SQL

What is sql

Structured Query Language, is a **standard programming language** used for

1. Managing
2. Manipulating relational databases.

Perform a variety of operations on database data,

1. Retrieving
2. Inserting
3. Updating
4. Deleting records

SQL is widely used in **data management and is essential for interacting** with **relational database management systems (RDBMS)** like **MySQL, PostgreSQL, SQL Server, Oracle, and SQLite.**

SQL provides powerful, optimized ways to access, analyze, and manipulate large volumes of data, making it crucial for database management in applications across industries.

Importance of SQL

Why SQL is Important

- **Standardized Language**: SQL is widely supported across database systems.
- **Declarative Approach**: Instead of specifying *how* to do something, SQL allows users to specify *what* they want.
- **Efficient Data Management**: SQL provides powerful, o**ptimized ways to access, analyze, and manipulate large volumes of data,** making it crucial for database management in applications across industries.

Key Purpose of SQL

1. **Schema Management**
2. Data Manipulation (CRUD Operations)
3. Permissions/DATA control
4. Transactions
5. **Data Retrieval**

Types of SQL Statements

- **Data Query Language (DQL)**: Focused on querying or retrieving data (`SELECT`).
- **Data Definition Language (DDL)**: For defining database structure (`CREATE`, `ALTER`, `DROP`).
- **Data Manipulation Language (DML)**: For modifying data (`INSERT`, `UPDATE`, `DELETE`).
- **Data Control Language (DCL)**: For controlling access to data. (`GRANT`, `REVOKE`).
- **Transaction Control Language (TCL)**: For managing the integrity of transactions. (`COMMIT`, `ROLLBACK`).

1. Schema Management

Schema Management: SQL provides **commands to create, modify, and delete** database structures (like tables, indexes, and relationships) using **Data Definition Language (DDL)** statements like

`CREATE`, `ALTER`, and `DROP`

Command	Purpose	Example Usage
CREATE	Define new objects (tables, indexes, etc.)	`CREATE TABLE Employees (EmployeeID INT PRIMARY KEY, ...);`
ALTER	Modify existing objects	`ALTER TABLE Employees ADD Email VARCHAR(100);`
DROP	Delete objects	`DROP TABLE Employees;`

2. Data Manipulation

Data Manipulation refers to the **CRUD** operations. CRUD stands for **Create, Read, Update, and Delete**, and each of these operations corresponds to a specific SQL command:

1. CREATE - CREATE/INSERT
2. READ
3. UPDATE
4. DELETE

Create (`INSERT`) new records.

Read (`SELECT`) data from the database.

Update (`UPDATE`) existing data.

Delete (`DELETE`) records.

3. Permissions

SQL allows for managing user access to data. `GRANT` **and** `REVOKE` **statements control who can view or modify data within a database.**

```
GRANT SELECT, INSERT ON employees TO user_name;
```

4. Transactions

SQL can manage **multiple operations** as a **single unit of work**, known as a **transaction**, which ensures **data integrity**. Commands like **`BEGIN`, `COMMIT`, and `ROLLBACK` control transactions.**

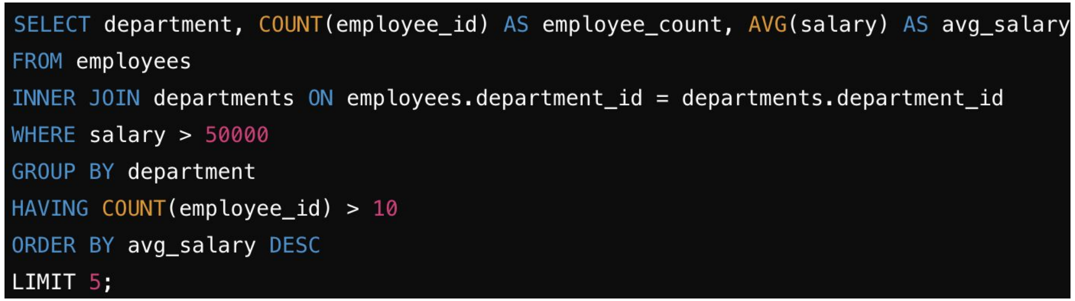
```
BEGIN; UPDATE employees SET age = 31 WHERE name = 'John Doe'; COMMIT;
```

5. Data Retrieval / Querying

SQL allows users to retrieve specific data from databases based on various conditions. T**he `SELECT` statement is the most common command, used to extract data.**

Basic SQL Query

A basic SQL query is structured to define **what data to retrieve, from where, and how to organize it.**



```
SELECT department, COUNT(employee_id) AS employee_count, AVG(salary) AS avg_salary
FROM employees
INNER JOIN departments ON employees.department_id = departments.department_id
WHERE salary > 50000
GROUP BY department
HAVING COUNT(employee_id) > 10
ORDER BY avg_salary DESC
LIMIT 5;
```

Explanation of the Example Query

1. **SELECT**: Retrieves the `department`, `employee_count`, and `avg_salary`.
2. **FROM**: Specifies `employees` as the source table.
3. **JOIN**: Combines `employees` and `departments` based on `department_id`.
4. **WHERE**: Filters for employees with a salary greater than 50,000.
5. **GROUP BY**: Groups results by department.
6. **HAVING**: Filters out departments with fewer than 10 employees.
7. **ORDER BY**: Sorts results by `avg_salary` in descending order.
8. **LIMIT**: Returns only the top 5 departments in the results.

Now we know how to build a query Lets go to the advanced

Manipulate

1. Data
2. Perform calculations on query

Functions

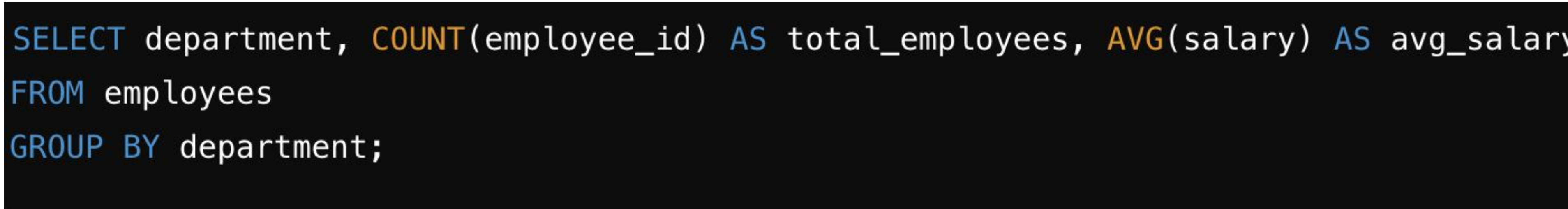
1. **Aggregate Functions**
2. **Window (or Analytic) Functions**
3. **String Functions**
4. **Date and Time Functions**
5. **Mathematical Functions**
6. **Conditional Functions**
7. **Conversion Functions**
8. **JSON Functions** (Available in databases supporting JSON data types)

1. Aggregate Functions

Aggregate functions perform calculations on a set of rows and return a single result, often used with `GROUP BY` **clauses**.

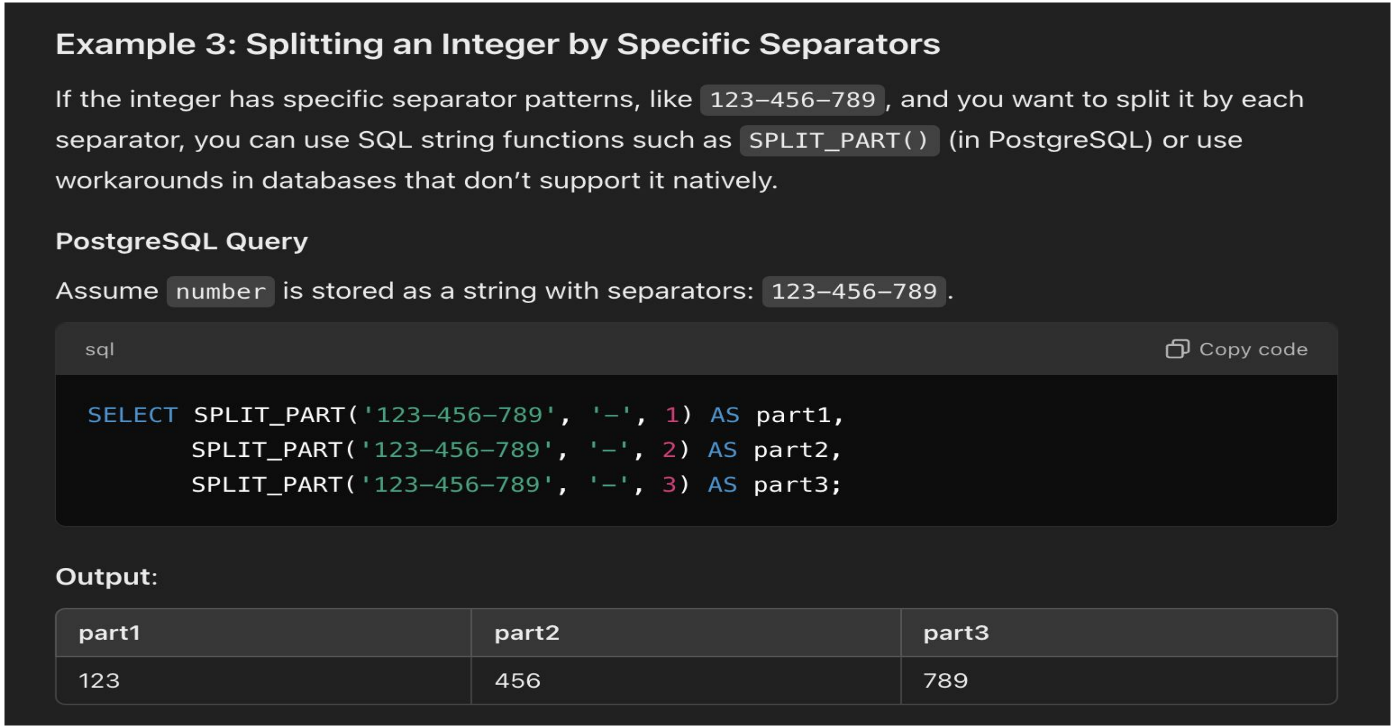
- **SUM()**: Calculates the total sum of a numeric column.
- **COUNT()**: Counts the number of rows or non-null values in a column.
- **AVG()**: Calculates the average of a numeric column.
- **MIN()**: Returns the minimum value in a column.
- **MAX()**: Returns the maximum value in a column.

