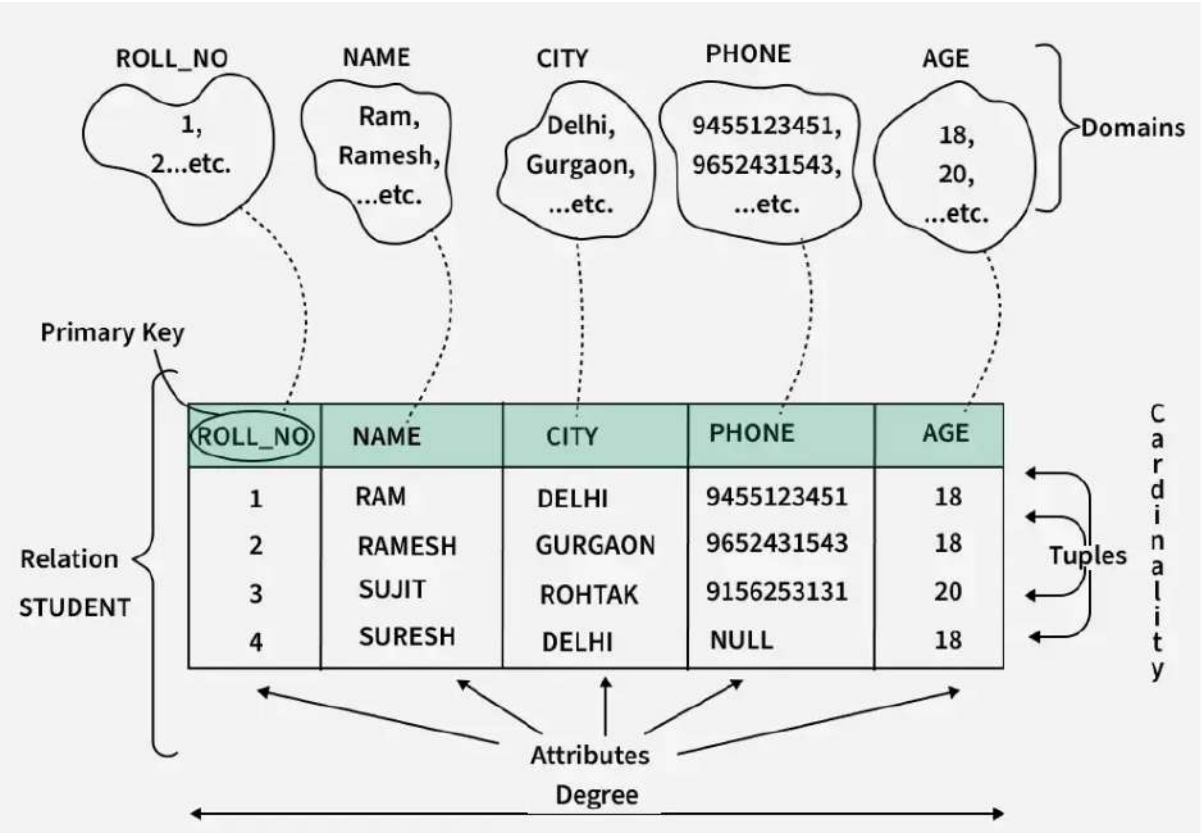
Introduction to Relational Data Model

The Relational Model **organizes data using tables** (relations) consisting of **rows** and **columns**.

- The relational model **represents how data is stored** and **managed in Relational Databases** where data is organized into tables, each known as a relation.
- Each **row** of a table represents an **entity or record** and each **column** represents a **particular attribute** of that entity.
- The relational model transforms conceptual designs from ER diagrams into implementable structures. These structures are used in relational database systems like Oracle SQL and MySQL.

Example: Consider a relation STUDENT with attributes ROLL_NO, NAME, ADDRESS, PHONE and AGE shown in the table.



Relational Model

Key Terms in the Relational Model

1. **Attribute:** Attributes are the properties that define an entity. For example, ROLL_NO, NAME, ADDRESS etc.
2. **Relation Schema:** A relation schema defines the structure of the relation and represents the name of the relation with its attributes.

   For example, STUDENT (ROLL_NO, NAME, ADDRESS, PHONE and AGE) is the relation schema for STUDENT. If a schema has more than 1 relation it is called Relational Schema.
3. **Tuple:** A Tuple represents a row in a relation. Each tuple contains a set of attribute values that describe a particular entity.

   For example, (1, RAM, DELHI, 9455123451, 18) is a tuple in the STUDENT table.
4. **Relation Instance:** The **set of tuples of a relation** at a **particular instance of time** is called a relation instance.

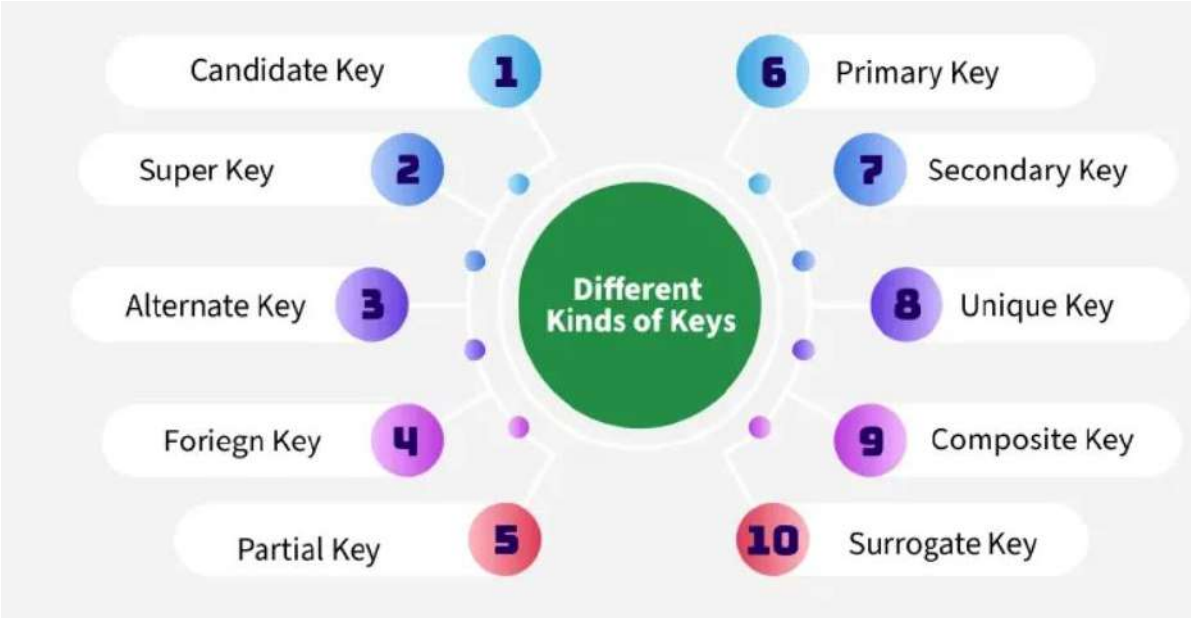
   It can change whenever there is an insertion, deletion or update in the database.
5. **Degree:** The **number of attributes** in the relation is known as the degree of the relation.

   For example, The STUDENT relation has a degree of 5, as it has 5 attributes.
6. **Cardinality:** The **number of tuples in a relation** is known as cardinality. For example, The STUDENT relation defined above has cardinality 4.
7. **NULL Values:** The value which is not known or unavailable is called a NULL value. It is represented by NULL. For example, PHONE of STUDENT having ROLL_NO 4 is NULL.

Types of Keys

In the context of a relational database, keys are one of the basic requirements of a relational database model.

- **Keys** are **fundamental components** that **ensure data integrity, uniqueness**, and **efficient access**.
- It is widely **used to identify the tuples**(rows) **uniquely** in the table.
- We also use keys to **set up relations** amongst various columns and tables of a relational database.



Keys in DBMS

Why do we require Keys in a DBMS?

Keys are important in a Database Management System (DBMS) for several reasons:

- **Uniqueness**: Keys ensure that **each record in a table is unique** and can be identified distinctly.

- **Data Integrity**: Keys **prevent data duplication** and **maintain** the **consistency** of the data.
- **Efficient Data Retrieval**: By defining relationships between tables, keys **enable faster querying** and better **data organization**. Without keys, it would be extremely difficult to manage large datasets and queries would become inefficient and prone to errors.

Types of Database Keys

1. Super Key

The **set of one or more attributes** (columns) that can **uniquely identify** a tuple (record) is known as Super Key.

It may include extra attributes that are not important for uniqueness but still uniquely identify the row. For Example, STUD_NO, (STUD_NO, STUD_NAME), etc.

