**SQL for Data Analytics Study Resource**

By Aditya Patil

**Content to include:**

1. **Introduction to SQL**
2. **SQL Basics**
   1. **Database Concepts: Tables, rows (records), and columns (fields).**
3. **Basic SQL Syntax:**
   1. **Database Concepts**
   2. **SELECT, FROM, WHERE.**
4. **SQL Functions and Clauses**
   1. **Aggregate Functions:**
      1. SUM, AVG, COUNT, MIN, MAX.
   2. **Filtering and Sorting:**
      1. WHERE clause (filtering data using conditions).
      2. ORDER BY (sorting results in ascending/descending order).
   3. **GROUP BY & HAVING:**
5. **Data Manipulation**
   1. **JOINs:**
      1. INNER JOIN, LEFT JOIN, RIGHT JOIN, FULL OUTER JOIN.
   2. **UNION & UNION ALL:**
6. **Subqueries and Aliases**
   1. **Subqueries:**
   2. **Aliases:**
7. **SQL Data Types and Constraints**
   1. **Data Types:** INT, VARCHAR, DATE, etc.
   2. **Constraints:** PRIMARY KEY, FOREIGN KEY, NOT NULL, UNIQUE.
8. **Data Analysis with SQL**
   1. **Window Functions:**
   2. **Case Statements:**
9. **SQL Performance Optimization (Optional for Advanced Learners)**
   1. **Introduction to basic performance tuning techniques:**
      1. Indexing.
      2. Query optimization tips.
10. **Shortcuts and Best Practices**

**Introduction**

Structured query language (SQL) is a programming language for storing and processing information in a relational database. A relational database stores information in tabular form, with rows and columns representing different data attributes and the various relationships between the data values. You can use SQL statements to store, update, remove, search, and retrieve information from the database. You can also use SQL to maintain and optimize database performance.

**Why is SQL important?**

Structured query language (SQL) is a popular query language that is frequently used in all types of applications. Data analysts and developers learn and use SQL because it integrates well with different programming languages. For example, they can embed SQL queries with the Java programming language to build high-performing data processing applications with major [SQL database systems](https://aws.amazon.com/what-is/sql-database/) such as Oracle or MS SQL Server. SQL is also fairly easy to learn as it uses common English keywords in its statements

**Why it’s crucial for data analytics?**

SQL (Structured Query Language) plays a fundamental role in data analytics by enabling efficient data retrieval, transformation, and reporting from large relational databases. It is widely used in industries such as e-commerce, finance, healthcare, and business intelligence for making data-driven decisions. Below are the key reasons why SQL is essential for data analytics

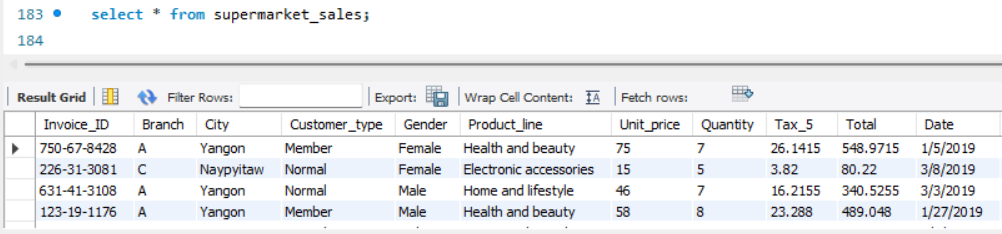
* Efficient Data Retrieval - Analyzer extracts highly relevant data from thousands of terabytes of data without laborious efforts. In simple queries, users can filter, sort, and retrieve specific information, which will make the analysis efficient.
* Data Cleaning & Transformation (ETL) - The original data is, most of the time, incomplete with highly inconsistency or repetition and lack of most values. SQL serves the data cleaning to be applicable for missing values, erroneous records, etc., and transformations, which form the main ingredients for further statistical analysis and reporting.
* Aggregation and Business Insights - SQL can also be used for it to summarize data through its helper functions like SUM(), AVG(), COUNT(), GROUP BY, by which a business will understand how their trends are further with customer activity, financial performance over time, and overall picture shape.
* Joins Multi-row Tables - As is known, the tables in which relational databases hold the data are multiple ones. To merge the data, such as customer information with purchase history, it becomes an efficient way of joining them in the SQL join operations which is an essential part of a comprehensive report.
* Integration with BI tools-SQL is also most probably used with Power BI, Tableau, and Google Data Studio to make dashboards interactive and report out performance. This ensures speed and comprehending performance visually, as the data that will be visualized is already preprocessed using SQL.
* Performance Optimization & Scalability - With large datasets, performance improvements can also be done on query performance through indexes, views, and stored procedures, providing fast, efficient retrieval and less load on the system. Certainly, this is useful, particularly in analyzing big data.
* Decision Making & Predictive Analysis- The SQL Trend Analysis, Revenue Prediction, and Customer Segmentation are indeed strong tools allowing business intelligence for internal decision-making based on data informed insights.

Describe the role of SQL in querying databases and how it helps in retrieving, manipulating, and analyzing data.

SQL (Structured Query Language) plays a crucial role in managing and interacting with relational databases. It provides a standardized way to **retrieve, manipulate, and analyze** data, making it an essential tool for businesses, analysts, and developers. SQL enables efficient data handling by allowing users to query large datasets, transform raw information, and generate meaningful insights.

