Understanding SQL

ABOUT SQL

- **SELECT * FROM Countries WHERE Continent = ‘Europe’;**
- Create, Read, Update, Delete - CRUD
- SELECT
- **SELECT * FROM Customer;**
- CREATE
- **INSERT INTO Customer (name, city, state)**

  VALUES (‘Jimi Hendrix’, ‘Renton’, ‘WA’);
- UPDATE
- **UPDATE Customer**

  SET

  Address = ‘123 Music Avenue’ ,

  Zip = ‘98056’

  WHERE id = 5;
- DELETE
- **DELETE FROM Customer WHERE id = 4;**

DATABASE ORGANIZATION

- **USE world;**

  SELECT Name, LifeExpectancy AS "Life Expectancy" FROM COUNTRY

  ORDER BY Name;

- Selects two Country Columns from World Database, and renames one. Also orders by name in ascending order.

SELECTING ROWS

- ORDER BY has to be past any WHERE clause. LIMIT & OFFSET need to be last.
- **SELECT Name, Continent, Region FROM Country WHERE Continent = 'Europe' ORDER BY Name LIMIT 5 OFFSET 10;**
- The offset function allows you to peruse through your results a little at a time with each submission.

SELECTING COLUMNS

- **SELECT Name as Country, Continent, Region FROM Country;**
- You can list the columns in any order that you want, and that is how they will appear on the chart.

COUNTING ROWS

- **SELECT COUNT(*) FROM Country;**
- This selects the count of how much data is in that column. How many rows?
- **SELECT COUNT(*) FROM Country WHERE Population > 100000000 AND Continent = 'Europe';**
- This code narrows down the rows extensively. In order to be in the count the population must reach the parameter, and be on the European Continent.
- **SELECT COUNT(LifeExpectancy) FROM Country;** Only counts registered data.

INSERTING DATA

- **SELECT * FROM customer;**

  INSERT INTO customer (name, address, city, state, zip)

  VALUES ('Fred Flinstone', '123 Cobblestone Way', 'Bedrock', 'CA' , '91234');

  INSERT INTO customer (name, city, state)

  VALUES('Jimi Hendrix', 'Renton', 'WA')

- The code inserts two values into the customer table. One which fully fills all categories - Fred Flinstone, and the other which is inserted but has null values in some categories - Jimi Hendrix.

UPDATING DATA

- **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;

- The first bit of code corresponds to WHERE id = 5. Address & zip code is changed accordingly.
- The following lines just overwrite that same ID.

DELETING DATA

- **SELECT * FROM customer WHERE id = 4;**

  DELETE FROM customer WHERE id = 4;

  SELECT * FROM customer;

  DELETE FROM customer WHERE id = 5;

  SELECT * FROM customer;

- The code selects id 4 and then deletes that same id from the table. Then the same thing is done from id 5.

CREATE A TABLE

- **USE test;**

  CREATE TABLE test (

  a INTEGER,

  b TEXT

  );

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

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

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

  SELECT * FROM test;

- The first bit of code creates a table within the test database. The table test has two columns a, and b. The insert into statements put in the values into a, and b.

DELETING A TABLE

- **CREATE TABLE test (a TEXT, b TEXT);**

  INSERT INTO test VALUES (‘one’, ‘two’);

  SELECT * FROM test;

  DROP TABLE test;

  DROP TABLE IF EXISTS test;
- When initially trying to create this table it already exists. Therefore you must drop the table before creating again. - DROP TABLE test; must be run first.
- The code creates a table with two columns a & b. These columns allow test input. The insert function puts in one & two as text.

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;

  SELECT * FROM test;
- The third line of code specifically chooses columns b & c to input text in each. In column a it returns a null value or lack of value.
- The insert into test default values provides an outcome of all null values.

- The last insert into specificies the columns a, b, c. Then selects other columns from the specific table ITEM to be filtered in.

DELETING ROWS

- **SELECT * FROM test;**

  DELETE FROM test WHERE a = 3;
- It is best to verify the rows that you want to delete are accurate by verifying first:
- **SELECT * FROM test WHERE a = 3;**

THE NULL VALUE

- Important to distinguish between a zero value, an empty string, or a non value result
- **USE test;**

  SELECT * FROM test WHERE a = NULL;

  NULL is the absence of a value. Therefore you can not test a = NULL. It must be...
- **SELECT * FROM test WHERE a IS NULL;**
- **SELECT * FROM test WHERE a IS NOT NULL;**
- The code above represents two opposing statements.
- **INSERT INTO test (a, b, c) values (0, NULL, ‘ ‘);**
- **SELECT * FROM test WHERE b is NULL;**
- The code above returns the entire rows where b is null.
- **SELECT * FROM test WHERE c = ‘ ‘;**
- This returns the entire new row which has ‘ ‘ which represents an empty string.

