Oracle - SQL - Aggregates

By Dhandapani Yedappalli Krishnamurthi Sep 25, 2025

1. Aggregate Functions in Oracle SQL

Aggregate functions perform calculations on a **set of rows** and return a **single value**.

Common Aggregate Functions:

1. **COUNT()** → returns number of rows.
2. **SUM()** → returns sum of values.
3. **AVG()** → returns average value.
4. **MIN()** → returns minimum value.
5. **MAX()** → returns maximum value.

👉 Example Table: **employees**

emp_id	emp_name	dept	salary
1	Arjun	IT	50000
2	Meena	HR	30000
3	Karthik	IT	60000
4	Divya	HR	35000
5	Sameer	Finance	40000

Example Queries:

```
-- Count employees
SELECT COUNT(*) AS total_employees FROM employees;

-- Total salary
SELECT SUM(salary) AS total_salary FROM employees;

-- Average salary
SELECT AVG(salary) AS avg_salary FROM employees;

-- Min and Max salary
```

SELECT MIN(salary) AS lowest, MAX(salary) AS highest FROM employees;

✅ Output Example:

- total_employees → 5
- total_salary → 215000
- avg_salary → 43000
- lowest → 30000, highest → 60000

2. GROUP BY

GROUP BY groups rows that have the same values into summary rows.

Example:

```
-- Total salary by department
SELECT dept, SUM(salary) AS dept_total
FROM employees
GROUP BY dept;
```

✅ Output:

dept	dept_total
IT	110000
HR	65000
Finance	40000

3. HAVING Clause

HAVING filters groups (like WHERE but for aggregate results).

Example:

```
-- Departments with total salary more than 50,000
SELECT dept, SUM(salary) AS dept_total
FROM employees
GROUP BY dept
HAVING SUM(salary) > 50000;
```

✅ Output:

dept	dept_total
IT	110000
HR	65000

(Finance excluded since 40,000 < 50,000)

4. ROLLUP

ROLLUP generates **subtotals + grand total** in a hierarchy.

Example:

-- Salary by department with subtotal and grand total
SELECT dept, SUM(salary) AS total_salary
FROM employees
GROUP BY ROLLUP(dept);

✅ Output:

dept	total_salary
Finance	40000
HR	65000
IT	110000
(null)	215000

5. CUBE

CUBE generates **all combinations** of subtotals + grand total.

Example with dept and emp_name:
SELECT dept, emp_name, SUM(salary) AS total_salary
FROM employees
GROUP BY CUBE(dept, emp_name);

✅ Output (partial):

dept	emp_name	total_salary
IT	Arjun	50000
IT	Karthik	60000
IT	**(null)**	**110000**
HR	Meena	30000
HR	Divya	35000
HR	**(null)**	**65000**
Finance	Sameer	40000
Finance	**(null)**	**40000**
(null)	**(null)**	**215000**

6. ROLLUP vs CUBE (Quick Comparison)

Feature	ROLLUP	CUBE
Purpose	Hierarchical subtotals + total	All possible subtotals + total
Use Case	Department → Company total	Dept totals + Employee totals + All

So in summary:

- **Aggregate Functions** → calculate values (SUM, AVG, COUNT…).
- **GROUP BY** → group rows by column.
- **HAVING** → filter groups after aggregation.
- **ROLLUP** → subtotals + grand total.
- **CUBE** → all possible subtotal combinations.

1. GROUP BY (Basic Grouping)

👉 Groups rows strictly by the chosen column(s).

Example:

```
SELECT dept, SUM(salary)
FROM employees
GROUP BY dept;
```

Visual:

```
employees
   |
   +-- IT → SUM = 110000
   +-- HR → SUM = 65000
   +-- Finance → SUM = 40000
```

✅ Only **exact group results**, no totals.

---

2. ROLLUP (Hierarchical Totals)

👉 Adds **subtotals + grand total** in a top-down hierarchy.

