CURD

Contents

[Introduction to CRUD 2](#_Toc186142894)

[Creating a Database and Tables 2](#_Toc186142895)

[Inserting Data into Tables 3](#_Toc186142896)

[UPDATE Operation 5](#_Toc186142897)

[Deleting Data 7](#_Toc186142898)

[Delete vs. Truncate vs. Drop 8](#_Toc186142899)

[Read Data 10](#_Toc186142900)

[Student Table 12](#_Toc186142901)

[DISTINCT and SELECT Queries in SQL 15](#_Toc186142902)

[Operators 20](#_Toc186142903)

[SQL Null Handling 23](#_Toc186142904)

[ORDER BY 26](#_Toc186142905)

[LIMIT Clause 30](#_Toc186142906)

# Introduction to CRUD

* **CRUD** stands for four basic operations performed on stored data:
  + **C**reate: Inserting new records into a database.
  + **R**ead: Fetching or querying data.
  + **U**pdate: Modifying existing data.
  + **D**elete: Removing data from the database.
* Among these, **Read (R)** is the most comprehensive and will be discussed in a future session.

**SQL Overview**

* SQL (Structured Query Language) is used for:
  + Querying data.
  + Manipulating data (insertion, updates, deletions).
  + Defining data (creating tables, specifying data types and constraints).

## Creating a Database and Tables

1. **Creating a Database**:

CREATE DATABASE database\_name;

1. **Creating a Table**:

|  |
| --- |
| CREATE TABLE table\_name (      column\_name1 data\_type [constraint],      column\_name2 data\_type [constraint],      ...,      PRIMARY KEY (column\_name1, column\_name2, ...)  ); |

* + Define each column with:
    - **Name**
    - **Data Type** (e.g., INT, VARCHAR(50), BIGINT)
    - **Constraints**:
      * **NOT NULL**: Disallows NULL values in the column.
      * **UNIQUE**: Ensures no duplicate values in the column.
      * **DEFAULT**: Assigns a default value if no value is provided.
      * **PRIMARY KEY**: Unique identifier for each row, combining one or more columns.
        + A **composite key** consists of multiple columns that together form a unique identifier.

**Example: Creating a students Table**

|  |
| --- |
| CREATE TABLE students (      ID BIGINT NOT NULL,      Name VARCHAR(20) NOT NULL,      Email VARCHAR(50) NOT NULL UNIQUE,      PhoneNumber INT,      PRIMARY KEY (ID)  ); |

* **ID**: Primary key.
* **Email**: Must be unique.
* **PhoneNumber**: Optional.

## Inserting Data into Tables

* **Syntax**:

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

* You can insert multiple rows in one query:

INSERT INTO table\_name (column1, column2, ...) VALUES (value1, value2, ...), (value3, value4, ...);

* If column names are omitted, values must follow the table’s column order:

INSERT INTO table\_name VALUES (value1, value2, ...);

* + **Not Recommended**: Prone to errors and confusion.

**Good Practice for Insert Queries**

* Specify column names to:
  + Avoid confusion.
  + Improve readability.
  + Reduce errors.

**Example: Insert into students Table**

INSERT INTO students (ID, Name, Email, PhoneNumber)

VALUES (1, 'Alice', 'alice@example.com', 9996203771),

       (2, 'Bob', 'bob@example.com', NULL);

* **Partial Inserts**: If not all columns are specified, omitted columns must allow NULL or have a DEFAULT value.

INSERT INTO students (ID, Name) VALUES (3, 'Charlie');

* + Here, Email and PhoneNumber will be NULL or take their default values.

**Constraints Recap**

1. **NOT NULL**: Ensures a value must be provided.
2. **UNIQUE**: No duplicate values allowed.
3. **DEFAULT**: Provides a fallback value.
4. **PRIMARY KEY**:
   * Combines uniqueness and NOT NULL.
   * Can consist of one or more columns (composite key).

**Composite Primary Keys**

* Example Scenario:
  + Table: addresses
  + Columns: HouseNumber, Street, City.
  + **Composite Primary Key**:

PRIMARY KEY (HouseNumber, Street, City)

* + Ensures that the combination of HouseNumber, Street, and City is unique.

**Key Points:**

* A table can have only one primary key, but it can span multiple columns.
* Composite keys ensure uniqueness across the specified columns collectively.

**Summary**

1. CRUD operations are fundamental to data management.
2. SQL is not just for querying but also for defining and manipulating data.
3. Good practices like specifying column names in insert statements help avoid confusion and errors.
4. Constraints like NOT NULL, UNIQUE, and DEFAULT enforce data integrity.
5. Composite primary keys allow unique identification using multiple columns.