VARIANCE() - Calculates the variance of a set of values.
STDDEV() - Calculates the standard deviation of a set of values.
GROUP_CONCAT() - Concatenates values from a group into a single string.
FIRST() - Retrieves the first value in a column for a group (not universally supported).
LAST() - Retrieves the last value in a column for a group (not universally supported).
PERCENTILE_CONT() - Returns a percentile value as a continuous distribution (in some SQL systems).
PERCENTILE_DISC() - Returns a percentile value as a discrete distribution (in some SQL systems).
MEDIAN() - Finds the median value of a column (supported in some databases).
MODE() - Retrieves the most frequently occurring value in a column (rarely supported directly).
ARRAY_AGG() - Aggregates column values into an array (PostgreSQL, SQL Server).



```sql
SELECT department, COUNT(employee_id) AS total_employees, AVG(salary) AS avg_salar
FROM employees
GROUP BY department;
```

Splitting with a separator - SPLIT_PART(column, ‘-’, 1) AS part1



Example 3: Splitting an Integer by Specific Separators

If the integer has specific separator patterns, like `123-456-789`, and you want to split it by each separator, you can use SQL string functions such as `SPLIT_PART()` (in PostgreSQL) or use workarounds in databases that don't support it natively.

PostgreSQL Query

Assume `number` is stored as a string with separators: `123-456-789`.

```sql
SELECT SPLIT_PART('123-456-789', '-', 1) AS part1,
       SPLIT_PART('123-456-789', '-', 2) AS part2,
       SPLIT_PART('123-456-789', '-', 3) AS part3;
```

Output:

part1	part2	part3
123	456	789

2. Window Function

Window functions perform calculations across a set of table rows that are related to the current row, providing more granular analysis over groups of rows. They use the `OVER()` clause to define a “window” of rows.

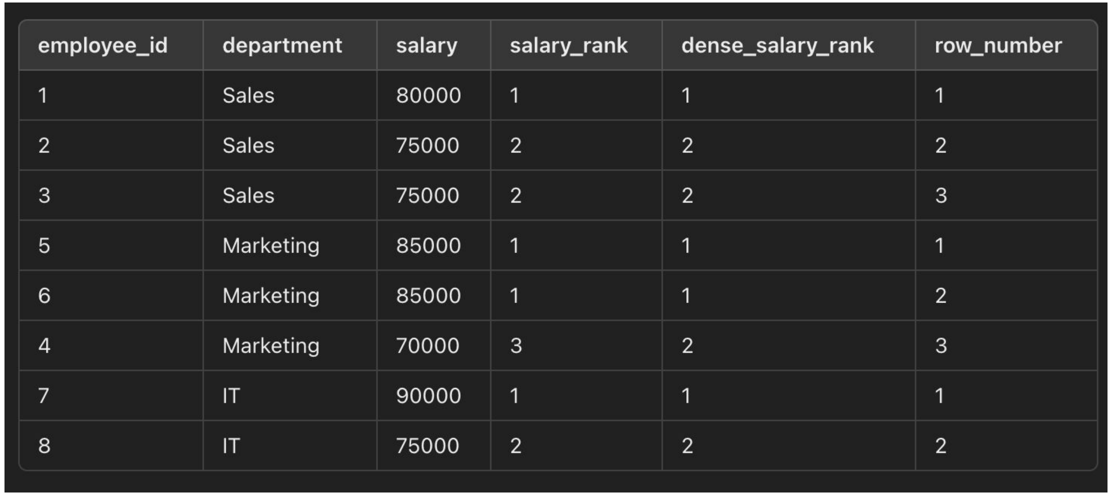
- **ROW_NUMBER()**: Assigns a unique number to each row in a partition.
- **RANK()**: Assigns a rank to each row within a partition, skipping ranks for tied values.
- **DENSE_RANK()**: Similar to `RANK()`, but without gaps for tied values.
- **LAG() / LEAD()**: Accesses data from previous or next rows in a window.
- **SUM(), AVG(), MIN(), MAX()**: Also work as window functions to calculate rolling or cumulative totals.

NTILE() - Distributes rows into a specified number of ranked groups.
FIRST_VALUE() - Returns the first value in a partition.
LAST_VALUE() - Returns the last value in a partition.
SUM() OVER() - Calculates a running or cumulative sum.
AVG() OVER() - Calculates a running or cumulative average.
PERCENT_RANK() - Calculates the relative rank as a percentage.
CUME_DIST() - Calculates the cumulative distribution, showing the proportion of rows.
NTH_VALUE() - Returns the nth value in a partition.
MIN() OVER() - Returns a running minimum.
MAX() OVER() - Returns a running maximum.

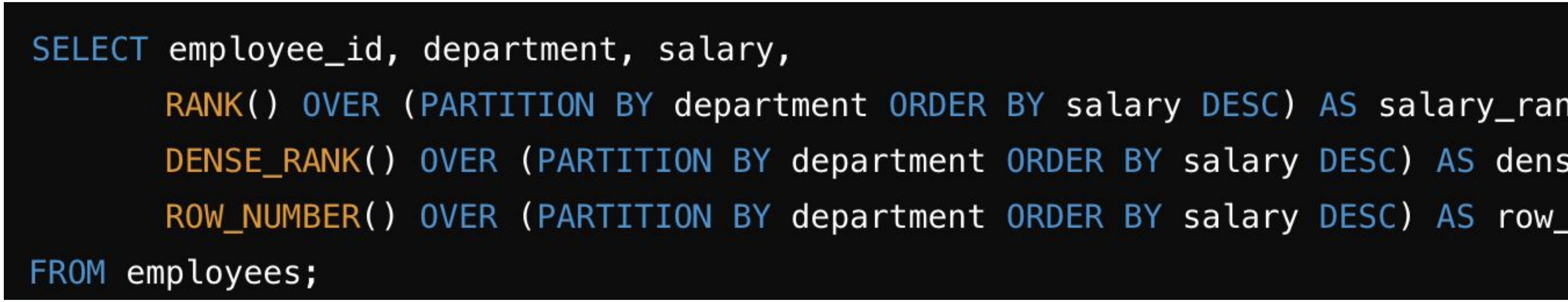
RANK()

1. RANK()
2. DENSE_RANK()
3. ROW_NUMBER()

WITH OVER() clause



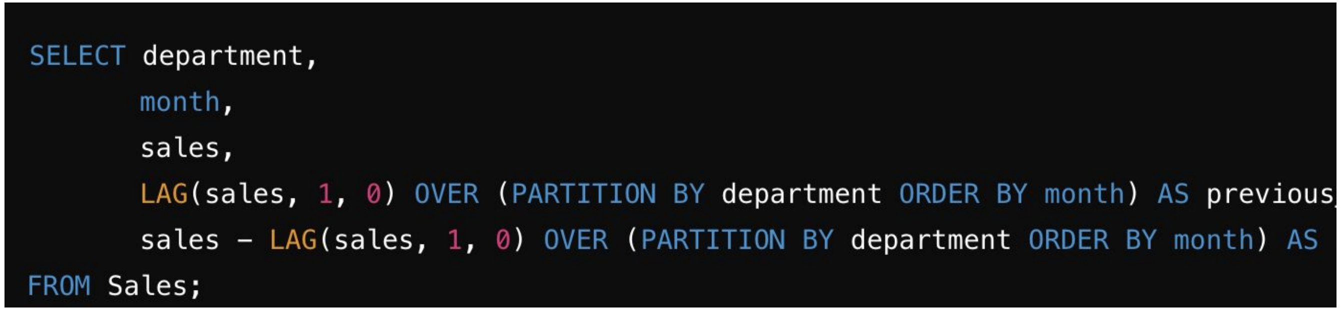
employee_id	department	salary	salary_rank	dense_salary_rank	row_number
1	Sales	80000	1	1	1
2	Sales	75000	2	2	2
3	Sales	75000	2	2	3
5	Marketing	85000	1	1	1
6	Marketing	85000	1	1	2
4	Marketing	70000	3	2	3
7	IT	90000	1	1	1
8	IT	75000	2	2	2



