DBMS
UNIT -2

In DBMS (Database Management System), **keys** are used to uniquely identify records in a table or establish relationships between tables.
Here are some basic types of keys:

1. **Primary Key**: A unique identifier for each record in a table. It cannot be null.
2. **Foreign Key**: A key used to link two tables. It refers to the primary key in another table.
3. **Candidate Key**: A set of attributes that can uniquely identify a record in a table. One of these is selected as the primary key.
4. **Alternate Key**: A candidate key that is not chosen as the primary key.
5. **Composite Key**: A key that consists of two or more columns to uniquely identify a record.
6. **Super Key** is a set of one or more attributes (columns) in a table that can uniquely identify each record (row) in that table. It can include additional attributes beyond what is strictly necessary for uniqueness.

1. Primary Key

A **Primary Key** uniquely identifies each record in a table. It cannot have duplicate or NULL values.

Example: Consider a table called Students:

StudentID	Name	Age	Gender
1	Alice	20	Female
2	Bob	22	Male
3	Charlie	21	Male

Here, StudentID can be the **Primary Key** because it uniquely identifies each student.

2. Foreign Key

A **Foreign Key** is a field (or a set of fields) in one table that uniquely identifies a row of another table.

Example: Consider two tables, Orders and Customers:

Customers Table:

CustomerID	Name	Email
1	Alice	alice@mail.com
2	Bob	bob@mail.com

Orders Table:

OrderID	CustomerID	Product	Quantity
101	1	Laptop	2
102	2	Smartphone	1

Here, CustomerID in the Orders table is a **Foreign Key** that refers to the CustomerID in the Customers table.

3. Candidate Key

A **Candidate Key** is a set of one or more columns that can uniquely identify each record in a table. The table may have more than one candidate key.
Example: Consider a Users table:

UserID	Email	Username
1	alice@mail.com	alice_123
2	bob@mail.com	bob_456

Both UserID and Email are candidate keys, as they can each uniquely identify a record.

4. Alternate Key

An **Alternate Key** is a candidate key that was not chosen as the primary key.
Example: In the Users table above, if we choose UserID as the **Primary Key**, then Email becomes an **Alternate Key**.

5. Composite Key

A **Composite Key** is made up of two or more columns that together uniquely identify a record.
Example: Consider a Course_Enrollments table:

StudentID	CourseID	EnrollmentDate
1	101	2025-01-10
2	102	2025-01-09

In this case, both StudentID and CourseID together form a **Composite Key** because neither field alone is sufficient to uniquely identify a record.

These keys help structure and relate the data efficiently in relational databases.

6.Super Key:

Consider a Students table:

StudentID	Name	Email	Age
1	Alice	alice@mail.com	20
2	Bob	bob@mail.com	22
3	Charlie	charlie@mail.com	21

In this table:

- StudentID is enough to uniquely identify each student (Primary Key).
- However, the following are also **Super Keys** because they can also uniquely identify each record, even though they include extra attributes:

- {StudentID, Name}
- {StudentID, Email}
- {Name, Email, Age}

In these cases, the Super Keys contain more attributes than necessary for uniqueness, but they still ensure that each row is unique.

Relational Algebra

Relational Algebra is a mathematical system used to query and manipulate data in a relational database. It is a collection of operations that work on one or more relations (tables) to produce a result. The operations in relational algebra are used to retrieve, insert, update, and delete data from a database.

Basic Operations of Relational Algebra:

1. **Selection (σ)**:
   - This operation is used to retrieve rows from a relation that satisfy a given condition.
   - **Syntax**: σ(condition)(Relation)
   - **Example**: Retrieve students who are 20 years old from the Students table:

     σ(Age = 20)(Students)

2. **Projection (π)**:
   - This operation is used to retrieve specific columns (attributes) from a relation.
   - **Syntax**: π(column1, column2, ...)(Relation)
   - **Example**: Retrieve only the Name and Age of all students:

     π(Name, Age)(Students)

