**DBMS  
Here’s a list of all major Normal Forms (NFs) in Relational Database Design, along with a brief explanation for each:**

**🔹 1NF (First Normal Form)**

**Rule:**

* **Each column contains atomic (indivisible) values.**
* **No repeating groups or arrays.**

**Goal:  
Eliminate multi-valued attributes.**

**🔹 2NF (Second Normal Form)**

**Rule:**

* **Must be in 1NF.**
* **No partial dependency of any column on the primary key (i.e., no non-prime attribute is dependent on part of a composite key).**

**Goal:  
Eliminate partial dependencies.**

**🔹 3NF (Third Normal Form)**

**Rule:**

* **Must be in 2NF.**
* **No transitive dependency (i.e., non-prime attribute should not depend on another non-prime attribute).**

**Goal:  
Eliminate transitive dependencies.**

**🔹 BCNF (Boyce-Codd Normal Form)**

**Rule:**

* **A stricter version of 3NF.**
* **For every functional dependency X → Y, X must be a super key.**

**Goal:  
Resolve anomalies not handled by 3NF when candidate keys are involved.**

**🔹 4NF (Fourth Normal Form)**

**Rule:**

* **Must be in BCNF.**
* **No multi-valued dependencies (MVDs).**

**Goal:  
Handle cases where multiple independent multivalued facts about an entity exist.**

**🔹 5NF (Fifth Normal Form) / PJ/NF (Projection-Join Normal Form)**

**Rule:**

* **Must be in 4NF.**
* **Deals with join dependencies and ensures that lossless joins can be done without redundancy.**

**Goal:  
Eliminate join dependencies.**

**🔹 6NF (Sixth Normal Form) *(Rare, used in temporal databases)***

**Rule:**

* **Decomposed to irreducible components.**
* **Used in temporal and distributed systems.**

**Goal:  
Support non-trivial join dependencies and time-varying data.**

**Summary Table**

| **Normal Form** | **Key Point Eliminated** |
| --- | --- |
| **1NF** | **Repeating groups / Non-atomic values** |
| **2NF** | **Partial dependency** |
| **3NF** | **Transitive dependency** |
| **BCNF** | **Functional dependencies violating superkey** |
| **4NF** | **Multivalued dependency** |
| **5NF** | **Join dependency** |
| **6NF** | **Temporal anomalies (advanced use case)** |

**Sure! Here's a clear explanation and comparison of the SQL commands used to delete data or tables: DELETE, TRUNCATE, and DROP.**

**Great! Let's walk through normalization step-by-step — from unnormalized form (UNF) all the way up to 5NF (also known as Project-Join Normal Form). We'll start with an example table that is not normalized, then normalize it step-by-step:**

**🔴 Unnormalized Table (UNF)**

**Let's say we have this table:**

| **StudentID** | **StudentName** | **Courses** | **Instructor** |
| --- | --- | --- | --- |
| **1** | **Alice** | **Math, Physics** | **John, Sarah** |
| **2** | **Bob** | **Chemistry** | **Emma** |
| **3** | **Charlie** | **Math, Chemistry, Biology** | **John, Emma, Paul** |

**Here:**

* **Courses and Instructor contain multiple values — violating 1NF.**

**✅ Step 1: Convert to 1NF (Atomic Values)**

**First Normal Form requires:**

* **Atomic (indivisible) values**
* **No repeating groups or multivalued fields**

**1NF Table:**

| **StudentID** | **StudentName** | **Course** | **Instructor** |
| --- | --- | --- | --- |
| **1** | **Alice** | **Math** | **John** |
| **1** | **Alice** | **Physics** | **Sarah** |
| **2** | **Bob** | **Chemistry** | **Emma** |
| **3** | **Charlie** | **Math** | **John** |
| **3** | **Charlie** | **Chemistry** | **Emma** |
| **3** | **Charlie** | **Biology** | **Paul** |

**Now each column holds atomic values.**

**✅ Step 2: Convert to 2NF (Remove Partial Dependencies)**

**Second Normal Form requires:**

* **It must be in 1NF**
* **No partial dependency (non-prime attributes must depend on the whole primary key)**

**Here, the composite key is (StudentID, Course). But StudentName depends only on StudentID, not on Course.**

**So, we split the table:**

**Student Table:**

| **StudentID** | **StudentName** |
| --- | --- |
| **1** | **Alice** |
| **2** | **Bob** |
| **3** | **Charlie** |

**Enrollment Table:**

| **StudentID** | **Course** | **Instructor** |
| --- | --- | --- |
| **1** | **Math** | **John** |
| **1** | **Physics** | **Sarah** |
| **2** | **Chemistry** | **Emma** |
| **3** | **Math** | **John** |
| **3** | **Chemistry** | **Emma** |
| **3** | **Biology** | **Paul** |

**Now there are no partial dependencies.**

**✅ Step 3: Convert to 3NF (Remove Transitive Dependencies)**

**Third Normal Form requires:**

* **It must be in 2NF**
* **No transitive dependencies (non-prime attribute depends on another non-prime attribute)**

**Suppose Instructor depends only on Course, not on StudentID. That’s a transitive dependency.**

**Solution: Split again.**

**Course Table:**

| **Course** | **Instructor** |
| --- | --- |
| **Math** | **John** |
| **Physics** | **Sarah** |
| **Chemistry** | **Emma** |
| **Biology** | **Paul** |

**Updated Enrollment Table:**

| **StudentID** | **Course** |
| --- | --- |
| **1** | **Math** |
| **1** | **Physics** |
| **2** | **Chemistry** |
| **3** | **Math** |
| **3** | **Chemistry** |
| **3** | **Biology** |

**Now all dependencies are direct and based on candidate keys.**

**✅ Step 4: Convert to BCNF (Boyce-Codd Normal Form)**

**BCNF requires:**

* **Every determinant must be a candidate key**
* **Slightly stricter than 3NF**

**In this case, everything is fine — Course → Instructor is OK if Course is a candidate key in the Course table.**

**No further changes needed.**

**✅ Step 5: Convert to 4NF (Remove Multivalued Dependencies)**

**4NF requires:**

* **It must be in BCNF**
* **No multivalued dependencies (where two independent multi-valued facts exist)**

**Let’s assume:**

| **StudentID** | **PhoneNumbers** | **Hobbies** |
| --- | --- | --- |
| **1** | **9999, 8888** | **Reading, Painting** |

**This violates 4NF — two independent multi-valued attributes.**

**Fix: Break into two tables**

**StudentPhone Table:**

| **StudentID** | **PhoneNumber** |
| --- | --- |
| **1** | **9999** |
| **1** | **8888** |

**StudentHobby Table:**

| **StudentID** | **Hobby** |
| --- | --- |
| **1** | **Reading** |
| **1** | **Painting** |

**Now multivalued dependencies are eliminated.**

**✅ Step 6: Convert to 5NF (Project-Join Normal Form)**

**5NF removes:**

* **Join dependencies — when a table can be reconstructed from multiple joins, but the table itself is not necessary.**

**It’s rare in practice.**

**Suppose we have this table:**

| **Student** | **Course** | **Project** |
| --- | --- | --- |
| **Alice** | **Math** | **AI Project** |
| **Alice** | **Physics** | **AI Project** |
| **Alice** | **Math** | **ML Project** |

**Here, student-course-project has all combinations. This could be broken into:**

* **Student–Course**
* **Student–Project**
* **Course–Project**

**Then join them when needed — that’s 5NF.**

**✅ Summary Table:**

| **Normal Form** | **Goal** |
| --- | --- |
| **1NF** | **Atomic values, no repeating groups** |
| **2NF** | **Remove partial dependencies** |
| **3NF** | **Remove transitive dependencies** |
| **BCNF** | **Determinant must be a candidate key** |
| **4NF** | **Remove multivalued dependencies** |
| **5NF** | **Eliminate join dependencies** |

**🔄 Concurrency Problem in DBMS (Database Management System)**

**Concurrency in DBMS refers to multiple users accessing or modifying the database simultaneously. While concurrency is desirable for performance, it can lead to conflicts or inconsistencies known as concurrency problems.**

**⚠️ Common Concurrency Problems**

**Here are the main types:**

| **#** | **Problem** | **Description** | **Example** |
| --- | --- | --- | --- |
| **1** | **Lost Update** | **When two transactions read the same data and then update it, the second update overwrites the first one.** | **- T1 and T2 both read balance = 1000. - T1 adds 200 → writes 1200. - T2 subtracts 100 → writes 900. ➜ Final balance is 900, T1's update is lost.** |
| **2** | **Temporary Update (Dirty Read)** | **A transaction reads data written by another uncommitted transaction.** | **- T1 updates salary to 5000. - T2 reads salary = 5000. - T1 rolls back. ➜ T2 used data that never actually existed.** |
| **3** | **Incorrect Summary** | **A transaction reads a summary while other transactions are updating the data.** | **- T1 is summing all account balances. - T2 is transferring money between accounts during T1’s read. ➜ Total may be inaccurate.** |
| **4** | **Unrepeatable Read** | **A transaction reads the same row twice and gets different data because another transaction modified it in between.** | **- T1 reads record X. - T2 updates X. - T1 reads X again. ➜ Value has changed.** |
| **5** | **Phantom Read** | **A transaction re-executes a query and sees new rows inserted by another transaction.** | **- T1 runs SELECT \* FROM Orders WHERE amount > 1000; - T2 inserts a matching new row. - T1 re-runs query and sees extra row.** |

