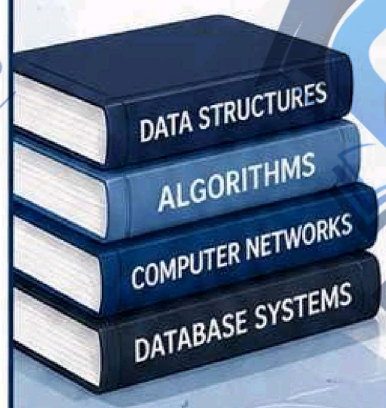
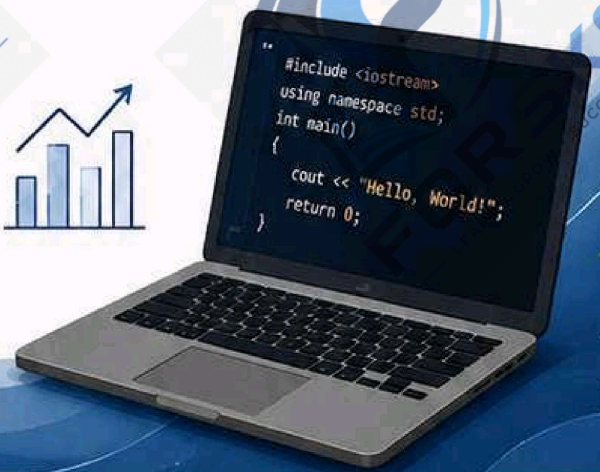

1010101010
0101010101
1010101010
0101010101
Infographics
Visual Learning
Better understanding through visual content.
Simplified Concepts
Complex topics explained in simple and visual form.
Quick Revision
Easy to revise important points and diagrams.
Exam Ready
Helps in faster revision and better exam preparation.
DATA STRUCTURES
ALGORITHMS
COMPUTER NETWORKS
DATABASE SYSTEMS

Unit - II


DATA STRUCTURES
ALGORITHMS
COMPUTER NETWORKS
DATABASE SYSTEMS


#include <iostream>
using namespace std;
int main()
{
cout << "Hello, World!";
return 0;
}

RDBMS CONSTRAINTS

Rules that ensure the **accuracy**, **reliability** and **integrity** of data in a database.

TYPES OF CONSTRAINTS

1 PRIMARY KEY

Uniquely identifies each row in a table.
No NULL values allowed.

Example

Each StudentID is unique. No two students can have the same StudentID.

SQL Example

```
StudentID INT
  PRIMARY KEY
```

2 UNIQUE

Ensures all values in a column or set of columns are unique.
Allows NULL values.

Example

Email must be unique for every user.

SQL Example

```
Email VARCHAR(100)
UNIQUE
```

3 NOT NULL

Ensures a column cannot have a NULL value.

Example

Name must be provided. It cannot be left empty.

SQL Example

```
Name VARCHAR(50)
NOT NULL
```

4 CHECK

Ensures that all values in a column satisfy a specific condition.

Example

Age must be greater than or equal to 18.

SQL Example

```
Age INT
CHECK (Age >= 18)
```

5 FOREIGN KEY

Ensures referential integrity by linking one table to another.

Example

DeptID in Employee table must exist in Department table.

SQL Example

```
DeptID INT
FOREIGN KEY (DeptID)
REFERENCES Department(DeptID)
```

6 DEFAULT

Sets a default value for a column when no value is specified.

Example

Status will be 'Active' by default.

SQL Example

```
Status VARCHAR(20)
DEFAULT 'Active'
```

WHY CONSTRAINTS MATTER?

- Maintain data accuracy and consistency
- Prevent invalid or duplicate data
- Enforce business rules
- Ensure referential integrity between tables

REFERENTIAL INTEGRITY (FOREIGN KEY)

Department

DeptID	DeptName
1	IT
2	HR
3	Sales

Employee

EmpID	EmpName	DeptID
101	Asha	1
102	Ravi	2
103	Neha	1

Together, constraints help build a **strong** and **reliable** database.

DATA CONSTRAINTS
AND ITS TYPES

Data Constraints are rules applied to table columns to ensure the accuracy, consistency and integrity of data in a database.

WHY WE USE CONSTRAINTS?

- Ensure data accuracy
- Prevent invalid or duplicate data
- Maintain consistency across the database
- Enforce business rules

TYPES OF DATA CONSTRAINTS

1 PRIMARY KEY

Uniquely identifies each row in a table.
No NULL values allowed.

Example

StudentID	Name	Age
1	Asha	20
2	Ravi	21
3	Neha	22

SQL Example

```
StudentID INT
PRIMARY KEY
```

2 UNIQUE

Ensures all values in a column or set of columns are unique.
Allows NULL values.

Example

Email
asha@email.com
ravi@email.com
neha@email.com

SQL Example

```
Email VARCHAR(100)
UNIQUE
```

3 NOT NULL

Ensures a column cannot have a NULL value.

Example

Name
Asha
Ravi
Neha

SQL Example

```
Name VARCHAR(50)
NOT NULL
```

4 CHECK

Ensures that all values in a column satisfy a specific condition.

Example

Age
21
18
25

SQL Example

```
Age INT
CHECK (Age >= 18)
```

5 FOREIGN KEY

Ensures referential integrity by linking one table to another.

Example

Department

DeptID	DeptName
1	IT
2	HR
3	Sales

Employee

EmpID	EmpName	DeptID
101	Asha	1
102	Ravi	1
103	Neha	2

SQL Example

```
DeptID INT
FOREIGN KEY (DeptID)
REFERENCES Department(DeptID)
```

6 DEFAULT

Sets a default value for a column when no value is specified.

Example

Status
Active
Active
Inactive

SQL Example

```
Status VARCHAR(20)
DEFAULT 'Active'
```

7 COMPOSITE KEY

A primary key that consists of two or more columns.

Example

OrderID	ProductID	Quantity
101	1	2
101	2	1
102	1	3

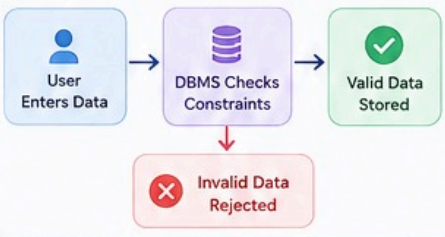
SQL Example

```
PRIMARY KEY (OrderID, ProductID)
```

KEY POINTS

- Constraints help in maintaining valid and reliable data.
- They reduce errors and improve database performance.
- Constraints are enforced by the DBMS when data is inserted or updated.

HOW CONSTRAINTS WORK?



SUMMARY TABLE

Constraint	Ensures	Allows NULL?
Primary Key	Unique + Not NULL	No
Unique	All values unique	Yes
Not NULL	No NULL values	No
Check	Values satisfy condition	Yes
Foreign Key	Referential integrity	Yes
Default	Default value set	Yes
Composite Key	Uniqueness of combination	No

OPERATIONS ON TABLE
WITH CONSTRAINTS

Constraints enforce rules on data. When we perform operations on a table, these rules are checked to maintain **accuracy**, **consistency** and **integrity**.