Example:

```
SELECT dept, SUM(salary)
FROM employees
GROUP BY ROLLUP(dept);
```

Visual:

```
employees
```

```
   |
   +-- IT → 110000
   +-- HR → 65000
   +-- Finance → 40000
   |
   +-- Grand Total → 215000
```

✅ Notice the **grand total row** (`null`) appears.

---

3. CUBE (All Possible Subtotals)

👉 Generates **every subtotal combination** + grand total.

Example:

```
SELECT dept, emp_name, SUM(salary)
FROM employees
GROUP BY CUBE(dept, emp_name);
```

Visual:

```
employees
   |
   +-- IT
   |     +-- Arjun → 50000
   |     +-- Karthik → 60000
   |     +-- Subtotal IT → 110000
   |
   +-- HR
   |     +-- Meena → 30000
   |     +-- Divya → 35000
   |     +-- Subtotal HR → 65000
```

```
|
+-- Finance
|     +-- Sameer → 40000
|     +-- Subtotal Finance → 40000
|
+-- Subtotal by emp_name (ignoring dept)
|     +-- Arjun → 50000
|     +-- Karthik → 60000
|     +-- Meena → 30000
|     +-- Divya → 35000
|     +-- Sameer → 40000
|
+-- Grand Total → 215000
```

✅ CUBE includes **both department-wise subtotals, employee-wise subtotals, and overall total.**

---

4. Quick Comparison Diagram

```
GROUP BY dept → Only dept totals
ROLLUP(dept) → Dept totals + Grand total
CUBE(dept, emp_name) → Dept totals + Emp totals + Grand total
```

Oracle SQL Aggregation Cheat Sheet

Aggregate Functions

Function	Description	Example	Output
COUNT(*)	Number of rows	SELECT COUNT(*) FROM employees;	5
SUM(col)	Total sum	SELECT SUM(salary) FROM employees;	215000
AVG(col)	Average value	SELECT AVG(salary) FROM employees;	43000
MIN(col)	Minimum value	SELECT MIN(salary) FROM employees;	30000
MAX(col)	Maximum value	SELECT MAX(salary) FROM employees;	60000

GROUP BY

Groups rows and returns one row per group.

```
SELECT dept, SUM(salary) AS dept_total
FROM employees
GROUP BY dept;
```

Result:

dept	dept_total
IT	110000
HR	65000
Finance	40000

Visual:

employees
- ├── IT → 110000
- ├── HR → 65000
- └── Finance → 40000

HAVING

Filters groups after aggregation.

```
SELECT dept, SUM(salary) AS dept_total
FROM employees
GROUP BY dept
HAVING SUM(salary) > 50000;
```

Result:

dept	dept_total
IT	110000
HR	65000

ROLLUP

Creates subtotals + grand total in hierarchy.

```
SELECT dept, SUM(salary) AS total_salary
FROM employees
GROUP BY ROLLUP(dept);
```

Result:

dept	total_salary
Finance	40000
HR	65000
IT	110000
(null)	215000

employees
- ├── IT → 110000
- ├── HR → 65000
- ├── Finance → 40000
- └── Grand Total → 215000

CUBE

Generates **all possible subtotals** + grand total.

```
SELECT dept, emp_name, SUM(salary) AS total_salary
FROM employees
GROUP BY CUBE(dept, emp_name);
```

Partial Result:

dept	emp_name	total_salary
IT	Arjun	50000
IT	Karthik	60000
IT	**(null)**	**110000**
HR	Meena	30000
HR	Divya	35000
HR	**(null)**	**65000**
Finance	Sameer	40000
Finance	**(null)**	**40000**
(null)	**(null)**	**215000**

employees
- **├── IT**
- **│ ├── Arjun → 50000**
- **│ ├── Karthik → 60000**
- **│ └── Subtotal IT → 110000**
- **├── HR**
- **│ ├── Meena → 30000**
- **│ ├── Divya → 35000**
- **│ └── Subtotal HR → 65000**
- **├── Finance**
- **│ ├── Sameer → 40000**
- **│ └── Subtotal Finance → 40000**
- **├── Subtotals by emp_name (ignoring dept)**

```
│   ├── Arjun → 50000
│   ├── Karthik → 60000
│   ├── Meena → 30000
│   ├── Divya → 35000
│   └── Sameer → 40000
└── Grand Total → 215000
```