**CRUD Operations Overview**

* **CRUD**: Acronym for Create, Read, Update, Delete.
  + **C**: Create
  + **R**: Read
  + **U**: Update
  + **D**: Delete
* **Significance**:
  + Common operations performed on stored data.
  + CRUD principles extend to HTTP requests:
    - Create → POST
    - Read → GET
    - Update → PUT/PATCH
    - Delete → DELETE
  + CRUD operations form the foundation of a software engineer's work.

## UPDATE Operation

**SQL Syntax for UPDATE**

* General structure:

|  |
| --- |
| UPDATE table\_name  SET column1 = value1, column2 = value2, ...  WHERE condition; |

* Steps to update data:
  1. Specify the table to update: UPDATE table\_name.
  2. Define the new values with SET.
  3. Use WHERE to filter rows for update (optional but crucial to avoid affecting all rows).

**Example Problem**

* **Problem Statement**:
  + Update the PSP column by adding 10 for all rows where batch\_id = 2 in a students table.
* **Columns in students table**:
  + id, name, PSP, batch\_id
* **SQL Query**:

|  |
| --- |
| UPDATE students  SET PSP = PSP + 10  WHERE batch\_id = 2; |

**Important Notes on SQL Behaviour**

* SQL queries operate **row by row**.
  + Without a WHERE clause, all rows will be affected.
* Example:
  + **Table before query**:

| **ID** | **Name** | **PSP** | **Batch\_ID** |
| --- | --- | --- | --- |
| 1 | Alice | 60 | 2 |
| 2 | Bob | 70 | 1 |
| 3 | Carol | 50 | 2 |

* + **Query**:

UPDATE students

SET PSP = PSP + 10

WHERE batch\_id = 2;

* + **Table after query**:

| **ID** | **Name** | **PSP** | **Batch\_ID** |
| --- | --- | --- | --- |
| 1 | Alice | 70 | 2 |
| 2 | Bob | 70 | 1 |
| 3 | Carol | 60 | 2 |

**Translation into a Programming Language**

* Conceptual representation in Python-like pseudocode:

for student in students:

    if student.batch\_id == 2:

        student.PSP += 10

* Explanation:
  1. Iterate through each row in the students dataset.
  2. Check if batch\_id equals 2.
  3. Update the PSP value by adding 10 for matching rows.

**Key Takeaways**

1. **SQL Update Queries**:
   * Always use a WHERE clause to prevent unintended updates to all rows.
   * The SET clause specifies the columns and new values.
2. **Programming Equivalent**:
   * Mimics row-by-row updates using loops and conditions.
3. **Understanding CRUD**:
   * Fundamental to data management in databases and HTTP interactions.

## Deleting Data

**Deleting Data in SQL**

* SQL operates row by row to perform operations like deleting data.
* **Basic Delete Query Syntax:**

DELETE FROM table\_name;

* + If no condition is provided, all rows in the table will be deleted.
  + This is typically undesirable unless you want to clear the entire table.
* **Conditioned Delete Syntax:**

DELETE FROM table\_name WHERE condition;

* + Deletes only rows that match the specified condition.
  + Example: Delete students from a specific batch.

**Impact of Delete on Data Relationships**

* Other tables linked via **foreign key constraints** remain unaffected unless explicitly cascaded.
* Deleting data in one table does not delete linked records unless cascades are configured.
* Developers must manage these relationships to maintain consistency.

**Altering Table Structure**

* **Deleting a Column:** Not possible via DELETE. Use ALTER TABLE to change table structure:

ALTER TABLE table\_name DROP COLUMN column\_name;

* **Renaming Columns:** Use ALTER TABLE:

ALTER TABLE table\_name RENAME COLUMN old\_name TO new\_name;

## Delete vs. Truncate vs. Drop

1. **DELETE:**
   * Removes rows one by one based on the condition.
   * Table structure and schema remain intact.
   * Can use WHERE to specify conditions.
   * Data can be recovered using **commit logs**.
   * Slower due to row-by-row operation.
2. **TRUNCATE:**
   * Removes all rows from the table instantly.
   * Deletes the entire table storage and recreates a new empty table with the same schema.
   * Does not log individual row deletions, making data recovery impossible.
   * Faster as it does not process rows individually.
3. **DROP:**
   * Completely deletes the table, including its schema.
   * No table or data remains post-operation.
   * Example usage: DROP TABLE table\_name;

**Comparison Analogy:**

* **DELETE:** Cleaning a dirty laptop using software row by row.
* **TRUNCATE:** Factory resetting the laptop to start fresh.
* **DROP:** Destroying the laptop entirely.

**Choosing Between Delete and Truncate**

* **DELETE:** Use when:
  + You need selective deletion.
  + Recovering data is essential.
* **TRUNCATE:** Use when:
  + Clearing all data quickly is required.
  + Recovery is not a priority.