SAMPLE TABLE: STUDENT

```
CREATE TABLE Student (
  RollNo      INT PRIMARY KEY,
  Email       VARCHAR(100) UNIQUE,
  Name        VARCHAR(50) NOT NULL,
  Age         INT CHECK (Age >= 18),
  DeptID      INT
  FOREIGN KEY (DeptID) REFERENCES Department(DeptID)
);
```

CONSTRAINTS IN THIS TABLE

- **PRIMARY KEY** (RollNo) – Uniquely identifies each row. No NULL, no duplicate.
- **UNIQUE** (Email) – All emails must be unique.
- **NOT NULL** (Name) – Name cannot have NULL value.
- **CHECK** (Age >= 18) – Age must be 18 or greater.
- **FOREIGN KEY** (DeptID) – DeptID must exist in Department table.

DEPARTMENT TABLE (REFERENCE)

DeptID	DeptName
1	IT
2	HR
3	Sales

OPERATIONS AND EFFECT OF CONSTRAINTS

1 INSERT OPERATION

Adds new rows into the table.

EXAMPLE INSERT	RESULT	REASON
`INSERT INTO Student VALUES (101, 'asha@email.com', 'Asha', 20, 1);`	Success Row inserted.	All constraints satisfied.
`INSERT INTO Student VALUES (101, 'ravi@email.com', 'Ravi', 21, 2);`	Failed Duplicate RollNo.	Violates PRIMARY KEY (RollNo must be unique).
`INSERT INTO Student VALUES (102, 'asha@email.com', 'Asha', 22, 1);`	Failed Duplicate Email.	Violates UNIQUE constraint (Email must be unique).
`INSERT INTO Student VALUES (103, 'neha@email.com', NULL, 19, 1);`	Failed NULL in Name.	Violates NOT NULL (Name cannot be NULL).
`INSERT INTO Student VALUES (104, 'neha2@email.com', 'Neha', 17, 1);`	Failed Age less than 18.	Violates CHECK constraint (Age >= 18).
`INSERT INTO Student VALUES (105, 'karan@email.com', 'Karan', 20, 99);`	Failed Invalid DeptID.	Violates FOREIGN KEY (DeptID 99 does not exist).

2 UPDATE OPERATION

Modifies existing rows in the table.

EXAMPLE UPDATE	RESULT	REASON
`UPDATE Student SET Age = 22 WHERE RollNo = 101;`	Success Row updated.	Age remains >= 18.
`UPDATE Student SET Email = 'ravi@email.com' WHERE RollNo = 102;`	Failed Duplicate Email.	Violates UNIQUE constraint.
`UPDATE Student SET Name = NULL WHERE RollNo = 101;`	Failed NULL in Name.	Violates NOT NULL constraint.
`UPDATE Student SET Age = 15 WHERE RollNo = 103;`	Failed Age less than 18.	Violates CHECK constraint.
`UPDATE Student SET DeptID = 99 WHERE RollNo = 101;`	Failed Invalid DeptID.	Violates FOREIGN KEY constraint.

3 DELETE OPERATION

Removes rows from the table.

EXAMPLE DELETE	RESULT	REASON
`DELETE FROM Student WHERE RollNo = 101;`	Success Row deleted.	No constraint violation.
`DELETE FROM Department WHERE DeptID = 1;`	Failed Referential integrity error.	DeptID = 1 is used in Student table. (If ON DELETE RESTRICT)
`DELETE FROM Department WHERE DeptID = 1;`	Success Row deleted.	Allowed if foreign key has ON DELETE CASCADE.

HOW CONSTRAINTS AFFECT OPERATIONS

- Prevent invalid data from being inserted.
- Stop updates that break data rules.
- Restrict deletions that may break relationships.
- Maintain accuracy, consistency and integrity of the database.

COMMON CONSTRAINTS USED

- **PRIMARY KEY** – Uniquely identifies a row.
- **UNIQUE** – Ensures all values are unique.
- **NOT NULL** – Disallows NULL values.
- **CHECK** – Ensures values satisfy a condition.
- **FOREIGN KEY** – Ensures referential integrity.

KEY TAKEAWAY

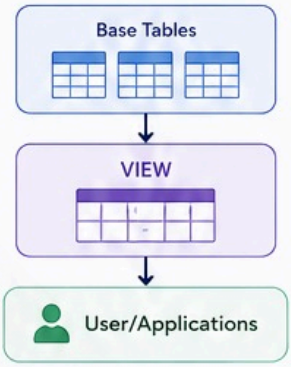
Every operation (INSERT, UPDATE, DELETE) is checked against the constraints of the table.
If a rule is broken, the operation is rejected to keep the data reliable and consistent.

Constraints + Operations = Reliable Data

VIEWS IN DBMS

NEED & ADVANTAGES

A VIEW is a virtual table based on the result of a SELECT query.
It does not store data physically.



WHAT IS A VIEW?

- A VIEW is a named, stored SELECT query.
- It presents data from one or more tables in a specific way.
- It provides a logical way to represent data.

EXAMPLE

Base Table: EMPLOYEE

EmpID	Name	DeptID	Salary
101	Asha	1	45000
102	Ravi	1	50000
103	Neha	2	48000
104	Karan	2	52000

(Shows only selected columns)

View: EMP_VIEW

EmpID	Name	Salary
101	Asha	45000
102	Ravi	50000
103	Neha	48000

```
CREATE VIEW EMP_VIEW AS SELECT EmpID, Name, Salary FROM EMPLOYEE;
```

NEED FOR VIEWS

1. SECURITY
Hide sensitive data by giving access only to required columns or rows.

2. SIMPLICITY
Simplify complex queries and present data in a user-friendly manner.

3. DATA ABSTRACTION
Provide a level of abstraction; users see what they need, not how data is stored.

4. DATA INDEPENDENCE
Changes in base tables do not affect the application if view structure remains same.

5. REUSABILITY
A view can be reused in multiple queries, reports and applications.

ADVANTAGES OF VIEWS

1. ENHANCED SECURITY
Restrict access to sensitive data. Users see only the data they are authorized to access.

2. SIMPLIFIED DATA ACCESS
Complex joins and calculations can be encapsulated in a view. Users can query the view instead of writing complex SQL.

3. DATA ABSTRACTION
Users are insulated from changes in the underlying tables. Only the view definition needs to be updated.

4. CONSISTENT DATA PRESENTATION
Views ensure consistent presentation of data across different users and applications.

5. REDUCED DATA REDUNDANCY
Data is stored in base tables only. Views do not store data, saving storage space.

6. IMPROVED PRODUCTIVITY
Developers can build views once and use them repeatedly, saving time and effort in writing queries.

EXAMPLE USE CASES

- HR department can view only employee details without salary.
- Managers can view department-wise summaries.
- Applications can use views to fetch required data easily.

UPDATABILITY OF VIEWS

Views can be updatable if:

- ✓ They are based on a single table
- ✓ They do not use DISTINCT, GROUP BY, aggregate functions, etc.

- ✗ Otherwise, they are read-only.

IMPORTANT NOTE

A View is a virtual table.
Any change in the base table data is reflected in the view automatically.

Views provide security, simplicity and abstraction, making database systems more powerful and user-friendly.

VIEWS IN DBMS

TYPES OF VIEWS

A view is a **virtual table** based on the **result of a SELECT query.**
Views can be of different types based on their function and characteristics.