Quick Comparison

Clause	Purpose
GROUP BY	Groups rows into summary rows
HAVING	Filters groups after aggregation
ROLLUP	Adds subtotals + grand total
CUBE	Adds all possible subtotals + grand total

---

1. What is a Transaction?

A **transaction** is a logical unit of work that consists of one or more SQL statements executed as a single unit.
✅ Properties of transactions are called **ACID**:

- **Atomicity** → all or nothing
- **Consistency** → maintains database integrity
- **Isolation** → transactions run independently
- **Durability** → once committed, changes are permanent

2. COMMIT

COMMIT saves all changes made in the current transaction permanently.

Example:

-- Start transaction

```
INSERT INTO employees VALUES (6, 'Ravi', 'IT', 55000);
-- Make changes permanent
COMMIT;
```

✅ After COMMIT:

- Changes are visible to other users.
- Cannot be undone with ROLLBACK.

3. ROLLBACK

ROLLBACK undoes changes made in the current transaction **before commit**.

Example:

-- Start transaction

```
UPDATE employees SET salary = 70000 WHERE emp_name = 'Arjun';
-- Decide to cancel changes
ROLLBACK;
```

✅ After ROLLBACK:

- Salary of Arjun remains unchanged (previous value).
- Database returns to last committed state.

4. SAVEPOINT

SAVEPOINT allows marking a point in a transaction to which you can later roll back, **without affecting earlier work**.

Example:

```
-- Start transaction
INSERT INTO employees VALUES (7, 'Priya', 'HR', 45000);
SAVEPOINT sp1;   -- mark savepoint
INSERT INTO employees VALUES (8, 'Suresh', 'Finance', 38000);
SAVEPOINT sp2;   -- another savepoint
-- Now rollback to sp1
ROLLBACK TO sp1;
-- Commit remaining work
COMMIT;
```

✅ What happens here?

- Row for Priya (id 7) is inserted.
- Row for Suresh (id 8) is undone (rolled back to sp1).
- After COMMIT, only Priya's row is saved.

5. Putting It All Together (Transaction Flow)

```
-- Begin Transaction (implicit)
UPDATE employees SET salary = salary + 5000 WHERE dept = 'IT';
SAVEPOINT sp1;
DELETE FROM employees WHERE dept = 'Finance';
ROLLBACK TO sp1; -- Finance deletion undone, IT salary increment
remains
COMMIT; -- Save IT salary increment permanently
```

6. Quick Summary

Command	Purpose
COMMIT	Makes changes permanent
ROLLBACK	Cancels changes since last COMMIT or SAVEPOINT
SAVEPOINT	Marks a point in a transaction to rollback to

✅ So:

- Use **COMMIT** when you are sure changes are correct.
- Use **ROLLBACK** when you want to cancel all uncommitted changes.
- Use **SAVEPOINT** for partial rollbacks inside long transactions.

ACID Properties in Oracle SQL

1. Atomicity (All or Nothing)

- A transaction is **indivisible** — either all its operations succeed, or none of them are applied.

Example:

```
BEGIN;
UPDATE accounts SET balance = balance - 500 WHERE acc_id =
101;  -- Debit
UPDATE accounts SET balance = balance + 500 WHERE acc_id =
202;  -- Credit
COMMIT;
```

- ✅ If both succeed → money transfers.
  ❌ If debit succeeds but credit fails → ROLLBACK ensures no money disappears.

---

2. Consistency (Valid State → Valid State)

- Ensures that data moves from one **valid state** to another, maintaining rules and constraints.
- **Example:**
  - Suppose account balances must always be **non-negative**.
  - If an update tries to reduce balance below zero, the transaction fails.
  - **Constraints, triggers,** and **referential integrity** guarantee **consistency**.

