# UNIT – II (RELATIONAL DATA MODEL AND LANGUAGE)

A relational database stores data in the form of relations (tables). In this model, the data is organized into a collection of two-dimensional inter-related tables 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 represent 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.

**Degree**: The total number of attributes which in the relation is called the degree of the relation.

## Different types of Database Users

Database users are categorized based up on their interaction with the database. These are seven types of database users in DBMS.

**Database Administrator (DBA) :**

Database Administrator (DBA) is a person/team who defines the schema and also controls the 3 levels of database. The DBA will then create a new account id and password for the user if he/she need to access the database. DBA is also responsible for providing security to the database and he allows only the authorized users to access/modify the data base. DBA is responsible for the problems such as security breaches and poor system response time.

- DBA also monitors the recovery and backup and provide technical support.
- The DBA has a DBA account in the DBMS which called a system or superuser account.
- DBA repairs damage caused due to hardware and/or software failures.

DBA is the one having privileges to perform DCL (Data Control Language) operations such as GRANT and REVOKE, to allow/restrict a particular user from accessing the database.

**Naive / Parametric End Users :**

Parametric End Users are the unsophisticated who don't have any DBMS knowledge but they frequently use the database applications in their daily life to get the desired results. For examples, Railway's ticket booking users are naive users. Clerks in any bank is a naive user because they don't have any DBMS knowledge but they still use the database and perform their given task.

**System Analyst :**

System Analyst is a user who analyzes the requirements of parametric end users. They check whether all the requirements of end users are satisfied.

**Sophisticated Users :**

Sophisticated users can be engineers, scientists, business analyst, who are familiar with the database. They can develop their own database applications according to their requirement. They don't write the program code but they interact the database by writing SQL queries directly through the query processor.

**Database Designers :**

Data Base Designers are the users who design the structure of database which includes tables, indexes, views, triggers, stored procedures and constraints which are usually enforced before the database is created or populated with data. He/she controls what data must be stored and how the data items to be related. It is responsibility of Database Designers to understand the requirements of different user groups and then create a design which satisfies the need of all the user groups.

**Application Programmers :**

Application Programmers also referred as System Analysts or simply Software Engineers, are the back-end programmers who writes the code for the application programs. They are the computer professionals. These programs could be written in Programming languages such as Visual Basic, Developer, C, FORTRAN, COBOL etc. Application programmers design, debug, test, and maintain set of programs called "canned transactions" for the Naive (parametric) users in order to interact with database.

**Casual Users / Temporary Users :**

Casual Users are the users who occasionally use/access the database but each time when they access the database they require the new information, for example, Middle or higher level manager.

**Specialized users :**

Specialized users are sophisticated users who write specialized database application that does not fit into the traditional data-processing framework. Among these applications are computer aided-design systems, knowledge-base and expert systems etc.

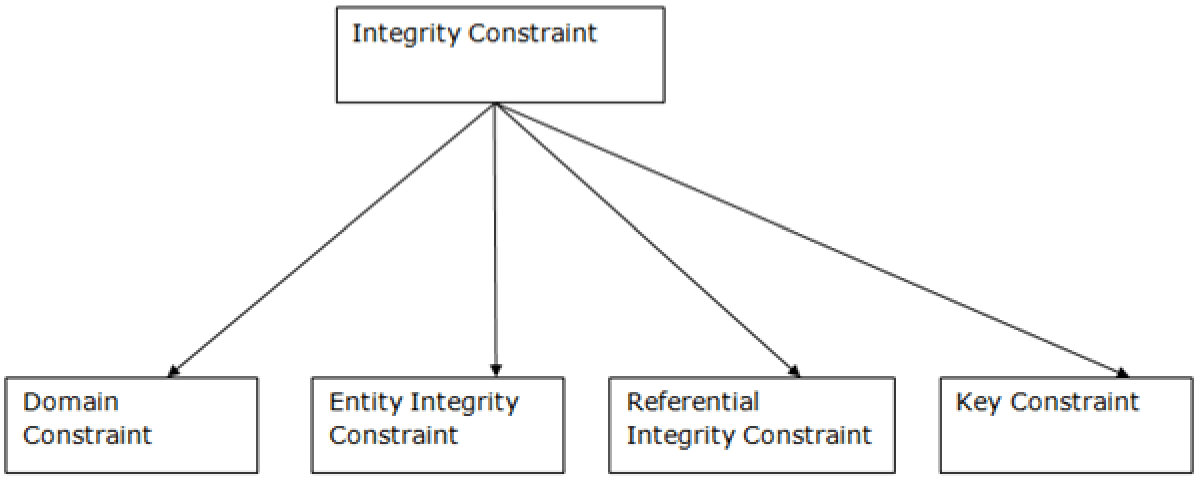
## Integrity Constraints

Integrity constraints are a set of rules. It is used to maintain the quality of information.

