**SQL Reference Guide**

SQL stands for Structured Query Language. It is a programming language used to manage and manipulate relational databases. SQL allows users to interact with a database, perform various operations, and retrieve, insert, update, or delete data.

The primary functions of SQL are:

1. Data Retrieval: SQL can be used to query the database and retrieve specific data based on various conditions and criteria.
2. Data Insertion: SQL allows users to add new records to the database.
3. Data Modification: SQL can be used to update existing records in the database, altering the data as needed.
4. Data Deletion: SQL enables users to remove records from the database that are no longer required.

SQL operates on relational databases, where data is organized into tables consisting of rows and columns. Each row represents a single record, and each column represents an attribute of that record.

There are various database management systems (DBMS) that support SQL, such as MySQL, PostgreSQL, SQLite, Oracle, Microsoft SQL Server, etc. These DBMSs handle the storage and retrieval of data and provide an interface through which users can execute SQL queries and commands.

SQL is a vital skill for developers, data analysts, and anyone working with databases, as it provides a standardized way to interact with and manage the data efficiently.

To create a database, access it, and create a table within that database, you will need to use specific SQL commands. The exact commands may vary slightly depending on the database management system (DBMS) you are using. Below are the general steps to achieve this in SQL:

1. Create a Database:
   * The command for creating a database varies between different DBMSs. For example, in MySQL, you can use the following command:

CREATE DATABASE your\_database\_name;

* + Replace **your\_database\_name** with the desired name for your database.

1. Access the Database:
   * After creating the database, you need to connect to it. Again, the method of connecting can differ based on the DBMS you are using.
   * In MySQL, you can connect to the database using the following command:

USE your\_database\_name;

* + This command tells MySQL to start using the specified database.

1. Create a Table:
   * Now that you have accessed the database, you can create a table within it. To create a table, use the **CREATE TABLE** command:

CREATE TABLE your\_table\_name ( column1 datatype1 constraints, column2 datatype2 constraints, ...);

* + Replace **your\_table\_name** with the desired name for your table. Define the columns of the table along with their data types and any constraints, such as primary key, foreign key, not null, etc.
  + Example (MySQL):

CREATE TABLE employees ( employee\_id INT PRIMARY KEY, first\_name VARCHAR(50), last\_name VARCHAR(50), hire\_date DATE );

* + This example creates a table named "employees" with four columns: employee\_id, first\_name, last\_name, and hire\_date.

To insert data into a table in SQL, you can use the **INSERT INTO** statement. The syntax for the **INSERT INTO** statement is as follows:

INSERT INTO table\_name (column1, column2, column3, ...) VALUES (value1, value2, value3, ...);

Here's a step-by-step explanation:

1. Identify the Table:
   * Replace **table\_name** with the name of the table into which you want to insert data.
2. Specify Columns:
   * If you want to insert data into specific columns, list those column names within parentheses after the table name. If you want to insert data into all columns, you can omit this part.
3. Provide Values:
   * After specifying the columns (if necessary), use the **VALUES** keyword, followed by a list of values that correspond to the columns. Each value should match the data type of the respective column.

Let's see an example of how to insert data into a table:

*Suppose we have a table named "employees" with the following columns: employee\_id, first\_name, last\_name, and hire\_date.*

*Inserting a single row of data INSERT INTO employees (employee\_id, first\_name, last\_name, hire\_date) VALUES (1, 'John', 'Doe', '2023-07-26'); -- Inserting multiple rows of data in a single query*

*INSERT INTO employees (employee\_id, first\_name, last\_name, hire\_date) VALUES (2, 'Jane', 'Smith', '2023-07-27'), (3, 'Mike', 'Johnson', '2023-07-28');*

In the above example, we first insert a single row of data into the "employees" table, and then we insert multiple rows in a single query.

Make sure that the order of the values in the **VALUES** clause matches the order of the columns specified or the order in which they appear in the table definition (if not explicitly specified).

**SQL Data Types**

SQL data types are used to specify the type of data that can be stored in a column of a table. Here are some common SQL data types along with one example:

1. INTEGER: Used for storing whole numbers (positive or negative) without a fractional component. Example: **age INTEGER**
2. VARCHAR(n): Variable-length character string with a maximum length of 'n' characters. Example: **name VARCHAR(50)**
3. DATE: Used for storing dates in the format 'YYYY-MM-DD'. Example: **birth\_date DATE**
4. DECIMAL(p, s): Fixed-point number with 'p' total digits and 's' digits after the decimal point. Example: **price DECIMAL(10, 2)**
5. BOOLEAN: Represents a boolean value, either true or false. Example: **is\_active BOOLEAN**
6. CHAR(n): Fixed-length character string with a length of 'n'. Example: **country\_code CHAR(3)**
7. FLOAT: Used for storing floating-point numbers. Example: **temperature FLOAT**
8. TIMESTAMP: Represents a combination of date and time. Example: **created\_at TIMESTAMP**
9. BLOB: Used for storing binary data like images or files. Example: **profile\_picture BLOB**
10. ENUM: Represents a predefined list of values that a column can take. Example: **gender ENUM('Male', 'Female', 'Other')**

