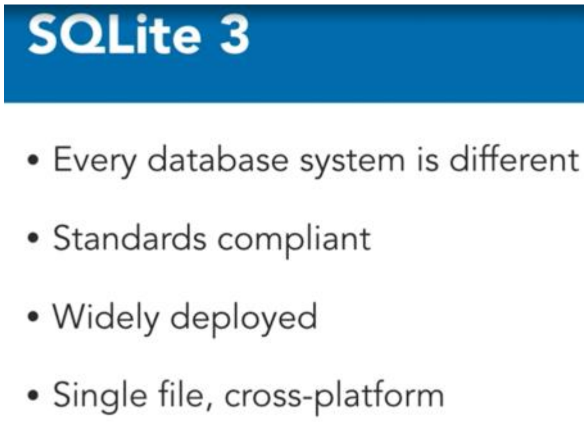
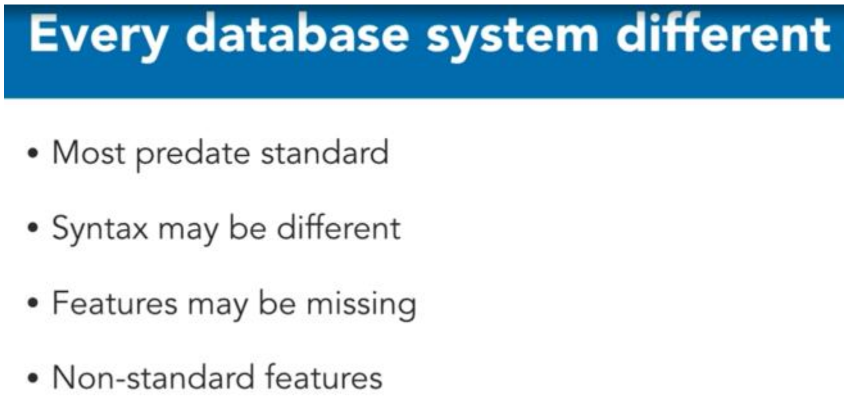
SQLite

Monday, February 1, 2021 1:56 PM



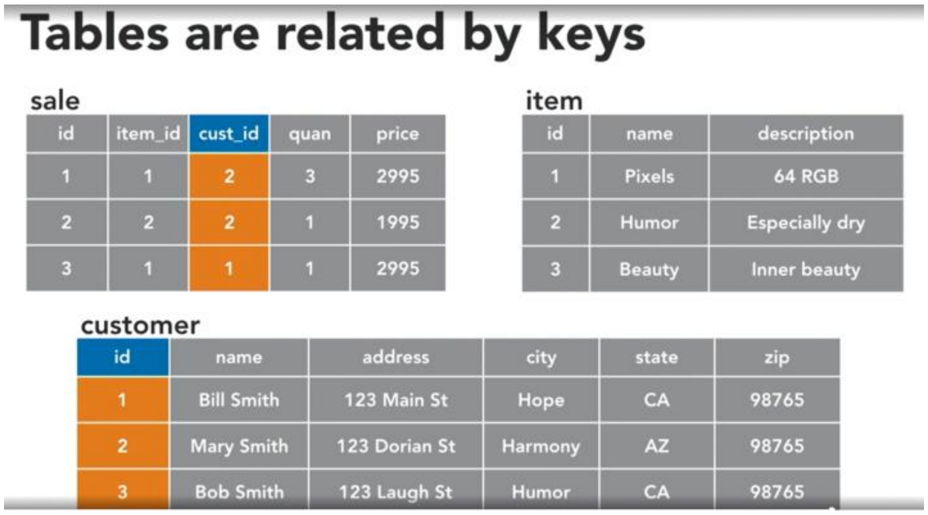
SQLite 3

- Every database system is different
- Standards compliant
- Widely deployed
- Single file, cross-platform



Every database system different

- Most predate standard
- Syntax may be different
- Features may be missing
- Non-standard features