Integrity constraints ensure that the data insertion, updating, and other processes have to be performed in such a way that data integrity is not affected.

Thus, integrity constraint is used to guard against accidental damage to the database.

### Types of Integrity Constraint

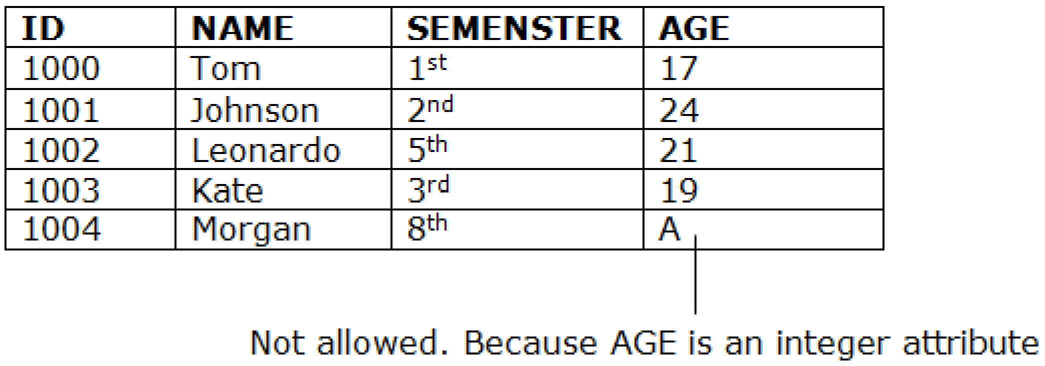


**1. Domain constraints**

Domain constraints can be defined as the definition of a valid set of values for an attribute.

The data type of domain includes string, character, integer, time, date, currency, etc. The value of the attribute must be available in the corresponding domain.

Example:



| ID | NAME | SEMENSTER | AGE |
|---|---|---|---|
| 1000 | Tom | 1st | 17 |
| 1001 | Johnson | 2nd | 24 |
| 1002 | Leonardo | 5th | 21 |
| 1003 | Kate | 3rd | 19 |
| 1004 | Morgan | 8th | A |

Not allowed. Because AGE is an integer attribute

There are two types of constraints that come under domain constraint and they are:

**A) Domain Constraints – Not Null:**

Null values are the values that are unassigned or we can also say that which are unknown or the missing attribute values and by default, a column can hold the null values. Now as we know that the Not Null constraint restricts a column to not accept the null values which means it only restricts a field to always contain a value which means you cannot insert a new record or update a record without adding a value into the field.

Example: In the 'employee' database, every employee must have a name associated with them.

Create table employee

(employee_id varchar(30),

employee_name varchar(30) not null,

salary NUMBER);

**B) Domain Constraints – Check:**

It defines a condition that each row must satisfy which means it restricts the value of a column between ranges or we can say that it is just like a condition or filter checking before saving data into a column. It ensures that when a tuple is inserted inside the relation must satisfy the predicate given in the check clause.

Example: We need to check whether the entered id number is greater than 0 or not for the employee table.

Create table employee

(employee_id varchar(30) not null check(employee_id > 0),

employee_name varchar(30),

salary NUMBER);

The above example creates CHECK constraints on the employee_id column and specifies that the column employee_id must only include integers greater than 0.

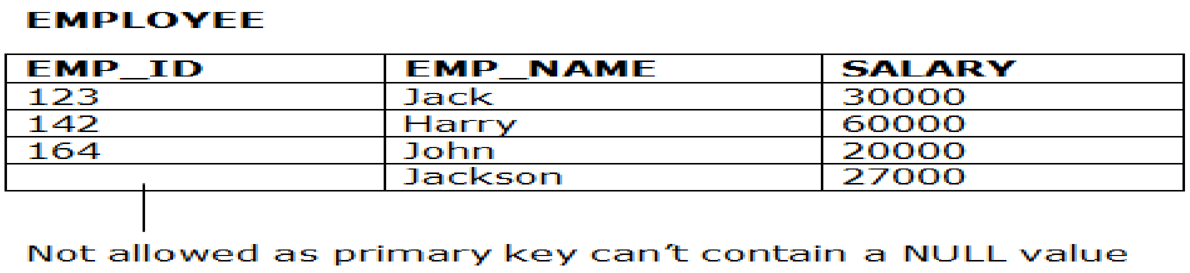
**2. Entity integrity constraints**

The entity integrity constraint states that primary key value can't be null.

