**UNIT II**

**SYLLABUS**

**Relation data model:** Relational Data Model - keys, referential integrity and foreign keys, Relational Algebra - SQL fundamentals- Introduction, data definition in SQL, table, key and foreign key definitions, update behaviours- Views, Triggers, Joins, Constraints, Stored Procedure-Intermediate SQL- Advanced SQL features -Embedded SQL- Dynamic SQL.

## Relation Data Model

‘

Relational data model is the primary data model, which is used widely around the world for data storage and processing. This model is simple and it has all the properties and capabilities required to process data with storage efficiency.

## Concepts

**Tables** − In relational data model, relations are saved in the format of Tables. This format stores the relation among entities. A table has rows and columns, where rows represents records and columns represent the attributes.

**Tuple** − A single row of a table, which contains a single record for that relation is called a tuple.

**Relation instance** − A finite set of tuples in the relational database system represents relation instance. Relation instances do not have duplicate tuples.

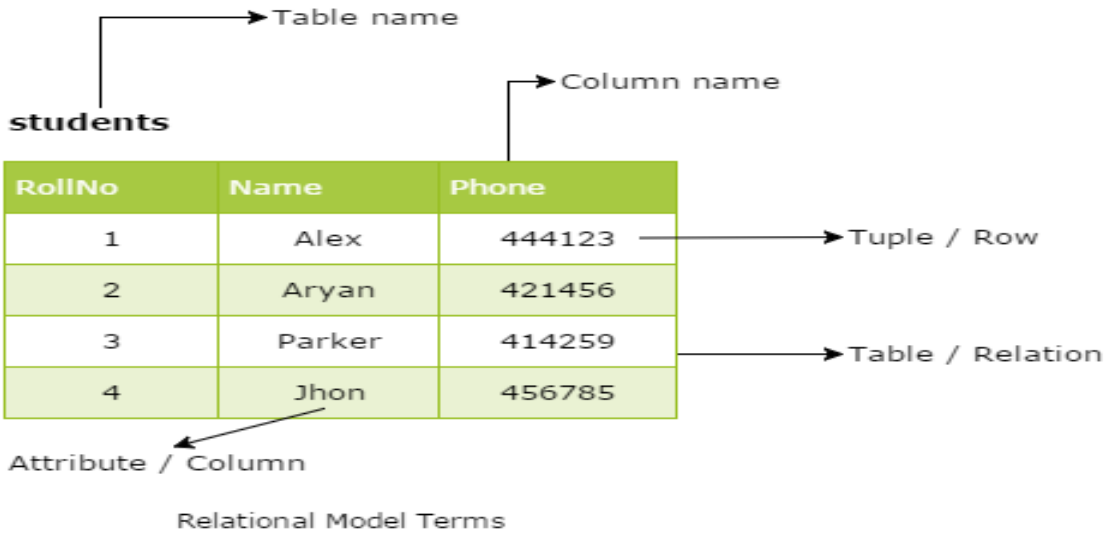
**Relation schema** − A relation schema describes the relation name (table name), attributes, and their names.

**Relation key** − Each row has one or more attributes, known as relation key, which can identify the row in the relation (table) uniquely.

**Attribute domain** − Every attribute has some pre-defined value scope, known as attribute domain.

**UNIT II**

**Some Common Relational Model Terms**




Relational Model Terms

- Relation: A relation is a table with columns and rows.
- Attribute: An attribute is a named column of a relation.
- Domain: A domain is the set of allowable values for one or more attributes.
- Tuple: A tuple is a row of a relation.

## Relational Constraints

**What are Database Constraints in DBMS ??**

Database constraints are restrictions on the contents of the database or on database operations. It is a condition specified on a database schema that restricts the data to be inserted in an instance of the database.

Every relation has some conditions that must hold for it to be a valid relation. These conditions are called **Relational Integrity Constraints**.

**UNIT II**

**Need of Constraints :**

**Constraints in the database provide a way to guarantee that :**

- the values of individual columns are valid.
- in a table, rows have a valid primary key or unique key values.
- in a dependent table, rows have valid foreign key values that reference rows in a parent table.

**Types of constraints in DBMS:**

- **Domain Constraints**
- **Tuple Uniqueness Constraints**
- **Key Constraints**
- **Single Value Constraints**
- **Integrity Rule 1 (Entity Integrity Rule or Constraint)**
- **Integrity Rule 2 (Referential Integrity Rule or Constraint)**
- **General Constraints**

**Domain Constraints –**

Domain Constraints specifies that what set of values an attribute can take. Value of each attribute X must be an atomic value from the domain of X. The data type associated with domains include integer, character, string, date, time, currency etc. An attribute value must be available in the corresponding domain. Consider the example below –

| SID | Name | Class (semester) | Age |
|---|---|---|---|
| 8001 | Ankit | 1st | 19 |
| 8002 | Srishti | 1st | 18 |
| 8003 | Somvir | 4th | 22 |
| 8004 | Sourabh | 6th | A |

Not Allowed. Because Age is an Integer Attribute.

