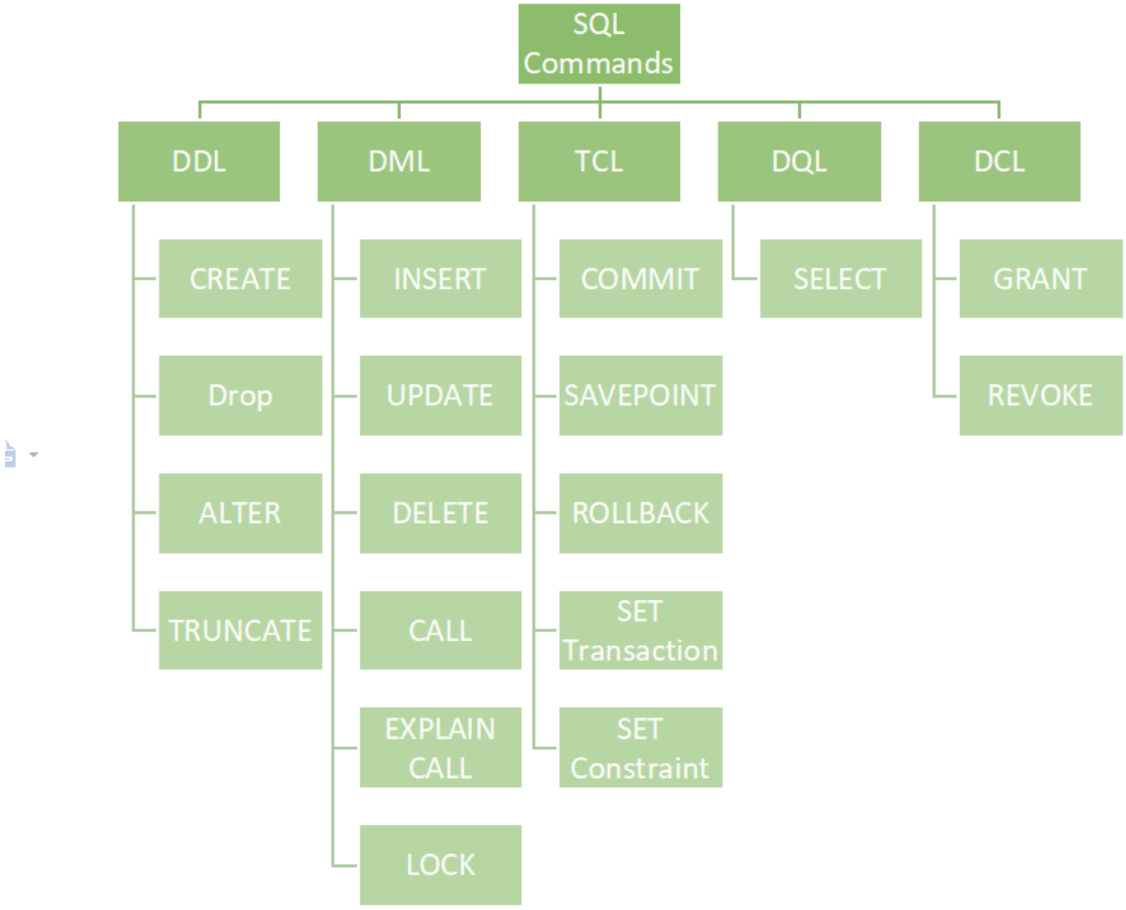
SQL hierarchy : Server → Database → Table → Data

ACID stands for Atomicity, Consistency, Isolation, and Durability, and these properties are crucial for maintaining the integrity of data in a database management system (DBMS).



Warning: The provided syntax is not common for MySQL and PostgreSQL, but variations may exist in keywords or options. Always refer to the official documentation of your specific database system for accurate commands and details.

Below is some example:

1. **Rename column**

MySQL: ALTER TABLE student_info CHANGE COLUMN mobile phone INT;
PostgreSQL: ALTER TABLE student_info RENAME COLUMN mobile TO phone;

2. **Modify column**

MySQL: ALTER TABLE student_info MODIFY COLUMN mobile INT;
PostgreSQL: ALTER TABLE student_info ALTER COLUMN mobile TYPE INT;

3. **Changing the table name (varies between databases):**

MySQL: RENAME TABLE oldTableName TO newTableName;
PostgreSQL: ALTER TABLE oldTableName RENAME TO newTableName;

DDL (Data Definition Language)

1. **Create -** Used to create database objects such as tables, indexes, views, and stored procedures.
   - **create database:** create database database_name
     Eg 1 create database student
   - **create table:** create table table_name (
     column1 datatype1,
     column2 datatype2,
     ...
     primary key (one_or_more_columns)
)
     Eg 1 create table student_info(rollno int primary key, name varchar(50), bloodgroup varchar(10))