**SQL Commands**

SQL commands are used to interact with a database, perform operations, and manage data. Here are some essential SQL commands:

1. SELECT: Used to retrieve data from a database. Example: **SELECT column1, column2 FROM table\_name WHERE condition;**
2. INSERT INTO: Used to add new records (rows) to a table. Example: **INSERT INTO table\_name (column1, column2) VALUES (value1, value2);**
3. UPDATE: Used to modify existing records in a table. Example: **UPDATE table\_name SET column1 = value1, column2 = value2 WHERE condition;**
4. DELETE FROM: Used to remove records from a table. Example: **DELETE FROM table\_name WHERE condition;**
5. CREATE TABLE: Used to create a new table in the database. Example: **CREATE TABLE table\_name (column1 datatype1, column2 datatype2, ...);**
6. DROP TABLE: Used to delete an existing table from the database. Example: **DROP TABLE table\_name;**
7. ALTER TABLE: Used to modify the structure of an existing table. Example: **ALTER TABLE table\_name ADD column datatype;**
8. CREATE DATABASE: Used to create a new database. Example: **CREATE DATABASE database\_name;**
9. DROP DATABASE: Used to delete an existing database. Example: **DROP DATABASE database\_name;**
10. GRANT: Used to give specific privileges to a user or role. Example: **GRANT SELECT, INSERT ON table\_name TO user\_name;**
11. REVOKE: Used to revoke previously granted privileges from a user or role. Example: **REVOKE SELECT, INSERT ON table\_name FROM user\_name;**
12. JOIN: Used to combine rows from two or more tables based on a related column. Example: **SELECT column1, column2 FROM table1 INNER JOIN table2 ON table1.column = table2.column;**
13. ORDER BY: Used to sort the result of a SELECT query. Example: **SELECT column1, column2 FROM table\_name ORDER BY column1 DESC;**
14. GROUP BY: Used to group rows based on the values in a specific column. Example: **SELECT column, COUNT(\*) FROM table\_name GROUP BY column;**
15. HAVING: Used with GROUP BY to filter the grouped results. Example: **SELECT column, COUNT(\*) FROM table\_name GROUP BY column HAVING COUNT(\*) > 2;**

**SQL Group by**

In SQL, the **GROUP BY** clause is used in combination with aggregate functions to group rows based on the values in one or more columns. It allows you to perform calculations and summarize data on groups of rows rather than individual rows. The **GROUP BY** clause is commonly used with aggregate functions like **COUNT**, **SUM**, **AVG**, **MIN**, and **MAX** to produce meaningful results for each group.

Syntax of **GROUP BY**:

*SELECT column1, column2, aggregate\_function(column3) FROM table\_name GROUP BY column1, column2;*

Explanation:

* The **SELECT** statement retrieves data from the specified columns of the table.
* The **FROM** clause specifies the table from which to retrieve the data.
* The **GROUP BY** clause comes after the **FROM** clause and identifies the columns that will be used to group the data.
* The **aggregate\_function** is applied to a specific column (column3 in the example) to calculate a result for each group.
* The result will be grouped based on the distinct values in the columns specified in the **GROUP BY** clause.

Example:

Let's assume we have a table named "orders" with the following columns: **order\_id**, **customer\_id**, and **order\_amount**. We want to calculate the total order amount for each customer.

*SELECT customer\_id, SUM(order\_amount) AS total\_amount FROM orders GROUP BY customer\_id;*

In this example, the data will be grouped based on the **customer\_id**, and the **SUM** function will calculate the total order amount for each customer. The result will display the **customer\_id** along with the corresponding **total\_amount** for each group.

**What are aggregate functions?**

Aggregate functions are SQL functions that operate on a set of rows and return a single value as a result. They are used with the **GROUP BY** clause to perform calculations on groups of rows rather than individual rows. Aggregate functions are commonly used to summarize data and derive meaningful insights from large datasets. Here are some of the most commonly used aggregate functions:

1. COUNT(): Returns the number of rows in a group or the total number of rows in a table when used without the **GROUP BY** clause.
2. SUM(): Calculates the sum of the values in a numeric column.
3. AVG(): Calculates the average (mean) of the values in a numeric column.
4. MIN(): Returns the minimum value in a column.
5. MAX(): Returns the maximum value in a column.