Tables are related by keys

sale

id	item_id	cust_id	quan	price
1	1	2	3	2995
2	2	2	1	1995
3	1	1	1	2995

item

id	name	description
1	Pixels	64 RGB
2	Humor	Especially dry
3	Beauty	Inner beauty

customer

id	name	address	city	state	zip
1	Bill Smith	123 Main St	Hope	CA	98765
2	Mary Smith	123 Dorian St	Harmony	AZ	98765
3	Bob Smith	123 Laugh St	Humor	CA	98765

==>> Column - Fields

==>> Rows - Records

Primary key - is unique for tables in order to access value

Basics

Monday, February 1, 2021 3:30 PM

```
SELECT 'Hello, World' AS Result;
```

(SELECT is used to retrieve data)

=> SELECT '<string value' AS <column name/identifier> ; (By default the string will be taken as identifier and the value)

```
SELECT * FROM Country ORDER BY Name;
```

==> Here '*' means select all data '**FROM**' the <Country> table '**ORDER BY'** <Name column>; (Sort it based on the name field

```
SELECT Name, LifeExpectancy AS "Life Expectancy" FROM Country ORDER BY Name;
```

(Selecting only the <Name> and <LifeExpectancy> cols from the Country table , here <LifeExpectancy> col has aliased to " **Life Expectancy"** string using **AS**

Selecting Rows and Columns

```
SELECT Name, Continent, Region FROM Country WHERE Continent = 'Europe' ORDER BY Name LIMIT 5 OFFSET 10;
```

(SQL command has to follow the order like this , in this **WHERE** clause takes the expression as <Continent>col = '<value in the col>' thus returns all the rows with 'Europe' continent , **ORDER BY** sorts the output and **LIMIT** constrains the number of rows returned in the output and **OFFSET** sets from where the LIMIT has to be applied

```
SELECT Region, Continent, Name AS Country FROM Country;
```

Columns to be returned is specified like this in the command.

Counting

```
SELECT COUNT(*) FROM Country;
```

==> Displays the **COUNT** of all the ROWS from the Country table

Inserting Values

```
SELECT * FROM customer;

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

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

==>> selecting/highlighting a line and by running the script the single line only be executed

Updating Values

```
SELECT * FROM customer;

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

UPDATE customer SET address = '2603 S Washington St', zip = '98056' WHERE id = 5;

UPDATE customer SET address = NULL, zip = NULL WHERE id = 5;
```

==>> Updating values is done using **SET** clause and **WHERE** is used to specify the ROW in which the values have to be updated or the change will happen for all rows in the table.

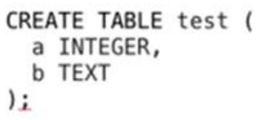
Deleting Values

```
DELETE FROM customer WHERE id = 4;
SELECT * FROM customer;
```

==>> need to specify where in order to delete a specific ROW

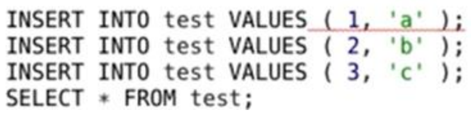
Creating Tables

Tuesday, February 2, 2021 11:29 AM



```
CREATE TABLE test (
  a INTEGER,
  b TEXT
);
```

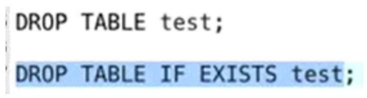
==>> The Column fields with the Data Types are mentioned in the parantheses .



