonstrate working with temporal data	✔
Apply aggregate functions	✔
Generating groups	
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	

GROUP BY

- People are not interested in looking at raw data
- In order to analyze the data better, it is useful to form groups
- The aggregate functions are better used when we group the data
- Example: Consider finding the value of the least populated city by country instead of the least populated city overall

```
-- Find the least population of a city for each country.
SELECT MIN(population) AS least_population, countrycode FROM city -- select attributes
GROUP BY CountryCode;                                             -- group by countrycode
```

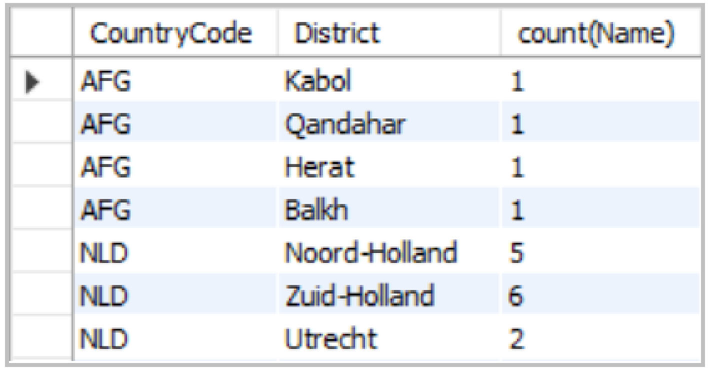


	least_population	countrycode
▶	29034	ABW
	127800	AFG
	118200	AGO
	595	AIA
	270000	ALB
	21189	AND
	2345	ANT

GROUP BY - multi column grouping

- We can also group data by more than one column

```
-- Find the number of cities in each district in each country.
SELECT CountryCode, District,     -- select the attributes
count(Name) FROM city             -- find the count
GROUP BY countrycode, district;   -- group by district and countrycode
```

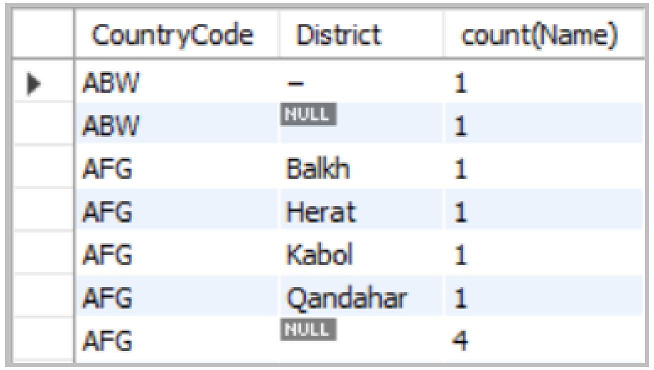


	CountryCode	District	count(Name)
▶	AFG	Kabol	1
	AFG	Qandahar	1
	AFG	Herat	1
	AFG	Balkh	1
	NLD	Noord-Holland	5
	NLD	Zuid-Holland	6
	NLD	Utrecht	2

GROUP BY - ROLLUP

- In multi-column grouping, if we want to perform the additional operation of finding the total value of each group, use the **ROLLUP** clause
- For example, in addition to grouping by district and countrycode, if we want to find the total number of cities in each country, use **ROLLUP** clause

```
SELECT CountryCode, District,                       -- select the attributes
COUNT(Name) FROM city                               -- find count of cities
GROUP BY countrycode, district WITH ROLLUP;         -- use rollup to find grand total
```

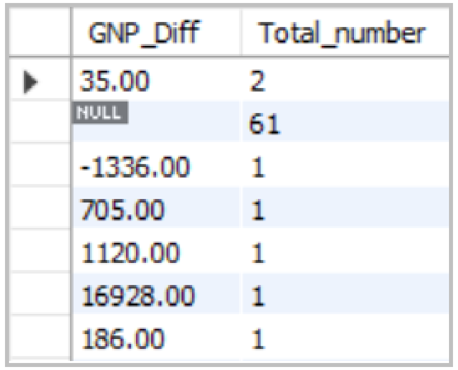


CountryCode	District	count(Name)
ABW	–	1
ABW	NULL	1
AFG	Balkh	1
AFG	Herat	1
AFG	Kabol	1
AFG	Qandahar	1
AFG	NULL	4

GROUP BY - expression

- Sometimes we might want to group by some value, which is not present in the data table

```
-- Find the number of countries who have the same GNP difference.
SELECT (GNP-GNPOld) AS GNP_Diff,     -- gnp difference expression
COUNT(*) Total_number FROM country   -- count number of countries
GROUP BY (GNP-GNPOld);               -- group by gnp difference
```

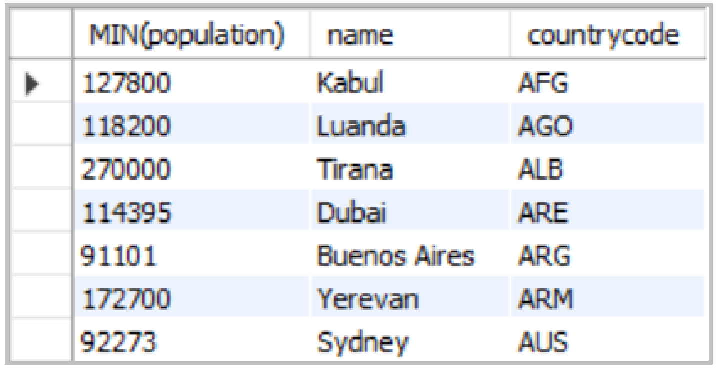


	GNP_Diff	Total_number
▶	35.00	2
	NULL	61
	-1336.00	1
	705.00	1
	1120.00	1
	16928.00	1
	186.00	1