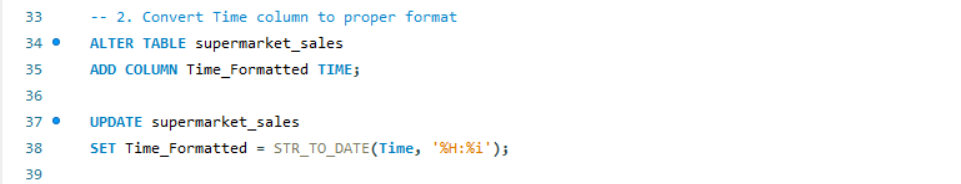
1. **Retrieving Data from Databases**

SQL is primarily used to fetch specific information from large datasets stored in relational databases. The **(SELECT)** statement enables users to extract relevant records based on certain conditions, reducing manual efforts and ensuring accuracy.



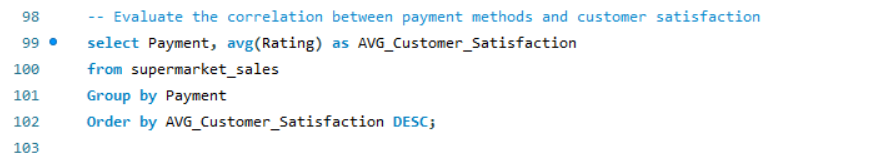
1. **Manipulating and Updating Data**

SQL allows users to **insert, update, and delete** records efficiently. This is essential for maintaining **data integrity** and keeping information up to date.



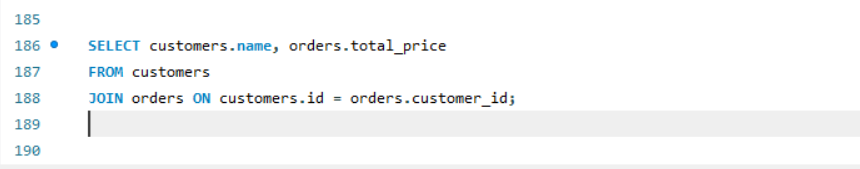
1. **Data Aggregation and Analysis**

SQL provides functions **(SUM(), AVG(), COUNT(), GROUP BY)** to perform **statistical calculations** on large datasets. This helps businesses analyze trends, customer behavior, and sales performance.



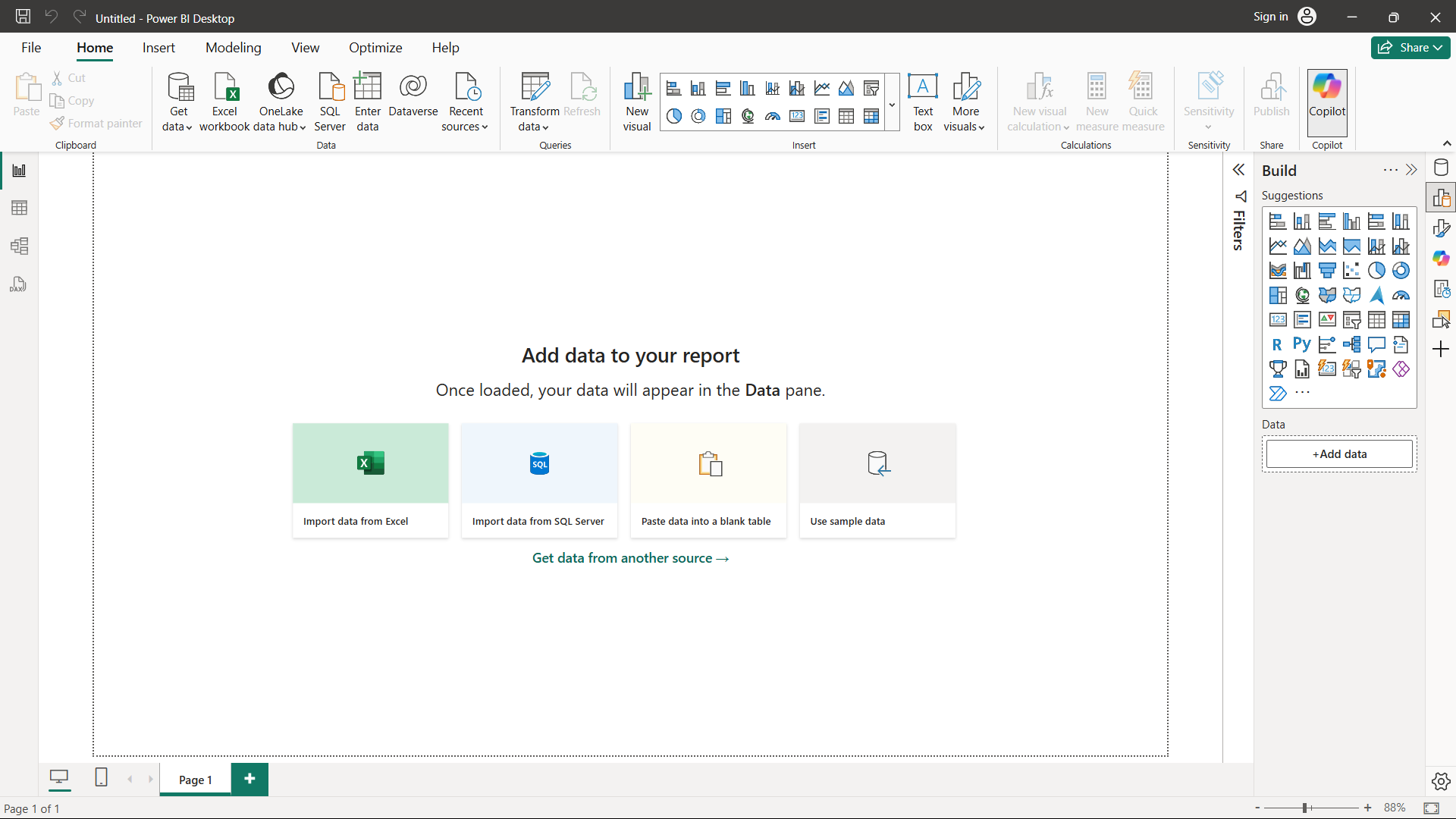
1. **Combining Data with Joins**

Data is often stored across multiple tables in a relational database. SQL **JOIN** operations help merge data from different tables, enabling comprehensive analysis.



