1. Difference between delete, drop and truncate?

Here’s a clear comparison of DELETE, DROP, and TRUNCATE in SQL:

**✅ 1. DELETE**

| **Feature** | **Description** |
| --- | --- |
| **Purpose** | Deletes **some or all rows** from a table **based on a condition** (WHERE). |
| **WHERE Clause** | ✅ Yes (optional). |
| **Rollback** | ✅ Yes (if inside a transaction). |
| **Triggers** | ✅ Fires triggers (e.g., ON DELETE). |
| **Performance** | Slower (row-by-row). |
| **Auto-increment** | Keeps the counter unchanged (unless reset explicitly). |

**Example**:

DELETE FROM employees WHERE department = 'HR';

**✅ 2. TRUNCATE**

| **Feature** | **Description** |
| --- | --- |
| **Purpose** | Deletes **all rows** from a table. |
| **WHERE Clause** | ❌ Not allowed. |
| **Rollback** | ❌ No (in many RDBMS; irreversible in MySQL, but can be rolled back in PostgreSQL if inside a transaction). |
| **Triggers** | ❌ Does not fire ON DELETE triggers (in most RDBMS). |
| **Performance** | ✅ Very fast (deallocates data pages). |
| **Auto-increment** | ✅ Resets in MySQL (may differ in other databases). |

**Example**:

TRUNCATE TABLE employees;

**✅ 3. DROP**

| **Feature** | **Description** |
| --- | --- |
| **Purpose** | Completely **removes the table** structure and data. |
| **WHERE Clause** | ❌ Not applicable. |
| **Rollback** | ❌ No. |
| **Triggers** | ❌ Not applicable. |
| **Performance** | N/A (removes object). |
| **Auto-increment** | N/A (table is gone). |

**Example**:

DROP TABLE employees;

**🔁 Quick Comparison Table**

| **Feature** | **DELETE** | **TRUNCATE** | **DROP** |
| --- | --- | --- | --- |
| Removes rows? | ✅ Yes | ✅ Yes (all) | ✅ Removes entire table |
| Uses WHERE clause? | ✅ Yes | ❌ No | ❌ No |
| Rollback possible? | ✅ Yes | ❌ Often No | ❌ No |
| Triggers executed? | ✅ Yes | ❌ No | ❌ No |
| Affects structure? | ❌ No | ❌ No | ✅ Yes |
| Speed | 🐢 Slower | ⚡ Faster | ❌ Not applicable |

1. How to select top 3 rows in DB

SELECT TOP 3 \* FROM Customers;

SELECT column\_name(s)  
FROM table\_name  
WHERE ROWNUM <= number;

SELECT \* FROM Customers  
ORDER BY CustomerName DESC  
LIMIT 3;

1. How to find the min and max ?

SELECT MIN(Price)  
FROM Products;

SELECT MAX(Price)  
FROM Products;

1. How to find the sum of all the employees

SELECT SUM(Quantity)  
FROM OrderDetails;

1. Between

SELECT \* FROM Products

WHERE Price >= 10 AND Price <= 20;

Or

SELECT \* FROM Products

WHERE Price BETWEEN 10 AND 20;

1. Joins

Explain joins

###### ChatGPT said:

Joins in SQL are used to **combine rows from two or more tables** based on a related column between them.

## 🔗 Types of Joins Explained with Example:

Assume we have two tables:

### Employees

| **EmpID** | **Name** | **DeptID** |
| --- | --- | --- |
| 1 | Alice | 10 |
| 2 | Bob | 20 |
| 3 | Charlie | 30 |
| 4 | David | null |

### Departments

| **DeptID** | **DeptName** |
| --- | --- |
| 10 | HR |
| 20 | IT |
| 40 | Marketing |

### ✅ 1. ****INNER JOIN****

* Returns only the matching records from both tables.