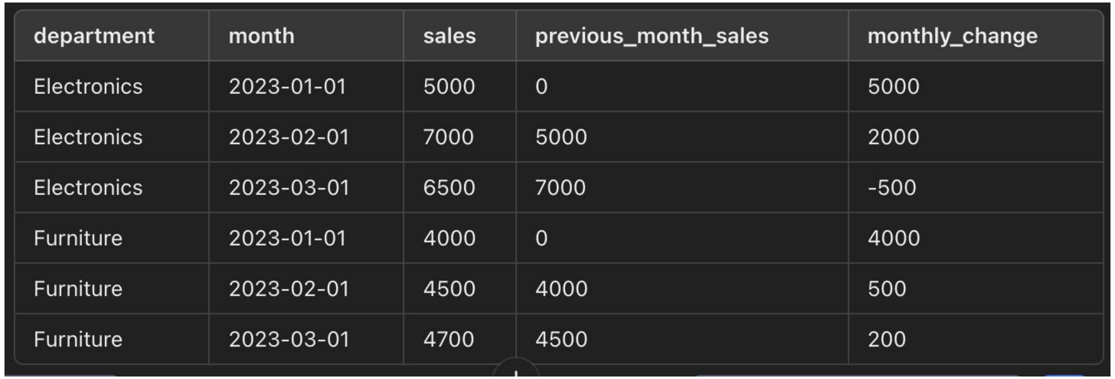
```
SELECT employee_id, department, salary,
       RANK() OVER (PARTITION BY department ORDER BY salary DESC) AS salary_ran
       DENSE_RANK() OVER (PARTITION BY department ORDER BY salary DESC) AS dens
       ROW_NUMBER() OVER (PARTITION BY department ORDER BY salary DESC) AS row_
FROM employees;
```

LAG() LEAD()

The `LAG()` and `LEAD()` functions in SQL are **window functions** used to access data from a previous (lag) or next (lead) row within the same result set, based on a specified ordering



```sql
SELECT department,
       month,
       sales,
       LAG(sales, 1, 0) OVER (PARTITION BY department ORDER BY month) AS previous
       sales - LAG(sales, 1, 0) OVER (PARTITION BY department ORDER BY month) AS
FROM Sales;
```



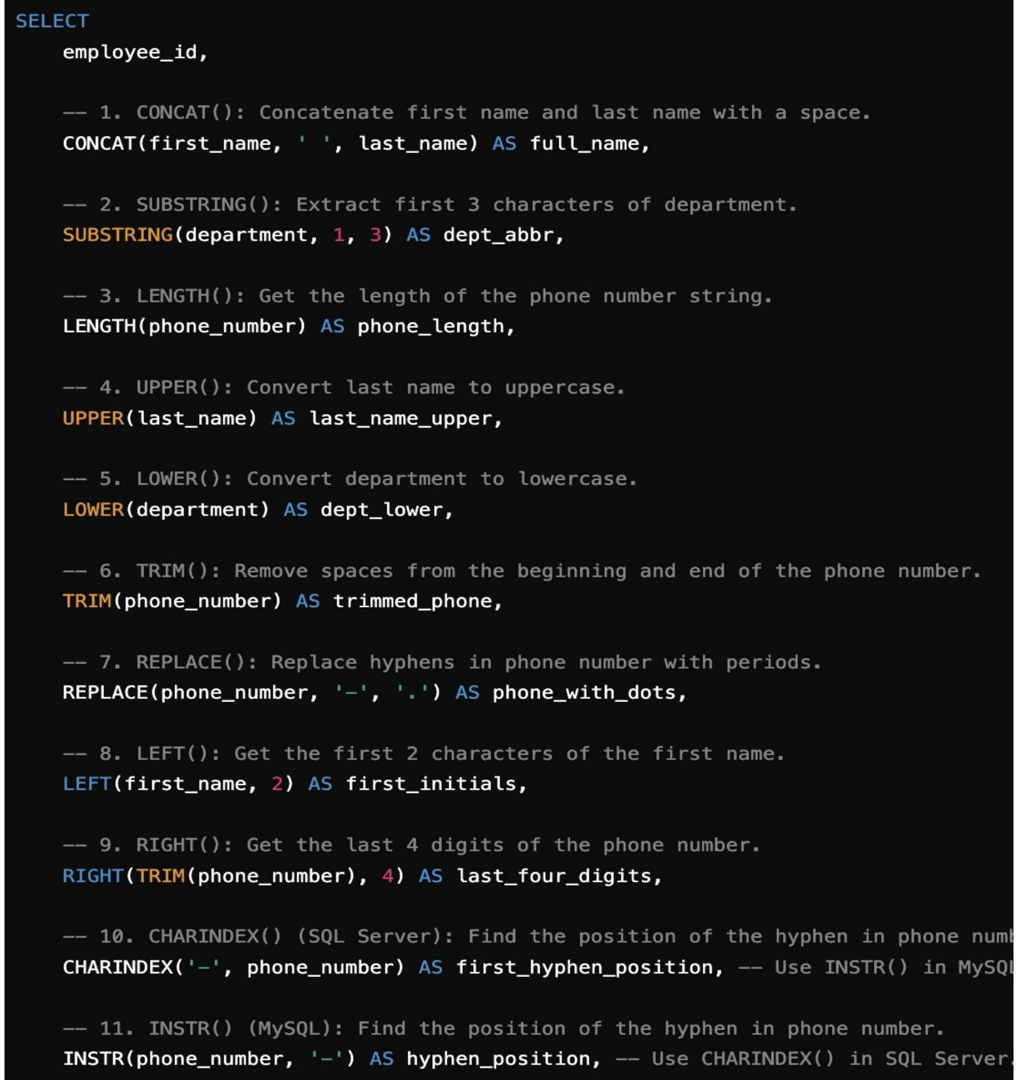
department	month	sales	previous_month_sales	monthly_change
Electronics	2023-01-01	5000	0	5000
Electronics	2023-02-01	7000	5000	2000
Electronics	2023-03-01	6500	7000	-500
Furniture	2023-01-01	4000	0	4000
Furniture	2023-02-01	4500	4000	500
Furniture	2023-03-01	4700	4500	200

3. String Function

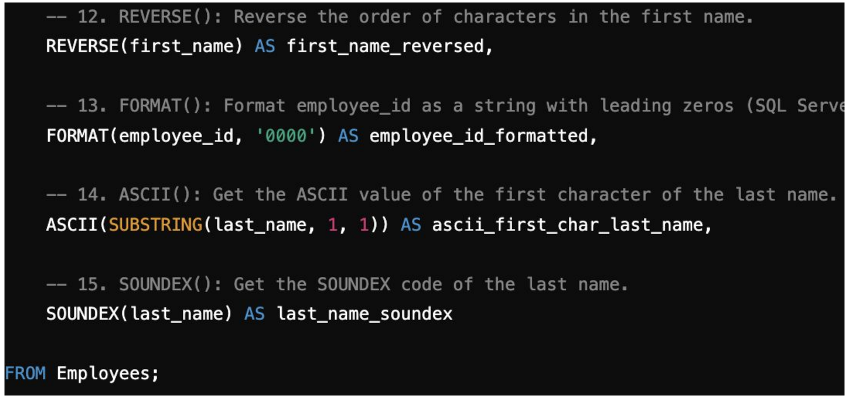
String functions allow you to manipulate text data.

- **UPPER() / LOWER()**: Converts text to uppercase or lowercase.
- **CONCAT()**: Combines two or more strings into one.
- **SUBSTRING()**: Extracts a substring from a string.
- **TRIM()**: Removes whitespace or specified characters from the beginning and end of a string.
- **LENGTH()**: Returns the length of a string.