```
INSERT INTO test VALUES ( 1, 'a' );
INSERT INTO test VALUES ( 2, 'b' );
INSERT INTO test VALUES ( 3, 'c' );
SELECT * FROM test;
```

==>> Inserting three rows in the table using **INSERT INTO** command

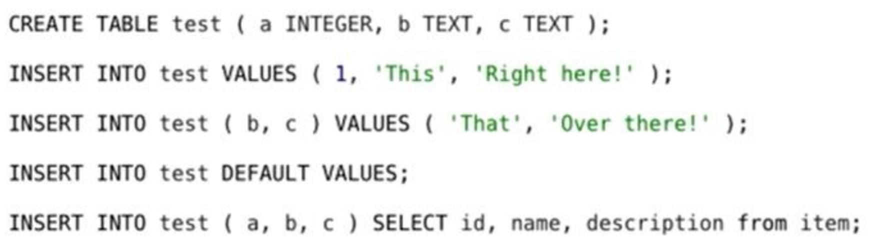
Deleting Table



```
DROP TABLE test;

DROP TABLE IF EXISTS test;
```

==>> **DROP** command is used to delete a table from the database , it can be used with **IF EXISTS** in order to precheck the table's availability.

Inserting Rows



```
CREATE TABLE test ( a INTEGER, b TEXT, c TEXT );

INSERT INTO test VALUES ( 1, 'This', 'Right here!' );

INSERT INTO test ( b, c ) VALUES ( 'That', 'Over there!' );

INSERT INTO test DEFAULT VALUES;

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

==> Creating a table

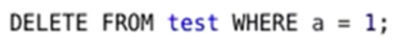
==> Inserting 3 values to all the three columns in the table

==> Inserting only 2 values to 2 specific column fields

==> **DEFAULT VALUES** inserts NULL values to the table ROW

==> Inserting values to the Table's ROW from another Table in the DB using **SELECT**

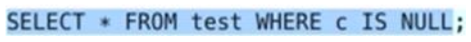
Deleting Rows



```
DELETE FROM test WHERE a = 1;
```

==>> It is destructive , that it cannot be recovered once deleted.

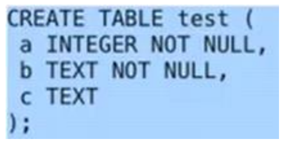
Selecting Rows with NULL values



```
SELECT * FROM test WHERE c IS NULL;
```

==>> **IS NULL** - for selecting rows with NULL values and **NOT NULL -** for selecting rows without NULL values

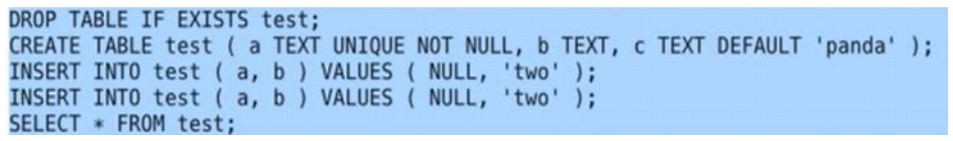
Creating Table with NOT NULL constrain



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

==>> Thus this table wont accept NULL values for the columns a and b

Constrains in Table



```
DROP TABLE IF EXISTS test;
CREATE TABLE test ( a TEXT UNIQUE NOT NULL, b TEXT, c TEXT DEFAULT 'panda' );
INSERT INTO test ( a, b ) VALUES ( NULL, 'two' );
INSERT INTO test ( a, b ) VALUES ( NULL, 'two' );
SELECT * FROM test;
```

==>> **UNIQUE** - makes the column unique so that it won't accepts repeating values

==>> **DEFAULT -** it will set a default value , so that whenever adding rows without values for that specific field will replace the field with the value specified instead of NULL

Adding a Column

```
ALTER TABLE test ADD e TEXT DEFAULT 'panda';
```

==>> Thus an extra column is created to the table

Primary Key

```
CREATE TABLE test (
  id INTEGER PRIMARY KEY,
  a INTEGER,
  b TEXT
);
```

==>> **PRIMARY KEY** - makes the field to have integer values which in itself will be populated whenever the values are added into the table.

Filtering Data

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

==>> here **%island%** is a wildcard , so that any Name values with island will be displayed % denoted anything before and after island .. Similarly **'island%'** - filters those starts with 'island' and can end with any values , **'_a%'** - Names with second letter 'a' will be displayed

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

==> providing a list of parameters in **WHERE** clause.