UNIT II

**Tuple Uniqueness Constraints –**

A relation is defined as a set of tuples. All tuples or all rows in a relation must be unique or distinct. Suppose if in a relation, tuple uniqueness constraint is applied, then all the rows of that table must be unique i.e. it does not contain the duplicate values. For example,

| SID | Name | Class (semester) | Age |
|---|---|---|---|
| 8001 | Ankit | 1st | 19 |
| 8002 | Srishti | 2nd | 18 |
| 8003 | Somvir | 4th | 22 |
| 8004 | Sourabh | 6th | 19 |

Not Allowed. Because all rows must be unique.

**Key Constraints –**

Keys are attributes or sets of attributes that uniquely identify an entity within its entity set. An Entity set E can have multiple keys out of which one key will be designated as the primary key. Primary Key must have unique and not null values in the relational table. In an subclass hierarchy, only the root entity set has a key or primary key and that primary key must serve as the key for all entities in the hierarchy.

**Types of keys in DBMS**

1. **Primary Key** – A primary is a column or set of columns in a table that uniquely identifies tuples (rows) in that table.

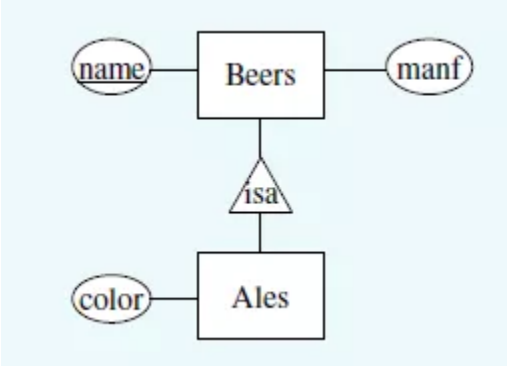
Example of Key Constraints in a simple relational table –

**UNIT II**

| SID | Name | Class (semester) | Age |
|---|---|---|---|
| 8001 | Ankit | 1st | 19 |
| 8002 | Srishti | 1st | 18 |
| 8003 | Somvir | 4th | 22 |
| 8004 | Sourabh | 6th | 45 |
| 8002 | Tony | 5th | 23 |

**Not allowed as Primary Key Values must be unique**

Example of Key Constraints in an subclass hierarchy –




2. **Super Key** – A super key is a set of one of more columns (attributes) to uniquely identify rows in a table. Often people get confused between super key and candidate key, so we will also discuss a little about candidate key here.

**How candidate key is different from super key?**

Answer is simple – Candidate keys are selected from the set of super keys, the only thing we take care while selecting candidate key is: It should not have any redundant attribute. That's the reason they are also termed as minimal super key.

Let's take an example to understand this: **Employee table**

**UNIT II**

| Emp_SSN | Emp_Number | Emp_Name |
|---|---|---|
| 123456789 | 226 | Steve |
| 999999321 | 227 | Ajeet |
| 888997212 | 228 | Chaitanya |
| 777778888 | 229 | Robert |

**Super keys**:

- {Emp_SSN}
- {Emp_Number}
- {Emp_SSN, Emp_Number}
- {Emp_SSN, Emp_Name}
- {Emp_SSN, Emp_Number, Emp_Name}
- {Emp_Number, Emp_Name}

All of the above sets are able to uniquely identify rows of the employee table.

3. **Candidate Key** – A super key with no redundant attribute is known as candidate key

A candidate key is a column, or set of columns, in a table that can uniquely identify any database record without referring to any other data. Each table may have one or more candidate keys, but one candidate key is unique, and it is called the primary key. This is usually the best among the candidate keys to use for identification.

When a key is composed of more than one column, it is known as a composite key.

A super key with no redundant attribute is known as candidate key. Candidate keys are selected from the set of super keys, the only thing we take care while selecting

candidate key is: It should not have any redundant attributes. That's the reason they are also termed as minimal super key.

**For example**:

| Emp_Id | Emp_Number | Emp_Name |
|---|---|---|
| E01 | 2264 | Steve |
| E22 | 2278 | Ajeet |
| E23 | 2288 | Chaitanya |
| E45 | 2290 | Robert |

There are two candidate keys in above table:

{Emp_Id}

{Emp_Number}

**Note**: A primary key is being selected from the group of candidate keys. That means we can either have Emp_Id or Emp_Number as primary key.

4.**Alternate Key** – Out of all candidate keys, only one gets selected as primary key, remaining keys are known as alternate or secondary keys.

**For example: Consider the below table**

| Emp_Id | Emp_Number | Emp_Name |
|---|---|---|
| E01 | 2264 | Steve |
| E22 | 2278 | Ajeet |

| | | |
|---|---|---|
| E23 | 2288 | Chaitanya |
| E45 | 2290 | Robert |

There are two candidate keys in above table:

{Emp_Id}

{Emp_Number}

Since we have selected Emp_Id as primary key, the remaining key Emp_Number would be called alternative or secondary key.