3. **Union (∪)**:
   - This operation combines the results of two relations and removes duplicates. Both relations must have the same attributes (columns).
   - **Syntax**: Relation1 ∪ Relation2
   - **Example**: Combine two lists of students from different departments:

     Students_CS ∪ Students_Maths

4. **Set Difference (−)**:
   - This operation returns rows that are in the first relation but not in the second.
   - **Syntax**: Relation1 − Relation2
   - **Example**: Find students who are enrolled in Maths but not in CS:

     Students_Maths − Students_CS

5. **Cartesian Product (×)**:
    - This operation combines every row of the first relation with every row of the second relation.
    - **Syntax**: Relation1 × Relation2
    - **Example**: Combine the Students and Courses tables to show all possible student-course combinations:

      Students × Courses

6. **Rename (ρ)**:
    - This operation is used to rename the attributes (columns) of a relation.
    - **Syntax**: ρ(new_name)(Relation)
    - **Example**: Rename the StudentID column to ID in the Students table:

      ρ(ID/StudentID)(Students)

Advanced Operations:

1. **Join (⋈)**:
    - This operation combines related tuples from two relations based on a condition (usually equality on a common attribute).
    - **Syntax**: Relation1 ⋈ Condition Relation2
    - **Example**: Join Students and Enrollments based on StudentID:

      Students ⋈ Students.StudentID = Enrollments.StudentID Enrollments

2. **Intersection (∩)**:
    - This operation returns the common rows between two relations.
    - **Syntax**: Relation1 ∩ Relation2
    - **Example**: Find students who are in both CS and Maths:

      Students_CS ∩ Students_Maths

Example of Relational Algebra Query:

Consider two tables: Students and Courses.

Students Table:

StudentID	Name	Age	Department
1	Alice	20	CS
2	Bob	22	Maths
3	Charlie	21	CS

Courses Table:

CourseID	CourseName	Department
101	Database	CS
102	AI	CS
103	Calculus	Maths

To find students who are in the CS department and are enrolled in the AI course, we can use relational algebra:

1. **Select students in CS**:

   σ(Department = 'CS')(Students)

2. **Select courses named AI**:

   σ(CourseName = 'AI')(Courses)

3. **Join students and courses on department**:

   σ(Department = 'CS')(Students) ⋈ σ(CourseName = 'AI')(Courses)

Relational algebra provides a theoretical foundation for SQL and helps in understanding and optimizing database queries.

Problem on Relational Algebra

Q.) Let R =(ABC) and S=(DEF) let r(R) and s(S) both relations on schema R and S. Give an expression in the Tuple relational calculus that is equivalent to each of the following.

i) ΠA® ii) σp=19® iii) rXs iv) ΠA,F,(σC=D(rXs)).

Solution:

To provide expressions in **Tuple Relational Calculus (TRC)** for the given queries, let's first define the schemas:

- **R = (A, B, C)** (relation r has attributes A,B,C)
- **S = (D, E, F)** (relation s has attributes D,E,F)

Now, we'll express the following queries in Tuple Relational Calculus:

i) ΠA1(r)

This refers to **projection** of attribute A (from relation r).

In Tuple Relational Calculus (TRC), the projection of a specific attribute is represented by selecting that attribute from the tuples of the relation.

TRC Expression:

{ t.A | t ∈ r }

SQL Query : SELECT A FROM r;

This means that we want to return all values of attribute A from relation r, where t is a tuple from relation r.

ii) σp=19(r)

This refers to **selection** where some predicate p=19 holds in relation r.

Assume p refers to the attribute A, and the condition is that A=19.

TRC Expression:

$$\{ t \mid t \in r \land t.A = 19 \}$$

SQL Query: SELECT * FROM r WHERE A = 19;

This means that we select all tuples t from relation r where the value of attribute A is equal to 19.

iii) r × s (Cartesian Product of r and s)

The **Cartesian product** of relations r and s means pairing each tuple from r with each tuple from s.