Omitting Duplicate values

```
SELECT DISTINCT Continent FROM Country;
```

==>> Thus **DISTINCT** displays unique values from the field instead of showing all duplicates.

Sorting Values

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

==>> **ASC -** ascending (it's the default)
==>> **DESC -** descending

Can have multiple fields in it can go as complex as is the pic.

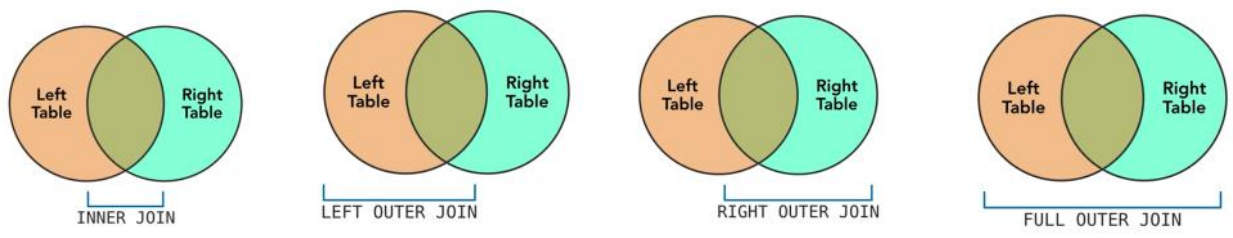
Conditional Expressions

```
SELECT
    CASE WHEN a THEN 'true' ELSE 'false' END as boolA,
    CASE WHEN b THEN 'true' ELSE 'false' END as boolB
    FROM booltest
;
```

==>> represent if (a) ? 'true' in bool A else : 'false' in bool A

Understanding JOIN

Wednesday, February 3, 2021 11:06 AM



INNER JOIN - default join in SQL

Inner JOIN

```
SELECT l.description AS left, r.description AS right
  FROM left AS l
  JOIN right AS r ON l.id = r.id
  ;
```

==>> Creating alias for the tables and executing JOIN below.

==>> **ON** if the expression clause where the condition for the join is specified.

```
SELECT l.description AS left, r.description AS right
  FROM left AS l
  LEFT JOIN right AS r ON l.id = r.id
  ;
```

==>> **LEFT JOIN -** contains the left table values and the intersection part

Relating Multiple tables

```
SELECT c.name AS Cust, c.zip, i.name AS Item, i.description, s.quantity AS Quan, s.price AS Price
  FROM sale AS s
  JOIN item AS i ON s.item_id = i.id
  JOIN customer AS c ON s.customer_id = c.id
  ORDER BY Cust, Item
```

==>> '**sale**' table on left , thereby joining customer and item based on their id's

```
SELECT c.name AS Cust, c.zip, i.name AS Item, i.description, s.quantity AS Quan, s.price AS Price
  FROM customer AS c
  LEFT JOIN sale AS s ON s.customer_id = c.id
  LEFT JOIN item AS i ON s.item_id = i.id
  ORDER BY Cust, Item
```

==>> Left join with customer table

Strings

Wednesday, February 3, 2021 12:13 PM

```
SELECT 'a literal SQL string';
```

In order to represent single quote

```
SELECT 'Here''s a single quote mark.';
```

String concatenation in std SQL <<==

```
SELECT 'This' || ' & ' || 'that';
```

```
SUBSTR( string, start, length );
LENGTH( string );
TRIM( string );
UPPER( string );
LOWER( string );
```

==>> String Functions

```
SELECT Name, LENGTH(Name) AS Len FROM City ORDER BY Len DESC, Name;
```

Takes in string parameter and returns the length

```
SELECT released,
  SUBSTR(released, 1, 4) AS Year,
  SUBSTR(released, 6, 2) AS Month,
  SUBSTR(released, 9, 2) AS Day
  FROM album ORDER BY released
```

