#standard SQL and SQLite queries:

#WHAT WE LEARN IN THIS COURSE:

#https://www.lynda.com/SQL-tutorials/SQL-Essential-Training/2825374-2.html

#Basic knowledge of SQL to use it effectively in any database environment.

#Basic syntaxes of SQL to be prepared to construct own database queries.

#Basic of CRUD (create, read, update, and delete), how various data types work in order to create effective database schemas in whichever systems you may be working with. More advanced features like joins, transactions, and sub-selects for understanding and making database applications.

#####################################################################################

#always end a statement with ;

It works without ; too.

#SQL statements are not case sensitive but you can use capital letters for them for more readability, specially for keywords.

#####################################################################################

#SELECT:

#a)read whole table (here country)

SELECT \* FROM Country;

#b) read selected columns (here two columns) from table (here country)

SELECT Name, LifeExpectancy FROM Country;

#ORDER BY: to change the order of representation (here Name which is a column) of data rows those selected to read

SELECT \* FROM Country ORDER BY Name;

#ascending and descending ordering also can be specified:

SELECT \* FROM Country ORDER BY Name DESC;

SELECT \* FROM Country ORDER BY Name ASC;

# AS “”: to change the identifying name of a column (which goes inside “” if there is space in the name and goes inside””, ‘’, or without anything, if there is no space in the name). Without AS, we can specify the name in the “” after the column name but using AS, helps the readability.

SELECT Name AS “N”, LifeExpectancy AS "LE" FROM Country ORDER BY Name;

SELECT Name AS ‘N’, LifeExpectancy AS ‘LE’ FROM Country ORDER BY Name;

SELECT Name AS N, LifeExpectancy AS LE FROM Country ORDER BY Name;

#remark: choosing specific order of selected columns results in that specific representation of the table:

SELECT Name, LifeExpectancy FROM Country;

#differs from

SELECT LifeExpectancy, Name FROM Country;

#WHERE: to put criteria on selecting specific rows of a table:

SELECT Name, Continent, Region FROM Country WHERE Continent = "Europe" ORDER BY Name;

#remark: ORDER BY should be always after WHERE

#get few first selected rows (WHERE+LIMIT):

SELECT Name, Continent, Region FROM Country WHERE Continent = "Europe" ORDER BY Name LIMIT 5;

#get few second selected rows (WHERE+LIMIT):

SELECT Name, Continent, Region FROM Country WHERE Continent = "Europe" ORDER BY Name LIMIT 5 OFFSET 5;

#get the number of rows in a table:

SELECT COUNT(\*) FROM Country;

#COUNT can be used together with other statements to have specific result. EG.:

SELECT COUNT(\*) FROM Country WHERE Population>1000000;

SELECT COUNT(\*) FROM Country WHERE Population>1000000 AND Continent=”Europe”;

#get the all values of a column:

SELECT LifeExpectancy FROM Country;

#get the number of nonzero values of a column:

SELECT COUNT(LifeExpectancy) FROM Country;

#####################################################################################

#INSERT INTO: insert single row to the table (costumer) using two set of (). First one provides the columns names and the second one which comes after VALUES provides the values of all columns those listed in the first ():

INSERT INTO customer (name, address, city, state, zip) VALUES ('Fred Flintstone', '123 Cobblestone Way', 'Bedrock', 'CA', '91234');

#if less items in the column () given, we will get the NULL for the absent columns. Eg. here we get NULL #for address, and zip columns.

INSERT INTO customer (name, city, state) VALUES ('Jimi Hendrix', 'Renton', 'WA');

#remark: if the name of the columns (first () are not provided, the items in the second () go into the #columns one by one until all values are substituted.

#remark: the number of items in both () should be the same.

#Less items in the second () compare to the first one, results in NULL for the corresponding column in #the added row (here NULL for state):

INSERT INTO customer (name, city, state) VALUES ('Jimi Hendrix', 'Renton');

#Less items in the first () compare to the second one results in error:

INSERT INTO customer (name) VALUES ('Jimi Hendrix', 'Renton');

#####################################################################################

#UPDATE: the values of the specific row (using id = 5 here) by SET the values to the specific columns (SET #address = '123 Music Avenue'):

UPDATE customer SET address = '123 Music Avenue', zip = '98056' WHERE id = 5;

#remark: new value can be even NULL (can be given without “”)

#####################################################################################

#shows a specific row of the table using its id:

SELECT \* FROM Customer WHERE id=5;

#delete the specific row (using its id):

DELETE FROM customer WHERE id = 5;

#delete the specific row (where the row has specific value for a specific column):

DELETE FROM item WHERE a=2;

#####################################################################################

#CREATE TABLE:

#by specifying the name of the table and the name of the column following by the type of each column.