These functions are often used in combination with the **GROUP BY** clause to perform calculations on subsets of data. When using aggregate functions, the **GROUP BY** clause is required to define the groups based on which the calculations will be made.

Example:

Let's use the "orders" table again to demonstrate how aggregate functions work:

| ***order\_id*** | ***customer\_id*** | ***order\_amount*** |
| --- | --- | --- |
| *1* | *101* | *50* |
| *2* | *101* | *30* |
| *3* | *102* | *25* |
| *4* | *103* | *40* |
| *5* | *103* | *20* |

To find the total order amount for each customer, we can use the **SUM()** function with the **GROUP BY** clause:

*SELECT customer\_id, SUM(order\_amount) AS total\_amount FROM orders GROUP BY customer\_id;*

*Result:*

| ***customer\_id*** | ***total\_amount*** |
| --- | --- |
| *101* | *80* |
| *102* | *25* |
| *103* | *60* |

In this example, the **SUM()** function calculates the total order amount for each customer group defined by the **customer\_id**, giving us the total amount spent by each customer.

**Keys in SQL**

In SQL, keys are used to establish relationships between tables and ensure data integrity. They play a crucial role in defining the structure and enforcing constraints within a relational database. There are several types of keys in SQL:

1. Primary Key (PK):
   * A primary key uniquely identifies each record (row) in a table.
   * Each table can have only one primary key.
   * It ensures that the values in the key column are unique and cannot be NULL.
   * Used to establish relationships between tables as a foreign key.
   * Example:

*CREATE TABLE employees ( employee\_id INT PRIMARY KEY, first\_name VARCHAR(50), last\_name VARCHAR(50), ... );*

1. Foreign Key (FK):
   * A foreign key is a column or a set of columns in a table that refers to the primary key of another table.
   * It establishes a relationship between two tables, representing a parent-child relationship.
   * It ensures referential integrity, meaning that values in the foreign key column must match values in the primary key column of the referenced table or be NULL.
   * Example:

*CREATE TABLE orders ( order\_id INT PRIMARY KEY, customer\_id INT, order\_date DATE, ... FOREIGN KEY (customer\_id) REFERENCES customers(customer\_id) );*

1. Unique Key:
   * A unique key ensures that the values in the key column are unique, but unlike the primary key, it can have NULL values (usually limited to one NULL value).
   * A table can have multiple unique keys, but only one primary key.
   * Example:

*CREATE TABLE products ( product\_id INT UNIQUE, product\_name VARCHAR(100), ... );*

1. Candidate Key:
   * A candidate key is a column or set of columns that can be used as a primary key.
   * It satisfies the requirements of a primary key (uniqueness and non-nullability).
   * A table can have multiple candidate keys, but only one is chosen as the primary key.
   * Example:

*CREATE TABLE students ( student\_id INT, student\_email VARCHAR(100) UNIQUE, ... );*

1. Composite Key:
   * A composite key is a primary key that consists of two or more columns.
   * Together, these columns uniquely identify each record in the table.
   * Example:

*CREATE TABLE orders ( order\_id INT, product\_id INT, PRIMARY KEY (order\_id, product\_id) );*

**Joins In SQL**

In SQL, joins are used to combine rows from two or more tables based on a related column. Joins allow you to retrieve data from multiple tables in a single query and create a result set that includes information from all the joined tables. They are fundamental for working with relational databases, where data is often distributed across multiple tables that are related to each other through common columns.

There are different types of joins in SQL:

1. INNER JOIN:
   * Returns only the rows that have matching values in both tables based on the specified condition.
   * Syntax:

*SELECT columns FROM table1 INNER JOIN table2 ON table1.column = table2.column;*

1. LEFT JOIN (or LEFT OUTER JOIN):
   * Returns all the rows from the left table and the matching rows from the right table. If there is no match, it returns NULL for the columns from the right table.
   * Syntax:

*SELECT columns FROM table1 LEFT JOIN table2 ON table1.column = table2.column;*

1. RIGHT JOIN (or RIGHT OUTER JOIN):
   * Returns all the rows from the right table and the matching rows from the left table. If there is no match, it returns NULL for the columns from the left table.
   * Syntax:

*SELECT columns FROM table1 RIGHT JOIN table2 ON table1.column = table2.column;*

1. FULL JOIN (or FULL OUTER JOIN):
   * Returns all the rows from both tables, including the unmatched rows. If there is no match, it returns NULL for the columns from the non-matching table.
   * Syntax:

*SELECT columns FROM table1 FULL JOIN table2 ON table1.column = table2.column;*

1. CROSS JOIN:
   * Returns the Cartesian product of the two tables, i.e., all possible combinations of rows between the tables.
   * Syntax:

*SELECT columns FROM table1 CROSS JOIN table2;*

**SQL Operators**

In SQL, operators are used to perform various operations on data, filter results, and create expressions. There are several types of operators in SQL. Let's explore some of the commonly used ones along with examples:

1. Arithmetic Operators:
   * These operators perform basic arithmetic operations.
   * Examples:

*SELECT 5 + 3; -- Addition (result: 8)*

*SELECT 10 - 4; -- Subtraction (result: 6)*

*SELECT 2 \* 6; -- Multiplication (result: 12)*

*SELECT 15 / 3; -- Division (result: 5)*

*SELECT 17 % 5; -- Modulus (result: 2, the remainder of 17 / 5)*

1. Comparison Operators:
   * Used to compare values and return true or false.
   * Examples:

*SELECT 10 > 5; -- Greater than (result: true)*

*SELECT 7 < 3; -- Less than (result: false)*

*SELECT 4 >= 4; -- Greater than or equal to (result: true)*

*SELECT 2 <= 1; -- Less than or equal to (result: false)*

*SELECT 5 = 5; -- Equal to (result: true)*

*SELECT 6 <> 8; -- Not equal to (result: true)*

1. Logical Operators:
   * Used to combine or negate conditions.
   * Examples:

*SELECT 10 > 5 AND 5 < 8; -- AND (result: true)*

*SELECT 10 > 5 OR 5 > 8; -- OR (result: true)*

*SELECT NOT(10 > 5); -- NOT (result: false)*

1. LIKE Operator:
   * Used to compare a value against a pattern using wildcard characters.
   * Examples:

*SELECT first\_name FROM employees WHERE first\_name LIKE 'J%'; -- Names starting with 'J'*

*SELECT product\_name FROM products WHERE product\_name LIKE '%widget%'; -- Names containing 'widget'*

1. IN Operator:
   * Used to specify multiple values in a WHERE clause.
   * Examples:

*SELECT product\_name FROM products WHERE product\_id IN (101, 102, 105);*

*Products with specific IDs*

*SELECT customer\_name FROM customers WHERE city IN ('New York', 'Los Angeles'); Customers from specific cities*

1. BETWEEN Operator:
   * Used to select a range of values.
   * Examples:

*SELECT product\_name FROM products WHERE price BETWEEN 10 AND 50;*

*Products with prices between 10 and 50*

*SELECT order\_date FROM orders WHERE order\_date BETWEEN '2023-01-01' AND '2023-03-31';*

*Orders between specific dates*

**Relational Algebra**

Relational algebra is a theoretical framework and mathematical language used to describe the operations that can be performed on relational databases. It provides a formal and precise way to express database operations, helping in the analysis and optimization of queries. SQL, the standard language for relational databases, is based on relational algebra principles.

The basic operations in relational algebra are:

1. Selection (σ): Selects rows from a table that satisfy a given condition.
2. Projection (π): Selects specific columns from a table.
3. Union (∪): Combines the rows of two tables without duplicate rows.
4. Set Difference (-): Retrieves rows from the first table that are not present in the second table.
5. Cartesian Product (×): Combines each row from the first table with each row from the second table.
6. Join (⋈): Combines rows from two tables based on a related column.

Here's an example of how relational algebra operations can be expressed in SQL:

Consider two tables:

"Students" table:

| **Student\_ID** | **Student\_Name** | **Age** | **Gender** |
| --- | --- | --- | --- |
| 1 | John | 20 | Male |
| 2 | Jane | 22 | Female |
| 3 | Mike | 19 | Male |

"Grades" table:

| **Student\_ID** | **Course** | **Grade** |
| --- | --- | --- |
| 1 | Math | A |
| 2 | Math | B |
| 3 | Math | C |
| 1 | Science | B |
| 2 | Science | A |

1. Selection (σ): SQL Example:

*SELECT \* FROM Students WHERE Age > 20;*

Result:

| **Student\_ID** | **Student\_Name** | **Age** | **Gender** |
| --- | --- | --- | --- |
| 2 | Jane | 22 | Female |

1. Projection (π): SQL Example:

*SELECT Student\_ID, Student\_Name FROM Students;*

Result:

| **Student\_ID** | **Student\_Name** |
| --- | --- |
| 1 | John |
| 2 | Jane |
| 3 | Mike |

