SQL

DATABASE – BASICS

Definition:
A **Database** is an organized collection of data stored electronically so that it can be **easily accessed, managed, and updated.**

Why we use a Database

- To store large amounts of data safely
- To retrieve data quickly
- To avoid data duplication
- To allow multiple users to access data simultaneously

Key Points

- Data is stored in **tables** (rows & columns)
- Databases are managed using **DBMS** (Database Management System)
- Provides **security, consistency, and backup**
- Supports **CRUD operations** (Create, Read, Update, Delete)

Real-Life Example (Easy)
Think of a **college register**:

- Each student = one row
- Student details (roll no, name, marks) = columns
- Whole register = database

Technical Example
A **Student Table** in a database:

id	name	course	marks
1	Ali	Java	85
2	Ravi	Python	78

Here:

- Table = Student
- Row = One student record
- Column = Student attributes

Types of Databases

- **Relational Database (RDBMS)** – MySQL, Oracle, PostgreSQL
- **NoSQL Database** – MongoDB, Cassandra
- **In-Memory Database** – Redis

For **Cognizant & Java roles**, focus mainly on **RDBMS.**

DBMS vs RDBMS

DBMS	RDBMS
Stores data as files	Stores data as tables
Less security	High security
No relationships	Supports relationships
Example: File system	Example: MySQL, Oracle

Common Database Terms

- **Table** – Structure to store data

- **Row (Record)** – One complete data entry
- **Column (Field)** – Data attribute
- **Primary Key** – Uniquely identifies a record
- **Foreign Key** – Links two tables

Where Databases Are Used (Real Projects)

- Banking systems
- E-commerce websites
- College management systems
- Spring Boot applications (with JPA/Hibernate)

Interview Questions (Very Important)

Q1. What is a database?

A. An organized collection of data that allows easy storage, retrieval, and management.

Q2. Why do we use DBMS instead of files?

A. For security, faster access, data consistency, and multi-user support.

Q3. What is CRUD?

A. Create, Read, Update, Delete operations on data.

Common Mistakes Freshers Make

- Thinking database = table
- Not understanding primary key importance
- Confusing DBMS and RDBMS

INTRODUCTION TO RDBMS (Relational Database Management System)

Definition:

RDBMS is a type of database management system that stores data in the form of **tables (relations)** and maintains **relationships between tables** using keys.

Why RDBMS is Used

- To organize data in a **structured format**
- To maintain **relationships** between data
- To ensure **data accuracy and consistency**
- To support **complex queries** using SQL

Key Points

- Data is stored in **tables (rows & columns)**
- Each table represents one **entity**
- Relationships are created using **Primary Key & Foreign Key**
- Follows **ACID properties**
- Uses **SQL** for data operations

Real-Life Example (Easy)

Think of a **college system**:

- One table for **Students**
- One table for **Courses**
- Student table stores course_id to link with Course table

This relationship avoids repeating course details for every student.

Technical Example

Student Table

student_id (PK)	name	course_id (FK)
1	Aamir	101
2	Ravi	102

Course Table

course_id (PK)	course_name
101	Java
102	Python

course_id connects both tables.

Features of RDBMS

- **Data Integrity** – Correct and reliable data
- **Normalization** – Reduces redundancy
- **Security** – User roles and permissions
- **Concurrency** – Multiple users can access data
- **Backup & Recovery**

ACID Properties (Very Important)

- **Atomicity** – All or nothing
- **Consistency** – Data remains valid
- **Isolation** – Transactions don't interfere
- **Durability** – Data is not lost after commit

Popular RDBMS Examples

- **Oracle**
- **MySQL**
- **PostgreSQL**
- **SQL Server**

Cognizant commonly expects **Oracle/MySQL basics.**

DBMS vs RDBMS

DBMS	RDBMS
No relations	Relations between tables
Less secure	More secure
File-based	Table-based
No normalization	Normalization supported

Where RDBMS is Used

- Banking applications
- Enterprise Java applications
- Spring Boot + JPA projects
- ERP systems

Interview Questions

Q1. What is RDBMS?

A. A database system that stores data in tables and maintains relationships using keys.

Q2. Why RDBMS is better than DBMS?

A. Because it supports relationships, normalization, data integrity, and security.

Q3. What is a relation?

A. A table in an RDBMS.

Common Fresher Mistakes

- Confusing table with database
- Not understanding foreign key usage
- Ignoring ACID properties

SQL vs NoSQL (Structured vs Non-Structured Databases)

Definition:

SQL Databases

SQL databases are **relational databases** that store data in **tables (rows & columns)** and use **SQL (Structured Query Language)** to manage data.

NoSQL Databases

NoSQL databases are **non-relational databases** that store data in **flexible formats** like documents, key-value pairs, columns, or graphs.

Why Both Exist

- **SQL** → When data is structured and relationships are important
- **NoSQL** → When data is large, unstructured, and needs high scalability

---

Key Points

SQL

- Fixed schema
- Table-based structure
- Supports joins
- Strong consistency
- ACID properties

NoSQL

- Flexible or schema-less
- Not table-based
- No joins (mostly)
- High scalability
- BASE properties

Real-Life Example (Easy)

SQL Example

A **bank system**:

- Customers
- Accounts
- Transactions

  All are related → **SQL is best**

NoSQL Example

A **social media app**:

- User posts
- Comments
- Likes

  Different structures → **NoSQL is better**

Technical Example

SQL (Table Format)

id	name	email
1	Ali	ali@gmail.com

NoSQL (Document Format – JSON)

```
{
  "id": 1,
  "name": "Ali",
  "email": "ali@gmail.com",
  "skills": ["Java", "Spring"]
}
```

Types

SQL Databases

- MySQL
- Oracle
- PostgreSQL
- SQL Server

NoSQL Databases

- **Document** – MongoDB
- **Key-Value** – Redis
- **Column** – Cassandra
- **Graph** – Neo4j

ACID vs BASE

ACID (SQL)

- Atomicity
- Consistency
- Isolation
- Durability

BASE (NoSQL)

- Basically Available
- Soft state
- Eventually consistent

Where Used in Real Projects

Use Case	Best Choice
Banking	SQL
E-commerce orders	SQL
Chat applications	NoSQL
Big data analytics	NoSQL
Spring Boot enterprise apps	SQL

Cognizant Java roles → SQL is more important.

Interview Questions

Q1. Difference between SQL and NoSQL?

A. SQL is relational and table-based; NoSQL is non-relational and flexible.

Q2. Which one is faster?

A. NoSQL is faster for large-scale, distributed data.

Q3. Can we use both together?

A. Yes, many applications use SQL + NoSQL together.

Common Fresher Mistakes

- Thinking NoSQL replaces SQL
- Assuming NoSQL has no structure at all
- Ignoring consistency needs

Exam & Interview Tip
First learn SQL completely, then understand NoSQL basics.

DATA TYPES – ORACLE DATABASE (Oracle 26c / 26ai oriented)

Definition:
In **Oracle RDBMS**, a **data type** specifies the **kind of data** a column can store and how Oracle allocates **storage and validation** for that data.

Why Data Types Are Important in Oracle

- Ensure **data accuracy**
- Improve **query performance**
- Save **storage space**
- Prevent **invalid data insertion**

Oracle is **strict** about data types compared to some databases.

Key Points (Oracle-Specific)

- Oracle uses **VARCHAR2** (not VARCHAR)
- Oracle has **NUMBER** instead of INT/DECIMAL
- DATE stores **date + time**
- Correct data type choice affects **indexes & performance**

Real-Life Example

- Employee ID → number
- Employee Name → text
- Salary → exact number
- Joining Date → date & time

Oracle provides special data types to handle all these efficiently.

Main Oracle Data Types (Very Important)

---

1. CHARACTER DATA TYPES

Data Type	Description
CHAR(n)	Fixed-length character data
VARCHAR2(n)	Variable-length character data (recommended)
NCHAR	Fixed-length Unicode
NVARCHAR2	Variable-length Unicode

Example

```
name VARCHAR2(50)
```

Interview Point:
Always use **VARCHAR2,** not VARCHAR.

---

2. NUMERIC DATA TYPES

Data Type	Description
NUMBER	Stores integers & decimals
NUMBER(p,s)	Precision & scale

Example

```
salary NUMBER(10,2)
age NUMBER
```

Oracle does NOT have INT, FLOAT, DOUBLE as main types.

3. DATE & TIME DATA TYPES

Data Type	Description
DATE	Stores date + time (to seconds)
TIMESTAMP	Date + time with fractional seconds
TIMESTAMP WITH TIME ZONE	Includes time zone
INTERVAL	Stores time intervals

Example

```
joining_date DATE
created_at TIMESTAMP
```

Very Important Interview Point

Oracle DATE stores **both date and time.**

---

4. LARGE OBJECT DATA TYPES (LOB)

Data Type	Used For
CLOB	Large text
BLOB	Images, videos
NCLOB	Unicode text

Example

```
profile_image BLOB
description CLOB
```

---

5. BOOLEAN (PL/SQL ONLY)

- BOOLEAN exists **only in PL/SQL**
- Not allowed in SQL table columns

```
In tables, use:
status CHAR(1)  -- 'Y' / 'N'
```

Technical Example (Oracle Table)

```
CREATE TABLE employee (
  emp_id NUMBER PRIMARY KEY,
  emp_name VARCHAR2(50),
  salary NUMBER(10,2),
  joining_date DATE,
  resume CLOB
);
```

Where Used in Real Oracle Projects

- VARCHAR2 → username, email
- NUMBER → id, salary, price
- DATE → order_date, joining_date
- TIMESTAMP → audit logs
- BLOB → documents, images

Interview Questions (Oracle-Focused)

Q1. Difference between VARCHAR and VARCHAR2?

A. VARCHAR2 is recommended; VARCHAR is reserved for future use.

Q2. Does Oracle DATE store time?

A. Yes, it stores date + time.

Q3. Which data type is used instead of INT in Oracle?

A. NUMBER.

Q4. Can we use BOOLEAN in Oracle tables?

A. No, only in PL/SQL.

Common Fresher Mistakes

- Using VARCHAR instead of VARCHAR2
- Thinking DATE stores only date
- Using NUMBER without precision everywhere
- Trying to use BOOLEAN in table columns

Best Practice (Oracle Standard)

Use **VARCHAR2 + NUMBER(p,s) + DATE/TIMESTAMP** properly for performance & clarity.

STRUCTURING DATA (Oracle RDBMS – Oracle 26ai Oriented)

Definition:

Structuring data means organizing data in a database **properly using tables, columns, keys, and relationships** so that data is **efficient, consistent, and easy to manage**.

Why Structuring Data Is Important

- Avoids **data duplication**
- Improves **data consistency**
- Makes queries **faster**
- Makes applications **easy to maintain**
- Supports **real-world business logic**

Key Points

- Data is stored in **tables**
- Each table represents **one entity**
- Tables are connected using **keys**
- Follow **normalization rules**
- Design database before writing SQL

Real-Life Example (Easy)

Instead of writing student details repeatedly for each subject:

Bad Structure

```
StudentName | Subject | Marks
Ali         | Java    | 85
Ali         | Python  | 78
```

Good Structure

- Student table
- Subject table
- Marks table

This avoids repetition.

Core Elements of Data Structuring (Oracle)

---

1. TABLES

- Store data in rows & columns
- Each table = one real-world object

```
CREATE TABLE student (
  student_id NUMBER,
  name VARCHAR2(50)
);
```

---

2. COLUMNS

- Each column stores one type of data
- Use proper **Oracle data types**

```
name VARCHAR2(50)
dob DATE
```

---

3. PRIMARY KEY

- Uniquely identifies each row
- Cannot be NULL or duplicate

```
student_id NUMBER PRIMARY KEY
```

---

4. FOREIGN KEY

- Creates relationship between tables
- Maintains **referential integrity**

```
course_id NUMBER REFERENCES course(course_id)
```

---

5. RELATIONSHIPS

Type	Example
One-to-One	Person → Passport
One-to-Many	Department → Employees
Many-to-Many	Students ⇔ Courses

Normalization (Very Important)

Normalization means **breaking data into multiple tables** to reduce redundancy.