2. **Alter -** Used to modify the structure of existing database objects, such as adding or drop or modifying columns from a table.
   - **Add a new column**: alter table table_name add column new_column1 datatype, add column new_column2 datatype;
     Eg 1 alter table student_info add column age int

   - **Modify a column:** alter table table_name [rename/change/alter] column old_column new_column type datatype
     Eg 1 alter table student_info rename column rollno reg_no -- column name
     Eg 2 alter table student_info alter column rollno type bigint -- column datatype

   - **Drop a column:** alter table table_name drop column column_to_drop
     Eg 1 alter table student_info drop column age

3. **Drop -** Used to delete database objects, such as tables, indexes, or views.
   - drop database database_name
     Eg 1 drop datavase student
   - drop table table_name
     Eg 1 drop table student_info

4. **Truncate -** Removes all records from a table but keeps the table structure for future use.
   - truncate table table_name
     Eg 1 truncate table student_info

DQL (Data Query Language)

1. **Select -** Used to retrieve data from one or more tables.
   - select column1, column2, ... from table_name where condition;

DML (Data Manipulation Language)

1. **Insert -** Used to add new records (rows) to a table.
    - insert into table_name (column1, column2, ...) values (value1, value2, ...);
2. **Update (set) -** Used to modify existing records in a table using **set**.
    - update table_name set column1 = value1, column2 = value2, ... where condition;
3. **Delete** - Used to remove records from a table.
    - delete from table_name where condition;

Example 1

1. create table employees (employee_id int primary key, first_name varchar(50), last_name varchar(50), salary int)

2. insert into employees (employee_id, first_name, last_name, salary) values (1, 'alice', 'johnson', 60000), (2, 'bob', 'smith', 70000), (3, 'charlie', 'brown', 55000)

3. select * from employees where salary > 60000

4. insert into employees (employee_id, first_name, last_name, salary) values (4, 'david', 'miller', 80000);

5. update employees set salary = 75000 where first_name = 'bob'

6. delete from employees where employee_id = 3;

Example 2

1. create table students (student_id int primary key, first_name varchar(50), last_name varchar(50), age int, grade varchar(2))

2. insert into students (student_id, first_name, last_name, age, grade) values (1, 'john', 'doe', 18, 'a'), (2, 'alice', 'smith', 20, 'b'), (3, 'bob', 'johnson', 19, 'c')

3. select * from students where age < 20

4. insert into students (student_id, first_name, last_name, age, grade) values (4, 'emma', 'brown', 22, 'b')

5. update students set age = 21 where first_name = 'alice'

6. delete from students where student_id = 3

Join operations

1. **Join** - Combines rows from two or more tables based on a related column between them.
    - select column1, column2, ... from table1 join table2 on table1.column_name = table2.column_name;

2. **Inner join** - Returns only the rows where there is a match in both tables.
    - select column1, column2, ... from table1 inner join table2 on table1.column_name = table2.column_name;

3. **Left join** - Returns all rows from the left table and the matched rows from the right table. If there is no match, NULL values are returned for columns from the right table.
    - select column1, column2, ... from table1 left join table2 on table1.column_name = table2.column_name;

4. **Right join (or right outer join)** - Returns all rows from the right table and the matched rows from the left table. If there is no match, NULL values are returned for columns from the left table.
    - select column1, column2, ... from table1 right join table2 on table1.column_name = table2.column_name;

5. **Full join (or full outer join)** - Returns all rows when there is a match in either the left or right table. If there is no match, NULL values are returned for columns from the table without a match.
    - select column1, column2, ... from table1 full join table2 on table1.column_name = table2.column_name;

Clause in SQL

1. **From** - Specifies the table or tables from which to retrieve data.
    - select column1, column2, ...from table_name;

2. **Where** - Filters the rows returned by a query based on specified conditions.
    - select column1, column2, ... from table_name where condition;

3. **Order by** - Sorts the result set based on specified columns and sort orders.
    - select column1, column2, ... from table_name where condition order by column1 [asc | desc], column2 [asc | desc], ...;

4. **Group by** - Groups rows that have the same values in specified columns into summary rows
    - select column1, column2, ..., aggregate_function(column) from table_name where condition group by column1, column2, ...;

5. **Limit** - Limits the number of rows returned by a query.
    - select column1, column2, ... from table_name where condition order by column1 [asc | desc], column2 [asc | desc], ... limit number_of_rows;

6. **Offset** - Skips a specified number of rows before starting to return rows.

