COMPLETE SQL with ME

- Nishant Kolapkar

SQL (Structural Query Language) 08.12.2022 3:35pm

- SQL is the programming language used to communicate with our Database.

Database Overview:

- Definitions of Database:
  - Database are systems that allow user to store and organize data.
  - They are useful when dealing with large amount of data.

- Spreadsheets (Such as Excel):
  - One-time analysis
  - Quickly need to chart something out
  - Reasonable data set size
  - Ability for untrained people to work with data

- Database:
  - Data Integrity (just like security)
  - Can handle massive amount of data
  - Quickly combine different datasets
  - Automate steps for re-use
  - Can support data for websites and application

- Excel – Database
  1. Tab Sheets = Tables
  2. Columns = Columns
  3. Rows = Rows

  - SQL is case-insensitive select and SELECT is same for it

SQL Statement Fundamental (SQL syntax)

- **SELECT Statement**
  - **SELECT** is the most common statement used, and it allows us to retrieve information from a table.
  -
  - Example **syntax** for SELECT statement:

    SELECT column_name **FROM** table_name;

    =SELECT c1, c2 FROM table_1; # for c1 and c2 from table_1
    =SELECT * FROM table_1; # for all columns from table_1
    (not general to use *, It will automatically query everything, which increase traffic between the database server and the application, which can slow down the retrieval of results.)

  - To see how many tables are present in database go to >>Schemas>>Public>>Tables

- **SELECT DISTINCT**
  - Sometimes a table contains a column that has duplicate values, and you may find yourself in a situation where you only want to list the unique/distinct values.
  - The **DISTINCT** keyword can be used to return only distinct values in a column.

- **Syntax:**
- **= SELECT DISTINCT(**column) **FROM** table; # No space between DISTNCT & ( )
  Or
  = SELECT DISTINCT column **FROM** table;

- **COUNT**

  - The **COUNT** function returns the number of input rows that match a specific condition of a query.
  - **Syntax:**
    = SELECT COUNT(name) FROM table; #COUNT needs parenthesis ( )

    ****COUNT with DISTINCT to count unique****

    = SELECT COUNT(DISTINCT name) FROM table;

- **SELECT WHERE**
  - The **WHERE** statement allows us to specify conditions on columns for the rows to be returned.

  - **Syntax:**

    = SELECT col_1, col_2 **FROM** table
    WHERE condition;

    Example,

    ```
    SELECT title FROM film
    WHERE rental_rate > 4 AND replacement_cost >= 5;

    SELECT email FROM customer
    WHERE first_name = 'Nancy' AND last_name = 'Thomas';

    SELECT description FROM film
    WHERE title = 'Outlaw Hanky';

    SELECT phone FROM address
    WHERE address = '259 Ipoh Drive';
    ```

  - The conditions are used to filter the rows returned from the SELECT statement.
  - **Comparison Operators:**
    = equal, >Greater than, < Less Than, >=, <=, <> or != Not equal.
  - **Logical Operators:**
    AND, OR, NOT

- **ORDER BY**
  - Its use for sort rows based on a column value, in either ascending or descending order.
  - **Syntax:**

```
= SELECT col_1, col_2 FROM table
   ORDE BY col_1 ASC/DESC        # ORDER BY is always at end of query.
                                   ASC is default.
SELECT * FROM customer
ORDER BY first_name DESC;

SELECT first_name, last_name, email FROM customer
ORDER BY store_id DESC, first_name ASC;
```

- **LIMIT**
  - The **LIMIT** command allows to limit the number of rows returned for query.
  - Useful for not wanting to return every single row in a table, but only view the top few rows to get an idea of the table layout.
  - Its also use with **ORDER BY**.
  - Its last command.
  - **Syntax:**

Example,

```
SELECT * FROM payment
WHERE amount != 0.00
ORDER BY payment_date ASC
LIMIT 10;                              # Limit the rows in return

SELECT customer_id FROM payment
ORDER BY payment_date DESC
LIMIT 10;

SELECT title FROM film
ORDER BY length
LIMIT 5;

Extra, SELECT count(*) FROM film
        WHERE length <= 50;
```

- **BETWEEN**
  - The **BETWEEN** operator can be used to match a value against a range of values.
    - Value **BETWEEN** low **AND** high
  - The BETWEEN operator is the same as:
    - Value >= low **AND** value <= high
    - Value **BETWEEN** low **AND** high # Also use **NOT BETWEEN**

Example,

```
SELECT * FROM payment
WHERE amount BETWEEN 8 AND 9;

SELECT * FROM payment
WHERE payment_date BETWEEN '2007-02-01' AND '2007-02-15';
```

- **IN**
  - Its use to create a condition that checks to see if a value in included in a list of multiple options.
  - Example
    - **SELECT** color **FROM** table
      WHERE color **IN** (‘red’, ‘blue’);
    - **SELECT** color **FROM** table
      WHERE color **NOT IN** (‘red’, ‘blue’);
    - **SELECT** * **FROM** payment
      WHERE amount **IN** (0.99, 1.98, 1.99)
      ORDER BY amount **ASC**;