**Handling Data Inconsistencies**

* **Foreign Key Constraints:**
  + Protect database integrity by preventing operations that violate constraints.
  + Attempting to delete or truncate a table referenced by another table with foreign key constraints will throw an error.

**Solutions to Delete a Referenced Table:**

1. Modify foreign key constraints (e.g., allow cascading deletes).
2. Delete dependent rows first (e.g., delete students before deleting their batch).
3. Drop foreign key constraints temporarily (not recommended).

**Key Points to Remember**

* Operations like DELETE, TRUNCATE, and DROP must align with your requirements for data recovery, performance, and database integrity.
* Cascading effects and constraints must be planned carefully to avoid unexpected errors or data inconsistencies.
* Use ALTER TABLE for structural changes to the table.

**Recovery and Commit Logs:**

* **DELETE** operations are logged in commit logs, allowing potential recovery.
* **TRUNCATE** and **DROP** do not log individual row deletions, making recovery impossible.

## Read Data

**Overview**

* **SELECT**: Keyword used to retrieve data.
* **FROM**: Specifies the table from which data is fetched.
* **WHERE**: Adds filtering conditions.
* **ORDER BY**: Sorts the output in a specific order.
* **LIMIT**: Restricts the number of rows returned.

**SQL Query Execution Logic**

1. **Step-by-step process**:
   * For each row in the table (‘R’), evaluate if the condition in the WHERE clause is true.
   * If true, add the entire row to an intermediate answer table.
   * After processing all rows, sort the answer table based on the ORDER BY clause.
   * Finally, filter the output to include only the columns specified in the SELECT statement.
2. **Intermediate Answer Table**:
   * Stores the entire row, not just the selected columns.
   * Ensures sorting and additional operations can be performed on columns not included in the SELECT output.

**Key SQL Keywords**

**SELECT**

* **Purpose**: Defines the columns to retrieve.
* **Usage**: Works like a print statement; outputs the specified data.
* Example:

SELECT name FROM students;

* **Advanced Operations**:
  + Use \* to select all columns:
    - Example: SELECT \* FROM students;
  + Apply operations on columns:
    - Example: SELECT PSP \* batch\_id FROM students WHERE batch\_id = 2;
* **DISTINCT**:
  + Eliminates duplicate values from the result set.
  + Example: SELECT DISTINCT(name) FROM students;
  + Note: Avoid using DISTINCT for large datasets due to high time and space complexity.

**FROM**

* **Purpose**: Specifies the source table for the data.
* Example:
  + FROM students

**WHERE**

* **Purpose**: Filters rows based on conditions.
* Example:
  + WHERE batch\_id = 2;

**ORDER BY**

* **Purpose**: Sorts the output rows.
* **Default**: Ascending order (unless specified otherwise).
* **Sorting by Multiple Columns**:
  + Example: ORDER BY PSP \* batch\_id;

**LIMIT**

* **Purpose**: Restricts the number of rows returned by the query.
* Example:
  + LIMIT 10;
  + Retrieves only the first 10 rows of the result set.

**General Notes:**

* By default, rows are sorted by the primary key if no ORDER BY clause is present.
* Use caution with complex expressions in ORDER BY clauses for performance reasons.

**Example Problem: SQL Query and Output**

**Problem**

* Retrieve the names of students in batch 2, sorted by PSP.
* Sample Data:

## Student Table

| **ID** | **Name** | **PSP** | **Batch ID** |
| --- | --- | --- | --- |
| 1 | F | 30 | 1 |
| 2 | G | 20 | 1 |
| 3 | I | 40 | 2 |
| 4 | D | 25 | 1 |
| 5 | E | 11 | 2 |
| 6 | A | 13 | 3 |
| 7 | C | 41 | 2 |
| 8 | K | 70 | 1 |
| 9 | L | 81 | 2 |

**Steps to Solve**

1. Identify rows where Batch\_ID = 2:
   * Rows: (3, I, 40, 2), (5, E, 11, 2), (7, C, 41, 2), (9, L, 81, 2)
2. Sort these rows by PSP (ascending):
   * Sorted Rows: (5, E, 11, 2), (3, I, 40, 2), (7, C, 41, 2), (9, L, 81, 2)
3. Select only the Name column from the sorted rows:
   * Output: E, I, C, L

**SQL Query**

SELECT name FROM students

WHERE batch\_id = 2

ORDER BY PSP;

