1. 📂 Database Basics (The Theory)

Before writing code, it helps to understand what you're working with.

- **Database:** A structured collection of data. Think of it as a huge digital filing cabinet.
- **Table:** A collection of related data organized in **rows** and **columns**. This is like an individual spreadsheet file within the filing cabinet (database).
    - **Row (Record/Tuple):** A single entry in a table, containing data for all columns. For example, a single customer's information.
    - **Column (Field/Attribute):** Defines the type of data stored. For example, the Customer_Name column or the Price column.
- **Relational Database:** A database where tables are linked, or **related**, to each other through common columns (keys). This prevents data duplication and keeps the data consistent.

  Example: You have a Customers table and an Orders table. They are related by a common column: Customer_ID. This is a core concept in **Relational Database Theory**.

---

2. 📝 Composing Basic Queries (SELECT & FROM)

The **SELECT** statement is how you ask the database for information. It's the most common operation.

SELECT and FROM

Keyword	Purpose	Example	Explanation
SELECT	Specifies **which columns** you want to see.	SELECT Name, Price	I want the data from the Name column and the Price column.
FROM	Specifies **which table** the columns are in.	FROM Tracks	Get this data from the Tracks table.
* **(Asterisk)**	A shortcut to select **all** columns.	SELECT *	Show me every column in the table.

Basic Query Example:

SQL

```
SELECT FirstName, LastName
FROM Customers;
```

Goal: Show the first name and last name of every customer in the Customers table.

Other Basic Clauses

Clause	Purpose	Example	Explanation
ORDER BY	Sorts the results based on one or more columns. Default is **ascending (ASC)**. Use **DESC** for descending.	ORDER BY LastName DESC	Sort the results so the last names go from Z to A.
LIMIT	Restricts the number of rows returned. Useful for quick checks or top-N reports.	LIMIT 10	Only show the first 10 rows of the result.

Sorting Example:

SQL

```
SELECT Name, AlbumId
FROM Tracks
ORDER BY Name ASC
LIMIT 5;
```

Goal: Show the first 5 track names, sorted alphabetically, along with their album ID.

Column Custom Names (Aliases)

You can give a column a temporary, more readable name using the **AS** keyword (creating an **alias**).

- **Syntax:** SELECT OriginalColumnName AS NewColumnName

Example:

SQL

```
SELECT Total AS OrderTotal, InvoiceDate AS Date
FROM Invoices;
```

Goal: Display the Total column but call it "**OrderTotal**" in the result, and display the InvoiceDate column but call it "**Date**".

---

3. 🔍 Discovering Insights (Filtering Data)

Filtering data means selecting only the rows that meet a specific condition. This is done using the **WHERE** clause.

The WHERE Clause and Operators

The WHERE clause comes after FROM and uses **operators** to define the condition.

Operator Type	Operator	Meaning	Example Condition
Comparison	=, $>, <, >=, <=, <>$ (or $!=$)	Equals, greater than, less than, not equals, etc.	WHERE UnitPrice > 0.99
Text Search	**LIKE**	Finds text that matches a pattern (use % as a wildcard for any number of characters).	WHERE Name LIKE 'B%'
Range	**BETWEEN**	Selects values within a specified range (inclusive).	WHERE Total BETWEEN 1.00 AND 5.00
List	**IN**	Selects values that match any value in a list.	WHERE Country IN ('USA', 'Canada')
Missing	**IS NULL**	Checks if a value is missing (blank/unknown).	WHERE Fax IS NULL

Filtering Example:

SQL

```
SELECT Name, UnitPrice
FROM Tracks
WHERE UnitPrice < 1.00 AND Name LIKE 'S%';
```

Goal: Find tracks that cost less than $1.00 **AND** whose names begin with the letter 'S'.

Logical Operators (Combining Conditions)

Use AND, OR, and NOT to combine multiple conditions in the WHERE clause.

- **AND:** Both conditions must be true.
- **OR:** At least one of the conditions must be true.
- **NOT:** Negates a condition.