This is because the primary key value is used to identify individual rows in relation and if the primary key has a null value, then we can't identify those rows.

A table can contain a null value other than the primary key field.

Example:



EMPLOYEE

| EMP_ID | EMP_NAME | SALARY |
|---|---|---|
| 123 | Jack | 30000 |
| 142 | Harry | 60000 |
| 164 | John | 20000 |
|  | Jackson | 27000 |

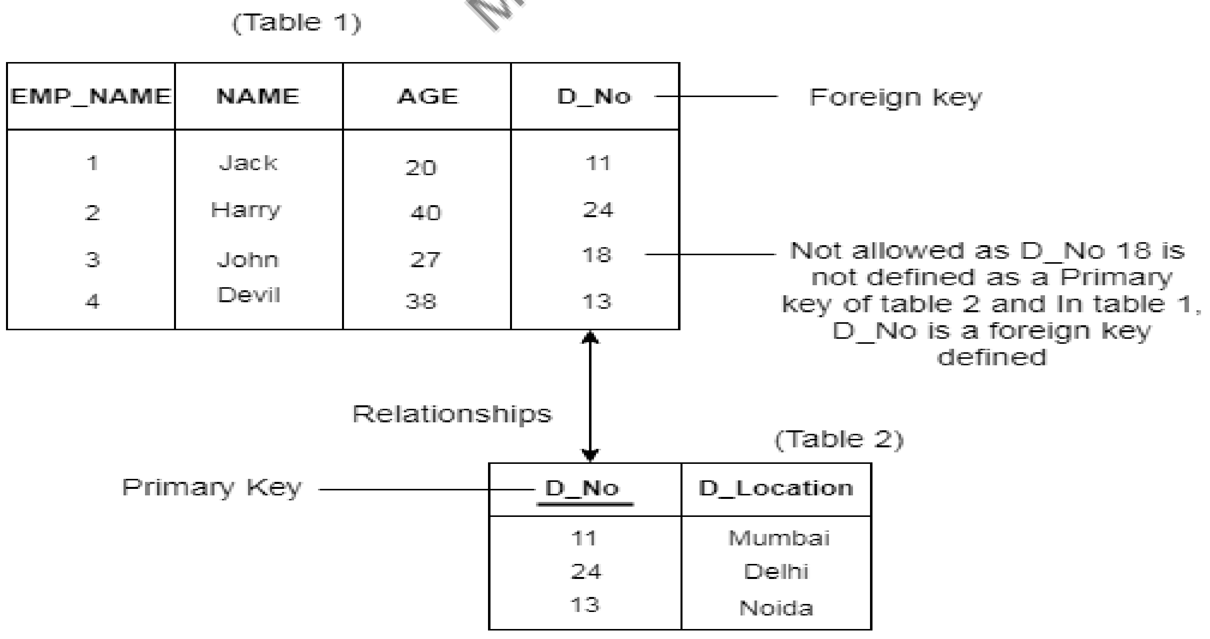
Not allowed as primary key can't contain a NULL value

## 3. Referential Integrity Constraints

A referential integrity constraint is specified between two tables.

In the Referential integrity constraints, 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.

Example:



(Table 1)

| EMP_NAME | NAME | AGE | D_No |
|---|---|---|---|
| 1 | Jack | 20 | 11 |
| 2 | Harry | 40 | 24 |
| 3 | John | 27 | 18 |
| 4 | Devil | 38 | 13 |

(Table 2)

| D_No | D_Location |
|---|---|
| 11 | Mumbai |
| 24 | Delhi |
| 13 | Noida |

Foreign key
Not allowed as D_No 18 is not defined as a Primary key of table 2 and In table 1, D_No is a foreign key defined
Relationships
Primary Key

A referential integrity constraint is also known **as foreign key constraint**. A foreign key is a key whose values are derived from the Primary key of another table.

The table from which the values are derived is known as **Master or Referenced Table** and the Table in which values are inserted accordingly is known as **Child or Referencing Table**, In other words, we can say that the table containing the foreign key is called the child table, and the table containing the Primary key/candidate key is called the referenced or parent table. When we talk about the database relational model, the candidate key can be defined as a set of attribute which can have zero or more attributes.

