SQL Basics For Beginners

explain sql in details for a beginner level to making him pro in sql

SQL Introduction

Structured Query Language (SQL) is a standardized programming language used for managing and manipulating relational databases. It allows you to perform various operations like querying data, updating records, and managing database structures.

Basics of SQL

1. **Database Basics**:
    - **Database**: A collection of related data organized in tables.
    - **Table**: A collection of rows and columns.
    - **Row (Record)**: A single entry in a table.
    - **Column (Field)**: A single category of data in a table.
2. **SQL Commands**:
    - **Data Definition Language (DDL)**: Commands that define the structure of the database.
        - `` `CREATE` ``: Create a new table or database.
        - `` `ALTER` ``: Modify an existing database object.
        - `` `DROP` ``: Delete an existing database object.
    - **Data Manipulation Language (DML)**: Commands that manipulate data stored in the database.
        - `` `SELECT` ``: Retrieve data from the database.
        - `` `INSERT` ``: Insert new data into a table.
        - `` `UPDATE` ``: Modify existing data in a table.
        - `` `DELETE` ``: Remove data from a table.
    - **Data Control Language (DCL)**: Commands that control access to data in the database.
        - `` `GRANT` ``: Provide user access privileges.
        - `` `REVOKE` ``: Remove user access privileges.
    - **Transaction Control Language (TCL)**: Commands that manage transactions in the database.
        - `` `COMMIT` ``: Save changes made during a transaction.
        - `` `ROLLBACK` ``: Undo changes made during a transaction.
        - `` `SAVEPOINT` ``: Set a point within a transaction to which you can later roll back.

Writing SQL Queries

1. **SELECT Statement**:
    - Retrieve data from one or more tables.

```sql
SELECT column1, column2, ...
FROM table_name;
```

- Example:

```sql
SELECT first_name, last_name
FROM employees;
```

2. **WHERE Clause**:
   - Filter records based on specific conditions.

```sql
SELECT column1, column2, ...
FROM table_name
WHERE condition;
```

   - Example:

```sql
SELECT first_name, last_name
FROM employees
WHERE department = 'Sales';
```

3. **INSERT INTO Statement**:
   - Add new records to a table.

```sql
INSERT INTO table_name (column1, column2, ...)
VALUES (value1, value2, ...);
```

   - Example:

```sql
INSERT INTO employees (first_name, last_name, department)
VALUES ('John', 'Doe', 'Sales');
```

4. **UPDATE Statement**:
   - Modify existing records in a table.

```sql
UPDATE table_name
SET column1 = value1, column2 = value2, ...
WHERE condition;
```

   - Example:

```sql
UPDATE employees
SET department = 'Marketing'
WHERE last_name = 'Doe';
```

5. **DELETE Statement**:
   - Remove records from a table.

```sql
DELETE FROM table_name
WHERE condition;
```

   - Example:

```sql
DELETE FROM employees
WHERE last_name = 'Doe';
```

Advanced SQL Concepts

1. **JOINs**:
   - Combine rows from two or more tables based on related columns.
   - **INNER JOIN**: Select records with matching values in both tables.

```sql
SELECT column1, column2, ...
FROM table1
INNER JOIN table2
ON table1.common_column = table2.common_column;
```

   - **LEFT JOIN**: Select all records from the left table and matched records from the right table.

```sql
SELECT column1, column2, ...
FROM table1
```

```
LEFT JOIN table2
ON table1.common_column = table2.common_column;
```

   - **RIGHT JOIN**: Select all records from the right table and matched records from the left table.

```sql
SELECT column1, column2, ...
FROM table1
RIGHT JOIN table2
ON table1.common_column = table2.common_column;
```

2. **Aggregate Functions**:
   - Perform calculations on a set of values and return a single value.
   - Common functions: `` `COUNT()` ``, `` `SUM()` ``, `` `AVG()` ``, `` `MIN()` ``, `` `MAX()` ``.

```sql
SELECT COUNT(*)
FROM employees;
```

3. **GROUP BY Clause**:
   - Group rows that have the same values into summary rows.

```sql
SELECT column1, COUNT(*)
FROM table_name
GROUP BY column1;
```

4. **HAVING Clause**:
   - Filter groups based on a specified condition.

```sql
SELECT column1, COUNT(*)
FROM table_name
GROUP BY column1
HAVING COUNT(*) > 1;
```

5. **Subqueries**:
   - A query within another query.

```sql
SELECT column1
FROM table_name
WHERE column2 = (SELECT column2 FROM another_table WHERE condition);
```