- select column1, column2, ... from table_name where condition order by column1 [asc | desc], column2 [asc | desc], ... limit number_of_rows offset offset_value;

7. **Union** - Combines the result sets of two or more SELECT statements into a single result set.
    - select column1, column2, ... from table1 where condition1 union select column1, column2, ... from table2 where condition2;

8. **Distinct** - Removes duplicate rows from the result set.
    - select distinct column1, column2 from table_name;

Aggregate functions

Aggregate functions perform a calculation on a set of values and return a single value. These functions are often used with the GROUP BY clause to perform calculations on groups of rows

Syntex: select column1, column2, ..., aggregate_function(column) from table_name where condition group by column1, column2, ...;

1. **sum()**: calculates the sum of values in a numeric column.
    - select sum(salary) as total_salary from employees;

2. **avg():** calculates the average (mean) of values in a numeric column.
    - select avg(salary) as avg_salary from employees;

3. **count():** counts the number of rows or non-null values in a column.
    - select count(employee_id) as employee_count from employees;

4. **min():** finds the minimum value in a column.
    - select min(salary) as min_salary from employees;

5. **max():** finds the maximum value in a column.
    - select max(salary) as max_salary from employees;

6. **having:** filters the results of a group by clause based on specified conditions.
    - select department_id, avg(salary) as avg_salary from employees group by department_id having avg(salary) > 50000;

7. **group_concat():** concatenates values from multiple rows into a single string, often used with group by.
    - select department_id, group_concat(employee_name) as employee_names from employees group by department_id;

SQL Operators

Comparison Operators:

1. = (Equal to)
2. != or <> (Not equal to)
3. < (Less than)
4. > (Greater than)
5. <= (Less than or equal to)
6. >= (Greater than or equal to)

Example:

select column1, column2, ... from table_name where column1 = value1 and column2 > value2,…;

- select * from employees where salary = 50000;
- select * from products where category_id <> 3;
- select * from orders where order_amount < 1000;
- select * from products where price > 50;
- select * from customers where registration_year <= 2020;
- select * from employees where years_of_service >= 5;
- select * from orders where order_amount > 100 and order_date >= '2023-01-01';

Logical Operators:

1. **and** - true if all the conditions separated by and is true
   - select column1, column2, ... from table_name where condition1 and condition2;
     Eg1 select * from orders where order_status = 'shipped' and total_amount > 1000;

2. **or** - true if any of the conditions separated by or is true
   - select column1, column2, ... from table_name where condition1 or condition2;
     Eg1 select * from orders where order_status = 'shipped' or order_status = 'in transit';

3. **not** - displays a record if the condition(s) is not true
   - select column1, column2, ... from table_name where condition1 not condition2;
     Eg1 select * from products where not discontinued = 1;

4. **between** - filters the result set to include only rows where a column value is within a specified range.
   - select column1, column2, ... from table_name where column_name between value1 and value2;
     Eg1 select * from employees where age between 25 and 35;
     Eg2 select * from orders where order_date between '2022-01-01' and '2022-12-31' and order_status = 'shipped';

5. **in** - specifies multiple values in a where clause for a column.
   - select column1, column2, ... from table_name where column_name in (value1, value2, ...);

Eg1 select * from products where category_id in (1, 2, 3);
Eg2 select * from employees where department in ('hr', 'it', 'finance');

6. **like** - used in a where clause to search for a specified pattern in a column.
   - select column1, column2, ... from table_name where column_name like pattern;
     Eg1 select * from employees where last_name like 'sm%';
     Eg2 select * from products where product_name like 'laptop%';

7. **is null** - true if the operand is null.
   - select column1, column2, ... from table_name where column_name is null;
     Eg1 select * from employees where middle_name is null;
     Eg2 select * from orders where shipping_address is null;

8. **is not null** - true if the operand is not null.
   - select column1, column2, ... from table_name where column_name is not null;
     Eg1 select * from orders where ship_date is not null;
     Eg2 select * from customers where email is not null;

9. **is true / is false** - true if the operand is explicitly true or false
   - select * from products where discontinued is true;

10. **all** - true if all of the subquery values meet the condition
    - select * from products where price > all (select average_price from price_stats);

11. **any** - true if any of the subquery values meet the condition
    - select * from products where price > any (select competitor_price from competitors);

12. **exists** - true if the subquery returns one or more records
    - select * from customers where exists (select 1 from orders where customers.customer_id = orders.customer_id);

13. **some** - true if any of the subquery values meet the condition
    - select * from products where price > some (select competitor_price from competitors);

Arithmetic Operators:
+ Add, - Subtract, * Multiply, / Divide and % Modulo
Example: select column1 + column2 as sum_result from table_name;

Concatenation Operator:
|| (Concatenates two strings)
Example: select first_name || ' ' || last_name as full_name from employees;

Special wildcard characters

In SQL, the asterisk is often used as a wildcard character, but it also has a specific meaning when used in a SELECT statement.
When used in the SELECT clause, the asterisk is a shorthand way of selecting all columns from a table. It is often used to retrieve all available columns without explicitly listing each one.