SELECT e.Name, d.DeptName

FROM Employees e

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

**Result:**

| **Name** | **DeptName** |
| --- | --- |
| Alice | HR |
| Bob | IT |

### ✅ 2. ****LEFT JOIN**** (or LEFT OUTER JOIN)

* Returns all rows from the left table and matched rows from the right.
* If no match, NULL is returned for right table columns.

SELECT e.Name, d.DeptName

FROM Employees e

LEFT JOIN Departments d ON e.DeptID = d.DeptID;

**Result:**

| **Name** | **DeptName** |
| --- | --- |
| Alice | HR |
| Bob | IT |
| Charlie | NULL |
| David | NULL |

### ✅ 3. ****RIGHT JOIN**** (or RIGHT OUTER JOIN)

* Opposite of LEFT JOIN: returns all rows from the right table and matched from the left.

SELECT e.Name, d.DeptName

FROM Employees e

RIGHT JOIN Departments d ON e.DeptID = d.DeptID;

**Result:**

| **Name** | **DeptName** |
| --- | --- |
| Alice | HR |
| Bob | IT |
| NULL | Marketing |

### ✅ 4. ****FULL OUTER JOIN****

* Returns all rows from both tables. NULLs where there is no match.

SELECT e.Name, d.DeptName

FROM Employees e

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

**Result:**

| **Name** | **DeptName** |
| --- | --- |
| Alice | HR |
| Bob | IT |
| Charlie | NULL |
| David | NULL |
| NULL | Marketing |

### ✅ 5. ****CROSS JOIN****

* Returns **Cartesian product**: every row from table A joined to every row from table B.

SELECT e.Name, d.DeptName

FROM Employees e

CROSS JOIN Departments d;

**Result: 4 (Employees) × 3 (Departments) = 12 rows**

### Summary Table

| **Join Type** | **Matching Records** | **Unmatched from A** | **Unmatched from B** |
| --- | --- | --- | --- |
| INNER JOIN | ✅ | ❌ | ❌ |
| LEFT JOIN | ✅ | ✅ | ❌ |
| RIGHT JOIN | ✅ | ❌ | ✅ |
| FULL OUTER JOIN | ✅ | ✅ | ✅ |
| CROSS JOIN | All combinations | – | – |

Displaying manager and his manager using self join

SELECT

e.Name AS Employee,

m.Name AS Manager

FROM Employees e

Self JOIN Employees m ON e.ManagerID = m.EmpID;

Union vs Union All

The UNION operator selects only distinct values by default. To allow duplicate values, use UNION ALL:

………………………………………………………………………………………………………………………………………………………………………

Group by having order by

SELECT COUNT(CustomerID), Country  
FROM Customers  
GROUP BY Country  
HAVING COUNT(CustomerID) > 5  
ORDER BY COUNT(CustomerID) DESC;

………………………………………………………………………………………………………………………………………………

## 📚 PL/SQL Examples

### ✅ 1. Anonymous Block

DECLARE

v\_message VARCHAR2(50);

BEGIN

v\_message := 'Hello, PL/SQL!';

DBMS\_OUTPUT.PUT\_LINE(v\_message);

END;

### ✅ 2. IF-ELSE

DECLARE

v\_num NUMBER := 15;

BEGIN

IF v\_num > 10 THEN

DBMS\_OUTPUT.PUT\_LINE('Greater than 10');

ELSE

DBMS\_OUTPUT.PUT\_LINE('10 or less');

END IF;

END;

### ✅ 3. LOOP

DECLARE

i NUMBER := 1;

BEGIN

WHILE i <= 5 LOOP

DBMS\_OUTPUT.PUT\_LINE('Iteration: ' || i);

i := i + 1;

END LOOP;

END;

### ✅ 4. PROCEDURE

CREATE OR REPLACE PROCEDURE say\_hello(name IN VARCHAR2) AS

BEGIN

DBMS\_OUTPUT.PUT\_LINE('Hello, ' || name);

END;