BASE TABLES (Example)

STUDENT

RollNo	Name	DeptID	City
101	Asha	1	Pune
102	Ravi	1	Mumbai
103	Neha	2	Nagpur
104	Karan	2	Pune

DEPARTMENT

DeptID	DeptName	HOD
1	Computer Science	Dr. Mehta
2	Electronics	Dr. Rao

SALARY

RollNo	Salary
101	45000
102	50000
103	48000
104	52000

TYPES OF VIEWS

1 SIMPLE VIEW

A view based on a single table.
It may include some or all columns of the table.

Example SQL

```
CREATE VIEW VW_STUDENT AS
SELECT RollNo, Name, City
FROM STUDENT;
```

Result (View Output)

RollNo	Name	City
101	Asha	Pune
102	Ravi	Mumbai
103	Neha	Nagpur
104	Karan	Pune

2 COMPLEX VIEW

A view based on multiple tables using JOIN.
It combines data from related tables.

Example SQL

```
CREATE VIEW VW_STUDENT_DEPT AS
SELECT S.RollNo, S.Name, D.DeptName
FROM STUDENT S
JOIN DEPARTMENT D ON S.DeptID = D.DeptID;
```

Result (View Output)

RollNo	Name	DeptName
101	Asha	Computer Science
102	Ravi	Computer Science
103	Neha	Electronics
104	Karan	Electronics

3 DERIVED VIEW

A view based on expressions, calculations or derived columns.

Example SQL

```
CREATE VIEW VW_SALARY_BONUS AS
SELECT RollNo, Salary,
    Salary * 0.10 AS Bonus
FROM SALARY;
```

Result (View Output)

RollNo	Salary	Bonus
101	45000	4500
102	50000	5000
103	48000	4800
104	52000	5200

4 AGGREGATE VIEW

A view that uses aggregate functions like COUNT(), SUM(), AVG(), etc.

Example SQL

```
CREATE VIEW VW_DEPT_SUMMARY AS
SELECT D.DeptName, COUNT(S.RollNo) AS TotalStudents,
      AVG(SALARY) AS AvgSalary
FROM  STUDENT S
JOIN  DEPARTMENT D ON S.DeptID = D.DeptID
JOIN  SALARY SA ON S.RollNo = SA.RollNo
GROUP BY D.DeptName;
```

Result (View Output)

DeptName	TotalStudents	AvgSalary
Computer Science	2	47500.00
Electronics	2	50000.00

5 INLINE VIEW

A view defined within another query.
(Also called subquery in FROM clause)

Example SQL

```
SELECT V.Name, V.Salary
FROM (SELECT S.Name, SA.Salary
      FROM STUDENT S
      JOIN SALARY SA ON S.RollNo = SA.RollNo)
      V
WHERE V.Salary > 48000;
```

Result (Output)

Name	Salary
Ravi	50000
Karan	52000

6 MATERIALIZED VIEW

A view whose result is physically stored.
It is a snapshot of data at a point of time.
(Supported in some DBMS)

Example SQL (Oracle Syntax)

```
CREATE MATERIALIZED VIEW MV_SALARY_SUM
BUILD IMMEDIATE
REFERSH COMPLETE ON DEMAND
AS
SELECT DeptID, SUM(Salary) AS TotalSalary
FROM SALARY S
JOIN STUDENT ST ON S.RollNo = ST.RollNo
GROUP BY DeptID;
```

Result (View Output)

DeptID	TotalSalary
1	95000
2	100000

SUMMARY COMPARISON

View Type	Based On	Uses	Updatable	Stores Data Physically	Example
Simple View	Single table	Hide columns, simplify data	Yes	No	VW_STUDENT
Complex View	Multiple tables (JOIN)	Combine related data	Sometimes	No	VW_STUDENT_DEPT
Derived View	Expressions / Calculations	Computed columns	Sometimes	No	VW_SALARY_BONUS
Aggregate View	Aggregate functions	Summary / Reports	No	No	VW_DEPT_SUMMARY
Inline View	Subquery in FROM	Temporary result set	–	No	(Used inside query)
Materialized View	Query result snapshot	Performance improvement	No	Yes	MV_SALARY_SUM

KEY TAKEAWAY

- Views do not store data (except materialized views).
- They provide data abstraction, security, and simplify complex queries.
- Different types of views serve different purposes in database systems.

Views make databases more secure, flexible and user-friendly by showing the right data in the right way.

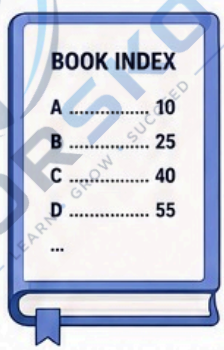
INDEX IN DBMS

NEED, TYPES & ADVANTAGES

An **Index** is a database object that improves the speed of data retrieval operations on a table, at the cost of additional storage space.

WHAT IS AN INDEX?

- An index is a data structure that improves the speed of searching and retrieving rows from a table.
- Indexes are like the index of a book, which helps find the desired topic without scanning every page.
- Indexes are automatically used by the DBMS when they can improve the performance of a query.



NEED FOR INDEX

Faster Data Retrieval
Indexes reduce the number of disk I/O operations by quickly locating the required rows.

Improved Query Performance
Speeds up SELECT queries, especially on large tables.

Efficient Sorting & Grouping
Helps in ORDER BY, GROUP BY and DISTINCT operations.

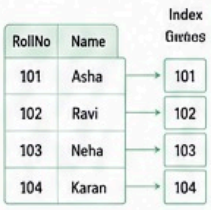
Faster Joins
Indexes on join columns improve the performance of JOIN operations.

Enforce Uniqueness
Unique indexes help in enforcing PRIMARY KEY and UNIQUE constraints.

TYPES OF INDEXES

1 PRIMARY INDEX

Index is created on the primary key of an ordered file (usually on clustered data). One primary index per table.



- Built on primary key
- Data is stored in sorted order

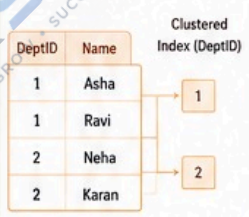
2 UNIQUE INDEX

Ensures all values in the indexed column(s) are unique. Allows at most one NULL value.

- Enforces uniqueness
- Used for UNIQUE constraint

3 CLUSTERED INDEX

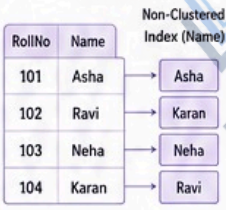
The data rows are stored in the order of the clustered index key. Only one clustered index per table.



- Data physically stored in index order

4 NON-CLUSTERED INDEX

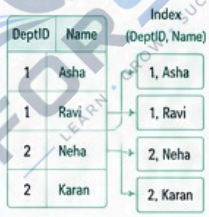
Index is created on a column that does not define the physical order of data.



- Data and index are stored separately

5 COMPOSITE INDEX

Index is created on two or more columns.



- Useful for multi-column search conditions

6 FULL-TEXT INDEX

Used to search text within large text data (varchar, char, text). Supported in some DBMS (MySQL, SQL Server).

- Optimized for text search
- Supports natural language search

ADVANTAGES OF INDEXES

Faster Data Retrieval
Greatly reduces the time taken to search and retrieve data.

Improved Query Performance
Speeds up complex queries, joins, sorting and filtering.

Efficient Sorting & Grouping
Helps in ORDER BY, GROUP BY and DISTINCT operations.