---

3. Isolation (Concurrent Transactions Do Not Interfere)

- Each transaction executes **as if it were alone** in the system.
- Isolation levels in Oracle: `READ COMMITTED`, `SERIALIZABLE`, etc.
- **Example:**
  - Transaction A: Transfers ₹500 from Account 101 → 202
  - Transaction B: Reads Account 101 balance at the same time
  - With proper isolation, B won't see the **intermediate state** (where money is deducted but not yet added).

---

4. Durability (Permanent Results)

- Once a transaction is **COMMITTED**, its changes survive system crashes, power failures, etc.
- Oracle achieves this via **redo logs & backup mechanisms**.
- **Example:**
  - After a `COMMIT`, even if the server restarts, the money transfer remains reflected in both accounts.

Real-World Analogy: ATM Withdrawal

- **Atomicity:** Cash dispensed only if the account is debited.
- **Consistency:** Account balance rules followed.
- **Isolation:** Your withdrawal unaffected by someone else's transfer at the same time.
- **Durability:** Once debited and recorded, the bank crash won't erase the transaction.

Joins in Oracle SQL

Joins combine rows from two or more tables based on a related column.

Example Tables

Employees

EmpID	EmpName	DeptID
1	Rajesh Sharma	10
2	Priya Singh	20
3	Amitabh Joshi	10
4	Sneha Patel	30

Departments

DeptID	DeptName
10	Sales
20	IT
30	HR
40	Finance

INNER JOIN

Returns rows where there is a match in both tables.

```
SELECT e.EmpName, d.DeptName
FROM Employees e
INNER JOIN Departments d ON e.DeptID = d.DeptID;
```

Output:

EmpName	DeptName
Rajesh Sharma	Sales
Priya Singh	IT
Amitabh Joshi	Sales
Sneha Patel	HR

LEFT JOIN

Returns all rows from the left table, and matched rows from the right. The result is NULL from the right side if there is no match.

```
SELECT e.EmpName, d.DeptName
FROM Employees e
LEFT JOIN Departments d ON e.DeptID = d.DeptID;
```

If there were employees with no DeptID match, they'd still appear.

Subquery

A query inside another query.

Example: Find employees whose department is 'Sales'

```
SELECT EmpName FROM Employees
WHERE DeptID = (SELECT DeptID FROM Departments WHERE DeptName =
'Sales');
```

Output:

EmpName
Rajesh Sharma
Amitabh Joshi

Correlated Subquery

A subquery that references columns from the outer query.

Example: Find employees whose salary is greater than the average salary in their department (assuming a Salaries table)

Salaries

EmpID	Salary
1	70000
2	90000
3	72000
4	60000

Query:

```
SELECT e.EmpName, s.Salary
FROM Employees e
JOIN Salaries s ON e.EmpID = s.EmpID
WHERE s.Salary > (
  SELECT AVG(Salary)
  FROM Salaries
  WHERE EmpID IN (
    SELECT EmpID FROM Employees WHERE DeptID = e.DeptID
  )
);
```

This selects employees earning more than their department's average salary.

UNION

Combines result sets of two queries, removing duplicates.

Example: List all employee and department names as a single column

```
SELECT EmpName AS Name FROM Employees
UNION
SELECT DeptName AS Name FROM Departments;
```

Output:

Name
Rajesh Sharma
Priya Singh
Amitabh Joshi
Sneha Patel
Sales
IT
HR
Finance

INTERSECT

Returns rows common to both queries.

Example: Find names that are both employee names and department names (unlikely in this data)

```
SELECT EmpName FROM Employees
INTERSECT
SELECT DeptName FROM Departments;
```

Output: (empty for given data)

MINUS

Returns rows in the first query but not in the second.

Example: Find employees whose names are not department names

```
SELECT EmpName FROM Employees
MINUS
SELECT DeptName FROM Departments;
```

Output: All employees since none match department names here.

Share & follow

For any inquiries, please contact:

Dhandapani Yedappalli Krishnamurthi

dhandapani_yk@hotmail.com | www.linkedin.com/in/dhandapaniyk