REPLACE() - Replaces occurrences of a substring within a string.
LEFT() - Returns the left part of a string.
RIGHT() - Returns the right part of a string.
CHARINDEX() - Finds the position of a substring within a string (SQL Server).
INSTR() - Finds the position of a substring within a string (MySQL).
REVERSE() - Reverses the characters in a string.
FORMAT() - Formats numbers as a string (SQL Server).
ASCII() - Returns the ASCII value of the first character in a string.
SOUNDEX() - Returns a code representing how a string sounds when spoken.



```
SELECT
    employee_id,

    -- 1. CONCAT(): Concatenate first name and last name with a space.
    CONCAT(first_name, ' ', last_name) AS full_name,

    -- 2. SUBSTRING(): Extract first 3 characters of department.
    SUBSTRING(department, 1, 3) AS dept_abbr,

    -- 3. LENGTH(): Get the length of the phone number string.
    LENGTH(phone_number) AS phone_length,

    -- 4. UPPER(): Convert last name to uppercase.
    UPPER(last_name) AS last_name_upper,

    -- 5. LOWER(): Convert department to lowercase.
    LOWER(department) AS dept_lower,

    -- 6. TRIM(): Remove spaces from the beginning and end of the phone number.
    TRIM(phone_number) AS trimmed_phone,

    -- 7. REPLACE(): Replace hyphens in phone number with periods.
    REPLACE(phone_number, '-', '.') AS phone_with_dots,

    -- 8. LEFT(): Get the first 2 characters of the first name.
    LEFT(first_name, 2) AS first_initials,

    -- 9. RIGHT(): Get the last 4 digits of the phone number.
    RIGHT(TRIM(phone_number), 4) AS last_four_digits,

    -- 10. CHARINDEX() (SQL Server): Find the position of the hyphen in phone numb
    CHARINDEX('-', phone_number) AS first_hyphen_position, -- Use INSTR() in MySQL

    -- 11. INSTR() (MySQL): Find the position of the hyphen in phone number.
    INSTR(phone_number, '-') AS hyphen_position, -- Use CHARINDEX() in SQL Server.
```



```
    -- 12. REVERSE(): Reverse the order of characters in the first name.
    REVERSE(first_name) AS first_name_reversed,

    -- 13. FORMAT(): Format employee_id as a string with leading zeros (SQL Serve
    FORMAT(employee_id, '0000') AS employee_id_formatted,

    -- 14. ASCII(): Get the ASCII value of the first character of the last name.
    ASCII(SUBSTRING(last_name, 1, 1)) AS ascii_first_char_last_name,

    -- 15. SOUNDEX(): Get the SOUNDEX code of the last name.
    SOUNDEX(last_name) AS last_name_soundex

FROM Employees;
```

4.Date and Time Function

ate functions manipulate and perform calculations on date and time data.

- **NOW() / CURRENT_DATE()**: Returns the current date and time or just the date.
- **DATEADD() / DATEDIFF()**: Adds a specified interval to a date or calculates the difference between two dates.
- **YEAR(), MONTH(), DAY()**: Extracts the year, month, or day from a date.
- **DATE_FORMAT()**: Formats a date based on specified output patterns.

NOW() - Returns the current date and time.
CURRENT_DATE() - Returns the current date.
CURRENT_TIME() - Returns the current time.
DATEADD() - Adds a specified interval to a date.
DATEDIFF() - Returns the difference between two dates.
YEAR() - Extracts the year from a date.
MONTH() - Extracts the month from a date.
DAY() - Extracts the day from a date.
DATE_FORMAT() - Formats a date in a specified pattern (MySQL).
TO_CHAR() - Formats a date or timestamp (Oracle, PostgreSQL).
EXTRACT() - Extracts parts of a date (PostgreSQL, MySQL).
DAYOFWEEK() - Returns the day of the week from a date.
TIMESTAMPDIFF() - Returns the difference between timestamps.
DATE_TRUNC() - Truncates a date to a specified unit (PostgreSQL).
STR_TO_DATE() - Converts a string to a date (MySQL).

5. Mathematical Functions

- **ROUND()**: Rounds a number to a specified number of decimal places.
- **FLOOR() / CEILING()**: Rounds a number down or up to the nearest integer.
- **ABS()**: Returns the absolute value of a number.
- **POWER()**: Raises a number to the power of another number.

ROUND() - Rounds a number to a specified number of decimal places.
CEILING() - Rounds a number up to the nearest integer.
FLOOR() - Rounds a number down to the nearest integer.
ABS() - Returns the absolute value of a number.
POWER() - Raises a number to the power of another number.
SQRT() - Calculates the square root of a number.
MOD() - Returns the remainder of a division.
EXP() - Returns e raised to the power of a given number.
LOG() - Returns the natural logarithm of a number.
LOG10() - Returns the base-10 logarithm of a number.
PI() - Returns the value of pi.
SIGN() - Returns the sign of a number (+1, 0, or -1).
SIN() - Calculates the sine of a number.
COS() - Calculates the cosine of a number.
TAN() - Calculates the tangent of a number.

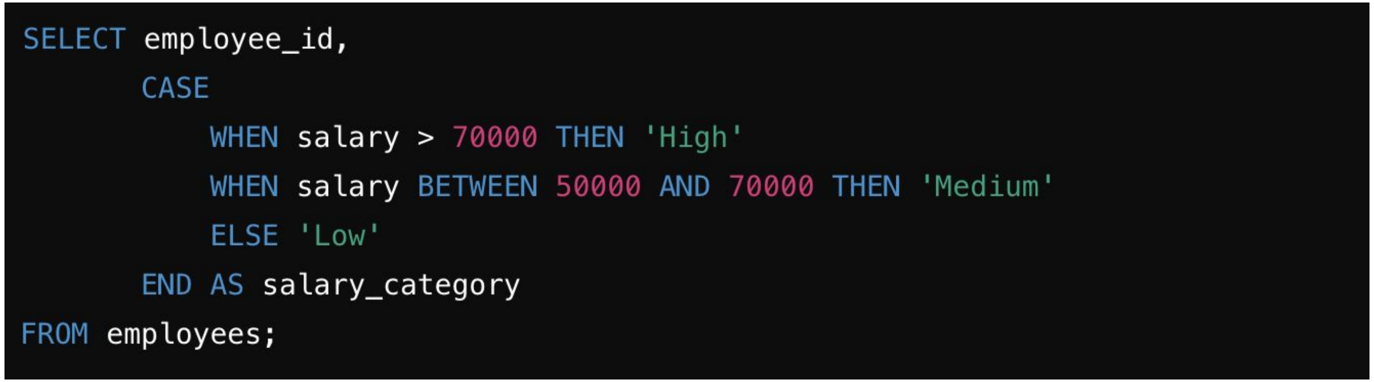
6. Conditional Functions

Conditional Functions

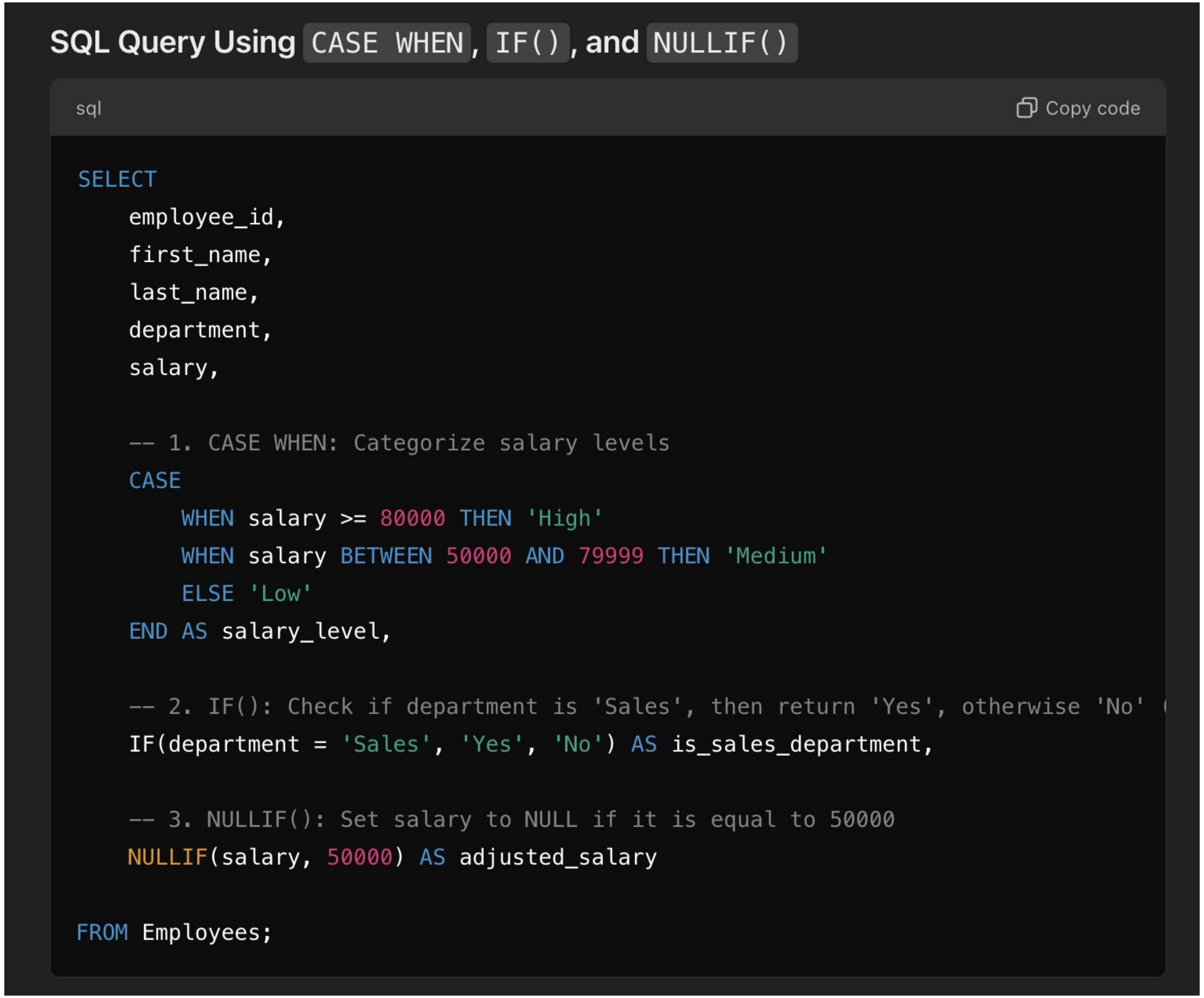
Conditional functions let you handle conditions within queries, often using the `CASE` statement.