Operator	Precedence	Example	Explanation
AND	High	A > 10 AND B = 'Yes'	Only returns rows where A is > 10 **and** B is 'Yes'.
OR	Low	C = 5 OR D = 1	Returns rows where C is 5 **or** D is 1 (or both).

Brackets and Order Example:

SQL

```
WHERE Country = 'USA' AND (State = 'NY' OR State = 'CA');
```

Goal: Find customers in the USA **who** are in New York **or** California. The brackets ensure the OR condition is evaluated first.

CASE Statements (IF THEN Logic)

The CASE statement allows you to apply "if/then/else" logic to create a new, custom column in your query result.

- **Syntax:** CASE WHEN condition1 THEN result1 WHEN condition2 THEN result2 ELSE final_result END AS NewColumnName

Example:

SQL

```
SELECT
  Total,
  CASE
    WHEN Total >= 10.00 THEN 'High Value'
    WHEN Total >= 5.00 THEN 'Medium Value'
    ELSE 'Low Value'
  END AS OrderCategory
FROM Invoices;
```

Goal: Create a new column called OrderCategory that assigns a text description based on the numeric value of the Total column.

---

4. 🤝 Accessing Data from Multiple Tables (JOINs)

JOINs are fundamental to relational databases. They temporarily combine two or more tables based on a relationship to retrieve a complete set of data.

- **How Tables Share a Relationship:** They share a common column. For example, Employee.ReportsTo = Manager.EmployeeId.

Types of JOINs

JOIN Type	Result	Analogy
INNER JOIN	Returns only the rows that have **matching values in both tables**. (Most common)	Like finding the intersection of two circles.
` LEFT JOIN	Returns **all rows from the left table** and only the matching rows from the right table. Non-matches on the right are NULL.	Keeps everything in the first table, adds matches from the second.

INNER JOIN Syntax

You typically use an **alias** (e.g., T1 and T2) to make the query shorter and clearer.

SQL
```
SELECT T1.Name, T2.Title
FROM Tracks AS T1  -- T1 is the alias for Tracks
INNER JOIN Albums AS T2 -- T2 is the alias for Albums
  ON T1.AlbumId = T2.AlbumId; -- The condition/key linking them
```

> **Goal:** Find the Name of every track (from Tracks table) and the Title of the album it belongs to (from Albums table), combining them on the shared AlbumId.

---

5. 🔢 SQL Functions (Calculations & Transformations)

SQL Functions perform calculations or transform data (like text or dates).

Aggregate Functions (Calculating across many rows)

These functions are used to calculate a single summary value from a set of rows.

Function	Purpose	Example	Result
COUNT()	Counts the number of rows or non-NULL values.	COUNT(InvoiceId)	The total number of invoices.
SUM()	Calculates the total sum of numeric values.	SUM(Total)	The grand total of all invoice amounts.
AVG()	Calculates the average (mean) of numeric values.	AVG(Milliseconds)	The average length of all tracks.
MIN()	Finds the smallest (minimum) value.	MIN(UnitPrice)	The lowest track price.
MAX()	Finds the largest (maximum) value.	MAX(InvoiceDate)	The date of the most recent invoice.

Example:

SQL

```
SELECT COUNT(*), AVG(Total)
FROM Invoices;
```

Goal: Count the total number of invoices and calculate the average total amount of all invoices.

Other Function Types (Transforming single values)

- **String Functions:** Manipulate text.
  - **UPPER(text)** / **LOWER(text)**: Converts text to all uppercase/lowercase.
  - **LENGTH(text)**: Returns the number of characters in the text.
  - **CONCAT(text1, text2, ...)**: Joins multiple strings together.
- **Date Functions:** Manipulate dates and times. (Specific functions vary by database type, e.g., STRFTIME in SQLite, DATEADD in SQL Server).

Example (String):

SQL

```
SELECT FirstName, LastName,
  UPPER(LastName) || ', ' || FirstName AS FullName
```

FROM Customers;

Goal: Concatenate the last name (in uppercase) and first name with a comma and space in between, creating a new FullName column. (|| is often used for concatenation).