**Translation into a Programming Language**

* Conceptual representation in Python-like pseudocode:

|  |
| --- |
| # Step 1: Define the students table  # (list of dictionaries, each dictionary is a row)  students\_table = [      {"ID": 1, "Name": "F", "PSP": 30, "Batch\_ID": 1},      {"ID": 2, "Name": "G", "PSP": 20, "Batch\_ID": 1},      {"ID": 3, "Name": "I", "PSP": 40, "Batch\_ID": 2},      {"ID": 4, "Name": "D", "PSP": 25, "Batch\_ID": 1},      {"ID": 5, "Name": "E", "PSP": 11, "Batch\_ID": 2},      {"ID": 6, "Name": "A", "PSP": 13, "Batch\_ID": 3},      {"ID": 7, "Name": "C", "PSP": 41, "Batch\_ID": 2},      {"ID": 8, "Name": "K", "PSP": 70, "Batch\_ID": 1},      {"ID": 9, "Name": "L", "PSP": 81, "Batch\_ID": 2}  ]  # Step 2: Apply the WHERE clause to filter rows  filtered\_rows = []  for row in students\_table:      if row["Batch\_ID"] == 2:          filtered\_rows.append(row)  # Step 3: Apply the ORDER BY clause  # to sort the filtered rows by PSP  sorted\_rows = sort(filtered\_rows, key=lambda x: x["PSP"])  # Step 4: Use SELECT to extract  # only the "Name" column  result = []  for row in sorted\_rows:      result.append(row["Name"])  # Final result  return result |

**Explanation:**

1. **Step 1: Initialize the Table**
   * The table is represented as a list of dictionaries where each dictionary corresponds to a row with column names as keys.
2. **Step 2: Apply WHERE**
   * Loop through each row and filter rows where Batch\_ID = 2.
3. **Step 3: Apply ORDER BY**
   * Sort the filtered rows by PSP in ascending order using a sorting function.
4. **Step 4: Apply SELECT**
   * Extract only the Name column from each sorted row.
5. **Final Output:**
   * Return a list of names based on the processed rows.

**Implementation Details**

**Data Storage and Retrieval**

* Data is not stored as a simple table but is optimized using structures like trees (e.g., for indexing).
* Direct row-by-row evaluation would be inefficient; indexing improves query performance.

**Efficiency Considerations**

* Avoid DISTINCT on large datasets unless necessary.
* Understand the impact of ORDER BY on performance, especially with complex expressions or large datasets.

**Summary of Key Insights**

1. SQL execution involves an intermediate step of storing complete rows for flexibility in sorting and filtering.
2. SELECT acts like a print statement but is typically combined with other clauses like FROM, WHERE, and ORDER BY.
3. DISTINCT helps eliminate duplicates but can be computationally expensive.
4. Efficient querying relies on understanding underlying data structures and avoiding unnecessary operations.

## DISTINCT and SELECT Queries in SQL

**Overview**

This document analyses and summarizes the key points discussed in a lecture on how the SELECT and DISTINCT keywords work in SQL, particularly focusing on their behaviour with single and multiple columns.

**Key Concepts and Definitions**

**SELECT Keyword**

* **Purpose**: Used to retrieve specific columns or all columns from a table.
* Syntax:

SELECT column\_name FROM table\_name;

* To retrieve all columns:

SELECT \* FROM table\_name;

**DISTINCT Keyword**

* **Purpose**: Ensures that duplicate rows in the query result are eliminated based on the specified columns.
* Syntax:

SELECT DISTINCT column\_name FROM table\_name;

* Works on the entire row when multiple columns are specified:

SELECT DISTINCT column1, column2 FROM table\_name;

* Removes duplicate rows **only from the final output**, not just individual columns.

**Example Scenarios**

**Basic Usage**

* Query:

SELECT DISTINCT state FROM customers;

* + This retrieves a list of unique states from the customers table.
  + Example Output:

| **ID** | **State** |
| --- | --- |
| 1 | MA |
| 2 | VA |
| 3 | CO |
| 4 | CA |
| 5 | GA |

**DISTINCT with Multiple Columns**

* Query:

SELECT DISTINCT state, first\_name FROM customers;

* + This considers the combination of state and first\_name as a unique pair.
  + If there are two rows with the same state but different first\_name, both will be included in the result.
  + Example Output:

| **State** | **First Name** |
| --- | --- |
| MA | John |
| GA | Alice |
| GA | Bob |
| CO | Emma |
| CA | Michael |

* + Explanation:
    - The result contains both rows for GA because the first\_name values are different.

**DISTINCT and Duplicate Rows**

* If there are duplicate rows across all selected columns, DISTINCT removes them.

SELECT DISTINCT state, first\_name FROM customers;

* + When two rows have identical values for both state and first\_name, only one is included in the result.