1. **Powering Business Intelligence & Reporting**

SQL is commonly used in **Power BI, Tableau, and Google Data Studio** for data preprocessing and visualization. By running SQL queries, analysts can prepare datasets before feeding them into BI tools, ensuring optimized performance and better insights.



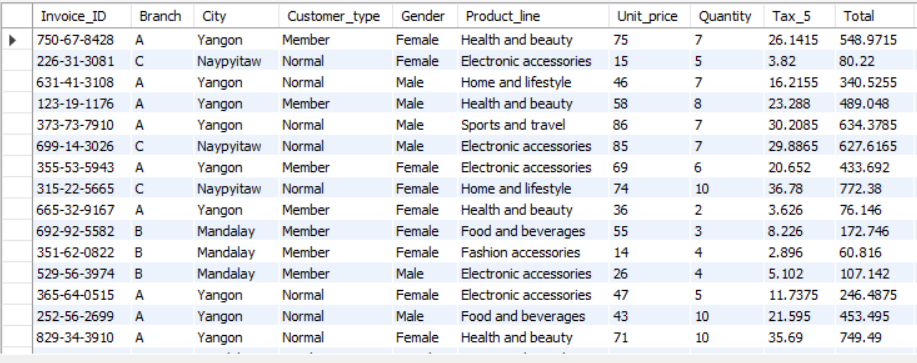
**SQL Basics**

SQL (Structured Query Language) is the foundation of database management. It allows users to interact with relational databases by retrieving, modifying, and managing structured data efficiently. Below are the fundamental concepts of SQL that every beginner should know.

Field

Columns

**Table:**

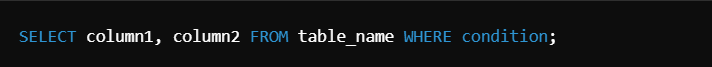


Rows

**Basic SQL Commands**

SQL is used to retrieve specific data from a database using the SELECT statement. The basic structure of a SQL query consists of three key components:

1. SELECT – Specifies the columns to retrieve.
2. FROM – Specifies the table from which to retrieve data.
3. WHERE – Filters data based on conditions (optional).
4. **Data Query Language (DQL) – Retrieving Data**

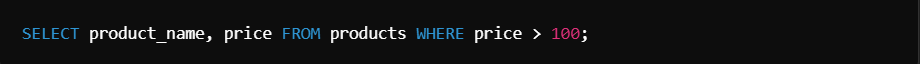
The SELECT statement is used to fetch data from a database.

**Example:**

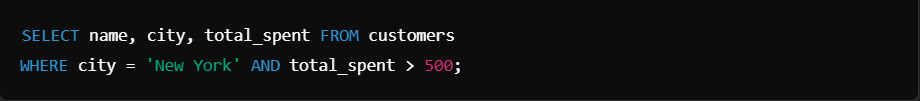
****

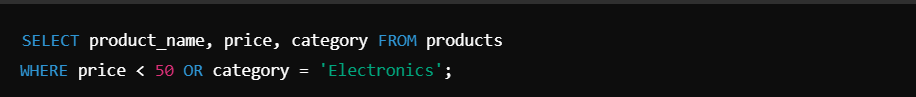
1. **Filtering Data Using WHERE**

****



1. **Using Multiple Conditions with AND & OR**

The **AND** operator ensures **both conditions** must be met. 

The **OR** operator allows **either condition** to be true.

**SQL Functions and Clauses**

1. **Aggregate Functions**:

Aggregate functions in SQL perform calculations on a group of rows and return a **single value**. They are commonly used for summarizing data, such as calculating totals, averages, and identifying minimum or maximum values. These functions are essential in **data analysis** and **reporting**.

* **SUM() – Calculating Total Values**

The **SUM** function adds the values in a column. It is useful for calculating totals like **total sales**, **total revenue**, or **total quantities**.

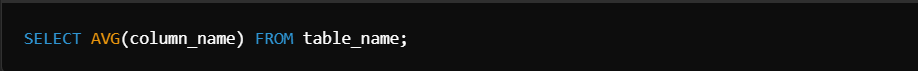


Example

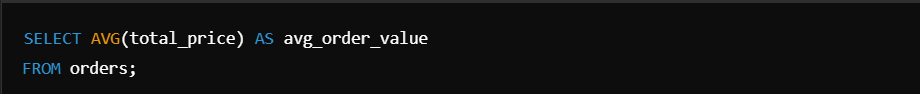


* **AVG – Calculate Average**

The **AVG** function returns the average value of a numeric column, perfect for calculating average ratings, prices, or order values.