CREATE TABLE test (

a INTEGER,

b TEXT

);

The row data goes into the () in front of the table name and separates by ,

INSERT INTO test VALUES ( 1, 'a' );

INSERT INTO test VALUES ( 2, 'b' );

INSERT INTO test VALUES ( 3, 'c' );

#delete the whole table by specifying its name:

DROP TABLE test;

# delete the whole table by specifying its name and if it exist:

DROP TABLE IF EXISTS test;

######################################################################################NULL:

#adding a row with empty (NULL) columns:

INSERT INTO test DEFAULT VALUES;

#adding rows those get its columns values from another table (here item). In this case all rows of the #selected columns of the table will be added.

INSERT INTO test ( a, b, c ) SELECT id, name, description from item;

#remark: NULL is the absence of the value, thus if we want to get rows those have NULL, we can’t use eg #a=NULL, we should use a IS NULL. There is also IS NOT NULL statement.

SELECT \* FROM item WHERE a IS NULL;

SELECT \* FROM item WHERE a IS NOT NULL;

#empty string can be assigned to and called back from a column:

INSERT INTO item VALUES (0, NULL, ‘’);

SELECT \* FROM item WHERE c=’’

#####################################################################################

#Constrains:

#specific criteria can be set to the type of the columns of the table. In this case the provided value for that column should satisfy that criteria otherwise an error will be resulted.

#Eg1 (NOT NULL): table test below can’t have NULL for the columns a and b. We try to put NULL (by #providing nothing for column a) which results in an error.

CREATE TABLE test (

a INTEGER NOT NULL,

b TEXT NOT NULL,

c TEXT

);

INSERT INTO test ( b, c ) VALUES ( 'one', 'two' );

#Eg2 (DEFAULT): specific column get the defined default value, if no value is provided:

CREATE TABLE test ( a TEXT, b TEXT, c TEXT DEFAULT 'panda' );

#Eg3 (UNIQUE): always unique value for the specific column should be provided otherwise an error will #be raised. Notice that NULL is not counted as value so several NULL for the same column of different #rows can exist even we asked for unique values for that column.

CREATE TABLE test ( a TEXT UNIQUE, b TEXT, c TEXT);

#remark: several constrains can be set together for values of a column:

CREATE TABLE test ( a TEXT UNIQUE NOT NULL, b TEXT, c TEXT);

#####################################################################################

#ALTER: can change the schema of the table. Eg. A new column (specific constrains can be defined) can #be added:

ALTER TABLE test ADD d TEXT DEAFULT ‘panda’;

#####################################################################################

#we can identify the rows of a table by assigning an identification column to that table. Each SQL system #has its own specific system of identification which is PRIMARY KEY for SQLite system. Instead of name #id for the column of identification, any other name can be used. id column will be created #automatically (as it is requested in create table statement). It means we insert below two values and #we get 3 columns (first one is id which is automatically created).

CREATE TABLE test (

id INTEGER PRIMARY KEY,

a INTEGER,

b TEXT

);

INSERT INTO test (a, b) VALUES ( 10, 'a' );

INSERT INTO test (a, b) VALUES ( 11, 'b' );

INSERT INTO test (a, b) VALUES ( 12, 'c' );

#####################################################################################

# OR, AND:

#filtering on the selected rows: combining two constrains (OR, AND)

SELECT Name, Continent, Population FROM Country WHERE Population < 100000 OR Population IS NULL ORDER BY Population DESC;

SELECT Name, Continent, Population FROM Country WHERE Population < 100000 AND Continent = 'Oceania' ORDER BY Population DESC;

#filter rows those have a specific string in their specific column (here island in their name):

SELECT Name, Continent, Population FROM Country WHERE Name LIKE '%island%' ORDER BY Name;

#’%island%’ means name should have island. ’island%’ means name should begin with island. ’%island’ #means name should end with island. ’\_a%’ means name should have a as its second letter. ’\_\_a%’ #means name should have a as its third letter. ’a\_%’ means name should have a letter as the letter #before the last letter.