5. **Composite Key** – A key that consists of more than one attribute to uniquely identify rows (also known as records & tuples) in a table is called composite key.

**Example: Table – Sales**

| cust_Id | order_Id | product_code | product_count |
|---|---|---|---|
| C01 | O001 | P007 | 23 |
| C02 | O123 | P007 | 19 |
| C02 | O123 | P230 | 82 |
| C01 | O001 | P890 | 42 |

Key in above table: {cust_id, order_id}

This is a composite key as it consists of more than one attribute.

6. **Foreign Key** – Foreign keys are the columns of a table that points to the primary key of another table. They act as a cross-reference between tables.

**UNIT II**

**Single Value Constraints –**

Single value constraints refers that each attribute of an entity set has a single value. If the value of an attribute is missing in a tuple, then we cal fill it with a "null" value. The null value for a attribute will specify that either the value is not known or the value is not applicable. Consider the below example-

| SID | Name | Class (semester) | Age | Driving License Number |
|---|---|---|---|---|
| 8001 | Ankit | 1st | 19 | DL-45698 |
| 8002 | Srishti | 2nd | 18 | DL-45871, DL-89740 |
| 8003 | Somvir | 4th | 22 | DL-95687 |
| 8004 | Sourabh | 6th | 19 | |

Not allowed as a person does not have two driving licenses.

Allowed as a person may or may not have a driving license.

**Integrity Rule 1 (Entity Integrity Rule or Constraint) –**

The Integrity Rule 1 is also called Entity Integrity Rule or Constraint. This rule states that no attribute of primary key will contain a null value. If a relation have a null value in the primary key attribute, then uniqueness property of the primary key cannot be maintained. Consider the example below-

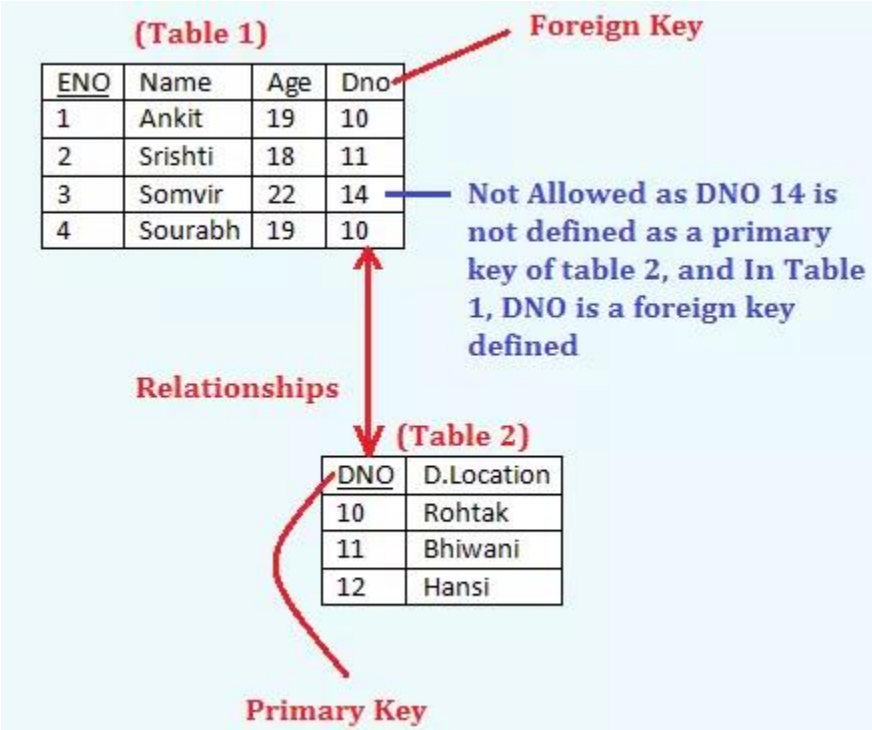
| SID | Name | Class (semester) | Age |
|---|---|---|---|
| 8001 | Ankit | 1st | 19 |
| 8002 | Srishti | 2nd | 18 |
| 8003 | Somvir | 4th | 22 |
| | Sourabh | 6th | 19 |

Not allowed as primary key cannot contain a NULL value

**UNIT II**

**Integrity Rule 2 (Referential Integrity Rule or Constraint) –**

The integrity Rule 2 is also called the Referential Integrity Constraints. This rule states that if a foreign key in Table 1 refers to the Primary Key of Table 2, then every value of the Foreign Key in Table 1 must be null or be available in Table 2. For example,




**Some more Features of Foreign Key –**

Let the table in which the foreign key is defined is Foreign Table or details table i.e. Table 1 in above example and the table that defines the primary key and is referenced by the foreign key is master table or primary table i.e. Table 2 in above example. Then the following properties must be hold :

- Records cannot be **inserted** into a F**oreign table** if corresponding records in the master table do not exist.
- Records of the **master table or Primary Table** cannot be **deleted** or **updated** if corresponding records in the detail table actually exist.