**✅ How to Prevent These Issues**

**DBMSs use Concurrency Control Mechanisms, including:**

* **Locking (Shared, Exclusive)**
* **Timestamps**
* **Multiversion Concurrency Control (MVCC)**
* **Isolation Levels:**
  + **READ UNCOMMITTED**
  + **READ COMMITTED**
  + **REPEATABLE READ**
  + **SERIALIZABLE**

**Each level prevents more issues, but may reduce performance due to increased locking.**

**🔹 1. DELETE**

**Purpose:  
Removes specific rows from a table based on a WHERE condition.**

**Syntax:**

**DELETE FROM table\_name WHERE condition;**

**Key Points:**

* **Deletes row-by-row, logs each deletion (slow for large tables).**
* **Can use WHERE clause to delete specific records.**
* **Can be rolled back if inside a transaction.**
* **Triggers are fired if defined.**

**Example:**

**DELETE FROM Employees WHERE Department = 'HR';**

**🔹 2. TRUNCATE**

**Purpose:  
Removes all rows from a table quickly, but keeps the table structure.**

**Syntax:**

**TRUNCATE TABLE table\_name;**

**Key Points:**

* **Deletes all records without logging each row (faster than DELETE).**
* **Cannot use WHERE clause — all rows are removed.**
* **Cannot be rolled back in some DBMS (like MySQL without transactions).**
* **Resets auto-increment counter (in most DBMS).**
* **Triggers are not fired.**

**Example:**

**TRUNCATE TABLE Employees;**

**🔹 3. DROP**

**Purpose:  
Removes the entire table, including all data and its structure.**

**Syntax:**

**DROP TABLE table\_name;**

**Key Points:**

* **Deletes the table permanently.**
* **Cannot be rolled back once committed.**
* **All data, structure, constraints, and relationships are lost.**
* **Triggers are not fired.**

**Example:**

**DROP TABLE Employees;**

**✅ Summary Table:**

| **Feature** | **DELETE** | **TRUNCATE** | **DROP** |
| --- | --- | --- | --- |
| **Removes Data Only** | **✅** | **✅** | **❌ (removes structure too)** |
| **Removes Structure** | **❌** | **❌** | **✅** |
| **Can Use WHERE Clause** | **✅** | **❌** | **❌** |
| **Rollback Possible** | **✅ (in transaction)** | **❌ / Limited** | **❌** |
| **Triggers Fired** | **✅** | **❌** | **❌** |
| **Resets Auto-Increment** | **❌** | **✅** | **✅ (if recreated)** |
| **Speed** | **Slower** | **Faster** | **Fastest** |

**The ACID properties in databases ensure that transactions are processed reliably and maintain the integrity of the database. ACID stands for:**

**🔹 A – Atomicity**

**Definition:  
A transaction is all or nothing — either all operations succeed or none are applied.**

**Example:  
Transferring ₹1000 from Account A to Account B:**

**BEGIN;**

**UPDATE Accounts SET balance = balance - 1000 WHERE account\_id = 'A';**

**UPDATE Accounts SET balance = balance + 1000 WHERE account\_id = 'B';**

**COMMIT;**

**If the second UPDATE fails, the first one is rolled back — ensuring ₹1000 isn't lost.**

**🔹 C – Consistency**

**Definition:  
A transaction brings the database from one valid state to another valid state — it maintains data integrity rules (constraints, foreign keys, etc.).**

**Example:  
A constraint requires that the balance >= 0.**

**If a transaction tries to withdraw ₹2000 from an account with ₹1000, it must fail to maintain consistency.**

**🔹 I – Isolation**

**Definition:  
Concurrent transactions do not interfere with each other — intermediate results of a transaction are invisible to others.**

**Example:  
Two users try to book the last movie ticket at the same time:**

* **Isolation ensures that only one transaction commits, and the other sees the result only after completion.**

**Isolation Levels (in SQL):**

* **Read Uncommitted**
* **Read Committed**
* **Repeatable Read**
* **Serializable (highest)**

**🔹 D – Durability**

**Definition:  
Once a transaction is committed, the changes are permanently saved, even in case of system failure.**

**Example:  
If a transaction completes and the system crashes immediately after:**

* **The committed changes (e.g., money transferred) remain intact when the system restarts.**

**✅ Summary Table**

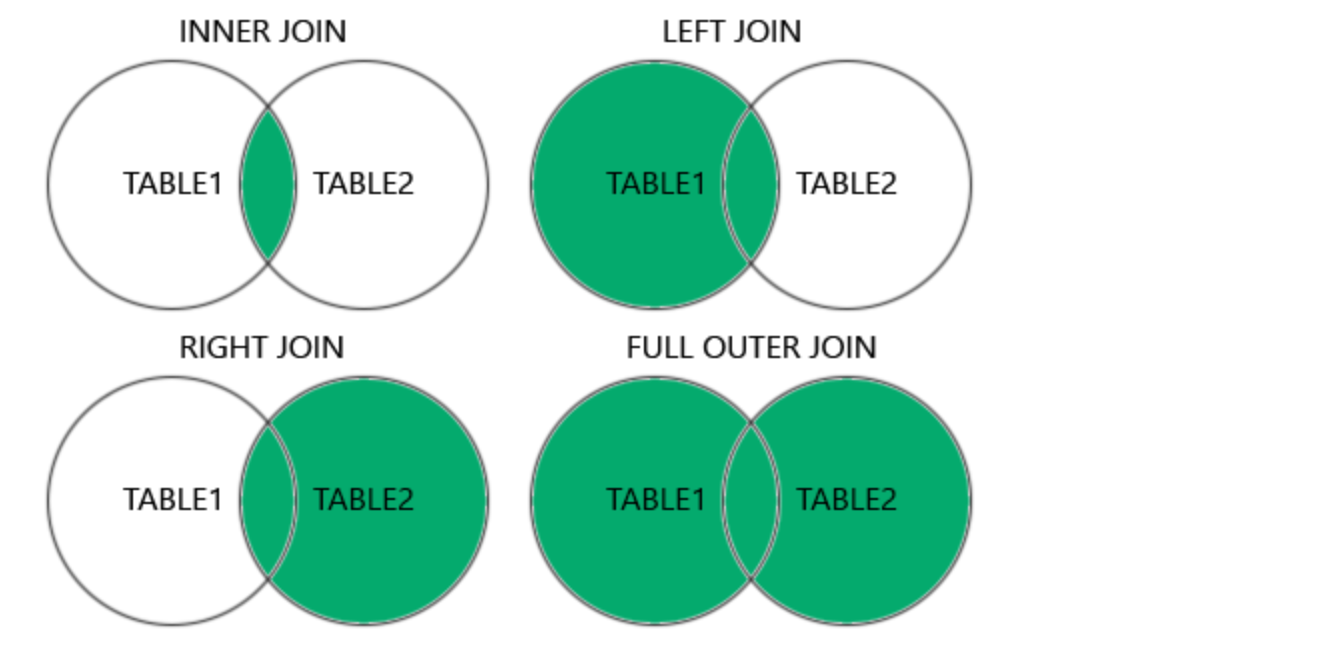
| **Property** | **Description** | **Example Scenario** |
| --- | --- | --- |
| **Atomicity** | **All steps in a transaction succeed or none do** | **Money deducted but not added = rollback** |
| **Consistency** | **Database remains valid before and after transaction** | **Balance should never go below zero** |
| **Isolation** | **Transactions are executed as if serially** | **Two users booking last ticket — one succeeds, one fails** |
| **Durability** | **Committed data persists even after crash** | **Money transfer saved even if power fails immediately after** |

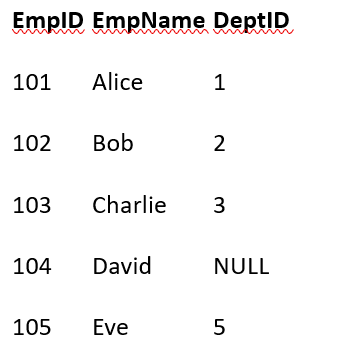
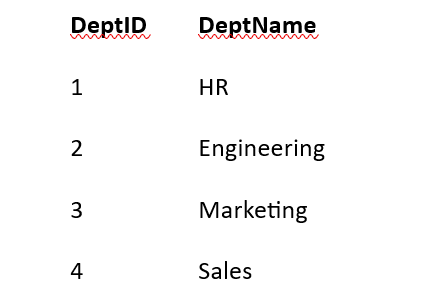
Join:

Different Types of SQL JOINs

Here are the different types of the JOINs in SQL:

* (INNER) JOIN: Returns records that have matching values in both tables
* LEFT (OUTER) JOIN: Returns all records from the left table, and the matched records from the right table
* RIGHT (OUTER) JOIN: Returns all records from the right table, and the matched records from the left table
* FULL (OUTER) JOIN: Returns all records when there is a match in either left or right table





**1. INNER JOIN**

SELECT e.EmpName, d.DeptName

FROM Employee e

INNER JOIN Department d ON e.DeptID = d.DeptID;

| **EmpName** | **DeptName** |
| --- | --- |
| Alice | HR |
| Bob | Engineering |
| Charlie | Marketing |
| **2. LEFT JOIN**  SELECT e.EmpName, d.DeptName  FROM Employee e  LEFT JOIN Department d ON e.DeptID = d.DeptID;   | **EmpName** | **DeptName** | | --- | --- | | Alice | HR | | Bob | Engineering | | Charlie | Marketing | | David | NULL | | Eve | NULL | | **3. RIGHT JOIN**  SELECT e.EmpName, d.DeptName  FROM Employee e  RIGHT JOIN Department d ON e.DeptID = d.DeptID;   | **EmpName** | **DeptName** | | --- | --- | | Alice | HR | | Bob | Engineering | | Charlie | Marketing | | NULL | Sales | |  |  | |  | |  |

**4.FULL OUTER JOIN**

SELECT e.EmpName, d.DeptName

FROM Employee e

FULL OUTER JOIN Department d ON e.DeptID = d.DeptID;

| **EmpName** | **DeptName** | |
| --- | --- | --- |
| Alice | HR | |
| Bob | Engineering | |
| Charlie | Marketing | |
| David | NULL | |
| Eve | NULL | |
| NULL | Sales | |
| **A FULL OUTER JOIN returns:**   * **All matched rows from both tables** * **All unmatched rows from both tables as well, filling in NULLs where there's no match.**   **5.CROSS JOIN**  **SELECT e.EmpName, d.DeptName**  **FROM Employee e**  **CROSS JOIN Department d; --without any condition**   | **EmpName** | **DeptName** | | --- | --- | | Alice | HR | | Alice | Engineering | | Alice | Marketing | | Alice | Sales | | Bob | HR | | ... | ... | | |  |

* A **CROSS JOIN** returns the **Cartesian product** of two tables.
* **That means: every row in the first table is joined with every row in the second table — without any condition.**

**🔹 Rows Returned:**

* If Table A has **m** rows and Table B has **n** rows → result has **m × n** rows.
* In this case 4 X 5=20

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Union vs union all**

**UNION** and **UNION ALL,**both commands are used to combine the result of two or more [**SELECT Statements**](https://www.geeksforgeeks.org/sql/sql-select-query/),

| **Feature** | **UNION** | **UNION ALL** |
| --- | --- | --- |
| Duplicates | Removes duplicates (DISTINCT) | Keeps all duplicates |
| Performance | Slower (needs to sort & remove) | Faster (no extra work) |
| Use Case | When you need **unique** values | When you need **complete** data set |
| NULL Handling | Keeps NULL (but unique only once) | Keeps all NULL values |

**Query:**

SELECT DeptID FROM Employee

**UNION**

SELECT DeptID FROM Department;

| **DeptID** |
| --- |
| 1 |
| 2 |
| 3 |
| NULL |
| 5 |
| 4 |

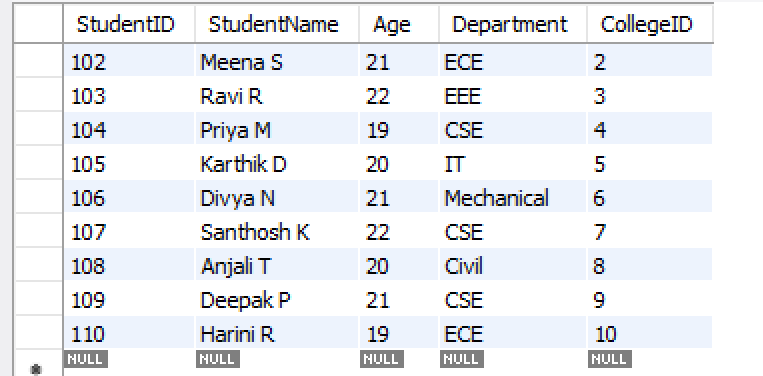
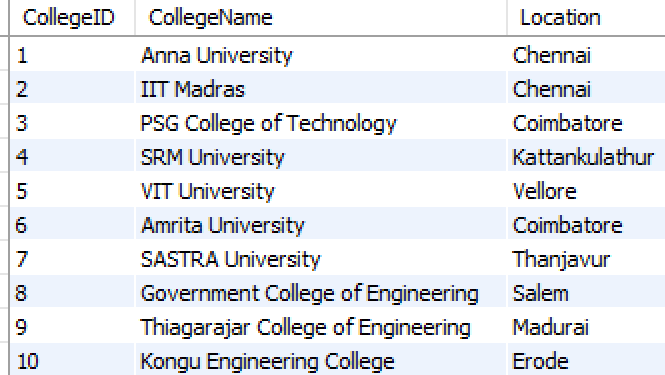
**Query:**

SELECT DeptID FROM Employee

**UNION ALL**

SELECT DeptID FROM Department;

| **DeptID** |
| --- |
| 1 |
| 2 |
| 3 |
| NULL |
| 5 |
| 1 |
| 2 |
| 3 |
| 4 |
|  |
| **Difference between UNION and UNION ALL in SQL:** | |  |  |
| **Feature** | | **UNION** | **UNION ALL** |
| Duplicates | | Removes duplicates | Keeps all duplicates |
| Performance | | Slower (due to duplicate elimination) | Faster (no extra step to remove duplicates) |
| Use Case | | When you want distinct results | When you want all records, including duplicates |

✅ UNION Example: removes duplicates

SELECT Department

FROM Student

WHERE CollegeID IN (

SELECT CollegeID FROM College WHERE Location = 'Chennai'

)

UNION

SELECT Department

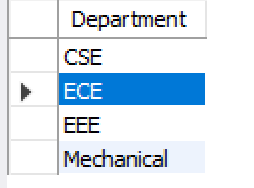
FROM Student

WHERE CollegeID IN (

SELECT CollegeID FROM College WHERE Location = 'Coimbatore'

);

Ouptut: without duplicates



**✅ UNION ALL Example: keeps duplicates**

SELECT Department

FROM Student

WHERE CollegeID IN (

SELECT CollegeID FROM College WHERE Location = 'Chennai'

)

UNION ALL

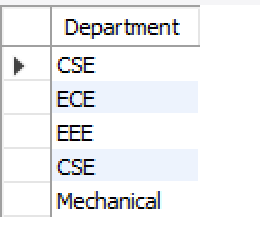
SELECT Department

FROM Student

WHERE CollegeID IN (

SELECT CollegeID FROM College WHERE Location = 'Coimbatore'

);

****

**Window Function in Simple Words:**

A window function in SQL is like saying:

"For each row, calculate something (like rank, total, average) based on a group of rows around it — but don’t remove any rows from the result."

**Window functions let you add extra calculations to each row without removing any rows from the result.**

| **StudentName** | **Department** | **Age** |
| --- | --- | --- |
| **Arun** | **CSE** | **20** |
| **Priya** | **CSE** | **21** |
| **Meena** | **ECE** | **19** |

**🔹 Using GROUP BY:**

**SELECT Department, AVG(Age) AS AvgAge**

**FROM Student**

**GROUP BY Department;**

**This gives:**

| **Department** | **AvgAge** |
| --- | --- |
| **CSE** | **20.5** |
| **ECE** | **19.0** |

**❗ But where are the students?**

**You lose individual student details (like names, ages).  
It only shows one row per department.**

**"What if I add StudentName and Age in a query with GROUP BY Department?"**

**Let me break it down very clearly — with real examples and what will happen in different databases.**

**🔸 1. What You Might Try:**

**SELECT StudentName, Age, Department, AVG(Age) AS AvgAge**

**FROM Student**

**GROUP BY Department;**

**🔥 What Will Happen?**

**✅ In MySQL (with ONLY\_FULL\_GROUP\_BY disabled)**

* **This query might work but give unexpected or incorrect results.**
* **It shows random StudentName and Age from the group, not all students.**

| **StudentName** | **Age** | **Department** | **AvgAge** |
| --- | --- | --- | --- |
| **Arun** | **20** | **CSE** | **20.5** |
| **Meena** | **19** | **ECE** | **19.0** |

**❗ But there are 2 students in CSE, you only get 1 name — this is wrong!**

**🔹 What if you want this?**

| **StudentName** | **Department** | **Age** | **AvgAge** |
| --- | --- | --- | --- |
| **Arun** | **CSE** | **20** | **20.5** |
| **Priya** | **CSE** | **21** | **20.5** |
| **Meena** | **ECE** | **19** | **19.0** |

**You want:**

* **Each student**
* **Their own age**
* **And the average age of their department (repeated in every row)**