**Usage:**

BEGIN

say\_hello('Ramu');

END;

### ✅ 5. FUNCTION

CREATE OR REPLACE FUNCTION get\_square(n IN NUMBER) RETURN NUMBER AS

BEGIN

RETURN n \* n;

END;

**Usage:**

BEGIN

DBMS\_OUTPUT.PUT\_LINE(get\_square(6));

END;

### ✅ 6. PACKAGE

CREATE OR REPLACE PACKAGE math\_pkg AS

FUNCTION square(n NUMBER) RETURN NUMBER;

FUNCTION cube(n NUMBER) RETURN NUMBER;

END math\_pkg;

CREATE OR REPLACE PACKAGE BODY math\_pkg AS

FUNCTION square(n NUMBER) RETURN NUMBER IS

BEGIN

RETURN n \* n;

END;

FUNCTION cube(n NUMBER) RETURN NUMBER IS

BEGIN

RETURN n \* n \* n;

END;

**Query Recap:**

SELECT salary

FROM (

SELECT salary,

DENSE\_RANK() OVER (ORDER BY salary DESC) AS rank

FROM Employees

) ranked

WHERE rank = 3;

**🧾 Sample Employees Table:**

| **employee\_id** | **name** | **salary** |
| --- | --- | --- |
| 1 | Alice | 5000 |
| 2 | Bob | 7000 |
| 3 | Charlie | 7000 |
| 4 | David | 6000 |
| 5 | Eve | 4000 |
| 6 | Frank | 5000 |

**🧠 Step-by-Step Execution:**

1. **Apply DENSE\_RANK() Over Salaries in Descending Order:**

sql

CopyEdit

SELECT salary,

DENSE\_RANK() OVER (ORDER BY salary DESC) AS rank

FROM Employees;

| **salary** | **rank** |
| --- | --- |
| 7000 | 1 |
| 7000 | 1 |
| 6000 | 2 |
| 5000 | 3 |
| 5000 | 3 |
| 4000 | 4 |

* DENSE\_RANK() gives the same rank to identical salaries.
* 7000 is the highest → rank 1
* 6000 → rank 2
* 5000 → rank 3
* 4000 → rank 4

1. **Filter the Row(s) Where rank = 3:**

SELECT salary

FROM (...)

WHERE rank = 3;

| **salary** |
| --- |
| 5000 |
| 5000 |

**✅ Final Output:**

| **salary** |
| --- |
| 5000 |
| 5000 |

This means **5000** is the **third highest salary**, and two employees earn it.

SELECT TOP 1 salary

FROM (

SELECT DISTINCT TOP 3 salary

FROM Employees

ORDER BY salary DESC

) AS temp

ORDER BY salary ASC;

**Second highest salary**

To find the **second highest salary department-wise**, you can use a similar approach as the first-highest, but filter for RANK = 2. Here's how:

### ✅ 1. Using RANK() or DENSE\_RANK() (to handle salary ties):

SELECT department\_id, employee\_name, salary

FROM (

SELECT department\_id, employee\_name, salary,

DENSE\_RANK() OVER (PARTITION BY department\_id ORDER BY salary DESC) AS rnk

FROM Employees

) ranked

WHERE rnk = 2;

Now also display department name  
  
  
SELECT d.department\_name, ranked.department\_id, ranked.employee\_name, ranked.salary

FROM (

SELECT department\_id, employee\_name, salary,

DENSE\_RANK() OVER (PARTITION BY department\_id ORDER BY salary DESC) AS rnk

FROM Employees

) ranked

JOIN Departments d ON ranked.department\_id = d.department\_id

WHERE ranked.rnk = 2;

……………………………………………………………………………………………………………………………………………….

Department wise total students

SELECT d.department\_name, COUNT(s.student\_id) AS student\_count

FROM Departments d

LEFT JOIN Students s ON d.department\_id = s.department\_id

GROUP BY d.department\_name;

………………………………………………………………………………………………………………………………………………………………………..