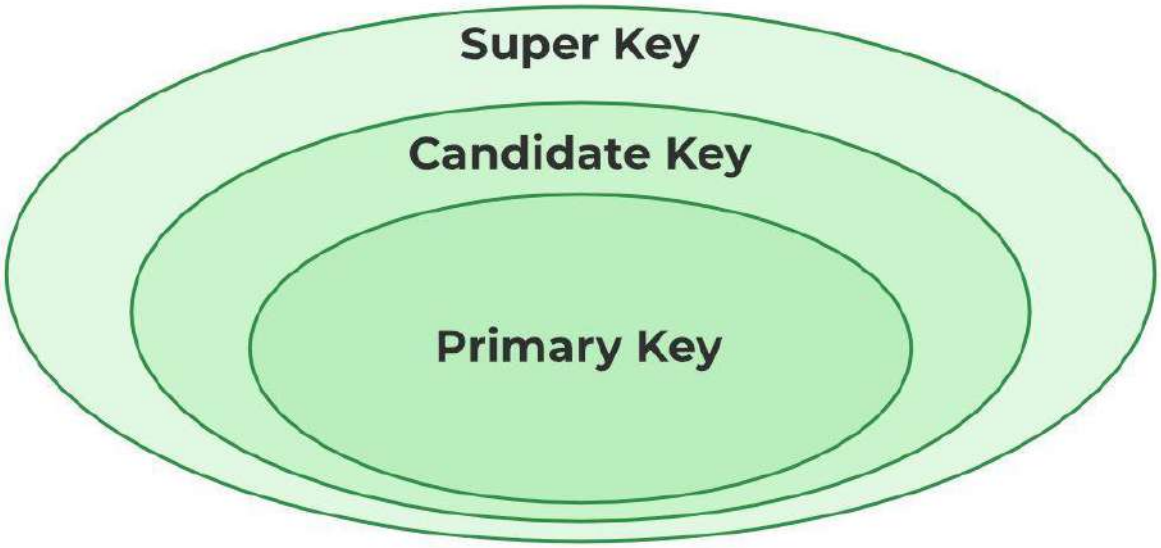
- A super key is a group of single or multiple keys that uniquely identifies rows in a table. It **supports NULL values** in rows.
- For example, if the "STUD_NO" column can uniquely identify a student, adding "SNAME" to it will still form a valid super key, though it's unnecessary.

Example: Consider the STUDENT table

STUD_NO	SNAME	ADDRESS	PHONE
1	Shyam	Delhi	123456789
2	Rakesh	Kolkata	223365796

STUD_NO	SNAME	ADDRESS	PHONE
3	Suraj	Delhi	175468965

A super key could be a combination of STUD_NO and PHONE, as this combination uniquely identifies a student.



Relation between Primary Key, Candidate Key, and Super Key

2. Candidate Key

The **minimal set of attributes** that can uniquely identify a tuple is known as a candidate key.

For Example, STUD_NO in STUDENT relation.

- A candidate key **is a minimal super key**, meaning it can uniquely identify a record but **contains no extra attributes**.
- It is a super key with no repeated data is called a candidate key.
- The minimal set of attributes that can uniquely identify a record.

- A **candidate key must contain unique values**, ensuring that no two rows have the same value in the candidate key's columns.
- **Every table must have at least a single candidate key**.
- A table can have **multiple candidate keys** but only one primary key.

Example: For the STUDENT table below, STUD_NO can be a candidate key, as it uniquely identifies each record.

STUD_NO	SNAME	ADDRESS	PHONE
1	Shyam	Delhi	123456789
2	Rakesh	Kolkata	223365796
3	Suraj	Delhi	175468965

Table: STUDENT_COURSE

STUD_NO	TEACHER_NO	COURSE_NO
1	001	C001
2	056	C005

A composite candidate key example: {STUD_NO, COURSE_NO} can be a candidate key for a STUDENT_COURSE table.

3. Primary Key

There can be more than one candidate key in relation out of which **one can be chosen as the primary key**.

For Example, STUD_NO, as well as STUD_PHONE, are candidate keys for relation STUDENT but STUD_NO can be chosen as the primary key (only one out of many candidate keys).

- A primary key is a unique key, meaning it can uniquely identify each record (tuple) in a table.
- It must have **unique values** and cannot contain any duplicate values.
- A primary key **cannot be NULL**, as it needs to provide a valid, unique identifier for every record.
- A primary key does not have to consist of a single column. In some cases, a **composite primary key** (made of multiple columns) can be used to uniquely identify records in a table.
- Databases typically store rows ordered in memory according to primary key for fast access of records using primary key.

Example:

STUDENT table -> Student(STUD_NO, SNAME, ADDRESS, PHONE) , STUD_NO is a primary key

Table: STUDENT

STUD_NO	SNAME	ADDRESS	PHONE
1	Shyam	Delhi	123456789
2	Rakesh	Kolkata	223365796

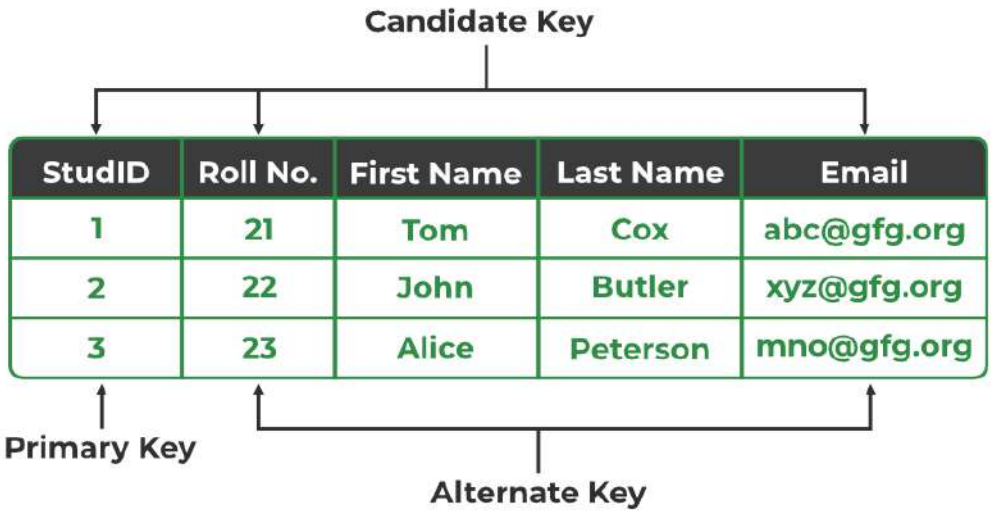
STUD_NO	SNAME	ADDRESS	PHONE
3	Suraj	Delhi	175468965

4. Alternate Key

An **alternate key is any candidate key** in a table **that is not chosen as the primary key**. In other words, all the keys that are not selected as the primary key are considered alternate keys.

- An alternate key is also **referred** to as a **secondary key** because it can uniquely identify records in a table, just like the primary key.
- An alternate key can **consist of one or more columns** (fields) that can **uniquely identify** a record, but it is not the primary key

Example: In the STUDENT table, both STUD_NO and PHONE are candidate keys. If STUD_NO is chosen as the primary key, then PHONE would be considered an alternate key.

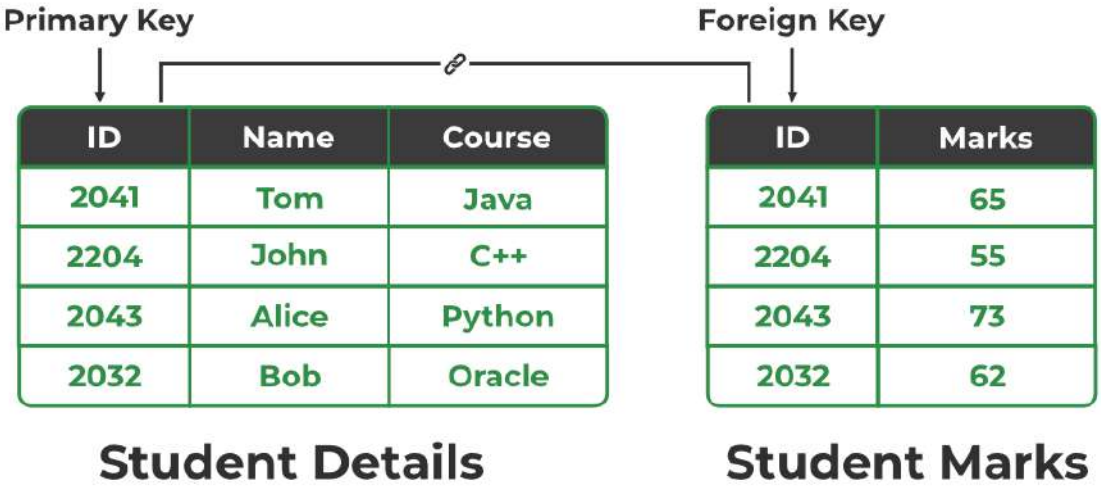


Primary Key, Candidate Key, and Alternate Key

5. Foreign Key

A foreign key is an attribute in one table **that refers to the primary key in another table**.

The table that contains the foreign key is called the **referencing table** and the table that is referenced is called the **referenced table.**



Relation between Primary Key and Foreign Key

- A foreign key in one table points to the primary key in another table, establishing a relationship between them.
- It helps **connect two or more tables**, enabling you to create relationships between them. This is important for maintaining data integrity and preventing data redundancy.
- They act as a cross-reference between the tables.