End 12:35am

- **LIKE and ILIKE** 09.12.2022 8.10pm
  - Its used when general pattern is in string.
  - **LIKE** operator allows to perform pattern matching against string data with **wildcard** characters:
    - **Percentage % (Matches any Sequence of characters)**

      Example,
      a. All names that begin with an ‘A’
         = **WHERE** name **LIKE** ‘A%’
      b. All names that end with an ‘a’
         = **WHERE** name **LIKE** ‘%a’

  - **LIKE** is case-sensitive
  - **ILIKE is case-insensitive**

    - **Underscore _ (Matches any single character)**
      a. It allows to replace just single character
         Example,
         Get all Mission Impossible films
         = **WHERE** title **LIKE** ‘Mission Impossible _’
      b. Can be use for multiple _
         Example,
         Format ‘Version#A4’, ‘Version#B7’,…
         WHERE value **LIKE** ‘Version#_ _’;

-% and _ can be combine

Example,

WHERE name **LIKE** ‘_her%’

- **GROUP BY and Aggregate functions**
  - **GROUP BY** will allow us to aggregate data and apply functions to better understand how data is distributed per category.

- **Aggregate Functions:**
  - **Most Common Aggregate Functions:**
    - **AVG( )**
    - **COUNT( )**
    - **MAX( )**
    - **MIN( )**
    - **SUM( )**
  - Aggregate function calls happen only in the **SELECT** clause or the **HAVING** clause.
  - **Special Note:**
    - **AVG( )** returns floating point value many decimal places (e.g. 2.345656..) We can use **ROUND( )**
    - **COUNT( )** simply returns the number of rows, just use **COUNT(*)**
  - **Example,**
    - **SELECT MIN**(replacement_cost**) FROM** film;
    - **SELECT MAX**(replacement_cost**) FROM** film;
    - **SELECT MAX**(replacement_cost), **MIN**(replacement_cost) **FROM** film;
    - **SELECT ROUND(AVG(**replacement_cost**),** 2**) FROM** film;
    - **SELECT SUM(**replacement_cost**) FROM** film**;**

- **GROUP BY:**
  - GROUP BY allow us to aggregate columns per some category.
  - **Syntax:**
    - **SELECT** category_col*, **AGG**(data_col) **FROM** table # AGG is for aggregate functions
      GROUP BY category_col*;

    - **SELECT** category_col*, **AGG**(data_col) **FROM** table
      WHERE category_col != 'A' # Filtering purpose or anything
      GROUP BY category_col*;
      - The **GROUP BY** clause must appear right after a **FROM** or **WHERE** Statement.
      - * Must be included in **SELECT** and **GROUP BY**

    - **SELECT** X, Y, **SUM**(sales) **FROM** table
      GROUP BY X, Y;

    - **SELECT** X, Y, **SUM**(sales) **FROM** table
      GROUP BY X, Y
      ORDER BY SUM(sales)
      LIMIT 5;

    - **SELECT** customer_id, **COUNT**(amount) **FROM** payment
      GROUP BY customer_id
      ORDER BY COUNT(amount)

    - **SELECT** customer_id, staff_id, **SUM**(amount) **FROM** payment
      GROUP BY staff_id, customer_id
      ORDER BY customer_id

- **Using DATE( ) to convert date time to date only**

```
SELECT DATE(payment_date), SUM(amount) FROM payment
GROUP BY DATE(payment_date)
ORDER BY SUM(amount)
```

- **HAVING**
  - The **HAVING** clause allows us to filter after aggregation has already taken place.
  - Example,
    - Before:

```
SELECT company, SUM(sales) FROM finance_table
WHERE company != 'Google'
GROUP BY company
```

    - After:
      - Filtering on base on SUM(sales)
      -

```
SELECT company, SUM(sales) FROM finance_table
WHERE company != 'Google'
GROUP BY company
HAVING SUM(sales) > 1000
```

    - **HAVING** allows us to use the aggregate result as a filter along with a **GROUP BY.**
    - **Example,**
      - **SELECT** customer_id, **SUM**(amount) **FROM** payment
        WHERE customer_id **NOT IN** (184,87,477)
        GROUP BY customer_id
        HAVING SUM(amount) > 200;

      - **SELECT** store_id, **COUNT**(*) **FROM** customer
        GROUP BY store_id
        HAVING COUNT(*) > 300

      - **SELECT** customer_id, **COUNT**(payment_id) **FROM** payment
        GROUP BY customer_id
        HAVING COUNT(payment_id) >= 40;

      - **SELECT** customer_id, staff_id, **SUM**(amount) **FROM** payment
        WHERE staff_id = 2
        GROUP BY customer_id, staff_id
        HAVING SUM(amount) >= 100
        ORDER BY customer_id;

- **AS**
  - **AS** clause which allows to create an 'alias' for column or result. (alias it's just a alternating name)
  - **Syntax:**
    1. **SELECT** column **AS** new_name **FROM** table;
    2. **SELECT SUM**(col) **AS** new_col **FROM** table;
  - **AS** operator executed the very end of query, meaning that we cannot use the ALIAS inside a WHERE operates.