- **CASE WHEN**: Allows for conditional logic, similar to an if-else statement.
- **IFNULL() / COALESCE()**: Returns a default value if a column is `NULL`.

CASE WHEN - Implements conditional logic (if-else).
IF() - Conditional function in MySQL (similar to CASE WHEN).
NULLIF() - Returns NULL if two expressions are equal.
COALESCE() - Returns the first non-null expression.
IFNULL() - Returns a specified value if an expression is NULL (MySQL).
ISNULL() - Checks if an expression is NULL (SQL Server).
IIF() - Inline conditional function (SQL Server).
DECODE() - Works like CASE in Oracle SQL.
GREATEST() - Returns the largest value among provided values.
LEAST() - Returns the smallest value among provided values.



```sql
SELECT employee_id,
       CASE
           WHEN salary > 70000 THEN 'High'
           WHEN salary BETWEEN 50000 AND 70000 THEN 'Medium'
           ELSE 'Low'
       END AS salary_category
FROM employees;
```



SQL Query Using `CASE WHEN`, `IF()`, and `NULLIF()`

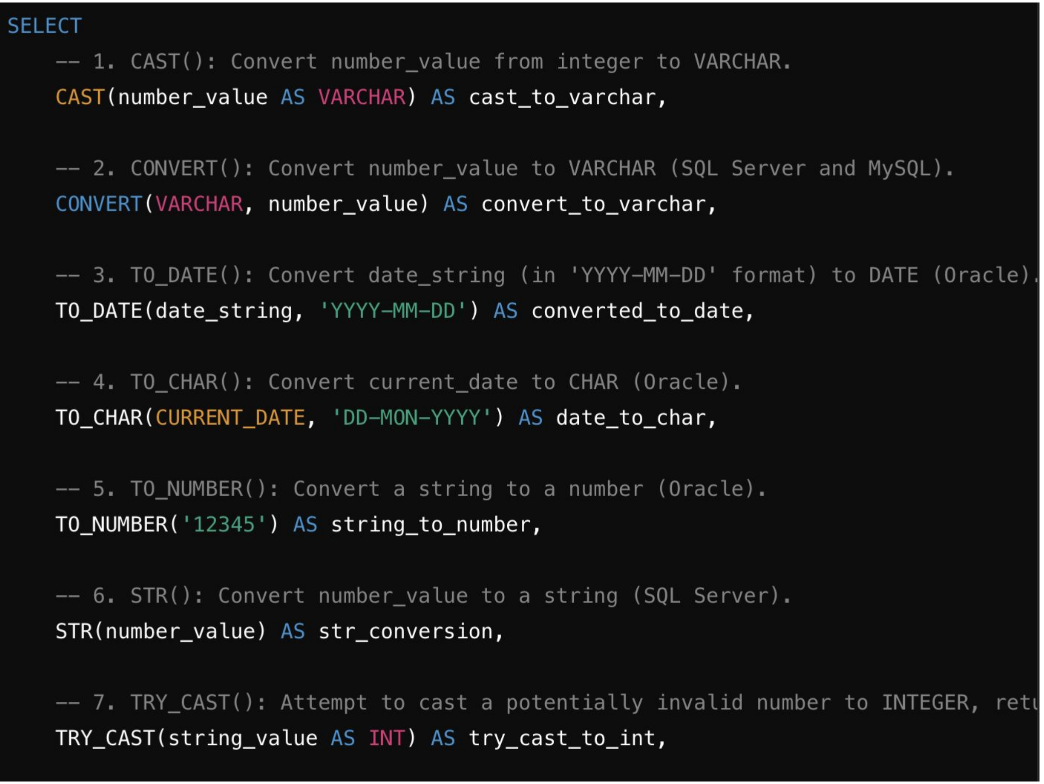
sql

Copy code

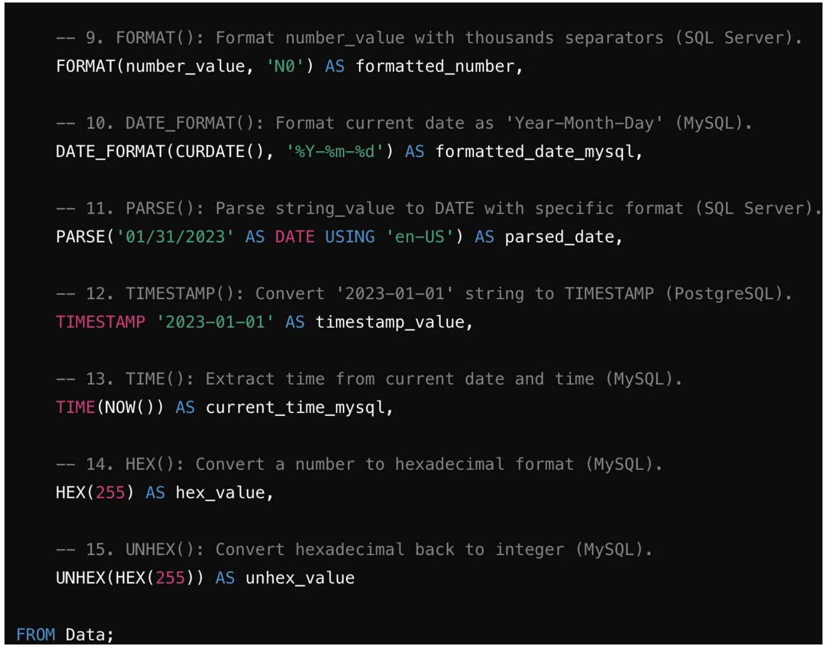
```sql
SELECT
    employee_id,
    first_name,
    last_name,
    department,
    salary,

    -- 1. CASE WHEN: Categorize salary levels
    CASE
        WHEN salary >= 80000 THEN 'High'
        WHEN salary BETWEEN 50000 AND 79999 THEN 'Medium'
        ELSE 'Low'
    END AS salary_level,

    -- 2. IF(): Check if department is 'Sales', then return 'Yes', otherwise 'No' (
    IF(department = 'Sales', 'Yes', 'No') AS is_sales_department,

    -- 3. NULLIF(): Set salary to NULL if it is equal to 50000
    NULLIF(salary, 50000) AS adjusted_salary

FROM Employees;
```

7. Conversion Function

CAST() - Converts data from one type to another.
CONVERT() - Similar to CAST, but with formatting options in some systems.
TO_DATE() - Converts a string to a date (Oracle).
TO_CHAR() - Converts a date or number to a string (Oracle).
TO_NUMBER() - Converts a string to a number (Oracle).
STR() - Converts a number to a string (SQL Server).
TRY_CAST() - Attempts to convert data; returns NULL if conversion fails (SQL Server).
TRY_CONVERT() - Attempts to convert data with formatting options (SQL Server).
FORMAT() - Formats a value as a string (SQL Server).
DATE_FORMAT() - Formats a date as a string (MySQL).
PARSE() - Parses a string and converts it to a specified data type (SQL Server).
TIMESTAMP() - Converts data to a timestamp (PostgreSQL).
TIME() - Converts a value to a time format (MySQL).
HEX() - Converts a number to a hexadecimal string (MySQL).
UNHEX() - Converts a hexadecimal string to a number (MySQL).



```sql
SELECT
    -- 1. CAST(): Convert number_value from integer to VARCHAR.
    CAST(number_value AS VARCHAR) AS cast_to_varchar,

    -- 2. CONVERT(): Convert number_value to VARCHAR (SQL Server and MySQL).
    CONVERT(VARCHAR, number_value) AS convert_to_varchar,

    -- 3. TO_DATE(): Convert date_string (in 'YYYY-MM-DD' format) to DATE (Oracle).
    TO_DATE(date_string, 'YYYY-MM-DD') AS converted_to_date,

    -- 4. TO_CHAR(): Convert current_date to CHAR (Oracle).
    TO_CHAR(CURRENT_DATE, 'DD-MON-YYYY') AS date_to_char,

    -- 5. TO_NUMBER(): Convert a string to a number (Oracle).
    TO_NUMBER('12345') AS string_to_number,

    -- 6. STR(): Convert number_value to a string (SQL Server).
    STR(number_value) AS str_conversion,

    -- 7. TRY_CAST(): Attempt to cast a potentially invalid number to INTEGER, retu
    TRY_CAST(string_value AS INT) AS try_cast_to_int,
```



```sql
    -- 9. FORMAT(): Format number_value with thousands separators (SQL Server).
    FORMAT(number_value, 'N0') AS formatted_number,

    -- 10. DATE_FORMAT(): Format current date as 'Year-Month-Day' (MySQL).
    DATE_FORMAT(CURDATE(), '%Y-%m-%d') AS formatted_date_mysql,

    -- 11. PARSE(): Parse string_value to DATE with specific format (SQL Server).
    PARSE('01/31/2023' AS DATE USING 'en-US') AS parsed_date,

    -- 12. TIMESTAMP(): Convert '2023-01-01' string to TIMESTAMP (PostgreSQL).
    TIMESTAMP '2023-01-01' AS timestamp_value,

    -- 13. TIME(): Extract time from current date and time (MySQL).
    TIME(NOW()) AS current_time_mysql,

    -- 14. HEX(): Convert a number to hexadecimal format (MySQL).
    HEX(255) AS hex_value,

    -- 15. UNHEX(): Convert hexadecimal back to integer (MySQL).
    UNHEX(HEX(255)) AS unhex_value

FROM Data;
```