Example: Consider the STUDENT_COURSE table

STUD_NO	TEACHER_NO	COURSE_NO
1	005	C001

STUD_NO	TEACHER_NO	COURSE_NO
2	056	C005

Explanation:

- Here, STUD_NO in the STUDENT_COURSE table is a foreign key that references the STUD_NO primary key in the STUDENT table.
- Unlike the Primary Key of any given relation, **Foreign Key can be NULL as well as may contain duplicate tuples** i.e. it need not follow uniqueness constraint. For Example, STUD_NO in the STUDENT_COURSE relation is not unique.
- It has been repeated for the first and third tuples. However, the STUD_NO in STUDENT relation is a primary key and it needs to be always unique and it cannot be null.

6. Composite Key

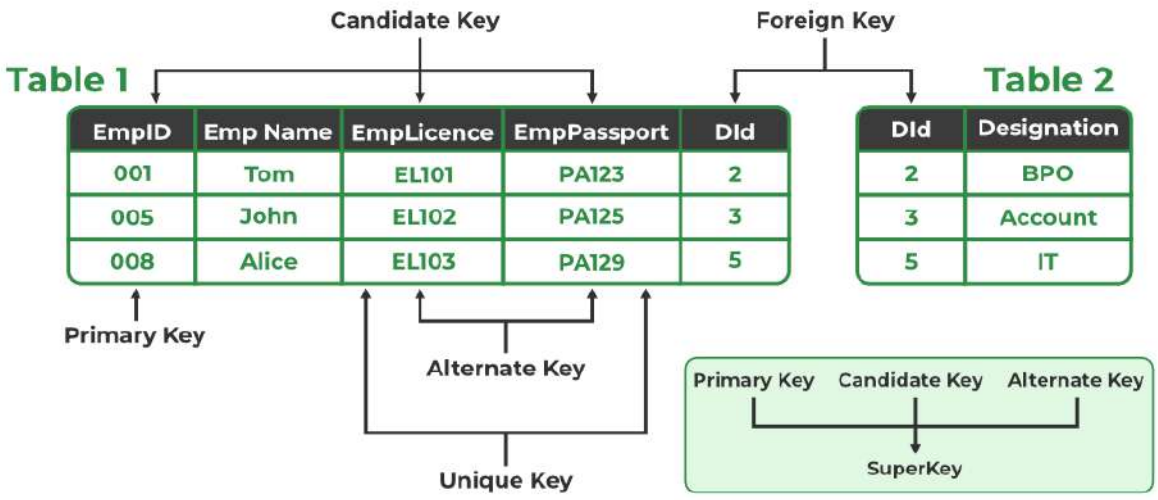
Sometimes, a table might not have a single column/attribute that uniquely identifies all the records of a table.

To uniquely identify rows of a table, a combination of two or more columns/attributes can be used.

It still **can give duplicate values** in rare cases. So, we need to find the optimal set of attributes that can uniquely identify rows in a table.

- It acts as a primary key if there is no primary key in a table
- Two or more attributes are used together to make a composite key.
- Different combinations of attributes may give different accuracy in terms of identifying the rows uniquely.

Example: In the STUDENT_COURSE table, {STUD_NO, COURSE_NO} can form a composite key to uniquely identify each record.



Different Types of Keys

Integrity Constraints

Integrity constraints are a **set of rules** used in DBMS to ensure that the data in a **database is accurate**, **consistent** and **reliable**.

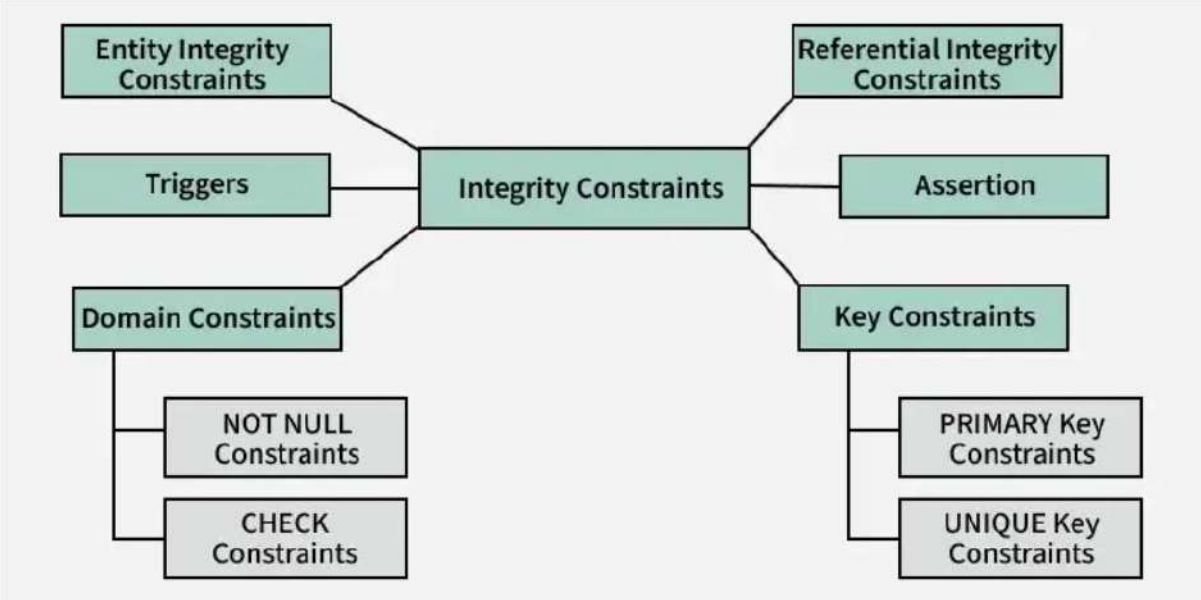
These rules helps in **maintaining** the **quality of data** by ensuring that the processes **like adding, updating or deleting** information do not harm the integrity of the database.

Integrity constraints also define **how different parts of the database are connected** and **ensure that these relationships remain valid.**

They play an essential role in making sure the data is meaningful and follows the logical structure of the database.

Integrity constraints in a Database Management System are rules that help keep the data in a database accurate, consistent and reliable.

They act like a set of guidelines that ensure all the information stored in the database follows specific standards.



Integrity Constraints

Example: Making sure every customer has a valid email address & ensuring that an order in the database is always linked to an existing customer.

Note: *These rules prevent mistakes, such as adding incomplete or incorrect data, and make sure the database is secure and well-organized.*

Types of Integrity Constraints

There are Different types of Integrity Constraints used in DBMS, these are:

1. Domain Constraints
2. Entity Integrity Constraints
3. Key Constraints
4. Referential integrity constraints
5. Assertion
6. Triggers

1. Domain Constraints

Domain constraints **are a type of integrity constraint that ensure the values stored in a column (or attribute) of a database are valid and within a specific range** or domain.

In simple terms, they define what type of data is allowed in a column and restrict invalid data entry.

The data type of domain include string, char, time, integer, date, currency etc. The value of the attribute must be available in comparable domains.

Example: Below table demonstrates domain constraints in action by enforcing rules for each column

Student_Id	Name	Semester	Age
21CSE100	Aniket Kumar	6th	20

Student_Id	Name	Semester	Age
21CSE101	Shashwat Dubey	7th	21
21CSE102	Manvendra Sharma	8th	22
21CSE103	Ashmit Dubey	5th	20

1. **Student_Id: Must be unique and follow a specific format like 21CSE###**. No duplicates or invalid formats allowed.
2. **Name: Accepts only valid text** (no numbers) and cannot be left empty (NOT NULL constraint).
3. **Semester:** Allows specific values like 5th, 6th, etc., and ensures valid input (e.g., no 10th if not permitted).
4. **Age:** Must be an integer within a reasonable range (e.g., 18-30) and cannot contain invalid data like negative numbers or text.

Types of Domain Constraints:

- **NOT NULL Constraint:** Ensures No records can have NULL value.
- **CHECK Constraint:** This Constraint Checks for any specified condition over any attribute.

Why Domain Constraints Are Important :

- They prevent invalid or inconsistent data from entering the database.

- They ensure the database is reliable and follows predefined business rules.
- They make the database easier to manage and maintain by reducing errors.

Example: Let, the not-null constraint be specified on the "Semester" attribute in the relation/table given below, then the data entry of 4th tuple will violate this integrity constraint, because the "Semester" attribute in this tuple contains null value. To make this database instance a legal instance, its entry must not be allowed by database management system.

Student_id	Name	Semester	Age
21CSE1001	Sonali Rao	5th	20
21CSE1012	Anjali Gupta	5th	21
21CSE1023	Aastha Singh	5th	22
21CSE1034	Ayushi Singh	NULL	20

2. Entity Integrity Constraints

Entity integrity constraints **state that primary key can never contain null value** because primary key is used to determine individual rows in a relation uniquely, if primary key contains null value then we cannot identify those rows.