**✅ For this, use a Window Function:**

**SELECT**

**StudentName,**

**Department,**

**Age,**

**AVG(Age) OVER (PARTITION BY Department) AS AvgAge**

**FROM Student;**

**It means:**

**"For each student, calculate the average age of their department, and show it alongside."**

**🔁 What’s the difference?**

| **Feature** | **GROUP BY** | **Window Function** |
| --- | --- | --- |
| **Shows all rows?** | **❌ No (only one row per group)** | **✅ Yes (all student rows kept)** |
| **Keeps names?** | **❌ No** | **✅ Yes** |
| **Adds info beside each row?** | **❌ No** | **✅ Yes (adds extra column per row)** |

**📌 Simple Example:**

* **You use GROUP BY when you want:**
  + **One row per department, with summary info.**
* **You use Window Function when you want:**
  + **Every student row,**
  + **Plus some info about the group (like department average).**

**🔹 What is a Window Function in SQL?**

**A Window Function performs a calculation across a set of table rows that are somehow related to the current row. Unlike aggregate functions, it does not collapse rows into a single result — you still get one row per input row, with an extra computed value.**

**🔸 Syntax:**

**FUNCTION\_NAME(...) OVER (**

**PARTITION BY column1**

**ORDER BY column2**

**)**

* **FUNCTION\_NAME: Example – ROW\_NUMBER(), RANK(), SUM(), AVG()**
* **PARTITION BY: Like GROUP BY, but for the window**
* **ORDER BY: Defines order within each partition**

**📘 Example Based on Your Database**

**Goal: Show each student’s rank within their college based on age (youngest = rank 1)**

**SELECT**

**s.StudentID,**

**s.StudentName,**

**c.CollegeName,**

**s.Age,**

**RANK() OVER (PARTITION BY s.CollegeID ORDER BY s.Age) AS AgeRank**

**FROM Student s**

**JOIN College c ON s.CollegeID = c.CollegeID;**

**📊 Output:**

| **StudentID** | **StudentName** | **CollegeName** | **Age** | **AgeRank** |
| --- | --- | --- | --- | --- |
| **101** | **Arun Kumar** | **Anna University** | **20** | **1** |
| **102** | **Meena S** | **IIT Madras** | **21** | **1** |
| **103** | **Ravi R** | **PSG College of Technology** | **22** | **1** |
| **104** | **Priya M** | **SRM University** | **19** | **1** |
| **...** | **...** | **...** | **...** | **...** |

**👆 The rank resets for each CollegeID (because of PARTITION BY).**

**🔹 Common Window Functions**

| **Function** | **Description** |
| --- | --- |
| **ROW\_NUMBER()** | **Unique number for each row in partition** |
| **RANK()** | **Rank with gaps (e.g., 1, 2, 2, 4)** |
| **DENSE\_RANK()** | **Rank without gaps (e.g., 1, 2, 2, 3)** |
| **SUM()** | **Running total in a window** |
| **AVG()** | **Running average** |
| **LAG()** | **Value from previous row** |
| **LEAD()** | **Value from next row** |

**🔸 Example: Cumulative Age Sum in Each College**

**SELECT**

**s.StudentID,**

**s.StudentName,**

**c.CollegeName,**

**s.Age,**

**SUM(s.Age) OVER (PARTITION BY s.CollegeID ORDER BY s.Age) AS RunningAgeTotal**

**FROM Student s**

**JOIN College c ON s.CollegeID = c.CollegeID;**

**✅ Summary**

| **Feature** | **Aggregate Function** | **Window Function** |
| --- | --- | --- |
| **Returns one row?** | **✅ Yes** | **❌ No, returns all rows** |
| **GROUP BY needed?** | **✅ Yes** | **❌ No (uses OVER clause)** |
| **Example** | **SELECT AVG(Age)** | **AVG(Age) OVER (...)** |

**SQL provides many built-in window functions, and while the exact number depends on the database system (MySQL, PostgreSQL, SQL Server, Oracle, etc.), they can be broadly categorized into these 5 main types:**

**🔹 1. Ranking Functions**

**Used to assign a rank or number to rows within partitions.**

| **Function** | **Description** |
| --- | --- |
| **ROW\_NUMBER()** | **Assigns a unique number to each row** |
| **RANK()** | **Like ROW\_NUMBER, but same rank for ties, with gaps** |
| **DENSE\_RANK()** | **Like RANK, but no gaps in rank** |
| **NTILE(N)** | **Divides rows into N equal groups (tiles)** |

**🔹 2. Aggregate Functions (as Window Functions)**

**These are standard aggregate functions used with OVER() to compute running totals, moving averages, etc.**

| **Function** | **Description** |
| --- | --- |
| **SUM()** | **Running total** |
| **AVG()** | **Running average** |
| **MIN()** | **Minimum value in window** |
| **MAX()** | **Maximum value in window** |
| **COUNT()** | **Count of rows in the window** |
| **VAR\_POP() / VAR\_SAMP()** | **Variance** |
| **STDDEV\_POP() / STDDEV\_SAMP()** | **Standard deviation** |

**🔹 3. Value Functions**

**Retrieve a value from a different row in the window.**

| **Function** | **Description** |
| --- | --- |
| **LAG(expr, offset, default)** | **Value from previous row** |
| **LEAD(expr, offset, default)** | **Value from next row** |
| **FIRST\_VALUE(expr)** | **First value in the partition** |
| **LAST\_VALUE(expr)** | **Last value in the partition** |
| **NTH\_VALUE(expr, N)** | **N-th value in the partition** |

**🔹 4. Analytic/Navigation Functions**

**More advanced functions (supported in some databases).**

| **Function** | **Description** |
| --- | --- |
| **PERCENT\_RANK()** | **Relative rank in the partition (0 to 1)** |
| **CUME\_DIST()** | **Cumulative distribution** |
| **RATIO\_TO\_REPORT()** | **Ratio of a value to the total in the partition (Oracle)** |

**🔹 5. Window Frame Clauses**

**Not functions themselves, but used with OVER() to define range of rows for the function to operate on.**

**OVER (**

**PARTITION BY ...**

**ORDER BY ...**

**ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING**

**)**

**✅ Summary**

| **Category** | **Approximate Count** |
| --- | --- |
| **Ranking Functions** | **4** |
| **Aggregate Functions** | **6+** |
| **Value Functions** | **5+** |
| **Analytic Functions** | **3+** |
| **Total (Commonly used)** | **~20+** |

**⚠️ Note:**

* **Not all databases support every function.**
* **Some (like RATIO\_TO\_REPORT) are specific to Oracle.**
* **MySQL, PostgreSQL, SQL Server, Oracle, and SQLite may have slight syntax differences.**

**🔹 SUBSTRING() in SQL**

**The SUBSTRING() function is used to extract a portion of a string based on a starting position and length.**

**✅ Syntax (Standard SQL):**

**SUBSTRING(string FROM start\_position FOR length)**

**✅ Syntax (MySQL, SQL Server):**

**SUBSTRING(string, start\_position, length)**

**🔸 Example:**

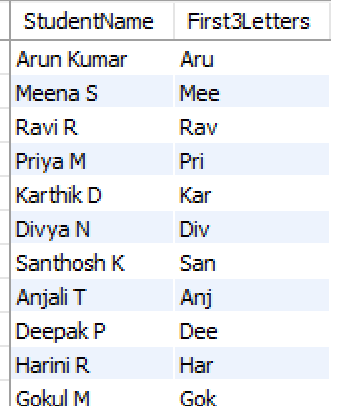
**Let’s use your Student table:**

**SELECT StudentName,**

**SUBSTRING(StudentName, 1, 3) AS FirstThreeLetters**

**FROM Student;**

**Output:**

****

**📌 This will return the first 3 characters of each student's name.**

**🔹 Real-Life Usage Examples:**

**1. Get first name from full name (assuming space between first and last name):**

**SELECT StudentName,**

**SUBSTRING(StudentName, 1, LOCATE(' ', StudentName) - 1) AS FirstName**

**FROM Student**

**WHERE LOCATE(' ', StudentName) > 0;**

**2. Get last 3 characters of the student name:**

**SELECT StudentName,**

**SUBSTRING(StudentName, LENGTH(StudentName) - 2, 3) AS Last3Chars**

**FROM Student;**

**🔹 Other Related String Functions:**

| **Function** | **Description** |
| --- | --- |
| **LEFT(str, n)** | **Gets first n characters** |
| **RIGHT(str, n)** | **Gets last n characters** |
| **LENGTH(str)** | **Returns length of string** |
| **LOCATE(substr, str)** | **Position of substring in string** |
| **CONCAT(str1, str2)** | **Combine strings** |
| **REPLACE(str, from, to)** | **Replace part of string** |