Normal Forms (Interview Level)

- **1NF** – No repeating groups
- **2NF** – No partial dependency
- **3NF** – No transitive dependency

Most projects follow **3NF.**

Technical Example (Well-Structured Data)

STUDENT

```
student_id NUMBER PRIMARY KEY,
name VARCHAR2(50)
```

COURSE

```
course_id NUMBER PRIMARY KEY,
course_name VARCHAR2(50)
```

ENROLLMENT

```
student_id NUMBER REFERENCES student(student_id),
course_id NUMBER REFERENCES course(course_id)
```

Where Structuring Data Is Used

- Banking systems
- ERP systems
- Spring Boot + Oracle applications
- HR & Payroll systems

Proper structure = **fewer bugs + faster queries.**

Interview Questions

Q1. What is structuring data?

A. Organizing data using tables, keys, and relationships to avoid redundancy.

Q2. Why normalization is needed?

A. To reduce duplication and maintain consistency.

Q3. What is referential integrity?
A. Ensuring foreign key values exist in parent table.
Common Fresher Mistakes

- Putting everything in one table
- Not using primary keys
- Ignoring normalization
- Wrong data type selection

Best Practice (Oracle Projects)

- One table = one entity
- Use **primary & foreign keys**
- Normalize till **3NF**
- Use correct Oracle data types

CREATE, ALTER, DROP, TRUNCATE (DDL Commands – Oracle 26ai)

Definition:
These are **DDL (Data Definition Language)** commands used to **define and modify database structure** (tables, columns, etc.).
DDL commands **change the structure**, not the data logic.

Why DDL Commands Are Used

- To create tables
- To modify table structure
- To remove tables
- To clear table data completely

Key Points (Important)

- DDL commands are **auto-commit** in Oracle
- Cannot be rolled back
- Affect **table structure or entire data**
- Used mainly by **DBAs & backend developers**

CREATE
Purpose
Used to **create database objects** (table, view, index).
Syntax

```
CREATE TABLE employee (
  emp_id NUMBER PRIMARY KEY,
  name VARCHAR2(50),
  salary NUMBER(10,2)
);
```

Interview Point
Table structure is created, no data yet.

ALTER
Purpose
Used to **modify an existing table structure.**
Common Operations

- Add column
- Modify column

- Drop column

Syntax Examples

```
ALTER TABLE employee ADD age NUMBER;
ALTER TABLE employee MODIFY name VARCHAR2(100);
ALTER TABLE employee DROP COLUMN age;
```

DROP

Purpose

Used to **completely remove a table** (structure + data).

Syntax

```
DROP TABLE employee;
```

- Cannot be recovered easily.

TRUNCATE

Purpose

Used to **delete all records from a table**, but keeps structure.

Syntax

```
TRUNCATE TABLE employee;
```

DROP vs TRUNCATE vs DELETE (Very Important)

Feature	DELETE	TRUNCATE	DROP
Type	DML	DDL	DDL
Removes Data	Yes (row-wise)	Yes (all)	Yes
Removes Structure	No	No	Yes
Rollback	Possible	No	No
WHERE clause	Yes	No	No
Faster	Slow	Fast	Fastest

Where Used in Real Projects

- Initial database setup
- Schema changes
- Clearing test data
- Removing unused tables

Interview Questions

Q1. Difference between DROP and TRUNCATE?

A. DROP removes table completely; TRUNCATE removes only data.

Q2. Can TRUNCATE be rolled back?

A. No, it is auto-commit.

Q3. Which is faster: DELETE or TRUNCATE?

A. TRUNCATE.

Common Fresher Mistakes

- Using DROP instead of TRUNCATE
- Expecting rollback after TRUNCATE
- Forgetting WHERE clause in DELETE
- Running DDL in production without caution

Best Practice (Oracle)

- Use **DELETE** when rollback is needed
- Use **TRUNCATE** for large data cleanup
- Use **DROP** only when object is not needed

CONSTRAINTS (Oracle RDBMS – Oracle 26ai)

Definition:
A **constraint** is a **rule applied on table columns** to ensure **data accuracy, integrity, and reliability** in the database.

Why Constraints Are Important

- Prevent **invalid data**
- Maintain **data consistency**
- Enforce **business rules**
- Protect database from wrong inserts/updates

Key Points

- Constraints can be applied at **column level or table level**
- Oracle enforces constraints **automatically**
- If constraint is violated → error is thrown
- Constraints improve **data quality**

Types of Constraints in Oracle (Very Important)

---

1. NOT NULL
Ensures column cannot have NULL values
name VARCHAR2(50) NOT NULL

---

2. UNIQUE
Ensures all values in a column are unique
email VARCHAR2(100) UNIQUE

---

3. PRIMARY KEY
Uniquely identifies each row
Combination of **NOT NULL + UNIQUE**
emp_id NUMBER PRIMARY KEY
Only **one primary key per table.**

---

4. FOREIGN KEY
Creates relationship between tables
Maintains **referential integrity**
dept_id NUMBER REFERENCES department(dept_id)
Child table value must exist in parent table.

---

5. CHECK
Validates data based on condition
age NUMBER CHECK (age >= 18)

---

6. DEFAULT
Assigns default value if none provided
status CHAR(1) DEFAULT 'Y'

Technical Example (All Constraints)

```
CREATE TABLE employee (
  emp_id NUMBER PRIMARY KEY,
  name VARCHAR2(50) NOT NULL,
  email VARCHAR2(100) UNIQUE,
  age NUMBER CHECK (age >= 18),
  status CHAR(1) DEFAULT 'Y',
```

```
  dept_id NUMBER,
  CONSTRAINT fk_dept FOREIGN KEY (dept_id)
  REFERENCES department(dept_id)
);
```

Adding Constraints Using ALTER

```
ALTER TABLE employee
ADD CONSTRAINT emp_email_uk UNIQUE (email);
```

Where Constraints Are Used

- Banking systems (account rules)
- HR systems (employee data)
- E-commerce (orders, users)
- Spring Boot + Oracle apps

Interview Questions (Must-Know)

Q1. What is a constraint?
A. A rule that enforces data integrity in a table.
Q2. Difference between UNIQUE and PRIMARY KEY?
A. Primary key doesn't allow NULL; UNIQUE allows one NULL.
Q3. Can a table have multiple UNIQUE constraints?
A. Yes.
Q4. Can a table have multiple FOREIGN KEYS?
A. Yes.

Common Fresher Mistakes

- Forgetting PRIMARY KEY
- Using UNIQUE instead of PRIMARY KEY
- Not defining FOREIGN KEY relationships
- Ignoring CHECK constraints

Best Practices (Oracle Projects)

- Always define **PRIMARY KEY**
- Use **FOREIGN KEY** for relationships
- Use **CHECK** for business rules
- Give meaningful constraint names

PRIMARY KEY & FOREIGN KEY (Oracle RDBMS – Oracle 26ai)

Definition:
Primary Key
A **Primary Key** is a column (or set of columns) that **uniquely identifies each record** in a table.
Foreign Key
A **Foreign Key** is a column that **creates a relationship** between two tables by **referencing the primary key** of another table.

Why They Are Important

- Ensure **data uniqueness**
- Maintain **relationships**
- Enforce **referential integrity**
- Prevent invalid or orphan records

Key Points

Primary Key

- Must be **unique**
- Cannot be **NULL**
- Only **one primary key per table**
- Can be **composite** (multiple columns)

Foreign Key

- Can have **duplicate values**
- Can be **NULL**
- Multiple foreign keys allowed
- Enforces relationship between tables

Technical Example (Oracle)

Parent Table

```
CREATE TABLE department (
  dept_id NUMBER PRIMARY KEY,
  dept_name VARCHAR2(50)
);
```

Child Table

```
CREATE TABLE employee (
  emp_id NUMBER PRIMARY KEY,
  emp_name VARCHAR2(50),
  dept_id NUMBER,
  CONSTRAINT fk_dept
  FOREIGN KEY (dept_id)
  REFERENCES department(dept_id)
);
```

Relationship Flow

```
DEPARTMENT (Parent)
   |
   |  dept_id (PK)
   |
EMPLOYEE (Child)
   |
   | dept_id (FK)
```

Composite Key

When **more than one column** is used as a key.

PRIMARY KEY (student_id, course_id)

Used in **many-to-many relationships**.

Primary Key vs Foreign Key (Very Important)

Feature	Primary Key	Foreign Key
Uniqueness	Yes	No
NULL allowed	No	Yes
Count per table	One	Many
Purpose	Identify record	Create relationship

Interview Questions

Q1. Can a table have multiple primary keys?

A. No, only one (can be composite).

Q2. Can a foreign key be NULL?

A. Yes.

Q3. Can a foreign key reference a UNIQUE key?
A. Yes (Oracle allows it).
Q4. What happens if parent record is deleted?
A. Child records will fail unless ON DELETE CASCADE is used.

ON DELETE CASCADE
FOREIGN KEY (dept_id)
REFERENCES department(dept_id)
ON DELETE CASCADE
Deletes child rows automatically when parent is deleted.

Common Fresher Mistakes

- Forgetting primary key
- Using foreign key without parent table
- Thinking foreign key must be unique
- Not understanding cascade delete

Best Practices (Oracle Projects)

- Always define **primary key**
- Use foreign keys to maintain integrity
- Use cascade carefully
- Name constraints clearly

UNIQUE, NOT NULL, CHECK, DEFAULT (Oracle RDBMS – Oracle 26ai)

Definition:

- **UNIQUE** → Ensures all values in a column are **different**
- **NOT NULL** → Ensures a column **cannot have NULL values**
- **CHECK** → Validates data based on a **condition**
- **DEFAULT** → Assigns a **default value** when no value is provided

---

Why These Constraints Are Used

- To **prevent invalid data**
- To **enforce business rules**
- To maintain **data consistency**
- To reduce application-level validation

Key Points (Oracle-Specific)

Constraint	Allows NULL	Allows Duplicates
UNIQUE	One NULL	No
NOT NULL	No	Yes
CHECK	Depends	Depends
DEFAULT	Yes	Yes

Oracle allows **only one NULL** in a UNIQUE column.

Real-Life Example (Easy)

- Email → UNIQUE
- Name → NOT NULL
- Age → CHECK (>=18)
- Status → DEFAULT 'ACTIVE'

Technical Examples (Oracle)
NOT NULL
name VARCHAR2(50) NOT NULL

UNIQUE
email VARCHAR2(100) UNIQUE

CHECK
age NUMBER CHECK (age >= 18)

DEFAULT
status VARCHAR2(10) DEFAULT 'ACTIVE'

All Constraints in One Table

```
CREATE TABLE user_account (
  user_id NUMBER PRIMARY KEY,
  username VARCHAR2(50) NOT NULL,
  email VARCHAR2(100) UNIQUE,
  age NUMBER CHECK (age >= 18),
  status VARCHAR2(10) DEFAULT 'ACTIVE'
);
```

Adding Constraints Using ALTER

```
ALTER TABLE user_account
ADD CONSTRAINT user_email_uk UNIQUE (email);
```

Interview Questions (Very Important)
Q1. Difference between UNIQUE and PRIMARY KEY?
A. UNIQUE allows one NULL; PRIMARY KEY does not allow NULL.
Q2. Can CHECK constraint contain multiple conditions?
A. Yes.

```
CHECK (age >= 18 AND age <= 60)
```

Q3. When is DEFAULT value used?
A. When no value is provided during INSERT.

Common Fresher Mistakes

- Thinking UNIQUE = PRIMARY KEY
- Forgetting NOT NULL with business-critical columns
- Writing wrong CHECK conditions
- Expecting DEFAULT to override given value

Best Practices (Oracle Projects)

- Use NOT NULL for mandatory fields
- Use UNIQUE for business identifiers (email, phone)
- Use CHECK for validation
- Use DEFAULT for status fields

DML – Data Manipulation Language

Definition:
DML is used to **insert, update, delete, and read data** stored inside database tables.
DML works on **table data**, not table structure.

Why We Use DML

- To **add new records**
- To **modify existing data**
- To **remove unwanted data**
- To **retrieve data for applications**

DML Commands (Core)

Command	Purpose
INSERT	Add new data
UPDATE	Modify data
DELETE	Remove data
SELECT	Retrieve data

In Oracle interviews, **SELECT is also treated as DML.**

Technical Examples (Oracle 26ai)

INSERT

```
INSERT INTO student (id, name, age)
VALUES (1, 'Ali', 20);
```

INSERT without column names

```
INSERT INTO student
VALUES (2, 'Rahim', 22);
```

Order must match table columns.

UPDATE

```
UPDATE student
SET age = 23
WHERE id = 2;
```

Without WHERE → all rows updated.

DELETE

```
DELETE FROM student
WHERE id = 1;
```

Without WHERE → all rows deleted.