A table can contain null value in it except primary key field.

Key Features of Entity Integrity Constraints:

- **Uniqueness:** The primary key value must be unique for each row in the table. No duplicate entries are allowed in the primary key column.
- **NOT NULL:** The primary key column cannot contain NULL values, as every row must have a valid identifier.
- **Essential for Table Design:** Ensures that every record in the table can be uniquely identified, preventing ambiguity.

Example: It is not allowed because it is containing primary key (Student_id) as NULL value.

Student_id	Name	Semester	Age
21CSE101	Ramesh	5th	20
21CSE102	Kamlesh	5th	21
21CSE103	Aakash	5th	22
NULL	Mukesh	5th	20

3. Key Constraints

Key constraints **ensure that certain columns or combinations of columns in a table uniquely identify** each row. These rules

are essential for maintaining data integrity and preventing duplicate or ambiguous records.

Why Key Constraints Are Important ?

- Prevent Duplicates: Ensure unique identification of rows.
- Maintain Relationships: Enable proper linking between tables (via foreign keys).
- Enforce Data Integrity: Prevent invalid or inconsistent data.

Example: It is now acceptable because all rows must be unique.

Student_id	Name	Semester	Age
21CSE101	Ramesh	5th	20
21CSE102	Kamlesh	5th	21
21CSE103	Aakash	5th	22
21CSE102	Mukesh	5th	20

4. Referential integrity constraints

Referential integrity constraints are **rules that ensure relationships between tables remain consistent**.

They enforce that a **foreign key in one table must either match a value in the referenced primary key** of another table **or be NULL.**

This **guarantees the logical connection** between related tables in a relational database.

Why Referential Integrity Constraints Are Important ?

- **Maintains Consistency**: Ensures relationships between tables are valid.
- **Prevents Orphan Records**: Avoids **cases where a record in a child table references a non-existent parent** record.
- **Enforces Logical Relationships**: Strengthens the logical structure of a relational database.

Example: Here, in below example Block_No 22 entry is not allowed because it is not present in 2nd table.

Student_id	Name	Semester	Block_No
22CSE101	Ramesh	5th	20
21CSE105	Kamlesh	6th	21
22CSE102	Aakash	5th	20
23CSE106	Mukesh	2nd	22

Block_No	Block Location
20	Chandigarh
21	Punjab
25	Delhi

Anomalies in Relational Model

When we notice any ***unexpected behavior*** **while working with relational databases**, there may be a **presence of too much redundancy** in the data stored in the **database**.

This **can cause anomalies** in the DBMS and it can be of various types such as:

- **Insertion Anomalies:** It is the **inability to insert data in the database due to the absence** of other data.
- **For example:** Suppose we are dividing the whole class into groups for a project and the *GroupNumber* attribute is defined so that null values are not allowed.
- If a new student is admitted to the class but not immediately assigned to a group then this student can't be inserted into the database.
- **Deletion Anomalies -** It is the **accidental loss of data** in the database upon deletion of any other data element.
- **For example:** Suppose, we have an employee relation that contains the details of the employee along with the department they are working in.
- Now, if a department has only one employee working in it and we remove the information of this employee from the table, **there will be a loss of data related to the department**.
- This **can lead to *data inconsistency***.
- **Modification/Update Anomalies -** It is the data inconsistency that arises **from data redundancy and partial updation of data** in the database.

- **For example:** Suppose, while updating the data into the database duplicate entries were entered.
- Now, if the user does not realize that the data is stored redundantly after updation, there will be data inconsistency in the database.

All these anomalies **can lead to unexpected behavior and inconvenience for the user**. These anomalies can be removed with the help of a process known as **normalization**.

Codd Rules in DBMS

Edgar F. Codd, the **creator of the relational model proposed 13 rules known as Codd Rules** that state:

For a **database to be considered as a perfect relational database**, it **must follow the following rules**:

1. **Foundation Rule -** The database must be able to manage data in relational form.
2. **Information Rule -** All data stored in the database must **exist as a value** of some table **cell**.
3. **Guaranteed Access Rule - Every unique data** element should be accessible by **only a combination of the table name, primary key** value, and the **column name**.
4. **Systematic Treatment of NULL values -** Database must support NULL values.
5. **Active Online Catalog -** The organization of the database must exist in an online catalog that can be queried by authorized users.

6. **Comprehensive Data Sub-Language Rule -** Database must **support at least one language** that supports: data definition, view definition, data manipulation, integrity constraints, authorization, and transaction boundaries.

   This rule says that a **database system must provide a single, comprehensive language.**

7. **View Updating Rule -** All views should be theoretically and practically updatable by the system.

   This rule says that: Every view defined in the database should be updatable.

   That means users should be able to insert, update, or delete data through the view.

   When a user modifies data in the view, those changes should correctly propagate back to the underlying base tables.

   Theoretically updatable: In principle, the **view should be** defined in such a way that an **update can be mapped back to the base tables** without **ambiguity** or **conflict**.

   Practically updatable: The **database system should actually support this operation without errors** or **manual intervention**. It means the DBMS's software can perform the **update automatically**.

8. **Relational Level Operation Rule -** The database must support high-level insertion, updation, and deletion operations.

9. **Physical Data Independence Rule -** Data stored in the database must be independent of the applications that can

access it i.e., the data stored in the database must not depend on any other data or an application.

10. **Logical Data Independence Rule -** Any change in the logical representation of the data (structure of the tables) must not affect the user's view.
11. **Integrity independence -** Changing the integrity constraints at the database level should not reflect any change at the application level.
12. **Distribution independence -** The database must work properly even if the data is **stored in multiple locations** or is being used by **multiple end-users**.
13. **Non-subversion Rule -** Accessing the **data by low-level relational language should not be able to bypass** the integrity rules and constraints **expressed in the high-level relational language.**

Relational Algebra

Relational Algebra is a **formal language used to query** and **manipulate relational databases**, consisting of a set of operations like **selection**, **projection**, **union**, and **join**.

It provides a **mathematical framework for querying databases**, ensuring **efficient** data **retrieval** and **manipulation**.

Relational algebra serves as the mathematical foundation for query SQL.

Relational algebra simplifies the process of querying databases and makes it **easier to understand** and **optimize query execution** for better performance.

It is essential for learning SQL because SQL queries are based on relational algebra operations, enabling users to retrieve data effectively.

Key Concepts in Relational Algebra

Before explaining relational algebra operations, let's define some **fundamental concepts**:

1. Relations: In relational algebra, a relation is a table that consists of rows and columns, representing data in a structured format. Each relation has a unique name and is made up of tuples.

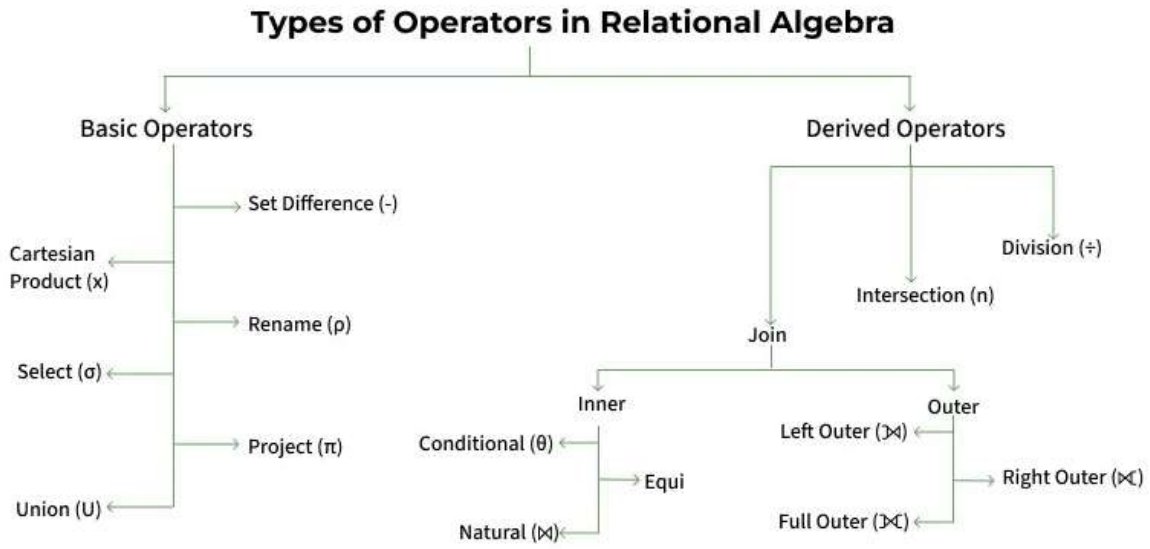
2. Tuples: A tuple is a single row in a relation, which contains a set of values for each attribute. It represents a single data entry or record in a relational table.

3. Attributes: Attributes are the columns in a relation, each representing a specific characteristic or property of the data. For example, in a "Students" relation, attributes could be "Name", "Age", and "Grade".

4. Domains: A domain is the set of possible values that an attribute can have. It defines the type of data that can be stored in each column of a relation, such as integers, strings, or dates.