**Dirty Read** → Read uncommitted data.

 **Non-Repeatable Read** → Same row’s value **changed** between reads.

 **Phantom Read** → **New row(s)** appeared/disappeared between reads.

**SQL Isolation Levels (ANSI Standard)**

There are **4 standard levels** (from weakest → strongest):

**1. READ UNCOMMITTED**

* Transactions can **read uncommitted (dirty) data** from other transactions.
* Fastest, but least safe.
* Problems: **Dirty Reads, Non-Repeatable Reads, Phantom Reads**.

✅ Example: Reading a balance that is updated but not committed yet.

**2. READ COMMITTED (Default in Oracle, SQL Server, PostgreSQL)**

* Transactions can **only read committed data**.
* Prevents **dirty reads**, but still allows:
  + **Non-Repeatable Reads**
  + **Phantom Reads**

✅ Example: You read salary = 5000, another transaction commits salary = 6000, you read again → changed.

**3. REPEATABLE READ (Default in MySQL InnoDB)**

* Ensures that if a transaction reads the same row twice, it gets the **same value**.
* Prevents **Dirty Reads** and **Non-Repeatable Reads**.
* But **Phantom Reads** are still possible.

✅ Example: You query "all employees with salary > 5000". Another transaction inserts a new employee with salary 6000. When you query again, you see an extra row (phantom).

**4. SERIALIZABLE**

* Strongest isolation level.
* Transactions are executed as if they run **one after another (serially)**.
* Prevents **Dirty Reads, Non-Repeatable Reads, and Phantom Reads**.
* Safest, but slowest (uses locks on ranges).

✅ Example: If you select employees with salary > 5000, no one can insert new rows in that range until you finish.

### Difference between View and Materialized View?

| **Feature** | **View** | **Materialized View** |
| --- | --- | --- |
| Storage | No data stored | Data physically stored |
| Freshness | Always current | Can be stale |
| Refresh | Not needed | Needs refresh (manual/on commit/on schedule) |
| Performance | Can be slow on big queries | Faster for reporting |

-- Step 1: Create MV log on base table

CREATE MATERIALIZED VIEW LOG ON employee WITH ROWID, SEQUENCE (department, salary) INCLUDING NEW VALUES;

-- Step 2: Create MV with FAST refresh

CREATE MATERIALIZED VIEW dept\_salary\_fast\_mv

REFRESH FAST ON COMMIT

AS

SELECT department, SUM(salary) AS total\_salary

FROM employee

GROUP BY department;

…………………………………………………………………………………………………………………………………………………………….

Group By example

SELECT product\_id, SUM(amount) AS total\_sales, COUNT(\*) AS num\_orders

FROM orders

WHERE order\_date BETWEEN '2025-08-01' AND '2025-08-31'

GROUP BY product\_id

ORDER BY total\_sales DESC;

…………………………………………………………………………………………………………………………………………………………………………………………………..

Group by multiple columns

SELECT department, city, COUNT(\*) AS emp\_count

FROM employee

GROUP BY department, city;

WHERE + GROUP BY + HAVING + ORDER BY

SELECT department, AVG(salary) AS avg\_salary, COUNT(\*) AS emp\_count

FROM employee

WHERE department IN ('IT', 'Finance') -- filter rows BEFORE grouping

GROUP BY department -- group by department

HAVING AVG(salary) > 6000 -- filter groups AFTER aggrega

ORDER BY avg\_salary DESC; -- sort result

SQL Joins

I confused with joins

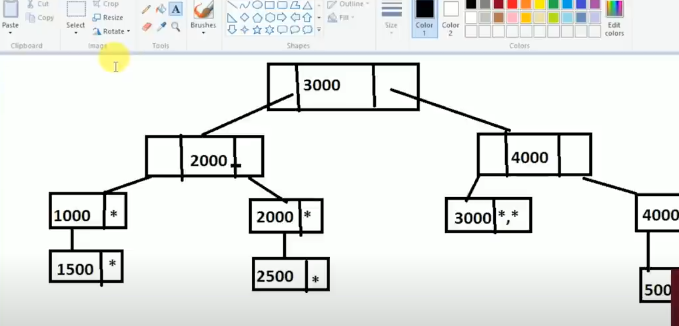
Indexes:

BTree Default

BITMAP

Create index i1 on emp(salary);

Btree



**so if we are using indexes it will get the data by using rowids?**

###### ChatGPT said:

💯 Exactly! You got it perfectly — that’s the core idea of how **Oracle indexes actually speed up data access**.

Let’s go step-by-step so you can picture what’s happening inside Oracle’s engine 👇

## 🧩 Step-by-Step: How Oracle Uses Index + ROWID

Let’s say you have this setup:

CREATE INDEX idx\_emp\_sal\_name ON employees(salary, name);

and your table has data like:

| **ROWID** | **EMP\_ID** | **NAME** | **SALARY** | **DEPT** |
| --- | --- | --- | --- | --- |
| AAB124 | 101 | ARUN | 25000 | 10 |
| AAB126 | 102 | KAVYA | 25000 | 20 |
| AAB123 | 103 | RAMU | 30000 | 30 |

and your **index** internally looks like:

| **Key (salary, name)** | **ROWID** |
| --- | --- |
| (25000, ARUN) | AAB124 |
| (25000, KAVYA) | AAB126 |
| (30000, RAMU) | AAB123 |

### 1️⃣ You run this query:

SELECT \* FROM employees WHERE salary = 25000;

### 2️⃣ Oracle checks if there’s an index on salary

Yes ✅ — so instead of scanning the whole table, it looks up the index.

### 3️⃣ Oracle traverses the ****B-Tree index****

* It searches the index keys (salary) in sorted order.
* Finds matching entries for salary = 25000.

**From the index**, it retrieves:

ROWIDs = AAB124, AAB126

### 4️⃣ Oracle uses these ****ROWIDs**** to fetch rows from the table

Each ROWID tells Oracle **exactly which data block, row, and file** the record lives in.

So Oracle performs a **"table access by ROWID"**:

Go directly to Datafile X, Block Y, Row Z

➡️ That’s a **direct pointer**, no scanning required!

### 5️⃣ Oracle returns full rows

Now it reads those table rows and returns:

| **EMP\_ID** | **NAME** | **SALARY** | **DEPT** |
| --- | --- | --- | --- |
| 101 | ARUN | 25000 | 10 |
| 102 | KAVYA | 25000 | 20 |

## ⚙️ Technically Speaking

Every Oracle **heap table** row has a unique **ROWID** (physical address).

A ROWID encodes:

[Datafile Number] + [Block Number] + [Row Slot Number]

That’s why using the ROWID is the fastest possible way to access a row — it’s like having its exact coordinates.

## 🔍 Query Plan Example

You can confirm this by checking the execution plan:

EXPLAIN PLAN FOR

SELECT \* FROM employees WHERE salary = 25000;

SELECT \* FROM TABLE(DBMS\_XPLAN.DISPLAY);

You’ll see something like:

---------------------------------------------------------------------------------

| Id | Operation | Name | Rows | Bytes | Cost |

---------------------------------------------------------------------------------

| 0 | SELECT STATEMENT | | | | |

| 1 | TABLE ACCESS BY INDEX ROWID | EMPLOYEES | 2 | 80 | |

| 2 | INDEX RANGE SCAN | IDX\_EMP\_SAL\_NAME | 2 | | |

---------------------------------------------------------------------------------

➡️ This means:

1. Oracle first did an **INDEX RANGE SCAN** on IDX\_EMP\_SAL\_NAME
2. Then performed a **TABLE ACCESS BY INDEX ROWID** to fetch full rows