## UNIT II

**General Constraints –**

General constraints are the arbitrary constraints that should hold in the database. Domain Constraints, Key Constraints, Tuple Uniqueness Constraints, Single Value Constraints, Integrity Rule 1 (Entity Integrity) and 2 (Referential Integrity Constraints) are considered to be a fundamental part of the relational data model. However, sometimes it is necessary to specify more general constraints like the CHECK Constraints or the Range Constraints etc.

**Check constraints** can ensure that only specific values are allowed in certain column. For example , if there is a need to allow only three values for the color like 'Bakers Chocolate', 'Glistening Grey' and 'Superior White', then we can apply the check constraint. All other values like 'GREEN' etc would yield an error. Range Constraints is implemented by BETWEEN and NOT BETWEEN. For example, if it is a requirement that student ages be within 16 to 35, then we can apply the range constraints for it.

The below example will explain Check Constraint and Range Constraint –

**UNIT II**

| CarID | Name | Model | Color |
|---|---|---|---|
| C-12378 | Wagon-R | 2008 | Bakers Chocolate |
| C-23478 | Wagon-R | 2008 | Glistening Grey |
| C-45823 | Wagon-R | 2004 | Superior White |
| C-45874 | Wagon-R | 2009 | Green |

**Not Allowed, as CHECK Constraint is applied.**

| SID | Name | Class (semester) | Age |
|---|---|---|---|
| 8001 | Ankit | 1st | 19 |
| 8002 | Srishti | 2nd | 18 |
| 8003 | Somvir | 4th | 22 |
| NULL | Sourabh | 6th | 65 |

**Not allowed, as the range defined is in between 16 and 35**

## RELATIONAL ALGEBRA

Relational Algebra is procedural query language, which takes Relation as input and generate relation as output. Relational algebra mainly provides theoretical foundation for relational databases and SQL.

- *Domain:* set of relations
- Based on set theory
- Contains extensions to manipulate tables
- Functional language
- Procedural, i.e., order to operations, algorithm implicit in the functional evaluation

**UNIT II**

## Relational Algebra Operations

Below are fundamental operations that are "complete". That is, this set of operations alone can define any retrieval.

- Select
- Project
- Rename
- Union
- Set Difference
- Cartesian Product

Convenient, natural additions to the set of operations makes

- Set Intersection
- Natural Join
- Division
- Assignment

## Projection

- Produce a subset of attributes from a relation
- Unselected columns are eliminated
- Duplicate rows are eliminated
- Result is a relation

**Syntax:** $\pi_{\text{attribute- list}}(\textit{relation})$

Example: The table **E** (for **EMPLOYEE**)

| nr | name | salary |
|---|---|---|
| 1 | John | 100 |
| 5 | Sarah | 300 |
| 7 | Tom | 100 |

| SQL | Result | Relational algebra |
|---|---|---|

| | | |
|---|---|---|
| select salary<br>from E | **salary**<br>100<br>300 | **PROJECT**$_{salary}$(E) |
| select nr, salary<br>from E | **nr** **salary**<br>1 100<br>5 300<br>7 100 | **PROJECT**$_{nr, salary}$(E) |

Note that there are no duplicate rows in the result.

**Selection**

Choose a subset of tuples from a relation based on some criteria, results in another relation called a ***"result set"***

Notation uses lower case sigma:

**Synatx: $\sigma_{condition}$(relation)**

The same table **E** (for **EMPLOYEE**) as above.

| SQL | Result | Relational algebra |
|---|---|---|
| select *<br>from E<br>where salary < 200 | **nr** **name** **salary**<br>1 John 100<br>7 Tom 100 | **SELECT**$_{salary < 200}$(E) |
| select *<br>from E<br>where salary < 200<br>and nr >= 7 | **nr** **name** **salary**<br>7 Tom 100 | **SELECT**$_{salary < 200 \text{ and } nr >= 7}$(E) |

**UNIT II**

**SELECT** $_{salary<200\ and\ nr>=7}$**(PROJECT** $_{nr,salary}$**(E))**

| nr | salary |
|---|---|
| 7 | 100 |

## Set Operators

One of the characteristics of RDBMS is that it should support all the transaction on the records in the table by means relational operations. That means it should have strong query language which supports relational algebra. There are three main relational algebras on sets – UNION, SET DIFFERENCE and SET INTERSECT. The same is implemented in database query language using set operators.

There are 4 main set operators used in the query language.

### 1. UNION

It combines the similar columns from two tables into one resultant table. All columns that are participating in the UNION operation should be Union Compatible. This operator combines the records from both the tables into one. If there are duplicate values as a result, then it eliminates the duplicate. The resulting records will be from both table and distinct.