Basic Operators in Relational Algebra

Relational algebra consists of various **basic operators** that help us to fetch and manipulate data from **relational tables** in the database to perform certain operations on relational data. **Basic operators** are fundamental operations that include selection (σ), projection (π), union (U), set difference (−), Cartesian product (×), and rename (ρ).



Operators in Relational Algebra

1. Selection(σ)

The Selection Operation is basically used to **filter out rows** from a given table **based on certain given condition**.

It basically allows us to retrieve only those rows that match the condition as per condition passed during SQL Query.

Example: If we have a relation **R** with attributes **A**, **B**, and **C**, and we want to select tuples where **C > 3**, we write:

A	B	C
1	2	4
2	2	3
3	2	3
4	3	4

σ(c>3)(R) *will select the tuples which have c more than 3.*

Output:

A	B	C
1	2	4
4	3	4

Explanation: The **selection operation only filters rows** but does not display or change their order. The **projection operator is used for displaying specific columns.**

2. Projection(π)

While **Selection operation works on rows**, similarly **projection operation of relational algebra works on columns.**

It basically **allows us to pick specific columns** from a given relational table based on the given condition and ignoring all the other remaining columns.

Example: Suppose we want columns B and C from Relation R.

π(B,C)(R) will show following columns.

Output:

B	C
2	4
2	3

B	C
3	4

Explanation: By Default, **projection operation removes duplicate values.**

3. Union(U)

The **Union** Operator is basically **used to combine the results of two queries into a single result**.

The **only condition is that both queries must return same number of columns with same data types**.

Union operation in relational algebra is the same as union operation in set theory.

Example: Consider the following table of Students having different optional subjects in their course.

FRENCH

Student_Name	Roll_Number
Ram	01
Mohan	02
Vivek	13
Geeta	17

GERMAN

Student_Name	Roll_Number
Vivek	13
Geeta	17
Shyam	21
Rohan	25

If **FRENCH** and **GERMAN** relations represent student names in two subjects, we can combine their student names as follows:

π(Student_Name)(FRENCH) U π(Student_Name)(GERMAN)

Output:

Student_Name
Ram
Mohan
Vivek
Geeta
Shyam

Student_Name
Rohan

Explanation: The only constraint in the union of two relations is that both relations **must have the same set of Attributes**.

4. Set Difference(-)

Set difference basically **provides the rows that are present in one table, but not in another tables.**

Set Difference in relational algebra is the same set difference operation as in set theory.

Example: To find students enrolled only in FRENCH but not in GERMAN, we write:

π(Student_Name)(FRENCH) - π(Student_Name)(GERMAN)

Student_Name
Ram
Mohan

Explanation: The **only constraint in the Set Difference between two relations is that both relations must have the same set of Attributes.**

5. Rename(ρ)

Rename operator basically **allows you to give a temporary name to a specific relational table or to its columns.**

It is very useful when we want to avoid ambiguity, especially in complex Queries.

Rename is a unary operation used for renaming attributes of a relation.

Example: We can rename an attribute **B** in relation **R** to **D**

A	B	C
1	2	4
2	2	3
3	2	3
4	3	4

ρ(D/B) R will rename the attribute 'B' of the relation by 'D".

Output Table:

A	D	C
1	2	4
2	2	3
3	2	3
4	3	4

6. Cartesian Product(X)

The **Cartesian product** combines **every row of one table with every row of another table**, **producing all the possible combination.**

It's mostly used as a precursor to more complex operation like joins.

Let's say A and B, so the **cross product between A X B** will **result** in all the **attributes of A followed by each attribute of B.**

Each record of A will pair with every record of B.

Relation A:

Name	Age	Sex
Ram	14	M
Sona	15	F
Kim	20	M

Relation B:

ID	Course
1	DS

ID	Course
2	DBMS

Output: If **relation A has 3 rows** and **relation B has 2 rows**, the Cartesian product **A × B** will **result in 6 rows.**

Name	Age	Sex	ID	Course
Ram	14	M	1	DS
Ram	14	M	2	DBMS
Sona	15	F	1	DS
Sona	15	F	2	DBMS
Kim	20	M	1	DS
Kim	20	M	2	DBMS

Explanation: If A has 'n' tuples and B has 'm' tuples then A X B will have 'n*m' tuples.

7. Set Intersection (∩)

Set Intersection basically allows to **fetches only those rows of data that are common between two sets of relational tables**.

Set Intersection in relational algebra is the same set intersection operation in set theory.

Example: Consider the following table of Students having different optional subjects in their course.

Relation FRENCH

Student_Name	Roll_Number
Ram	01
Mohan	02
Vivek	13
Geeta	17

Relation GERMAN

Student_Name	Roll_Number
Vivek	13
Geeta	17
Shyam	21

Student_Name	Roll_Number
Rohan	25

From the above table of FRENCH and GERMAN, the Set Intersection is used as follows:

π(Student_Name) (FRENCH)∩ π(Student_Name) (GERMAN)

Output:

Student_Name
Vivek
Geeta

Explanation: The only constraint in the Set Difference between two relations is that both relations must have the same set of Attributes.

Practice Questions

Schema Definitions (Assume these tables exist)

1. Student(SID, Name, Age, Dept)

2. Course(CID, Title, Credits)

3. Enrolled(SID, CID, Grade)

1. Find all students who are older than 20.
2. Get the details of students from the 'Computer Science' department.
3. List the names of all students.
4. Get the course titles.
5. Get the IDs of students who are enrolled in at least one course.
6. Get the IDs of all students who are **not enrolled** in any course.
7. List names of students along with the titles of courses they are enrolled in.
8. Find the names of students who got a grade 'A'
9. Rename the Student relation to S.
10. Get names of students and the titles of the courses they are enrolled in, where the course has more than 3 credits.

Answers –

1. σ Age > 20 (Student)
2. σ Dept = 'Computer Science' (Student)
3. π Name (Student)
4. π Title (Course)
5. π SID (Enrolled)
6. π SID (Student) − π SID (Enrolled)
7. Student ⋈ Enrolled ⋈ Course
8. π Name (σ Grade = 'A' (Student ⋈ Enrolled))
9. ρ S(Student)
10. π Name, Title (σ Credits > 3 (Student ⋈ Enrolled ⋈ Course))

Derived Operators in Relational Algebra

Derived operators are built using basic operators and include operations like join, intersection, and division. These operators help perform more complex queries by combining basic operations to meet specific data retrieval needs.

Joins in DBMS

- **Join** is an operation in DBMS (Database Management System) that **combines the rows of two or more tables based on related columns** between them.

  The **main purpose of join is to retrieve the data from multiple tables** in other words Join is used to perform multi-table queries.
  It is denoted by ⋈.

Syntax

$$R3 \leftarrow \bowtie^{(R1)}\ \langle join\_condition \rangle\ ^{(R2)}$$

where R1 and R2 are two relations to be joined and R3 is a relation that will hold the result of the join operation.

Example

$$Temp \leftarrow \bowtie^{(student)}_{S.roll=E.roll}{}^{(Exam)}$$

where S and E are aliases of the student and exam respectively.

JOIN Example

Consider the two tables below as follows:

ROLL_NO	NAME	ADDRESS	PHONE	Age
1	HARSH	DELHI	XXXXXXXXXX	18
2	PRATIK	BIHAR	XXXXXXXXXX	19
3	RIYANKA	SILIGURI	XXXXXXXXXX	20
4	DEEP	RAMNAGAR	XXXXXXXXXX	18
5	SAPTARHI	KOLKATA	XXXXXXXXXX	19
6	DHANRAJ	BARABAJAR	XXXXXXXXXX	20
7	ROHIT	BALURGHAT	XXXXXXXXXX	18
8	NIRAJ	ALIPUR	XXXXXXXXXX	19

COURSE_ID	ROLL_NO
1	1
2	2
2	3
3	4
1	5
4	9
5	10
4	11

Table 1 - Student

Table 2 - Student_Course

Both these tables are connected by one common key (column) i.e. ROLL_NO.