- **DROP TABLE IF EXISTS test;**

  CREATE TABLE test (

  a INTEGER NOT NULL,

  b TEXT NOT NULL,

  c TEXT

  );

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

  SELECT * FROM test;
- This code results in an error because of the NOT NULL parameter with a. The insert does not specify anything for a which results in a NULL product. However, because it does not comply with the parameter it causes an error.

CONSTRAINING COLUMNS

- **DROP TABLE IF EXISTS test;**

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

  INSERT INTO test (a, b) VALUES (‘one’ , ‘two’);

  SELECT * FROM test;
- This code will fail because within the CREATE TABLE test command, c is specified as NOT NULL. The insert has nothing for c resulting in NULL. Throws an error.
- **CREATE TABLE test (a TEXT, b TEXT, c TEXT DEFAULT ‘panda’);**
- If nothing is INSERT INTO c the table will default the string panda into the row.
- **CREATE TABLE test (a TEXT UNIQUE, b TEXT, c TEXT NOT NULL);**

  INSERT INTO test (a, b) VALUES (‘one’ , ‘two’);

INSERT INTO test (a, b) VALUES (‘one’ , ‘two’);

- This code throws an error because we are attempting to put ‘one’ into the a column multiple times when we specified it as unique in the CREATE TABLE. NULL values are exempt from the UNIQUE parameter. In order to combat that you would have to pair UNIQUE W/ NOT NULL

CHANGING A SCHEMA

- **DROP TABLE IF EXISTS test**

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

  INSERT INTO test VALUES (‘one’ , ‘two’ , ‘three’);

  INSERT INTO test VALUES (‘two’, ‘three’, ‘four’);

  INSERT INTO test VALUES (‘three’, ‘four’, ‘five’);

  SELECT * FROM TEST;

  ALTER TABLE test ADD d TEXT;

- The alter table adds the column d to the table test. When we SELECT * the column d displays null values.
- **ALTER TABLE test ADD e TEXT DEFAULT ‘panda’;**
- The text default creates a new column e and inserts panda into every column.

ID COLUMNS

- **DROP TABLE IF EXISTS test;**

  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');

  SELECT * FROM test;

- Within this code the ID column populates itself in sequential values. Meanwhile there is input for a & b in the table. Because the ID is the primary key it is binded to it.

FILTERING DATA

- **USE world;**

  SELECT Name, Continent, Population FROM Country;

  WHERE Population < 10000 OR Population IS NULL ORDER BY Population DESC;

- This code filters for certain columns for countries that have a population less than 10000. Also some countries have null values for the population, and it is included in this code.
- **WHERE Population < 10000 AND Continent = 'Oceania' ORDER BY Population DESC;**

- This further filters to specifically only include countries that are from the Oceania continent.
- **SELECT Name, Continent, Population, FROM Country**

  WHERE Name LIKE '%island%' ORDER BY Name;
- The code above selects all countries that have the word island anywhere within their name. The percent at the beginning will query for all countries that have island at the end & vice versa.
- **WHERE Name LIKE '_a%' ORDER BY Name;**
- This search will query for any Name with the second letter as a. The underscore is a wildcard meaning it can be any letter.
- **WHERE Continent IN ('Europe' , 'Asia') ORDER BY Name;**
- Simply matches any country that is in Europe or Asia.

REMOVING DUPLICATES

- **SELECT DISTINCT Continent FROM Country;**
- This displays all distinct continents from
- **DROP TABLE IF EXISTS test;**

  CREATE TABLE test (a int, b int);

  INSERT INTO test VALUES (1, 1);

  INSERT INTO test VALUES (2, 1);

  INSERT INTO test VALUES (3, 1);

  INSERT INTO test VALUES (4, 1);

  INSERT INTO test VALUES (5, 1);

INSERT INTO test VALUES (1, 2);

INSERT INTO test VALUES (1, 2);

INSERT INTO test VALUES (1, 2);

INSERT INTO test VALUES (1, 2);

INSERT INTO test VALUES (1, 2);

SELECT * FROM test;

SELECT DISTINCT a FROM test;

- The code displays all distinct values in the column a.
- **SELECT DISTINCT a, b FROM test;**
- This code displays all unique values where a & b are different.