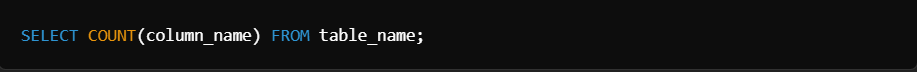


Example



* **COUNT – Count Rows**

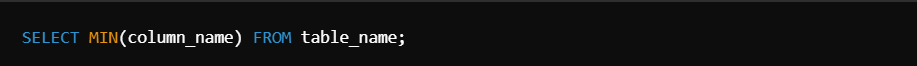
The **COUNT** function counts the number of rows that match a condition or exist in a table.



Example

* **MIN – Find Minimum Value**

The **MIN** function returns the smallest value in a column, often used to find the lowest price, earliest date, or smallest quantity.



Example

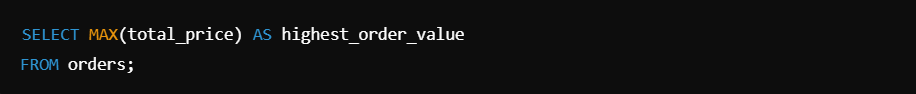
****

**MAX – Find Maximum Value**

The **MAX** function returns the largest value in a column, useful for identifying the highest sales, latest dates, or biggest orders.



Example



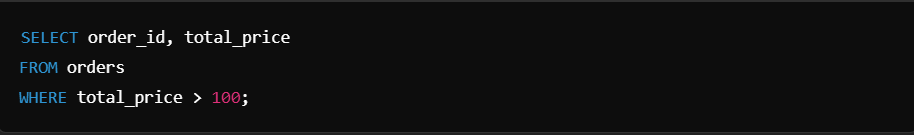
1. **Filtering and Sorting**:

* Filtering Data Using WHERE Clause

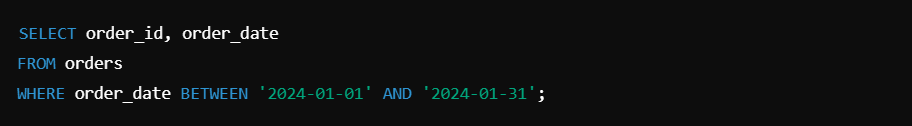
The **WHERE** clause is used to filter rows based on conditions. It allows you to **retrieve only the data that meets specific criteria**.



Example



* The BETWEEN operator filters rows **within a range**.



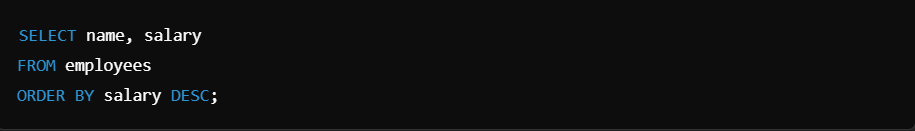
* **Sorting Results Using ORDER BY**

The ORDER BY clause sorts query results in ascending (ASC) or descending (DESC) order.

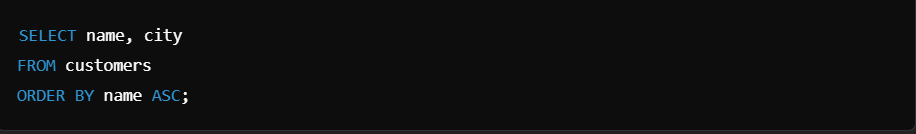


Example

* The DESC keyword sorts salaries from **highest to lowest**.



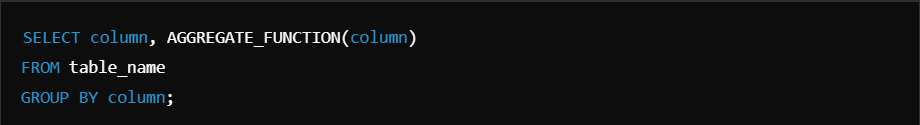
* The ASC keyword sorts customer names **from A to Z**.



1. **GROUP BY & HAVING**:

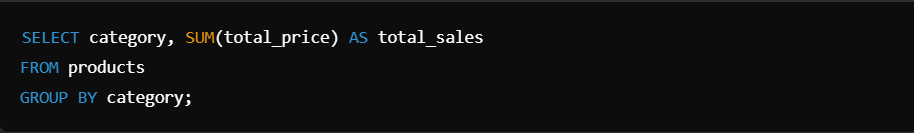
* Grouping Data Using GROUP BY

The GROUP BY clause **groups rows with the same values** into summary rows. It is often used with **aggregate functions** (SUM, AVG, COUNT, MIN, MAX).

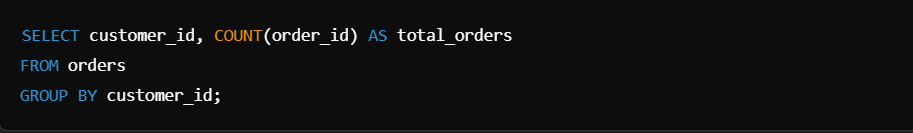


Example

* This groups sales by **category** and calculates total sales for each.



* This returns **the number of orders each customer has placed**.



**Data Manipulation**

Data manipulation is the process of changing or transforming data to make it more useful. It involves operations like filtering, sorting, aggregating, merging, and transforming data. 