Better Scalability
Maintains good performance even as data grows.

Enforce Constraints
Helps enforce PRIMARY KEY and UNIQUE constraints efficiently.

Better User Experience
Applications become faster and more responsive.

IMPORTANT POINTS

- Indexes improve read performance but may slow down INSERT, UPDATE, DELETE operations.
- Indexes require extra storage space.
- Too many indexes can degrade overall performance.
- Always create indexes on columns used in WHERE, JOIN, ORDER BY, GROUP BY clauses.
- The DBMS automatically decides whether to use an index or not.

EXAMPLE

Create Non-Clustered Index

```
CREATE INDEX idx_name ON Student(Name);
```

Create Composite Index

```
CREATE INDEX idx_dept_name ON Student(DeptID, Name);
```

Indexes are powerful tools to boost database performance, but they should be used wisely and only when needed.

SQL JOINS

Combine rows from two or more tables based on a related column between them.

Example Tables

STUDENT (S)

ID	Name	DeptID
1	Asha	10
2	Ravi	20
3	Neha	30
4	Karan	NULL

DEPARTMENT (D)

DeptID	DeptName
10	Computer
20	Electronics
40	IT

SYNTAX

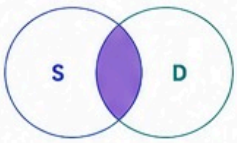
```
SELECT columns
FROM Table1 [JOIN TYPE] JOIN Table2
ON Table1.column = Table2.column;
```

Blue : Table 1 (STUDENT)
Green : Table 2 (DEPARTMENT)
Gray : No Match

TYPES OF JOINS

1 INNER JOIN

Returns only matching rows from both tables.



SQL

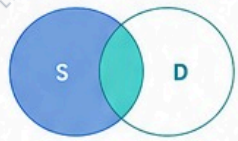
```
SELECT * FROM Student S
INNER JOIN Department D
ON S.DeptID = D.DeptID;
```

Result

ID	Name	DeptID	DeptName
1	Asha	10	Computer
2	Ravi	20	Electronics

2 LEFT JOIN

Returns all rows from the left table (S) and matching rows from the right table (D).



SQL

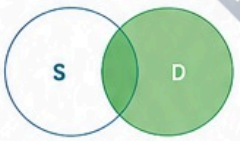
```
SELECT * FROM Student S
LEFT JOIN Department D
ON S.DeptID = D.DeptID;
```

Result

ID	Name	DeptID	DeptName
1	Asha	10	Computer
2	Ravi	20	Electronics
3	Neha	30	NULL
4	Karan	NULL	NULL

3 RIGHT JOIN

Returns all rows from the right table (D) and matching rows from the left table (S).



SQL

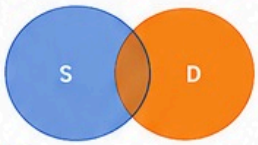
```
SELECT * FROM Student S
RIGHT JOIN Department D
ON S.DeptID = D.DeptID;
```

Result

ID	Name	DeptID	DeptName
1	Asha	10	Computer
2	Ravi	20	Electronics
NULL	NULL	40	IT

4 FULL OUTER JOIN

Returns all rows when there is a match in either left (S) or right (D) table. Unmatched rows are NULL.



SQL

```
SELECT * FROM Student S
FULL OUTER JOIN Department D
ON S.DeptID = D.DeptID;
```

Result

ID	Name	DeptID	DeptName
1	Asha	10	Computer
2	Ravi	20	Electronics
3	Neha	30	NULL
4	Karan	NULL	NULL
NULL	NULL	40	IT

5 CROSS JOIN

Returns the Cartesian product of both tables.

SQL

```
SELECT * FROM Student S
CROSS JOIN Department D;
```

Result (Total Rows = 4 × 3 = 12)

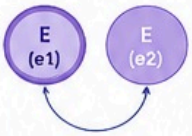
ID	Name	DeptID	DeptName
1	Asha	10	Computer
1	Asha	20	Electronics
1	Asha	40	IT
2	Ravi	10	Computer
...	...	...	...

6 SELF JOIN

Joins a table with itself. Useful for hierarchical data.

EMPLOYEE (E)

EmpID	Name	ManagerID
1	Alice	NULL
2	Bob	1
3	Carol	1
4	David	2



SQL

```
SELECT e1.Name AS Employee, e2.Name AS Manager
FROM Employee e1
LEFT JOIN Employee e2
ON e1.ManagerID = e2.EmpID;
```

Result

Employee	Manager	NdLL
Alice	Nargar	Alice
Bob	Alice	Alice
David	Boio	Bob

7 NATURAL JOIN

Automatically joins tables using all columns with the same name and data type.

TABLE A

ID	Name
1	Asha
2	Ravi
3	Neha

=

TABLE B

ID	City
1	Pune
2	Mumbai
4	Nagpur

SQL

```
SELECT * FROM TableA
NATURAL JOIN TableB;
```

Result

ID	Name	City
1	Asha	Pune
2	Ravi	Mumbai

JOIN TYPES COMPARISON

Join Type	Matches	Left Table Rows	Right Table Rows	Unmatched Rows Included?
INNER JOIN	Only matches	Yes	Yes	No
LEFT JOIN	All left + matches	Yes	Yes	Left only
RIGHT JOIN	All right + matches	Yes	Yes	Right only
FULL OUTER JOIN	All + matches	Yes	Yes	Both sides
CROSS JOIN	All combinations	Yes	Yes	No (all paired)
SELF JOIN	Within same table	Yes	Yes	Depends
NATURAL JOIN	Based on same column names	Yes	Yes	No

KEY TAKEAWAYS

- INNER JOIN returns only matching rows.
- LEFT JOIN returns all from left, matched from right.
- RIGHT JOIN returns all from right, matched from left.
- FULL OUTER JOIN returns all rows from both tables.
- CROSS JOIN returns all possible combinations.
- SELF JOIN is used to join a table with itself.
- NATURAL JOIN automatically matches same-named columns.

Joins help you combine data from multiple tables to get meaningful information.
Choose the right join type based on your requirement.

TYPES OF JOINS IN SQL

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

Table: EMPLOYEE (E)

EmpID	Name	DeptID
1	Asha	10
2	Ravi	20
3	Neha	30
4	Karan	NULL

Table: DEPARTMENT (D)

DeptID	DeptName	Location
10	Computer	Pune
20	Electronics	Mumbai
30	IT	Nagpur
40	HR	Delhi

= Table E (Employee)
= Table D (Department)
= Matching Rows

1 INNER JOIN

Returns only the rows that have matching values in both tables.