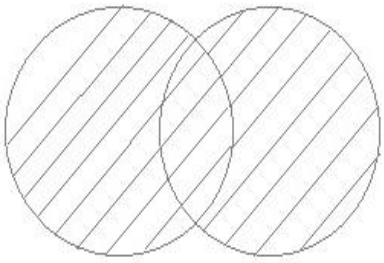
**UNIT II**

| EMP_TEST | | | |
|---|---|---|---|
| EMP_ID | EMP_NAME | EMP_ADDRESS | EMP_SSN |
| 100 | James | Troy | 232434 |
| 104 | Kathy | Holland | 324343 |

Union

| EMP_DESIGN | | | |
|---|---|---|---|
| EMP_ID | ENAME | EMP_ADDRESS | SSN |
| 103 | Rose | Freser Town | 6744545 |
| 102 | Marry | Novi | 343613 |
| 105 | Laurry | Rochester Hills | 97676 |
| 104 | Kathy | Holland | 324343 |

| UNION | | | |
|---|---|---|---|
| EMP_ID | EMP_NAME | EMP_ADDRESS | EMP_SSN |
| 100 | James | Troy | 232434 |
| 102 | Marry | Novi | 343613 |
| 103 | Rose | Freser Town | 6744545 |
| 104 | Kathy | Holland | 324343 |
| 105 | Laurry | Rochester Hills | 97676 |



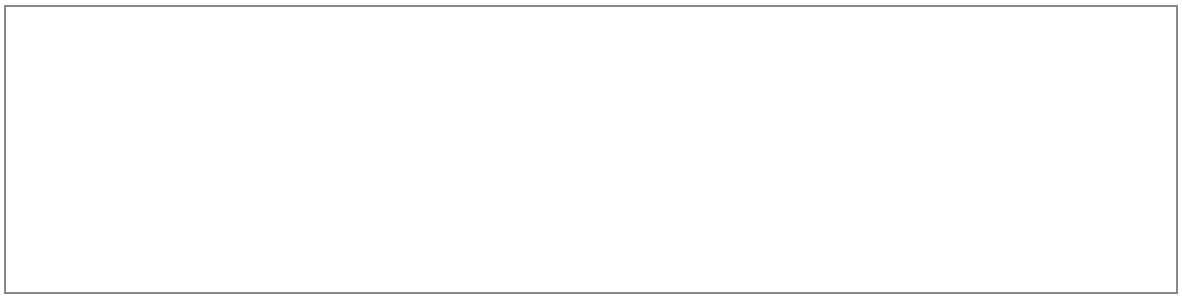
We can notice that Result will have same column names as first query. Duplicate record – 104 from EMP_TEST and EMP_DESIGN are showed only once in the result set. Records are sorted in the result.

**UNION ALL**

**UNIT II**

This operation is also similar to UNION, but it does not eliminate the duplicate records. It shows all the records from both the tables. All other features are same as UNION. We can have conditions in the SELECT query. It need not be a simple SELECT query.

Look at the same example below with UNION ALL operation.



EMP_TEST

| EMP_ID | EMP_NAME | EMP_ADDRESS | EMP_SSN |
|---|---|---|---|
| 100 | James | Troy | 232434 |
| 104 | Kathy | Holland | 324343 |

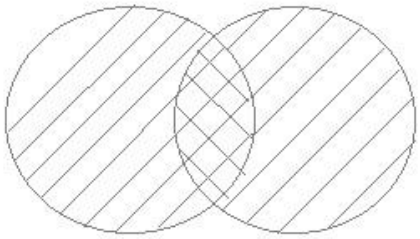
Union ALL

EMP_DESIGN

| EMP_ID | ENAME | EMP_ADDRESS | SSN |
|---|---|---|---|
| 103 | Rose | Freser Town | 6744545 |
| 102 | Marry | Novi | 343613 |
| 105 | Laurry | Rochester Hills | 97676 |
| 104 | Kathy | Holland | 324343 |

UNION

| EMP_ID | EMP_NAME | EMP_ADDRESS | EMP_SSN |
|---|---|---|---|
| 100 | James | Troy | 232434 |
| 102 | Marry | Novi | 343613 |
| 103 | Rose | Freser Town | 6744545 |
| 104 | Kathy | Holland | 324343 |
| 104 | Kathy | Holland | 324343 |
| 105 | Laurry | Rochester Hills | 97676 |

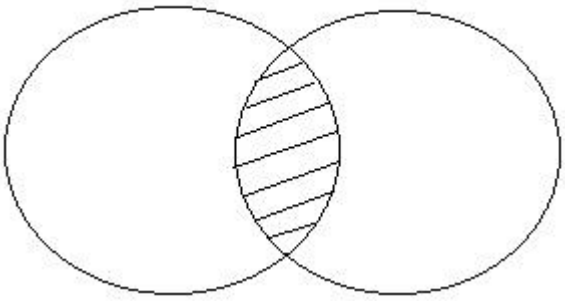


**2. INTERSECT**

This operator is used to pick the records from both the tables which are common to them. In other words it picks only the duplicate records from the tables. Even though it selects duplicate records from the table, each duplicate record will be displayed only once in the result set. It should have UNION Compatible columns to run the query with this operator.