The syntax of the Master Table or Referenced table is:

**CREATE TABLE Student (Roll int PRIMARY KEY, Name varchar(25) , Course varchar(10) );**

Here column Roll is acting as Primary Key, which will help in deriving the value of foreign key in the child table.

## Relational Algebra

Relational Algebra is a procedural query language. Relational algebra mainly provides a theoretical foundation for relational databases and SQL. The main purpose of using Relational Algebra is to define operators that transform one or more input relations into an output relation. Given that these operators accept relations as input and produce relations as output, they can be combined and used to express potentially complex queries that transform potentially many input relations (whose data are stored in the database) into a single output relation (the query results). As it is pure mathematics, there is no use of English Keywords in Relational Algebra and operators are represented using symbols.

**Fundamental Operators**

These are the basic/fundamental operators used in Relational Algebra.

- Selection(σ)
- Projection(π)
- Union(U)
- Set Difference(-)
- Set Intersection(∩)
- Rename(ρ)
- Cartesian Product(X)

**1. Selection(σ):** It is used to select required tuples of the relations.

Example:

| A | B | C |
|---|---|---|
| 1 | 2 | 4 |
| 2 | 2 | 3 |
| 3 | 2 | 3 |
| 4 | 3 | 4 |

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

| A | B | C |
|---|---|---|
| 1 | 2 | 4 |
| 4 | 3 | 4 |

Note: The selection operator only selects the required tuples but does not display them. For display, the data projection operator is used.

**2. Projection(π):** It is used to project required column data from a relation.

Example: Consider Table 1. Suppose we want columns B and C from Relation R.

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

| B | C |
|---|---|
| 2 | 4 |

| B | C |
|---|---|
| 2 | 3 |
| 3 | 4 |

Note: By Default, projection removes duplicate data.

**3. Union(U):** Union operation in relational algebra is the same as union operation in set theory.

Example:

FRENCH

| Student_Name | Roll_Number |
|---|---|
| Ram | 01 |
| Mohan | 02 |
| Vivek | 13 |
| Geeta | 17 |

GERMAN

| Student_Name | Roll_Number |
|---|---|
| Vivek | 13 |
| Geeta | 17 |

| Student_Name | Roll_Number |
|---|---|
| Shyam | 21 |
| Rohan | 25 |

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

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

| Student_Name |
|---|
| Ram |
| Mohan |
| Vivek |
| Geeta |
| Shyam |
| Rohan |

Note: 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 in relational algebra is the same set difference operation as in set theory.

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

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

| Student_Name |
|---|
| Ram |
| Mohan |

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

**5. Set Intersection(∩):** Set Intersection in relational algebra is the same set intersection operation in set theory.

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

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

| Student_Name |
|---|
| Vivek |
| Geeta |

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

**6. Rename(ρ):** Rename is a unary operation used for renaming attributes of a relation.

ρ(a/b)R will rename the attribute 'b' of the relation by 'a'.

**7. Cross Product(X):** Cross-product between two relations. 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.

Example:

A

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

B

| ID | Course |
|---|---|
| 1 | DS |
| 2 | DBMS |

A X B

| 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 |

| Name | Age | Sex | ID | Course |
|---|---|---|---|---|
| Kim | 20 | M | 1 | DS |
| Kim | 20 | M | 2 | DBMS |

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

.

## Join Operations:

A Join operation combines related tuples from different relations, if and only if a given join condition is satisfied. It is denoted by ⋈.

Example:

EMPLOYEE

| EMP_CODE | EMP_NAME |
|---|---|
| 101 | Stephan |
| 102 | Jack |
| 103 | Harry |