**✅ Example using multiple functions:**

**SELECT**

**StudentName,**

**LEFT(StudentName, 4) AS LeftPart,**

**RIGHT(StudentName, 2) AS RightPart,**

**LENGTH(StudentName) AS NameLength**

**FROM Student;**

**🔹 INTERVAL in SQL (MySQL)**

**Used to add or subtract time from a DATE, DATETIME, or TIMESTAMP.**

**✅ Syntax**

**-- Add**

**SELECT CURDATE() + INTERVAL 7 DAY;**

**-- Subtract**

**SELECT '2025-08-18' - INTERVAL 2 MONTH;**

| **Unit** | **Example** |
| --- | --- |
| **DAY** | **INTERVAL 7 DAY** |
| **MONTH** | **INTERVAL 1 MONTH** |
| **YEAR** | **INTERVAL 2 YEAR** |
| **HOUR** | **INTERVAL 3 HOUR** |
| **MINUTE** | **INTERVAL 10 MINUTE** |

**🔸 Examples**

1. **Add 7 days to today:**

**SELECT CURDATE() + INTERVAL 7 DAY;**

1. **Subtract 2 months from a date:**

**SELECT '2025-08-18' - INTERVAL 2 MONTH;**

1. **Last 30 days filter:**

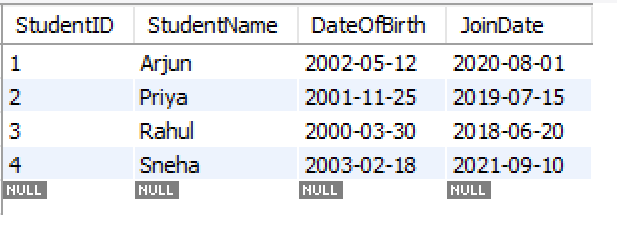
**SELECT \* FROM Student**

**WHERE RegisteredDate >= CURDATE() - INTERVAL 30 DAY;**

1. **Age from DOB (if column exists):**

**SELECT TIMESTAMPDIFF(YEAR, DOB, CURDATE()) AS Age;**

**Database!**

****

**Example A: Add 4 Years to Join Date (Expected Graduation Date)**

**SELECT**

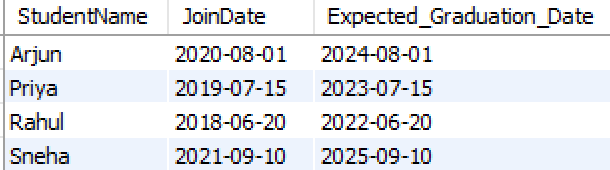
**StudentName,**

**JoinDate,**

**DATE\_ADD(JoinDate, INTERVAL 4 YEAR) AS Expected\_Graduation\_Date**

**FROM Student;**

**o/p:**

****

**✅ Example B: Students who Joined 3 Years Ago**

**SELECT**

**StudentName,**

**JoinDate**

**FROM Student**

**WHERE JoinDate = DATE\_SUB(CURDATE(), INTERVAL 3 YEAR);**

**✅ Example C: Update Join Date to Add 1 Month (For Delay Case)**

**UPDATE Student**

**SET JoinDate = DATE\_ADD(JoinDate, INTERVAL 1 MONTH)**

**WHERE StudentName = 'Sneha';**

**✅ Example D: Calculate Age from DOB (Approximate)**

**SELECT**

**StudentName,**

**DateOfBirth,**

**TIMESTAMPDIFF(YEAR, DateOfBirth, CURDATE()) AS Age**

**FROM Student;**

**🔹 NVL() in SQL**

**NVL() is a function used to replace NULL values with a specified default value.**

**✅ Syntax:**

**NVL(expression, replacement\_value)**

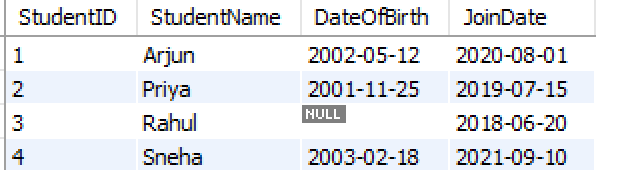
* **If expression is NULL, it returns replacement\_value.**
* **Otherwise, it returns expression.**

**🔸 Example:**

**SELECT StudentName, NVL(Department, 'Not Assigned') AS Dept**

**FROM Student;**

**📌 If Department is NULL, it will display 'Not Assigned'.**

****

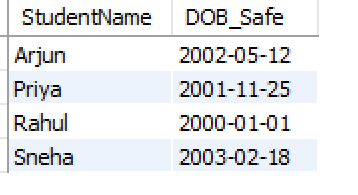
**SELECT**

**StudentName,**

**IFNULL(DateOfBirth, '2000-01-01') AS DOB\_Safe**

**FROM Student1;**

**o/p:**

****

**✅ Key Points:**

* **Available in Oracle.**
* **In MySQL, use IFNULL(expr, value).**
* **In SQL Server, use ISNULL(expr, value).**

**🔹 IFNULL() – MySQL**

**Replaces NULL with a given value.**

**✅ Syntax:**

**IFNULL(expression, replacement)**

**🔸 Example:**

**SELECT IFNULL(Department, 'Not Assigned') FROM Student;**

**✅ If Department is NULL, returns 'Not Assigned'.**

**🔹 ISNULL() – SQL Server**

**Same purpose as NVL() and IFNULL() but used in SQL Server.**

**✅ Syntax:**

**ISNULL(expression, replacement)**

**🔸 Example:**

**SELECT ISNULL(Department, 'Not Assigned') FROM Student;**

**🔹 COALESCE() – Standard SQL (Works in All Databases)**

**Returns the first non-NULL value in the list.**

**✅ Syntax:**

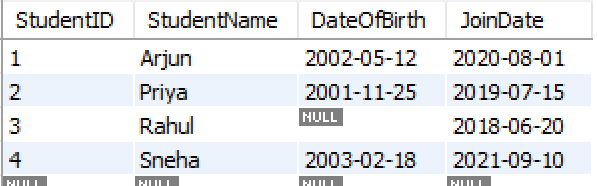
**COALESCE(expr1, expr2, ..., exprN)**

**🔸 Example:**

**SELECT COALESCE(Department, 'Not Assigned') FROM Student;**

**🔹 Works like NVL() but supports multiple fallback values:**

**COALESCE(NULL, NULL, 'CSE') → returns 'CSE'**

****

**SELECT**

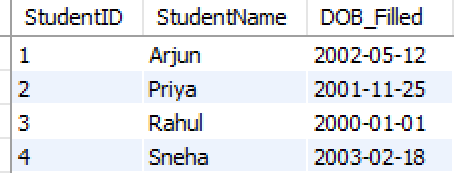
**StudentID,**

**StudentName,**

**COALESCE(DateOfBirth, '2000-01-01') AS DOB\_Filled**

**FROM Student1;**

**o/P:**

****

**✅ Summary Table:**

| **Function** | **Supported In** | **Description** |
| --- | --- | --- |
| **NVL()** | **Oracle** | **Replace NULL with given value** |
| **IFNULL()** | **MySQL** | **Replace NULL with given value** |
| **ISNULL()** | **SQL Server** | **Replace NULL with given value** |
| **COALESCE()** | **All (Standard SQL)** | **First non-NULL from multiple inputs** |

**MySQL InnoDB Limits:**

* **Max Rows: Up to 4 billion**
* **Max Columns: 1017 (limited by row size = 65,535 bytes)**

**✅ Wildcards in SQL (Used with LIKE operator)**

| **Wildcard** | **Description** | **Example** |
| --- | --- | --- |
| **%** | **Matches zero or more characters** | **WHERE name LIKE 'A%' → names starting with A** |
| **\_** | **Matches exactly one character** | **WHERE name LIKE '\_a%' → second letter is 'a'** |
| **[ ]** | **Matches any one character in set (used in some DBs like SQL Server)** | **WHERE name LIKE '[aeiou]%'** |
| **[^ ] or !** | **Matches any one character NOT in set (SQL Server)** | **WHERE name LIKE '[^aeiou]%'** |

**🔹 Common Examples**

* **LIKE 'J%n' → Matches "John", "Jin", "Jordan"**
* **LIKE '\_a%' → Matches "Mary", "Jack", "David"**

**✅ REGEX in SQL (Regular Expressions)**

**REGEX (Regular Expression) is used to match patterns in strings. Supported in MySQL, PostgreSQL, Oracle, etc., using different functions.**

**🔹 MySQL Syntax**

**SELECT \* FROM table\_name**

**WHERE column\_name REGEXP 'pattern';**

**✅ Common REGEXP Patterns in SQL**

| **Pattern** | **Meaning** | **Example** |
| --- | --- | --- |
| **^** | **Start of string** | **'^A' → Starts with A** |
| **$** | **End of string** | **'z$' → Ends with z** |
| **.** | **Any single character** | **'b.t' → bat, bit, but** |
| **[abc]** | **Match any character in the set** | **'[aeiou]' → any vowel** |
| **[^abc]** | **Not any of a, b, or c** | **'[^xyz]'** |
| **[a-z]** | **Range – any lowercase letter** | **'[a-z]'** |
| **\*** | **0 or more occurrences** | **'bo\*t' → bt, bot, boot** |
| **+** | **1 or more occurrences** | **'bo+t' → bot, boot (not bt)** |
| **{n}** | **Exactly n occurrences** | **'a{3}' → aaa** |
| **{n,}** | **n or more occurrences** | **'a{2,}' → aa, aaa, aaaa...** |
| **{n,m}** | **Between n and m occurrences** | **'a{2,4}' → aa, aaa, aaaa** |