Venn Diagram

SQL Syntax

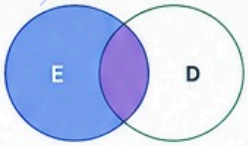
```
SELECT E.EmpID, E.Name, D.DeptName
FROM Employee E
INNER JOIN Department D
ON E.DeptID = D.DeptID;
```

Result (E JOIN D)

EmpID	Name	DeptName
1	Asha	Computer
2	Ravi	Electronics
3	Neha	IT

2 LEFT JOIN

Returns all rows from the left table (E) and the matching rows from the right table (D).




```
SELECT E.EmpID, E.Name, D.DeptName
FROM Employee E
LEFT JOIN Department D
ON E.DeptID = D.DeptID;
```

Result (E LEFT JOIN D)

EmpID	Name	DeptName
1	Asha	Computer
2	Ravi	Electronics
3	Neha	IT
4	Karan	NULL

3 RIGHT JOIN

Returns all rows from the right table (D) and the matching rows from the left table (E).


```
SELECT E.EmpID, E.Name, D.DeptName
FROM Employee E
RIGHT JOIN Department D
ON E.DeptID = D.DeptID;
```

Result (E RIGHT JOIN D)

EmpID	Name	DeptName
1	Asha	Computer
2	Ravi	Electronics
3	Neha	IT
NULL	NULL	HR

4 FULL OUTER JOIN

Returns all rows when there is a match in either table. Unmatched rows will contain NULL.


```
SELECT E.EmpID, E.Name, D.DeptName
FROM Employee E
FULL OUTER JOIN Department D
ON E.DeptID = D.DeptID;
```

Result (FULL OUTER JOIN)

EmpID	Name	DeptName
1	Asha	Computer
2	Ravi	Electronics
3	Neha	IT
4	Karan	NULL
NULL	NULL	HR

5 CROSS JOIN

Returns the Cartesian product of both tables. All combinations of rows.


```
SELECT E.EmpID, E.Name, D.DeptName
FROM Employee E
CROSS JOIN Department D;
```

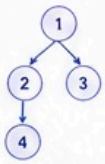
Result (CROSS JOIN)

EmpID	Name	DeptName
1	Asha	Computer
1	Asha	Electronics
1	Neha	IT
1	Asha	HR
...	...	...

6 SELF JOIN

Joins a table with itself. Useful for hierarchical data.

EmpID	Name	ManagerID
1	Alice	NULL
2	Bob	1
3	Carol	1
4	David	2




```
SELECT E1.Name AS Employee,
   E2.Name AS Manager
FROM Employee E1
LEFT JOIN Employee E2
ON E1.ManagerID = E2.EmpID;
```

Result (SELF JOIN)

Employee	Manager
Alice	NULL
Bob	Alice
Carol	Alice
David	Bob

7 NATURAL JOIN

Automatically joins tables using columns that have the same name and data type.

```
SELECT *
FROM Employee
NATURAL JOIN Department;
```

Result (NATURAL JOIN)

EmpID	Name	DeptID	DeptName
1	Asha	10	Computer
2	Ravi	20	Electronics
3	Neha	30	IT

QUICK COMPARISON

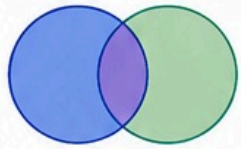
Join Type	Returns	Matched Rows	Unmatched from Left (E)	Unmatched from Right (D)
INNER JOIN	Only matching rows	Yes	No	No
LEFT JOIN	All from left + matched right	Yes	Yes	No
RIGHT JOIN	All from right + matched left	Yes	No	Yes
FULL OUTER JOIN	All from both sides	Yes	Yes	Yes
CROSS JOIN	All combinations	Yes	Yes	Yes
SELF JOIN	Table joined with itself	Yes	Depends	Depends
NATURAL JOIN	Auto join on same column names	Yes	No	No

KEY TAKEAWAYS

- INNER JOIN returns only matching rows.
- LEFT JOIN keeps all rows from the left table.
- RIGHT JOIN keeps all rows from the right table.
- FULL OUTER JOIN keeps everything from both tables.
- CROSS JOIN returns all possible combinations.
- SELF JOIN is used to compare rows within the same table.
- NATURAL JOIN matches same-named columns automatically.

Choose the right join type to get the data you need!

UNION OF TABLES AND ITS TYPES



The UNION operator is used to combine the result sets of two or more SELECT statements.

TABLE: EMPLOYEE

EmpID	Name	Department
1	Asha	IT
2	Ravi	IT
3	Neha	HR
4	Karan	HR

TABLE: CONTRACTOR

EmpID	Name	Department
3	Neha	HR
4	Karan	HR
5	Sohan	Admin
6	Pooja	Admin

BASIC UNION SYNTAX

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

Both SELECT statements must have:
- Same number of columns
- Similar data types
- Columns in the same order

TYPES OF UNION

1 UNION (DISTINCT)

Combines the result sets and eliminates duplicate rows. (Default behavior)

Syntax

```
SELECT column_list FROM table1
UNION
SELECT column_list FROM table2;
```

Example: EMPLOYEE ∪ CONTRACTOR (Using UNION)

EMPLOYEE

EmpID	Name	Department
1	Asha	IT
2	Ravi	IT
3	Neha	HR
4	Karan	HR

CONTRACTOR

EmpID	Name	Department
3	Neha	HR
4	Karan	HR
5	Sohan	Admin
6	Pooja	Admin

UNION →

RESULT (UNION)

EmpID	Name	Department
1	Asha	IT
2	Ravi	IT
3	Neha	HR
4	Karan	HR
5	Sohan	Admin
6	Pooja	Admin

Duplicate rows (Neha, Karan) appear only once.

2 UNION ALL

Combines the result sets and includes all rows, including duplicates.

Syntax

```
SELECT column_list FROM table1
UNION ALL
SELECT column_list FROM table2;
```

Example: EMPLOYEE ∪ CONTRACTOR (Using UNION ALL)

EMPLOYEE

EmpID	Name	Department
1	Asha	IT
2	Ravi	IT
3	Neha	HR
4	Karan	HR

CONTRACTOR

EmpID	Name	Department
3	Neha	HR
4	Karan	HR
5	Sohan	Admin
6	Pooja	Admin

UNION ALL →

RESULT (UNION ALL)

EmpID	Name	Department
1	Asha	IT
2	Ravi	IT
3	Neha	HR
4	Karan	HR
3	Neha	HR
4	Karan	HR
5	Sohan	Admin
6	Pooja	Admin

Duplicate rows (Neha, Karan) appear as many times as they occur.

3 UNION CORRESPONDING BY (SQL Standard)

Combines result sets by matching columns with the same names, not just by position.

Syntax

```
SELECT column_list FROM table1
UNION CORRESPONDING BY (column_name, ...)
SELECT column_list FROM table2;
```

Example: UNION CORRESPONDING BY (Name_ Department)

TABLE A

EmpID	Name	Department
1	Asha	IT
2	Ravi	IT
3	Neha	HR

TABLE B

Name	EmpID	Department
Neha	3	HR
Karan	4	HR
Sohan	5	Admin

UNION CORRESPONDING BY (Name, Department) →

RESULT

Name	Department
Asha	IT
Ravi	IT
Neha	HR
Karan	HR
Sohan	Admin

Matching is done by column names (Name, Department), not by position.

IMPORTANT POINTS

- UNION removes duplicates.
- UNION ALL keeps all duplicates.
- Both SELECT statements must have the same number of columns and compatible data types.
- Column names in the result are taken from the first SELECT.
- ORDER BY can be used only at the end of the final result.

Example with ORDER BY

```
SELECT Name FROM Employee
UNION
SELECT Name FROM Contractor
ORDER BY Name;
```

QUICK COMPARISON

Type	Duplicates	Column Match	Performance	Use When
UNION	Removes duplicates	By position	Slower (due to duplicate check)	You want unique records only
UNION ALL	Keeps duplicates	By position	Faster	You want all records, including duplicates
UNION CORRESPONDING BY	Removes duplicates	By column name	Varies	Column order may be different in tables

Use UNION for unique results, UNION ALL for complete results including duplicates, and UNION CORRESPONDING BY when column order is different.