Tips for Becoming Proficient in SQL

1. **Practice Regularly**:
    - Use online platforms like LeetCode, HackerRank, and SQLZoo to practice SQL queries.
2. **Work on Real Projects**:
    - Build and manage your own databases to gain practical experience.
3. **Learn from Resources**:
    - Follow tutorials on YouTube, Coursera, and Udemy. Muhammad Asif prefers learning from YouTube channels in Hindi or Urdu, particularly from Pakistan or India.
4. **Understand Database Design**:
    - Learn about normalization, primary keys, foreign keys, and indexing to optimize database performance.
5. **Explore Advanced Topics**:
    - Dive into stored procedures, triggers, views, and database security as you become more comfortable with SQL.

By following this structured approach and consistently practicing, you can progress from a beginner to a proficient SQL user.

import csv files

To import CSV files into a SQL database, you can use several methods depending on the database management system (DBMS) you are using. Here's a general guide to importing CSV files into popular SQL databases like MySQL, PostgreSQL, and SQLite.

Importing CSV Files into MySQL

1. **Using MySQL Workbench**:
    - Open MySQL Workbench and connect to your database.
    - Select your database and right-click on the Tables node.
    - Select "Table Data Import Wizard".
    - Follow the wizard steps to import the CSV file.
2. **Using the `LOAD DATA INFILE` Statement**:

```sql
LOAD DATA INFILE 'path/to/your/file.csv'
INTO TABLE table_name
FIELDS TERMINATED BY ','
ENCLOSED BY '"'
LINES TERMINATED BY '\n'
IGNORE 1 ROWS;
```

    - Ensure that the MySQL server has the required file permissions to access the CSV file.
    - You can specify the path to the CSV file, the table name, and the format of the CSV file.

Importing CSV Files into PostgreSQL

1. **Using pgAdmin**:
    - Open pgAdmin and connect to your database.
    - Right-click on the table you want to import data into and select "Import/Export".
    - Select the "Import" tab, choose the CSV file, and configure the import settings.
2. **Using the `COPY` Command**:

```sql
COPY table_name(column1, column2, ...)
FROM 'path/to/your/file.csv'
DELIMITER ','
CSV HEADER;
```

    - Make sure the PostgreSQL server has permission to read the CSV file.
    - Specify the path to the CSV file, the table name, and the format.

Importing CSV Files into SQLite

1. **Using SQLite CLI**:

```sql
sqlite> .mode csv
sqlite> .import /path/to/your/file.csv table_name
```

    - Open the SQLite command-line interface and use the `.mode csv` command to specify the format.
    - Use the `.import` command to import the CSV file into the specified table.

Python Script for Importing CSV Files

You can also use Python with libraries like `pandas` and `sqlalchemy` to import CSV files into any SQL database. Here's an example for MySQL:

1. **Install required libraries**:

```bash
pip install pandas sqlalchemy pymysql
```

2. **Python Script**:

```python
import pandas as pd
from sqlalchemy import create_engine

# Read the CSV file into a DataFrame
df = pd.read_csv('path/to/your/file.csv')
```

```
# Create a SQLAlchemy engine
engine = create_engine('mysql+pymysql://username:password@host:port/database')

# Write the DataFrame to a SQL table
df.to_sql('table_name', con=engine, if_exists='replace', index=False)
```

  - Replace `'path/to/your/file.csv'` with the path to your CSV file.
  - Replace `'mysql+pymysql://username:password@host:port/database'` with your MySQL connection string.
  - Replace `'table_name'` with the name of the table where you want to import the data.

Conclusion

These are some of the common methods to import CSV files into SQL databases. Choose the method that best fits your needs and the DBMS you are using. Practicing these methods will help you become proficient in handling data imports in SQL.

where,and,or,not,like,order by,in,not in,between

SQL Clauses and Operators

In SQL, clauses and operators like `WHERE`, `AND`, `OR`, `NOT`, `LIKE`, `ORDER BY`, `IN`, `NOT IN`, and `BETWEEN` are used to filter and sort data in a database. Here's a detailed explanation of each, with examples to illustrate their use:

1. `WHERE` Clause

The `WHERE` clause is used to filter records that meet a certain condition.

```sql
SELECT * FROM employees
WHERE department = 'Sales';
```