SALARY

| EMP_CODE | SALARY |
|---|---|
| 101 | 50000 |

|  |  |
|---|---|
| 102 | 30000 |
| 103 | 25000 |

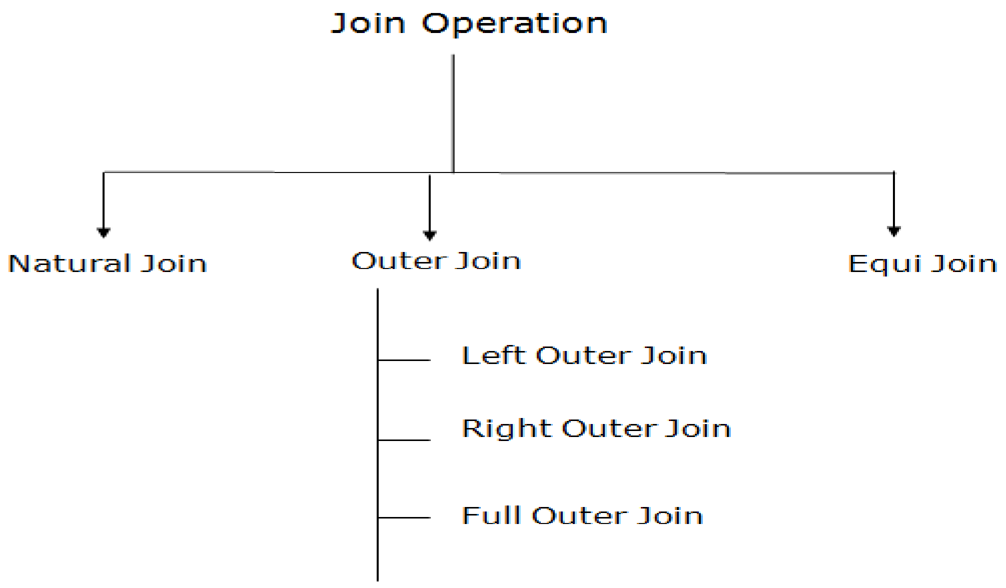
Operation: (EMPLOYEE ⋈ SALARY)

Result:

Backward Skip 10sPlay VideoForward Skip 10s

| EMP_CODE | EMP_NAME | SALARY |
|---|---|---|
| 101 | Stephan | 50000 |
| 102 | Jack | 30000 |
| 103 | Harry | 25000 |

**Types of Join operations:**



## 1. Natural Join:

A natural join is the set of tuples of all combinations in R and S that are equal on their common attribute names.

It is denoted by ⋈.

Example: Let's use the above EMPLOYEE table and SALARY table:

Input:

∏EMP_NAME, SALARY (EMPLOYEE ⋈ SALARY)

Output:

| EMP_NAME | SALARY |
|---|---|
| Stephan | 50000 |
| Jack | 30000 |
| Harry | 25000 |

## 2. Outer Join:

The outer join operation is an extension of the join operation. It is used to deal with missing information.

Example:

EMPLOYEE

| EMP_NAME | STREET | CITY |
|---|---|---|
| Ram | Civil line | Mumbai |
| Shyam | Park street | Kolkata |

| | | |
|---|---|---|
| Ravi | M.G. Street | Delhi |
| Hari | Nehru nagar | Hyderabad |

FACT_WORKERS

| EMP_NAME | BRANCH | SALARY |
|---|---|---|
| Ram | Infosys | 10000 |
| Shyam | Wipro | 20000 |
| Kuber | HCL | 30000 |
| Hari | TCS | 50000 |

Input:

(EMPLOYEE ⋈ FACT_WORKERS)

Output:

| EMP_NAME | STREET | CITY | BRANCH | SALARY |
|---|---|---|---|---|
| Ram | Civil line | Mumbai | Infosys | 10000 |
| Shyam | Park street | Kolkata | Wipro | 20000 |
| Hari | Nehru nagar | Hyderabad | TCS | 50000 |

## An outer join is basically of three types:

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

**a. Left outer join:**

Left outer join contains the set of tuples of all combinations in R and S that are equal on their common attribute names.

In the left outer join, tuples in R have no matching tuples in S.

It is denoted by ⟕.

Example: Using the above EMPLOYEE table and FACT_WORKERS table

Input:

EMPLOYEE ⟕ FACT_WORKERS

| EMP_NAME | STREET | CITY | BRANCH | SALARY |
|---|---|---|---|---|
| Ram | Civil line | Mumbai | Infosys | 10000 |
| Shyam | Park street | Kolkata | Wipro | 20000 |
| Hari | Nehru street | Hyderabad | TCS | 50000 |
| Ravi | M.G. Street | Delhi | NULL | NULL |

**b. Right outer join:**

Right outer join contains the set of tuples of all combinations in R and S that are equal on their common attribute names.

In right outer join, tuples in S have no matching tuples in R.

It is denoted by ⟖.