GROUP BY & HAVING - Filter groups

- If we need to filter out the unwanted groups, use the **HAVING** clause

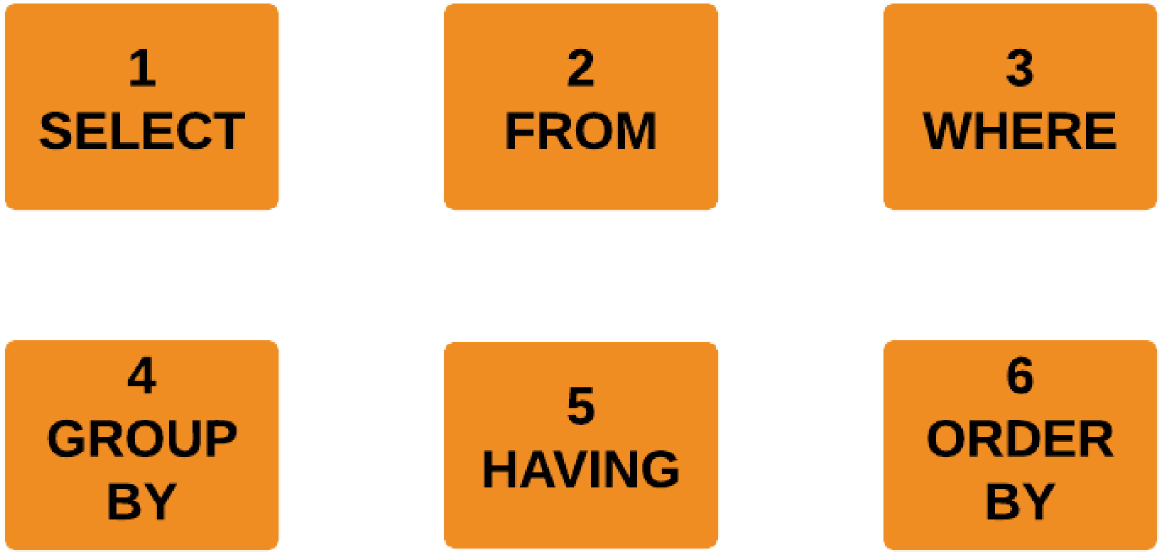
```
-- Find the least populated city from each country;
-- ignore if the population is less than 30000
SELECT MIN(population), name, countrycode  -- select the attributes
FROM city GROUP BY countrycode, name       -- group by countrycode, name
HAVING MIN(population) > 30000;            -- filter out if population is less than 30000
```



	MIN(population)	name	countrycode
▶	127800	Kabul	AFG
	118200	Luanda	AGO
	270000	Tirana	ALB
	114395	Dubai	ARE
	91101	Buenos Aires	ARG
	172700	Yerevan	ARM
	92273	Sydney	AUS

Order of the clause matters!

- Always remember that the order of the clauses matters!



1 SELECT

2 FROM

3 WHERE

4 GROUP BY

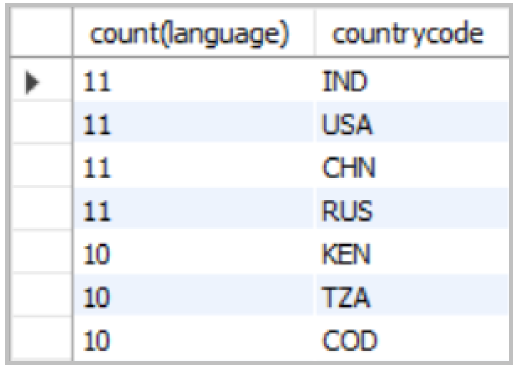
5 HAVING

6 ORDER BY

Example of a complex query

- Let's write a complex query to know how to make use of all the clauses

```
-- Find the number of unofficial languages for each country and arrange them in descending order.
-- Remove the entry if the number of unofficial languages is one.
SELECT COUNT(language), countrycode  -- select count of the languages
FROM countrylanguage                 -- from the table countrylanguage
WHERE IsOfficial != 'T'              -- filter out if the language is official
GROUP BY countrycode                 -- group by country
HAVING COUNT(language) > 1           -- remove if number is just one
ORDER BY COUNT(language) DESC;       -- order by the count
```



	count(language)	countrycode
▶	11	IND
	11	USA
	11	CHN
	11	RUS
	10	KEN
	10	TZA
	10	COD

Knowledge check 1

Exercise 1

Module completion checklist

Objective	Complete
Demonstrate working with temporal data	✔
Apply aggregate functions	✔
Generating groups	✔
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 nested subquery?

- **Subquery** is a query contained inside another SQL query
- It is called an **inner query** placed as part of another query called **outer query**
- Also called **nested subquery** since it is contained inside another query
- It is 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.