2. `AND` Operator

The `AND` operator is used to combine multiple conditions. All conditions must be true for the records to be selected.

```sql
SELECT * FROM employees
```

```sql
WHERE department = 'Sales' AND age > 30;
```

3. `OR` Operator

The `OR` operator is used to combine multiple conditions. At least one condition must be true for the records to be selected.

```sql
SELECT * FROM employees
WHERE department = 'Sales' OR department = 'Marketing';
```

4. `NOT` Operator

The `NOT` operator is used to negate a condition. Records that do not meet the condition are selected.

```sql
SELECT * FROM employees
WHERE NOT department = 'Sales';
```

5. `LIKE` Operator

The `LIKE` operator is used to search for a specified pattern in a column. Often used with wildcards `%` (zero or more characters) and `_` (a single character).

```sql
SELECT * FROM employees
WHERE first_name LIKE 'J%';
```

6. `ORDER BY` Clause

The `ORDER BY` clause is used to sort the result set in ascending or descending order. By default, it sorts in ascending order (`ASC`). Use `DESC` for descending order.

```sql
SELECT * FROM employees
ORDER BY last_name ASC;
```

7. `IN` Operator

The `IN` operator is used to specify multiple possible values for a column.

```sql
SELECT * FROM employees
WHERE department IN ('Sales', 'Marketing', 'HR');
```

8. `NOT IN` Operator

The `NOT IN` operator is used to exclude multiple possible values for a column.

```sql
SELECT * FROM employees
WHERE department NOT IN ('Sales', 'Marketing');
```

9. `BETWEEN` Operator

The `BETWEEN` operator is used to select values within a given range. The values can be numbers, text, or dates.

```sql
SELECT * FROM employees
WHERE age BETWEEN 30 AND 40;
```

Combining Clauses and Operators

You can combine these clauses and operators to build complex queries. For example:

```sql
SELECT * FROM employees
WHERE (department = 'Sales' OR department = 'Marketing')
AND age BETWEEN 25 AND 35
AND first_name LIKE 'J%'
ORDER BY last_name DESC;
```

Practical Examples

1. **Filter by Department and Age**:

   ```sql
   SELECT * FROM employees
   WHERE department = 'HR' AND age > 25;
   ```

2. **Filter by Multiple Departments**:

```sql
SELECT * FROM employees
WHERE department IN ('IT', 'Finance', 'Operations');
```

3. **Exclude Certain Departments**:

```sql
SELECT * FROM employees
WHERE department NOT IN ('HR', 'Admin');
```

4. **Search for Employees with Specific Pattern in Names**:

```sql
SELECT * FROM employees
WHERE first_name LIKE '_a%';
```

5. **Filter by Age Range and Sort by Last Name**:

```sql
SELECT * FROM employees
WHERE age BETWEEN 30 AND 50
ORDER BY last_name;
```

By mastering these clauses and operators, you can write powerful SQL queries to retrieve and manipulate data effectively.

limit

The `LIMIT` clause is used in SQL to specify the number of records to return from a query. It is often used with the `SELECT` statement to restrict the number of rows in the result set.

Basic Usage of `LIMIT`

1. **Retrieve a Specific Number of Rows**:

```sql
SELECT * FROM employees
```

```
LIMIT 10;
```

This query returns the first 10 rows from the `employees` table.

2. **Retrieve a Range of Rows Using `OFFSET`**:
   The `OFFSET` keyword is used to specify the starting point of the rows to return. When combined with `LIMIT`, it allows you to implement pagination.

   ```sql
   SELECT * FROM employees
   LIMIT 10 OFFSET 20;
   ```

   This query returns 10 rows starting from the 21st row in the result set (rows 21 to 30).

Combining `LIMIT` with Other Clauses

You can combine the `LIMIT` clause with other SQL clauses like `ORDER BY`, `WHERE`, etc., to create more complex queries.

1. **Order and Limit Results**:

   ```sql
   SELECT * FROM employees
   ORDER BY last_name ASC
   LIMIT 5;
   ```

   This query returns the first 5 rows from the `employees` table, ordered by `last_name` in ascending order.
2. **Filter, Order, and Limit Results**:

   ```sql
   SELECT * FROM employees
   WHERE department = 'Sales'
   ORDER BY hire_date DESC
   LIMIT 3;
   ```

   This query returns the latest 3 employees hired in the Sales department.

Practical Examples

1. **Top N Records**:

   ```sql
   SELECT * FROM products
   ORDER BY price DESC
   ```

```
LIMIT 5;
```

This query returns the 5 most expensive products.