==>> **SUBSTR**(<str value>,starting pos , number of characters to be returned from start pos)

```
SELECT TRIM('   string   ');
SELECT LTRIM('   string   ');
SELECT RTRIM('   string   ');
SELECT TRIM('...string...', '.');
```

==>> TRIM function to remove spaces

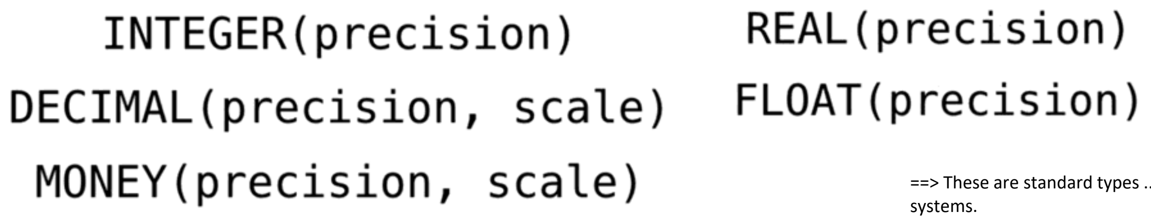
==>> specifying the character to be trimmed in the function.

```
SELECT 'StRiNg' = 'string';
SELECT LOWER('StRiNg') = LOWER('string');
SELECT UPPER('StRiNg') = UPPER('string');
SELECT UPPER(Name) FROM City ORDER BY Name;
SELECT LOWER(Name) FROM City ORDER BY Name;
```

==>> Folding cases

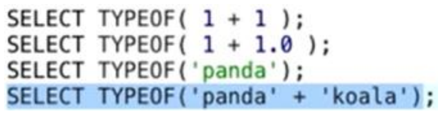
Types

Thursday, February 4, 2021 8:43 AM



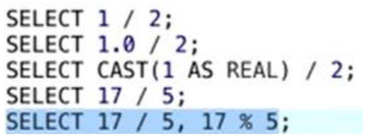
```
INTEGER(precision)        REAL(precision)
DECIMAL(precision, scale) FLOAT(precision)
MONEY(precision, scale)
```

==> These are standard types .. (money type is provided in some database systems.



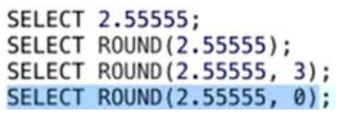
```
SELECT TYPEOF( 1 + 1 );
SELECT TYPEOF( 1 + 1.0 );
SELECT TYPEOF('panda');
SELECT TYPEOF('panda' + 'koala');
```

==>> Function to get the type of the variable passed in .



```
SELECT 1 / 2;
SELECT 1.0 / 2;
SELECT CAST(1 AS REAL) / 2;
SELECT 17 / 5;
SELECT 17 / 5, 17 % 5;
```

==>> INT division - produces int with no decimal points (therefore non real)
==>> REAL division - produces real output with dec point
==>> Another way of doing real division
==>> INT division
==>> INT divisoin and the MODULO operation to provide the remainder

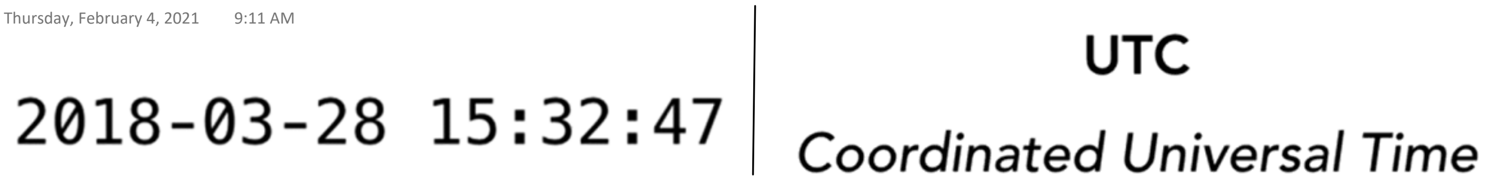


```
SELECT 2.55555;
SELECT ROUND(2.55555);
SELECT ROUND(2.55555, 3);
SELECT ROUND(2.55555, 0);
```