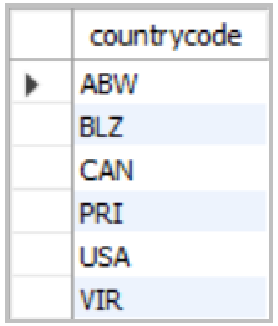
Correlated and noncorrelated subqueries

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

IN - noncorrelated subquery

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

```
-- 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
```

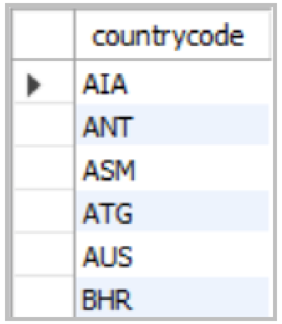


countrycode
ABW
BLZ
CAN
PRI
USA
VIR

NOT IN - noncorrelated 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
```



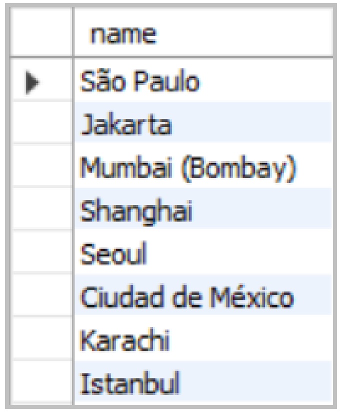
	countrycode
▶	AIA
	ANT
	ASM
	ATG
	AUS
	BHR

Set comparison - noncorrelated 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 - noncorrelated subquery

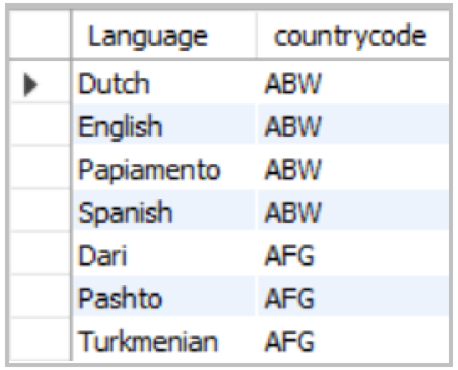
```
-- Find the names of all cities whose population is greater than
-- that of all cities of the USA.
SELECT name FROM city WHERE                              -- select attributes of outer query
population > ALL                                         -- comparison
(SELECT population FROM city WHERE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



name
São Paulo
Jakarta
Mumbai (Bombay)
Shanghai
Seoul
Ciudad de México
Karachi
Istanbul

SOME or ANY - noncorrelated 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
```

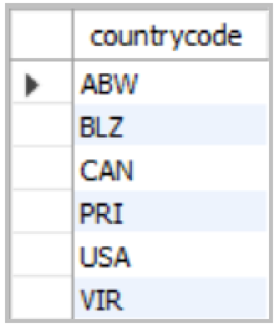


	Language	countrycode
▶	Dutch	ABW
	English	ABW
	Papiamento	ABW
	Spanish	ABW
	Dari	AFG
	Pashto	AFG
	Turkmenian	AFG

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
```

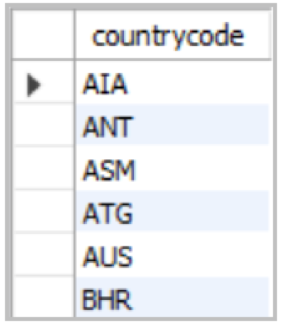


countrycode
ABW
BLZ
CAN
PRI
USA
VIR

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
```

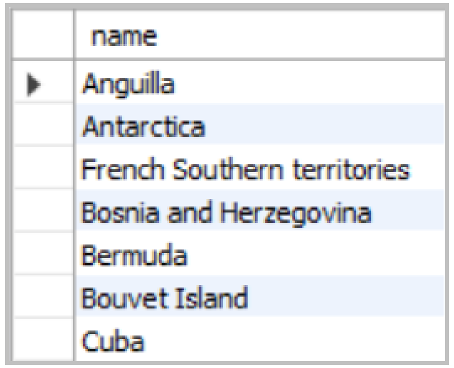


	countrycode
▶	AIA
	ANT
	ASM
	ATG
	AUS
	BHR

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
```

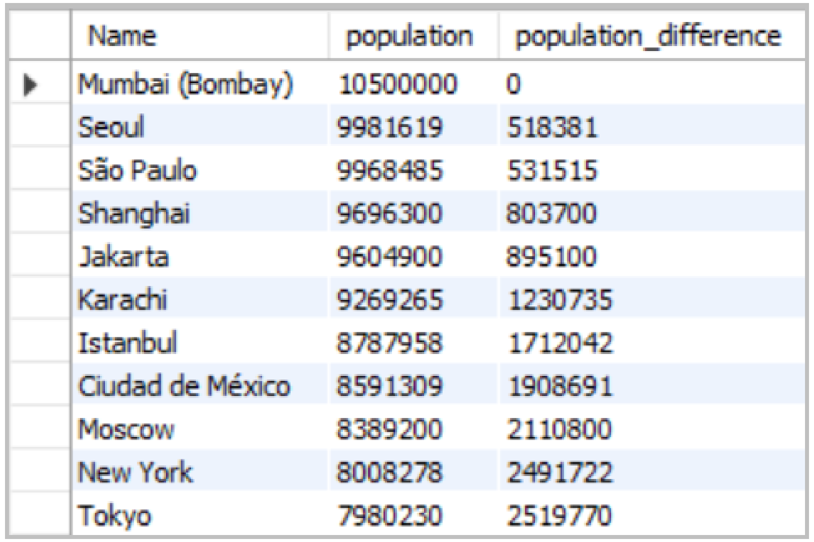


name
Anguilla
Antarctica
French Southern territories
Bosnia and Herzegovina
Bermuda
Bouvet Island
Cuba