2. **Pagination**:
   Suppose you want to display records 11 to 20 on the second page of a web application:

   ```sql
   SELECT * FROM products
   ORDER BY product_id
   LIMIT 10 OFFSET 10;
   ```

   This query returns rows 11 to 20 from the `` `products` `` table.
3. **Random Sampling**:
   To get a random sample of 10 rows (supported in some SQL databases like MySQL):

   ```sql
   SELECT * FROM employees
   ORDER BY RAND()
   LIMIT 10;
   ```

   This query returns 10 random rows from the `` `employees` `` table.

SQL Syntax Variations

Different SQL databases might have slight variations in the syntax or support for `` `LIMIT` `` and `` `OFFSET` ``.

- **MySQL, SQLite**:

  ```sql
  SELECT * FROM table_name
  LIMIT number_of_rows OFFSET starting_point;
  ```

- **PostgreSQL**:

  ```sql
  SELECT * FROM table_name
  LIMIT number_of_rows OFFSET starting_point;
  ```

- **SQL Server**: Uses `` `TOP` `` instead of `` `LIMIT` ``, and the syntax is slightly different.

  ```sql
  SELECT TOP number_of_rows * FROM table_name;
  ```

- **Oracle**: Uses `FETCH FIRST` and `ROWNUM`.

  ```sql
  SELECT * FROM table_name
  FETCH FIRST number_of_rows ROWS ONLY;
  ```

Conclusion

The `LIMIT` clause is a powerful tool to control the number of rows returned by your queries, making it essential for efficient data retrieval and pagination in applications.

String Functions
- Data Aggregation, Numeric Functions
- Date Functions, Case Operator
- Group By, Having Clause

String Functions in SQL

String functions are used to manipulate and query text strings in SQL. Here are some common string functions:

1. **CONCAT**: Concatenates two or more strings.

   ```sql
   SELECT CONCAT(first_name, ' ', last_name) AS full_name
   FROM employees;
   ```

2. **SUBSTRING**: Extracts a substring from a string.

   ```sql
   SELECT SUBSTRING(first_name, 1, 3) AS short_name
   FROM employees;
   ```

3. **UPPER**: Converts a string to uppercase.

   ```sql
   SELECT UPPER(first_name) AS upper_name
   ```

```sql
FROM employees;
```

4. **LOWER**: Converts a string to lowercase.

sql

```sql
SELECT LOWER(first_name) AS lower_name
FROM employees;
```

5. **LENGTH**: Returns the length of a string.

sql

```sql
SELECT LENGTH(first_name) AS name_length
FROM employees;
```

6. **TRIM**: Removes leading and trailing spaces from a string.

sql

```sql
SELECT TRIM(first_name) AS trimmed_name
FROM employees;
```

Data Aggregation Functions

Aggregation functions perform calculations on a set of values and return a single value. Common functions include:

1. **COUNT**: Counts the number of rows.

sql

```sql
SELECT COUNT(*) AS total_employees
FROM employees;
```

2. **SUM**: Sums up the values.

sql

```sql
SELECT SUM(salary) AS total_salary
FROM employees;
```

3. **AVG**: Calculates the average value.

sql

```sql
SELECT AVG(salary) AS average_salary
```

```
FROM employees;
```

4. **MIN**: Returns the minimum value.

```sql
SELECT MIN(salary) AS minimum_salary
FROM employees;
```

5. **MAX**: Returns the maximum value.

```sql
SELECT MAX(salary) AS maximum_salary
FROM employees;
```

Numeric Functions

Numeric functions perform operations on numeric data types.

1. **ABS**: Returns the absolute value.

```sql
SELECT ABS(-10) AS absolute_value;
```

2. **CEIL/CEILING**: Returns the smallest integer greater than or equal to a number.

```sql
SELECT CEIL(4.2) AS ceiling_value;
```

3. **FLOOR**: Returns the largest integer less than or equal to a number.

```sql
SELECT FLOOR(4.8) AS floor_value;
```

4. **ROUND**: Rounds a number to a specified number of decimal places.

```sql
SELECT ROUND(123.456, 2) AS rounded_value;
```

Date Functions

Date functions are used to manipulate date and time values.

1. **CURRENT_DATE**: Returns the current date.

   ```sql
   SELECT CURRENT_DATE AS today_date;
   ```

2. **CURRENT_TIME**: Returns the current time.

   ```sql
   SELECT CURRENT_TIME AS current_time;
   ```

3. **CURRENT_TIMESTAMP**: Returns the current date and time.