TRC Expression:

$$\{ (t, u) \mid t \in r \land u \in s \}$$

SQL Query: SELECT * FROM r CROSS JOIN s;

This means that for each tuple t from relation r and each tuple u from relation s, we return the pair (t,u).

iv) ΠA,F(σC=D(r × s))

This refers to **projection** on attributes A and F, after performing a **selection** on the Cartesian product of r and s where C=D.

First, we perform a **selection** where the attribute C from rrr is equal to attribute D from s. After that, we project only the attributes A (from r) and F (from s).

TRC Expression:

$$\{ (t.A, u.F) \mid t \in r \land u \in s \land t.C = u.D \}$$

SQL Query: SELECT r.A, s.F FROM r JOIN s ON r.C = s.D;

This means that we return the pair (t.A,u.F) where t is a tuple from r, u is a tuple from s, and the condition t.C=u.D holds.

Domain Relational Calculus: DRC is another formal query language used to describe the queries on a relational database. Unlike **Tuple Relational Calculus (TRC)**, which works

with tuples (rows) from relations, **Domain Relational Calculus** works with the **values of attributes** in a relation (i.e., the domain of the attributes).

In DRC, a query specifies the **values** of the attributes (domains) for which we want to find tuples. A query in DRC is expressed as a formula that specifies conditions on the values of attributes, and the result consists of those values that satisfy the conditions.

VIEW:

a **view** is a virtual table that is based on the result of a query. It does not store data physically, but instead, it displays data derived from one or more tables. Views are used to simplify complex queries, present a specific subset of data, or protect data by restricting access to certain parts of a database.

The reasons for using views are

- Security is increased - sensitive information can be excluded from a view.
- Views can represent a subset of the data contained in a table.
- Views can join and simplify multiple tables into a single virtual table.
- Views take very little space to store; the database contains only the definition of a view, not a copy of all the data it presents.
- Different views can be created on the same base table for different categories of users.

Creating Views syntax

```
CREATE VIEW view_name AS
SELECT column_list
FROMtable_name [ WHERE condition] ;
```

Examples: We can create a view to display only the `Name` and `Department`:

1. **Creating View:**

```
CREATE VIEW student_view AS SELECT Name, Department FROM Students;
```

2. **Querying a View:**

```
SELECT * FROM student_view;
```

3. **Updating Data through a View:**

```
UPDATE student_view SET Department = 'Computer Science' WHERE Name = 'Alice';
```

Advantages of Using Views:

1. **Data Abstraction**: Views provide an abstraction layer, allowing users to interact with data without worrying about underlying complexities.
2. **Consistency**: By using views, you can ensure that certain data is always presented in a consistent format.
3. **Security**: Views can limit access to sensitive data by showing only a subset of the data in a table.

JOINS :

In a **Database Management System (DBMS)**, a **join** is an operation that combines data from two or more tables based on a related column between them. Joins are used to retrieve meaningful information from multiple tables by combining their rows according to a specific condition.

Types of Joins:

1. **Inner Join**
2. **Left (Outer) Join**
3. **Right (Outer) Join**
4. **Full (Outer) Join**
5. **Cross Join**
6. **Self Join**

Each of these joins serves a different purpose and has its own behaviour when it comes to combining rows from the involved tables.

1. Inner Join

Inner Join returns only the rows where there is a match in both tables.

Syntax:

```
SELECT columns
FROM table1
INNER JOIN table2
ON table1.column = table2.column;
```

Example:

Consider the following two tables:

Employees:

EmployeeID	Name	DepartmentID
1	Alice	101
2	Bob	102
3	Carol	103

Departments:

DepartmentID	DepartmentName
101	HR
102	IT
104	Marketing

To get a list of employees and their department names, we can perform an **inner join** on DepartmentID.

```
SELECT Employees.Name, Departments.DepartmentName
FROM Employees
INNER JOIN Departments
```

ON Employees.DepartmentID = Departments.DepartmentID;

Result:

Name	DepartmentName
Alice	HR
Bob	IT

Explanation:

- The query returns only those rows where there is a matching DepartmentID in both the **Employees** and **Departments** tables.
- Carol is excluded because her DepartmentID (103) does not exist in the **Departments** table.