We can perform a JOIN operation using the given relational algebra:

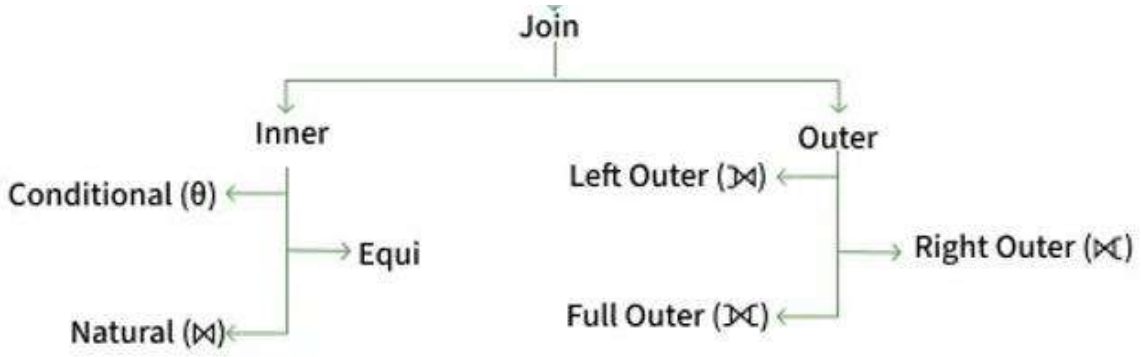
Student ⋈ Student_course

Output:

ROLL_NO	NAME	ADDRESS	PHONE	AGE	COURSE_ID
1	HARSH	DELHI	xxxxxxxxxx	18	1

ROLL_NO	NAME	ADDRESS	PHONE	AGE	COURSE_ID
2	PRATIK	BIHAR	xxxxxxxxxx	19	2
3	PRIYANKA	SILIGURI	xxxxxxxxxx	20	2
4	DEEP	RAMNAGAR	xxxxxxxxxx	18	3
5	SAPTARHI	KOLKATA	xxxxxxxxxx	19	1

Types of Join

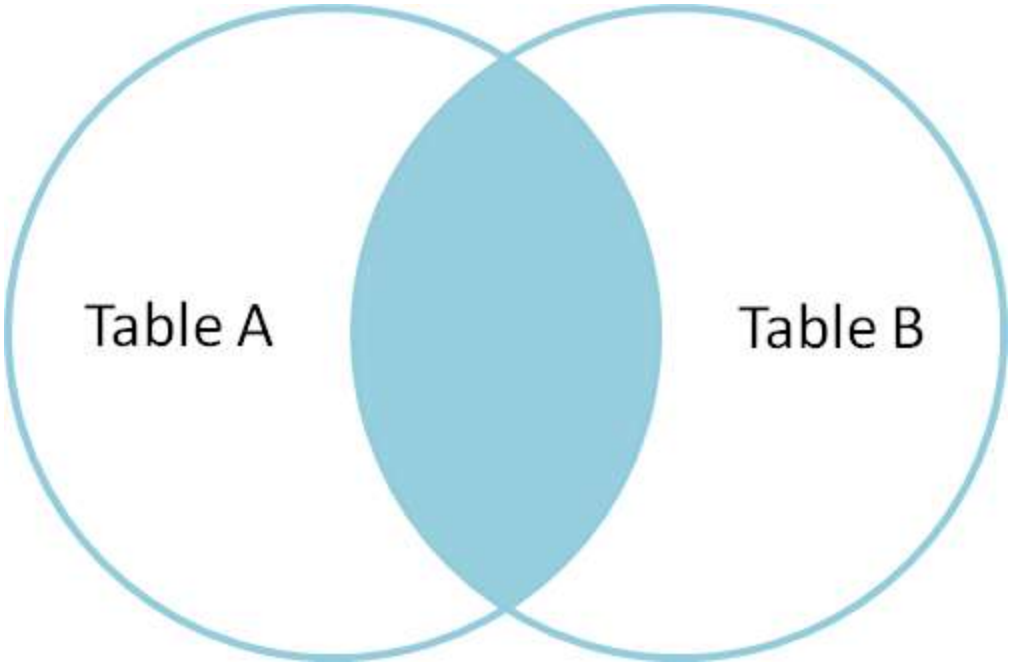


There are many types of Joins in SQL. Depending on the use case, you can use different types of SQL JOIN clauses.

Here are the frequently used SQL JOIN types:

1. Inner Join

Inner Join is a join operation in DBMS that **combines two or more tables based on related columns and returns only rows that have matching values among tables**.



Inner Join

Inner join has different types.

- Conditional join
- Equi Join
- Natural Join

(a) **Conditional Join**

Conditional join or Theta join is a type of inner join in which **tables are combined based on the specified condition.**

In conditional join, the **join condition can include <, >, <=, >=, ≠ operators in addition to the '=' operator.**

Example: Suppose two tables A and B

Table A

R	S
10	5
7	20

Table B

T	U
10	12

T	U
17	6

$$A \bowtie_{S<T} B$$

Output

R	S	T	U
10	5	10	12

Explanation: This query joins the table A, B and projects attributes R, S, T, U were the condition S < T is satisfied.

(b) Equi Join

Equi Join is a type of inner join **where the join condition uses the equality operator ('=') between columns.**

Example: Suppose there are two tables Table A and Table C

Table A

Column A	Column B
a	a
a	b

Table C

Column A	Column B
a	*a*
a	*c*

$A \bowtie_{A.Column\ B = C.Column\ B} (C)$

Output

Column A	Column B
a	a

Explanation: The data value "a" is available in both tables Hence we write that "a" is the table in the given output.

(c) Natural Join

Natural join is a type of inner join in **which we do not need any comparison operators.** In natural join, columns should have the same name and domain. There should be **at least one common attribute between the two tables.**
Example: Suppose there are two tables Table A and Table B

Table A

Number	Square
2	4
3	9

Table B

Number	Cube
2	8
3	27

$A \bowtie B$

Output

Number	Square	Cube
2	4	8
3	9	27

Explanation - Column Number is available in both tables Hence we write the "Number column once " after combining both tables.

2. Outer Join

Outer join is a type of join that **retrieves matching as well as non-matching records from related tables**. There are three types of outer join

- Left outer join
- Right outer join
- Full outer join

(a) **Left Outer Join**

It is also called left join. This type of outer join **retrieves all records from the left table and retrieves matching records from the right table.**
Example: Suppose there are two tables Table A and Table B
Table A

Number	Square
2	4
3	9
4	16

Table B

Number	Cube
2	8
3	27
5	125

$A \bowtie B$
Output

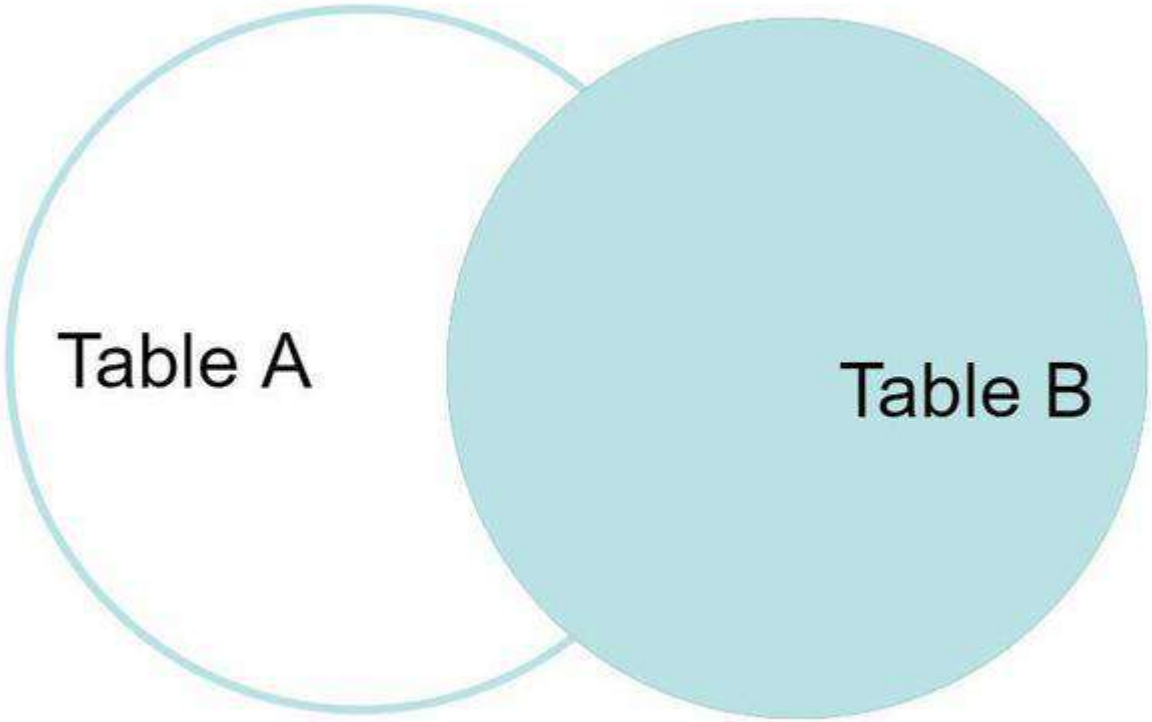
Number	Square	Cube
2	4	8
3	9	27

Number	Square	Cube
4	16	NULL

Explanation: Since we know in the left outer join we take all the columns from the left table (Here Table A) In the table A we can see that there is no Cube value for number 4. so we mark this as NULL.

(b) **Right Outer Join**

It is also called a right join. This type of outer **join retrieves all records from the right table and retrieves matching records from the left table**. And for the record which doesn't lies in Left table will be **marked as NULL** in result Set.



Right Outer Join

Example: Suppose there are two tables Table A and Table B
$A \rtimes B$
Output:

Number	Square	Cube
2	4	8

Number	Square	Cube
3	9	27
5	NULL	125

Explanation: Since we know in the right outer join we take all the columns from the right table (Here Table B) In table A we can see that there is no square value for number 5. So we mark this as NULL.