   ```sql
   SELECT CURRENT_TIMESTAMP AS current_timestamp;
   ```

4. **DATEADD**: Adds a specified number of days to a date.

   ```sql
   SELECT DATEADD(day, 7, '2024-07-11') AS next_week;
   ```

5. **DATEDIFF**: Returns the difference between two dates.

   ```sql
   SELECT DATEDIFF(day, '2024-07-11', '2024-08-11') AS date_difference;
   ```

6. **FORMAT**: Formats a date value.

   ```sql
   SELECT FORMAT('2024-07-11', 'yyyy-MM-dd') AS formatted_date;
   ```

CASE Operator

The `CASE` statement is used to create conditional queries.

```sql
SELECT first_name,
       salary,
       CASE
           WHEN salary < 50000 THEN 'Low'
           WHEN salary BETWEEN 50000 AND 100000 THEN 'Medium'
           ELSE 'High'
       END AS salary_level
FROM employees;
```

GROUP BY Clause

The `GROUP BY` clause is used to group rows that have the same values in specified columns into summary rows, such as counts, sums, averages, etc.

```sql
SELECT department, COUNT(*) AS employee_count
FROM employees
GROUP BY department;
```

HAVING Clause

The `HAVING` clause is used to filter groups based on a condition. It is similar to the `WHERE` clause but is used for groups.

```sql
SELECT department, COUNT(*) AS employee_count
FROM employees
GROUP BY department
HAVING COUNT(*) > 5;
```

Practical Example Combining Clauses and Functions

```sql
SELECT department,
       COUNT(*) AS employee_count,
       AVG(salary) AS average_salary,
       CASE
           WHEN AVG(salary) < 50000 THEN 'Low'
           WHEN AVG(salary) BETWEEN 50000 AND 100000 THEN 'Medium'
           ELSE 'High'
       END AS salary_level
FROM employees
GROUP BY department
HAVING COUNT(*) > 5
ORDER BY average_salary DESC
LIMIT 10;
```

This query groups employees by department, calculates the number of employees and average salary per department, classifies the salary level, filters groups with more than 5 employees, orders the result by average salary in descending order, and limits the output to the top 10 departments.

Joins, Set Operators, Subqueries, Views
- Stored Procedure, Window Functions

Joins

Joins are used in SQL to combine rows from two or more tables based on a related column.

1. **INNER JOIN**: Returns only the rows that have matching values in both tables.

```sql
SELECT employees.first_name, employees.last_name, departments.department_name
FROM employees
INNER JOIN departments ON employees.department_id = departments.department_id;
```

2. **LEFT JOIN** (or **LEFT OUTER JOIN**): Returns all rows from the left table and the matched rows from the right table. Unmatched rows from the right table are returned as NULL.

```sql
SELECT employees.first_name, employees.last_name, departments.department_name
FROM employees
LEFT JOIN departments ON employees.department_id = departments.department_id;
```

3. **RIGHT JOIN** (or **RIGHT OUTER JOIN**): Returns all rows from the right table and the matched rows from the left table. Unmatched rows from the left table are returned as NULL.

```sql
SELECT employees.first_name, employees.last_name, departments.department_name
FROM employees
RIGHT JOIN departments ON employees.department_id = departments.department_id;
```

4. **FULL JOIN** (or **FULL OUTER JOIN**): Returns rows when there is a match in one of the tables. Unmatched rows will also be included with NULL values for the missing side.

```sql
SELECT employees.first_name, employees.last_name, departments.department_name
FROM employees
```

```
FULL OUTER JOIN departments ON employees.department_id = departments.department_id;
```

5. **CROSS JOIN**: Returns the Cartesian product of both tables, i.e., each row from the first table is combined with all rows from the second table.

sql

```
SELECT employees.first_name, departments.department_name
FROM employees
CROSS JOIN departments;
```

Set Operators

Set operators are used to combine the results of two or more queries.

1. **UNION**: Combines the results of two queries and removes duplicate rows.

sql

```
SELECT first_name, last_name FROM employees
UNION
SELECT first_name, last_name FROM managers;
```

2. **UNION ALL**: Combines the results of two queries including duplicates.

sql

```
SELECT first_name, last_name FROM employees
UNION ALL
SELECT first_name, last_name FROM managers;
```

3. **INTERSECT**: Returns only the rows that are present in both queries.

sql

```
SELECT first_name, last_name FROM employees
INTERSECT
SELECT first_name, last_name FROM managers;
```

4. **EXCEPT** (or **MINUS** in some databases like Oracle): Returns rows from the first query that are not present in the second query.

sql

```
SELECT first_name, last_name FROM employees
EXCEPT
SELECT first_name, last_name FROM managers;
```

Subqueries

A subquery is a query within another query.