1. **Asterisk (*):** select * from table_name;

In SQL, the two main special wildcard characters used with the LIKE operator for pattern matching are the percent sign (%) and the underscore (_). Here's a summary of their use:

1. **Percent Sign %:** Represents zero, one, or multiple characters in a string.
   - Example: LIKE '**a%**' matches any string that starts with 'a', such as 'apple' or 'apricot'.
   - Example: LIKE '**%ing**' matches any string that ends with 'ing', like 'running' or 'singing'.
   - Example: LIKE '**%an%**' matches any string that contains 'an' anywhere within it, such as 'banana' or 'stand'.

2. **Underscore _:** Represents a single character in a string.
   - Example: LIKE '**_pple**' matches any five-letter word ending with 'pple', such as 'apple'.
   - Example: LIKE '**h_ll%**' matches any word of at least four characters, starting with 'h' and ending with 'll', like 'hello' or 'hall'.

These wildcard characters are essential for flexible pattern matching when working with string data. They are commonly used in conjunction with the WHERE clause to filter and retrieve specific data from a database based on patterns

Keyword

1. **database -** Used to create a new database in SQL. After creating a database, you can use the USE statement to switch to that database.
   - create database database_name;
2. **table -** Used to create a new table in a database. After creating a table, you can use the ALTER, DROP & TRUNCATE TABLE statement to modify, delete, remove all rows from a table,
   - create table table_name (column1 datatype, column2 datatype, ...);
   - alter table table_name add column new_column datatype
   - alter table table_name change old_column new_column datatype
   - drop table table_name
   - truncate table table_name
3. **use -** Used to switch to a specific database.
   - use database_name;
4. **Into -** Used in conjunction with SELECT / INSERT statement to insert the selected data into a new table.
   - select column1, column2, ... into new_table from existing_table where condition;
   - insert into table_name (column1, column2, ...) values (value1, value2, ...);
5. **Values -** Used in INSERT INTO statement to specify the values to be inserted into a table.
   - insert into table_name (column1, column2, ...) values (value1, value2, ...);
6. **Set -** Used in UPDATE statement to set new values for columns in an existing row.
   - update table_name set column1 = value1, column2 = value2, ... where condition;

7. **desc (Descending) -** Used in the ORDER BY clause to sort the result set in descending order.
    - select column1, column2, ... from table_name order by column1 desc;
8. **asc (Ascending) -** Used in the ORDER BY clause to sort the result set in ascending order (default).
    - select column1, column2, ... from table_name order by column1 asc;
9. **add -** Used in the ALTER TABLE statement to add a new column to an existing table.
    - alter table table_name add column new_column_name data_type;
10. **change -** Used in the ALTER TABLE statement to change the name and/or data type of an existing column.
    - alter table table_name change old_column_name new_column_name data_type;

Data type

Numeric Types:

1. **int** : (Integer) Whole numbers without decimal points.
2. **bigint**: A large integer. The size parameter specifies the maximum display width (which is 255)
3. **float (size, d)**: Floating-point numbers with decimal points.

Character Strings:

1. **char(n)**: Fixed-length character string with a specified length 'n'.
2. **varchar(n)**: Variable-length character string with a maximum length 'n'.
3. **text**: Variable-length character string with no specified maximum length.

Date and Time Types:

1. **date**: Stores a date value in the format 'YYYY-MM-DD'.
2. **time**: Stores a time value in the format 'HH:MI:SS'.
3. **datetime or timestamp**: Stores both date and time values.

Example

Example 1:

```
-- DDL: Create a database
CREATE DATABASE company;

-- DDL: Create tables
CREATE TABLE employees (
    employee_id INT PRIMARY KEY,
    first_name VARCHAR(50),
    last_name VARCHAR(50),
    salary INT
);
```

```
CREATE TABLE departments (
   department_id INT PRIMARY KEY,
   department_name VARCHAR(50)
);

-- DML: Insert data into tables
INSERT INTO employees (employee_id, first_name, last_name, salary)
VALUES (1, 'Alice', 'Johnson', 60000),
      (2, 'Bob', 'Smith', 70000),
      (3, 'Charlie', 'Brown', 55000);

INSERT INTO departments (department_id, department_name)
VALUES (1, 'HR'),
      (2, 'IT'),
      (3, 'Finance');

-- DQL: Select data from tables with a join
SELECT employees.employee_id, first_name, last_name, salary, department_name
FROM employees
INNER JOIN departments ON employees.employee_id = departments.department_id;

-- DML: Update records
UPDATE employees SET salary = 75000 WHERE first_name = 'Bob';

-- DML: Delete records
DELETE FROM employees WHERE employee_id = 3;

-- DQL: Select data with aggregate functions
SELECT department_name, AVG(salary) AS avg_salary, COUNT(employee_id) AS
employee_count FROM employees
INNER JOIN departments ON employees.employee_id = departments.department_id
GROUP BY department_name
HAVING AVG(salary) > 60000;

-- SQL Clauses: Order By, Limit, Offset
SELECT * FROM employees ORDER BY salary DESC LIMIT 2 OFFSET 1;

-- SQL Clauses: Group By, Having
SELECT department_id, AVG(salary) AS avg_salary
FROM employees
GROUP BY department_id
HAVING AVG(salary) > 60000;

-- SQL Clauses: Union
SELECT employee_id, first_name, last_name FROM employees
UNION
SELECT department_id, department_name, NULL FROM departments;

-- SQL Clauses: Distinct
SELECT DISTINCT department_id FROM employees;
```