**🔹 Example Queries**

**-- Names starting with 'S'**

**SELECT \* FROM employees WHERE name REGEXP '^S';**

**-- Names ending with 'n'**

**SELECT \* FROM employees WHERE name REGEXP 'n$';**

**-- Names with 3-letter word like 'cat', 'bat', 'rat'**

**SELECT \* FROM animals WHERE name REGEXP '^[cbr]at$';**

**✅ ROLLBACK in SQL**

**ROLLBACK is a transaction control command used to undo changes made by the current transaction before it is committed.**

**🔹 When to Use ROLLBACK**

* **When an error occurs during a transaction.**
* **To cancel a set of changes if a condition fails.**
* **Used with SAVEPOINT to roll back to a specific point.**

**🔹 Basic Syntax**

**ROLLBACK;**

**🔹 With Transactions:**

**BEGIN;**

**UPDATE accounts SET balance = balance - 100 WHERE id = 1;**

**UPDATE accounts SET balance = balance + 100 WHERE id = 2;**

**-- Suppose something fails here**

**ROLLBACK; -- Undoes both updates**

**🔹 Using SAVEPOINT and ROLLBACK TO**

**BEGIN;**

**SAVEPOINT step1;**

**UPDATE users SET age = age + 1;**

**SAVEPOINT step2;**

**DELETE FROM users WHERE age > 100;**

**-- Oops, too many rows deleted!**

**ROLLBACK TO step2; -- Undoes the DELETE only**

**COMMIT; -- Saves the rest**

**🔹 Key Points**

* **ROLLBACK only affects uncommitted changes.**
* **Cannot rollback after a COMMIT.**
* **Often used in error-handling logic in applications or stored procedures.**

**✅ Index in SQL**

**An index is a database object that improves the speed of data retrieval operations on a table at the cost of additional storage and slower write (INSERT/UPDATE/DELETE) operations.**

**🔹 Purpose of Index**

* **Speeds up SELECT queries.**
* **Works like the index of a book: quick access to data without scanning the whole table.**

**🔹 Basic Syntax**

**CREATE INDEX index\_name ON table\_name(column\_name);**

**Eg:**

**CREATE INDEX idx\_department ON Student(Department);**

**SELECT \* FROM Student WHERE Department = 'CSE';**

**DROP INDEX idx\_department ON Student;**

**observe that you might not see any immediate visible difference after creating an index like:**

**🔹 Types of Indexes**

| **Type** | **Description** |
| --- | --- |
| **1. Single-Column Index** | **Index on one column. Example: CREATE INDEX idx\_name ON users(name);** |
| **2. Composite (Multi-column) Index** | **Index on two or more columns. Example: CREATE INDEX idx\_multi ON orders(customer\_id, order\_date);** |
| **3. Unique Index** | **Ensures all values in the indexed column(s) are unique. Automatically created on PRIMARY KEY and UNIQUE columns.** |
| **4. Full-Text Index** | **Used for full-text searches on large text fields (e.g., blog content). Example: CREATE FULLTEXT INDEX idx\_ft ON articles(content); *(MySQL specific)*** |
| **5. Spatial Index** | **Used for spatial data types like geometry or location. *(MySQL/PostGIS)*** |
| **6. Clustered Index** | **Determines physical order of rows in the table. Only one clustered index per table (e.g., PRIMARY KEY in SQL Server).** |
| **7. Non-Clustered Index** | **Separate structure pointing to actual rows. A table can have multiple non-clustered indexes.** |

**🔹 1. Single-Column Index**

**✅ Use when: You frequently filter/search by one column (e.g., Department).**

**CREATE INDEX idx\_department ON Student(Department);**

* **Speeds up:**
* **SELECT \* FROM Student WHERE Department = 'CSE';**

**🔹 2. Composite Index (Multi-column index)**

**✅ Use when: You frequently filter/search using multiple columns together.**

**CREATE INDEX idx\_name\_dept ON Student(Name, Department);**

* **Speeds up:**
* **SELECT \* FROM Student WHERE Name = 'John' AND Department = 'CSE';**

**⚠️ Order matters! Index works best if you search from leftmost column(s).**

**🔹 3. Unique Index**

**✅ Use when: You want to prevent duplicate values in a column (like RollNo).**

**CREATE UNIQUE INDEX idx\_rollno\_unique ON Student(RollNo);**

* **Same as:**
* **ALTER TABLE Student ADD CONSTRAINT uniq\_rollno UNIQUE (RollNo);**

**🔹 4. Full-Text Index (MySQL)**

**✅ Use for: Searching keywords in text-based columns like Name or Address.**

**CREATE FULLTEXT INDEX idx\_name\_text ON Student(Name);**

* **Enables:**
* **SELECT \* FROM Student WHERE MATCH(Name) AGAINST ('Kumar');**

**🔹 5. Bitmap Index (Oracle-specific, for low-cardinality columns like Gender)**

**✅ Use when: Column has few unique values ('M', 'F')**

**-- Oracle only**

**CREATE BITMAP INDEX idx\_gender\_bitmap ON Student(Gender);**

**🔹 6. Function-Based Index (Oracle/PostgreSQL)**

**✅ Use when: You search using expressions**

**-- Oracle**

**CREATE INDEX idx\_upper\_name ON Student(UPPER(Name));**

* **Helps this query:**
* **SELECT \* FROM Student WHERE UPPER(Name) = 'KUMAR';**

**🔹 How to Delete an Index**

**DROP INDEX idx\_name\_dept;**

**🔹 Dropping an Index**

**DROP INDEX index\_name ON table\_name; -- MySQL**

**DROP INDEX index\_name; -- Oracle/PostgreSQL**

**✅ View in SQL**

**A View is a virtual table based on the result of a SQL query. It contains rows and columns just like a real table but does not store data physically.**

**🔹 Purpose of a View**

* **Simplifies complex queries.**
* **Provides security by restricting access to specific columns/rows.**
* **Helps in reusability of SQL queries.**

**🔹 Basic Syntax**

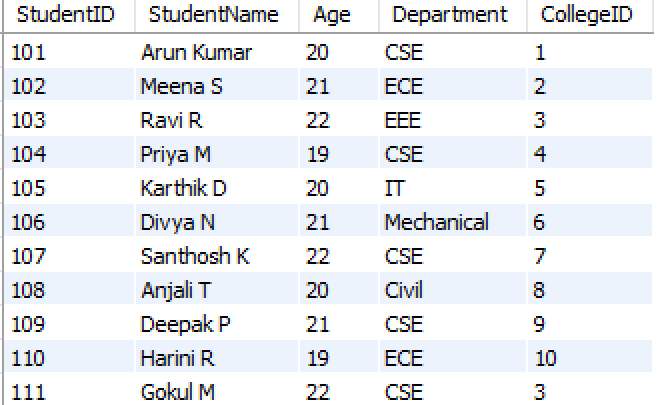
**-- Create a view**

**CREATE VIEW view\_name AS**

**SELECT column1, column2**

**FROM table\_name**

**WHERE condition;**

****

**Eg:**

**CREATE VIEW High\_Scorers AS**

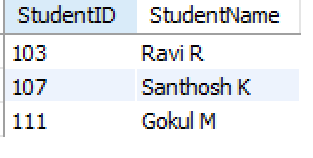
**SELECT StudentID, StudentName**

**FROM Student**

**WHERE Age > 21;**

**SELECT \* FROM High\_Scorers;**

**o/p:**

****

**🔹 Dropping a View**

**DROP VIEW view\_name;**

**🔹 Types of Views**

| **Type** | **Description** |
| --- | --- |
| **1. Simple View** | **Based on a single table, no functions or group by.** |
| **2. Complex View** | **Based on multiple tables, can include joins, aggregates, etc.** |
| **3. Materialized View** | **Stores actual data (used in Oracle/PostgreSQL). Needs manual/automatic refresh. Faster but consumes storage. *(Not in MySQL)*** |

**simple and complex view same but if we use joins,group,agg then its is complex views**

**What is a Materialized View?**

* Unlike a normal view, a materialized view stores the data physically.