- **JOIN** (Note: What's Venn Diagrams)

  - **JOINS** will allow us to combine information from multiple tables.
    - **DIFFERENT types of JOINS**
      - **INNER JOINS**
      - **OUTER JOINS**
      - **FULL JOINS**
      - **UNIONS**
- **INNER JOINS** (This clause use to founds matches in tables, table order won't matter)
  - **Syntax:**
    - **SELECT** * **FROM** TableA
      INNER JOIN TableB
      ON TableA.col_match = TableB.col_match

    - **SELECT** * **FROM** Registrations
      INNER JOIN Logins
      ON Registration.name = Logins.name

    - **SELECT** reg_id, Logins.name, log_id **FROM** Registrations
      INNER JOIN Logins
      ON Registration.name = Logins.name

    - **SELECT** payment_id, payment.customer_id, first_name
      FROM payment
      INNER JOIN customer
      ON payment.customer_id = customer.customer_id;

- **OUTER JOIN** (There are few different types of OUTER JOINS)
  - **1. FULL OUTER JOIN, 2. LEFT OUTER JOIN, 3. RIGHT OUTER JOIN**

1. **FULL OUTER JOIN**
   - It took all values and which value is not present in 2nd table or in 1st table became null value.
   - **Syntax:**
     - **SELECT** * **FROM** Registration
       FULL OUTER JOIN Logins
       ON Registration.name = Logins.name # It took all values

- **FULL OUTER JOIN with WHERE** (Get rows unique to either table == rows not found in both table == Opposite of **INNER JOIN**)
- **Syntax:**
  - **SELECT** * **FROM** Registration
    FULL OUTER JOIN Logins
    ON Registration.name = Logins.name
    WHERE Registration.is **IS** nus **OR** Logins.id **IS** null

  - **SELECT** * **FROM** customer
    FULL OUTER JOIN payment
    ON customer.customer_id = payment.customer_id
    WHERE customer.customer_id **IS** null
    OR payment.payment_id **IS** null # It took only unique values

2. **LEFT OUTER JOIN (IMP)*****
- A **LEFT OUTER JOIN** results in the set of records that are in the left table, if there is no match with the right table, the results are null.
- **Syntax:**
  - **SELECT * FROM** Registrations # (its consider ad left table)
    LEFT OUTER JOIN Logins #(Its consider as Right Table)
    ON Registrations.name = Logins.name

  - **SELECT** film.film_id, title, inventory_id, store_id
    FROM film
    LEFT JOIN inventory
    ON inventory.film_id = film.film_id
    WHERE inventory .film_id **IS** null

3. **RIGHT JOIN**
- A **RIGHT JOIN** is essentially the same as a **LEFT JOIN,** except the tables are switched.

- **UNION**
  - The **UNION** operator is used to combine the result-set of two or more **SELECT** statements.
  - It basically serves to directly concatenate (+) two results together, essentially “pasting” them together.
  - **Syntax:**
    - **SELECT** column_name(s) FROM table 1
      UNION
      SELECT column_name(s) FROM table 2;

    - **Example,**
      SELECT * FROM Sales2022_Q1
      UNION
      SELECT * FROM Sales2022_Q2;

  - **SELECT** title, first_name, last_name **FROM** film # Two INNER JOINs used
    INNER JOIN film_actor
    ON film.film_id = film_actor.film_id
    INNER JOIN actor
    ON film_actor.actor_id = actor.actor_id
    WHERE actor.first_name = 'Nick' **AND** actor.last_name = 'Wahlberg'

- **Timestamps and Extract**
  - These will be more useful when creating our own tables and databases, rather than when querying a database.
    - **TIME –** Contains only Time.
    - **DATE –** Contains only date
    - **TIMESTAMP –** Contains date and time
    - **TIMESTAMPTZ –** Contains date, time and time zone
  - Remember, you can always remove historical information, but you can't add it. (u cant go back and change time date or zone)
  - Use levels as per condition
  - **SHOW ALL ( )**
  - **SHOW TIMEZONE**
  - **SELECT NOW( ) : date time time zone**
  - **SELECT TIMEOFDAY( ) : day date time timezone year**
  - **SELECT CURRENT_TIME**
  - **SELECT CURRENT_DATE**

A. EXTRACT( )

- Allows to "extract" or obtain a sub-component of a sate value
  YEAR, MONTH, DAY, WEEK, QUARTE
  - **EXTRACT**(**YEAR FROM** date_col)

    - **SELECT EXTRACT**(**YEAR FROM** last_update) **AS** year **FROM** customer

B. AGE( )

- Calculate and return the current age given a timestamp.
  - **AGE**(date_col)
  - Returns:
    13 years 1 mon 5 days 01:34:13.003423

    - **SELECT AGE**(last_update) **FROM** customer

C. TO_CHAR( )

- General function to convert data types to text
- Useful for timestamp formatting
  - **TO_CHAR**(date_col, 'mm-dd-yyyy')

    - **SELECT TO_CHAR**(last_update, 'hh:mm') **FROM** customer

  - **SELECT DISTINCT(TO_CHAR(**payment_date,'MONTH'**)) FROM** payment

  - **SELECT COUNT(*) FROM** payment
  - **WHERE EXTRACT(dow FROM** payment_date**)** = 1 # dow is keyword google it (day of the week = dow)

- **Mathematical Functions and Operator (Click for Documentation)**

  - **SELECT ROUND**(rental_rate/replacement_cost*100, 2) **FROM** film;

- **String Functions and Operators (Click for Documentation)**

  - **SELECT** first_name || ' ' || last_name **FROM** customer

  - **SELECT LOWER**(**LEFT**(first_name,1) || last_name || '@gmail.com')
    AS custom_email
    FROM customer

- **SubQuery ***
  - Sub query allows to construct complex queries, essentially performing a query on the results of another query.
  - The syntax is straightforward and involves two SELECT statements.