SELECT

```
SELECT name, age
FROM student
WHERE age > 21;
```

Transaction Control (Very Important)

DML changes are **not permanent** until committed.

Command	Purpose
COMMIT	Save changes
ROLLBACK	Undo changes
SAVEPOINT	Partial rollback

```
COMMIT;
ROLLBACK;
```

Where DML Is Used in Real Projects

- User registration → INSERT
- Profile update → UPDATE
- Account deletion → DELETE

- Dashboard data → SELECT

Interview Questions & Answers

Q1. Difference between DELETE and TRUNCATE?

A. DELETE is DML (can rollback), TRUNCATE is DDL (cannot rollback).

Q2. Is SELECT a DML command?

A. Yes, in Oracle it is considered DML.

Q3. Can we rollback INSERT?

A. Yes, before COMMIT.

Common Fresher Mistakes

- Forgetting WHERE clause
- Not using COMMIT
- Confusing DELETE with TRUNCATE
- Assuming DML auto-saves data

Best Practices (Oracle Projects)

- Always use WHERE with UPDATE/DELETE
- Use SELECT first before UPDATE
- Commit after successful transaction
- Use rollback during testing

INSERT, UPDATE, DELETE (DML Commands)

Definition:

- **INSERT** → Adds new rows into a table
- **UPDATE** → Modifies existing rows in a table
- **DELETE** → Removes rows from a table

These commands **change table data.**

Why We Use Them

- To store new information
- To correct or modify data
- To remove invalid or old data
- Used daily in **real-time applications**

Key Points (Very Important)

Command	Affects Data	Needs WHERE	Rollback Possible
INSERT	Yes	No	Yes
UPDATE	Yes	Recommended	Yes
DELETE	Yes	Recommended	Yes

Technical Examples (Oracle 26ai)

INSERT (Single Row)

```
INSERT INTO employee (emp_id, name, salary)
VALUES (101, 'Ahmed', 30000);
```

INSERT Multiple Rows

```
INSERT ALL
  INTO employee VALUES (102, 'Rahim', 35000)
```

```
  INTO employee VALUES (103, 'Salman', 40000)
SELECT * FROM dual;
```

INSERT from Another Table

```
INSERT INTO emp_backup
SELECT * FROM employee;
```

UPDATE with WHERE

```
UPDATE employee
SET salary = 45000
WHERE emp_id = 103;
```

UPDATE Multiple Columns

```
UPDATE employee
SET salary = 38000, name = 'Rafi'
WHERE emp_id = 102;
```

DELETE with WHERE

```
DELETE FROM employee
WHERE emp_id = 101;
```

DELETE All Rows

```
DELETE FROM employee;
```

Can be rolled back before COMMIT.

Transaction Control

```
COMMIT;
ROLLBACK;
SAVEPOINT sp1;
```

Interview Questions (Must-Prepare)

Q1. What happens if WHERE is missing in UPDATE?

A. All rows get updated.

Q2. Can we rollback DELETE?

A. Yes, before COMMIT.

Q3. Difference between DELETE and TRUNCATE?

A. DELETE is DML (rollback possible); TRUNCATE is DDL.

Common Fresher Mistakes

- Running UPDATE without WHERE
- Forgetting COMMIT
- Using DELETE instead of TRUNCATE
- Assuming INSERT auto-saves data

Best Practices

- Always test with SELECT first
- Use WHERE carefully
- Commit only after verification
- Use transactions in production

Querying Data (SELECT Statement)

Definition:
Querying data means **retrieving required information** from one or more tables using the **SELECT** statement.
We query data to **read**, not change it.

Why We Query Data

- To display data to users
- To generate reports
- To filter required records
- To analyze business information

Basic SELECT Syntax

```
SELECT column1, column2
FROM table_name;
```

Common Querying Clauses (Very Important)

Clause	Purpose
SELECT	Choose columns
FROM	Choose table
WHERE	Filter rows
ORDER BY	Sort results
DISTINCT	Remove duplicates
GROUP BY	Group rows
HAVING	Filter groups

Technical Examples (Oracle 26ai)
Select All Data

```
SELECT * FROM student;
```

Select Specific Columns

```
SELECT name, age FROM student;
```

WHERE Clause (Filtering)

```
SELECT * FROM student
WHERE age > 20;
```

DISTINCT (Remove Duplicates)

```
SELECT DISTINCT city FROM student;
```

ORDER BY (Sorting)

```
SELECT name, age
FROM student
ORDER BY age DESC;
```

AND / OR Conditions

```
SELECT * FROM student
WHERE age > 18 AND city = 'Vijayawada';
```

BETWEEN

```
SELECT * FROM student
WHERE age BETWEEN 18 AND 25;
```

IN

```
SELECT * FROM student
WHERE city IN ('Hyderabad', 'Vijayawada');
```

LIKE (Pattern Matching)

```
SELECT * FROM student
WHERE name LIKE 'A%';
```

Aggregate Functions (Used in Queries)

Function	Purpose
COUNT	Number of rows
SUM	Total
AVG	Average
MAX	Highest
MIN	Lowest

```
SELECT COUNT(*) FROM student;
```

Where Querying Is Used in Real Projects

- Login validation
- Dashboard data display
- Report generation
- Search functionality

Interview Questions & Answers

Q1. Difference between WHERE and HAVING?

A. WHERE filters rows, HAVING filters groups.

Q2. What is DISTINCT used for?

A. To remove duplicate values.

Q3. Which executes first: WHERE or SELECT?

A. WHERE executes first.

Common Fresher Mistakes

- Using = instead of LIKE
- Forgetting quotes for strings
- Using WHERE with aggregate functions
- Confusing ORDER BY with GROUP BY

Best Practices

- Select only required columns
- Use WHERE to reduce data load
- Avoid SELECT * in production
- Use proper indexes

SELECT Statement (Oracle 26ai)

Definition:

The **SELECT** statement is used to **retrieve (read) data** from one or more tables in a database.

- It does **not modify data,** only displays it.

Why We Use SELECT

- To view table data
- To filter required records
- To generate reports
- To supply data to applications (UI, APIs)

Basic Syntax

```
SELECT column1, column2
FROM table_name;
```

SELECT Statement Clauses (Order Matters)

```
SELECT
FROM
WHERE
GROUP BY
HAVING
ORDER BY
```

Execution order (interview favorite):

```
FROM → WHERE → GROUP BY → HAVING → SELECT → ORDER BY
```

Common SELECT Examples (Oracle)

Select All Columns

```
SELECT * FROM employee;
```

Select Specific Columns

```
SELECT emp_id, name FROM employee;
```

WHERE Clause (Filter Rows)

```
SELECT * FROM employee
WHERE salary > 30000;
```

DISTINCT (Remove Duplicates)

```
SELECT DISTINCT department FROM employee;
```

ORDER BY (Sorting)

```
SELECT name, salary
FROM employee
ORDER BY salary DESC;
```

Multiple Conditions

```
SELECT * FROM employee
WHERE salary > 25000 AND department = 'IT';
```

LIKE (Pattern Search)

```
SELECT * FROM employee
WHERE name LIKE 'A%';
```

Aggregate Functions with SELECT

```
SELECT COUNT(*) FROM employee;
SELECT AVG(salary) FROM employee;
```

Real Project Usage

- Login data validation
- Dashboard listing
- Search and filter screens

- Reports and analytics

Interview Questions (Must Know)

Q1. Is SELECT a DML command?

A. Yes (in Oracle).

Q2. Difference between WHERE and HAVING?

A. WHERE filters rows, HAVING filters groups.

Q3. Why avoid SELECT * in projects?

A. Performance and security reasons.

Common Fresher Mistakes

- Forgetting WHERE condition
- Using wrong quotes (' ' vs " ")
- Using WHERE with aggregate functions
- Assuming SELECT changes data

Best Practices

- Select only required columns
- Always use WHERE
- Use aliases for readability
- Index columns used in WHERE

WHERE Clause (Oracle 26ai)

Definition:

The **WHERE clause** is used to **filter records** from a table based on a **condition**.

- Only rows that satisfy the condition are returned.

Why WHERE Clause Is Important

- Retrieve only **needed data**
- Improves **query performance**
- Prevents **processing unnecessary rows**
- Enforces **business logic** in queries

Key Points

- Works with **SELECT, UPDATE, DELETE**
- Supports **logical, comparison, and pattern operators**
- Multiple conditions can be combined with **AND / OR**
- Can use **BETWEEN, IN, LIKE, IS NULL**

Real-Life Example

Student database:

- Get students **above 20** → WHERE age > 20
- Get students from **Vijayawada** → WHERE city = 'Vijayawada'
- Get students with names **starting with A** → WHERE name LIKE 'A%'

Comparison Operators

Operator	Meaning
=	Equals
!= or <>	Not equal
>	Greater than

Operator	Meaning
<	Less than
>=	Greater or equal
<=	Less or equal

Example

```
SELECT * FROM student
WHERE age >= 18;
```

Logical Operators

Operator	Meaning
AND	Both conditions true
OR	Either condition true
NOT	Negates condition

Example

```
SELECT * FROM student
WHERE city = 'Vijayawada' AND age > 20;
```

BETWEEN Operator

- Selects values **within a range**

```
SELECT * FROM student
WHERE age BETWEEN 18 AND 25;
```

IN Operator

- Selects rows with values **in a given list**

```
SELECT * FROM student
WHERE city IN ('Vijayawada', 'Hyderabad');
```

LIKE Operator (Pattern Matching)

Pattern	Meaning
'A%'	Starts with A
'%A'	Ends with A
'%A%'	Contains A
'_a%'	Second character is a

Example

```
SELECT * FROM student
WHERE name LIKE 'A%';
```

IS NULL / IS NOT NULL

```
SELECT * FROM student
WHERE phone IS NULL;

SELECT * FROM student
WHERE phone IS NOT NULL;
```

Where Used in Real Projects

- Filtering users in login forms
- Search functionality
- Reports with conditions
- Updating specific records

Interview Questions

Q1. Difference between WHERE and HAVING?

A. WHERE filters **rows**, HAVING filters **groups**.

Q2. Can WHERE be used with aggregate functions?

A. No, use HAVING for aggregates.

Q3. What happens if WHERE is omitted?

A. All rows are affected (SELECT/UPDATE/DELETE).

Common Fresher Mistakes

- Forgetting quotes for strings
- Using = instead of LIKE for patterns
- Using WHERE with GROUP BY aggregates
- Forgetting AND / OR precedence

Best Practices

- Always test SELECT before UPDATE/DELETE
- Use parentheses for multiple conditions
- Index columns used in WHERE for faster queries
- Use BETWEEN/IN for cleaner code

Sorting Data – ORDER BY (Oracle 26ai)

Definition:

The **ORDER BY** clause is used to **sort the result set** of a query in **ascending (ASC) or descending (DESC) order** based on one or more columns.

Why ORDER BY Is Important

- To present data in a **meaningful order**
- To make reports **readable**
- To identify **top or bottom performers**
- Used in dashboards, analytics, and applications

Key Points

- Default sorting is **ascending (ASC)**
- Multiple columns can be used
- Sorting happens **after filtering (WHERE)**
- Can be combined with **GROUP BY**

Basic Syntax

```
SELECT column1, column2
FROM table_name
ORDER BY column1 ASC|DESC, column2 ASC|DESC;
```

Technical Examples (Oracle 26ai)

Sort Single Column (Ascending)

```
SELECT name, salary
FROM employee
ORDER BY salary ASC;
```

Sort Single Column (Descending)

```
SELECT name, salary
FROM employee
```

```
ORDER BY salary DESC;
```

Sort Multiple Columns

```
SELECT name, department, salary
FROM employee
ORDER BY department ASC, salary DESC;
```

Combine ORDER BY with WHERE

```
SELECT name, salary
FROM employee
WHERE department = 'IT'
ORDER BY salary DESC;
```

Interview Questions

Q1. What is default sort order of ORDER BY?

A. ASC (ascending).

Q2. Can ORDER BY use column position?

```
SELECT name, salary FROM employee ORDER BY 2 DESC;
```

A. Yes, 2 means second column.

Q3. Can ORDER BY be used with GROUP BY?

A. Yes, after GROUP BY and HAVING.

Common Fresher Mistakes

- Forgetting ASC/DESC → assumes wrong order
- Trying ORDER BY without SELECT columns
- Using ORDER BY in subqueries incorrectly

Best Practices

- Always specify ASC/DESC for clarity
- Use ORDER BY with indexes for performance
- Combine with WHERE to reduce rows before sorting
- Avoid ORDER BY * in large tables

Filtering Data – WHERE Clause (Oracle 26ai)

Definition:

The **WHERE clause** is used to **filter rows** in a table that meet a **specific condition**.