Example: Using the above EMPLOYEE table and FACT_WORKERS Relation

Input:

EMPLOYEE ⟖ FACT_WORKERS

Output:

| EMP_NAME | BRANCH | SALARY | STREET | CITY |
|---|---|---|---|---|
| Ram | Infosys | 10000 | Civil line | Mumbai |
| Shyam | Wipro | 20000 | Park street | Kolkata |
| Hari | TCS | 50000 | Nehru street | Hyderabad |
| Kuber | HCL | 30000 | NULL | NULL |

**c. Full outer join:**

Full outer join is like a left or right join except that it contains all rows from both tables.

In full outer join, tuples in R that have no matching tuples in S and tuples in S that have no matching tuples in R in their common attribute name.

It is denoted by ⟗.

Example: Using the above EMPLOYEE table and FACT_WORKERS table

Input:

EMPLOYEE ⟗ FACT_WORKERS

Output:

| EMP_NAME | STREET | CITY | BRANCH | SALARY |
|---|---|---|---|---|
| Ram | Civil line | Mumbai | Infosys | 10000 |
| Shyam | Park street | Kolkata | Wipro | 20000 |
| Hari | Nehru street | Hyderabad | TCS | 50000 |
| Ravi | M.G. Street | Delhi | NULL | NULL |
| Kuber | NULL | NULL | HCL | 30000 |

**3. Equi join:**

It is also known as an inner join. It is the most common join. It is based on matched data as per the equality condition. The equi join uses the comparison operator(=).

Example:

CUSTOMER RELATION

| CLASS_ID | NAME |
|---|---|
| 1 | John |
| 2 | Harry |
| 3 | Jackson |

PRODUCT

| PRODUCT_ID | CITY |
|---|---|
| 1 | Delhi |
| 2 | Mumbai |
| 3 | Noida |

Input:

CUSTOMER ⋈ PRODUCT

Output:

| CLASS_ID | NAME | PRODUCT_ID | CITY |
|---|---|---|---|
| 1 | John | 1 | Delhi |
| 2 | Harry | 2 | Mumbai |
| 3 | Harry | 3 | Noida |

## Relational Calculus

As Relational Algebra is a procedural query language, Relational Calculus is a non-procedural query language. It basically deals with the end results. It always tells me what to do but never tells me how to do it.

There are two types of Relational Calculus

- **Tuple Relational Calculus(TRC)**
- **Domain Relational Calculus(DRC)**

Relational Calculus is the formal query language. It is also known as Declarative language. In Relational Calculus, the order is not specified in which the operation has to be performed. Relational Calculus means what result we have to obtain.

**1. Tuple Relational Calculus (TRC)**

It is a non-procedural query language which is based on finding a number of tuple variables also known as range variable for which predicate holds true. It describes the desired information without giving a specific procedure for obtaining that information. The tuple relational calculus is specified to select the tuples in a relation. In TRC, filtering variable uses the tuples of a relation. The result of the relation can have one or more tuples.

Notation:

A Query in the tuple relational calculus is expressed as following notation

{T | P (T)}   or {T | Condition (T)}

Where

T is the resulting tuples

P(T) is the condition used to fetch T.

For example:

{ T.name | Author(T) AND T.article = 'database' }

Output: This query selects the tuples from the AUTHOR relation. It returns a tuple with 'name' from Author who has written an article on 'database'.

TRC (tuple relation calculus) can be quantified. In TRC, we can use Existential (∃) and Universal Quantifiers (∀).

For example:

{ R| ∃T ∈ Authors(T.article='database' AND R.name=T.name)}

Output: This query will yield the same result as the previous one.

**2. Domain Relational Calculus (DRC)**

The second form of relation is known as Domain relational calculus. In domain relational calculus, filtering variable uses the domain of attributes. Domain relational calculus uses the same operators as tuple calculus. It uses logical connectives ∧ (and), ∨ (or) and ┓ (not). It uses Existential (∃) and Universal Quantifiers (∀) to bind the variable. The QBE or Query by example is a query language related to domain relational calculus.

Notation:

{ a1, a2, a3, ..., an | P (a1, a2, a3, ... ,an)}

Where

a1, a2 are attributes
P stands for formula built by inner attributes

For example:

{< article, page, subject > | ∈ javatpoint ∧ subject = 'database'}

Output: This query will yield the article, page, and subject from the relational javatpoint, where the subject is a database.

## Join Operations:

A Join operation combines related tuples from different relations, if and only if a given join condition is satisfied. It is denoted by ⋈.