1. Union (∪): SQL Example:

*SELECT \* FROM Students UNION SELECT \* FROM Grades;*

Result:

| **Student\_ID** | **Student\_Name** | **Age** | **Gender** | **Course** | **Grade** |
| --- | --- | --- | --- | --- | --- |
| 1 | John | 20 | Male | NULL | NULL |
| 2 | Jane | 22 | Female | NULL | NULL |
| 3 | Mike | 19 | Male | NULL | NULL |
| 1 | NULL | NULL | NULL | Math | A |
| 2 | NULL | NULL | NULL | Math | B |
| 3 | NULL | NULL | NULL | Math | C |
| 1 | NULL | NULL | NULL | Science | B |
| 2 | NULL | NULL | NULL | Science | A |

1. Set Difference (-): SQL Example:

*SELECT \* FROM Students EXCEPT SELECT \* FROM Grades;*

Result:

| **Student\_ID** | **Student\_Name** | **Age** | **Gender** |
| --- | --- | --- | --- |
| 3 | Mike | 19 | Male |

1. Cartesian Product (×): SQL Example:

*SELECT \* FROM Students, Grades;*

Result:

| **Student\_ID** | **Student\_Name** | **Age** | **Gender** | **Course** | **Grade** |
| --- | --- | --- | --- | --- | --- |
| 1 | John | 20 | Male | Math | A |
| 1 | John | 20 | Male | Math | B |
| 1 | John | 20 | Male | Math | C |
| 1 | John | 20 | Male | Science | A |
| 1 | John | 20 | Male | Science | B |
| 2 | Jane | 22 | Female | Math | A |
| 2 | Jane | 22 | Female | Math | B |
| 2 | Jane | 22 | Female | Math | C |
| 2 | Jane | 22 | Female | Science | A |
| 2 | Jane | 22 | Female | Science | B |
| 3 | Mike | 19 | Male | Math | A |
| 3 | Mike | 19 | Male | Math | B |
| 3 | Mike | 19 | Male | Math | C |
| 3 | Mike | 19 | Male | Science | A |
| 3 | Mike | 19 | Male | Science | B |

1. Join (⋈): SQL Example:

*SELECT Students.Student\_Name, Grades.Course, Grades.Grade FROM Students JOIN Grades ON Students.Student\_ID = Grades.Student\_ID;*

Result:

| **Student\_Name** | **Course** | **Grade** |
| --- | --- | --- |
| John | Math | A |
| Jane | Math | B |
| Mike | Math | C |
| John | Science | B |
| Jane | Science | A |

**SQL Constraints**

In SQL, constraints are rules defined on columns or tables to maintain data integrity and enforce specific conditions on the data being inserted, updated, or deleted in a database. Constraints help ensure that the data follows certain rules, preventing invalid or inconsistent data from being stored in the database. There are several types of constraints in SQL:

1. NOT NULL Constraint:
   * Ensures that a column must have a value; it cannot be NULL.
   * Example:

*CREATE TABLE employees ( employee\_id INT PRIMARY KEY, first\_name VARCHAR(50) NOT NULL, last\_name VARCHAR(50) NOT NULL, hire\_date DATE NOT NULL );*

1. UNIQUE Constraint:
   * Ensures that each value in the column is unique, i.e., no duplicate values are allowed.
   * Example:

*CREATE TABLE products ( product\_id INT PRIMARY KEY, product\_name VARCHAR(100) UNIQUE, price DECIMAL(10, 2) NOT NULL );*

1. PRIMARY KEY Constraint:
   * Specifies a unique identifier for each row in the table.
   * It automatically implies the UNIQUE and NOT NULL constraints.
   * Example:

*CREATE TABLE students ( student\_id INT PRIMARY KEY, student\_name VARCHAR(100), age INT, ... );*

1. FOREIGN KEY Constraint:
   * Establishes a relationship between two tables, where the column containing the foreign key refers to the primary key of another table.
   * It ensures referential integrity, meaning that the value in the foreign key column must exist in the referenced table or be NULL.
   * Example:

*CREATE TABLE orders ( order\_id INT PRIMARY KEY, customer\_id INT, order\_date DATE, ... FOREIGN KEY (customer\_id) REFERENCES customers(customer\_id) );*

1. CHECK Constraint:
   * Specifies a condition that must be satisfied for the column values in a table.
   * It ensures that only valid data is inserted into the table.
   * Example:

*CREATE TABLE employees ( employee\_id INT PRIMARY KEY, first\_name VARCHAR(50), last\_name VARCHAR(50), age INT CHECK (age >= 18) );*

1. DEFAULT Constraint:
   * Sets a default value for a column when no value is specified during an insertion.
   * Example:

*CREATE TABLE students ( student\_id INT PRIMARY KEY, student\_name VARCHAR(100), age INT DEFAULT 18 );*