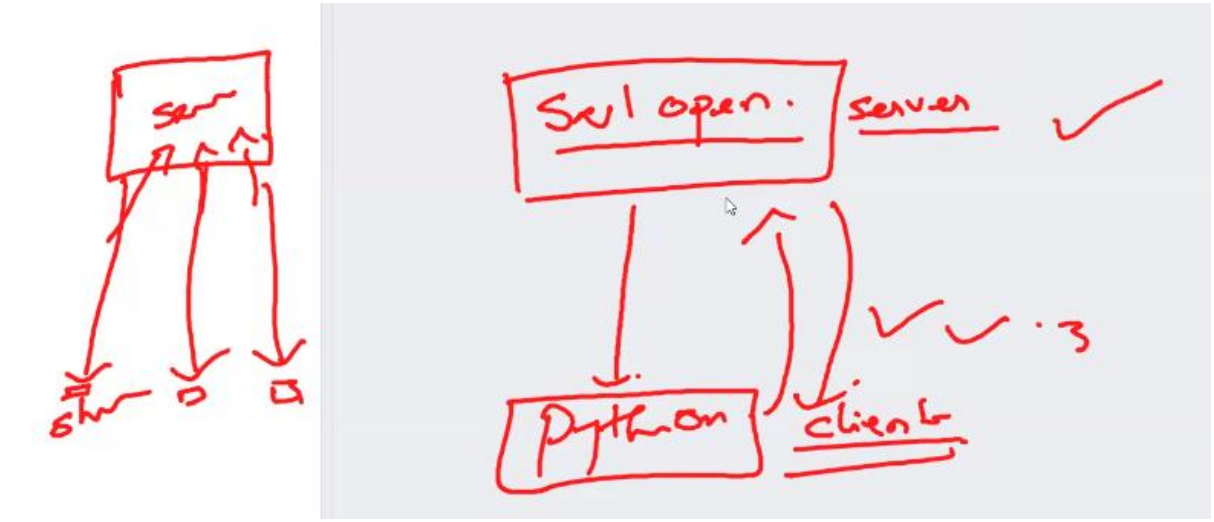
```
-- SQL Wildcard Characters
SELECT * FROM employees WHERE last_name LIKE 'S%';

-- DDL: Drop the database (clean-up)
DROP DATABASE company;
```