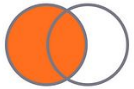
Working with Multiple Tables

Multiple tables

1. **Joins – Combining Data from Multiple Tables**
2. **Subqueries – Queries within Queries**
3. **Set Operations** – Combining Results from Multiple Queries
4. **Table Aliases** – Simplifying Table and Column Names

1. Joins

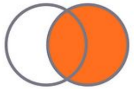
LEFT JOIN



Everything on the left
+
anything on the right that matches

```sql
SELECT *
FROM TABLE_1
LEFT JOIN TABLE_2
ON TABLE_1.KEY = TABLE_2.KEY
```

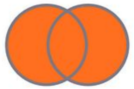
RIGHT JOIN



Everything on the right
+
anything on the left that matches

```sql
SELECT *
FROM TABLE_1
RIGHT JOIN TABLE_2
ON TABLE_1.KEY = TABLE_2.KEY
```

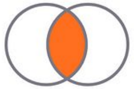
OUTER JOIN



Everything on the right
+
Everything on the left

```sql
SELECT *
FROM TABLE_1
OUTER JOIN TABLE_2
ON TABLE_1.KEY = TABLE_2.KEY
```

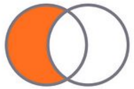
INNER JOIN



Only the things that match on the left AND the right

```sql
SELECT *
FROM TABLE_1
INNER JOIN TABLE_2
ON TABLE_1.KEY = TABLE_2.KEY
```

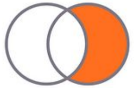
ANTI LEFT JOIN



Everything on the left that is NOT on the right

```sql
SELECT *
FROM TABLE_1
LEFT JOIN TABLE_2
ON TABLE_1.KEY = TABLE_2.KEY
WHERE TABLE_2.KEY IS NULL
```

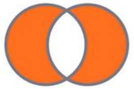
ANTI RIGHT JOIN



Everything on the right that is NOT on the left

```sql
SELECT *
FROM TABLE_1
RIGHT JOIN TABLE_2
ON TABLE_1.KEY = TABLE_2.KEY
WHERE TABLE_1.KEY IS NULL
```

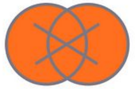
ANTI OUTER JOIN



Everything on the left and right that is unique to each side

```sql
SELECT *
FROM TABLE_1
OUTER JOIN TABLE_2
ON TABLE_1.KEY = TABLE_2.KEY
WHERE TABLE_1.KEY IS NULL
OR TABLE_2.KEY IS NULL
```

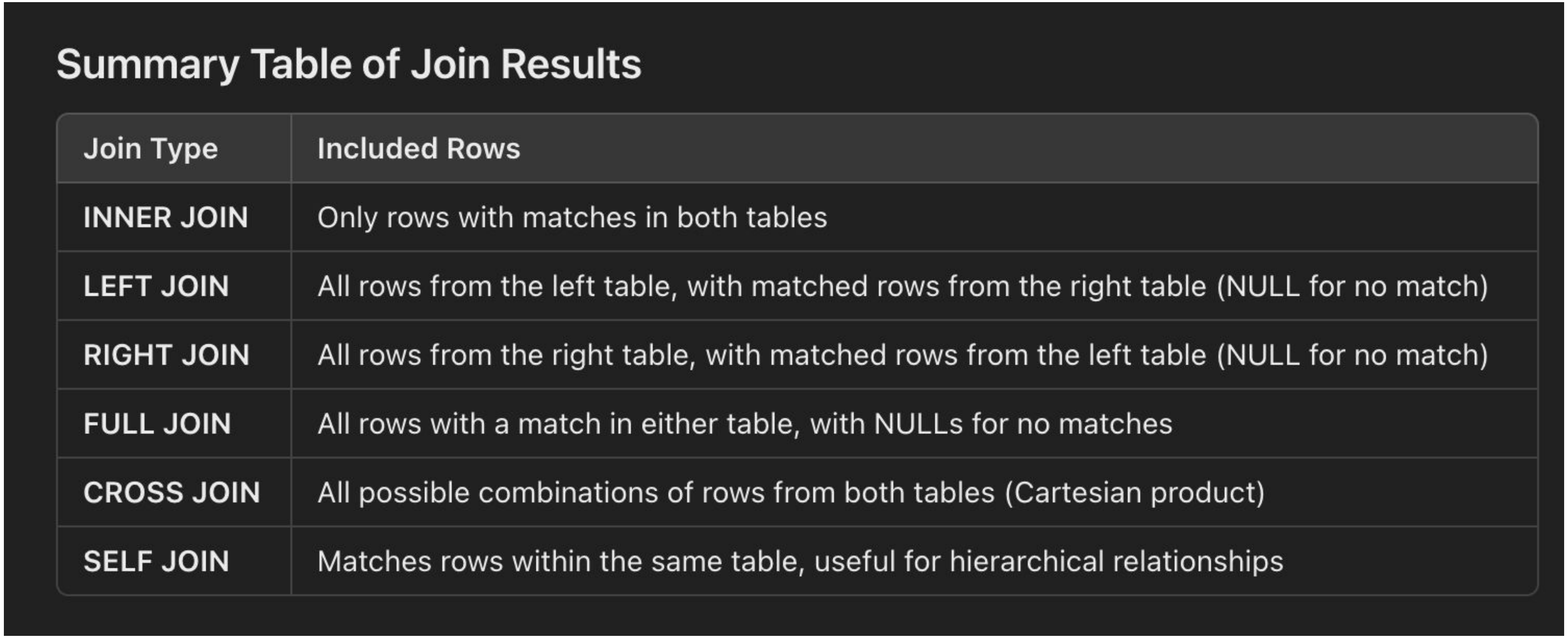
CROSS JOIN



All combination of rows from the right and the left (cartesean product)

```sql
SELECT *
FROM TABLE_1
CROSS JOIN TABLE_2
```

TABLE JOINS

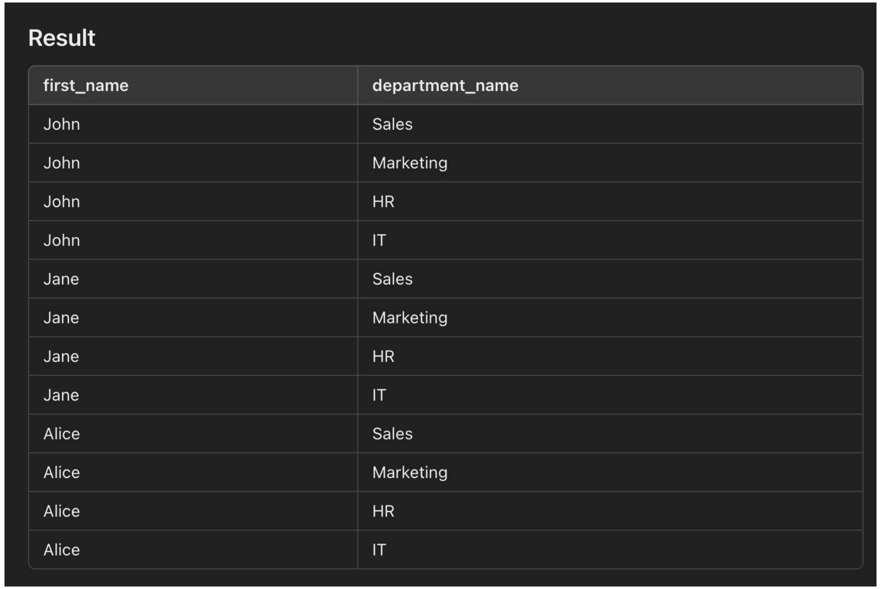


Summary Table of Join Results

Join Type	Included Rows
INNER JOIN	Only rows with matches in both tables
LEFT JOIN	All rows from the left table, with matched rows from the right table (NULL for no match)
RIGHT JOIN	All rows from the right table, with matched rows from the left table (NULL for no match)
FULL JOIN	All rows with a match in either table, with NULLs for no matches
CROSS JOIN	All possible combinations of rows from both tables (Cartesian product)
SELF JOIN	Matches rows within the same table, useful for hierarchical relationships