#OR statement can be restated as IN (,). Two below statements give the same results:

SELECT Name, Continent, Population FROM Country WHERE Continent IN ( 'Europe', 'Asia' ) ORDER BY Name;

SELECT Name, Continent, Population FROM Country WHERE Continent='Europe' OR Continent='Asia' ORDER BY Name;

#####################################################################################

#DISTINCT:

#get the distinct values (unique and not repeated ones) of a column:

SELECT DISTINCT Continent FROM Country;

#get the distinct combination of two columns:

SELECT DISTINCT Continent, Population FROM Country;

######################################################################################ORDER BY:

#ordering of the selected part of a table can be done based on one, two, or even more columns. The #rows get to be ordered first by the first column and within first column by second column,…

#Eg. Descending order of the countries based on their continent. Within each continent the regions are #alphabetically ordered and within each region, the names are alphabetically ordered.

SELECT Name, Continent, Region FROM Country ORDER BY Continent DESC, Region, Name;

#####################################################################################

#CASE:???

#conditional expression applied on table booltest:

CREATE TABLE booltest (a INTEGER, b INTEGER);

INSERT INTO booltest VALUES (1, 0);

#type 1: gives true when there is a non-empty and non-zero value for the selected column and #otherwise gives false.

SELECT

CASE WHEN a THEN 'true' ELSE 'false' END AS boolA,

CASE WHEN b THEN 'true' ELSE 'false' END AS boolB

FROM booltest;

#type2: gives true when the specific value is provided for the selected column and otherwise gives false.

SELECT

CASE a WHEN 1 THEN 'true' ELSE 'false' END AS boolA,

CASE b WHEN 1 THEN 'true' ELSE 'false' END AS boolB

FROM booltest;

#SQLite does not have Boolean type for conditional cases.

######################################################################################JOIN:

#joint two tables can be: a) inner join: gives the rows of both tables where the condition met. b) left #outer join: gives the rows of both tables where the condition met plus the rest of the rows of the left #table. c) right outer join: gives the rows of both tables where the condition met plus the rest of the #rows of the right table. d) full outer join: gives the rows of both tables where the condition met plus #the rest of the rows of the both tables.

#remark: if right outer join is not supported by the system of SQL, right join can be achieved by #changing the order of the tables. Sometimes full outer join is also not supported but can be achieved #by some options in the join clause.

#in making a left join, the name of the table that is wanted to be appear at left, should be called after #FROM.

#join two tables based on their id column. Eg the common column between sale table and item table is #used to join two tables. More columns of each table can be added to the joined table by providing its #name. Order of appearance of columns in the joined table is the ordering of specifying columns names.

SELECT s.id AS sale, i.name, s.price

FROM sale AS s

JOIN item AS i ON s.item\_id = i.id;

#####################################################################################

#Strings:

#length of a string:

Length (‘the string’);

SUBSTR(‘the string’, 5,6);

#gives part of the substring that begins from 5th item and the length of the substring if it is present 6. If #the length of substring is not present, the substring is from 5th item to the end of the string.

#TRIM: to remove the defined repeated item (eg. White space of points) in one or both sides of a string.

SELECT TRIM(‘ string ’);

#gives: string

SELECT TRIM(‘ string string ’);

#gives: string string

SELECT LTRIM(‘ string ’);

#gives: string (only the left side white space is removed)

SELECT RTRIM(‘ string ’);

#gives: string(only the right side white space is removed)

SELECT TRIM(‘……string….’,’.’);

#gives: string

#remark: get the code at the html on windows using CTRL+U

#upper and lower case of the strings. It works in most systems only for ascii characters.

UPPER(string);

LOWER(STRING);

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

#Numbers:

#get the type of a value

TYPEOF(1+0.1);

#Gives real

TYPEOF(‘panda’+’coala’);

#gives integer because SQLite considers + as a mathematical operator between integers and then considers panda and coala as 0 ad gives 0 as the final result. Note that some systems considers + for concatenating.

SELECT 17%5;

#gives the remainder

#important remark: simple calculus in SQL does not give always what you expect (SELECT 17/5 gives 3) #because of the data type.

ROUND(2.5555,0);

ROUND(2.5555);

#Both gives 3.0, because 0 precision is the default.

#####################################################################################

#date and time:

#it represents in SQL in the format of considering most important item to the left and least important #item to the right. Eg. 2021-01-18 21:48:52. UTC (coordinated universal time), time of the Greenwich #without summer time adjustment is normally internally in SQL is used.