1. **JOINs**:

In relational databases, data is often **stored across multiple tables**. SQL **JOINs** allow us to combine this data based on a common key, making analysis more powerful

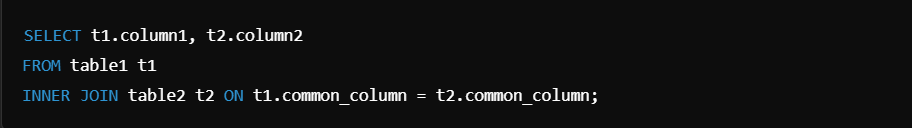
**Common use case:**

* Combining **customer** and **order** data
* Linking **employees** with their **departments**
* Merging **product** details with **sales transactions**

**Types:**

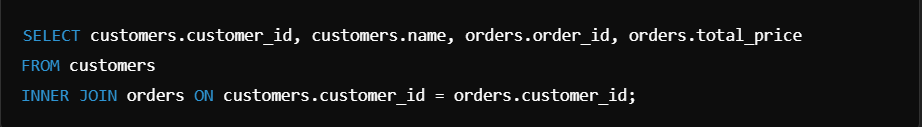
* **INNER JOIN (Only Matching Rows)**

**Returns only records where there’s a match in both tables.**



Example

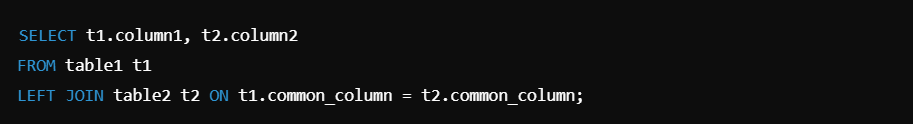
This adds **David** to the customers table.





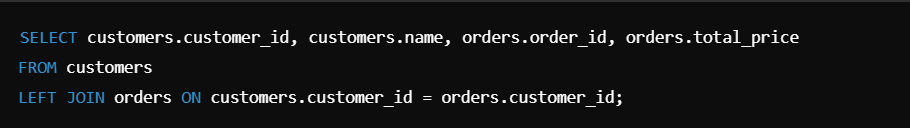
* **LEFT JOIN (All Left Table + Matching Right Table Records)**

**Returns all rows from the left table + matching rows from the right table (NULL if no match).**



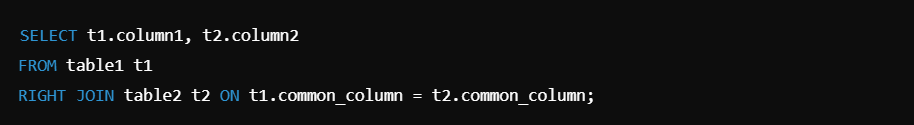
Example

This updates **John's email** where customer\_id = 1

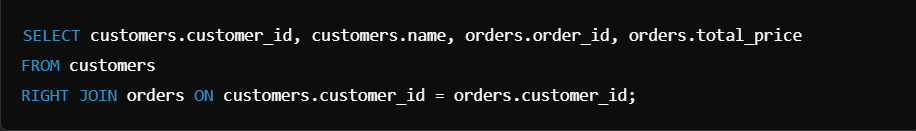




* **RIGHT JOIN (All Right Table + Matching Left Table Records)**

Returns all rows from the right table + matching rows from the left table (NULL if no match). 

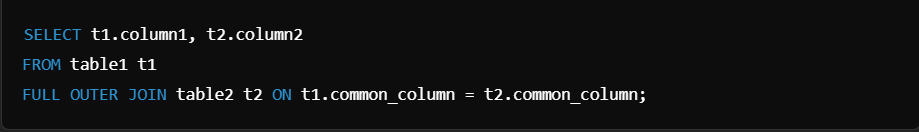
Example



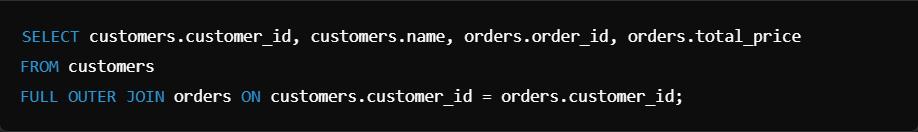


* **FULL OUTER JOIN (All Records from Both Tables)**

Returns all rows from both tables. If there’s no match, NULLs are placed where data is missing.



Example





SQL JOINs allow powerful **data merging** across tables, crucial for analytics and reporting.

* Use **INNER JOIN** when you need **only matching records**.
* Use **LEFT JOIN** to include **all left table records** (even without matches).
* Use **RIGHT JOIN** when you need **all right table records**.
* Use **FULL OUTER JOIN** when you want **everything** from both tables.



1. **UNION & UNION ALL**:

In SQL, UNION and UNION ALL are used to combine results from multiple SELECT statements into a single result set.

* **UNION**

**UNION removes duplicate rows** from the combined result.  
The column **names and data types** in all SELECT statements must be the same.



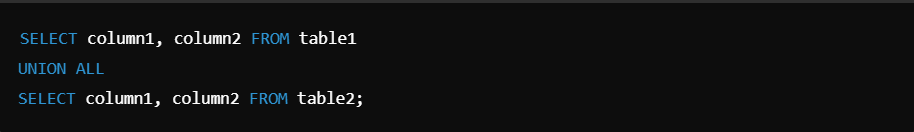
Example



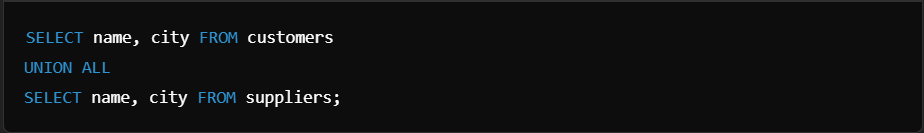


* **UNION ALL**

UNION ALL includes all rows, even duplicates.  
Faster than UNION because it does not check for duplicates



Example



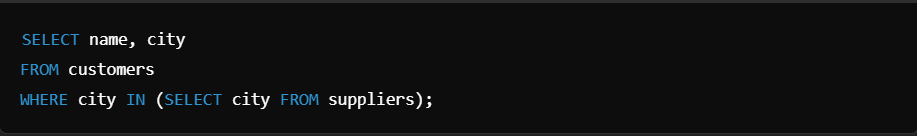


**Subqueries and Aliases**

1. **SUBQUERIES:**

A **subquery** is a query that is nested inside another SQL query. It helps **filter, calculate, or retrieve data** dynamically based on another query’s results.