**ACID Properties**

ACID properties are a set of four essential properties that ensure the reliability and consistency of transactions in a database management system. These properties are used to maintain data integrity, prevent data corruption, and guarantee that database operations are carried out reliably even in the event of failures. The ACID properties stand for:

1. Atomicity:
   * Atomicity ensures that a transaction is treated as a single, indivisible unit of work. It means that either all the operations within a transaction are completed successfully, or none of them are executed at all.
   * If any part of a transaction fails, the entire transaction is rolled back to its original state, and no changes are made to the database.
   * For example, when transferring money from one account to another, both the debit from one account and the credit to another account should happen together as a single atomic transaction.
2. Consistency:
   * Consistency ensures that the database moves from one consistent state to another consistent state after a successful transaction.
   * The database must satisfy certain rules and constraints before and after the execution of each transaction.
   * If a transaction violates any database constraint, it is rolled back to maintain consistency.
3. Isolation:
   * Isolation ensures that multiple transactions can execute concurrently without interfering with each other.
   * Each transaction is isolated from other transactions until it is completed.
   * Isolation prevents the "dirty read," "non-repeatable read," and "phantom read" anomalies that may occur when multiple transactions are executed simultaneously.
4. Durability:
   * Durability ensures that once a transaction is successfully committed, its changes are permanent and will survive any subsequent failures, such as power outages or crashes.
   * The changes made by a committed transaction are stored in a stable storage medium (such as disk) and are not lost even if the system crashes.

**SQL Queries**

**Easy Queries**

1. *Query: Retrieve all columns from a table. Solution:*

*SELECT \* FROM table\_name;*

1. *Query: Retrieve specific columns from a table. Solution:*

*SELECT column1, column2 FROM table\_name;*

1. *Query: Filter rows based on a condition. Solution:*

*SELECT \* FROM table\_name WHERE column\_name = value;*

1. *Query: Retrieve unique/distinct values from a column. Solution:*

*SELECT DISTINCT column\_name FROM table\_name;*

1. *Query: Sort the result in ascending order. Solution:*

*SELECT \* FROM table\_name ORDER BY column\_name ASC;*

1. *Query: Sort the result in descending order. Solution:*

*SELECT \* FROM table\_name ORDER BY column\_name DESC;*

1. *Query: Count the number of rows in a table. Solution:*

*SELECT COUNT(\*) FROM table\_name;*

1. *Query: Sum the values of a numeric column. Solution:*

*SELECT SUM(column\_name) FROM table\_name;*

1. *Query: Find the maximum value in a column. Solution:*

*SELECT MAX(column\_name) FROM table\_name;*

1. *Query: Find the minimum value in a column. Solution:*

*SELECT MIN(column\_name) FROM table\_name;*

1. *Query: Calculate the average of a numeric column. Solution:*

*SELECT AVG(column\_name) FROM table\_name;*

1. *Query: Retrieve data from multiple tables using INNER JOIN. Solution:*

*SELECT column1, column2 FROM table1 INNER JOIN table2 ON table1.column = table2.column;*

1. *Query: Retrieve data from multiple tables using LEFT JOIN. Solution:*

*SELECT column1, column2 FROM table1 LEFT JOIN table2 ON table1.column = table2.column;*

1. *Query: Retrieve data from multiple tables using RIGHT JOIN. Solution:*

*SELECT column1, column2 FROM table1 RIGHT JOIN table2 ON table1.column = table2.column;*

1. *Query: Retrieve data from multiple tables using FULL JOIN. Solution:*

*SELECT column1, column2 FROM table1 FULL JOIN table2 ON table1.column = table2.column;*

1. *Query: Insert a new row into a table. Solution:*

*INSERT INTO table\_name (column1, column2) VALUES (value1, value2);*

1. *Query: Update data in a table. Solution:*

*UPDATE table\_name SET column\_name = new\_value WHERE condition;*

1. *Query: Delete data from a table. Solution:*

*DELETE FROM table\_name WHERE condition;*

1. *Query: Filter rows using multiple conditions (AND). Solution:*

*SELECT \* FROM table\_name WHERE condition1 AND condition2;*

1. *Query: Filter rows using multiple conditions (OR). Solution:*

*SELECT \* FROM table\_name WHERE condition1 OR condition2;*

1. *Query: Retrieve data between two dates. Solution:*

*SELECT \* FROM table\_name WHERE date\_column BETWEEN 'start\_date' AND 'end\_date';*

1. *Query: Retrieve data from the top N rows. Solution:*

*SELECT TOP N \* FROM table\_name;*

*(Note: The syntax may vary depending on the database system.)*

1. *Query: Retrieve data using the LIKE operator for pattern matching. Solution:*