---

6. 📊 Grouping Results

The **GROUP BY** clause is used to divide the rows into groups and then calculate an aggregate value (like a SUM or COUNT) for each group.

- **Rule:** Any column selected that is **not** an aggregate function **must** be listed in the GROUP BY clause.

Example:

SQL

```
SELECT Country, COUNT(CustomerId) AS CustomerCount
FROM Customers
GROUP BY Country
ORDER BY CustomerCount DESC;
```

Goal: Count how many customers live in **each** Country, grouping by country, and showing the country with the most customers first.

Filtering Grouped Data (HAVING)

You **cannot** use the WHERE clause to filter the results of an aggregate function. Instead, you use the **HAVING** clause, which acts like a WHERE but on the aggregated results.

- **Order of Operations:** WHERE (filters individual rows) $\rightarrow$ GROUP BY (groups remaining rows) $\rightarrow$ HAVING (filters the groups) $\rightarrow$ ORDER BY (sorts the final result).

Example:

SQL

```
SELECT AlbumId, COUNT(TrackId) AS TrackCount
FROM Tracks
GROUP BY AlbumId
HAVING COUNT(TrackId) > 10;
```

Goal: Count the number of tracks on **each** album, but **only** show the albums that have more than 10 tracks.

---

7. 🧩 Advanced Querying (Subqueries)

A **Subquery** (or inner query) is a SELECT statement nested inside another query (the outer query). They are used to perform an operation in steps.

- **Concept:** The inner query runs first, and its result is used as a value or a set of values for the outer query.

Subquery Location	Purpose	Example Use Case
SELECT Clause	Returns a single value that is treated as a column.	Display the average track price next to every track's price.
FROM Clause	The inner query's result set is treated as a temporary table.	Join a complex pre-filtered group of customers with another table.
WHERE Clause	Used to filter results using an operator like **IN**, **=**, **or** **$>**.	Find all tracks whose price is greater than the overall average price.

IN Clause Subquery Example:

SQL

```
SELECT InvoiceId, CustomerId
FROM Invoices
WHERE CustomerId IN (
  SELECT CustomerId
  FROM Customers
  WHERE Country = 'Brazil'
);
```

Goal: Find all invoices **for** the customers who live in Brazil. The inner query first gets the list of Brazilian CustomerIds. The outer query then uses that list to find matching invoices.

---

8. 💾 Stored Queries (Views)

A **View** is a saved SQL query. It acts like a **virtual table**—it doesn't store data itself, but every time you query the view, it runs the saved query and shows the most up-to-date result.

Creating a View:

SQL

```
CREATE VIEW BrazilCustomers AS
SELECT *
```

```
FROM Customers
WHERE Country = 'Brazil';
```

-

Querying a View:

```
SQL
SELECT CustomerId, FirstName
FROM BrazilCustomers; -- The database runs the saved SELECT query
```

-
- **Why use Views?** They simplify complex queries (like multi-table JOINs) and hide sensitive columns from users.

---

9. ✏️ Adding, Modifying, and Deleting Data

These commands are crucial for **administration** (managing the data).

INSERT (Adding new data)

Adds new rows to a table.

```
SQL
INSERT INTO Artists (ArtistId, Name)
VALUES (300, 'The New Band');
```

> **Goal:** Add a new artist with ArtistId **300** and Name **'The New Band'** into the Artists table.

UPDATE (Modifying existing data)

Changes values in existing rows. **Always** use a WHERE clause!

```
SQL
UPDATE Tracks
SET UnitPrice = 0.50
WHERE Name = 'The Old Song';
```

> **Goal:** Change the UnitPrice to **$0.50** for the track whose Name is **'The Old Song'**.

DELETE (Removing data)

Removes entire rows from a table. **Always** use a WHERE clause!

```
SQL
DELETE FROM Artists
WHERE ArtistId = 300;
```

> **Goal:** Remove the artist whose ArtistId is **300** from the Artists table. **Caution:** If you omit the WHERE clause, you delete **all** rows in the table!