* A **subquery (inner query)** runs first and passes its result to the **main query (outer query)**.
* It is used inside SELECT, FROM, or WHERE clauses.
* Subqueries must be enclosed in **parentheses ()**.

****

* Inner Query (SELECT city FROM suppliers) retrieves a list of cities where suppliers exist.
* Outer Query (SELECT name, city FROM customers) returns customers only if their city is in the supplier list.

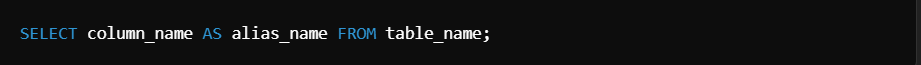


1. **ALIASES**:

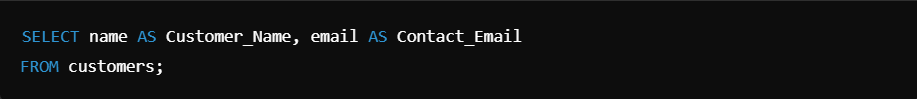
SQL **aliases** allow you to rename **columns or tables** in queries, making them more readable and organized.

* **Column Aliases**

**Used to rename columns** in the output.  
**Does not change the actual column name** in the database.  
**Use AS keyword** (optional) or just a space.



Example



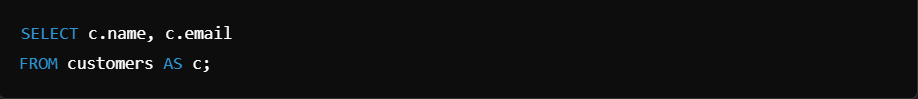


* **Table Aliases**

**Used to rename tables** to make queries shorter and easier to read.  
**Helpful in JOINs** when working with multiple tables.



Example





**SQL Data Types and Constraints**

* **DATA TYPES**

Each column in a table must have a **specific data type** to define what kind of data it can store.

|  |  |  |
| --- | --- | --- |
| Data Type | Description | Example |
| INT | Stores whole numbers | 100, 250, -75 |
| VARCHAR(n) | Stores text with a max length n | 'John Doe' |
| CHAR(n) | Stores fixed-length text | 'USA' (CHAR(3)) |
| TEXT | Stores long text | 'This is a long paragraph…' |
| DATE | Stores a date (YYYY-MM-DD) | 2025-02-14 |
| DATETIME | Stores date & time | 2025-02-14 12:30:00 |
| DECIMAL(p,s) | Stores decimal numbers | DECIMAL(10,2) → 99.99 |
| BOOLEAN | Stores TRUE or FALSE values | 1 (TRUE), 0 (FALSE) |

Example



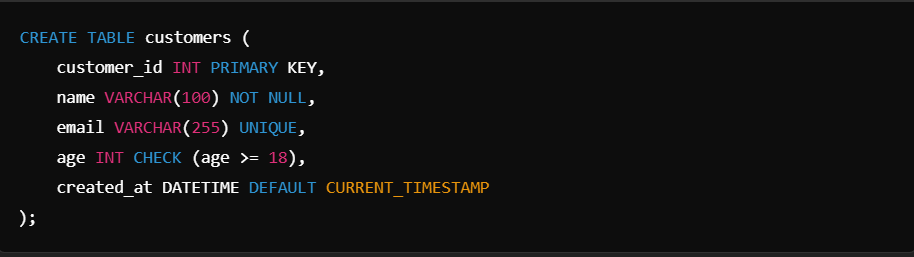


* **CONSTRAINTS**:

Constraints enforce **rules** on the data to maintain its accuracy and integrity.

|  |  |  |
| --- | --- | --- |
| Constraint | Description | Example |
| PRIMARY KEY | Uniquely identifies each row | employee\_id INT PRIMARY KEY |
| FOREIGN KEY | Links to a column in another table | FOREIGN KEY (dept\_id) REFERENCES departments(dept\_id) |
| NOT NULL | Ensures the column cannot be empty | name VARCHAR(100) NOT NULL |
| UNIQUE | Ensures all values in a column are unique | email VARCHAR(255) UNIQUE |
| CHECK | Ensures values meet a condition | CHECK (salary > 0) |
| DEFAULT | Sets a default value if none is provided | status VARCHAR(10) DEFAULT 'Active' |

Example





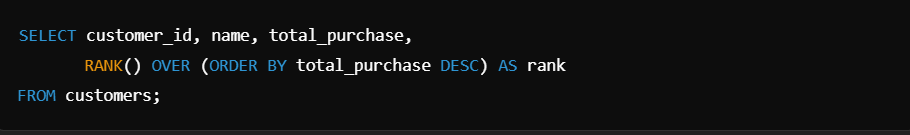
**DATA ANALYSIS WITH SQL**

* **Window Functions:**

Window functions perform calculations across a set of table rows related to the current row.  
Unlike GROUP BY, window functions do not collapse rows.  
Used for ranking, running totals, moving averages, and more.