  - **SELECT** student, grade **FROM** test_scores
    WHERE grade > (**SELECT AVG**(grade) **FROM** test_scores) # sub query and it runs 1st
  - The subquery is performed first since it is inside the parenthesis.

  - We can also use the **IN** operator in conjunction with a subquery to check against multiple results returned.
  - **SELECT** student, grade **FROM** test_scores
    WHERE student **IN** (**SELECT** student **FROM** honor_roll_table) # it checks result in 2nd table which is >> honor_roll_table

- **EXISTS*^**
  - The **EXISTS** operator is used to test for existence of rows in a **subquery.**
  - Typically a subquery is passed in the **EXISTS( )** function to check if any rows are returned with the subquery.
  - **Syntax:**
    - **SELECT** col_name **FROM** table_name
      WHERE EXISTS(**SELECT** col_name **FROM** table_name **WHERE** condition);

    - Example,
      SELECT title, rental_rate **FROM** film
      WHERE rental_rate > (**SELECT AVG**(rental_rate) **FROM** film)
      # first execute subquery and also it becomes single value

    - **SELECT** film_id, title **FROM** film
      WHERE film_id **IN** (**SELECT** inventory.film_id **FROM** rental
      INNER JOIN inventory **ON** inventory.inventory_id = rental.inventory_id
      WHERE return_date **BETWEEN** '2005-05-29' **AND** '2005-05-30')

- **Self-Join**
  - A self-join is a query in which a table is joined to itself.
  - Self-join are useful for comparing values in a column of rows within the same table.
  - The self join can be viewed as a join of two copies of the same table.
  - The table is not actually copied, but SQL performs the command as though it were.

- There is no special keyword for self join, its simply standard JOIN syntax with the same table in both parts.
- Alias is necessary for self-join (**AS**).
- **Syntax:**
    - **SELECT** tableA.col, tableB.col
      FROM table **AS** tableA
      JOIN table **AS** tableB
      ON tableA.some_col = tableB.other_col

        - Exampl,

```
SELECT f1.title, f2.title, f1.length
FROM film AS f1
INNER JOIN film AS f2
ON f1.film_id != f2.film_id
AND f1.length = f2.length
```

- **Creating Database and Tables**

- **Data Types**

    1. Boolean : True or False
    2. Character : char, varchar, and text
    3. Numeric : integer and floating-point number
    4. Temporal: data, time, timestamp, and interval
    5. UUID : Universal Unique Identifiers
    6. Array : Stores an array of strings, numbers, etc.
    7. JSON
    8. Hstore key-value pair
    9. Special types such as network address and geometric data.

- **Primary Keys and Foreign Keys**
    - **Primary key(PK)** is a column or a group of columns used to identify a row uniquely in a table.
    - **Primary Keys** are important since they allow us to easily discern what columns should be used for joining tables together.
    - Its integer based and unique.
    - **Foreign Key** is a field or group of fields in a table that uniquely identifies a row in another table.
    - A **foreign key** is defined in a table that references to the **primary key** of the other table.
    - The table that contains the **foreign key** is called referencing table or chilled table.
    - The table to which the **foreign key** references is called referenced table or parent table.
    - Table can have multiple foreign keys depending on its relationships with other tables.

- **Constraints**
    - Constraints are the rules enforced on data column on table.
    - These are used to prevent invalid data from being entered into the database.
    - This ensures the accuracy and reliability of the data in the database.
    - Its divided into two main categories:
        - Column Constraints
            - Constrains the data in a column to adhere to certain conditions.
        - Table Constraints
            - Applied to the entire table rather than to an individual column.
    - The most common constraints used:
        - **NOT NULL** Constraint
            - Ensure that a column cannot have NULL value.
        - **UNIQUE( )** Constraint
            - Ensure that all values in a column are different.
        - **PRIMARY KEY( )**
        - **FOREIGN KEY( )**
        - **CHECK( )**
            - **CHECK** ensures that all values in column satisfy certain conditions.