*SELECT \* FROM table\_name WHERE column\_name LIKE 'pattern%';*

1. *Query: Group data and apply aggregate functions using GROUP BY. Solution:*

*SELECT column1, COUNT(\*) FROM table\_name GROUP BY column1;*

1. *Query: Apply aggregate functions on grouped data using HAVING. Solution:*

*SELECT column1, COUNT(\*) FROM table\_name GROUP BY column1 HAVING COUNT(\*) > N;*

***Intermediate Queries***

1. *Query: Retrieve the names of all employees along with their department names. Solution:*

*SELECT employees.employee\_name, departments.department\_name FROM employees INNER JOIN departments ON employees.department\_id = departments.department\_id;*

1. *Query: Retrieve the highest salary from the "salaries" table. Solution:*

*SELECT MAX(salary) AS highest\_salary FROM salaries;*

1. *Query: Retrieve the average salary of all employees. Solution:*

*SELECT AVG(salary) AS average\_salary FROM salaries;*

1. *Query: Retrieve the total salary expense of the company. Solution:*

*SELECT SUM(salary) AS total\_salary\_expense FROM salaries;*

1. *Query: Retrieve the number of employees in each department. Solution:*

*SELECT department\_id, COUNT(\*) AS employee\_count FROM employees GROUP BY department\_id;*

1. *Query: Retrieve the top N employees with the highest salary. Solution:*

*SELECT employee\_name, salary FROM employees ORDER BY salary DESC LIMIT N;*

*(Note: The syntax may vary depending on the database system.)*

1. *Query: Retrieve the names of employees whose names contain "John" and have a salary greater than 50000. Solution:*

*SELECT employee\_name FROM employees WHERE employee\_name LIKE '%John%' AND salary > 50000;*

1. *Query: Retrieve the names of employees who have not been assigned to any department. Solution:*

*SELECT employee\_name FROM employees WHERE department\_id IS NULL;*

1. *Query: Retrieve the employees who joined in the last 6 months. Solution:*

*SELECT employee\_name FROM employees WHERE hire\_date >= DATE\_SUB(NOW(), INTERVAL 6 MONTH);*

1. *Query: Retrieve the employees and their corresponding managers (if available). Solution:*

*SELECT e.employee\_name, m.employee\_name AS manager\_name FROM employees e LEFT JOIN employees m ON e.manager\_id = m.employee\_id;*

1. *Query: Retrieve the highest salary in each department. Solution:*

*SELECT department\_id, MAX(salary) AS highest\_salary FROM employees GROUP BY department\_id;*

1. *Query: Retrieve the average salary for each job title. Solution:*

*SELECT job\_title, AVG(salary) AS average\_salary FROM employees GROUP BY job\_title;*

1. *Query: Retrieve the employees and their direct and indirect reports in a hierarchical manner. Solution:*

*WITH RECURSIVE employee\_hierarchy AS ( SELECT employee\_id, employee\_name, manager\_id, 0 AS level FROM employees WHERE manager\_id IS NULL UNION ALL SELECT e.employee\_id, e.employee\_name, e.manager\_id, eh.level + 1 FROM employees e JOIN employee\_hierarchy eh ON e.manager\_id = eh.employee\_id ) SELECT employee\_id, employee\_name, level FROM employee\_hierarchy;*

1. *Query: Retrieve the average salary for each department and job title combination. Solution:*

*SELECT department\_id, job\_title, AVG(salary) AS average\_salary FROM employees GROUP BY department\_id, job\_title;*

1. *Query: Retrieve the employees who have the same salary as their managers. Solution:*

*SELECT e.employee\_name FROM employees e JOIN employees m ON e.manager\_id = m.employee\_id WHERE e.salary = m.salary;*

1. *Query: Retrieve the department with the highest average salary. Solution:*

*SELECT department\_id, AVG(salary) AS average\_salary FROM employees GROUP BY department\_id ORDER BY average\_salary DESC LIMIT 1;*

1. *Query: Retrieve the top N departments with the highest total salary expense. Solution:*

*SELECT department\_id, SUM(salary) AS total\_salary\_expense FROM employees GROUP BY department\_id ORDER BY total\_salary\_expense DESC LIMIT N;*

1. *Query: Retrieve the employees who have the same job title as their colleagues in their department. Solution:*

*SELECT e.employee\_name FROM employees e JOIN employees c ON e.department\_id = c.department\_id AND e.job\_title = c.job\_title WHERE e.employee\_id <> c.employee\_id;*

1. *Query: Retrieve the employees in the same department with a higher salary than their colleagues. Solution:*