**🔹 Modifying a View**

**CREATE OR REPLACE VIEW view\_name AS**

**SELECT ...**

**🔹 Can We Update a View?**

* **Yes, if the view is simple (one table, no aggregation).**
* **No, if the view has:**
  + **Aggregates (SUM, AVG, etc.)**
  + **GROUP BY or DISTINCT**
  + **Joins (in most cases)**

**UNIQUE Key in SQL**

**A UNIQUE key ensures that all values in a column or a set of columns are unique — no duplicate values are allowed.**

**Unlike the Primary Key, a table can have multiple UNIQUE keys, but each UNIQUE key column can accept NULL values (only once per column).**

**eg**

**CREATE TABLE employees (**

**emp\_id INT,**

**email VARCHAR(100) UNIQUE**

**);**

**Here’s a brief explanation of each of these SQL keywords/functions:**

**✅ 1. LIMIT and OFFSET (Used in MySQL and PostgreSQL)**

* **LIMIT: Specifies the maximum number of rows to return.**
* **OFFSET: Specifies how many rows to skip before starting to return rows.**

**🔹 Syntax:**

**SELECT \* FROM employees**

**LIMIT 10 OFFSET 5;**

**This will return 10 rows, skipping the first 5.**

**🔄 Shortcut:**

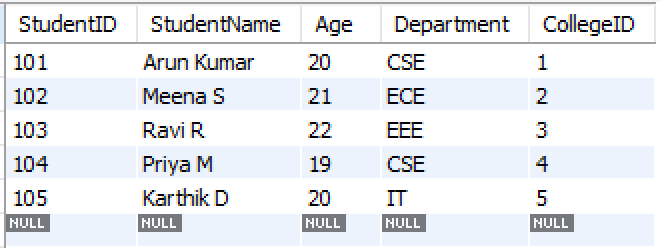
**SELECT \* FROM employees**

**LIMIT 5, 10;**

**Here, 5 is the OFFSET, 10 is the LIMIT.**

**Eg:** **Fetch only first 5 rows from the Student table:**

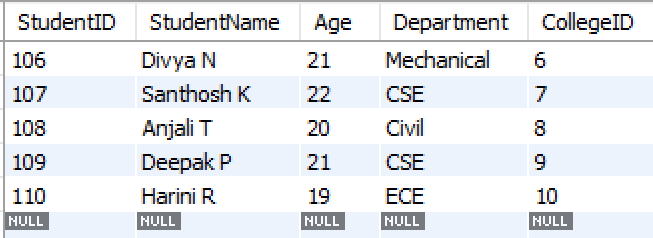
**SELECT \* FROM Student**

**LIMIT 5;  
  
**

**Eg:** **Skip the first 5 rows and show the next 5 rows:**

**SELECT \* FROM Student**

**LIMIT 5 OFFSET 5;**

****

**✅ 2. COMMIT**

* **Used to save all the changes made by the current transaction permanently to the database.**

**🔹 Example:**

**START TRANSACTION;**

**UPDATE employees SET salary = salary + 1000 WHERE emp\_id = 101;**

**COMMIT;**

**If no COMMIT is done and there's an error, you can use ROLLBACK to undo changes.**

**✅ 3. EXISTS**

* **Used to check if a subquery returns any rows. Returns TRUE if the subquery has at least one row.**

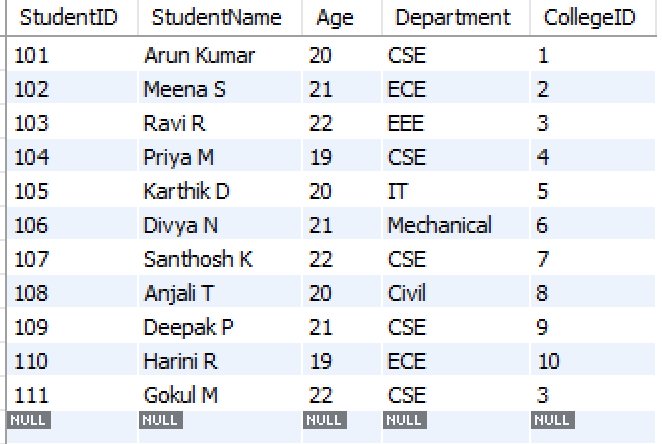
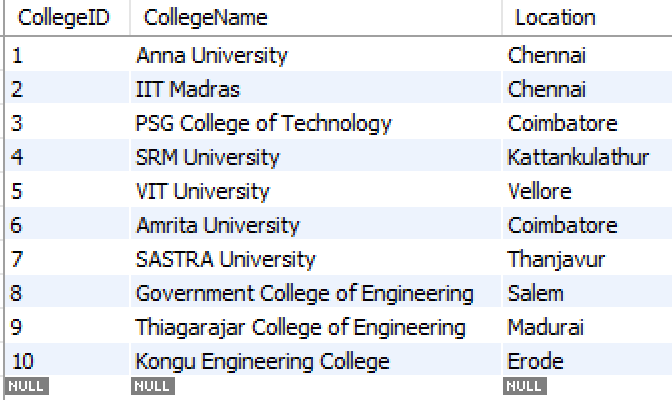
**🔹 Syntax:**

**SELECT \* FROM employees e**

**WHERE EXISTS (**

**SELECT 1 FROM departments d WHERE d.manager\_id = e.emp\_id**

**);**

**Returns employees who are managers of some department.  
  
** ****

Eg:

**-- List all colleges that have at least one student**

**SELECT \***

**FROM College C**

**WHERE EXISTS (**

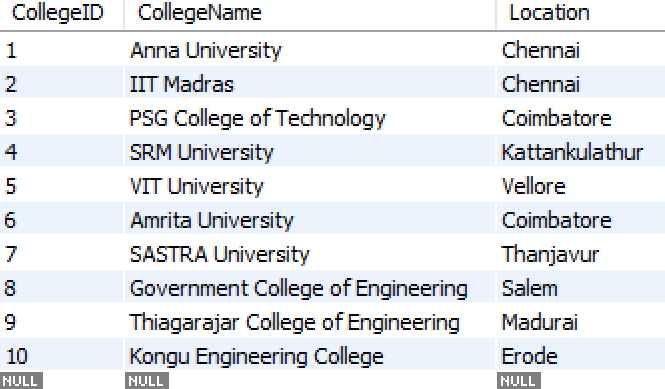
**SELECT 1**

**FROM Student S**

**WHERE S.CollegeID = C.CollegeID**

**);**

**o/p:**

****

**Explanation:**

* **EXISTS checks if the subquery returns any row.**
* **SELECT 1 is valid — it’s just a placeholder, as EXISTS only cares about the existence of rows, not their actual content.**
* **If even one student exists with a matching CollegeID, the College row will be returned.**

**✅ 4. CURDATE()**

* **Returns the current date in YYYY-MM-DD format.**

**🔹 Example:**

**SELECT CURDATE(); -- Output: 2025-08-18**

**You can use it in conditions:**

**SELECT \* FROM orders WHERE order\_date = CURDATE();**

**✅ COUNT(\*) in SQL**

**COUNT(\*) is an aggregate function that returns the total number of rows in a table or result set — including NULLs.**

**🔹 Syntax:**

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

**🔹 Example:**

**SELECT COUNT(\*) FROM employees;**

**This returns the total number of employees in the table.**

**🔸 With WHERE Clause:**

**SELECT COUNT(\*) FROM employees WHERE department = 'HR';**

**Counts only employees in the HR department.**

**🔸 With GROUP BY:**

**SELECT department, COUNT(\*)**

**FROM employees**

**GROUP BY department;**

**Gives the number of employees per department.**

**🔸 COUNT(\*) vs COUNT(column\_name):**

| **Expression** | **What it counts** |
| --- | --- |
| **COUNT(\*)** | **All rows, even if they have NULLs** |
| **COUNT(column)** | **Only non-NULL values in the column** |

**Example:**

**SELECT COUNT(email) FROM users;**

**-- Excludes rows where email IS NULL**

**SELECT COUNT(\*) FROM users;**

**-- Includes all rows**

**Here’s a short and complete explanation of PL/SQL for your document:**

**🔹 PL/SQL – Procedural Language/Structured Query Language**

**PL/SQL is Oracle's procedural extension of SQL that allows you to write programs, control flow, and combine SQL with procedural logic.**

**✅ Features:**

| **Feature** | **Description** |
| --- | --- |
| **Block-structured** | **Code is written in blocks (DECLARE, BEGIN, END)** |
| **Supports variables** | **Can declare and use variables** |
| **Control structures** | **IF, LOOP, WHILE, CASE** |
| **Error handling** | **Uses EXCEPTION block** |
| **Reusable code** | **Supports procedures, functions, triggers** |

**✅ Basic Structure:**

**DECLARE**

**-- Variable declarations**

**v\_name VARCHAR2(50);**

**BEGIN**

**-- Executable statements**

**v\_name := 'Praveen';**

**DBMS\_OUTPUT.PUT\_LINE('Hello, ' || v\_name);**

**EXCEPTION**

**-- Error handling**

**WHEN OTHERS THEN**

**DBMS\_OUTPUT.PUT\_LINE('Error occurred!');**

**END;**

**🔹 Common PL/SQL Constructs:**

| **Construct** | **Example** |
| --- | --- |
| **Variable** | **v\_age NUMBER := 25;** |
| **IF-ELSE** | **IF v\_age > 18 THEN ... END IF;** |
| **Loop** | **FOR i IN 1..10 LOOP ... END LOOP;** |
| **Cursor** | **Used to fetch multiple rows** |
| **Procedure** | **CREATE PROCEDURE proc\_name IS ...** |
| **Function** | **CREATE FUNCTION func\_name RETURN ...** |