==>> ROUND function produces rounded values
==>> 3 is the precision level after the dec point

Standard format

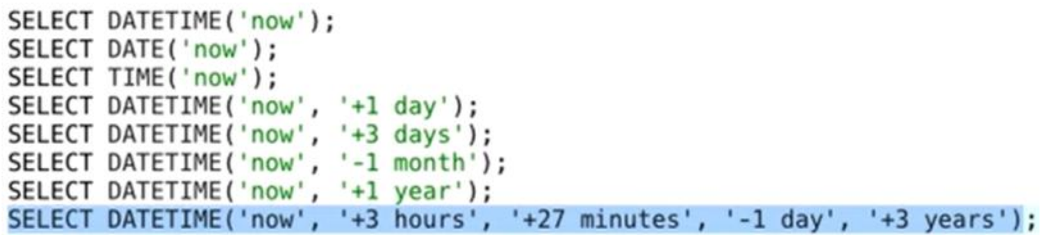
Thursday, February 4, 2021 9:11 AM



2018-03-28 15:32:47

UTC

Coordinated Universal Time

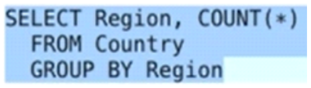


```
SELECT DATETIME('now');
SELECT DATE('now');
SELECT TIME('now');
SELECT DATETIME('now', '+1 day');
SELECT DATETIME('now', '+3 days');
SELECT DATETIME('now', '-1 month');
SELECT DATETIME('now', '+1 year');
SELECT DATETIME('now', '+3 hours', '+27 minutes', '-1 day', '+3 years');
```

==>> Working with date time in SQLite
(These are not standardized)

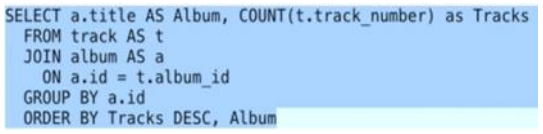
Aggregates (Group By)

Thursday, February 4, 2021 9:24 AM



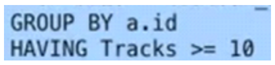
```
SELECT Region, COUNT(*)
  FROM Country
  GROUP BY Region
```

==>> Will group the table by regions and provide the count of values per group in the count field



```
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
  GROUP BY a.id
  ORDER BY Tracks DESC, Album
```

==>> Aggregating the JOINED table.

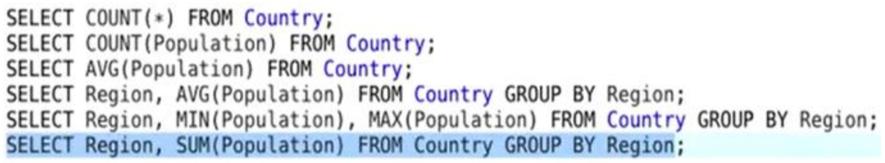


```
GROUP BY a.id
HAVING Tracks >= 10
```

==>> HAVING clause is like conditioning/filtering the aggregating data.

Note: WHERE clause should be used before the GROUP BY function.