- **EXCUSION** Constraint
  - Ensures that if any two rows are compared on the specified column or expression using the specified operator, not all of these comparisons will return TRUE.

xxxxxxxxx

- **CREATE**
  - **Full General Syntax:**
    - **CREATE TABLE** table_name(
      Column_name **TYPE** column_constraint,
      Column_name **TYPE** column_constraint,
      table_constraint table_constraint
) **INHERITS** existing_table_name;
    - Example Syntax:
      CREATE TABLE player(
      player_id **SERIAL PRIMARY** KEY,
      age **SMALLINT NOT NULL**
      );
    - **SERIAL :**
      - It will create a sequence object and set the next value generated by the sequence as the default value for the column.
      - This is perfect for a primary key, because it logs unique integer entries for you automatically upon insertion.
      - If row is removed, the column with **SERIAL** data type will not adjust.
        For Example,
        1,2,3,5,6,7..
        - You know row 4 was removed at some point
- Steps to create new data_base and new table:
  - **Table 1 :**

CREATE TABLE account(
user_id **SERIAL PRIMARY KEY**, # It creates unique serial ID's and Constraint is PK
user_name **VARCHAR(50) UNIQUE NOT NULL**, # VARCHAR = Variable Charecter
password **VARCHAR(50) NOT NULL**,
email **VARCHAR(250) UNIQUE NOT NULL**,
created_on **TIMESTAMP NOT NULL**,
last_login **TIMESTAMP**
);

Table 2 :

```
CREATE TABLE job(
        job_id SERIAL PRIMARY KEY,
        job_name VARCHAR(200) UNIQUE NOT NULL
);
```

Table 3 :

```
CREATE TABLE account_job(
        user_id INTEGER REFERENCES account(user_id),  #its same ase account table but its it not req PE need to use ref. and interger act as serial FK

        job_id INTEGER REFERENCES job(job_id), # REFERENCE makes these column as foreign keys^^ FK

        hire_date TIMESTAMP
)
```

- **INSERT**
- **INSERT** allows you to add in rows to a table.
- **Syntax:**

```
INSERT INTO table(col1,col2,…..)
VALUES
        (value1,value2,…)
        (value1,value2,…), …..;
```

- **Syntax for inserting Values from another table:**

```
INSERT INTO table(col1,col2,..)
SELECT col1,col2,…
FROM another_table
WHERE condition;
```

- Remember, the inserted row values must match up for table, including constraint also not null
- **SERIAL** column do not need to be provide a value.

Inserting values in Table 1 :

```
INSERT INTO account(user_name, password, email, created_on)
VALUES
('Pavan', 'password', 'pavan@gmail.com', CURRENT_TIMESTAMP);
```

Inserting values in Table 2 :

```
INSERT INTO job(job_name)
VALUES
('Data Eng');
```

Inserting values in Table 3 :
INSERT INTO account_job(user_id, job_id, hire_date)
VALUES
(1,1,**CURRENT_TIMESTAMP**);

- **UPDATE**

- The **UPDATE** keyword allows for the changing of values of the columns in a table.
  Syntax:

```
UPDATE table
SET col1 = value1,
    col2 = value2,….
WHERE
Condition;
```

Example,

```
UPDATE account
SET last_login = CURRENT_TIMESTAMP
WHERE last_login IS NULL;
```

  - **Reset** everything without **WHERE** condition :
    1. **UPDATE** account
       SET last_login = **CURRENT_TIMESTAMP**

    2. **UPDATE** account
       SET last_login = created_on

  - Using another table's values (UPDATE join) # join keyword is not used.
    - **UPDATE** tableA
      SET original_col = tableB.new_col
      FROM tableB
      WHERE tableA.id = tableB.id

      UPDATE account_job
      SET hire_date = account.created_on
      FROM account
      WHERE account_job.user_id=account.user_id

  - Return affected rows
    - UPDATE account
      SET last_login = created_on
      RETURNING account_id, last_login

- **DELETE**

- **DELETE** clause used to remove rows from table.
  Example,

```
DELETE FROM table
WHERE row_id = 1
```

- We can delete rows based on their presence in other tables.
  Example,

```
DELETE FROM tableA
```

USING tableB
WHERE tableA.id = tableB.id

- Similar to **UPDATE** command, you can also add in a **RETURNING** call to return rows that were removed.

DELETE FROM job
WHERE job_name = 'Python Dev'
RETURNING job_id, job_name

- **ALTER clause**

- The **ALTER** clause allows for changes to an existing table structure, such as:
  - Adding, dropping, or renaming columns
  - Changing a column's data type
  - Set DEFAULT values for a column
  - Add CHECK constraints
  - Rename table
    Example,
    ALTER TABLE table_name action

    ALTER TABLE table_name **ADD COLUMN** new_col TYPE

    ALTER TABLE table_name **DROP COLUMN** col_name

    ALTER TABLE table_name ALTER COLUMN col_name **SET DEFAULT** value

    ALTER TABLE table_name ALTER COLUMN col_name **SET NOT NULL**

    ALTER TABLE table_name ALTER COLUMN col_name **DROP NOT NULL**

    ALTER TABLE table_name ALTER COLUMN col_name **ADD CONSTRAINT** constraint_name

  - Rename
    ALTER TABLE information **RENAME TO** new_info
    ALTER TABLE new_info **RENAME COLUMN** person **TO** people
  - **NOT NULL error**
    ALTER TABLE new_info ALTER COLUMN people DROP NOT NULL
    INSERT INTO new_info(title) VALUES ('some_new_title') # before above line execution this insertion is gave some error regarding not null values after this we are able to insert single value with other nulls.