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
```

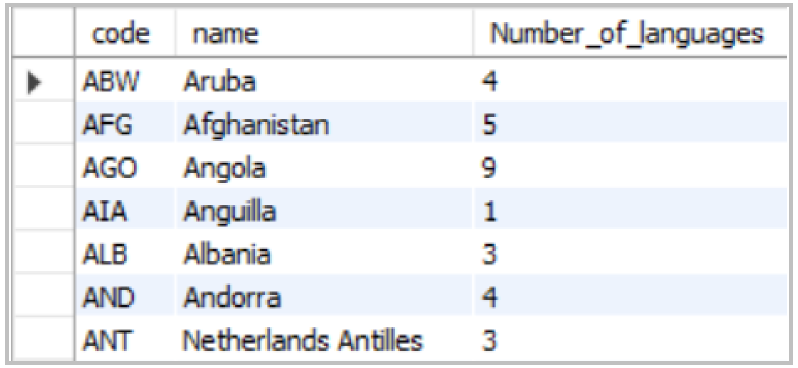


	Name	population	population_difference
▶	Mumbai (Bombay)	10500000	0
	Seoul	9981619	518381
	São Paulo	9968485	531515
	Shanghai	9696300	803700
	Jakarta	9604900	895100
	Karachi	9269265	1230735
	Istanbul	8787958	1712042
	Ciudad de México	8591309	1908691
	Moscow	8389200	2110800
	New York	8008278	2491722
	Tokyo	7980230	2519770

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
```



code	name	Number_of_languages
ABW	Aruba	4
AFG	Afghanistan	5
AGO	Angola	9
AIA	Anguilla	1
ALB	Albania	3
AND	Andorra	4
ANT	Netherlands Antilles	3

When to use subquery?

- Nested subqueries can often be confusing and complex
- Use the 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 2

Exercise 2

Module completion checklist

Objective	Complete
Demonstrate working with temporal data	✔
Apply aggregate functions	✔
Generating groups	✔
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 the table
- 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 summarise certain data to generate reports
- **Your manager wants you to alter the view of your database for a client to view**

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, -- select attributes
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;              -- condition in base table
```

- **Alter a view:**

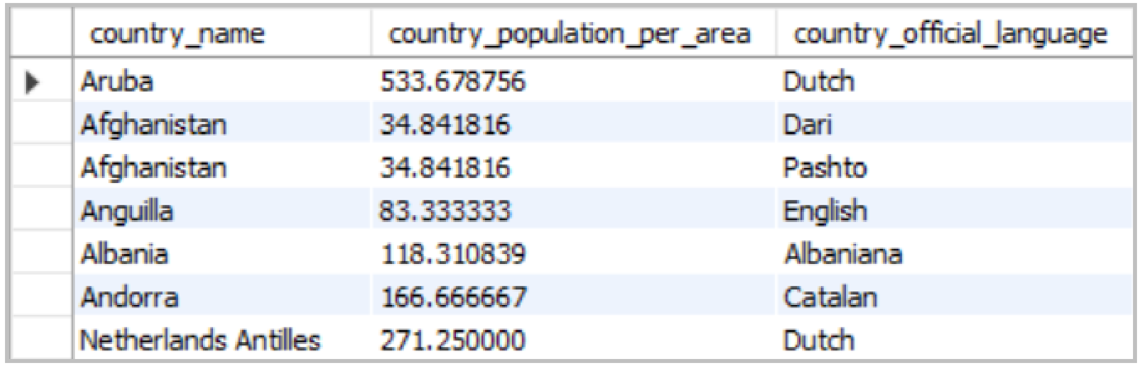
CREATE OR REPLACE VIEW view_name(view_attributes) AS select_statement;

```
-- 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
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