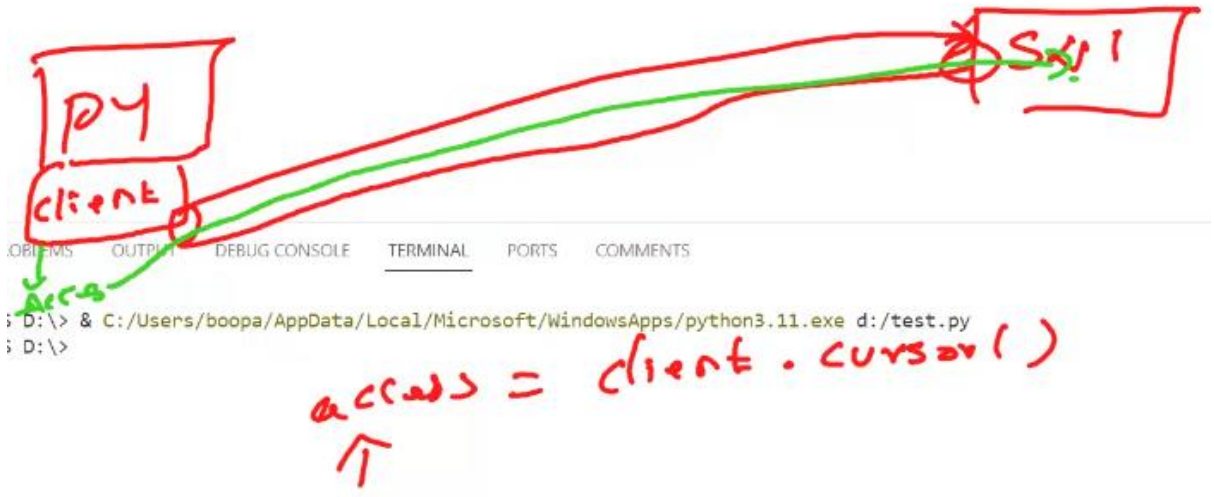
Example 2:

```
-- DDL: Create a database
CREATE DATABASE university;

-- DDL: Create tables
CREATE TABLE students (
    student_id INT PRIMARY KEY,
    first_name VARCHAR(50),
    last_name VARCHAR(50),
    age INT,
    grade VARCHAR(2)
);

CREATE TABLE courses (
    course_id INT PRIMARY KEY,
    course_name VARCHAR(50),
    department VARCHAR(50)
);

CREATE TABLE enrollments (
    enrollment_id INT PRIMARY KEY,
    student_id INT,
    course_id INT,
    grade VARCHAR(2),
    FOREIGN KEY (student_id) REFERENCES students(student_id),
    FOREIGN KEY (course_id) REFERENCES courses(course_id)
);

-- DML: Insert data into tables
INSERT INTO students (student_id, first_name, last_name, age, grade)
VALUES (1, 'John', 'Doe', 18, 'A'),
       (2, 'Alice', 'Smith', 20, 'B'),
       (3, 'Bob', 'Johnson', 19, 'C');

INSERT INTO courses (course_id, course_name, department)
VALUES (101, 'Introduction to Programming', 'Computer Science'),
       (102, 'Mathematics I', 'Mathematics'),
       (103, 'English Composition', 'English');

INSERT INTO enrollments (enrollment_id, student_id, course_id, grade)
```

```
VALUES (1, 1, 101, 'A'),
       (2, 2, 102, 'B'),
       (3, 3, 103, 'A');

-- DQL: Select data from multiple tables with join
SELECT students.student_id, first_name, last_name, age, grade, course_name, department
FROM students
JOIN enrollments ON students.student_id = enrollments.student_id
JOIN courses ON enrollments.course_id = courses.course_id;

-- DQL: Select data with aggregate function and grouping
SELECT courses.course_id, course_name, AVG(age) AS avg_age, COUNT(student_id) AS
student_count
FROM courses
JOIN enrollments ON courses.course_id = enrollments.course_id
JOIN students ON enrollments.student_id = students.student_id
GROUP BY courses.course_id, course_name
HAVING COUNT(student_id) > 1;

-- SQL Clauses: Order By, Limit
SELECT first_name, last_name, grade
FROM students
ORDER BY age DESC
LIMIT 2;

-- SQL Clauses: Group By, Having
SELECT department, AVG(age) AS avg_age
FROM courses
JOIN enrollments ON courses.course_id = enrollments.course_id
JOIN students ON enrollments.student_id = students.student_id
GROUP BY department
HAVING AVG(age) > 18;

-- SQL Clauses: Union
SELECT first_name, last_name FROM students
UNION
SELECT course_name, department FROM courses;

-- SQL Clauses: Distinct
SELECT DISTINCT grade FROM enrollments;

-- SQL Wildcard Characters
SELECT * FROM students WHERE last_name LIKE 'S%';

-- DDL: Drop the database (clean-up)
DROP DATABASE university;
```

Integrating python with SQL





Syntex

```
import pymysql  # pip install pymysql

# Connect to the database
client(object) = pymysql.connect(
                                    host='your_host',
                                    user='your_username',
                                    password='your_password',
                                    database='your_database_name'
                                    )

# Create a cursor object
cursor(object) = client.cursor()

# Create a table
cursor.execute(("""create table Table_name(object) (Colum1 datatype,
                                                    Colum2 datatype,
                                                    Colum3 datatype,
```

```
                                                    Colum4 datatype)""")
client.commit()


# Execute SQL queries
cursor.execute("SELECT * FROM your_table")
connection.commit()

# Fetch data
data = cursor.fetchall()

# Close the connection
connection.close()
```