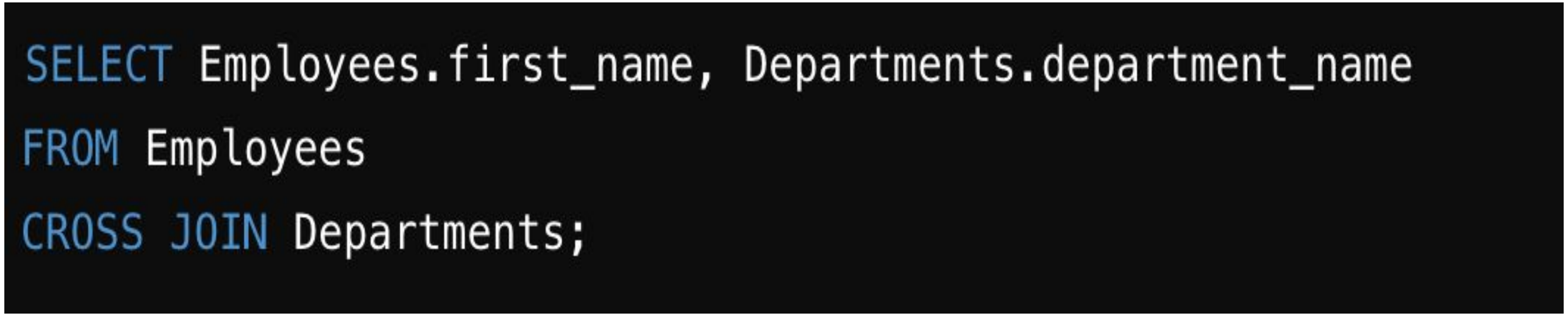
CROSS JOIN

A **CROSS JOIN** returns the Cartesian product of two tables, which means every row from the first table is paired with every row from the second table. This join can generate a large result set, as it multiplies the number of rows in both tables.



Result

first_name	department_name
John	Sales
John	Marketing
John	HR
John	IT
Jane	Sales
Jane	Marketing
Jane	HR
Jane	IT
Alice	Sales
Alice	Marketing
Alice	HR
Alice	IT

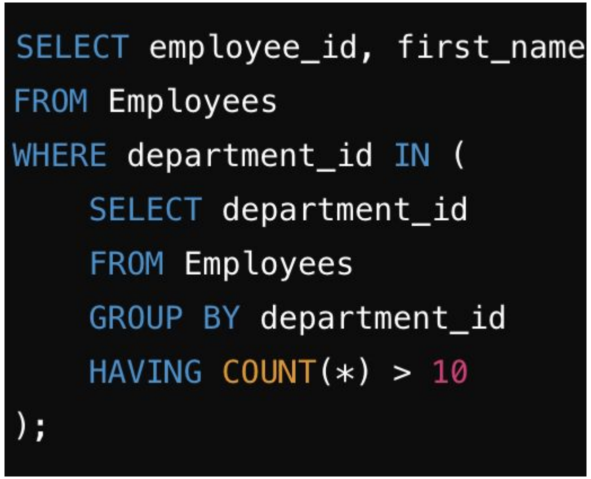


```sql
SELECT Employees.first_name, Departments.department_name
FROM Employees
CROSS JOIN Departments;
```

2. SUB QUERIES

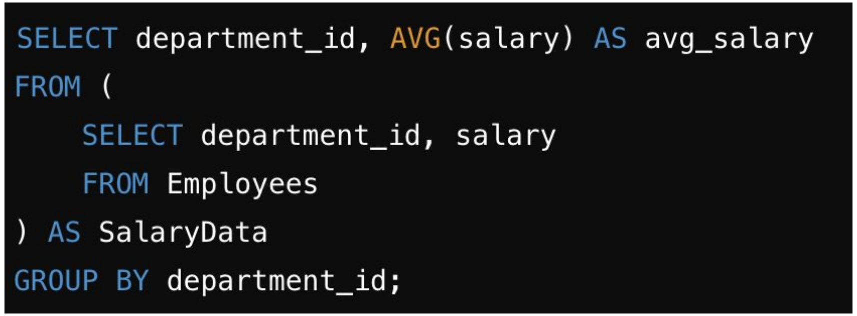
Subqueries are queries nested within another SQL query. They're useful for filtering or aggregating data from one table to apply to another. Subqueries can be used in the `SELECT`, `FROM`, `WHERE`, or `HAVING` clauses.

a) Subquery in the `WHERE` Clause . Select employees who work in a department with more than 10 employees.



```
SELECT employee_id, first_name
FROM Employees
WHERE department_id IN (
    SELECT department_id
    FROM Employees
    GROUP BY department_id
    HAVING COUNT(*) > 10
);
```

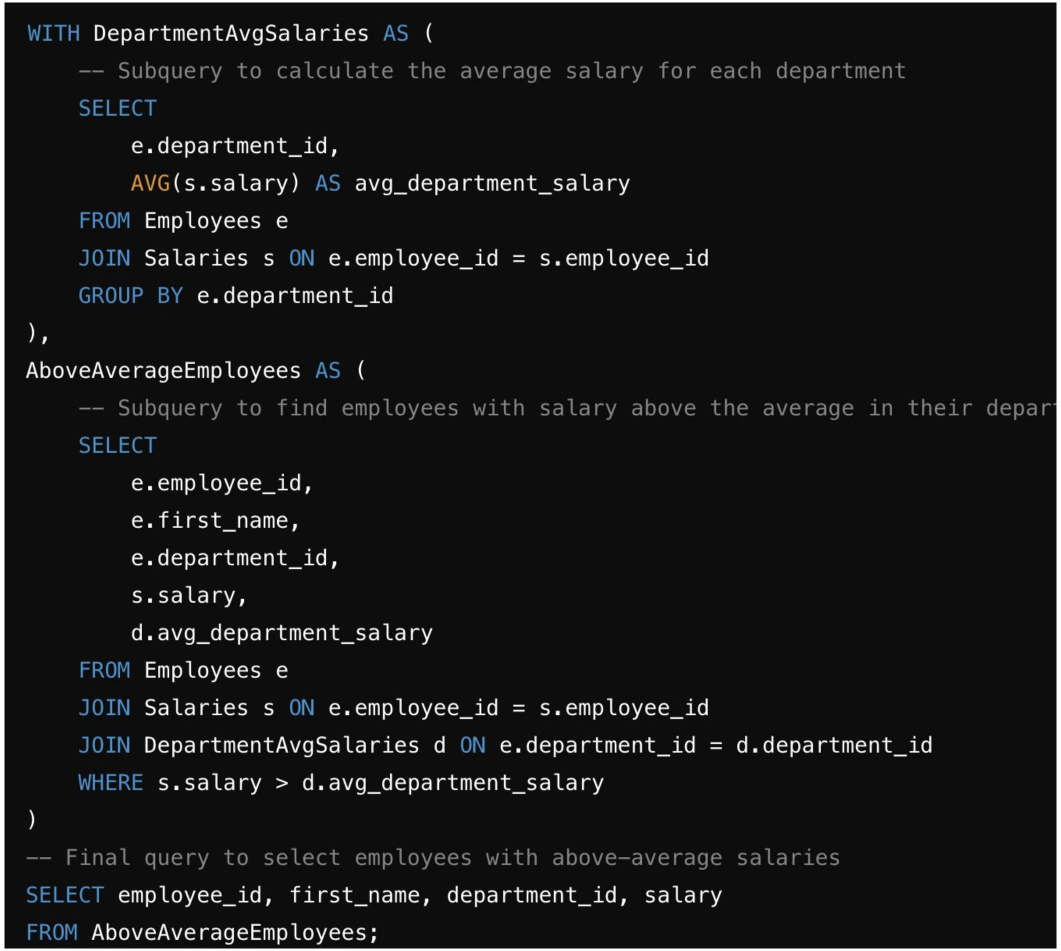
b) Subquery in the `FROM` Clause



```
SELECT department_id, AVG(salary) AS avg_salary
FROM (
    SELECT department_id, salary
    FROM Employees
) AS SalaryData
GROUP BY department_id;
```

WITH

easier to read, maintain, and debug, especially when you’re working with multiple subqueries or layers of logic.WITH CLAUSE



```sql
WITH DepartmentAvgSalaries AS (
    -- Subquery to calculate the average salary for each department
    SELECT
        e.department_id,
        AVG(s.salary) AS avg_department_salary
    FROM Employees e
    JOIN Salaries s ON e.employee_id = s.employee_id
    GROUP BY e.department_id
),
AboveAverageEmployees AS (
    -- Subquery to find employees with salary above the average in their depar
    SELECT
        e.employee_id,
        e.first_name,
        e.department_id,
        s.salary,
        d.avg_department_salary
    FROM Employees e
    JOIN Salaries s ON e.employee_id = s.employee_id
    JOIN DepartmentAvgSalaries d ON e.department_id = d.department_id
    WHERE s.salary > d.avg_department_salary
)
-- Final query to select employees with above-average salaries
SELECT employee_id, first_name, department_id, salary
FROM AboveAverageEmployees;
```

3. SET OPERATIONS

Set operations combine the results of two or more `SELECT` queries. Common set operations include `UNION`, `UNION ALL`, `INTERSECT`, and `EXCEPT` (or `MINUS` in Oracle).

a) UNION

`UNION` combines the results of two queries and removes duplicates.