Example:

EMPLOYEE

| EMP_CODE | EMP_NAME |
|---|---|
| 101 | Stephan |

| | |
|---|---|
| 102 | Jack |
| 103 | Harry |

SALARY

| EMP_CODE | SALARY |
|---|---|
| 101 | 50000 |
| 102 | 30000 |
| 103 | 25000 |

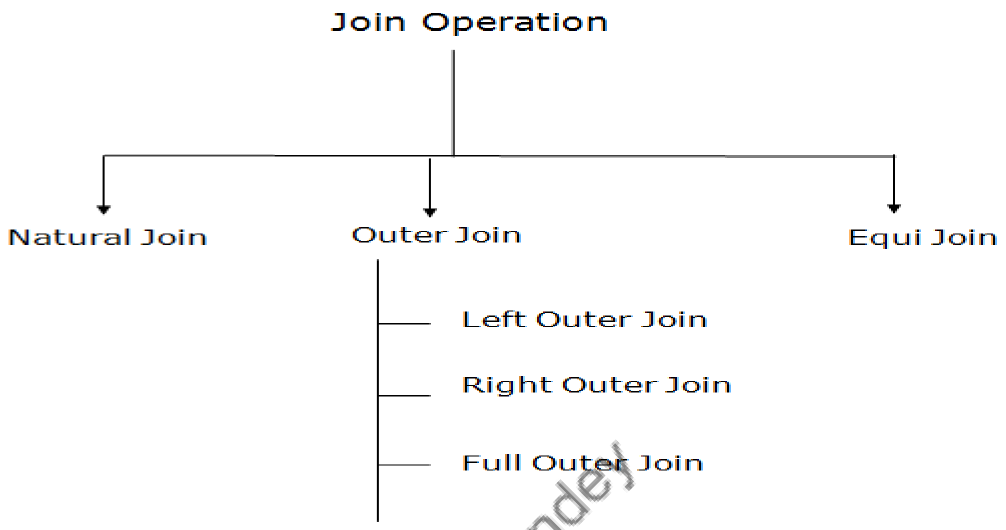
Operation: (EMPLOYEE ⋈ SALARY)

Result:

Backward Skip 10sPlay VideoForward Skip 10s

| EMP_CODE | EMP_NAME | SALARY |
|---|---|---|
| 101 | Stephan | 50000 |
| 102 | Jack | 30000 |
| 103 | Harry | 25000 |

## Types of Join operations:



**1. Natural Join:**

A natural join is the set of tuples of all combinations in R and S that are equal on their common attribute names.

It is denoted by ⋈.

Example: Let's use the above EMPLOYEE table and SALARY table:

Input:

∏EMP_NAME, SALARY (EMPLOYEE ⋈ SALARY)

Output:

| EMP_NAME | SALARY |
|---|---|
| Stephan | 50000 |

|  |  |
|---|---|
| Jack | 30000 |
| Harry | 25000 |

**2. Outer Join:**

The outer join operation is an extension of the join operation. It is used to deal with missing information.

Example:

EMPLOYEE

| EMP_NAME | STREET | CITY |
|---|---|---|
| Ram | Civil line | Mumbai |
| Shyam | Park street | Kolkata |
| Ravi | M.G. Street | Delhi |
| Hari | Nehru nagar | Hyderabad |

FACT_WORKERS

| EMP_NAME | BRANCH | SALARY |
|---|---|---|
| Ram | Infosys | 10000 |
| Shyam | Wipro | 20000 |
| Kuber | HCL | 30000 |

|  |  |  |
|---|---|---|
| Hari | TCS | 50000 |

Input:

(EMPLOYEE ⋈ FACT_WORKERS)

Output:

| EMP_NAME | STREET | CITY | BRANCH | SALARY |
|---|---|---|---|---|
| Ram | Civil line | Mumbai | Infosys | 10000 |
| Shyam | Park street | Kolkata | Wipro | 20000 |
| Hari | Nehru nagar | Hyderabad | TCS | 50000 |

**An outer join is basically of three types**:

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

**a. Left outer join:**

Left outer join contains the set of tuples of all combinations in R and S that are equal on their common attribute names.

In the left outer join, tuples in R have no matching tuples in S.

It is denoted by ⟕.

Example: Using the above EMPLOYEE table and FACT_WORKERS table

Input:

EMPLOYEE ⟕ FACT_WORKERS

| EMP_NAME | STREET | CITY | BRANCH | SALARY |
|---|---|---|---|---|
| Ram | Civil line | Mumbai | Infosys | 10000 |
| Shyam | Park street | Kolkata | Wipro | 20000 |
| Hari | Nehru street | Hyderabad | TCS | 50000 |
| Ravi | M.G. Street | Delhi | NULL | NULL |

**b. Right outer join:**

Right outer join contains the set of tuples of all combinations in R and S that are equal on their common attribute names.