Sample

```
import [psycopg2 / mysql.connetor]      #pip install [psycopg2 / mysql-connector-python]

client (object) = [psycopg2 / mysql.connetor].connect (host='localhost', user='postgres',
password='*****', database='student', port=5432)

access (object) = client.cursor()

access.execute("""create table student1(
rollno int,
name varchar(50),
bloodgroup varchar(10),
address text,
age int,
department varchar(10),
gender varchar(10),
mobile bigint,
mark int
)""")
client.commit()

access.execute("""insert into student1 values
(14530,'Guru', 'B +','Salem',18,'CSE','Male',984154154,94),
(14880,'Lavanya','AB +','Chennai',34,'EEE','Female',984444044,64),
(13330,'Sankar','AB -','Vilupuram',23,'AUTOMOBILE','Male',983333354,74),
(16530,'Venkat','O +','Covai',25,'ECE','Male',63111154,84),
(12530,'Vithya','B -','Salem',27,'MECH','Female',98000054,44),
(13330,'Raja','A +','Chennai',29,'CIVIL','Male',8844554154,34),
(14460,'Ramya','A -','Madurai',31,'B TECH','Female',78706654,634)""")
client.commit()

access.execute("""select * from student1""")
x=access.fetchall()
for i in x:
    print(i)
```

```
access.execute("""update student_info set mark=25 where address='Chennai'""")
client.commit()

access.execute("""select * from student1""")
x=access.fetchall()
for i in x:
    print(i)

access.execute("""select * from student1 where age>30""")
x=access.fetchall()
for i in x:
    print(i)
```

commit() and **fetchall()** are methods commonly used in database programming, especially when working with Python's Database API (PEP 249). These methods are typically associated with database connections and cursors.

1. The **commit()** method is used to save changes made to the database since the last call to commit(). It essentially commits the current transaction, making the changes permanent in the database.
    - Without the commit() call, changes made with INSERT, UPDATE, or DELETE statements won't be saved permanently.
2. The fetchall() method is used to retrieve all the rows of a query result set as a list of tuples. It fetches all the rows returned by the last executed SELECT statement.

Example

Example 1:

```
import psycopg2

# Connect to the PostgreSQL database
client = psycopg2.connect(
    host='localhost',
    user='postgres',
    password='******',
    database='student',
    port=5432
)

# Create a cursor object for database operations
access = client.cursor()

# DDL: Create a table
access.execute('''
    CREATE TABLE courses (
        course_id SERIAL PRIMARY KEY,
        course_name VARCHAR(50),
        credits INT )''')

# DML: Insert data into the table
access.execute('''
    INSERT INTO courses (course_name, credits)
    VALUES (%s, %s)''', ('Mathematics', 3))

access.execute('''
    INSERT INTO courses (course_name, credits)
    VALUES (%s, %s)''', ('Computer Science', 4))

# DQL: Select data from the table
access.execute('SELECT * FROM courses')
courses_data = access.fetchall()

print("Courses Data:")
for course in courses_data:
    print(course)

# DML: Update a record
access.execute('''
    UPDATE courses
    SET credits = %s
    WHERE course_name = %s
''', (5, 'Computer Science'))

# DQL: Select updated data
access.execute('SELECT * FROM courses')
```

```
updated_courses_data = access.fetchall()

print("\nUpdated Courses Data:")
for course in updated_courses_data:
    print(course)

# DML: Delete a record
access.execute('''
    DELETE FROM courses
    WHERE course_name = %s
''', ('Mathematics',))

# DQL: Select remaining data
access.execute('SELECT * FROM courses')
remaining_courses_data = access.fetchall()

print("\nRemaining Courses Data:")
for course in remaining_courses_data:
    print(course)

# Cleanup and close the connection
client.commit()
access.close()
client.close()
```

Example 2:

```
import psycopg2

# Connect to the PostgreSQL database
client = psycopg2.connect(
    host='localhost',
    user='postgres',
    password='*****',
    database='university',
    port=5432
)

# Create a cursor object for database operations
access = client.cursor()

# DDL: Create tables
access.execute('''
    CREATE TABLE departments (
        department_id SERIAL PRIMARY KEY,
        department_name VARCHAR(50)
    )
''')

access.execute('''
```

```
    CREATE TABLE professors (
        professor_id SERIAL PRIMARY KEY,
        first_name VARCHAR(50),
        last_name VARCHAR(50),
        department_id INT,
        FOREIGN KEY (department_id) REFERENCES departments(department_id)
    )
''')

# DML: Insert data into tables
access.execute('''
    INSERT INTO departments (department_name)
    VALUES (%s)
''', ('Computer Science',))

access.execute('''
    INSERT INTO professors (first_name, last_name, department_id)
    VALUES (%s, %s, %s)
''', ('John', 'Doe', 1))

# DQL: Select data from multiple tables with a join
access.execute('''
    SELECT professors.professor_id, first_name, last_name, department_name
    FROM professors
    JOIN departments ON professors.department_id = departments.department_id
''')
professors_data = access.fetchall()

print("Professors Data:")
for professor in professors_data:
    print(professor)

# Cleanup and close the connection
client.commit()
access.close()
client.close()
```

To migrate data from MongoDB to PostgreSQL

Export Data from MongoDB:
Use a tool or script to export data from MongoDB into a format that can be imported into PostgreSQL. Common formats include JSON or CSV.