ORDERING OUTPUT

- **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;
- Desc goes from Z to A. Asc is the default.
- The order by Continent automates ASC order with all countries from Africa coming first.
- The last code commands the continent to be DESC, but the other categories are asc by default.

CONDITIONAL EXPRESSIONS

- **DROP TABLE IF EXISTS booltest;**

  CREATE TABLE booltest (a INTEGER, b INTEGER);

  INSERT INTO booltest VALUES (1, 0);

  SELECT * FROM booltest;

  SELECT

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

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

  FROM booltest

  ;

- In SQL 1 is true, and 0 is false. Therefore when running the case the column a has 1 so it evaluates to true, and is renamed as boolA.

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

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

  FROM booltest

- This returns the exact same result as above, but we are specifically searching for a certain number comparison

UNDERSTANDING JOIN

- You can visualize a joined query as a venn diagram. The primary key in one table, and foreign key in another are what is in the middle of a venn diagram.

- The most common join is the inner join. This is the default and what happens when you use join by itself. Left outer join is everything within and in the middle. Vice versa with the right

ACCESSING RELATED TABLES

- **CREATE TABLE left (id INTEGER, description TEXT);**

  CREATE TABLE right (id INTEGER, description TEXT);

  INSERT INTO left VALUES (1, 'left 01');

  INSERT INTO left VALUES (2, 'left 02');

  INSERT INTO left VALUES (3, 'left 03');

  INSERT INTO left VALUES (4, 'left 04');

  INSERT INTO left VALUES (5, 'left 05');

  INSERT INTO left VALUES (6, 'left 06');

  INSERT INTO left VALUES (7, 'left 07');

  INSERT INTO left VALUES (8, 'left 08');

  INSERT INTO left VALUES (9, 'left 09');

  INSERT INTO right VALUES (6, 'right 06');

  INSERT INTO right VALUES (7, 'right 07');

  INSERT INTO right VALUES (8, 'right 08');

  INSERT INTO right VALUES (9, 'right 09');

  INSERT INTO right VALUES (10, 'right 10');

INSERT INTO right VALUES (11, 'right 11');

INSERT INTO right VALUES (11, 'right 12');

INSERT INTO right VALUES (11, 'right 13');

INSERT INTO right VALUES (11, 'right 14');

SELECT * FROM left;

SELECT * FROM right;

- **SELECT l.description AS left, r.description AS right**

  FROM left as l

  JOIN right as r ON l.id = r.id

  ;
- l is an alias for the left table, and r is an alias from the right table. Both aliases are created later in the code. Within the SELECT statement it is also renaming the columns how they will be output and calling on the description column
- The two tables l & r are being joined on ID. Essentially it what would be in the middle of the Venn Diagram aka an inner join
- **LEFT JOIN right AS r ON l.id = r.id;**
- The code above displays what is in the middle of the venn diagram also what is solely responsible to the left. Where there is data on the left & nothing on the right null is displayed.

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

  FROM sale AS s

  JOIN item AS i ON s.item_id = i.id

- In the code we are referring to sale and item as I & S. We are joining on s.item_id = i.id in other words sale item id with item id and also bringing in certain called columns as well.

RELATING MULTIPLE TABLES

- **INSERT INTO customer (name) VALUES ('Jane Smith');**

  SELECT * FROM customer

  SELECT c.name AS Cust, c.zip, i.name AS Item, i.description, s.quantity AS Quan, s.price AS Price

  FROM sales 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

  ;

- From the customer table we are taking name as cust, and zip. From the item column we are taking name as Item, and description. From the sale table we are taking quantity as Quan, and price as Price.
- Because customer & item tables both have name column 'name' they must be renamed with an alias.

- The table is started from the Sales table. Therefore when you input Jane Smith into the customer table and then reload here row is nowhere to be found. That is because the table stems form Sales. We are joining sales item id into the parent database of sales. At the end we will only get the output of what is in common with both from their ids.
- **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 sales AS s on s.customer_id = c.id

  LEFT JOIN item AS i on s.item_id = i.id

  ORDER BY Cust, Item

  ;
- The top line of code selects all the columns that we want to display from multiple tables. The left join takes all ids in the sales table & also takes what is in common with the customer table to display an output. The same can be said for item…. We take all ids from the item table, and then whatever we have in common with the sales table. This is due to the left join.

ABOUT SQL STRINGS

- SELECT 'Here' 's a single quote mark.';
- **SELECT CONCAT ('This', '&', 'that');**
- **SUBSTR(string, start, length)**

  LENGTH(string);

  TRIM(string);

UPPER(string);

LOWER(string);

FINDING THE LENGTH OF A STRING

- **USE world;**

  SELECT LENGTH('string');
- The return is 6.
- **SELECT name, LENGTH(Name) AS Len FROM City ORDER BY Len DESC,**

  Name;
- The return is cities with the longest names in descending order.