- Only the rows satisfying the condition are retrieved or affected by DML (UPDATE/DELETE).

Why Filtering Is Important

- To retrieve **specific data**
- To update or delete **targeted records**
- To improve **query performance**
- To apply **business rules**

Key Points

- Works with **SELECT, UPDATE, DELETE**
- Can use **comparison, logical, pattern, range, or list conditions**
- Multiple conditions can be combined using **AND, OR, NOT**
- Filtering happens **before sorting or grouping**

Logical Operators

```
SELECT * FROM employee
WHERE department = 'IT' AND salary > 25000;

SELECT * FROM employee
WHERE department = 'IT' OR department = 'HR';

SELECT * FROM employee
WHERE NOT department = 'HR';
```

Pattern Matching with LIKE

```
SELECT * FROM employee
WHERE name LIKE 'A%'; -- Starts with A
SELECT * FROM employee
WHERE name LIKE '%ah'; -- Ends with ah
SELECT * FROM employee
WHERE name LIKE '%am%'; -- Contains am
```

IN Operator (Multiple Values)

```
SELECT * FROM employee
WHERE department IN ('IT','HR','Finance');
```

BETWEEN Operator (Range)

```
SELECT * FROM employee
WHERE salary BETWEEN 20000 AND 50000;
```

IS NULL / IS NOT NULL

```
SELECT * FROM employee
WHERE manager_id IS NULL;
```

Real Project Usage

- Filter users by location
- Retrieve orders within a date range
- Search products based on price or category
- Validate records before updates

Interview Questions

Q1. Difference between WHERE and HAVING?
A. WHERE filters rows before grouping; HAVING filters groups after GROUP BY.
Q2. Can WHERE be used with aggregate functions?
A. No, use HAVING for aggregates.
Q3. What is the use of LIKE?
A. Pattern matching in text columns.

Common Fresher Mistakes

- Forgetting quotes for strings
- Confusing = and LIKE
- Using WHERE with aggregates incorrectly
- Forgetting logical operator precedence

Best Practices

- Use WHERE to reduce dataset before ORDER BY
- Combine conditions carefully (AND vs OR)
- Use indexes on filtered columns for performance
- Avoid functions on columns in WHERE (slows query)

DISTINCT, LIKE, Wildcards (Oracle 26ai)

Definition:

- **DISTINCT** → Removes **duplicate rows** from query results.
- **LIKE** → Used with **pattern matching** in string/text columns.
- **Wildcards** → Special characters used with LIKE to search patterns.

Why They Are Important

- DISTINCT → Ensures **unique data**
- LIKE & Wildcards → Search **partial or flexible data**
- Useful in **reports, search filters, and dashboards**

Key Points

Feature	Details
DISTINCT	Removes duplicates
LIKE	Works with % (any number of chars) & _ (single char)
Case-sensitive	Oracle LIKE is case-sensitive by default
Can combine	LIKE + WHERE + ORDER BY + DISTINCT

Syntax

DISTINCT

```
SELECT DISTINCT column1
FROM table_name;
```

LIKE with Wildcards

```
SELECT column1
FROM table_name
WHERE column1 LIKE 'pattern';
```

Wildcards in Oracle

Wildcard	Meaning	Example
%	Any number of characters	LIKE 'A%' → Starts with A
_	Single character	LIKE '_li' → 3-letter word ending with "li"
[]	Not used in Oracle	Use REGEXP_LIKE instead

Technical Examples (Oracle 26ai)

DISTINCT

```
SELECT DISTINCT city
FROM employee;
```

LIKE – Starts with

```
SELECT name
FROM employee
WHERE name LIKE 'A%';
```

LIKE – Ends with

```
SELECT name
FROM employee
WHERE name LIKE '%ah';
```

LIKE – Contains

```
SELECT name
FROM employee
WHERE name LIKE '%am%';
```

LIKE – Single character

```
SELECT name
FROM employee
```

```
WHERE name LIKE '_li';
```

Real Project Usage

- Customer search → LIKE '%text%'
- Reporting → DISTINCT cities, products, categories
- Auto-suggest in search boxes

Interview Questions

Q1. Difference between DISTINCT and GROUP BY?
A. DISTINCT removes duplicates; GROUP BY is used with aggregates.
Q2. What are the wildcards in Oracle?
A. % = any number of characters, _ = single character.
Q3. Is LIKE case-sensitive in Oracle?
A. Yes, by default.

Common Fresher Mistakes

- Using DISTINCT unnecessarily (performance hit)
- Forgetting % or _ in LIKE
- Confusing LIKE with equality =
- Expecting case-insensitive match by default

Best Practices

- Use DISTINCT only when needed
- Use indexes on columns frequently filtered with LIKE
- Combine LIKE with WHERE for efficient search
- Consider UPPER(column) for case-insensitive search

IN & BETWEEN Operators (Oracle 26ai)

Definition:

- **IN** → Filters rows where a column's value **matches one of the specified values**.
- **BETWEEN** → Filters rows where a column's value **falls within a range** (inclusive).

Why They Are Important

- **IN** → Simplifies multiple OR conditions
- **BETWEEN** → Simplifies range queries
- Both make queries **readable and efficient**

Key Points

Feature	IN	BETWEEN
Number of values	Multiple discrete values	Continuous range
Inclusive?	Yes	Yes
Can be used with numbers, strings, dates	Yes	Yes
Alternative to	OR conditions	>= AND <=

Real-Life Examples

- IN → Employees in departments IT, HR, Finance
- BETWEEN → Students aged 18 to 25
- BETWEEN → Orders between two dates

Syntax

IN

```
SELECT column1
FROM table_name
WHERE column1 IN (value1, value2, ...);
```

BETWEEN

```
SELECT column1
FROM table_name
WHERE column1 BETWEEN value1 AND value2;
```

Technical Examples (Oracle 26ai)

Using IN

```
SELECT * FROM employee
WHERE department IN ('IT', 'HR', 'Finance');
```

Equivalent to:

```
WHERE department = 'IT' OR department = 'HR' OR department = 'Finance';
```

Using BETWEEN (Numbers)

```
SELECT * FROM student
WHERE age BETWEEN 18 AND 25;
```

Using BETWEEN (Dates)

```
SELECT * FROM orders
WHERE order_date BETWEEN TO_DATE('2026-01-01','YYYY-MM-DD')
                     AND TO_DATE('2026-01-31','YYYY-MM-DD');
```

Real Project Usage

- Employee reporting → department IN (…)
- Student database → age BETWEEN …
- Sales report → date BETWEEN start & end
- Filtering products → price BETWEEN low & high

Interview Questions

Q1. Difference between IN and OR?

A. IN is shorthand for multiple OR conditions.

Q2. BETWEEN is inclusive or exclusive?

A. Inclusive (includes both boundary values).

Q3. Can BETWEEN work with strings?

A. Yes, alphabetically.

Common Fresher Mistakes

- Using BETWEEN with wrong boundaries
- Using IN with a subquery incorrectly
- Forgetting quotes for string values
- Assuming BETWEEN excludes boundaries

Best Practices

- Use IN for multiple discrete values
- Use BETWEEN for ranges (numbers, dates)
- Always test date ranges carefully
- Combine with ORDER BY for meaningful output

IS NULL & IS NOT NULL (Oracle 26ai)

Definition:

- **IS NULL** → Checks if a column contains **NULL values.**
- **IS NOT NULL** → Checks if a column **does not contain NULL values.**

NULL represents **missing or unknown data** in Oracle.

Why They Are Important

- To **identify missing data**
- To **filter valid records**
- To **ensure data quality**
- Used in **reports, validation, and data cleaning**

Key Points

Feature	IS NULL	IS NOT NULL
Purpose	Find missing values	Find non-missing values
Can be used with	SELECT, UPDATE, DELETE	SELECT, UPDATE, DELETE
Returns	TRUE if NULL	TRUE if not NULL
Common with	Optional columns	Mandatory columns

Syntax

```
SELECT column1, column2
FROM table_name
WHERE column1 IS NULL;

SELECT column1, column2
FROM table_name
WHERE column1 IS NOT NULL;
```

Technical Examples (Oracle 26ai)

Find rows with NULL

```
SELECT * FROM employee
WHERE manager_id IS NULL;
```

Find rows with non-NULL

```
SELECT * FROM employee
WHERE manager_id IS NOT NULL;
```

Update NULL values

```
UPDATE employee
SET manager_id = 100
WHERE manager_id IS NULL;
```

Delete rows with NULL

```
DELETE FROM employee
WHERE manager_id IS NULL;
```

Interview Questions

Q1. Difference between = NULL and IS NULL?

A. = NULL does NOT work; must use IS NULL or IS NOT NULL.

Q2. Can IS NULL be used with UPDATE?

A. Yes, to update missing values.

Q3. Can IS NOT NULL filter mandatory columns?

A. Yes, ensures column has values.

Common Fresher Mistakes

- Using = NULL instead of IS NULL

- Forgetting NOT in IS NOT NULL
- Confusing with empty string ''
- Not handling NULLs in joins

Best Practices

- Always use IS NULL / IS NOT NULL for NULL checks
- Handle NULLs explicitly in queries
- Combine with WHERE for data validation
- Avoid NULLs in mandatory business-critical columns

AND, OR, NOT Operators (Oracle 26ai)

Definition:

Logical operators used in SQL **WHERE clauses** to **combine or negate conditions**:

- **AND** → Both conditions must be true
- **OR** → At least one condition must be true
- **NOT** → Negates a condition (opposite of true)

Why They Are Important

- Filter data with **complex conditions**
- Control **query precision**
- Apply **business rules** in SELECT, UPDATE, DELETE
- Useful in **real-time filtering & reporting**

Key Points

Operator	Result
AND	TRUE if all conditions TRUE
OR	TRUE if any condition TRUE
NOT	TRUE if condition FALSE

- **Precedence:** NOT > AND > OR
- Use **parentheses ()** to control evaluation

Real-Life Examples

- AND → Employees in IT department **and** salary > 30000
- OR → Students in class A **or** class B
- NOT → Products **not** in Electronics category

Syntax

```
SELECT column1, column2
FROM table_name
WHERE condition1 AND condition2;

SELECT column1, column2
FROM table_name
WHERE condition1 OR condition2;

SELECT column1, column2
FROM table_name
WHERE NOT condition;
```

Technical Examples (Oracle 26ai)

Using AND

```
SELECT * FROM employee
WHERE department = 'IT' AND salary > 30000;
```

Using OR

```
SELECT * FROM employee
WHERE department = 'IT' OR department = 'HR';
```

Using NOT

```
SELECT * FROM employee
WHERE NOT department = 'Finance';
```

Using Parentheses (Multiple Operators)

```
SELECT * FROM employee
WHERE (department = 'IT' OR department = 'HR') AND salary > 25000;
```

Real Project Usage

- Filter users based on **role and status**
- Retrieve orders from **specific regions or dates**
- Exclude test data using **NOT**
- Complex reports with **AND/OR conditions**

Interview Questions

Q1. What is the precedence of AND, OR, NOT?
A. NOT > AND > OR

Q2. How to combine AND and OR correctly?
A. Use parentheses to avoid ambiguity.