	info_id [PK] integer	title character varying (500)	peopel character varying (50)
1	1	some_new_title	[null]

- **DROP keyword**
  - **DROP** allows for the complete removal of a column in a table.
  - In PostgreSQL this will also automatically remove all of its indexes and contraints involving the column.
  - However, it will not remove columns used in views, triggers, or stored procedures without the additional CASCADE clause.(i.e. Depending columns)
  - **Syntax:**

    ALTER TABLE table_name
    DROP COLUMN col_name

    - To remove all dependencies

      ALTER TABLE table_name
      DROP COLUMN col_name **CASCADE**

    - Check for existence to avoid error you get Notice

      ALTER TABLE table_name

      DROP COLUMN IF EXISTS col_name

    - **Drop multiple columns**

      ALTER TABLE table_name
      DROP COLUMN col_one,
      DROP COLUMN col_two;

- **CHECK Constraint**
  - The **CHECK** constraint allows us to create more customized constraints that adhere to a certain condition.
  - Such as making sure all inserted integer values fall below a certain threshold.
  - **Syntax**

    CREATE TABLE example(
    Ex_id **SERIAL PRIMARY KEY**,
    Age **SMALLINT CHECK** (age>21),
    Parent_age **SMALLINT CHECK**(parent_age > age)
    );

    ^^^
    In this case if your age is less than 21 or
    Parents age is less than your age both gives an error

```
CREATE TABLE employees(
        emp_id SERIAL PRIMARY KEY,
        first_name VARCHAR(50) NOT NULL,
        last_name VARCHAR(50) NOT NULL,
        birthdate DATE CHECK (birthdate > '1900-01-01'),
        hire_date DATE CHECK (hire_date >  birthdate),
        salary INTEGER CHECK (salary >0)
)
```

```
INSERT INTO employees(
	first_name,
	last_name,
	birthdate,
	hire_date,
	salary
)
VALUES
('Akshay',
'Kulthe',
 '1899-11-06',
 '2010-01-01',
 10000
)
```

Result :

ERROR: new row for relation "employees" violates check constraint "employees_birthdate_check" DETAIL: Failing row contains (1, Akshay, Kulthe, 1899-11-06, 2010-01-01, 10000).

```
INSERT INTO employees(
	first_name,
	last_name,
	birthdate,
	hire_date,
	salary
)
VALUES
('Pavan',
'Bharambe',
 '1995-11-06',
 '2010-01-01',
 10000
)
```

Result : Successful insertion.

```
CREATE TABLE students(
	student_id SERIAL PRIMARY KEY,
	first_name VARCHAR(50) NOT NULL,
	last_name VARCHAR(50) NOT NULL,
	homeroom_number SMALLINT CHECK (homeroom_number>0),
	phone INTEGER CHECK (phone>0),
	email VARCHAR(250) UNIQUE,
	graduation_year SMALLINT

)

INSERT INTO students(
	first_name,
	last_name,
	phone,
	graduation_year,
	homeroom_number
)
VALUES
('Mark',
 'Watney',
 777551234,
 2035,
 5
)
```

****

```
CREATE TABLE teachers(
	teacher_id SERIAL PRIMARY KEY,
	first_name VARCHAR(50) NOT NULL,
	last_name VARCHAR(50) NOT NULL,
	homeroom_number SMALLINT CHECK (homeroom_number > 0),
	department VARCHAR(50) NOT NULL,
	email VARCHAR(250) UNIQUE NOT NULL,
	phone INTEGER
)

INSERT INTO teachers(
	first_name,
	last_name,
	homeroom_number,
	department,
	email,
	phone
)
VALUES
(
	'Jonas',
	'Salk',
	5,
	'Biology',
	'jsalk@school.org',
	777554321
)
```

- **Conditional Expressions and Procedures**

- **CASE, COALESCE, NULLIF, CAST, Views, Import and Export Functionality**
- These^^ keyword and functions will allow us to add logic to our commands and workflow in SQL

- **CASE statement**

- Same as **IF/ELSE** statement, **CASE** statement only execute SQL code when certain conditions are met.
- There are two main ways to use **CASE** statement, either a general **CASE** or a **CASE expression**.
- Both methods can lead to the same results.

- **Syntax :**

```
CASE
        WHEN condition1 THEN result1
        WHEN condition2 THEN result2
        ELSE some_other_result

END
```

Example,

```
        SELECT a,
        CASE WHEN a =1 THEN 'ONE'
                WHEN a = 2 THEN 'TWO'
        ELSE 'other' AS label                   # by alias we use any name to new case table
        END
        FROM table1;   # Result came into new column
```

- **CASE expression**

- The **CASE expression** syntax first evaluates an expression then compares the result with each value in the **WHEN** clause sequentially.