Same example above when used with INTERSECT operator, gives below result.

**UNIT II**



| EMP_TEST | | | |
|---|---|---|---|
| EMP_ID | EMP_NAME | EMP_ADDRESS | EMP_SSN |
| 100 | James | Troy | 232434 |
| 104 | Kathy | Holland | 324343 |

INTERSECT

| EMP_DESIGN | | | |
|---|---|---|---|
| EMP_ID | ENAME | EMP_ADDRESS | SSN |
| 103 | Rose | Freser Town | 6744545 |
| 102 | Marry | Novi | 343613 |
| 105 | Laurry | Rochester Hills | 97676 |
| 104 | Kathy | Holland | 324343 |

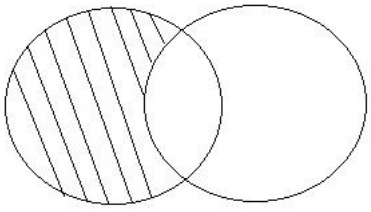
| UNION | | | |
|---|---|---|---|
| EMP_ID | EMP_NAME | EMP_ADDRESS | EMP_SSN |
| 104 | Kathy | Holland | 324343 |

**3. MINUS**

This operator is used to display the records that are present only in the first table or query, and doesn't present in second table / query. It basically subtracts the first query results from the second.

Let us see the same example with MINUS operator.

**UNIT II**



| EMP_TEST | | | |
|---|---|---|---|
| EMP_ID | EMP_NAME | EMP_ADDRESS | EMP_SSN |
| 100 | James | Troy | 232434 |
| 104 | Kathy | Holland | 324343 |

MINUS

| EMP_DESIGN | | | |
|---|---|---|---|
| EMP_ID | ENAME | EMP_ADDRESS | SSN |
| 103 | Rose | Freser Town | 6744545 |
| 102 | Marry | Novi | 343613 |
| 105 | Laurry | Rochester Hills | 97676 |
| 104 | Kathy | Holland | 324343 |

| UNION | | | |
|---|---|---|---|
| EMP_ID | EMP_NAME | EMP_ADDRESS | EMP_SSN |
| 100 | James | Troy | 232434 |

We can notice in the above result that only the records that do not exists in EMP_DESIGN are displayed in the result. The record which appears in both the tables is

eliminated. Similarly, the records that appear in second query but not in the first query are also eliminated.

**4. Division Operator (÷):**

**Table 1**

**STUDENT_SPORTS**

| ROLL_NO | SPORTS |
|---|---|
| 1 | Badminton |
| 2 | Cricket |
| 2 | Badminton |
| 4 | Badminton |

**Table 2**

**ALL_SPORTS**

| SPORTS |
|---|
| Badminton |
| Cricket |

***Division Operator (÷):*** Division operator A÷B can be applied if and only if:

- Attributes of B is proper subset of Attributes of A.
- The relation returned by division operator will have attributes = (All attributes of A – All Attributes of B)
- The relation returned by division operator will return those tuples from relation A which are associated to every B's tuple.

**STUDENT_SPORTS÷ ALL_SPORTS**

- The operation is valid as attributes in ALL_SPORTS is a proper subset of attributes in STUDENT_SPORTS.

- The attributes in resulting relation will have attributes {ROLL_NO,SPORTS}-{SPORTS}=ROLL_NO
- The tuples in resulting relation will have those ROLL_NO which are associated with all B's tuple {Badminton, Cricket}. ROLL_NO 1 and 4 are associated to Badminton only. ROLL_NO 2 is associated to all tuples of B. So the resulting relation will be:

| ROLL_NO |
|---|
| 2 |

## Join in SQL

SQL Join is used to fetch data from two or more tables, which is joined to appear as single set of data. SQL Join is used for combining column from two or more tables by using values common to both tables.**Join** Keyword is used in SQL queries for joining two or more tables. Minimum required condition for joining table, is **(n-1)** where **n**, is number of tables. A table can also join to itself known as, **Self Join**.

### Types of Join

The following are the types of JOIN that we can use in SQL.

- Inner
- Outer
- Left
- Right

### Cross JOIN or Cartesian Product

This type of JOIN returns the cartesian product of rows from the tables in Join. It will return a table which consists of records which combines each row from the first table with each row of the second table.