Q3. Can NOT be used with multiple conditions?
A. Yes, e.g., NOT (condition1 OR condition2)

Common Fresher Mistakes

- Ignoring operator precedence → wrong results
- Forgetting parentheses with multiple conditions
- Using AND instead of OR (and vice versa)
- Using NOT incorrectly with NULLs

Best Practices

- Always use parentheses for complex logic
- Test queries step by step
- Combine with WHERE, IN, BETWEEN for clarity
- Use meaningful column names to avoid confusion

Aggregate Functions (Oracle 26ai)

Definition:
Aggregate functions **perform calculations on a set of rows** and return a **single value.**
They are used for **summary reports, analytics, and business insights.**

Why They Are Important

- Summarize large datasets
- Calculate totals, averages, counts
- Identify max/min values
- Generate meaningful reports

Key Points

- Works with **numeric and sometimes date columns**
- Often used with **GROUP BY**
- Ignore **NULL values** (except COUNT(*))
- Examples: COUNT, SUM, AVG, MIN, MAX

Real-Life Examples

- COUNT → Total students in a class
- SUM → Total salary of employees
- AVG → Average marks of students
- MAX → Highest sales in a month
- MIN → Lowest product price

Syntax

```
SELECT AGG_FUNC(column)
FROM table_name
[WHERE condition]
[GROUP BY column];
```

Common Aggregate Functions (Oracle 26ai)

Function	Purpose	Example
COUNT(*)	Count rows	SELECT COUNT(*) FROM employee;
COUNT(column)	Count non-NULL values	SELECT COUNT(salary) FROM employee;
SUM(column)	Sum values	SELECT SUM(salary) FROM employee;
AVG(column)	Average value	SELECT AVG(salary) FROM employee;
MAX(column)	Maximum value	SELECT MAX(salary) FROM employee;
MIN(column)	Minimum value	SELECT MIN(salary) FROM employee;

Technical Examples

Count Employees

```
SELECT COUNT(*) AS total_employees
FROM employee;
```

Total Salary

```
SELECT SUM(salary) AS total_salary
FROM employee;
```

Average Salary by Department

```
SELECT department, AVG(salary) AS avg_salary
FROM employee
GROUP BY department;
```

Maximum Salary

```
SELECT MAX(salary) AS highest_salary
FROM employee;
```

Minimum Salary

```
SELECT MIN(salary) AS lowest_salary
FROM employee;
```

Combine WITH WHERE

```
SELECT COUNT(*)
FROM employee
WHERE department = 'IT';
```

Interview Questions

Q1. Difference between COUNT(*) and COUNT(column)?

A. COUNT(*) counts all rows; COUNT(column) counts only non-NULL values.

Q2. Can aggregate functions be used with GROUP BY?

A. Yes, to summarize per group.

Q3. Can MIN/MAX be used with strings?

A. Yes, alphabetically.

Common Fresher Mistakes

- Forgetting GROUP BY when using multiple columns
- Using aggregate functions without column
- Confusing COUNT(*) with COUNT(column)
- Expecting aggregates to ignore NULLs in COUNT(*)

Best Practices

- Use meaningful aliases (AS total_salary)
- Combine WHERE to filter before aggregation
- Use GROUP BY only when needed
- Avoid using aggregates on huge tables without filtering

String Functions (Oracle 26ai)

Definition:

String functions are used to **manipulate and process text (VARCHAR2/CHAR) data** in Oracle.

They help in **formatting, extracting, or comparing strings.**

Why They Are Important

- Clean and format text data
- Extract meaningful information from strings
- Useful in reports, validations, and data processing

Key Points

- Operate only on **string data types**
- Commonly used in **SELECT, UPDATE, and WHERE clauses**
- Can be **nested** for complex processing

Real-Life Examples

- Extract first 3 letters of a name → SUBSTR(name,1,3)
- Convert text to uppercase → UPPER(name)
- Remove leading/trailing spaces → TRIM(name)
- Count characters in product code → LENGTH(code)

Common String Functions (Oracle 26ai)

Function	Purpose	Example
LENGTH(str)	Number of characters	LENGTH('Ahmed') → 5
SUBSTR(str, start, length)	Extract substring	SUBSTR('Ahmed',1,3) → 'Ahm'
INSTR(str, substring, start, occurrence)	Find position	INSTR('Ahmed','h') → 2
UPPER(str)	Convert to uppercase	UPPER('ahmed') → 'AHMED'

Function	Purpose	Example
LOWER(str)	Convert to lowercase	LOWER('AHMED') → 'ahmed'
TRIM(str)	Remove spaces	TRIM(' Ali ') → 'Ali'
CONCAT(str1,str2)	Concatenate strings	CONCAT('Ali',' Ahmed') → 'Ali Ahmed'
REPLACE(str, old, new)	Replace substring	REPLACE('Ali','A','O') → 'Oli'

Technical Examples

LENGTH

```
SELECT LENGTH(name) AS name_length FROM student;
```

SUBSTR

```
SELECT SUBSTR(name,1,3) AS short_name FROM student;
```

INSTR

```
SELECT INSTR(name,'h') AS position FROM student;
```

UPPER / LOWER

```
SELECT UPPER(name) AS upper_name, LOWER(name) AS lower_name FROM student;
```

TRIM

```
SELECT TRIM(name) FROM student;
```

CONCAT

```
SELECT CONCAT(first_name, last_name) AS full_name FROM student;
```

REPLACE

```
SELECT REPLACE(name,'a','@') FROM student;
```

Real Project Usage

- Format user names in dashboard
- Generate initials or short codes
- Validate product codes or IDs
- Data cleaning before analytics

Interview Questions

Q1. Difference between CONCAT and || operator?

A. CONCAT works with 2 strings only; || can concatenate multiple strings.

Q2. Can SUBSTR accept negative positions?

A. Yes, in Oracle it counts from end (advanced usage).

Q3. How to remove both leading and trailing spaces?

A. Use TRIM() or RTRIM() + LTRIM() combination.

Common Fresher Mistakes

- Forgetting string quotes ' '
- Confusing INSTR position vs occurrence parameters
- Using LENGTH on numbers (causes error)
- Not knowing CONCAT works only for 2 strings

Best Practices

- Always use aliases (AS) for readability
- Combine functions for complex formatting
- Use TRIM before comparison or filtering
- Use UPPER/LOWER for case-insensitive search

Date & Numeric Functions (Oracle 26ai)

Date Functions

Definition:

Date functions are used to **manipulate, format, or calculate dates** in Oracle.
Oracle stores dates in **DATE** or **TIMESTAMP** data types.

Why They Are Important

- Extract parts of dates (day, month, year)
- Calculate durations, differences, or intervals
- Format dates for reports and dashboards

Common Date Functions

Function	Purpose	Example
SYSDATE	Current system date & time	SELECT SYSDATE FROM dual;
CURRENT_DATE	Current date in session timezone	SELECT CURRENT_DATE FROM dual;
ADD_MONTHS(date, n)	Add n months to a date	ADD_MONTHS(SYSDATE, 2)
MONTHS_BETWEEN(date1, date2)	Months between 2 dates	MONTHS_BETWEEN(SYSDATE, DATE '2026-01-01')
NEXT_DAY(date, 'DAY')	Next specified weekday	NEXT_DAY(SYSDATE,'MONDAY')
LAST_DAY(date)	Last day of month	LAST_DAY(SYSDATE)
EXTRACT(part FROM date)	Extract year, month, day	EXTRACT(MONTH FROM SYSDATE)
TO_CHAR(date, 'format')	Convert date to string	TO_CHAR(SYSDATE,'DD-MON-YYYY')
TO_DATE(string, 'format')	Convert string to date	TO_DATE('12-01-2026','DD-MM-YYYY')

Examples

```
-- Current Date
SELECT SYSDATE FROM dual;

-- Add 3 months
SELECT ADD_MONTHS(SYSDATE,3) FROM dual;

-- Months between two dates
SELECT MONTHS_BETWEEN(DATE '2026-12-31', DATE '2026-01-01') FROM dual;

-- Extract year and month
SELECT EXTRACT(YEAR FROM SYSDATE) AS year,
       EXTRACT(MONTH FROM SYSDATE) AS month
FROM dual;

-- Format date
SELECT TO_CHAR(SYSDATE,'DD-MON-YYYY') AS formatted_date FROM dual;
```

Real Project Usage

- Employee joining & retirement calculations
- Invoice due dates, subscription expiry
- Reports: monthly, quarterly, yearly summaries
- Event reminders & dashboards

Numeric Functions
Definition:
Numeric functions perform **mathematical operations** on numeric columns or values.

Why They Are Important

- Round numbers for display
- Find maximum/minimum values
- Generate random numbers
- Useful in calculations, analytics, and reports

Common Numeric Functions

Function	Purpose	Example
ROUND(number, decimals)	Round to specified decimal	ROUND(123.456,2) → 123.46
TRUNC(number, decimals)	Truncate to decimal	TRUNC(123.456,2) → 123.45
CEIL(number)	Round up	CEIL(123.12) → 124
FLOOR(number)	Round down	FLOOR(123.99) → 123
MOD(number, divisor)	Modulus (remainder)	MOD(10,3) → 1
POWER(number, n)	Raise to power	POWER(2,3) → 8
ABS(number)	Absolute value	ABS(-25) → 25
SQRT(number)	Square root	SQRT(16) → 4
ROUND(DBMS_RANDOM.VALUE,2)	Random number generation	DBMS_RANDOM.VALUE → 0.1234

Examples

```
-- Round salary to 2 decimals
SELECT ROUND(salary,2) FROM employee;

-- Truncate marks
SELECT TRUNC(marks,0) FROM student;

-- Ceiling and Floor
SELECT CEIL(123.45), FLOOR(123.45) FROM dual;

-- Modulus
SELECT MOD(10,3) FROM dual;

-- Power
SELECT POWER(5,2) FROM dual;

-- Absolute and Square Root
SELECT ABS(-100), SQRT(49) FROM dual;
```

Real Project Usage

- Salary rounding for payroll
- Discounts & tax calculations
- Analytics: top/bottom values
- Random numbers for test data generation

Interview Questions

Q1. Difference between ROUND and TRUNC?
A. ROUND rounds to nearest value, TRUNC simply cuts off decimal.
Q2. CEIL vs FLOOR?
A. CEIL → round up, FLOOR → round down.

Q3. How to generate a random number in Oracle?
A. Use DBMS_RANDOM.VALUE() function.

Common Fresher Mistakes

- Forgetting decimals in ROUND/TRUNC
- Confusing CEIL and FLOOR
- Using MOD with negative numbers incorrectly
- Forgetting TO_NUMBER when numeric conversion needed

Best Practices

- Use ROUND for financial calculations
- Use TRUNC for reporting purposes
- Combine numeric functions with date or string functions
- Test DBMS_RANDOM for range constraints

Conversion Functions (Oracle 26ai)

Definition:
Conversion functions are used to **convert data from one type to another** in Oracle.
Helps in **type compatibility, formatting, and calculations.**

Why They Are Important

- Convert **string to number/date** and vice versa
- Format data for reports
- Handle type mismatch in queries
- Useful in **calculations, date comparisons, and reporting**

Key Points

- Common types: **TO_CHAR, TO_DATE, TO_NUMBER**
- Works with **strings, numbers, and dates**
- Often used in **SELECT, WHERE, INSERT, UPDATE**
- Essential for **data type consistency**

Real-Life Examples

- Convert **date to readable format** → TO_CHAR(SYSDATE,'DD-MON-YYYY')
- Convert **string to number** → TO_NUMBER('123')
- Convert **number to string for concatenation** → TO_CHAR(2500)

Common Conversion Functions

Function	Purpose	Example
TO_CHAR(expr, format)	Convert number/date → string	TO_CHAR(SYSDATE,'DD-MON-YYYY')
TO_NUMBER(expr)	Convert string → number	TO_NUMBER('12345')
TO_DATE(expr, format)	Convert string → date	TO_DATE('12-01-2026','DD-MM-YYYY')
CAST(expr AS datatype)	Convert any datatype	CAST(123 AS VARCHAR2(10))

Technical Examples
Convert Date to String

```
SELECT TO_CHAR(SYSDATE,'DD-MON-YYYY') AS formatted_date FROM dual;
```

Convert String to Number

```
SELECT TO_NUMBER('5000') + 100 AS total FROM dual;
```

Convert String to Date

```
SELECT TO_DATE('12-01-2026','DD-MM-YYYY') AS new_date FROM dual;
```

Using CAST

```
SELECT CAST(salary AS VARCHAR2(10)) AS salary_str FROM employee;
```

Real Project Usage

- Format **invoice dates**
- Calculate **totals from string inputs**
- Concatenate **numbers with text** for reports
- Convert **imported CSV data** to correct type

Interview Questions

Q1. Difference between TO_DATE and TO_CHAR?
A. TO_DATE → string → date, TO_CHAR → date/number → string

Q2. Can TO_NUMBER handle decimals?
A. Yes, e.g., TO_NUMBER('123.45') → 123.45

Q3. Why use CAST when TO_CHAR/TO_NUMBER exists?
A. CAST is more general-purpose, works with multiple types.