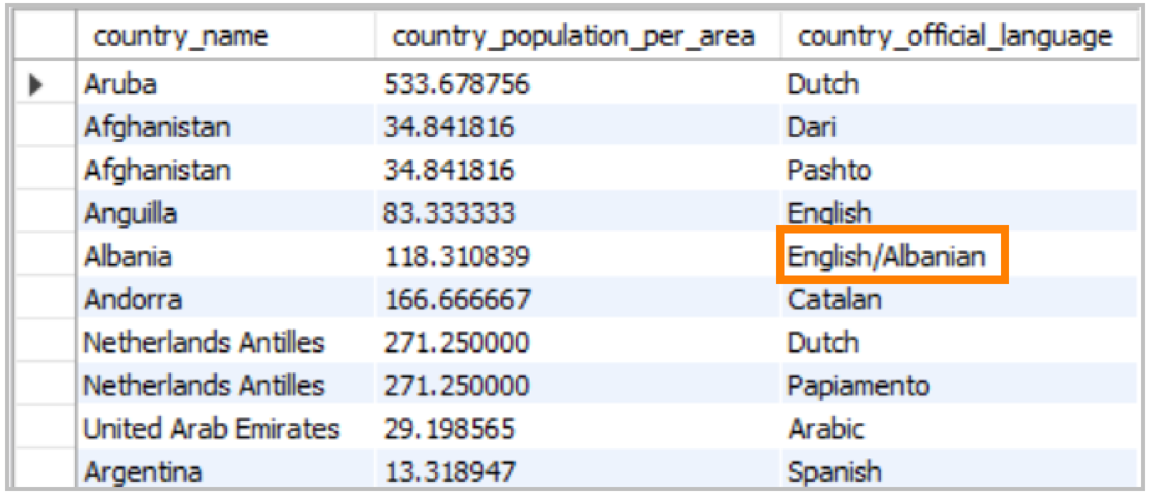
Updating a view

- 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
```

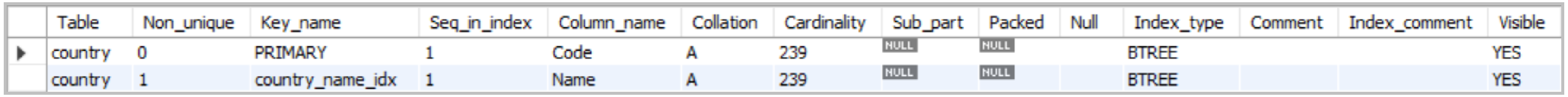


	Table	Non_unique	Key_name	Seq_in_index	Column_name	Collation	Cardinality	Sub_part	Packed	Null	Index_type	Comment	Index_comment	Visible
▶	country	0	PRIMARY	1	Code	A	239	NULL	NULL		BTREE			YES
	country	1	country_name_idx	1	Name	A	239	NULL	NULL		BTREE			YES

- **Delete an index**

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

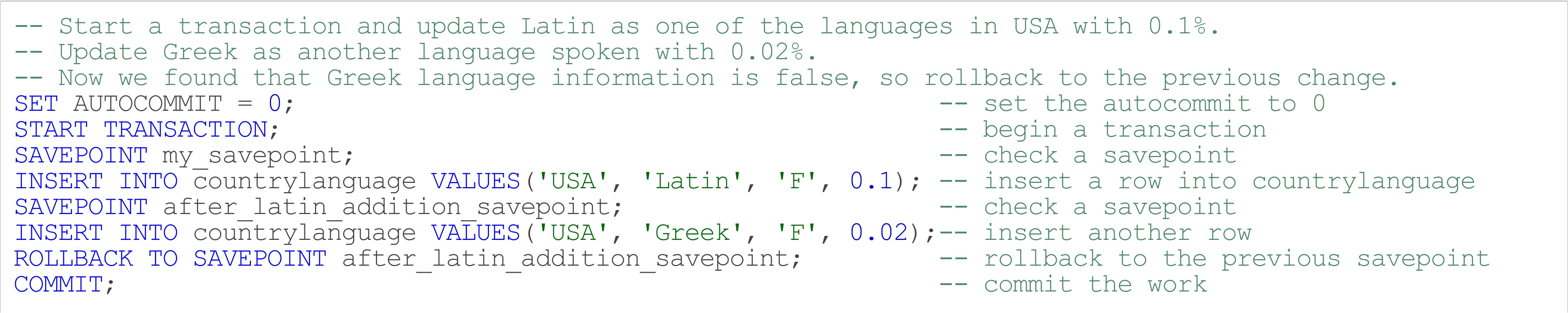
Transaction

- 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

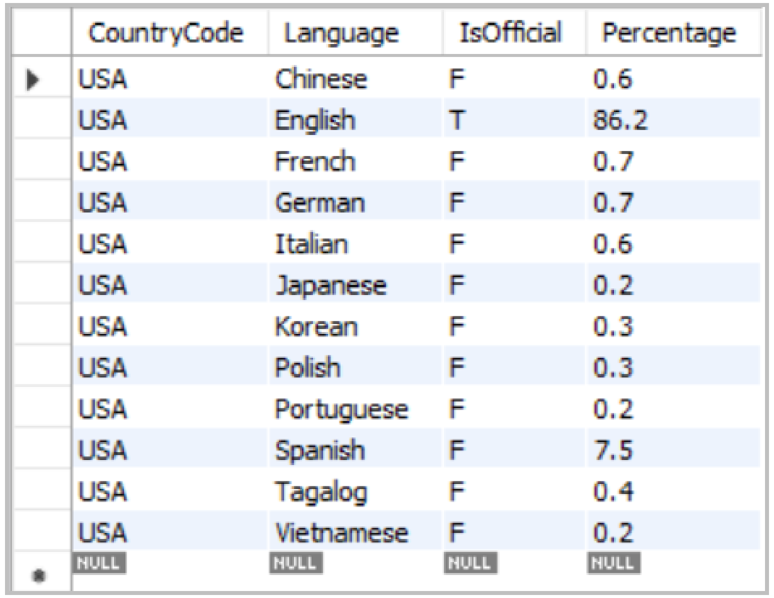
Transaction control example