# Each system has its specific date and time functions. For SQLite:

SELECT DATETIME('now');

SELECT DATE('now');

SELECT TIME('now');

SELECT DATETIME('now', '+1 day');

SELECT DATETIME('now', '+3 days');

SELECT DATETIME('now', '-1 month');

SELECT DATETIME('now', '+1 year');

SELECT DATETIME('now', '+3 hours', '+27 minutes', '-1 day', '+3 years');

#####################################################################################

#aggregate data: aggregation is the task of collecting a set of values to return a single value. It is done with the help of aggregate functions, such as SUM, COUNT, and AVG.

#COUNT:

#It is an aggregate function. In below example, first the rows are grouped by the region and then the #number of each group (region) is counted.

SELECT region, COUNT(\*) FROM Country GROUP BY region;

#aggregate function can be combined with other clauses like ORDER BY:

SELECT region, COUNT(\*) FROM Country GROUP BY region ORDER BY COUNT DESC, Region;

#COUNT be done on distinct values of a column:

SELECT COUNT(DISTINCT HeadOfState) FROM Country;

#HAVING:

# It works like WHERE but for aggregated data. Eg.

SELECT a.title AS Album, COUNT(t.track\_number) as Tracks

FROM track AS t

JOIN album AS a

ON a.id = t.album\_id

WHERE a.artist = "The Beatles"

GROUP BY a.id

HAVING Tracks >= 10

ORDER BY Tracks DESC, Album

#sum/average/minimum/maximum of the values of a column which can be combined with other #clauses:

SELECT AVG(Population) FROM Country;

SELECT Region, AVG(Population), MIN(Population), MAX(Population), SUM(Population) FROM Country GROUP BY Region;

######################################################################################TRANSACTION:

#Transaction is a set of operations those apply separately to the tables and if an error in any line of the #transaction happens, the whole part of transaction will not do anything and the status of the table rolls #back to the previous status (before applying the transaction).

BEGIN TRANSACTION;

INSERT INTO widgetSales ( inv\_id, quan, price ) VALUES ( 1, 5, 500 );

UPDATE widgetInventory SET onhand = ( onhand - 5 ) WHERE id = 1;

END TRANSACTION;

#we can roll back the transaction (no transaction will occur). In this case we can check what the status of #the data table before transaction was:

BEGIN TRANSACTION;

INSERT INTO widgetInventory ( description, onhand ) VALUES ( 'toy', 25 );

ROLLBACK;

#remark: performing the same set of operations as a transaction is much faster than doing those #operations as some single operations because in the last case the program check each operation #separately and perform it before going to the next one but the operations of a transaction are #considered as one operator.

#####################################################################################

#TRIGGER:

#It can be used to perform two operations in sequence. Eg. The table below is first created

CREATE TABLE widgetCustomer ( id INTEGER PRIMARY KEY, name TEXT, last\_order\_id INT );

CREATE TABLE widgetSale ( id INTEGER PRIMARY KEY, item\_id INT, customer\_id INT, quan INT, price INT );

INSERT INTO widgetCustomer (name) VALUES ('Bob');

INSERT INTO widgetCustomer (name) VALUES ('Sally');

INSERT INTO widgetCustomer (name) VALUES ('Fred');

SELECT \* FROM widgetCustomer;

#then the trigger (emphasized with statements between BEGIN and END) is used to insert #newWidgetSale into the WidgetSale after another operation.

CREATE TRIGGER newWidgetSale AFTER INSERT ON widgetSale

BEGIN

UPDATE widgetCustomer SET last\_order\_id = NEW.id WHERE widgetCustomer.id = NEW.customer\_id;

END;

INSERT INTO widgetSale (item\_id, customer\_id, quan, price) VALUES (1, 3, 5, 1995);

INSERT INTO widgetSale (item\_id, customer\_id, quan, price) VALUES (2, 2, 3, 1495);

INSERT INTO widgetSale (item\_id, customer\_id, quan, price) VALUES (3, 1, 1, 2995);

SELECT \* FROM widgetSale;

SELECT \* FROM widgetCustomer;

Q: What does NEW do?