Common Fresher Mistakes

- Using wrong format in TO_DATE → ORA errors
- Forgetting quotes around strings
- Mixing data types in arithmetic without conversion
- Using TO_NUMBER on non-numeric strings

Best Practices

- Always use proper **format masks** in TO_DATE/TO_CHAR
- Validate strings before TO_NUMBER
- Use CAST for portability and type consistency
- Combine with string/date/numeric functions for reports

Grouping Data – GROUP BY & HAVING (Oracle 26ai)

Definition:

- **GROUP BY** → Groups rows that have **the same values** in specified columns into **summary rows.**
- **HAVING** → Filters groups created by GROUP BY based on **aggregate conditions.**

Together, they are used for **aggregated reporting.**

Why They Are Important

- Summarize data efficiently
- Generate **department-wise, category-wise, or region-wise reports**
- Combine with aggregate functions (SUM, COUNT, AVG, MAX, MIN)
- Filter aggregated results

Key Points

Feature	GROUP BY	HAVING
Purpose	Group rows	Filter groups
Works with	Aggregate functions	Aggregate functions
Must follow	SELECT columns	GROUP BY columns

Feature	GROUP BY	HAVING
Cannot filter rows	Use WHERE for row-level	Use HAVING for group-level

---

Syntax

```
-- Basic GROUP BY
SELECT column1, AGG_FUNC(column2)
FROM table_name
GROUP BY column1;

-- GROUP BY with HAVING
SELECT column1, AGG_FUNC(column2)
FROM table_name
GROUP BY column1
HAVING AGG_FUNC(column2) condition;
```

Technical Examples (Oracle 26ai)

Count Employees per Department

```
SELECT department, COUNT(*) AS total_employees
FROM employee
GROUP BY department;
```

Average Salary per Department

```
SELECT department, AVG(salary) AS avg_salary
FROM employee
GROUP BY department;
```

Using HAVING to Filter Groups

```
SELECT department, COUNT(*) AS total_employees
FROM employee
GROUP BY department
HAVING COUNT(*) > 5; -- Only departments with more than 5 employees
```

Max Salary per Department

```
SELECT department, MAX(salary) AS highest_salary
FROM employee
GROUP BY department
HAVING MAX(salary) > 50000;
```

Real Project Usage

- HR → Department-wise headcount
- Sales → Monthly or regional sales summaries
- Education → Class-wise average marks
- Analytics dashboards → Summarized KPIs

Interview Questions

Q1. Difference between WHERE and HAVING?

A. WHERE filters rows **before grouping**; HAVING filters **after grouping**.

Q2. Can you use HAVING without GROUP BY?

A. Yes, in Oracle, but only for aggregate functions on entire table.

Q3. Why use GROUP BY instead of DISTINCT?

A. GROUP BY allows aggregation; DISTINCT only removes duplicates.

Common Fresher Mistakes

- Using WHERE to filter aggregates (wrong)
- Forgetting to include all non-aggregated columns in GROUP BY
- Using HAVING without aggregate function
- Confusing GROUP BY order and SELECT columns

Best Practices

- Always use meaningful aliases for aggregated columns
- Filter rows first with WHERE before grouping
- Use HAVING only for aggregate conditions
- Keep queries readable with proper indentation

Interview Questions

Q1. Difference between WHERE and HAVING?

- WHERE → Row-level filter before grouping
- HAVING → Group-level filter after aggregation

Q2. Can HAVING be used without GROUP BY?

- Yes, only to filter on aggregate for entire table.

Q3. Why use GROUP BY instead of DISTINCT?

- GROUP BY → aggregates data; DISTINCT → removes duplicates only

---

Common Mistakes

- Using WHERE for aggregate conditions
- Not including non-aggregated SELECT columns in GROUP BY
- Misplacing HAVING before GROUP BY

Best Practices

- Filter rows first with WHERE
- Use HAVING only for aggregate conditions
- Always give **aliases** for aggregate columns
- Keep queries readable

GROUP BY with Multiple Columns (Oracle 26ai)

Definition:

- **GROUP BY with multiple columns** → Groups rows based on **combinations of two or more columns.**
- Each unique combination forms a separate group for aggregation.

Think of it as **grouping by department AND job title** to see stats per role in each department.

Why It's Important

- Generate **more granular summaries**
- Analyze data with multiple dimensions
- Useful in **HR, sales, and analytics reporting**

Key Points

- All **non-aggregated columns in SELECT** must appear in GROUP BY
- Aggregates (SUM, COUNT, AVG, MAX, MIN) are applied **per group combination**
- HAVING can filter **grouped results**

Syntax

```
SELECT column1, column2, AGG_FUNC(column3)
FROM table_name
GROUP BY column1, column2;
```

```
-- With HAVING
SELECT column1, column2, AGG_FUNC(column3)
FROM table_name
GROUP BY column1, column2
HAVING AGG_FUNC(column3) condition;
```

Examples

Count Employees by Department & Job

```
SELECT department, job_title, COUNT(*) AS total_emp
FROM employee
GROUP BY department, job_title;
```

Average Salary by Department & Job

```
SELECT department, job_title, AVG(salary) AS avg_salary
FROM employee
GROUP BY department, job_title;
```

Departments & Jobs with > 3 Employees (HAVING)

```
SELECT department, job_title, COUNT(*) AS total_emp
FROM employee
GROUP BY department, job_title
HAVING COUNT(*) > 3;
```

Max Salary per Department & Job

```
SELECT department, job_title, MAX(salary) AS highest_salary
FROM employee
GROUP BY department, job_title
HAVING MAX(salary) > 50000;
```

Real-Life Usage

- HR → Headcount per department and role
- Sales → Revenue per region and product
- Education → Average marks per class and subject
- Analytics → Multi-dimensional KPIs

Interview Questions

Q1. Can you use multiple columns in GROUP BY?

- Yes, separates groups for each unique combination.

Q2. Can HAVING filter multiple aggregates?

- Yes, e.g., HAVING COUNT(*) > 3 AND MAX(salary) > 50000.

Q3. Difference between single-column and multi-column GROUP BY?

- Single-column → group by one attribute
- Multi-column → group by combination of attributes

Common Fresher Mistakes

- Forgetting to include **all non-aggregated columns** in GROUP BY
- Misusing HAVING without aggregate
- Confusing order of columns in GROUP BY

Best Practices

- Always **list columns in SELECT in same order** as GROUP BY
- Use **aliases** for aggregated columns
- Combine with WHERE to filter rows before grouping
- Use HAVING only for aggregate-based conditions

INNER JOIN (Oracle 26ai)

Definition:

- **INNER JOIN** → Returns rows that have **matching values in both tables.**
- Rows without a match in either table are **excluded.**

Think: "Only show records that exist in **both tables.**"

Why It's Important

- Combine data from multiple tables
- Useful when **foreign key relationships exist**
- Essential for **real-world reports, dashboards, and analytics**

Key Points

- Requires **common column(s)** (primary key → foreign key)
- Filters **only matching records**
- Can join **two or more tables**

Syntax

```
SELECT a.column1, a.column2, b.column3
FROM table1 a
INNER JOIN table2 b
ON a.common_column = b.common_column;
```

Examples

Employees and Departments

```
SELECT e.employee_id, e.name, d.department_name
FROM employee e
INNER JOIN department d
ON e.department_id = d.department_id;
```

- Returns **only employees with a valid department**

Orders and Customers

```
SELECT o.order_id, c.customer_name, o.order_date
FROM orders o
INNER JOIN customers c
ON o.customer_id = c.customer_id;
```

- Returns **orders where customer exists**

Real-Life Usage

- HR → Employee details with department names
- Sales → Orders with customer info
- Education → Students with class info
- Finance → Transactions with account details

Interview Questions

Q1. Difference between INNER JOIN and LEFT JOIN?

- INNER JOIN → only matching rows
- LEFT JOIN → all rows from left table, even without match

Q2. Can you join more than 2 tables with INNER JOIN?

- Yes, chain multiple INNER JOINs.

Q3. What happens if ON condition is missing?

- Cartesian product occurs → all possible row combinations

Common Fresher Mistakes

- Forgetting ON condition → huge unintended results
- Using wrong columns for joining

- Confusing INNER JOIN with LEFT/RIGHT JOIN
- Not using table aliases → confusing column names

Best Practices

- Always use **ON condition** with proper keys
- Use **aliases** for readability
- Combine with **WHERE, GROUP BY, HAVING** for filtered aggregation
- Avoid joining unnecessary tables for performance

LEFT JOIN & RIGHT JOIN (Oracle 26ai)

Definition:

- **LEFT JOIN** → Returns **all rows from the left table** and **matching rows from the right table.**
  - If no match, right table columns show **NULL.**
- **RIGHT JOIN** → Returns **all rows from the right table** and **matching rows from the left table.**
  - If no match, left table columns show **NULL.**

Think:

- LEFT JOIN → "Keep everything from **left**, attach match from **right**"
- RIGHT JOIN → "Keep everything from **right**, attach match from **left**"

Why Important

- Handle **optional or missing relationships**
- Useful for **reporting all records with or without matches**
- Essential in **data analysis and dashboards**

Key Points

Feature	**LEFT JOIN**	**RIGHT JOIN**
All rows from	Left table	Right table
Matching rows from	Right table	Left table
Non-matching side	NULL values	NULL values
Use case	Keep all primary table records	Keep all secondary table records

- Can combine with **WHERE, GROUP BY, HAVING, aggregates**
- Can join **multiple tables**

Syntax

```
-- LEFT JOIN
SELECT a.column1, b.column2
FROM table1 a
LEFT JOIN table2 b
ON a.common_column = b.common_column;

-- RIGHT JOIN
SELECT a.column1, b.column2
FROM table1 a
RIGHT JOIN table2 b
ON a.common_column = b.common_column;
```

Examples

LEFT JOIN: Employees & Departments

```
SELECT e.employee_id, e.name, d.department_name
FROM employee e
LEFT JOIN department d
ON e.department_id = d.department_id;
```

- Shows **all employees**, even if they don't belong to a department

RIGHT JOIN: Employees & Departments

```
SELECT e.employee_id, e.name, d.department_name
FROM employee e
RIGHT JOIN department d
ON e.department_id = d.department_id;
```

- Shows **all departments**, even if no employees exist in some departments

---

Real-Life Usage

- LEFT JOIN → List all employees with their department (even if not assigned)
- RIGHT JOIN → List all products with sales info (even if no sales yet)
- Analytics dashboards → Include all categories, regions, or users

---

Interview Questions

Q1. Difference between INNER JOIN and LEFT/RIGHT JOIN?

- INNER → only matching rows
- LEFT → all left table + matches
- RIGHT → all right table + matches

Q2. Can LEFT JOIN = RIGHT JOIN?

- Yes, by swapping table positions.

Q3. When to use LEFT JOIN vs RIGHT JOIN?

- Depends on **which table you want all rows from**

Common Fresher Mistakes

- Forgetting ON condition → Cartesian product
- Confusing LEFT vs RIGHT JOIN
- Not handling NULLs in results
- Joining unnecessary tables → performance issues

Best Practices

- Always use **table aliases** for readability
- Handle NULLs using **NVL()** if needed
- Use LEFT JOIN when you want **all left table records**
- Use RIGHT JOIN when you want **all right table records**

FULL OUTER JOIN (Oracle 26ai)

Definition:

- **FULL OUTER JOIN** → Returns **all rows from both tables**, matching rows where possible.
- If there is no match on either side, the **unmatched side shows NULL.**

Think: "Give me **everything from both tables**, and match where possible."

Why It's Important

- Retrieve **complete data from two tables**, even if some rows don't match
- Useful for **consolidated reports, analytics, and reconciliation**

- Covers scenarios not handled by INNER, LEFT, or RIGHT JOIN individually

---

Key Points

Feature	FULL OUTER JOIN
Rows returned	All rows from both tables
Matching rows	Joined normally
Non-matching rows	NULL in missing columns
Use case	Data reconciliation, complete summary reports

- Combines behavior of **LEFT JOIN + RIGHT JOIN**
- Can be combined with **WHERE, GROUP BY, HAVING, aggregates**

Syntax

```
SELECT a.column1, b.column2
FROM table1 a
FULL OUTER JOIN table2 b
ON a.common_column = b.common_column;
```

Examples

Employees & Departments

```
SELECT e.employee_id, e.name, d.department_name
FROM employee e
FULL OUTER JOIN department d
ON e.department_id = d.department_id;
```

- Shows **all employees and all departments**
- Employees without a department → department_name = NULL
- Departments without employees → employee_id/name = NULL

Orders & Customers

```
SELECT o.order_id, c.customer_name
FROM orders o
FULL OUTER JOIN customers c
ON o.customer_id = c.customer_id;
```

- Includes **all orders** and **all customers**, even unmatched ones

---

Real-Life Usage

- HR → List all employees and all departments, including empty departments
- Sales → Reconcile all orders vs all customers
- Education → List all students and all courses, including unassigned ones
- Analytics → Full dataset comparison for reporting

Interview Questions

Q1. Difference between FULL OUTER JOIN and LEFT/RIGHT JOIN?