Cross JOIN Syntax is,

```
SELECT column-name-list
from table-name1
CROSS JOIN
table-name2;
```

---

**Example of Cross JOIN**

The **class** table,

| ID | NAME |
|---|---|
| 1 | abhi |
| 2 | adam |
| 4 | alex |

The **class_info** table,

| ID | Address |
|---|---|
| 1 | DELHI |
| 2 | MUMBAI |
| 3 | CHENNAI |

**Cross** JOIN query will be,

```
SELECT *
 from class,
 cross JOIN class_info;
```

---

The result table will look like,

| ID | NAME | ID | Address |
|---|---|---|---|
| 1 | abhi | 1 | DELHI |
| 2 | adam | 1 | DELHI |
| 4 | alex | 1 | DELHI |
| 1 | abhi | 2 | MUMBAI |
| 2 | adam | 2 | MUMBAI |
| 4 | alex | 2 | MUMBAI |
| 1 | abhi | 3 | CHENNAI |
| 2 | adam | 3 | CHENNAI |
| 4 | alex | 3 | CHENNAI |

## INNER Join or EQUI Join

This is a simple JOIN in which the result is based on matched data as per the equality condition specified in the query.

Inner Join Syntax is,

```
SELECT column-name-list
from table-name1
INNER JOIN
table-name2
WHERE table-name1.column-name = table-name2.column-name;
```

### Example of Inner JOIN

The **class** table,

| ID | NAME |
|---|---|
| 1 | abhi |
| 2 | adam |
| 3 | alex |
| 4 | anu |

The **class_info** table,

| ID | Address |
|---|---|
| 1 | DELHI |
| 2 | MUMBAI |
| 3 | CHENNAI |

**Inner** JOIN query will be,

```
SELECT * from class, class_info where class.id = class_info.id;
```

The result table will look like,

| ID | NAME | ID | Address |
|---|---|---|---|
| 1 | abhi | 1 | DELHI |
| 2 | adam | 2 | MUMBAI |

| | | | |
|---|---|---|---|
| 3 | alex | 3 | CHENNAI |

## Natural JOIN

Natural Join is a type of Inner join which is based on column having same name and same datatype present in both the tables to be joined.

Natural Join Syntax is,

```
SELECT *
from table-name1
NATURAL JOIN
table-name2;
```

### Example of Natural JOIN

The **class** table,

| ID | NAME |
|---|---|
| 1 | abhi |
| 2 | adam |
| 3 | alex |
| 4 | anu |

The **class_info** table,

| ID | Address |
|---|---|
| 1 | DELHI |
| 2 | MUMBAI |

| | |
|---|---|
| 3 | CHENNAI |

**Natural join query will be,**

SELECT * from class NATURAL JOIN class_info;

The result table will look like,

| ID | NAME | Address |
|---|---|---|
| 1 | abhi | DELHI |
| 2 | adam | MUMBAI |
| 3 | alex | CHENNAI |

In the above example, both the tables being joined have ID column(same name and same datatype), hence the records for which value of ID matches in both the tables will be the result of Natural Join of these two tables.

## Outer JOIN

Outer Join is based on both matched and unmatched data. Outer Joins subdivide further into,

- Left Outer Join
- Right Outer Join
- Full Outer Join

## Left Outer Join

The left outer join returns a result table with the **matched data** of two tables then remaining rows of the**left** table and null for the **right** table's column.

Left Outer Join syntax is,

```
SELECT column-name-list
from table-name1
LEFT OUTER JOIN
table-name2
on table-name1.column-name = table-name2.column-name;
```

Left outer Join Syntax for **Oracle** is,

```
select column-name-list
from table-name1,
table-name2
on table-name1.column-name = table-name2.column-name(+);
```

**Example of Left Outer Join**

The **class** table,

| ID | NAME |
|---|---|
| 1 | abhi |
| 2 | adam |
| 3 | alex |
| 4 | anu |
| 5 | ashish |

The **class_info** table,

| ID | Address |
|---|---|
| 1 | DELHI |

| | |
|---|---|
| 2 | MUMBAI |
| 3 | CHENNAI |
| 7 | NOIDA |
| 8 | PANIPAT |

**Left Outer Join** query will be,

```
SELECT * FROM class LEFT OUTER JOIN class_info ON (class.id=class_info.id);
```

The result table will look like,

| ID | NAME | ID | Address |
|---|---|---|---|
| 1 | abhi | 1 | DELHI |
| 2 | adam | 2 | MUMBAI |
| 3 | alex | 3 | CHENNAI |
| 4 | anu | null | null |
| 5 | ashish | null | null |

## Right Outer Join

The right outer join returns a result table with the **matched data** of two tables then remaining rows of the**right table** and null for the **left** table's columns.