- **Syntax :**

```
CASE expression
        WHEN value 1 THEN result1
        WHEN value2 THEN result2
        ELSE some_other_result
END
```

```
SELECT a,

        CASE a WHEN 1 THEN 'ONE'  # Here a is a expression

                WHEN 2 THEN 'TWO'

                ELSE 'other'

        END

FROM table1;
```

Examples,

```
SELECT customer_id,
        CASE
                WHEN (customer_id <= 100) THEN 'Premium'
                WHEN (customer_id BETWEEN 100 and 200) THEN 'Plus'
                ELSE 'Normal'

END AS customer_class
FROM customer
**********************************************************************************************
SELECT
        CASE customer_id
                WHEN 100 THEN 'Premium'
                WHEN 200 THEN 'Plus'
                ELSE 'Normal'

END AS customer_class
FROM customer
**********************************************************************************************
SELECT
SUM(CASE rating
                WHEN 'R' THEN 1
END) AS r,
SUM(CASE rating
                WHEN 'PG' THEN 1
END) AS pg,
SUM(CASE rating
                WHEN 'PG-13' THEN 1
END) AS pg13
FROM film;
```

	r bigint	pg bigint	pg13 bigint
1	195	194	223

- **COALESCE**(useful to find NON NULL vales from table columns)

- This function accepts an unlimited number of arguments. It returns the first argument that is not null. If all arguments are null, the COALESCE function will return null.
- **COALESCE(arg_1,arg_2,….)**
- Example,

```
SELECT COALESCE(1, 2)
= 1
SELECT COALESCE(NULL,1, 2)
= 2
```

- The COALESCE function becomes useful when querying a table that contains null values and substituting it with another value.
- Null changed to 0 (zero) It will help in mathematical operations.
- Example,

SELECT item, (price – **COALESCE**(discount,0)) **AS** final **FROM** table

- **CAST** operator or function

- THE **CAST** operator convert one data type into another data type.
- **Syntax for CAST function**
  SELECT CAST(‘5’ **AS INTEGER**)

- **PostgreSQL CAST Operator**
  SELECT ‘5’ **: : INTEGER**

  Example,
  SELECT CHAR_LENGTH(CAST(inventory_id **AS VARCHAR))FROM rental**

- **NULLIF**
- The NULLIF function takes in 2 inputs and returns NULL if both are equal, otherwise it returns the first argument passed. NULLIF(arg_1,arg_2)
- Example,
  NULLIF(10, 10)
  = Returns NULL

  NULLIF(10, 12)
  = Returns 10
- Tis becomes very useful in cases where a NULL value would cause an error or unwanted result.
- Example,

SELECT * **FROM** depts

	first_name character varying (50)	department character varying (50)
1	Akshay	Civil
2	Pavan	Civil
3	Nishat	Electrical

SELECT(
SUM(**CASE WHEN** department = 'Civil' **THEN** 1 **ELSE** 0 **END**)/
SUM(**CASE WHEN** department = 'Electrical' **THEN** 1 **ELSE** 0 **END**)
) **AS** department_ratio
FROM depts

	department_ratio bigint
1	2

DELETE FROM depts **WHERE** department = 'Electrical'

SELECT * **FROM** depts

	first_name character varying (50)	department character varying (50)
1	Akshay	Civil
2	Pavan	Civil

SELECT(
SUM(**CASE WHEN** department = 'Civil' **THEN** 1 **ELSE** 0 **END**)/
SUM(**CASE WHEN** department = 'Electrical' **THEN** 1 **ELSE** 0 **END**)
) **AS** department_ratio
FROM depts

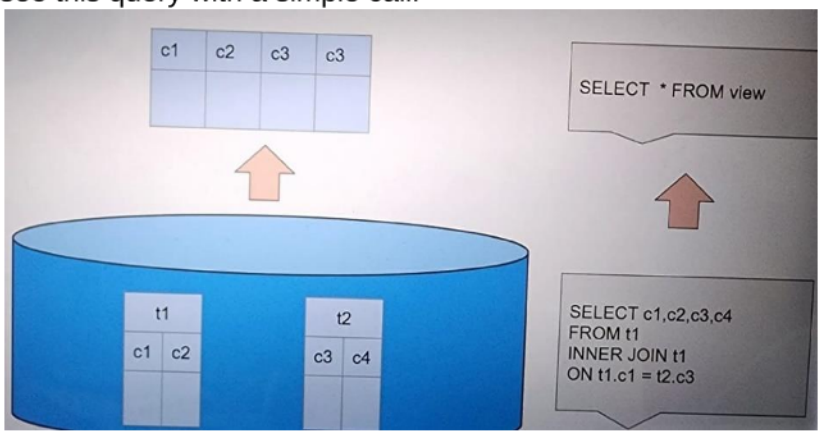
```
ERROR:  division by zero
```

```
SELECT(
SUM(CASE WHEN department = 'Civil' THEN 1 ELSE 0 END)/
NULLIF(SUM(CASE WHEN department = 'Electrical' THEN 1 ELSE 0 END), 0) # NULLIF(arg_1,arg_2)
) AS department_ratio
FROM depts
```

	department_ratio bigint
1	[null]

- **VIEWS**

- Often there are specific combinatios of tables and conditions that you find yourself using quite often for project.
- Instead of having to perform the same query over and over again as a starting point, you can create a VIEW to quickly see this query with a simple call.