- LEFT → all left + matching right
- RIGHT → all right + matching left
- FULL → all rows from both tables

Q2. Can FULL OUTER JOIN produce NULLs?

- Yes, in unmatched columns from either table

Q3. Can FULL OUTER JOIN be simulated with UNION?

- Yes, LEFT JOIN UNION RIGHT JOIN can simulate FULL OUTER JOIN

Common Fresher Mistakes

- Forgetting ON condition → huge Cartesian product
- Confusing FULL OUTER JOIN with INNER JOIN
- Not handling NULLs in aggregates
- Using it unnecessarily → performance impact

Best Practices

- Always **use table aliases**
- Handle NULLs with **NVL()** or **COALESCE()**
- Use FULL OUTER JOIN only when **both tables' unmatched data is needed**
- Combine with WHERE/HAVING for filtered reports

CROSS JOIN & SELF JOIN (Oracle 26ai)

CROSS JOIN
Definition:

- **CROSS JOIN** → Returns **Cartesian product** of two tables.
- Every row in table1 is combined with **every row in table2.**

Think: "All possible combinations of rows between the tables."

Why Important

- Generate **all combinations** for analysis
- Used in **testing, combinatorial scenarios, or dummy data generation**

Key Points

- No ON condition is used
- Can produce **very large result sets**
- Mostly used for **small tables** due to performance

Syntax

```
SELECT a.column1, b.column2
FROM table1 a
CROSS JOIN table2 b;
```

Example

```
SELECT e.name, d.department_name
FROM employee e
CROSS JOIN department d;
```

- If employee has 5 rows and department has 3 rows → result = 5 × 3 = 15 rows

---

Real-Life Usage

- Generate **all product-color combinations**
- Test scenarios with **all user-permission combinations**
- Generate dummy data for reports

Interview Tips

- Cartesian product is **result of CROSS JOIN**
- Avoid using on **large tables**

SELF JOIN
Definition:

- **SELF JOIN** → Join a table **with itself** to compare rows.
- Typically uses **aliases** to differentiate the table instances.

Think: "Compare one row of a table with another row in the same table."

Why Important

- Find **hierarchies** (e.g., managers & employees)
- Compare rows within same table
- Useful in **reporting and recursive relationships**

Key Points

- Table must have **primary key or unique identifier**
- Use **aliases** for clarity
- Can be **INNER, LEFT, RIGHT JOIN** style

Syntax

```
SELECT a.column1, b.column2
FROM table_name a
JOIN table_name b
ON a.common_column = b.common_column;
```

Example: Employee & Manager

```
SELECT e1.employee_id AS emp_id, e1.name AS emp_name,
       e2.name AS manager_name
FROM employee e1
LEFT JOIN employee e2
ON e1.manager_id = e2.employee_id;
```

- e1 → employee
- e2 → manager (same table)

Real-Life Usage

- HR → Employee and manager relationships
- Students → Pair students for mentorship programs
- Products → Compare different versions of same product
- Analytics → Self-referential reporting

Interview Questions

Q1. Why use SELF JOIN?

- To compare or relate rows within same table

Q2. Difference between SELF JOIN and INNER JOIN?

- SELF JOIN is **same table joined with itself**
- INNER JOIN is typically **between two different tables**

Q3. Can SELF JOIN use LEFT JOIN?

- Yes, to include unmatched rows

Common Fresher Mistakes

- Forgetting aliases → column ambiguity
- Using SELF JOIN unnecessarily → performance hit
- Confusing CROSS JOIN with SELF JOIN

Best Practices

- Always use **table aliases**
- Keep **ON conditions clear**
- Avoid SELF JOIN for very large tables unless necessary
- Combine with **WHERE, GROUP BY, aggregates** for better reporting

---

Multiple Joins (Oracle 26ai)

Definition:

- **Multiple joins** → Combine **more than two tables** in a single query using INNER, LEFT, RIGHT, or FULL JOINs.
- Helps retrieve **comprehensive data** from multiple related tables.

Think: "Join table A → table B → table C to get complete info in one query."

Why It's Important

- Real-world databases have **many related tables**
- Generate **complex reports** (e.g., employees, departments, salaries)
- Essential for **analytics, dashboards, and interview scenarios**

Key Points

- **Order of joins matters** when using LEFT/RIGHT JOIN
- Always use **table aliases** for readability
- Combine **ON conditions** carefully to avoid Cartesian products
- Can mix **different types of joins** in one query

Syntax

```
SELECT a.col1, b.col2, c.col3
FROM table1 a
JOIN table2 b
  ON a.common_column = b.common_column
LEFT JOIN table3 c
  ON b.common_column = c.common_column;
```

Examples

Employee, Department & Location

```
SELECT e.employee_id, e.name, d.department_name, l.location_name
FROM employee e
INNER JOIN department d
  ON e.department_id = d.department_id
LEFT JOIN location l
  ON d.location_id = l.location_id;
```

- Combines **3 tables**: employees → departments → locations
- Shows **all employees**; if a department has no location → location_name = NULL

Orders, Customers & Products

```
SELECT o.order_id, c.customer_name, p.product_name, o.quantity
FROM orders o
INNER JOIN customers c
  ON o.customer_id = c.customer_id
INNER JOIN products p
  ON o.product_id = p.product_id;
```

- Returns **all orders** with customer and product info

Real-Life Usage

- HR → Employee, Department, Manager, Location report
- Sales → Orders, Customers, Products, Payment status
- Education → Student, Class, Subject, Teacher report
- Analytics dashboards → Multi-dimensional KPIs

Interview Questions

Q1. Can you join more than 3 tables?

- Yes, multiple joins can combine as many tables as needed

Q2. Does join order matter?

- Yes, especially for LEFT/RIGHT JOINs

Q3. Can we mix INNER, LEFT, RIGHT joins in one query?

- Yes, must carefully manage ON conditions

Q4. How to avoid Cartesian product in multiple joins?

- Always specify correct **ON conditions**

Common Fresher Mistakes

- Forgetting aliases → column ambiguity
- Misordering LEFT/RIGHT JOINs → wrong NULL results
- Missing or incorrect ON conditions → Cartesian product
- Joining unnecessary tables → performance impact

Best Practices

- Always use **aliases** for all tables
- Start with **INNER JOIN** then add LEFT/RIGHT as required
- Use parentheses if needed for clarity
- Combine with **WHERE, GROUP BY, HAVING** for filtered, summarized data
- Test query step by step for correctness

Subqueries (Oracle 26ai)

Definition:

- **Subquery** → A **query inside another query.**
- Can be used in **SELECT, WHERE, FROM, or HAVING** clauses.

Think: “Use one query’s result as input for another query.”

Why It’s Important

- Break complex queries into **manageable parts**
- Retrieve **filtered, aggregated, or calculated data**
- Essential for **real-world reporting, analytics, and interview questions**

Key Points

- Subqueries are enclosed in **parentheses ()**
- Can return **single value (scalar)** or **multiple values**
- Can be **correlated** (depends on outer query) or **non-correlated**

Types of Subqueries

Type	Description	Example Use
Single-row	Returns one value	=, <, > operators
Multiple-row	Returns multiple values	IN, ANY, ALL operators
Correlated	Depends on outer query	Compare row-by-row with outer table
Non-correlated	Independent query	Executes once

Syntax

Single-row subquery

```
SELECT employee_id, name, salary
FROM employee
WHERE salary = (SELECT MAX(salary) FROM employee);
```

- Finds employee(s) with **highest salary**

Multiple-row subquery

```
SELECT employee_id, name
FROM employee
WHERE department_id IN
      (SELECT department_id FROM department WHERE location_id = 1);
```

- Finds employees in **all departments located in location 1**

Correlated subquery

```
SELECT e1.name, e1.salary
FROM employee e1
WHERE e1.salary >
      (SELECT AVG(e2.salary)
       FROM employee e2
       WHERE e1.department_id = e2.department_id);
```

- Finds employees earning **more than the average in their department**

Real-Life Usage

- HR → Employees above department average salary
- Sales → Orders above average amount per region
- Education → Students scoring above class average
- Analytics → Compare metrics within groups

Interview Questions

Q1. Difference between correlated and non-correlated subquery?

- Correlated → depends on outer query, runs **row-by-row**
- Non-correlated → independent, runs **once**

Q2. Can subquery return multiple columns?

- Yes, but usually in **FROM clause or with EXISTS**

Q3. Difference between IN and = in subquery?

- = → single-row subquery
- IN → multiple-row subquery

Common Fresher Mistakes

- Using = with multi-row subquery → ORA-01427 error
- Forgetting parentheses
- Correlated subquery without proper alias → ambiguity
- Performance issues with large tables

Best Practices

- Always test **inner query separately** first
- Use **aliases** for correlated subqueries
- Prefer JOIN over subquery if it improves performance
- Use EXISTS for **existence check**, IN for **value match**

Single-row & Multi-row Subqueries (Oracle 26ai)

Definition:

- **Single-row subquery** → Returns **exactly one row with one column.**
- **Multi-row subquery** → Returns **one or more rows**, usually **one column.**

Think:

- Single-row → “Just one answer”
- Multi-row → “Multiple possible answers”

Why It's Important

- Single-row → Compare value with aggregate or fixed value
- Multi-row → Check for **membership in a set of values**
- Essential for **filtering, reporting, and analytics**

Key Points

Feature	Single-row	Multi-row
Rows returned	1	≥1
Operators	=, <, >, <=, >=, !=	IN, ANY, ALL, EXISTS
Example use	Max salary, Min marks	Departments, Locations, Multiple IDs
Error risk	ORA-01427 if >1 row	Works with multiple values

Syntax & Examples

Single-row Subquery

```
-- Find employee with highest salary
SELECT employee_id, name, salary
FROM employee
WHERE salary = (SELECT MAX(salary) FROM employee);
```

- Returns **only one employee with the highest salary**

```
-- Find employee with minimum salary in department 10
SELECT employee_id, name
FROM employee
WHERE salary = (SELECT MIN(salary)
                FROM employee
                WHERE department_id = 10);
```

Multi-row Subquery

```
-- Find employees in departments 10, 20, 30
SELECT employee_id, name
FROM employee
WHERE department_id IN
      (SELECT department_id
       FROM department
       WHERE location_id = 1);
```

- Returns employees in **all matching departments**

```
-- Employees earning more than any employee in department 20
SELECT employee_id, name, salary
FROM employee
WHERE salary > ANY
      (SELECT salary
       FROM employee
       WHERE department_id = 20);
-- Employees earning more than all employees in department 30
SELECT employee_id, name, salary
FROM employee
WHERE salary > ALL
      (SELECT salary
       FROM employee
       WHERE department_id = 30);
```

Interview Questions

Q1. Difference between single-row and multi-row subquery?

- Single-row → returns 1 value → use =, >, <
- Multi-row → returns multiple values → use IN, ANY, ALL

Q2. What happens if single-row subquery returns multiple rows?

- ORA-01427: Single-row subquery returns more than one row

Q3. Can multi-row subquery be used with = operator?

- No, use IN or ANY/ALL

Common Fresher Mistakes

- Using = with multi-row subquery → error
- Forgetting parentheses
- Not aliasing in correlated subqueries
- Using multi-row where single-row expected

Best Practices

- Always test **inner query separately**
- Use **IN/ANY/ALL operators** correctly
- Prefer JOIN if it improves **performance**
- Always handle NULL values properly

Correlated Subqueries (Oracle 26ai)

Definition:

- A **correlated subquery** is a **subquery that depends on the outer query** for its values.
- It is executed **once for each row** of the outer query.

Think: "The inner query **looks at the current row** of the outer query before returning a value."