```
-- 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
```

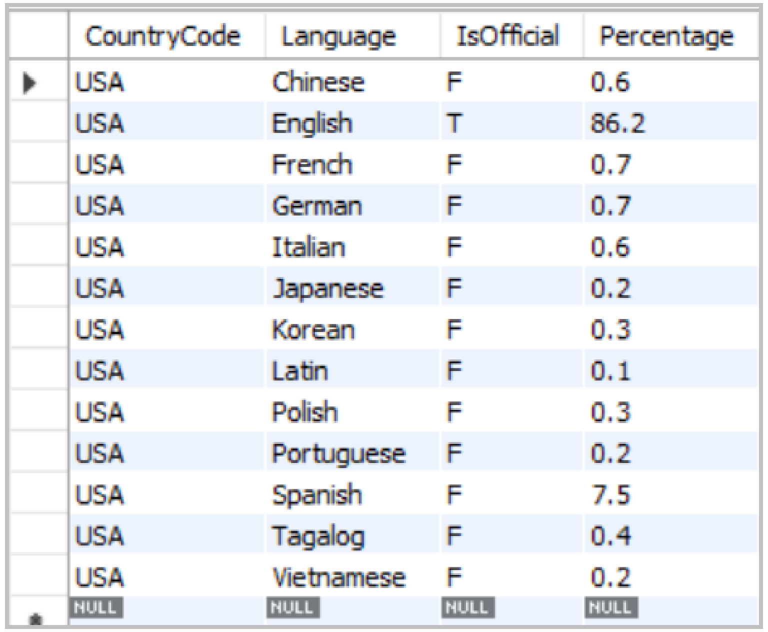
Transaction control example

- **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 3

Exercise 3

Module completion checklist

Objective	Complete
Demonstrate working with temporal data	✔
Apply aggregate functions	✔
Generating groups	✔
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	

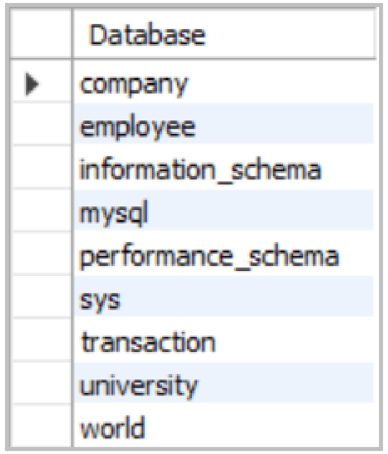
Metadata

- 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;
```

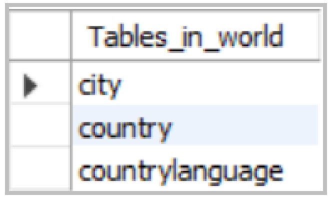


Database
company
employee
information_schema
mysql
performance_schema
sys
transaction
university
world

SHOW TABLES & COLUMNS

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

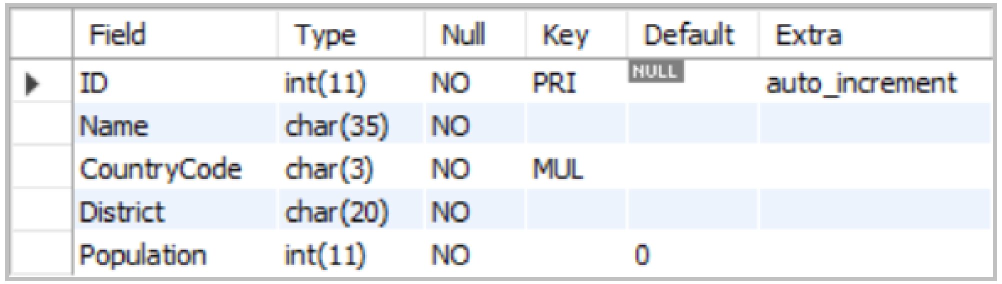
```
-- Show tables from a specific database.
SHOW TABLES FROM world;
```



Tables_in_world
city
country
countrylanguage

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

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

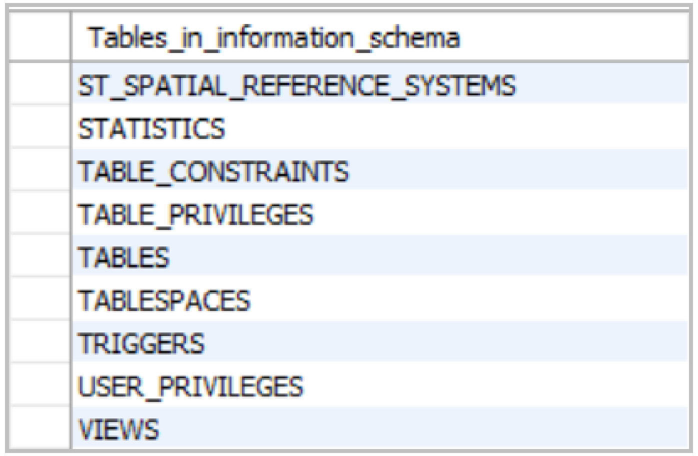


Field	Type	Null	Key	Default	Extra
ID	int(11)	NO	PRI	NULL	auto_increment
Name	char(35)	NO			
CountryCode	char(3)	NO	MUL		
District	char(20)	NO			
Population	int(11)	NO		0	

Information_schema database

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

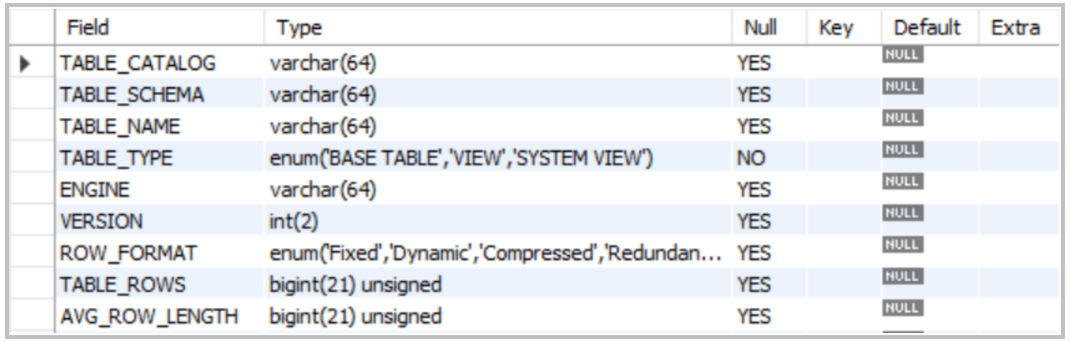
```
-- Show tables from a database.
SHOW TABLES FROM INFORMATION_SCHEMA;
```