IN-BUILT FUNCTIONS

CHARACTER FUNCTIONS & NUMBER FUNCTIONS

In-built functions are predefined functions in DBMS that perform operations on values and return a result.

A CHARACTER FUNCTIONS (String Functions)

These functions are used to perform operations on character or string data.

Function	Description	Syntax	Example	Result
UPPER()	Converts a string to upper case.	UPPER(string)	UPPER('forsko')	'FORSKO'
LOWER()	Converts a string to lower case.	LOWER(string)	LOWER('FORSkO')	'forsko'
INITCAP()	Converts the first character of each word to upper case.	INITCAP(string)	INITCAP('hello world from dbms')	'Hello World From Dbms'
LENGTH() or LEN()	Returns the length (number of characters) of a string.	LENGTH(string) LEN(string)	LENGTH('forsko')	6
SUBSTR() or SUBSTRING()	Extracts a part of a string.	SUBSTR(string, start, length)	SUBSTR('database', 1, 4)	'data'
INSTR() or CHARINDEX()	Returns the position of a substring in a string.	INSTR(string, substring) (Starts from 1)	INSTR('database', 'base')	5
LTRIM()	Removes leading (spaces) from a string.	LTRIM(string)	LTRIM(' forsko')	'forsko'
RTRIM()	Removes trailing (spaces) from a string.	RTRIM(string)	RTRIM('forsko ')	'forsko'
TRIM()	Removes leading and trailing spaces.	TRIM(string)	TRIM(' forsko ')	'forsko'
CONCAT()	Concatenates two or more strings.	CONCAT(str1, str2, ...)	CONCAT('Data', ' Base')	'Data Base'
REPLACE()	Replaces a substring with another string.	REPLACE(string, old, new)	REPLACE('dbms is fun','fun', 'easy')	'dbms is easy'
REVERSE()	Reverses a string.	REVERSE(string)	REVERSE('hello')	'olleh'
A→65 ASCII()	Returns ASCII value of the first character.	ASCII(string)	ASCII('A')	65
65→A CHAR()	Returns character for the given ASCII value.	CHAR(ascii_code)	CHAR(65)	'A'

EXAMPLE QUERY (CHARACTER FUNCTIONS)

```
SELECT Name,
       UPPER(Name) AS UpperName,
       LENGTH(Name) AS NameLength,
       SUBSTR(Name, 1, 3) AS First3Chars
FROM   Student;
```

Student Table

Name
Asha
Ravi
Neha

Output (Partial)

Name	UpperName	NameLength	First3Chars
Asha	ASHA	4	Ash
Ravi	RAVI	4	Rav
Neha	NEHA	4	Neh

B NUMBER FUNCTIONS

These functions are used to perform operations on numeric data.

Function	Description	Syntax	Example	Result
ABS()	Returns the absolute value.	ABS(number)	ABS(-25)	25
SQRT()	Returns the square root.	SQRT(number)	SQRT(16)	4
POWER() or POW()	Returns x raised to the power y.	POWER(x, y) POW(x, y)	POWER(2, 3)	8
ROUND()	Rounds a number to a specified number of decimal places.	ROUND(number, decimals)	ROUND(45.678,2)	45.68
CEIL() or CEILING()	Returns the smallest integer greater than or equal to the number.	CEIL(number) CEILING(number)	CEIL(7.2)	8
FLOOR()	Returns the largest integer less than or equal to the number.	FLOOR(number)	FLOOR(7.8)	7
TRUNC()	Truncates the decimal part of a number.	TRUNC(number, decimals)	TRUNC(45.678)	45
MOD() or (Modulus)	Returns the remainder after division.	MOD(a, b) a % b	MOD(10,3) 10 % 3	1
SIGN()	Returns the sign of a number.	SIGN(number)	SIGN(-10)	-1
EXP()	Returns e raised to the given power.	EXP(number)	EXP(1)	2.71828
LOG() or LN()	Returns logarithm (base 10) or natural logarithm.	LOG(number) LN(number)	LOG(100) LN(2.71828)	2 1
GREATEST()	Returns the largest value in the list.	GREATEST(a, b, c, ...)	GREATEST(5,7,3)	7
LEAST()	Returns the smallest value in the list.	LEAST(a, b, c, ...)	LEAST(5,7,3)	3

EXAMPLE QUERY (NUMBER FUNCTIONS)

```
SELECT Marks,
       ABS(Marks) AS AbsMarks,
       ROUND(Marks,1) AS Rounded,
       SQRT(Marks) AS SquareRoot,
       MOD(Marks,3) AS ModResult
FROM   Exam;
```

Exam Table

Marks
25
-16
49.5

Output (Partial)

Marks	AbsMarks	Rounded	SquareRoot	ModResult
25	25	25.0	5	1
-16	16	-16.0	4	2
49.5	49.5	49.5	7.03	1

KEY POINTS

- Character functions work on string (CHAR, VARCHAR, TEXT) data.
- Number functions work on numeric (INT, FLOAT, DECIMAL, etc.) data.
- These functions make queries powerful and results more meaningful.
- They are widely used in WHERE, SELECT, ORDER BY, GROUP BY, HAVING clauses.

COMMONLY USED TOGETHER

```
ROUND(AVG(Marks),2)
UPPER(TRIM(Name))
LENGTH(CONCAT(FirstName, ' ', LastName))
MOD(TotalMarks,2) = 0  → Check Even/Odd
```

In-built functions save time, reduce complexity and make your SQL queries smarter!

IN-BUILT FUNCTIONS IN DBMS

DATE FUNCTIONS & GROUP FUNCTIONS

In-built functions are predefined functions in DBMS that perform operations on values and return a result.

A DATE FUNCTIONS

These functions are used to perform operations on date and time data.

Function	Description	Syntax	Example	Result
CURRENT_DATE()	Returns the current date.	CURRENT_DATE()	SELECT CURRENT_DATE();	2025-05-15
CURRENT_TIME()	Returns the current time.	CURRENT_TIME()	SELECT CURRENT_TIME();	14:30:25
CURRENT_TIMESTAMP()	Returns the current date and time.	CURRENT_TIMESTAMP()	SELECT CURRENT_TIMESTAMP();	2025-05-15 14:30:25
DATE()	Extracts date part from a datetime value.	DATE(datetime)	SELECT DATE('2025-05-15 14:30:25');	2025-05-15
TIME()	Extracts time part from a datetime value.	TIME(datetime)	SELECT TIME('2025-05-15 14:30:25');	14:30:25
YEAR()	Returns the year part.	YEAR(date)	SELECT YEAR('2025-05-15');	2025
MONTH()	Returns the month part (1-12).	MONTH(date)	SELECT MONTH('2025-05-15');	5
DAY()	Returns the day of month (1-31).	DAY(date)	SELECT DAY('2025-05-15');	15
HOUR()	Returns the hour part (0-23).	HOUR(time)	SELECT HOUR('14:30:25');	14
MINUTE()	Returns the minute part (0-59).	MINUTE(time)	SELECT MINUTE('14:30:25');	30
SECOND()	Returns the second part (0-59).	SECOND(time)	SELECT SECOND('14:30:25');	25
DATE_ADD()	Adds interval to a date.	DATE_ADD(date, INTERVAL expr unit)	SELECT DATE_ADD('2025-05-15', INTERVAL 10 DAY);	2025-05-25
DATE_SUB()	Subtracts interval from a date.	DATE_SUB(date, INTERVAL expr unit)	SELECT DATE_SUB('2025-05-15', INTERVAL 5 DAY);	2025-05-10
DATEDIFF()	Returns the number of days between two dates.	DATEDIFF(date1, date2)	SELECT DATEDIFF('2025-05-15', '2025-05-01');	14