b) UNION ALL

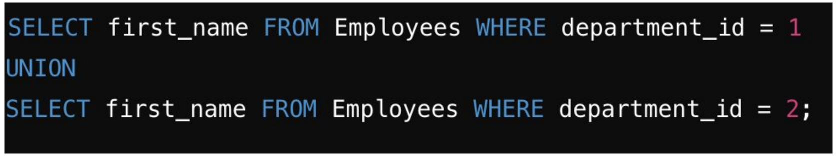
`UNION ALL` combines results without removing duplicates.

c) INTERSECT

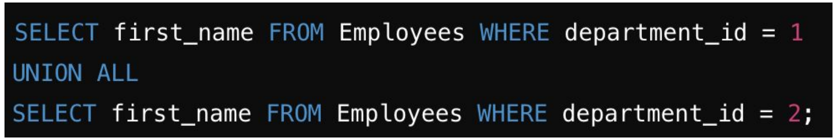
`INTERSECT` returns only rows that are common in both queries.

d) EXCEPT (or MINUS)

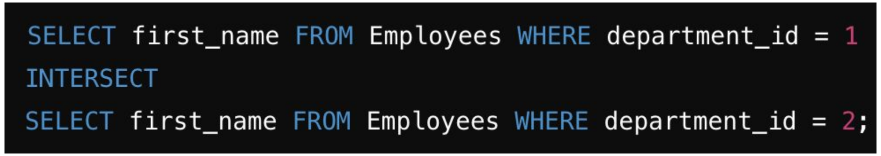
`EXCEPT` returns rows from the first query that aren't in the second query.



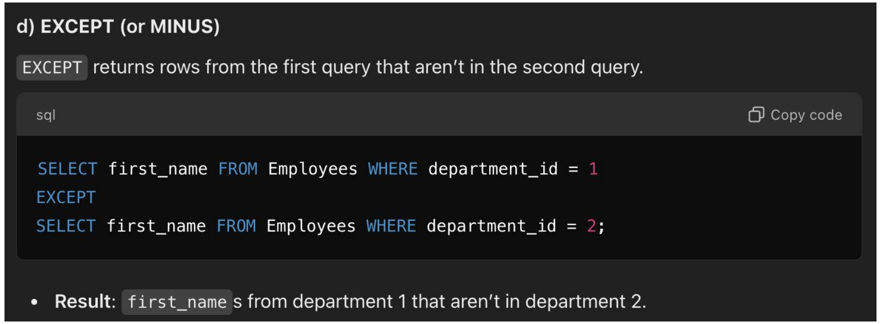
```sql
SELECT first_name FROM Employees WHERE department_id = 1
UNION
SELECT first_name FROM Employees WHERE department_id = 2;
```



```sql
SELECT first_name FROM Employees WHERE department_id = 1
UNION ALL
SELECT first_name FROM Employees WHERE department_id = 2;
```



```sql
SELECT first_name FROM Employees WHERE department_id = 1
INTERSECT
SELECT first_name FROM Employees WHERE department_id = 2;
```



d) EXCEPT (or MINUS)

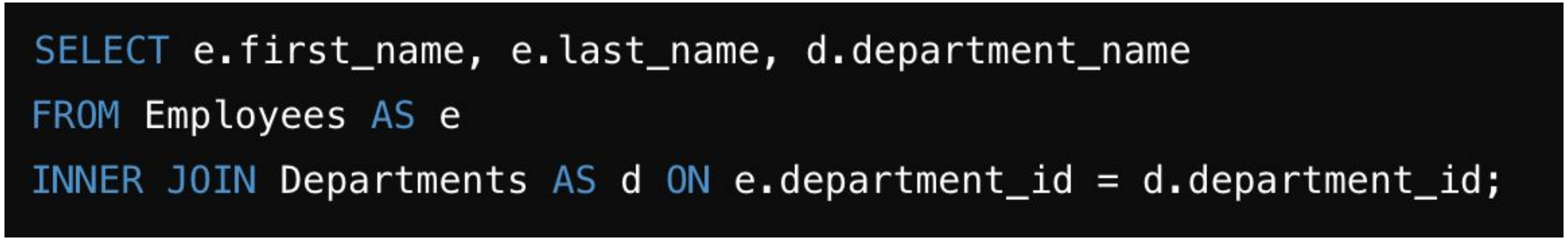
`EXCEPT` returns rows from the first query that aren't in the second query.

```sql
SELECT first_name FROM Employees WHERE department_id = 1
EXCEPT
SELECT first_name FROM Employees WHERE department_id = 2;
```

- **Result**: `first_name`s from department 1 that aren't in department 2.

4. Table Aliases

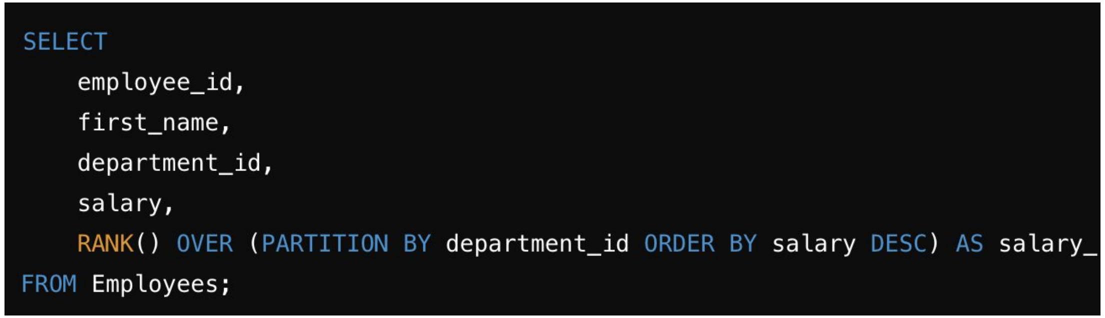
Table aliases are temporary names assigned to tables (or columns) in SQL queries to make queries easier to read, write, and understand. They are especially useful when working with multiple tables, as they allow you to reference tables and columns with shorter names instead of the full table names.



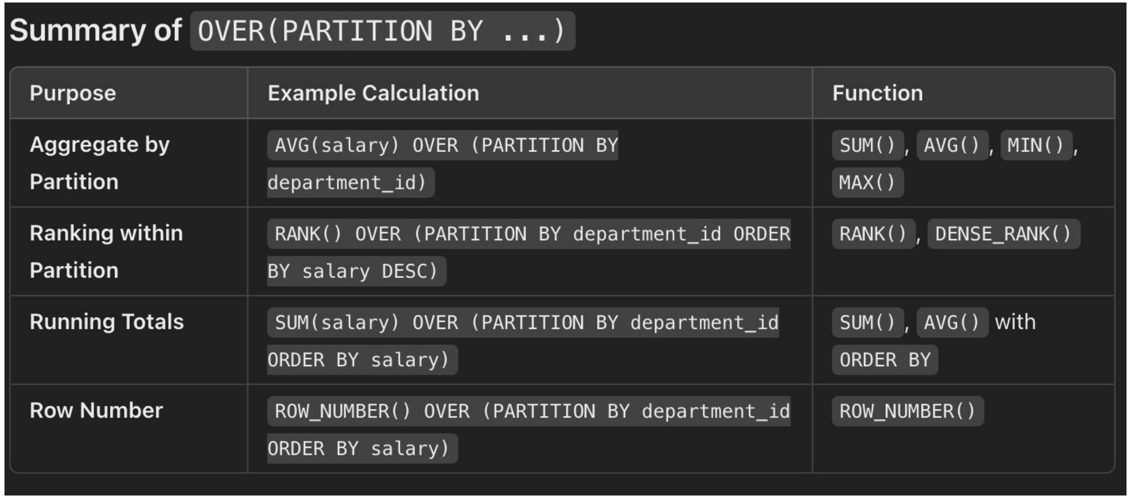
```sql
SELECT e.first_name, e.last_name, d.department_name
FROM Employees AS e
INNER JOIN Departments AS d ON e.department_id = d.department_id;
```

OVER

The `OVER(PARTITION BY ...)` clause in SQL is used in conjunction with **window functions** to perform calculations across a set of table rows that are related to the current row. By partitioning data, you can create "windows" or groups of rows over which a function (like `SUM()`, `AVG()`, `RANK()`, etc.) is calculated.



```sql
SELECT
    employee_id,
    first_name,
    department_id,
    salary,
    RANK() OVER (PARTITION BY department_id ORDER BY salary DESC) AS salary_
FROM Employees;
```



Summary of `OVER(PARTITION BY ...)`

Purpose	Example Calculation	Function
Aggregate by Partition	`AVG(salary) OVER (PARTITION BY department_id)`	`SUM()`, `AVG()`, `MIN()`, `MAX()`
Ranking within Partition	`RANK() OVER (PARTITION BY department_id ORDER BY salary DESC)`	`RANK()`, `DENSE_RANK()`
Running Totals	`SUM(salary) OVER (PARTITION BY department_id ORDER BY salary)`	`SUM()`, `AVG()` with `ORDER BY`
Row Number	`ROW_NUMBER() OVER (PARTITION BY department_id ORDER BY salary)`	`ROW_NUMBER()`