|  |  |
| --- | --- |
| Function | Description |
| ROW\_NUMBER() | Assigns a unique row number to each record in a partition |
| RANK() | Assigns a rank with gaps (e.g., 1, 2, 2, 4) for ties |
| DENSE\_RANK() | Assigns a rank without gaps (e.g., 1, 2, 2, 3) |
| NTILE(n) | Divides rows into n equal-sized groups |
| LAG() | Retrieves the value from the previous row |
| LEAD() | Retrieves the value from the next row |

Example



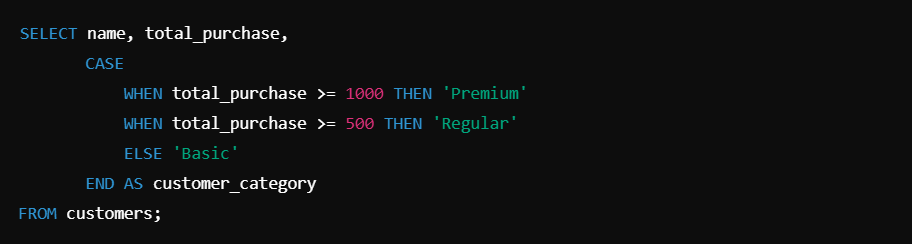


* **Case Statements**

**CASE is used for conditional logic** within SQL queries.  
Works like IF-ELSE statements in programming.  
Often used for **categorizing, transforming, or filtering** data.



Example





**SQL PERFORMANCE OPTIMIZATION**

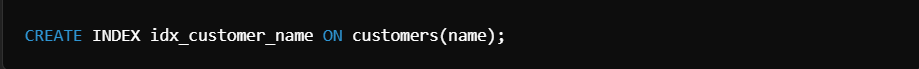
* **Indexing**

**Indexes store a sorted version of column values**, making searches much faster.  
Without indexes, SQL must scan the entire table (Full Table Scan), which is slow.  
**Primary keys and foreign keys** automatically create indexes, but additional indexes can be added manually.

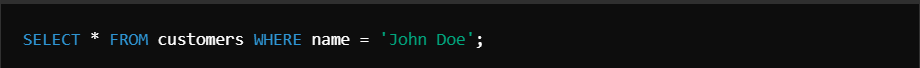
**Types of Indexes**

|  |  |  |
| --- | --- | --- |
| Index Type | Description | Use Case |
| B-Tree Index (Default) | Speeds up equality and range searches | WHERE column = value or BETWEEN |
| Unique Index | Ensures column values are unique | email in a users table |
| Full-Text Index | Used for searching large text fields | Searching in articles or blogs |
| Composite Index | Index on multiple columns | ORDER BY last\_name, first\_name |

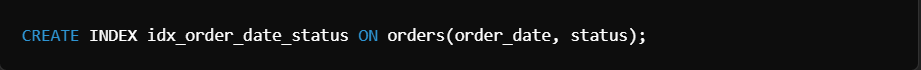
Creating an Index on a Column



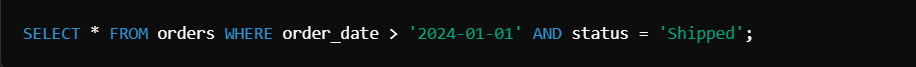
Example



Composite Index for Faster Sorting & Filtering



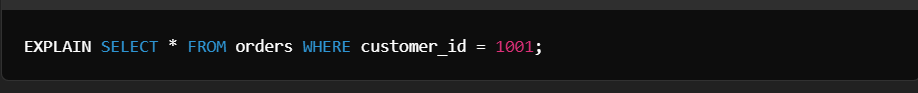
Example





**Query Optimization**

1. **Use EXPLAIN to Analyze Queries**  
   EXPLAIN shows how SQL executes a query, helping find bottlenecks.





1. **Avoid SELECT \* (Specify Columns Instead)**

**Bad Practice** (Slow & Uses More Memory)



**Better** (Only Fetch Needed Data)





1. **Use LIMIT for Large Queries**

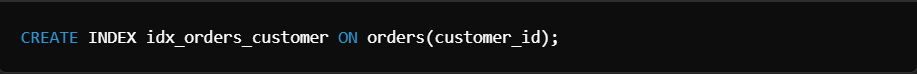
Limiting rows reduces memory usage in large datasets.



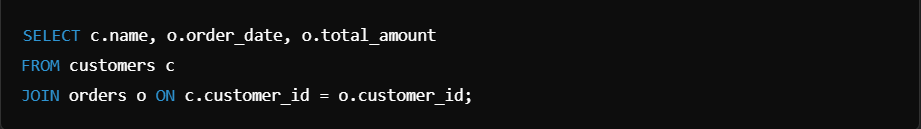


1. **Optimize JOINs with Proper Indexing**

Always index foreign keys in JOIN queries



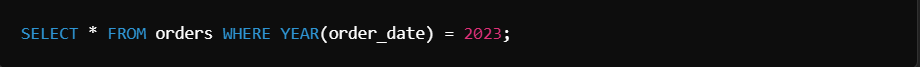
Optimized JOIN query



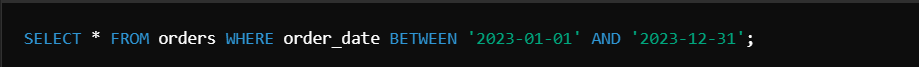


1. **Avoid Using Functions in WHERE Clauses**

Bad Practice (Prevents Index Usage)