Right Outer Join Syntax is,

```
select column-name-list
from table-name1
RIGHT OUTER JOIN
```

*table-name2*

on table-name1.column-name = table-name2.column-name;

Right outer Join Syntax for **Oracle** is,

select column-name-list

from *table-name1*,

*table-name2*

on table-name1.column-name(+) = table-name2.column-name;

**Example of Right Outer Join**

The **class** table,

| ID | NAME |
|---|---|
| 1 | abhi |
| 2 | adam |
| 3 | alex |
| 4 | anu |
| 5 | ashish |

The **class_info** table,

| ID | Address |
|---|---|
| 1 | DELHI |
| 2 | MUMBAI |
| 3 | CHENNAI |

| | |
|---|---|
| 7 | NOIDA |
| 8 | PANIPAT |

**Right Outer Join** query will be,

```
SELECT * FROM class RIGHT OUTER JOIN class_info on (class.id=class_info.id);
```

The result table will look like,

| ID | NAME | ID | Address |
|---|---|---|---|
| 1 | abhi | 1 | DELHI |
| 2 | adam | 2 | MUMBAI |
| 3 | alex | 3 | CHENNAI |
| null | null | 7 | NOIDA |
| null | null | 8 | PANIPAT |

## Full Outer Join

The full outer join returns a result table with the **matched data** of two table then remaining rows of both**left** table and then the **right** table.

Full Outer Join Syntax is,

```
select column-name-list
from table-name1
FULL OUTER JOIN
table-name2
on table-name1.column-name = table-name2.column-name;
```

**Example of Full outer join is,**

The **class** table,

| ID | NAME |
|---|---|
| 1 | abhi |
| 2 | adam |
| 3 | alex |
| 4 | anu |
| 5 | ashish |

The **class_info** table,

| ID | Address |
|---|---|
| 1 | DELHI |
| 2 | MUMBAI |
| 3 | CHENNAI |
| 7 | NOIDA |
| 8 | PANIPAT |

**Full Outer Join** query will be like,

```
SELECT * FROM class FULL OUTER JOIN class_info on (class.id=class_info.id);
```

The result table will look like,

| ID | NAME | ID | Address |
|---|---|---|---|

| | | | |
|---|---|---|---|
| 1 | abhi | 1 | DELHI |
| 2 | adam | 2 | MUMBAI |
| 3 | alex | 3 | CHENNAI |
| 4 | anu | null | null |
| 5 | ashish | null | null |
| null | null | 7 | NOIDA |
| null | null | 8 | PANIPAT |

## Aggregate Functions in SQL

In database management an aggregate function is a function where the values of multiple rows are grouped together as input on certain criteria to form a single value of more significant meaning.

**Various Aggregate Functions**

1) Count()

2) Sum()

3) Avg()

4) Min()

5) Max()

Now let us understand each Aggregate function with a example: Table : E

```
Id    Name    Salary
----------------------
1     A       80
2     B       40
```

| | | |
|---|---|---|
| 3 | C | 60 |
| 4 | D | 70 |
| 5 | E | 60 |
| 6 | F | Null |

**1. COUNT()**

**Syntax:**

SELECT COUNT(*column_name*)
FROM *table_name*
WHERE *condition*;

**Example**

SELECT COUNT(ID) FROM E;

***Count(*):*** Returns total number of records .i.e 6.

***Count(salary):*** Return number of Non Null values over the column salary. i.e 5.

***Count(Distinct Salary):*** Return number of distinct Non Null values over the column salary .i.e 4

**2. SUM()**

**Syntax**

SELECT SUM(*column_name*)
FROM *table_name*
WHERE *condition*;

**Example**

SELECT sum(ID) FROM E

***sum(salary):*** Sum all Non Null values of Column salary i.e., 310

***sum(Distinct salary):*** Sum of all distinct Non-Null values i.e., 250.

**3. AVG()**

**Syntax**

SELECT AVG(*column_name*)
FROM *table_name*
WHERE *condition*;

**Example**

SELECT AVG(salary) FROM E;

***Avg(salary)*** = Sum(salary) / count(salary) = 310/5

***Avg(Distinct salary)*** = sum(Distinct salary) / Count(Distinct Salary) = 250/4

**4. MIN()**

**Syntax**

SELECT MIN(*column_name*)
FROM *table_name*
WHERE *condition*;

**Example**

SELECT MIN(salary) FROM E;

***Min(salary):*** Minimum value in the salary column except NULL i.e., 40.

5**. MAX()**

UNIT II

**Syntax**

SELECT MAX(*column_name*)
FROM *table_name*
WHERE *condition*;

**Example**

SELECT MAX(salary) FROM E;

***Max(salary):*** Maximum value in the salary i.e., 80.

**POSSIBLE QUESTIONS**

**2 MARK QUESTIONS**

1. Draw an ER diagram for 1:M relationship.
2. Draw an ER diagram for M:M relationship
3. Write the syntax for DELETE FROM WHERE command in SQL.
4. What is an Entity type?
5. Define cardinalities.
6. Draw an ER model for M:M relationship.

## UNIT II

**6 MARK QUESTIONS**

1. IIustrate with syntax and example of relational constraints.
2. Discuss relational algebra with proper example.
3. Explain the following: i) Key constraints ii) Integrity constraints.
4. Discuss in detail the operators SELECT, PROJECT and UNION with suitable examples.
4. Describe SET operators and Joins in DBMS with example
5. Explain in detail about various key constraints used in database system.