1. **Subquery in SELECT**:

```sql
SELECT first_name, last_name,
       (SELECT department_name FROM departments WHERE departments.department_id =
employees.department_id) AS department_name
FROM employees;
```

2. **Subquery in WHERE**:

```sql
SELECT first_name, last_name
FROM employees
WHERE department_id = (SELECT department_id FROM departments WHERE department_name =
'Sales');
```

3. **Subquery in FROM (also called Inline View)**:

```sql
SELECT avg_salary
FROM (SELECT AVG(salary) AS avg_salary FROM employees) AS avg_salaries;
```

Views

A view is a virtual table based on the result-set of a SQL query.

1. **Create a View**:

```sql
CREATE VIEW employee_details AS
SELECT first_name, last_name, department_name
FROM employees
JOIN departments ON employees.department_id = departments.department_id;
```

2. **Query a View**:

```sql
SELECT * FROM employee_details;
```

3. **Update a View**:

```sql
CREATE OR REPLACE VIEW employee_details AS
SELECT first_name, last_name, department_name, salary
FROM employees
JOIN departments ON employees.department_id = departments.department_id;
```

Stored Procedures

A stored procedure is a prepared SQL code that you can save and reuse.

1. **Create a Stored Procedure**:

```sql
CREATE PROCEDURE GetEmployeeDetails()
BEGIN
    SELECT first_name, last_name, department_name
    FROM employees
    JOIN departments ON employees.department_id = departments.department_id;
END;
```

2. **Execute a Stored Procedure**:

```sql
CALL GetEmployeeDetails();
```

3. **Stored Procedure with Parameters**:

```sql
CREATE PROCEDURE GetEmployeeByDepartment(dept_name VARCHAR(50))
BEGIN
    SELECT first_name, last_name
    FROM employees
    JOIN departments ON employees.department_id = departments.department_id
    WHERE departments.department_name = dept_name;
END;
```

4. **Execute a Stored Procedure with Parameters**:

```sql
CALL GetEmployeeByDepartment('Sales');
```

Window Functions

Window functions perform calculations across a set of table rows related to the current row.

1. **ROW_NUMBER()**: Assigns a unique number to each row within the partition of a result set.

```sql
SELECT first_name, last_name,
       ROW_NUMBER() OVER (PARTITION BY department_id ORDER BY salary DESC) AS row_num
FROM employees;
```

2. **RANK()**: Assigns a rank to each row within the partition of a result set.

```sql
SELECT first_name, last_name,
       RANK() OVER (PARTITION BY department_id ORDER BY salary DESC) AS rank
FROM employees;
```

3. **DENSE_RANK()**: Similar to `RANK()`, but ranks without gaps in the ranking values.

```sql
SELECT first_name, last_name,
       DENSE_RANK() OVER (PARTITION BY department_id ORDER BY salary DESC) AS
dense_rank
FROM employees;
```

4. **NTILE()**: Distributes the rows in an ordered partition into a specified number of approximately equal groups.

```sql
SELECT first_name, last_name,
       NTILE(4) OVER (PARTITION BY department_id ORDER BY salary DESC) AS quartile
FROM employees;
```

5. **LAG() and LEAD()**: Accesses data from a subsequent or preceding row in the same result set without the use of a self-join.

```sql
SELECT first_name, last_name, salary,
       LAG(salary, 1) OVER (ORDER BY salary) AS prev_salary,
       LEAD(salary, 1) OVER (ORDER BY salary) AS next_salary
FROM employees;
```

6. **SUM(), AVG(), MIN(), MAX() as Window Functions**:

```sql
SELECT first_name, last_name, salary,
       SUM(salary) OVER (PARTITION BY department_id) AS dept_total_salary,
        AVG(salary) OVER (PARTITION BY department_id) AS dept_avg_salary
FROM employees;
```

Conclusion

Understanding and effectively using joins, set operators, subqueries, views, stored procedures, and window functions can greatly enhance your ability to query and manipulate data in SQL. By practicing these concepts, you'll be able to write more efficient and powerful SQL queries.

ChatGPT can make mistakes. Check important info.