**DISTINCT on a Single Column**

* Query:

SELECT DISTINCT state FROM customers;

* + This retrieves all unique state values. Even if the same state appears multiple times with different first\_name values, it will only appear once in the result.

**Duplicate Removal Logic**

* DISTINCT applies to the final output, ensuring no duplicate rows remain.
* For example:
  + Without DISTINCT:

| **State** | **First Name** |
| --- | --- |
| GA | Alice |
| GA | Bob |
| GA | Alice |

* + With DISTINCT:

| **State** | **First Name** |
| --- | --- |
| GA | Alice |
| GA | Bob |

**Complex Scenarios**

**Changing Data and Effects on DISTINCT**

1. **Scenario**: Changing a customer’s state to create duplicates.
   * Update Query:

UPDATE customers SET state = 'GA' WHERE customer\_id = 2;

* + Result: Two rows now have state = GA, which affects the DISTINCT result.

1. **Query**:

SELECT DISTINCT state, first\_name FROM customers;

* + Behaviour:
    - Rows with unique combinations of state and first\_name are included.
    - If two rows have identical values for both columns, only one is kept.

1. **Example with Modified Data**:

| **State** | **First Name** |
| --- | --- |
| GA | Alice |
| GA | Alice |

* + Applying DISTINCT:

| **State** | **First Name** |
| --- | --- |
| GA | Alice |

**DISTINCT and NULL Values**

* DISTINCT treats NULL values as equal.
* Example:

SELECT DISTINCT column\_name FROM table\_name;

* + If column\_name contains NULL, it appears only once in the result, regardless of how many NULL values exist.

**DISTINCT with GROUP BY**

* **Scenario**: Retrieve one customer and their state from each state.
* Query:

SELECT state, MAX(first\_name) AS representative\_customer

FROM customers

GROUP BY state;

* + Explanation:
    - GROUP BY groups all rows by state.
    - MAX(first\_name) selects one representative name per group.
  + Example Output:

| **State** | **Representative\_Customer** |
| --- | --- |
| GA | Bob |
| MA | John |

**DISTINCT vs GROUP BY**

* DISTINCT removes duplicates from the output.
* GROUP BY aggregates rows and allows calculations (e.g., COUNT, MAX).

**Pseudo Code Explanation: Understanding DISTINCT Logic**

1. Input Table:

| **ID** | **State** | **First Name** |
| --- | --- | --- |
| 1 | MA | John |
| 2 | GA | Alice |
| 3 | GA | Bob |
| 4 | CO | John |

1. Query:

SELECT DISTINCT state, first\_name FROM customers;

1. Steps:
   * Retrieve rows from the table.
   * For each row, check if the combination of state and first\_name has already been added to the output.
   * If not, include the row.
   * Example Result:

| **State** | **First Name** |
| --- | --- |
| MA | John |
| GA | Alice |
| GA | Bob |
| CO | John |

**Key Takeaways**

1. DISTINCT works on the entire row or the specified columns.
2. When using DISTINCT with multiple columns, it considers the combination of values in those columns.
3. Null values are treated as duplicates by DISTINCT.
4. GROUP BY can achieve similar results to DISTINCT but is more powerful for aggregation.
5. Always analyse the output logic when combining DISTINCT with other operations.

## Operators

**1. Purpose of WHERE Clause**

* The WHERE clause is used to specify a condition.
* Rows that match the condition are included in the result set.

**2. Comparison Operators**

* Used to compare values in SQL.
* Common operators:
  + = : Equal to
  + > : Greater than
  + < : Less than
  + >= : Greater than or equal to
  + <= : Less than or equal to
  + <> or != : Not equal to

**Notes:**

* = is used for equality instead of == (as in programming languages).
* Both <> and != can be used for inequality.

**3. String Comparison**

* String comparison in SQL is **case insensitive** by default.
* Example:

SELECT \* FROM students WHERE state = 'Georgia';

* + Matches 'Georgia', 'georgia', 'GEORGIA', etc.

**4. Date Comparison**

* Dates can be compared using the same operators as numbers.
* Dates should be in the format: YYYY-MM-DD.
* Example:

SELECT \* FROM students WHERE date\_of\_birth < '1999-01-01';

**5. Combining Multiple Conditions**

* Use **AND** and **OR** to combine conditions:
  + AND: All conditions must be true.
  + OR: At least one condition must be true.
* Example:

SELECT \* FROM students WHERE PSP > 90 AND state = 'Georgia';

**Best Practice:**

* Always use parentheses () to avoid ambiguity and ensure clarity.
* Example:

SELECT \* FROM students WHERE (PSP > 31 AND state = 'HR') OR name = 'North';

**6. NOT Operator**

* Reverses the condition.
* Example:

SELECT \* FROM students WHERE NOT state = 'HR';