- A view is a database object that is of a stored query.
- A view can be accessed as a virtual table in PostgreSQL.
- Notice that a VIEW dose not store data physically, it simply stores the query.
- Example,

```
CREATE VIEW customer_info AS
SELECT first_name, last_name, address FROM customer
INNER JOIN address
ON customer.address_id = address.address_id
```

```
CREATE VIEW

Query returned successfully in 54 msec.
```

```
SELCT * FROM customer_info
```

	first_name character varying (45)	last_name character varying (45)	address character varying (50)
1	Jared	Ely	1003 Qinhuangdao Street
2	Mary	Smith	1913 Hanoi Way
3	Patricia	Johnson	1121 Loja Avenue
4	Linda	Williams	692 Joliet Street
5	Barbara	Jones	1566 Inegl Manor
6	Elizabeth	Brown	53 Idfu Parkway
7	Jennifer	Davis	1795 Santiago de Compostela Way
8	Maria	Miller	900 Santiago de Compostela Parkway
9	Susan	Wilson	478 Joliet Way
10	Margaret	Moore	613 Korolev Drive
11	Dorothy	Taylor	1531 Sal Drive
12	Lisa	Anderson	1542 Tarlac Parkway

Total rows: 599 of 599 | Query complete 00:00:00.061

CREATE OR REPLACE VIEW customer_info **AS**
SELECT first_name, last_name, address, district **FROM** customer
INNER JOIN address
ON customer.address_id = address.address_id

	first_name character varying (45)	last_name character varying (45)	address character varying (50)	district character varying (20)
1	Jared	Ely	1003 Qinhuangdao Street	West Java
2	Mary	Smith	1913 Hanoi Way	Nagasaki
3	Patricia	Johnson	1121 Loja Avenue	California
4	Linda	Williams	692 Joliet Street	Attika

- **To delete VIEW**
  DROP VIEW IF EXISTS customer_info

- **To rename**
  ALTER VIEW customer_info **RENAME to** c_info

- **Import and Export**

- Importing data from a .csv file to an already existing table.
- IMP links for import and export
- https://postgresql.org/docs/12/sql-copy.html
- https://stackoverflow.com/questions/2987433/how-to-import-csv-file-data-into-a-postgresql-table
- https://www.enterprisedb.com/postgres-tutorials/how-import-and-export-data-using-csv-files-postgresql
- https://stackoverflow.com/questions/21018256/can-i-automatically-create-a-table-in-postgresql-from-a-csv-file-with-headers

```
CREATE TABLE bank_customer(
    customer_id INTEGER,
    credit_score INTEGER,
    country VARCHAR(100),
    gender VARCHAR(50),
    age SMALLINT,
    tenure SMALLINT,
    balance FLOAT,
    product_number SMALLINT,
    credit_card SMALLINT,
    active_member SMALLINT,
    estimated_salary FLOAT,
    churn SMALLINT
)
```

	customer_id integer	credit_score integer	country character varying (100)	gender character varying (50)	age smallint	tenure smallint	balance double precision	product_number smallint	credit_card smallint	active_member smallint	estimated_salary double precision	churn smallint

>>database>>Schema>>public>>Tables>>right-click on bank_customer select import-export after creating table 1st with correct data types for each column.

	customer_id integer	credit_score integer	country character varying (100)	gender character varying (50)	age smallint	tenure smallint	balance double precision	product_number smallint	credit_card smallint	active_member smallint	estimated_salary double precision	churn smallint
1	15634602	619	France	Female	42	2	0	1	1	1	101348.88	1
2	15647311	608	Spain	Female	41	1	83807.86	1	0	1	112542.58	0
3	15619304	502	France	Female	42	8	159660.8	3	1	0	113931.57	1
4	15701354	699	France	Female	39	1	0	2	0	0	93826.63	0

- **PostGreSQL with Python**

➢ Retrieve a cursor **(pointer)

In [ ]:
```
# After installing with pip install psycopg2
import psycopg2 as pg2
```

In [ ]:
```
# Create a connection with PostgreSQL
# 'password' is whatever password you set, we set password in the install video
conn = pg2.connect(database='postgres', user='postgres',password='password')
```

In [ ]:
```
# Establish connection and start cursor to be ready to query
cur = conn.cursor()
```

In [ ]:
```
# Pass in a PostgreSQL query as a string
cur.execute("SELECT * FROM payment")
```

In [ ]:
```
# Return a tuple of the first row as Python objects
cur.fetchone()
```

In [ ]:
```
# Return N number of rows
cur.fetchmany(10)
```

In [ ]:
```
# Return All rows at once
cur.fetchall()
```

In [ ]:
```
# To save and index results, assign it to a variable
data = cur.fetchmany(10)
```

Inserting Information

In [2]:
```
query1 = '''
        CREATE TABLE new_table (
            userid integer
            , tmstmp timestamp
            , type varchar(10)
        );
        '''
```

In [ ]:
```
cur.execute(query1)
```

In [ ]:
```
# commit the changes to the database
cur.commit()
```

In [ ]:
```
# Don't forget to close the connection!
# killing the kernel or shutting down juptyer will also close it
conn.close()
```

- **MS SQL Vs PostgreSQL**

  https://www.enterprisedb.com/blog/microsoft-sql-server-mssql-vs-postgresql-comparison-details-what-differences