---

2. Left (Outer) Join

A **Left Join** (or **Left Outer Join**) returns all the rows from the **left table** (the first table), and the matched rows from the **right table**. If no match is found, NULL values are returned for columns from the right table.

Syntax:

SELECT columns
FROM table1
LEFT JOIN table2
ON table1.column = table2.column;

Example:

Using the same **Employees** and **Departments** tables:

SELECT Employees.Name, Departments.DepartmentName
FROM Employees
LEFT JOIN Departments
ON Employees.DepartmentID = Departments.DepartmentID;

Result:

Name	DepartmentName
Alice	HR
Bob	IT
Carol	NULL

Explanation:

- The query returns all employees, including Carol, whose DepartmentID does not have a match in the **Departments** table.
- For Carol, the **DepartmentName** is NULL because there is no matching row in the **Departments** table.

---

3. Right (Outer) Join:

A **Right Join** (or **Right Outer Join**) returns all the rows from the **right table** (the second table), and the matched rows from the **left table**. If no match is found, NULL values are returned for columns from the left table.

Syntax:

```
SELECT columns
FROM table1
RIGHT JOIN table2
ON table1.column = table2.column;
```

Example:

Again, using the same tables:

```
SELECT Employees.Name, Departments.DepartmentName
FROM Employees
RIGHT JOIN Departments
ON Employees.DepartmentID = Departments.DepartmentID;
```

Result:

Name	DepartmentName
Alice	HR
Bob	IT
NULL	Marketing

Explanation:

- The query returns all departments, including **Marketing**, which does not have any matching employees.
- For the **Marketing** department, there is no matching employee, so the **Name** is NULL.

---

4. Full (Outer) Join

A **Full Join** (or **Full Outer Join**) returns all rows from both the left and right tables. If there is no match, NULL values are returned for columns from the table that lacks the matching row.

Syntax:

```
SELECT columns
FROM table1
FULL JOIN table2
ON table1.column = table2.column;
```

Example:

Using the same tables:

```
SELECT Employees.Name, Departments.DepartmentName
FROM Employees
FULL JOIN Departments
ON Employees.DepartmentID = Departments.DepartmentID;
```

Result:

Name	DepartmentName
Alice	HR
Bob	IT
Carol	NULL
NULL	Marketing

Explanation:

- The query returns all employees and all departments.
- Carol has no matching department, so her DepartmentName is NULL.
- The **Marketing** department has no matching employees, so Name is NULL.

---

5. Cross Join

A **Cross Join** returns the Cartesian product of both tables, meaning it combines every row from the first table with every row from the second table. This can result in a very large number of rows.

Syntax:

```
SELECT columns
FROM table1
CROSS JOIN table2;
```

Example:

For the **Employees** and **Departments** tables:

```
SELECT Employees.Name, Departments.DepartmentName
FROM Employees
CROSS JOIN Departments;
```

Result:

Name	DepartmentName
Alice	HR
Alice	IT
Alice	Marketing
Bob	HR
Bob	IT
Bob	Marketing
Carol	HR
Carol	IT
Carol	Marketing

Explanation:

- The query returns every possible combination of rows between the **Employees** and **Departments** tables (i.e., each employee is paired with every department).

---

6. Self Join

A **Self Join** is a join where a table is joined with itself. This is useful for finding relationships within the same table, like finding pairs of employees in the same department.

Syntax:

```
SELECT columns
FROM table1 t1
```

JOIN table1 t2
ON t1.column = t2.column;

Example:

Assume the **Employees** table also has a **ManagerID** column to indicate which employee manages another employee.

Employees:

EmployeeID	Name	DepartmentID	ManagerID
1	Alice	101	NULL
2	Bob	101	1
3	Carol	102	2

To find pairs of employees and their managers:

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

Result:

Employee	Manager
Bob	Alice
Carol	Bob

Explanation:

- The query performs a self join on the **Employees** table to find each employee and their manager by matching the **ManagerID** in one row to the **EmployeeID** in another row.