* + Equivalent to:

SELECT \* FROM students WHERE state <> 'HR';

**7. IN Operator**

* Used to check if a value matches any value in a list.
* Example:

SELECT \* FROM students WHERE batch\_name IN ('April 21', 'May 21');

* For negation, use **NOT IN**:

SELECT \* FROM students WHERE batch\_name NOT IN ('April 21', 'May 21');

**8. BETWEEN Operator**

* Used to check if a value is within a range (inclusive).
* Example:

SELECT \* FROM students WHERE PSP BETWEEN 40 AND 70;

**9. LIKE Operator**

* Used for pattern matching with strings.
* Wildcards:
  + % : Represents any number of characters (including zero).
  + \_ : Represents exactly one character.

**Examples:**

* Ends with 'A':

SELECT \* FROM students WHERE name LIKE '%A';

* Starts with 'DSML':

SELECT \* FROM students WHERE batch\_name LIKE 'DSML%';

* Contains 'April 21':

SELECT \* FROM students WHERE batch\_name LIKE '%April 21%';

* Exactly 2 characters, ending with 'A':

SELECT \* FROM students WHERE name LIKE '\_A';

**10. NULL Handling**

* NULL cannot be compared directly using = or !=.
* Use **IS NULL** or **IS NOT NULL** to check for null values.
* Example:

SELECT \* FROM students WHERE batch\_name IS NULL;

**11. Key Takeaways**

* Always use parentheses for clarity with AND/OR.
* Use **IN** for checking against multiple values in the same column.
* Use **LIKE** for pattern matching with wildcards (% and \_).
* BETWEEN is inclusive; use it as a shorthand for range conditions.
* Handle NULL explicitly using **IS NULL** or **IS NOT NULL**.

## SQL Null Handling

**Key Concepts of NULL in SQL**

1. **What is NULL?**
   * NULL represents missing or undefined values in SQL.
   * It is not equivalent to 0, an empty string, or any other value.
2. **Comparison with NULL:**
   * NULL cannot be compared using standard comparison operators like = or !=.
   * Any comparison involving NULL results in NULL (unknown), which is treated as false in conditional checks.
3. **Checking for NULL:**
   * Use IS NULL to check if a value is NULL.
   * Use IS NOT NULL to check if a value is not NULL.

Example:

SELECT \* FROM students WHERE batch\_id IS NULL;

SELECT \* FROM students WHERE batch\_id IS NOT NULL;

1. **Behavior in Queries:**
   * Filters like WHERE batch\_id = NULL or WHERE batch\_id != NULL will not work because NULL = NULL is not true.
   * Always use IS NULL or IS NOT NULL for filtering rows with or without NULL values.

**Demonstration with Examples**

1. **Finding Rows with NULL Values:**

SELECT \* FROM students WHERE batch\_id IS NULL;

1. **Finding Rows with Non-NULL Values:**

SELECT \* FROM students WHERE batch\_id IS NOT NULL;

1. **Why NULL = NULL is Undefined:**
   * NULL signifies the absence of a value; comparing two absent values does not yield true.
   * Example:

SELECT \* FROM customers WHERE phone = NULL; -- Returns no rows

SELECT \* FROM customers WHERE phone != NULL; -- Returns no rows

1. **Pattern Matching with NULL:**
   * If a column contains NULL, filtering using comparison or patterns (e.g., LIKE) will not include those rows unless explicitly handled with IS NULL.

**Advanced Use Cases**

1. **Filtering Rows Excluding Specific Values (Handling NULL):**

SELECT \* FROM students WHERE marks != 9 OR marks IS NULL;

* + Rows with NULL values need explicit handling.

1. **Combining Conditions with IS NULL and NOT NULL:**
2. SELECT \* FROM employees

WHERE department\_id IS NULL OR salary > 5000;

1. **Behaviour in GROUP BY:**
   * GROUP BY can group rows with NULL values as a separate group.
   * Example:

SELECT department\_id, COUNT(\*)

FROM employees

GROUP BY department\_id;

1. **Case Sensitivity in SQL:**
   * SQL is generally case-insensitive for keywords (e.g., SELECT, WHERE).
   * Table and column names may be case-sensitive depending on the database.

**Practical Exercises**

1. **Basic NULL Handling:**
   * Find students without a batch:

SELECT \* FROM students WHERE batch\_id IS NULL;

* + Find students with a batch:

SELECT \* FROM students WHERE batch\_id IS NOT NULL;

1. **Filtering on NULL and Other Conditions:**
   * Filter rows with a specific condition while handling NULL:

SELECT \* FROM students WHERE marks != 90 OR marks IS NULL;

1. **Understanding Rollbacks in Updates:**
   * When using UPDATE with a condition involving NULL, ensure proper handling:

UPDATE students SET grade = 'A' WHERE marks > 90 AND marks IS NOT NULL;

**Miscellaneous**

1. **Batch Update vs. IN Clause:**

IN clause updates can handle multiple conditions in one statement.