#Trigger can also be used to raise error to prevent changing in a dataset. Eg. The table below is created:

CREATE TABLE widgetSale ( id integer primary key, item\_id INT, customer\_id INTEGER, quan INT, price INT, reconciled INT );

INSERT INTO widgetSale (item\_id, customer\_id, quan, price, reconciled) VALUES (1, 3, 5, 1995, 0);

INSERT INTO widgetSale (item\_id, customer\_id, quan, price, reconciled) VALUES (2, 2, 3, 1495, 1);

INSERT INTO widgetSale (item\_id, customer\_id, quan, price, reconciled) VALUES (3, 1, 1, 2995, 0);

SELECT \* FROM widgetSale;

#the trigger below prevents changes requested by the transaction:

CREATE TRIGGER updateWidgetSale BEFORE UPDATE ON widgetSale

BEGIN

SELECT RAISE(ROLLBACK, 'cannot update table "widgetSale"') FROM widgetSale

WHERE id = NEW.id AND reconciled = 1;

END;

BEGIN TRANSACTION;

UPDATE widgetSale SET quan = 9 WHERE id = 2;

END TRANSACTION;

SELECT \* FROM widgetSale;

#####################################################################################

#subselect (application1) can be used to select piece of a string which is useful to get specific #information from columns values. Eg. We can make new table using subselect of data from old table:

#old table

CREATE TABLE t ( a TEXT, b TEXT );

INSERT INTO t VALUES ( 'NY0123', 'US4567' );

INSERT INTO t VALUES ( 'AZ9437', 'GB1234' );

INSERT INTO t VALUES ( 'CA1279', 'FR5678' );

SELECT \* FROM t;

#new table

SELECT SUBSTR(a, 1, 2) AS State, SUBSTR(a, 3) AS SCode,

SUBSTR(b, 1, 2) AS Country, SUBSTR(b, 3) AS CCode FROM t;

#new table can be made using susbselect of old table and joining data from other tables (here #Country):

SELECT co.Name, ss.CCode FROM (

SELECT SUBSTR(a, 1, 2) AS State, SUBSTR(a, 3) AS SCode,

SUBSTR(b, 1, 2) AS Country, SUBSTR(b, 3) AS CCode FROM t

) AS ss

JOIN Country AS co

ON co.Code2 = ss.Country;

#subselect (application2) in combination with WHERE is a search method to get required information #from data set.

SELECT DISTINCT album\_id FROM track WHERE duration <= 90;

#the search can get more information by applying the search results on the table or another coupled #table. Eg here search is performed on track table and more information got from album table.

SELECT \* FROM album

WHERE id IN (SELECT DISTINCT album\_id FROM track WHERE duration <= 90);

#the tables can be joined to expand the information those are correlated with the search results:

SELECT a.title AS album, a.artist, t.track\_number AS seq, t.title, t.duration AS secs

FROM album AS a

JOIN track AS t

ON t.album\_id = a.id

WHERE a.id IN (SELECT DISTINCT album\_id FROM track WHERE duration <= 90)

ORDER BY a.title, t.track\_number;

######################################################################################VIEW:

#CREATE VIEW saves the view of the results those are created once, under given name (eg. trackview #here). Calling the name of the created view (here trackview), the results can be used over and over as #an existing table.

CREATE VIEW trackView AS SELECT id, album\_id, title, track\_number, duration / 60 AS m, duration % 60 AS s FROM track;

SELECT \* FROM trackView;

#delete created view:

DROP VIEW trackview

DROP VIEW IF EXISTS trackview

#####################################################################################

#trick to get time in format of minute:second

#Q:?

t.m || ':' || substr('00' || t.s, -2, 2)

#####################################################################################

#user defined functions (udf):

# These are the functions those are not defined in the SQL itself or other database or programming #language and should be defined by user in case of requirement and can be written in php or other #scripting languages like python (eg. Here the UDF is Sid.php)

#CRUD is a combination of 4 main operators in SQL (create, read, update, and delete) and an interface #can be created by writing the code in a scripting language (eg. crud.php)

#the existing sid.php and crud.php written by Bill Weinman (http://bw.org/contact/) provides the #UDF’s and CRUD interface for three main systems of SQL (PostgreSQL, SQLite 3, MySQL)

List of points: UDF, SELECT, CREATE/DROP TABLE, data types, date and time, INSERT INTO, JOIN,

TRIGGER BEFORE/AFTER BEGIN END,WHERE, HAVING, CREATE/DROP VIEW, SUBSELECT, SUBSTR, ORDER BY, UPDATE, AND/OR, ALTER, aggregate data (count/sum/average/minimum/maximum),