(c) Full Outer Join

FULL JOIN creates the **result set by combining the results of both LEFT JOIN and RIGHT JOIN.** The result set will contain all the rows from both tables. **For the rows for which there is no matching, the result set will contain *NULL* values.**

Example: Table A and Table B are the same as in the left outer join

$A \bowtie B$

Output:

Number	Square	Cube
2	4	8
3	9	27
4	16	NULL
5	NULL	125

Explanation: Since we know in full outer join **we take all the columns** from both tables (Here Table A and Table B) In the table A and Table B we can see that there is no Cube value for number 4 and No Square value for 5 so we mark this as NULL.

SQL Joins (Inner, Left, Right and Full Join)

- SQL joins are **fundamental tools for combining data from multiple tables in relational databases.**

Types of SQL Joins

1. SQL INNER JOIN

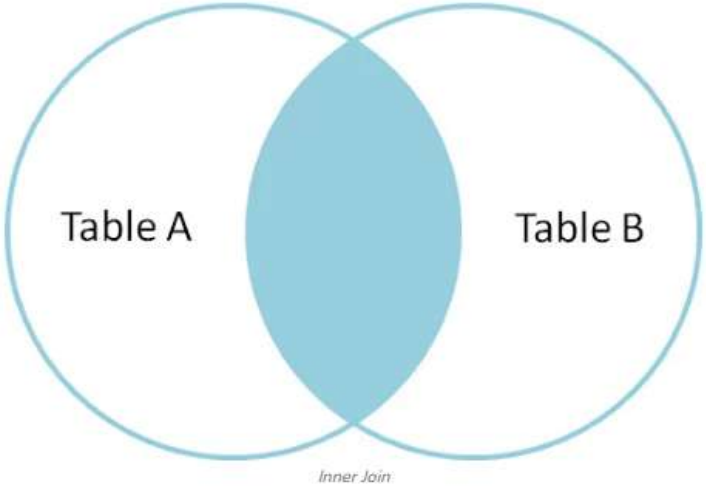
The INNER JOIN keyword **selects all rows from both the tables as long as the condition is satisfied.**

This keyword will create the result set by combining all rows from both the tables where the condition satisfies i.e value of the common field will be the same.

Syntax:

```
SELECT table1.column1,table1.column2,table2.column1,.... FROM table1  INNER JOIN
 table2 ON  table1.matching_column = table2.matching_column;
```

Note: We can also write JOIN instead of INNER JOIN. JOIN is same as INNER JOIN.



Inner Join

Inner Join

Example of INNER JOIN

Consider the **two tables, Student and StudentCourse**, which share a common column ROLL_NO. Using SQL JOINS, we can combine data from these tables based on their relationship, allowing us to retrieve meaningful information like student details along with their enrolled courses.

1. Student Table:

ROLL_NO	NAME	ADDRESS	PHONE	AGE
1	HARSH	DELHI	XXXXXXXXXX	18
2	PRATIK	BIHAR	XXXXXXXXXX	19
3	RIYANKA	SILIGURI	XXXXXXXXXX	20
4	DEEP	RAMNAGAR	XXXXXXXXXX	18

2. StudentCourse Table:

COURSE_ID	ROLL_NO
1	1
2	2
2	3
3	4

This **query will show the names and age of students enrolled in different courses.**

Query:

```
SELECT StudentCourse.COURSE_ID, Student.NAME, Student.AGE FROM Student
INNER JOIN StudentCourse
ON Student.ROLL_NO = StudentCourse.ROLL_NO;
```

Output:

COURSE_ID	NAME	AGE
1	HARSH	18

COURSE_ID	NAME	AGE
2	PRATIK	19
2	RIYANKA	20
3	DEEP	18

2. SQL LEFT JOIN

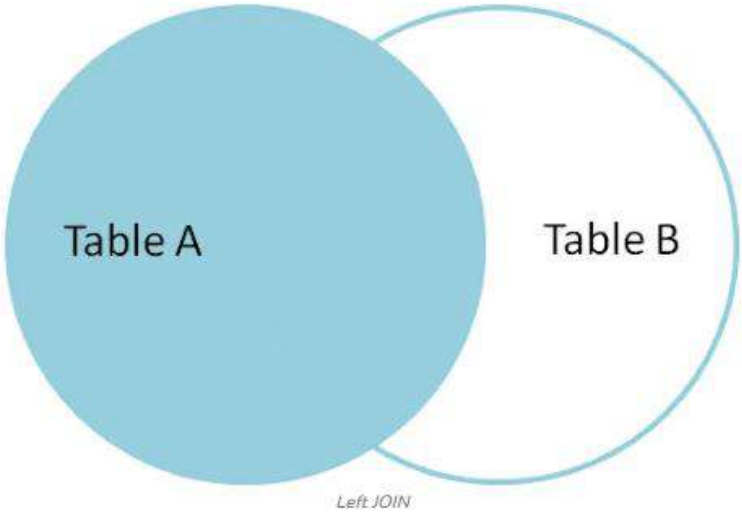
A **LEFT JOIN** returns **all rows from the left table, along with matching rows from the right table.**

If there is no match, NULL values are returned for columns from the right table. LEFT JOIN is also known as LEFT OUTER JOIN.

Syntax

```
SELECT table1.column1,table1.column2,table2.column1,....
FROM table1
LEFT JOIN table2
ON table1.matching_column = table2.matching_column;
```

***Note**: We can also use LEFT OUTER JOIN instead of LEFT JOIN, both are the same.*



Left JOIN

LEFT JOIN Example

In this example, the LEFT JOIN retrieves all rows from the Student table and the matching rows from the StudentCourse table based on the `ROLL_NO` column.

Query:

```
SELECT Student.NAME,StudentCourse.COURSE_ID
```

```
FROM Student
LEFT JOIN StudentCourse
ON StudentCourse.ROLL_NO = Student.ROLL_NO;
```

Output:

NAME	COURSE_ID
HARSH	1
PRATIK	2
RIYANKA	2
DEEP	3

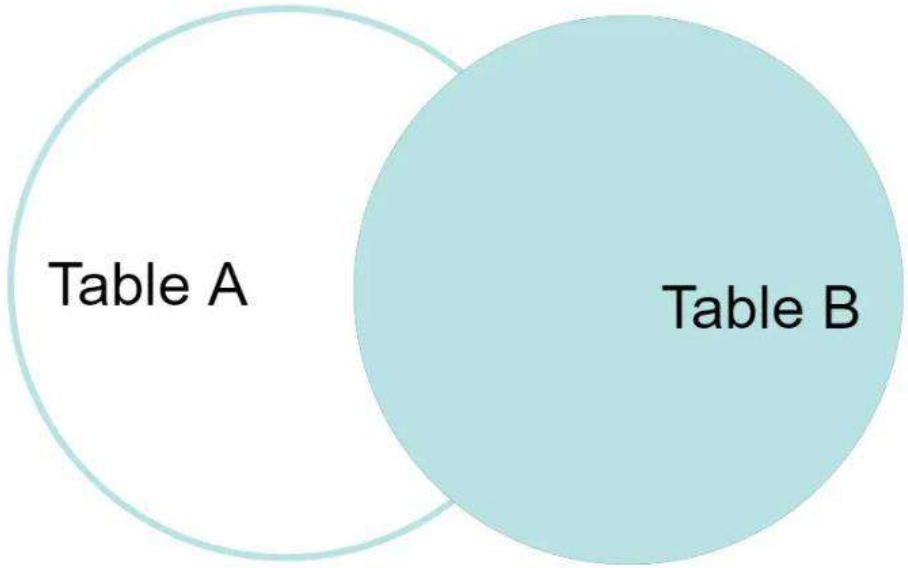
3. SQL RIGHT JOIN

RIGHT JOIN returns all the rows of the table on the right side of the join and matching rows for the table on the left side of the join. It is very similar to LEFT JOIN for the rows for which there is no matching row on the left side, the result-set will contain null. RIGHT JOIN is also known as RIGHT OUTER JOIN.

Syntax

```
SELECT table1.column1,table1.column2,table2.column1,....
FROM table1
RIGHT JOIN table2
ON table1.matching_column = table2.matching_column;
```

***Note**: We can also use RIGHT OUTER JOIN instead of RIGHT JOIN, both are the same.*



RIGHT JOIN Example

In this example, the RIGHT JOIN retrieves all rows from the StudentCourse table and the matching rows from the Student table based on the `ROLL_NO` column.

Query:

```
SELECT Student.NAME,StudentCourse.COURSE_ID
FROM Student
RIGHT JOIN StudentCourse
ON StudentCourse.ROLL_NO = Student.ROLL_NO;
```

Output:

NAME	COURSE_ID
HARSH	1
PRATIK	2
RIYANKA	2
DEEP	3
NULL	4

4. SQL FULL JOIN

FULL JOIN creates the result-set by combining results of both LEFT JOIN and RIGHT JOIN. The result-set will contain all the rows from both tables. For the rows for which there is no matching, the result-set will contain NULL values.