Why It's Important

- Allows **row-by-row comparison**
- Useful for **finding relative values** like max/min per group, employee above dept avg, etc.
- Common in **real-world HR, Sales, and Analytics reports**

Key Points

- Uses **aliases** for outer query table
- Runs **multiple times** → can affect performance
- Typically found in **WHERE or SELECT clauses**

Syntax

```
SELECT outer_table.column1, outer_table.column2
FROM outer_table
WHERE outer_table.column =
      (SELECT aggregate_function(inner_table.column)
       FROM inner_table
       WHERE inner_table.column = outer_table.column);
```

Examples

Employees earning more than dept average

```
SELECT e1.employee_id, e1.name, e1.salary, e1.department_id
FROM employee e1
WHERE e1.salary >
      (SELECT AVG(e2.salary)
       FROM employee e2
```

```
       WHERE e1.department_id = e2.department_id);
```

- For **each employee**, inner query calculates **dept average**
- Returns employees earning **above their department average**

Students scoring above class average

```
SELECT s1.student_id, s1.name, s1.marks, s1.class_id
FROM students s1
WHERE s1.marks >
      (SELECT AVG(s2.marks)
       FROM students s2
       WHERE s1.class_id = s2.class_id);
```

Departments with employee count > 5 (using correlated HAVING)

```
SELECT d.department_id, d.department_name
FROM department d
WHERE (SELECT COUNT(*)
       FROM employee e
       WHERE e.department_id = d.department_id) > 5;
```

Real-Life Usage

- HR → Employees above department average salary
- Education → Students above class average marks
- Sales → Orders higher than region average
- Analytics → Row-by-row comparison with group-level metrics

Interview Questions

Q1. Difference between correlated and non-correlated subquery?

- Correlated → depends on outer query, executes **row by row**
- Non-correlated → independent, executes **once**

Q2. Can correlated subqueries use aggregates?

- Yes, often used with AVG, COUNT, MAX, MIN

Q3. Performance impact?

- Can be **slower** on large tables; use JOINs if possible

Common Fresher Mistakes

- Forgetting alias for outer query table → ORA errors
- Using correlated subquery when non-correlated is enough → performance hit
- Misplacing parentheses

Best Practices

- Always use **table aliases**
- Test inner query **independently**
- Prefer JOINs for large datasets
- Combine with **WHERE, HAVING, ORDER BY** for clear reports

UNION, UNION ALL, INTERSECT, MINUS (Oracle 26ai)

Definition:

Operator	Description
UNION	Combines results of **two queries** and **removes duplicates**
UNION ALL	Combines results of **two queries** and **includes duplicates**
INTERSECT	Returns only **common rows** between two queries

Operator	Description
MINUS	Returns rows from **first query not in second query**

Think: **Set operations on query results**

Why It's Important

- Combine or compare data from **similar structured queries**
- Useful in **reporting, analytics, and data validation**
- Helps **filter, merge, or find differences** between datasets

---

Key Points

- Number of columns and **data types must match** in all queries
- Column names → usually **from first query**
- Can combine multiple set operations in a single query
- **ORDER BY** applies at the **end of final result**

Syntax

```
-- UNION
SELECT column1, column2 FROM table1
UNION
SELECT column1, column2 FROM table2;

-- UNION ALL
SELECT column1, column2 FROM table1
UNION ALL
SELECT column1, column2 FROM table2;

-- INTERSECT
SELECT column1, column2 FROM table1
INTERSECT
SELECT column1, column2 FROM table2;

-- MINUS
SELECT column1, column2 FROM table1
MINUS
SELECT column1, column2 FROM table2;
```

Examples

UNION (removes duplicates)

```
SELECT department_id FROM employee
UNION
SELECT department_id FROM department;
```

- Combines **all department IDs** from employee & department tables, removes duplicates

UNION ALL (keeps duplicates)

```
SELECT department_id FROM employee
UNION ALL
SELECT department_id FROM department;
```

- Combines all, **including duplicates**

INTERSECT (common rows)

```
SELECT department_id FROM employee
INTERSECT
SELECT department_id FROM department;
```

- Returns **department IDs present in both tables**

MINUS (rows in first not in second)

```
SELECT department_id FROM employee
MINUS
SELECT department_id FROM department;
```

- Returns **departments assigned to employees but not present in department table**

Real-Life Usage

- HR → List **all departments from multiple sources**
- Sales → Compare **customers in two regions**
- Education → Students enrolled in **both courses** or only in **one course**
- Analytics → **Data reconciliation and difference reports**

Interview Questions

Q1. Difference between UNION and UNION ALL?

- UNION → removes duplicates
- UNION ALL → includes duplicates

Q2. Difference between INTERSECT and MINUS?

- INTERSECT → common rows
- MINUS → rows in first query **not in second query**

Q3. Can column names be different in the two queries?

- Yes, Oracle uses **first query's column names**

Q4. Can ORDER BY be applied in between queries?

- No, ORDER BY **applies only at the end of final combined query**

Common Fresher Mistakes

- Different **number of columns** → ORA-01789 error
- Column **data types mismatch** → ORA-01790 error
- Using ORDER BY in the middle of set operations
- Confusing MINUS with NOT IN

Best Practices

- Always **ensure column count and types match**
- Use **aliases** for readability if needed
- Use UNION ALL when **duplicates are acceptable** → better performance
- Use INTERSECT or MINUS for **data comparison and validation**

Modifying Data in Oracle (DML) (Oracle 26ai)

Definition:

- **Modifying data** → Using SQL commands to **insert, update, or delete rows** in tables.
- Part of **DML (Data Manipulation Language)**.

Think: "Change the table contents without altering structure."

Why It's Important

- Maintain **accurate and updated data** in applications
- Essential for **real-world database operations** like HR, sales, inventory
- Core skill for **fresher interviews**

Key Points

- **INSERT** → Add new rows
- **UPDATE** → Modify existing rows
- **DELETE** → Remove rows

- **WHERE clause** → Always use to prevent accidental full-table changes
- Use **COMMIT** to save changes and **ROLLBACK** to undo

Syntax & Examples

INSERT

```
-- Single row
INSERT INTO employee (employee_id, name, department_id, salary)
VALUES (101, 'Ali', 10, 50000);

-- Multiple rows (using SELECT)
INSERT INTO employee (employee_id, name, department_id, salary)
SELECT employee_id+100, name, department_id, salary
FROM employee
WHERE department_id = 20;
```

- Adds **new employees** to the table

UPDATE

```
-- Update salary of an employee
UPDATE employee
SET salary = 55000
WHERE employee_id = 101;

-- Increase salary by 10% for department 10
UPDATE employee
SET salary = salary * 1.10
WHERE department_id = 10;
```

- Modifies **existing data**
- **Without WHERE → updates all rows**

DELETE

```
-- Delete specific employee
DELETE FROM employee
WHERE employee_id = 101;

-- Delete all employees from department 20
DELETE FROM employee
WHERE department_id = 20;
```

- Removes **rows from table**
- Use **TRUNCATE** for faster deletion of **all rows**

Real-Life Usage

- HR → Add new employees, update salary, remove resigned employees
- Sales → Update order status, delete canceled orders
- Inventory → Insert new products, update stock, remove discontinued items
- Analytics → Maintain clean datasets

Interview Questions

Q1. Difference between DELETE and TRUNCATE?

- DELETE → Row-by-row removal, can use WHERE, can ROLLBACK
- TRUNCATE → Removes all rows, faster, cannot use WHERE, cannot ROLLBACK

Q2. Difference between UPDATE with and without WHERE?

- With WHERE → changes specific rows
- Without WHERE → changes all rows → risky

Q3. Can INSERT use SELECT statement?

- Yes, to insert multiple rows from another table

Common Fresher Mistakes

- Forgetting **WHERE** in UPDATE or DELETE → affects all rows
- Using TRUNCATE on table with **foreign key constraints** → error
- Not committing after DML → changes not saved
- Trying to UPDATE a primary key that violates uniqueness

Best Practices

- Always use **WHERE clause** in UPDATE/DELETE
- Use **COMMIT and ROLLBACK** wisely
- Backup table or test on a copy before massive changes
- Use **INSERT SELECT** for batch operations
- Keep **transaction logs clean** for auditing

Database Views (Oracle 26ai)

Definition:

- **View** → A **virtual table** based on the result of a SQL query.
- Does **not store data physically**; retrieves data from underlying tables when queried.

Think: "A saved SELECT query that behaves like a table."

Why It's Important

- Simplifies **complex queries**
- Provides **security** by restricting column/table access
- Helps in **reusable queries and reports**
- Essential in **real-world applications and interviews**

Key Points

- **Virtual table** → no physical storage of data
- Can include **JOINs, WHERE, GROUP BY, aggregates**
- Can be **read-only** or **updatable** depending on query
- Changes in underlying tables **reflect in the view automatically**

Syntax

Create a Simple View

```
CREATE VIEW view_name AS
SELECT column1, column2
FROM table_name
WHERE condition;
```

Create a View with JOIN

```
CREATE VIEW emp_dept_view AS
SELECT e.employee_id, e.name, d.department_name
FROM employee e
INNER JOIN department d
ON e.department_id = d.department_id;
```

Query a View

```
SELECT * FROM emp_dept_view;
```

Drop a View

```
DROP VIEW emp_dept_view;
```

Types of Views

Type	Description	Example
Simple View	Based on a single table, no aggregates	CREATE VIEW v_emp AS SELECT name, salary FROM employee;
Complex View	Includes JOIN, GROUP BY, aggregates	CREATE VIEW v_sales AS SELECT c.name, SUM(o.amount) FROM customers c JOIN orders o GROUP BY c.name;

Real-Life Usage

- HR → View employees with department names
- Sales → View total sales per region/product
- Education → View students with average marks
- Analytics → Reusable datasets for dashboards

Interview Questions

Q1. Difference between Table and View?

- Table → stores data physically
- View → virtual table, query result, no physical data

Q2. Can we update a view?

- Simple views → usually updatable
- Complex views (with JOINs/aggregates) → read-only

Q3. Can views improve security?

- Yes, restricts columns/rows visible to users

Q4. What happens if underlying table changes?

- View reflects **latest data automatically**

Common Fresher Mistakes

- Trying to insert into complex views → error
- Forgetting aliases in JOINs → ORA errors
- Assuming view stores data → it only fetches from base tables
- Dropping a table without knowing dependent views → affects view

---

Best Practices

- Name views **meaningfully**
- Use **views for reporting, not heavy DML**
- Always **test queries before creating view**
- Use **aliases in complex views** for clarity
- Grant **access to views instead of tables** for security

Creating, Updating, Dropping Views (Oracle 26ai)

Definition:

- **View** → A virtual table based on a query.
- Can be **created, modified, or removed** as needed.

Think: “Views are like **saved SELECT queries** that you can manage easily.”

Why It’s Important

- Simplifies **complex queries**
- Helps in **data security** by restricting access to certain columns
- Supports **reusable datasets** for reporting
- Learning CRUD operations on views is **common in interviews**

Key Points

- **CREATE VIEW** → defines a new view
- **CREATE OR REPLACE VIEW** → updates an existing view
- **DROP VIEW** → deletes a view
- Changes in **underlying tables** are automatically reflected in the view

---

Syntax & Examples

Creating a View

```
CREATE VIEW emp_dept_view AS
SELECT e.employee_id, e.name, d.department_name
FROM employee e
INNER JOIN department d
ON e.department_id = d.department_id;
```

- View shows **employee details with department name**

Updating a View

```
CREATE OR REPLACE VIEW emp_dept_view AS
SELECT e.employee_id, e.name, d.department_name, e.salary
FROM employee e
INNER JOIN department d
ON e.department_id = d.department_id;
```

- Added **salary column**
- No need to drop first → automatically replaces

Dropping a View

```
DROP VIEW emp_dept_view;
```

- Removes the **view permanently**
- Does **not delete underlying tables**

Real-Life Usage

- HR → Update view to include **new employee columns**
- Sales → Drop old view for discontinued reports
- Analytics → Create views for **dashboards and summaries**
- Security → Give users access to **view only**, not base tables

---

Interview Questions

Q1. Difference between CREATE VIEW and CREATE OR REPLACE VIEW?

- CREATE → creates a new view, error if exists
- CREATE OR REPLACE → updates existing view or creates if not exist

Q2. Does DROP VIEW affect base tables?

- No, only removes the view

Q3. Can we update data through a view?

- Simple views → yes
- Complex views (JOINs, aggregates) → read-only

Q4. How to add a new column to a view?

- Use **CREATE OR REPLACE VIEW** with updated SELECT query

Common Fresher Mistakes

- Dropping a view **without knowing dependencies** → breaks reports
- Using **CREATE VIEW** instead of **CREATE OR REPLACE** → error if exists
- Trying to **insert/update complex views** → not allowed
- Forgetting **aliases** in multi-table views

Best Practices

- Use **CREATE OR REPLACE VIEW** for updates

- Always use **meaningful view names**
- Avoid heavy DML on views
- Test SELECT query before creating view
- Grant **access to views instead of tables** for security