**✅ Example: Simple Procedure**

**CREATE OR REPLACE PROCEDURE welcome\_user(p\_name IN VARCHAR2) IS**

**BEGIN**

**DBMS\_OUTPUT.PUT\_LINE('Welcome, ' || p\_name || '!');**

**END;**

**🔹 Advantages of PL/SQL:**

* **Reduces network traffic (logic runs on DB server)**
* **Supports modular programming**
* **Better performance for batch processing**
* **Strong exception handling**

**Let me know if you want short notes on PL/SQL cursors, exceptions, or stored procedures/functions with examples.**

**Here is a short and complete explanation of SQL Triggers — perfect for your document:**

**🔹 What is a Trigger in SQL?**

**A trigger is a stored procedure that automatically executes in response to specific events (like INSERT, UPDATE, or DELETE) on a table or view.**

**✅ Syntax (Generic Format):**

**CREATE TRIGGER trigger\_name**

**AFTER | BEFORE INSERT | UPDATE | DELETE**

**ON table\_name**

**FOR EACH ROW**

**BEGIN**

**-- Trigger logic**

**END;**

**🔸 Types of Triggers:**

| **Type** | **Fires When** |
| --- | --- |
| **BEFORE INSERT** | **Before inserting a row** |
| **AFTER INSERT** | **After inserting a row** |
| **BEFORE UPDATE** | **Before updating a row** |
| **AFTER UPDATE** | **After updating a row** |
| **BEFORE DELETE** | **Before deleting a row** |
| **AFTER DELETE** | **After deleting a row** |

**🔸 Example (MySQL): Log inserts into StudentLog table**

**CREATE TRIGGER log\_insert**

**AFTER INSERT ON Student**

**FOR EACH ROW**

**BEGIN**

**INSERT INTO StudentLog (StudentID, Action, LogTime)**

**VALUES (NEW.StudentID, 'INSERT', NOW());**

**END;**

**🔸 Accessing Old & New Values:**

| **Keyword** | **Description** |
| --- | --- |
| **NEW** | **Refers to new row data (INSERT/UPDATE)** |
| **OLD** | **Refers to old row data (UPDATE/DELETE)** |

**✅ Uses of Triggers:**

* **Logging changes**
* **Enforcing complex rules**
* **Automatically updating audit tables**
* **Preventing invalid operations**

**🔹 Points to Note:**

* **Triggers run automatically — no need to call them.**
* **Avoid complex logic in triggers to prevent performance issues.**
* **Not all databases support BEFORE triggers (e.g., SQL Server supports only AFTER).**

**Let me know if you want examples of triggers for UPDATE, DELETE, or cross-table validations.**

**🔹 Session in SQL (Database Context) – *Short & Complete for Document***

**✅ Definition:**

**A session in SQL refers to the period of interaction between a user and the database, from login to logout (or connection open to close).**

**🔸 Key Characteristics:**

| **Feature** | **Description** |
| --- | --- |
| **Starts when** | **A user/application connects to the database** |
| **Ends when** | **The connection is closed or times out** |
| **Unique per user** | **Each session has a unique session ID** |
| **Tracks** | **Transactions, temporary tables, session variables, settings** |

**🔹 Common Uses:**

1. **Transaction Management:**
2. **BEGIN;**
3. **-- SQL statements**
4. **COMMIT;**
5. **Session Variables (MySQL):**
6. **SET @user\_id = 101;**
7. **SELECT @user\_id;**
8. **Session Settings (Oracle):**
9. **ALTER SESSION SET NLS\_DATE\_FORMAT = 'YYYY-MM-DD';**
10. **Temporary Tables (valid for session only):**
11. **CREATE TEMPORARY TABLE temp\_students (...);**

**🔹 View Active Sessions (Examples):**

**✅ Oracle:**

**SELECT SID, SERIAL#, USERNAME, STATUS FROM V$SESSION;**

**✅ SQL Server:**

**SELECT \* FROM sys.dm\_exec\_sessions;**

**✅ MySQL:**

**SHOW PROCESSLIST;**

**🔹 Summary Table:**

| **Database** | **How to Track Session Info** |
| --- | --- |
| **Oracle** | **V$SESSION, ALTER SESSION** |
| **SQL Server** | **sys.dm\_exec\_sessions, @@SPID** |
| **MySQL** | **SHOW SESSION STATUS, SET @var = value** |

**In SQL, the CASE statement acts like a switch-case structure in programming languages. It allows you to execute conditional logic within SQL queries.**

**✅ Syntax:**

**SELECT**

**column1,**

**CASE**

**WHEN condition1 THEN result1**

**WHEN condition2 THEN result2**

**...**

**ELSE default\_result**

**END AS alias\_name**

**FROM table\_name;**

**🧾 Example using your Student table:**

**-- Add a column to categorize age groups**

**SELECT**

**StudentID,**

**StudentName,**

**Age,**

**CASE**

**WHEN Age < 18 THEN 'Minor'**

**WHEN Age BETWEEN 18 AND 22 THEN 'Young Adult'**

**WHEN Age > 22 THEN 'Adult'**

**ELSE 'Unknown'**

**END AS AgeGroup**

**FROM Student;**

**This acts like a switch-case:**

* **If age < 18 → 'Minor'**
* **If age between 18–22 → 'Young Adult'**
* **If age > 22 → 'Adult'**

**📦 Another Example: Mapping Departments**

**SELECT**

**StudentID,**

**StudentName,**

**Department,**

**CASE Department**

**WHEN 'CSE' THEN 'Computer Science'**

**WHEN 'ECE' THEN 'Electronics'**

**WHEN 'MECH' THEN 'Mechanical'**

**ELSE 'Other'**

**END AS FullDepartment**

**FROM Student;**

**💡 CAP Theorem stands for:**

* **C – Consistency**
* **A – Availability**
* **P – Partition Tolerance**

**✅ What do these mean?**

1. **Consistency  
   → Every user sees the same data at the same time.  
   *Example:* If you transfer money from one account to another, both account balances update instantly for everyone.**
2. **Availability  
   → The system always responds, even if some data is outdated.  
   *Example:* Even during heavy traffic, the website still shows some data (might not be the latest).**
3. **Partition Tolerance  
   → The system keeps working even if there’s a communication break between servers.  
   *Example:* Some servers go offline, but the system still works in parts.**

**⚠️ The Key Point:**

**You can't achieve all 3 at the same time in a distributed system.**

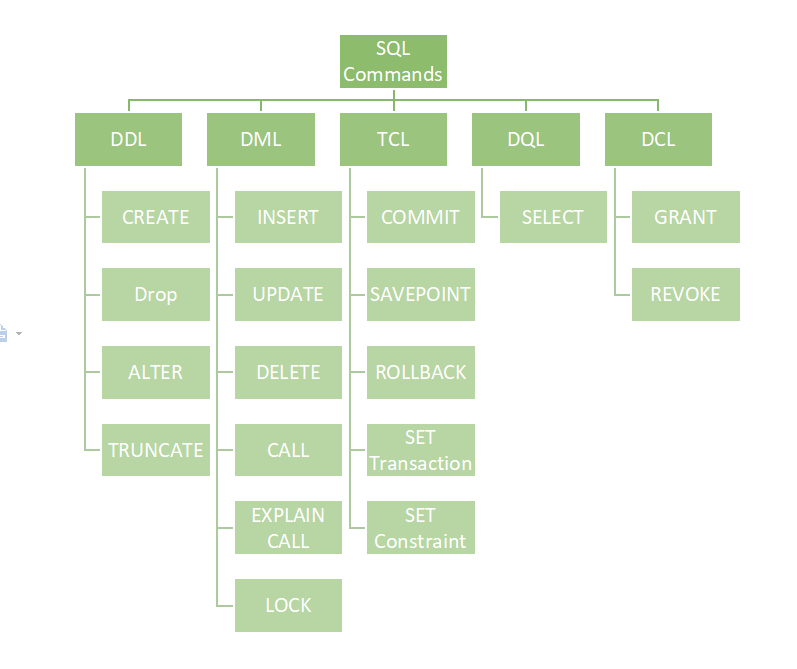
**You can only choose any two out of the three:**

* **CP (Consistency + Partition Tolerance): Data is correct, but might be unavailable during failure.**
* **CA (Consistency + Availability): Works only if there’s no network failure (rare).**
* **AP (Availability + Partition Tolerance): Always responds, but might show old/inconsistent data.**

**📌 Simple Example:**

**Imagine a shared Google Doc being edited by friends in different locations:**

* **If they always see the same content (Consistency) and can keep editing it (Availability), that’s great — but if the network breaks, it won’t work (no Partition Tolerance).**
* **If it keeps working during network issues (Partition Tolerance) and everyone can keep using it (Availability), changes may not sync right away (no Consistency).**



[**https://chatgpt.com/share/687df449-b6f8-800b-8835-00cf06917cba---sql**](https://chatgpt.com/share/687df449-b6f8-800b-8835-00cf06917cba---sql)

**https://chatgpt.com/share/687df3ba-3704-800b-b819-3999fb2a1070--hr**

**Coding Question:**

**duplicate--find,remove**

**second highest**