Transform Data (if necessary):
Depending on the structure of your data in MongoDB and the schema of your PostgreSQL database, you may need to transform the data to fit the target schema. This might involve restructuring JSON documents or performing data type conversions.

Set Up PostgreSQL:
Ensure that you have PostgreSQL installed and configured properly on your system. Create the necessary tables and indexes to match the structure of the MongoDB data.

Import Data into PostgreSQL:
Use PostgreSQL's built-in tools or utilities like pgloader to import the exported data into PostgreSQL. Ensure that the data is imported into the correct tables and columns.

Here's a more detailed explanation of these steps:

Step 1: Export Data from MongoDB
You can export data from MongoDB using mongoexport utility or by writing a script in a programming language like Python using MongoDB drivers.

Example using mongoexport:

```
mongoexport --db your_database --collection your_collection --out data.json
```

Step 2: Transform Data (if necessary)
If the structure of your data needs to be transformed to fit the PostgreSQL schema, you'll need to write a script to perform this transformation. This could involve parsing the exported JSON file and reshaping the data as needed.

Step 3: Set Up PostgreSQL
Ensure that PostgreSQL is installed and running. You can install it using your package manager or by downloading it from the official PostgreSQL website.

```
# For Ubuntu/Debian
sudo apt-get install postgresql

# For CentOS/RHEL
sudo yum install postgresql-server
```

Once installed, you may need to initialize the database cluster and start the PostgreSQL service.

```
sudo postgresql-setup initdb
sudo systemctl start postgresql
```

Step 4: Import Data into PostgreSQL

You can import data into PostgreSQL using the psql command-line tool, or you can use tools like pgloader for more complex migrations.

Example using psql:

psql -U username -d your_database -c "COPY your_table FROM 'data.json' DELIMITER ',' CSV;"

Example using pgloader:

pgloader mongodb.load

Ensure that the data is imported correctly and verify the integrity of the migrated data.

By following these steps, you should be able to migrate data from MongoDB to PostgreSQL

To migrate a DataFrame to a SQL database using SQLAlchemy

pip install sqlalchemy psycopg2 # for PostgreSQL

from sqlalchemy import create_engine

# Step 1: Create a SQLAlchemy Engine
engine = create_engine('postgresql://username:password@localhost/database_name')

Step 2: Convert DataFrame to SQL Table
df.to_sql('table_name', engine, if_exists='replace', index=False)

Replace 'table_name' with the name you want for your SQL table
Set if_exists to 'replace' if you want to overwrite the table if it already exists
Set index=False to avoid writing DataFrame index as a column in the SQL table

Here's a general approach to preprocess your DataFrame before migrating it to SQL:

1. Flatten the nested structures in the DataFrame, such as dictionaries or lists, into a format that can be easily represented in SQL. For example, you can convert dictionaries into JSON strings.
2. Handle missing or null values appropriately.
3. Drop or transform any columns that cannot be directly mapped to SQL types.

Here's an example of how you can preprocess the DataFrame before migrating it to SQL:

Flatten the 'Comments' column into a JSON string
df['Column_name] = df[' Column_name '].apply(lambda x: json.dumps(x))
df.to_sql('table_name', engine, if_exists='replace', index=False)

Remove multiple columns from the DataFrame
columns_to_drop = ['column1', 'column2', 'column3']
df_without_columns = df.drop(columns=columns_to_drop)

Migrate the modified DataFrame to SQL
df_without_columns.to_sql('Table_name', engine, if_exists='replace', index=False)

Step 1: Migrate main data (excluding 'Comments') to one table
columns_to_exclude = ['Comments'] # Add other columns to exclude if needed
df_main = df.drop(columns=columns_to_exclude)
df_main.to_sql('Videos', engine, if_exists='replace', index=False)

Step 2: Extract 'Comments' data into a separate DataFrame
df_comments = df[['Video_Id', 'Comments']] # Assuming 'Video_Id' is a unique identifier for each video
Replace 'Video_Id' with the appropriate identifier column name
If you have multiple columns related to comments, include them as needed

Step 3: Migrate 'Comments' DataFrame to another table
df_comments.to_sql('VideoComments', engine, if_exists='replace', index=False)

```
# Define the data types for individual columns
dtype_mapping = {
    'Channel_Id': 'VARCHAR(50)',  # Example: VARCHAR with length 50
    'Video_Id': 'VARCHAR(20)',
    'Video_Name': 'TEXT',
    'PublishedAt': 'TIMESTAMP',
    'View_Count': 'INTEGER',
    'Like_Count': 'INTEGER',
    'Comment_Count': 'INTEGER',
    'Duration': 'VARCHAR(20)',  # Example: VARCHAR with length 20
}

# Migrate the DataFrame to SQL, specifying the data types
df.to_sql('Videos', engine, if_exists='replace', index=False, dtype=dtype_mapping)
```