SELECTING PART OF A STRING

- **SELECT SUBSTR('this string', 6);**
- From the point of 6, whatever is after that is returned. It is a substring.
- **SELECT SUBSTR('this string', 6, 3);**
- The 3 returns the number of characters that are returned.
- **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;
- This parses out the year, month, & day forming separate categories. Extremely useful tool for conjoined data.

REMOVING SPACES

- **SELECT TRIM(‘ string ‘);**

  SELECT LTRIM(‘ string ‘);

  SELECT RTRIM(‘ string ‘);

  SELECT TRIM(‘.....string….’, ‘.’);
- In the last code, what we want to trim is specifically specified through ‘.’
- TRIMMING IS CASE SENSITIVE!

FOLDING CASE

- **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;
- The first three lines are confirming if it is true or false. The last lines of code select the name in all upper case from City and put it in alphabetical order.

NUMERIC TYPES

- **INTEGER(precision)**

  DECIMAL(precision, scale)

  MONEY(precision, scale)

  REAL(precision)

FLOAT(precision)

- Precision refers to how many digits are represented. Scale is the magnitude of the number that may be represented.

WHAT TYPE IS THAT VALUE?

- **SELECT TYPEOF(1+1);**

  SELECT TYPEOF(1+1.0);

  SELECT TYPEOF('panda');

  SELECT TYPEOF('panda' + 'koala');
- Integer, real(integer was converted to a real number for the addition operation) , text, integer (two strings come back as integers because the + is not a concat function. The system determines them as numbers.)
- Real number and float are synonymous

<u>INTEGER DIVISION</u>

- **SELECT 1 / 2;**

  0 (two integers are divided, therefore the system defaults to an integer value)
- **SELECT 1.0 / 2;**

  0.5 (result is a real number because one of the operands are also a real number
- **SELECT CAST(1 AS REAL) / 2;**

  0.5 (same thing as above just different wording)
- **SELECT 17 / 5;**

  3 (the output returns an integer, but has no present remainder)

SELECT 17 / 5, 17 % 5;

- 3, 2 (returns 3 with a remainder of 2)

ROUNDING NUMBERS

- **SELECT 2.55555;**

  The select returns the exact same number

- **SELECT ROUND(2.55555);**

  The return is 3

- **SELECT ROUND(2.555555, 3);**

  Result is 2.556, the 3 tells that you want to round 3 places

- **SELECT ROUND(2.555555, 0);**

  Result is 3, round to 0 decimal places which is also the default of the round function.

DATES AND TIMES

- 2018-03-28 15:32:47 : STANDARD SQL FORMAT
- UTC coordinated universal time, or Greenwich england time.

DATE- AND TIME- RELATED FUNCTIONS

- **SELECT DATETIME('now');**

  2019-10-29 21:21:32 - current UTC time

- **SELECT DATE('now');**

  2019-10-29

- **SELECT TIME('now');**

21:22:25

- **SELECT DATETIME('now', '+1 day')**

  2019-10-30 21:22:43

- **SELECT DATETIME('now', '+3 days')**

  2019-11-01 21:21:32

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

  2019-09-29 21:21:32

- **SELECT DATETIME('now', '+1 year')**

  2020-10-29 21:21:32

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

  2022-10-29 00:50:06

AGGREGATES

- **USE world;**

  SELECT Region, COUNT(*);

  FROM Country

  GROUP BY Region

- Sorts by region, and groups by region. The results are grouped which allows the aggregate function to apply to all.

- **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

;

- The code selects title, and renames it as Album. The count function counts the track_numbers and renames as tracks. Tracks & albums are joined on their ids from tracks. From that point the tracks are grouped by their corresponding album id. Having tracks more than or equal to 10 filters the results further.
- The having clause is like a where clause for aggregate data. It is important to have a separate keyword this because the where clause may still need to operate for the non aggregate part of the query. Where clause is for non aggregate data.

USING 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;

AGGREGATING DISTINCT VALUES

- **SELECT COUNT(HeadOfState) FROM Country;**

  SELECT HeadOfState FROM Country ORDER BY HeadOfState;

  SELECT Count (DISTINCT HeadOfState) FROM Country;

- The first line of code returns the count of how many head of states, second line orders the names of Head Of states in alphabetical order. The third line counts the total of distinct non repeating names.