Aggregate Functions



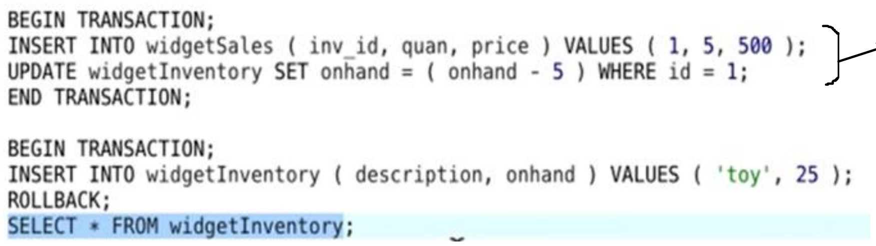
```
SELECT COUNT(*) FROM Country;
SELECT COUNT(Population) FROM Country;
SELECT AVG(Population) FROM Country;
SELECT Region, AVG(Population) FROM Country GROUP BY Region;
SELECT Region, MIN(Population), MAX(Population) FROM Country GROUP BY Region;
SELECT Region, SUM(Population) FROM Country GROUP BY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

==>> Provides the average population
==>> Grouping the data by region and showing the avg population per region
==>> Similarly MIN , MAX , SUM used for those specific actions.

Transactions

Thursday, February 4, 2021 6:44 PM

- It will increase the performance of the system (while executing commands inside the transactions.
- Transactions ensure that a number of statements are performed as a unit.



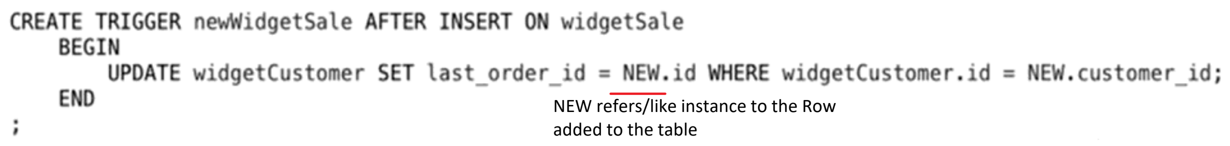
```
BEGIN TRANSACTION;
INSERT INTO widgetSales ( inv_id, quan, price ) VALUES ( 1, 5, 500 );
UPDATE widgetInventory SET onhand = ( onhand - 5 ) WHERE id = 1;
END TRANSACTION;

BEGIN TRANSACTION;
INSERT INTO widgetInventory ( description, onhand ) VALUES ( 'toy', 25 );
ROLLBACK;
SELECT * FROM widgetInventory;
```

$\rightarrow$ Set of instructions/command to be done inside a transaction.

==>> **ROLLBACK** is used to undo things in a transaction.

Triggers

- These are set of instructions which will be performed automatically on a specified case.

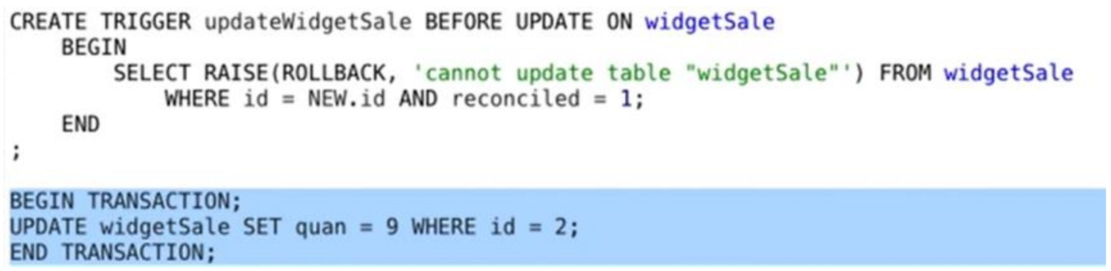


```
CREATE TRIGGER newWidgetSale AFTER INSERT ON widgetSale
    BEGIN
        UPDATE widgetCustomer SET last_order_id = NEW.id WHERE widgetCustomer.id = NEW.customer_id;
    END
;
```

NEW refers/like instance to the Row added to the table

==>> In this the commands inside trigger will be performed when rows are inserted into the 'widgetSale' table.

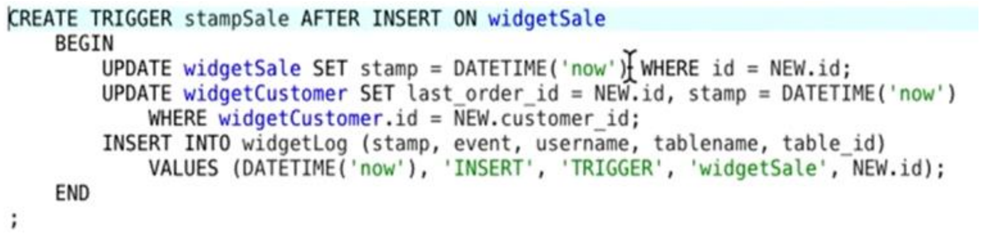
- Another way of using triggers is to raise exceptions on specified case automatically.