Better (Uses Indexes)

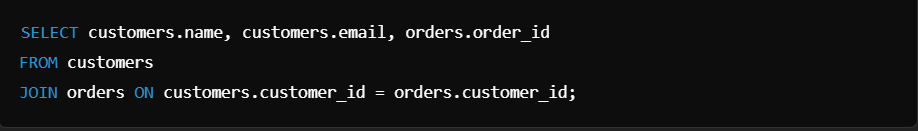




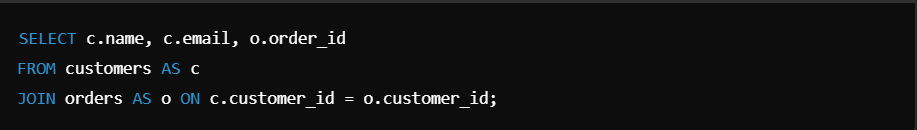
**SQL Shortcuts & Writing Tips**

1. **Use Table Aliases for Cleaner Queries**

Instead of writing long table names multiple times:  
**Without Alias (Hard to Read)**



**With Alias (Cleaner & Shorter)**

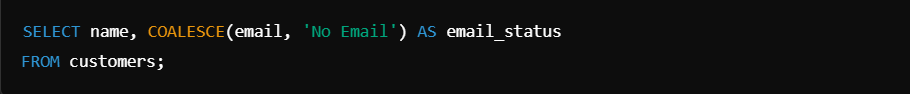


**Tip:** AS for table aliases is optional but improves readability.



1. **Use COALESCE() to Handle NULL Values**

Instead of checking NULL manually:

****

**If email is NULL, it will display "No Email" instead.**

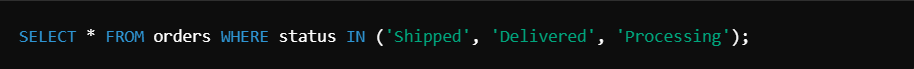


1. **Use IN Instead of Multiple OR Conditions**

Long & Inefficient:

****

**Better Using IN (Shorter & Faster):**



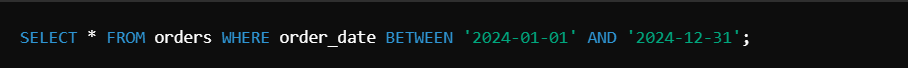


1. **Use BETWEEN for Ranges Instead of >= and <=**

**Less Readable:**



**Better Using BETWEEN:**





1. **Avoid Using SELECT \* (Specify Columns Instead)**

**Bad (Slow & Wastes Memory):**

****

**Good (Optimized & Faster):**

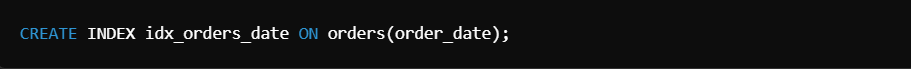


**Why?** Fetching unnecessary columns increases processing time.



1. **Index Frequently Queried Columns**

Indexing **speeds up** searches on frequently used columns:

****

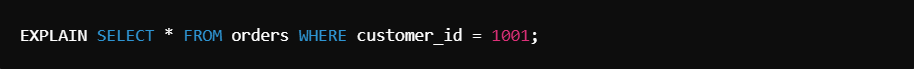
**When to Avoid Indexing?**

* On **small tables** (Indexing overhead slows queries).
* On **frequently updated columns** (Slows INSERT and UPDATE).



1. **Use EXPLAIN to Analyze Query Performance**

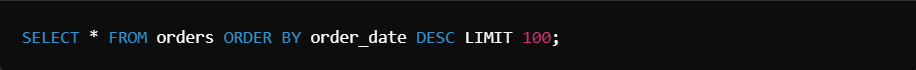
Check if SQL is using indexes efficiently:





1. **Use LIMIT for Faster Query Execution**

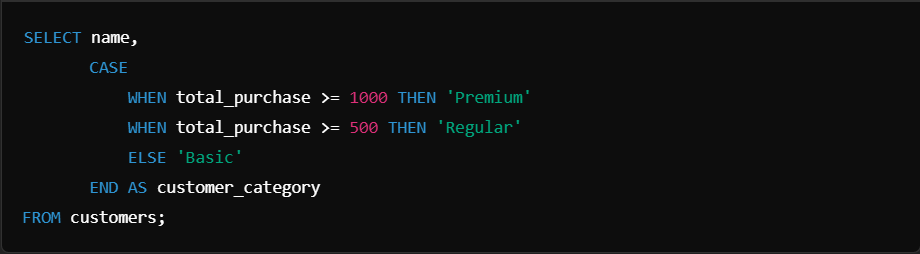
When analyzing large datasets, **limit the number of rows**:





1. **Use CASE for Conditional Logic**

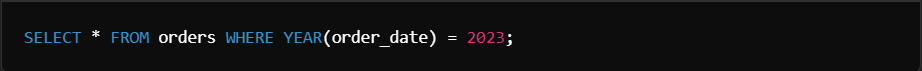
Instead of multiple queries, categorize data using CASE:





1. **Avoid Using Functions on Indexed Columns in WHERE**

**Bad (Prevents Index Usage & Slows Query)**



**Good (Allows Index Usage)**