EXAMPLE (DATE FUNCTIONS)

```
SELECT  Name,
        DOB,
        YEAR(DOB) AS BirthYear,
        AGE(CURDATE(), DOB) AS Age,
        DATE_ADD(DOB, INTERVAL 1 YEAR) AS NextBirthday
FROM    Student;
```

Output (Partial)

Name	DOB	BirthYear	Age	NextBirthday
Asha	2003-01-10	2003	22	2026-01-10
Ravi	2004-06-20	2004	20	2025-06-20
Neha	2003-11-05	2003	21	2026-11-05

DATE UNITS

YEAR	MONTH	HOUR	MINUTE	SECOND
INTERVAL 1 YEAR	INTERVAL 1 MONTH	INTERVAL 1 DAY	INTERVAL 1 MIOUR	INTERVAL 1 SECOND

B GROUP FUNCTIONS (Aggregate Functions)

These functions perform calculations on a set of values and return a single result.

Function	Description	Syntax	Example	Result
COUNT()	Returns the number of rows.	COUNT(*) or COUNT(column)	SELECT COUNT(*) FROM Student;	120
SUM()	Returns the sum of numeric values.	SUM(column)	SELECT SUM(Marks) FROM Exam;	6150
AVG()	Returns the average of numeric values.	AVG(column)	SELECT AVG(Marks) FROM Exam;	51.25
MIN()	Returns the minimum value.	MIN(column)	SELECT MIN(Marks) FROM Exam;	12
MAX()	Returns the maximum value.	MAX(column)	SELECT MAX(Marks) FROM Exam;	98
SUM(DISTINCT)	Returns sum of distinct values.	SUM(DISTINCT column)	SELECT SUM(DISTINCT Salary) FROM Emp;	1250000
COUNT(DISTINCT)	Returns the count of distinct values.	COUNT(DISTINCT column)	SELECT COUNT(DISTINCT DeptID) FROM Emp;	5

EXAMPLE (GROUP FUNCTIONS)

```
SELECT  DeptID,
        COUNT(*) AS TotalEmp,
        AVG(Salary) AS AvgSalary,
        MIN(Salary) AS MinSalary,
        MAX(Salary) AS MaxSalary,
        SUM(Salary) AS TotalSalary
FROM    Employee
GROUP BY DeptID;
```

Output

DeptID	TotalEmp	AvgSalary	MinSalary	MaxSalary	TotalSalary
10	25	45560.00	22000	90000	1139000
20	30	48266.67	21000	95000	1448000
30	20	39850.00	20000	85000	797000
40	15	51666.67	25000	100000	775000
50	30	46200.00	23000	90000	1386000

GROUP BY with HAVING

```
SELECT  DeptID, AVG(Salary) AS AvgSal
FROM    Employee
GROUP BY DeptID
HAVING  AVG(Salary) > 45000;
```

→

Output

DeptID	AvgSal
20	48266.67
40	51666.67
50	46200.00

KEY POINTS

- Group functions work on a set of values and return a single result.
- They are commonly used with GROUP BY to group rows.
- COUNT(*) counts all rows including NULLs.
- COUNT(column) counts only non-NULL values in that column.
- GROUP BY comes before HAVING in SQL execution.
- Use HAVING to filter groups (not rows).

QUICK COMPARISON

Aspect	Date Functions	Group Functions
Purpose	Work with date and time values	Work with groups of rows
Input	A date/time value	A set of values (column)
Output	A single date/time or part of it	A single aggregated value
Example	YEAR(DOB), DATE_ADD(DOB, INTERVAL 1 DAY)	COUNT(*), SUM(Salary), AVG(Marks)

COMMONLY USED TOGETHER

```
SELECT  DeptID,
        COUNT(*) AS TotalEmp,
        SUM(Salary) AS TotalSalary
FROM    Employee
WHERE   JoinDate >= '2025-01-01'
GROUP BY DeptID
HAVING COUNT(*) > 5;
```

Use the right function for the right task to make your SQL queries efficient and powerful!

CONVERSION FUNCTIONS

IN SQL (DBMS)

Conversion functions are used to convert a value from one data type to another.

A → B
Convert Data Type

WHY USE CONVERSION FUNCTIONS?

- Ensure compatibility between different data types.
- Perform calculations on different data types.
- Display data in the required format.
- Avoid errors in implicit type conversion.

SYNTAX

```
FUNCTION ( expression AS target_data_type )
or
FUNCTION ( expression, target_data_type )
```

SUPPORTED DATA TYPES

- CHAR / VARCHAR / TEXT
- INTEGER / BIGINT / SMALLINT
- DECIMAL / NUMERIC / FLOAT
- DATE / TIME / TIMESTAMP

COMMON CONVERSION FUNCTIONS

Function	Description	Syntax	Example	Input (Type)	Output (Type)	Result
CAST()	Converts a value of one data type to another.	CAST(expression AS data_type)	SELECT CAST('123' AS INT);	'123' (TEXT)	INT	123
CONVERT()	Converts a value of one data type to another (using style for date).	CONVERT(data_type, expression [, style])	SELECT CONVERT(INT, '456');	'456' (TEXT)	INT	456
TO_CHAR()	Converts a value to a character string.	TO_CHAR(expression [, format])	SELECT TO_CHAR(12345.678, '99999.99');	12345.678 (NUMBER)	VARCHAR	12345.68
TO_NUMBER()	Converts a value to a numeric value.	TO_NUMBER(expression, [format])	SELECT TO_NUMBER('789.56');	'789.56' (TEXT)	NUMBER	789.56
TO_DATE()	Converts a value to a DATE.	TO_DATE(expression, 'format')	SELECT TO_DATE('15-05-2025', 'DD-MM-YYYY');	'15-05-2025' (TEXT)	DATE	15-MAY-25
TO_TIMESTAMP()	Converts a value to a TIMESTAMP.	TO_TIMESTAMP(expression, 'format')	SELECT TO_TIMESTAMP('15-05-2025 14:30:00', 'DD-MM-YYYY HH24:MI:SS');	'15-05-2025 14:30:00' (TEXT)	TIMESTAMP	15-MAY-2025 14:30:00
TO_DATE (with style)	Converts a string to date using style (SQL Server).	CONVERT(DATE, expression, style)	SELECT CONVERT(DATE, '05/15/2025', 101);	'05/15/2025' (TEXT)	DATE	2025-05-15
TO_BINARY()	Converts a value to binary.	CAST(expression AS BINARY(n))	SELECT CAST('AB' AS BINARY(2));	'AB' (TEXT)	BINARY	0x4142
TO_BOOLEAN()	Converts a value to Boolean (TRUE/FALSE).	CAST(expression AS BOOLEAN)	SELECT CAST(1 AS BOOLEAN);	1 (INT)	BOOLEAN	TRUE