*SELECT e.employee\_name FROM employees e JOIN employees c ON e.department\_id = c.department\_id WHERE e.salary > c.salary;*

1. *Query: Retrieve the names of employees who have changed their job title at least once. Solution:*

*SELECT DISTINCT employee\_name FROM employees GROUP BY employee\_name, job\_title HAVING COUNT(DISTINCT job\_title) > 1;*

***Hard Queries***

1. *Query: Retrieve the names of employees and their corresponding managers in a hierarchical manner, including all levels of management.*

*WITH RECURSIVE employee\_hierarchy AS ( SELECT employee\_id, employee\_name, manager\_id, 0 AS level FROM employees WHERE manager\_id IS NULL UNION ALL SELECT e.employee\_id, e.employee\_name, e.manager\_id, eh.level + 1 FROM employees e JOIN employee\_hierarchy eh ON e.manager\_id = eh.employee\_id ) SELECT employee\_id, employee\_name, level FROM employee\_hierarchy;*

*Output:*

| ***employee\_id*** | ***employee\_name*** | ***level*** |
| --- | --- | --- |
| *1* | *John* | *0* |
| *2* | *Jane* | *0* |
| *3* | *Mike* | *1* |
| *4* | *Mary* | *2* |
| *5* | *Alice* | *3* |

1. *Query: Retrieve the top N departments with the highest total salary expense.*

*SELECT department\_id, SUM(salary) AS total\_salary\_expense FROM employees GROUP BY department\_id ORDER BY total\_salary\_expense DESC LIMIT N;*

*Output:*

| ***department\_id*** | ***total\_salary\_expense*** |
| --- | --- |
| *101* | *120000* |
| *102* | *110000* |
| *103* | *100000* |

1. *Query: Retrieve the employees who are in the same department and have a salary greater than their colleagues.*

*SELECT e.employee\_name FROM employees e JOIN employees c ON e.department\_id = c.department\_id WHERE e.salary > c.salary;*

*Output:*

| ***employee\_name*** |
| --- |
| *John* |
| *Jane* |
| *Mike* |
| *Alice* |

1. *Query: Retrieve the employees who have changed their job title at least once.*

*SELECT DISTINCT employee\_name FROM employees GROUP BY employee\_name, job\_title HAVING COUNT(DISTINCT job\_title) > 1;*

*Output:*

| ***employee\_name*** |
| --- |
| *John* |
| *Jane* |
| *Mike* |

1. *Query: Retrieve the departments with more than 5 employees and their average salary.*

*SELECT department\_id, COUNT(\*) AS employee\_count, AVG(salary) AS average\_salary FROM employees GROUP BY department\_id HAVING COUNT(\*) > 5;*

*Output:*

| ***department\_id*** | ***employee\_count*** | ***average\_salary*** |
| --- | --- | --- |
| *101* | *6* | *30000* |
| *103* | *8* | *25000* |

1. *Query: Retrieve the names of employees who have the highest salary in their department.*

*SELECT employee\_name FROM employees e WHERE salary = ( SELECT MAX(salary) FROM employees WHERE department\_id = e.department\_id );*

*Output:*

| ***employee\_name*** |
| --- |
| *John* |
| *Jane* |
| *Mike* |

1. *Query: Retrieve the employees who joined in the last 6 months.*

*SELECT employee\_name FROM employees WHERE hire\_date >= DATE\_SUB(NOW(), INTERVAL 6 MONTH);*

*Output:*

| ***employee\_name*** |
| --- |
| *John* |
| *Jane* |

1. *Query: Retrieve the departments and the difference between the maximum and minimum salary in each department.*

*SELECT department\_id, MAX(salary) - MIN(salary) AS salary\_difference FROM employees GROUP BY department\_id;*

*Output:*

| ***department\_id*** | ***salary\_difference*** |
| --- | --- |
| *101* | *15000* |
| *102* | *10000* |
| *103* | *8000* |

1. *Query: Retrieve the names of employees who have the same job title as their colleagues in other departments.*

*SELECT e.employee\_name FROM employees e JOIN employees c ON e.job\_title = c.job\_title AND e.department\_id <> c.department\_id;*

*Output:*

| ***employee\_name*** |
| --- |
| *John* |
| *Jane* |

1. *Query: Retrieve the employees who have the highest salary in their job title.*

*SELECT employee\_name, job\_title, salary FROM employees e WHERE salary = ( SELECT MAX(salary) FROM employees WHERE job\_title = e.job\_title );*

*Output:*

| ***employee\_name*** | ***job\_title*** | ***salary*** |
| --- | --- | --- |
| *John* | *Manager* | *40000* |
| *Alice* | *Analyst* | *25000* |