```
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;
```

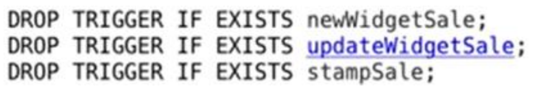
==>> In this case "**error**" will be thrown before Updating the row values in widgetSale where it Satifies the WHERE clause.

- Creating Log using Triggers



```
CREATE TRIGGER stampSale AFTER INSERT ON widgetSale
    BEGIN
        UPDATE widgetSale SET stamp = DATETIME('now') WHERE id = NEW.id;
        UPDATE widgetCustomer SET last_order_id = NEW.id, stamp = DATETIME('now')
            WHERE widgetCustomer.id = NEW.customer_id;
        INSERT INTO widgetLog (stamp, event, username, tablename, table_id)
            VALUES (DATETIME('now'), 'INSERT', 'TRIGGER', 'widgetSale', NEW.id);
    END
;
```

==>> So it creates log file with DATETIME stamps using automated trigger function



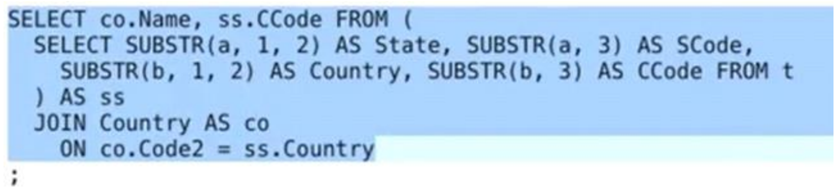
```
DROP TRIGGER IF EXISTS newWidgetSale;
DROP TRIGGER IF EXISTS updateWidgetSale;
DROP TRIGGER IF EXISTS stampSale;
```

==>> Used to drop/deactivate triggers

Sub Selects

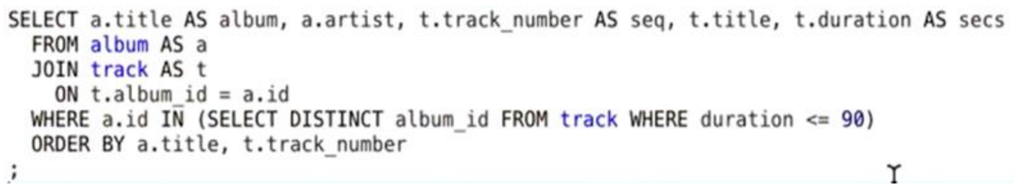
(selecting from an selected value)

Thursday, February 4, 2021 9:12 PM



```
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
;
```

==>> In this case we r selecting values and displaying from the selected values in inside.

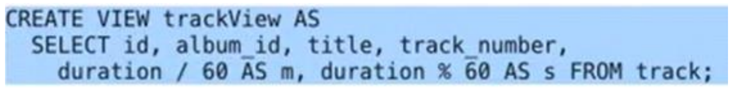


```
SELECT a.title AS album, a.artist, t.track_number AS seq, t.title, t.duration AS secs
  FROM album AS a
  JOIN track AS t
    ON t.album_id = a.id
  WHERE a.id IN (SELECT DISTINCT album_id FROM track WHERE duration <= 90)
  ORDER BY a.title, t.track_number
;
```

==>> Using subselect in WHERE clause and creating JOIN table.

Creating Views

(view is a saved form of query (select query) which can be used as a table in the commands)



```
CREATE VIEW trackView AS
  SELECT id, album_id, title, track_number,
    duration / 60 AS m, duration % 60 AS s FROM track;
```

==>> Creating a view