Syntax

```
SELECT table1.column1,table1.column2,table2.column1,....
FROM table1
FULL JOIN table2
ON table1.matching_column = table2.matching_column;
```

FULL JOIN Example

This example demonstrates the use of a FULL JOIN, which combines the results of both LEFT JOIN and RIGHT JOIN. The query retrieves all rows from the Student and StudentCourse tables. If a record in one table does not have a matching record in the other table, the result set will include that record with `NULL` values for the missing fields

Query:

```
SELECT Student.NAME,StudentCourse.COURSE_ID
FROM Student
FULL JOIN StudentCourse
ON StudentCourse.ROLL_NO = Student.ROLL_NO;
```

Output :

NAME	COURSE_ID
HARSH	1
PRATIK	2
RIYANKA	2
DEEP	3

5. SQL Natural Join

A **Natural Join is a type of INNER JOIN that automatically joins two tables based on one common column with the same name and data type**.

It returns **only the rows where the values in the common columns match.**

- It returns rows where the values in these common columns are the same in both tables.

- **Common columns appear only once in the result**, even if they exist in both tables.
- Unlike a CROSS JOIN, which creates all possible combinations of rows, a Natural Join only includes rows with matching values

Example:

Look at the two tables below: Employee and Department

Employee

Emp_id	Emp_name	Dept_id
1	Ram	10
2	Jon	30
3	Bob	50

Department

Dept_id	Dept_name
10	IT
30	HR
40	TIS

Find all Employees and their respective departments.

```
(Employee) ? (Department)
```

Output:

Emp_id	Emp_name	Dept_id	Dept_id	Dept_name
1	Ram	10	10	IT

Emp_id	Emp_name	Dept_id	Dept_id	Dept_name
2	Jon	30	30	HR

Difference between JOIN and UNION in SQL

- **JOIN in SQL is used to combine data from many tables based on a matched condition between them**. The data combined using the JOIN statement results in new columns. Consider the two tables:

Boys

Roll No.	Name	Age
1	Ram	16
8	Jayant	17
16	Ritik	15
27	Mayank	19
30	Prakhar	17
45	Sanjay	16

Girls

Roll No.	Name	Address
9	Naina	Delhi
6	Rinki	Mumbai
7	Reema	Jaipur
30	Rimi	Bhopal
45	Seema	Goa
16	Mona	Delhi

Example:

```
sql> SELECT Boys.Name, Boys.Age, Girls.Address,
     FROM Boys INNER JOIN Girls
     ON Boys.Rollno = Girls.Rollno;
```

The resultant table is:

Name	Age	Address
Ritik	15	Delhi
Prakhar	17	Bhopal
Sanjay	16	Goa

UNION in SQL is used to combine the result set of two or more SELECT statements. The data combined using the UNION statement is into results into new distinct rows.

Example:

```
sql> SELECT Name FROM Boys
     WHERE Rollno < 16
     UNION
     SELECT Name FROM Girls
     WHERE Rollno > 9
```

Output:

Name
Ram
Jayant
Rimi
Seema

Name
Mona

Difference between JOIN and UNION in SQL

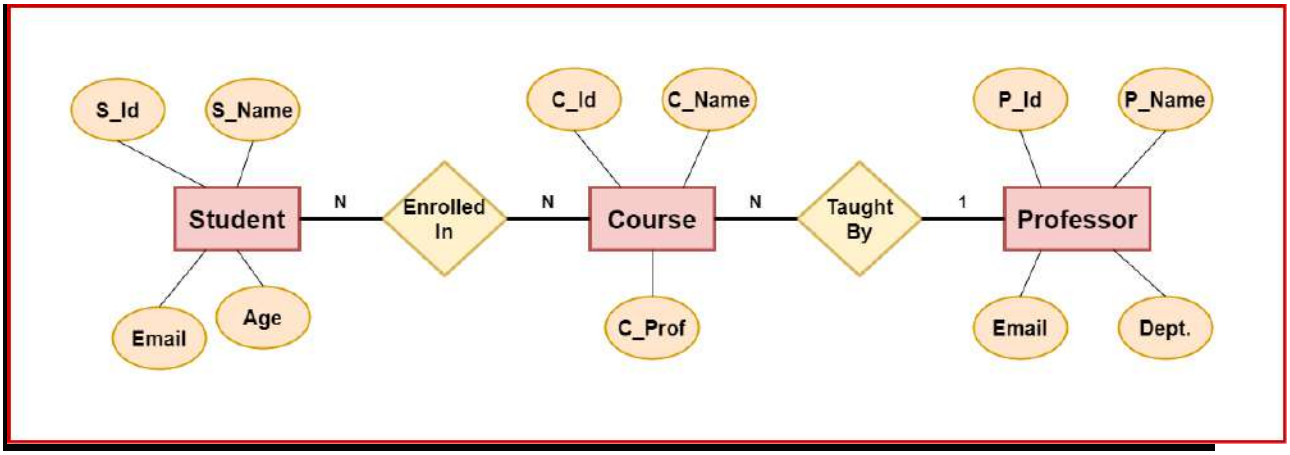
JOIN	UNION
JOIN combines data from many tables based on a matched condition between them	SQL combines the result set of two or more SELECT statements.
It combines data into new **columns**.	It combines data into **new rows**
The number of columns selected from each table may not be the same.	The number of columns selected from each table should be the same.
Datatypes of corresponding columns selected from each table can be different.	The data types of corresponding columns selected from each table should be the same.
It may not return distinct columns.	It returns distinct rows.

Convert ER Model to Relational Model

The ER model is the **visual design of the database system**. **For efficient management** and **storage of data**, we require the **conversion of the ER model to a relational model** that facilitates the storing and retrieval of data due to the format of storing in tables.

Converting an Entity-Relationship (ER) model to a relational model **involves transforming the entities, relationships, and attributes depicted in the ER diagram into tables, columns, and keys** in a relational database. The

following are the steps that need to be followed, explained with the **example** of a typical **college database system**:



1. Identify Entities and Create Tables

Each entity in the ER model becomes a table in the relational model. The **attributes of the entity become the columns** of the table.

In our example, we have the following entities and respective attributes:

- **Student:** S_Id, S_Name, Email, Age
- **Course:** C_Id, C_Name, C_Prof
- **Professor:** P_Id, P_Name, Email, Dept.

We can separate the **S_Name** and **P_Name** attributes into **S_FirstName**, **S_LastName** and similarly for the Professor table to create atomic attributes.

2. ***Key Selection***

Choose the primary key for each table. For some, it can be in the form of a composite key (Weak entity).

In our example, we can have **S_Id**, **C_Id**, and **P_Id** as **primary keys** for the Student, Course, and Professor tables respectively, since they would satisfy the unique and non-null property constraints.

3. ***Identify Relationships and Create Foreign Keys***

If **entities have relationships, break them down and reduce the tables** if possible. Relationships in the ER model are represented by foreign keys in the relational model. Below are the handling mechanisms for different types of relationships:

- **1-1 Relationship:** 2 tables; the primary key can lie on any side.
- **1-Many Relationship:** 2 tables; the primary key can lie on the many side.
- **Many-1 Relationship:** 2 tables; the primary key can lie on the many side.
- **Many-Many Relationship:** 3 tables; the primary key lies in the relation table having primary keys from both tables acting as foreign keys.

In our example, we have the following relationships:

- **Enrollment (between Student and Course):** A student can enroll in multiple courses, and a **course can have multiple students (many-to-many relationship)**. So we would **create a separate** enrollment relationship **table**.
- **Enrollment Table:** S_Id (FK), C_Id (FK), E_Date
  Primary Key: Composite key consisting of S_Id and C_Id.
- **Teaching (between Instructor and Course):** A professor can teach multiple courses, but a course is typically taught by one professor (many-to-one relationship). Hence, here we would have the primary key on the Course Table.
- **Course Table (updated):** C_Id, C_Name, C_Prof (FK)

Hence the final relational model would look like:

1. **Student:** `S_Id` (PK), `S_Name`, `Email`, `Age`
2. **Course:** `C_Id` (PK), `C_Name`, `C_Prof` (FK)
3. **Professor:** `P_Id` (PK), `P_Name`, `Email`, `Dept.`
4. **Enrollment:** `S_Id` (FK), `C_Id` (FK), `E_Date`; Primary Key (`S_Id`, `C_Id`)

P.K.

S_Id	S_Name	Email	Age
s1	Ram	abc	18
s2	Shyam	xyz	19
s3	Rani	uio	18

STUDENT

P.K. F.K.

C_Id	C_Name	C_Prof
c1	OS	p1
c2	DBMS	p2
c3	CN	p3

COURSE

P.K.

P_Id	P_Name	Email	Dept.
p1	Rahul	jhg	CSE
p2	Ritu	asd	IT
p3	Anil	yui	ECE

PROFESSOR

P.K.

F.K. F.K.

S_Id	C_Id	E_Date
s1	c2	July
s1	c1	Feb
s2	c2	Jan

ENROLLMENT