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;
```

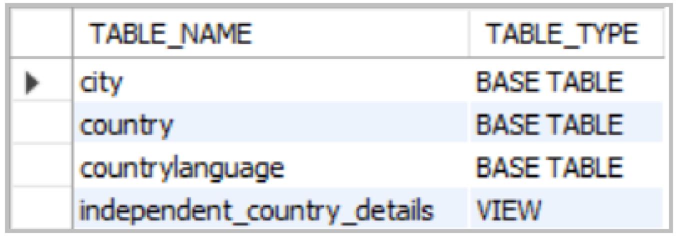


Field	Type	Null	Key	Default	Extra
TABLE_CATALOG	varchar(64)	YES		NULL	
TABLE_SCHEMA	varchar(64)	YES		NULL	
TABLE_NAME	varchar(64)	YES		NULL	
TABLE_TYPE	enum('BASE TABLE','VIEW','SYSTEM VIEW')	NO		NULL	
ENGINE	varchar(64)	YES		NULL	
VERSION	int(2)	YES		NULL	
ROW_FORMAT	enum('Fixed','Dynamic','Compressed','Redundan...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



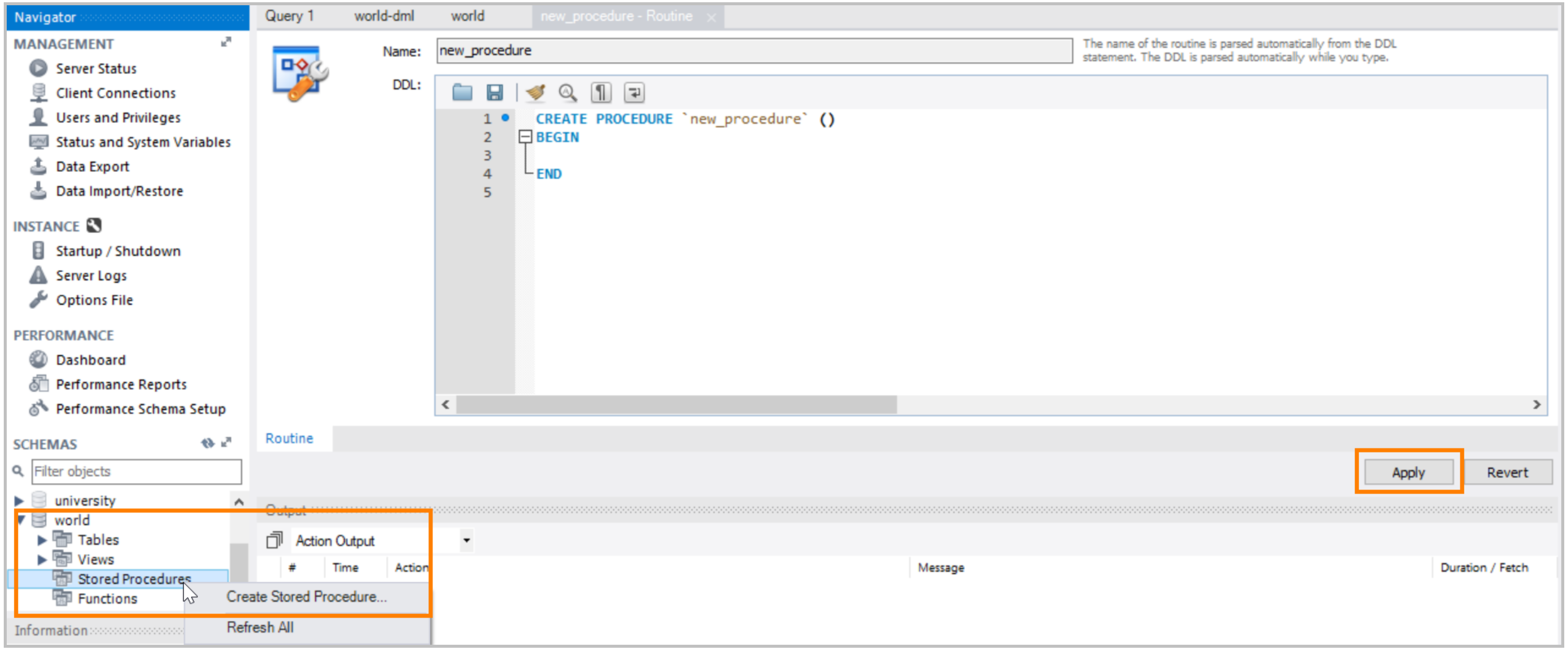
TABLE_NAME	TABLE_TYPE
city	BASE TABLE
country	BASE TABLE
countrylanguage	BASE TABLE
independent_country_details	VIEW

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**

Stored procedures

- Let's create a procedure to display all the details of the US cities



Name: new_procedure

The name of the routine is parsed automatically from the DDL statement. The DDL is parsed automatically while you type.

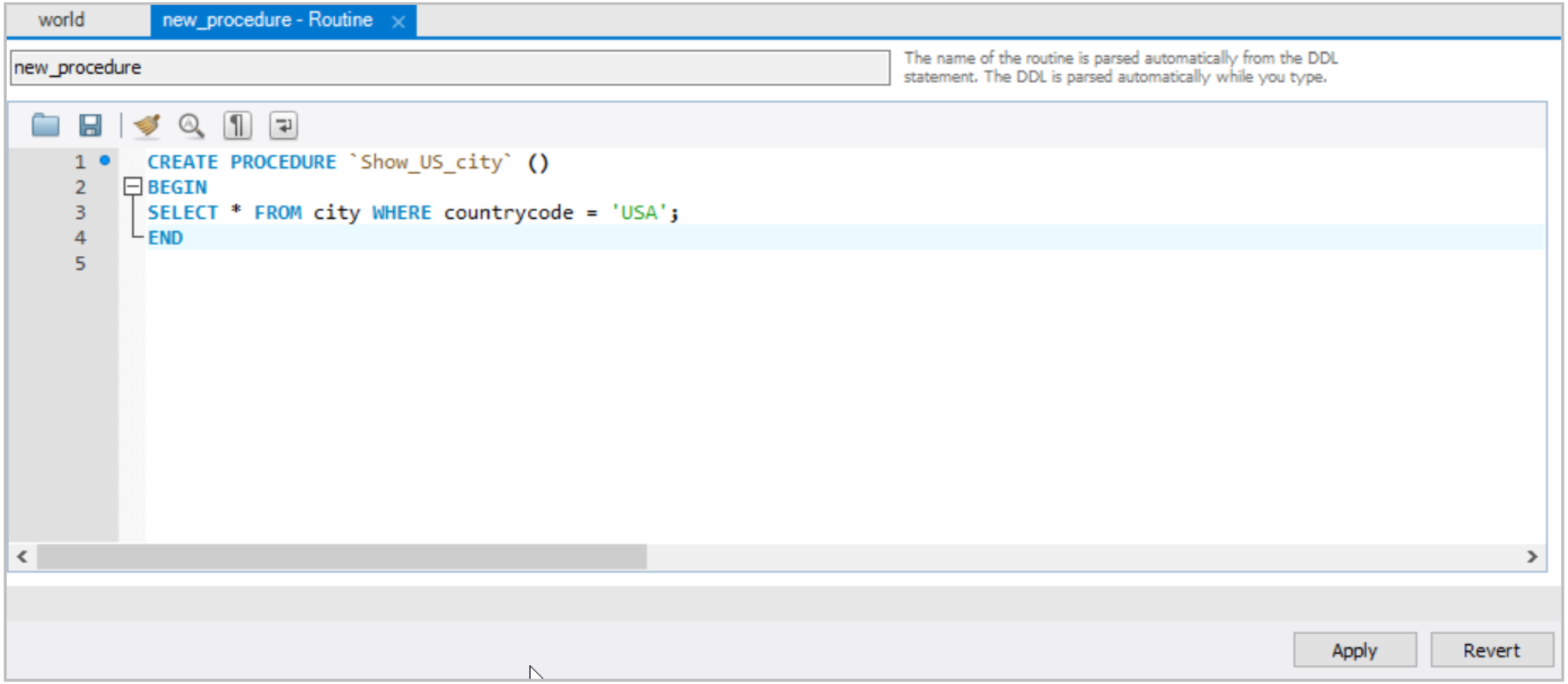
DDL:

```
CREATE PROCEDURE `new_procedure` ()
BEGIN

END
```

Stored procedures

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



new_procedure

The name of the routine is parsed automatically from the DDL statement. The DDL is parsed automatically while you type.

```
CREATE PROCEDURE `Show_US_city` ()
BEGIN
SELECT * FROM city WHERE countrycode = 'USA';
END
```

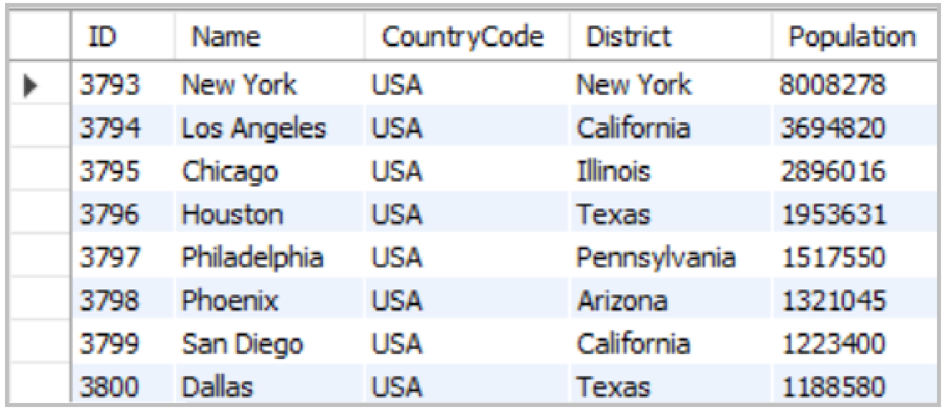
Stored procedures

- 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

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
- **What did you enjoy learning about SQL?**
- **How do you think it can impact that work you already do?**
- **Does SQL improve how you work with spreadsheets?**

Knowledge check 4

Exercise 4

Module completion checklist

Objective	Complete
Demonstrate working with temporal data	✔
Apply aggregate functions	✔
Generating groups	✔
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	✔

Workshop!

- Workshops are to be completed in the afternoon either with a dataset for a capstone project or with another dataset of your choosing
- Make sure to annotate and comment your code so that it is easy for others to understand what you are doing
- This is an exploratory exercise to get you comfortable with the content we discussed today
- Today you will:
  - Load your dataset into your workbench
  - Practice writing sub queries and working with date variables in your dataset
  - Create views in your dataset for other people to view
  - Create stored procedures to do redundant tasks

This completes our module

Congratulations!