EXAMPLE QUERIES

Purpose	Query	Output (Example)	Explanation
String to Integer	SELECT CAST('100' AS INT);	100	Converts string '100' to integer.
String to Date	SELECT TO_DATE('2025-05-15','YYYY-MM-DD');	15-MAY-25	Converts string to DATE.
Date to String	SELECT TO_CHAR(SYSDATE,'DD-MON-YYYY');	15-MAY-2025	Converts date to formatted string.
String to Number (Decimal)	SELECT TO_NUMBER('123.45');	123.45	Converts string to decimal number.
Number to String	SELECT TO_CHAR(98765,'99999');	98765	Converts number to string.
Date using CONVERT (SQL Server)	SELECT CONVERT(DATE,'15/05/2025',103);	2025-05-15	Converts string to date using style code.
Timestamp from String	SELECT TO_TIMESTAMP('15-05-2025 14:30:00', 'DD-MM-YYYY HH24:MI:SS');	15-MAY-2025 14:30:00	Converts string to timestamp.
Boolean Conversion	SELECT CAST(0 AS BOOLEAN);	FALSE	Converts number to Boolean.

KEY POINTS

- Conversion functions help in explicit conversion of data types.
- Always use correct format while converting dates.
- Invalid conversions may cause errors.
- CAST() is ANSI standard, CONVERT() is more flexible (SQL Server).
- Use TO_CHAR() and TO_NUMBER() mainly in Oracle.

COMMONLY USED TOGETHER

TO_CHAR(SYSDATE, 'DD-MON-YYYY')	→	Display date as string
TO_DATE('15-05-2025', 'DD-MM-YYYY')	→	Convert string to date
CAST('100' AS INT)	→	Convert string to integer
TO_NUMBER('123.45')	→	Convert string to number

Use conversion functions to ensure accurate calculations, proper comparisons, and formatted outputs in your queries!

USE OF ALIAS, NULL, NVL & SUB QUERIES

Simplify Queries • Handle Missing Data • Retrieve Smart Results

1 USE OF ALIAS

An alias is a temporary name given to a table or a column in a query. It is used to make queries easier to read and understand.

TYPES OF ALIAS

- **Column Alias** – Gives a new name to a column.
- **Table Alias** – Gives a short name to a table.

SYNTAX

Column Alias

```
SELECT column_name AS alias_name
FROM table_name;
```

Table Alias

```
SELECT column_name(s)
FROM table_name AS alias_name;
```

EXAMPLE

Column Alias

```
SELECT emp_name AS "Employee Name",
       salary AS "Monthly Salary"
FROM employee;
```

Employee Name	Monthly Salary
Asha	45000
Ravi	52000
Neha	47000

Table Alias

```
SELECT e.emp_id, e.emp_name, e.salary
FROM employee e;
```

Alias makes queries shorter and improves readability.

2 NULL

NULL represents a missing, unknown or inapplicable value. It is not equal to zero or an empty string.

KEY POINTS

- NULL means no value.
- NULL cannot be compared using =.
- Use IS NULL or IS NOT NULL to check NULL.
- NULL in arithmetic operations returns NULL.

EXAMPLE TABLE

Student

RollNo	Name	Dept	Marks
1	Asha	IT	85
2	Ravi	CS	NULL
3	Neha	IT	78
4	Karan	CS	NULL

EXAMPLES

```
-- Students with NULL in Marks
SELECT * FROM Student WHERE Marks IS NULL;

-- Students with NOT NULL in Marks
SELECT * FROM Student WHERE Marks IS NOT NULL;
```

NOTES

- NULL is treated as unknown.
- COUNT(column_name) ignores NULL values.
- COUNT(*) counts all rows.

NULL COMPARISON

```
-- Wrong
SELECT * FROM Student WHERE Marks = NULL;

-- Correct
SELECT * FROM Student WHERE Marks IS NULL;
```

3 NVL (NULL VALUE)

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

SYNTAX

NVL(expression, replacement_value)

- expression – The column or value to check.
- replacement_value – The value returned if expression is NULL.

EXAMPLE TABLE

Employee

EmpID	Name	Dept	Bonus
1	Asha	IT	5000
2	Ravi	CS	NULL
3	Neha	IT	NULL
4	Karan	CS	3000

EXAMPLES

```
-- Replace NULL bonus with 0
SELECT emp_id, name, NVL(bonus, 0) AS bonus
FROM employee;
```

EmpID	Name	Bonus
1	Asha	5000
2	Ravi	0
3	Neha	0
4	Karan	3000

```
-- Replace NULL dept with 'Not Assigned'
SELECT emp_id, name, NVL(dept, 'Not Assigned') AS dept
FROM employee;
```

EmpID	Name	Dept
1	Asha	IT
2	Ravi	CS
3	Neha	IT
4	Karan	CS

KEY POINTS

- NVL() returns the expression value if it is not NULL.
- If expression is NULL, it returns replacement_value.
- Helps in calculations and display with default values.

COMMON USES

Replace NULL salary with 0

```
SELECT NVL(salary, 0) FROM employee;
```

Replace NULL text with 'N/A'

```
SELECT NVL(address, 'N/A') FROM employee;
```

Replace NULL date with today's date

```
SELECT NVL(join_date, SYSDATE) FROM employee;
```

NVL(expr, value) is same as
IF expr IS NULL THEN value ELSE expr END

4 SUB QUERIES

A sub query (or inner query) is a query nested inside another query. The inner query is executed first, and its result is used by the outer query.

TYPES OF SUB QUERIES

- **Single Row Sub Query** – Returns one row.
- **Multiple Row Sub Query** – Returns multiple rows.
- **Correlated Sub Query** – Refers to outer query.

SYNTAX

Single Row Sub Query

```
SELECT column(s)
FROM table
WHERE column = (sub query);
```

Multiple Row Sub Query

```
SELECT column(s)
FROM table
WHERE column IN (sub query);
```

Correlated Sub Query

```
SELECT column(s)
FROM table t1
WHERE EXISTS
   (SELECT 1 FROM table t2
    WHERE t2.col = t1.col);
```

EXAMPLES

1. Single Row Sub Query

```
-- Employees earning more than average salary
SELECT * FROM employee
WHERE salary > (SELECT AVG(salary) FROM employee);
```

2. Multiple Row Sub Query

```
-- Employees from departments located in location 'Pune' or 'Mumbai'
SELECT * FROM employee
WHERE dept_id IN (SELECT dept_id FROM department
                  WHERE location IN ('Pune', 'Mumbai'));
```

3. Correlated Sub Query

```
-- Employees who earn more than the average salary of their department
SELECT e1.* FROM employee e1
WHERE salary >  (SELECT AVG(salary)
                 FROM employee e2
                 WHERE e2.dept_id = e1.dept_id);
```

KEY POINTS

- Sub queries can be used in SELECT, FROM, WHERE, HAVING clauses.
- Use IN/ANY/ALL for multiple row sub queries.
- Use EXISTS for correlated sub queries.

Inner Query (Executed First) → Result → Outer Query (Uses the Result)

BENEFITS

- Break complex problems into smaller parts.
- Improve readability and maintainability.
- Powerful filtering and comparison.

Use these features to write smarter, cleaner and more efficient SQL queries!