* + Batch updates group multiple queries for performance improvement.

1. **Error Handling with NULL:**

Always account for NULL values in comparisons to avoid unexpected results.

1. **Exercises on LeetCode:**
   * Solve easy SQL questions on LeetCode to practice concepts without joins.

**Summary**

* NULL values require special handling using IS NULL or IS NOT NULL.
* Avoid using = or != for comparisons involving NULL.
* Use GROUP BY and conditional filtering carefully when NULL values are present.
* Practice with real-world scenarios to solidify understanding.

## ORDER BY

#### **Purpose of** ORDER BY

* The ORDER BY clause specifies how the rows in the output of a query should be ordered.
* By **default**, rows are ordered by the **primary key** of the table.

#### **How SQL Queries Process Rows**

1. Each row is evaluated to check if it satisfies the WHERE condition.
2. Rows meeting the condition are added to the output.
3. Without an ORDER BY clause, the output rows are ordered by the primary key.

#### **Syntax**

|  |
| --- |
| SELECT      column\_names  FROM      table\_name  WHERE      condition  ORDER BY      column\_name [ASC | DESC]; |

* ASC: Orders rows in ascending order (default).
* DESC: Orders rows in descending order.
* **Default Sorting Order**:
  + If ASC or DE`SC is not specified, rows are sorted in **ascending** order.

#### **Examples**

1. **Default Order (by Primary Key):**

|  |
| --- |
| SELECT \*  FROM students  WHERE PSP > 30      AND Batch = 2; |

Output will be ordered by the primary key.

1. **Order by Specific Column:**

|  |
| --- |
| SELECT \*  FROM students  WHERE PSP > 30      AND Batch = 2  ORDER BY PSP DESC; |

Orders rows by the PSP column in descending order.

1. **Multiple Columns:**

|  |
| --- |
| SELECT \*  FROM students  ORDER BY PSP DESC, Name ASC; |

* Orders by PSP in descending order.
* For rows with the same PSP, it orders by Name in ascending order.

#### **Key Details**

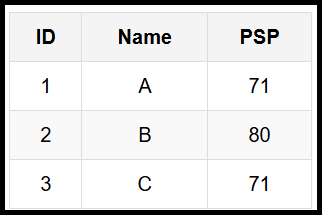
1. **Multiple Column Sorting:**
   * Sorting is hierarchical.
   * The first column is used for ordering.
   * If there’s a tie, the next column is used as a tiebreaker.
2. **Default Tie-Breaking Rule:**
   * If two rows are tied on all specified columns, the **primary key** is used as the final tiebreaker.
3. **Sorting by Columns Not in** SELECT**:**
   * You can order by a column even if it is not included in the SELECT clause.
   * This works because the entire row is considered during query execution.

#### **Behavior Examples**

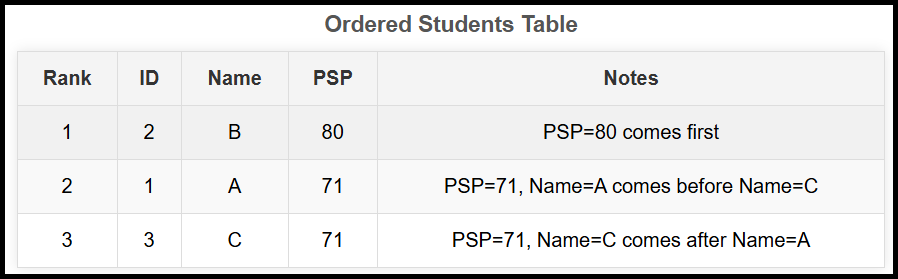
1. **Ascending and Descending Order:**
   * Query:

|  |
| --- |
| SELECT Name  FROM students  ORDER BY PSP DESC, Name ASC; |

* + Data:

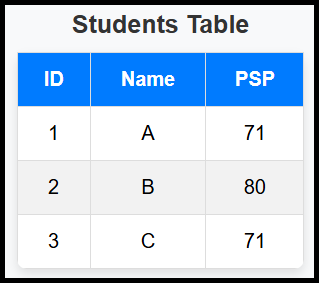


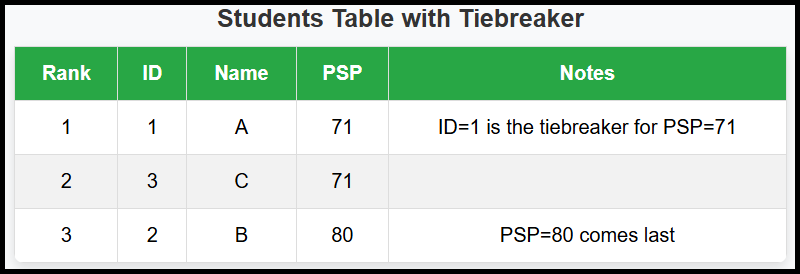
* + Result:



1. **Using Only** PSP**:**
   * Query:

|  |
| --- |
| SELECT Name  FROM students  ORDER BY PSP; |

* + Data:
  + Result:



#### **Conclusion**

* ORDER BY is crucial for controlling the sorting of query results.
* Defaults:
  + Sorting is ascending unless specified otherwise.
  + Tiebreakers are determined by subsequent columns or the primary key.
* Flexibility:
  + Can sort by columns not included in the SELECT clause.

## LIMIT Clause

* The LIMIT clause restricts the number of rows returned in a query result.
* Useful for:
  + Preventing large datasets from overwhelming systems.
  + Optimizing query performance by transferring fewer rows.
  + Implementing pagination in web applications.

#### **Syntax**

|  |
| --- |
| SELECT column\_names  FROM table\_name  WHERE condition  LIMIT [offset,] row\_count; |

* **row\_count**: Specifies the maximum number of rows to return.
* **offset (**optional**)**: Specifies the number of rows to skip before starting to return rows.

#### **Examples**

1. **Basic LIMIT Usage:**

|  |
| --- |
| SELECT \* FROM students  LIMIT 1; |

* + Returns only the first row of the result set.

1. **LIMIT with OFFSET:**

|  |
| --- |
| SELECT \* FROM products  WHERE title LIKE '%T-shirt%'  LIMIT 5, 10; |

* + Skips the first 5 rows and returns the next 10 rows (rows 6 to 15).

1. **Pagination with LIMIT:**
   * For page 1 (showing 20 rows):

|  |
| --- |
| SELECT \* FROM products  LIMIT 0, 20; |

* + For page 2:

SELECT \* FROM products

LIMIT 40, 20;

* + For page 3:

SELECT \* FROM products

LIMIT 40, 20;

#### **Common Use Cases**

1. **Search Results:**
   * Websites like Amazon or Flipkart use LIMIT for pagination (e.g., showing 20 items per page).
2. **Large Data Sets:**
   * Prevents transferring billions of rows at once, which can cause slow query execution and high data transfer costs.
3. **Efficient Data Retrieval:**
   * Fetches only a subset of data for analysis or display.

#### **Key Concepts**

1. **OFFSET and Row Count:**
   * The first parameter after LIMIT is the **offset** (number of rows to skip).
   * The second parameter is the **row count** (number of rows to fetch).
2. **Behavior with New Data:**
   * Inconsistent results can occur if new data is added between queries (e.g., during pagination).
3. **Common Pagination Query:**
   * For a website showing 20 results per page:
     + Page 1: LIMIT 0, 20
     + Page 2: LIMIT 20, 20
     + Page 3: LIMIT 40, 20

#### **Potential Issues**

1. **Data Overlap:**
   * Rows at the boundary of pages can appear in both pages due to data changes during query execution.
   * Example: A new row inserted between page 1 and page 2 can cause duplication of rows.
2. **Large Offsets:**
   * Fetching rows with high offsets can be inefficient, as the database still scans skipped rows.

#### **Best Practices**

1. **Limit Data Size:**
   * Use LIMIT to avoid overwhelming the application or network with large data sets.
2. **Combine with Sorting:**
   * Always use ORDER BY to ensure consistent and predictable results:

|  |
| --- |
| SELECT \* FROM products  WHERE title LIKE '%T-shirt%'  ORDER BY price ASC  LIMIT 10; |

1. **Track Offsets:**
   * Calculate offsets dynamically in web applications based on page number and page size.

#### **Example Use Case: Pagination in E-Commerce**

* **Scenario:**
  + A user searches for "T-shirts" on Flipkart.
  + Database query:

|  |
| --- |
| SELECT \*  FROM products  WHERE title LIKE '%T-shirt%'  LIMIT 0, 20; |

* + First 20 results are displayed.
  + For the second page, query changes to:

|  |
| --- |
| SELECT \*  FROM products  WHERE title LIKE '%T-shirt%'  LIMIT 20, 20; |

#### **Key Takeaways**

1. LIMIT is essential for handling large datasets efficiently.
2. Pagination is achieved using LIMIT with an offset.
3. Be cautious of potential data inconsistencies with dynamic datasets.
4. Always use ORDER BY with LIMIT for consistent results.