In right outer join, tuples in S have no matching tuples in R.

It is denoted by ⟖.

Example: Using the above EMPLOYEE table and FACT_WORKERS Relation

Input:

EMPLOYEE ⟖ FACT_WORKERS

Output:

| EMP_NAME | BRANCH | SALARY | STREET | CITY |
|---|---|---|---|---|
| Ram | Infosys | 10000 | Civil line | Mumbai |
| Shyam | Wipro | 20000 | Park street | Kolkata |
| Hari | TCS | 50000 | Nehru street | Hyderabad |

|  |  |  |  |  |
|---|---|---|---|---|
| Kuber | HCL | 30000 | NULL | NULL |

**c. Full outer join:**

Full outer join is like a left or right join except that it contains all rows from both tables.

In full outer join, tuples in R that have no matching tuples in S and tuples in S that have no matching tuples in R in their common attribute name.

It is denoted by ⟗.

Example: Using the above EMPLOYEE table and FACT_WORKERS table

Input:

EMPLOYEE ⟗ FACT_WORKERS

Output:

| EMP_NAME | STREET | CITY | BRANCH | SALARY |
|---|---|---|---|---|
| Ram | Civil line | Mumbai | Infosys | 10000 |
| Shyam | Park street | Kolkata | Wipro | 20000 |
| Hari | Nehru street | Hyderabad | TCS | 50000 |
| Ravi | M.G. Street | Delhi | NULL | NULL |
| Kuber | NULL | NULL | HCL | 30000 |

**3. Equi join:**

It is also known as an inner join. It is the most common join. It is based on matched data as per the equality condition. The equi join uses the comparison operator(=).

Example:

CUSTOMER RELATION

| CLASS_ID | NAME |
|---|---|
| 1 | John |
| 2 | Harry |
| 3 | Jackson |

PRODUCT

| PRODUCT_ID | CITY |
|---|---|
| 1 | Delhi |
| 2 | Mumbai |
| 3 | Noida |

Input:

CUSTOMER ⋈ PRODUCT

Output:

| CLASS_ID | NAME | PRODUCT_ID | CITY |
|---|---|---|---|
| 1 | John | 1 | Delhi |
| 2 | Harry | 2 | Mumbai |

|  |  |  |  |
|---|---|---|---|
| 3 | Harry | 3 | Noida |

is denoted as:

{ t | P(t) }

Where,

t: the set of tuples

p: is the condition which is true for the given set of tuples.

## **Difference between Relational Algebra and Relational Calculus:**

| S.NO | Basis of Comparison | Relational Algebra | Relational Calculus |
|---|---|---|---|
| 1. | Language Type | It is a Procedural language. | Relational Calculus is a Declarative (non-procedural) language. |
| 2. | Procedure | Relational Algebra means how to obtain the result. | Relational Calculus means what result we have to obtain. |
| 3. | Order | In Relational Algebra, the order is specified in which the operations have to be performed. | In Relational Calculus, the order is not specified. |
| 4. | Domain | Relational Algebra is independent of the domain. | Relation Calculus can be domain-dependent because of domain relational calculus. |

| S.NO | Basis of Comparison | Relational Algebra | Relational Calculus |
|---|---|---|---|
| 5. | Programming language | Relational Algebra is nearer to a programming language. | Relational Calculus is not nearer to programming language but to natural language. |
| 6. | Inclusion in SQL | The SQL includes only some features from the relational algebra. | SQL is based to a greater extent on the tuple relational calculus. |
| 7. | Relationally completeness | Relational Algebra is one of the languages in which queries can be expressed but the queries should also be expressed in relational calculus to be relationally complete. | For a database language to be relationally complete, the query written in it must be expressible in relational calculus. |
| 8. | Query Evaluation | The evaluation of the query relies on the order specification in which the operations must be performed. | The order of operations does not matter in relational calculus for the evaluation of queries. |
| 9. | Database access | For accessing the database, relational algebra provides a solution in terms of what is required and how to get that information by following a step-by-step description. | For accessing the database, relational calculus provides a solution in terms as simple as what is required and lets the system find the solution for that. |

| S.NO | Basis of Comparison | Relational Algebra | Relational Calculus |
|---|---|---|---|
| 10. | Expressiveness | The expressiveness of any given language is judged using relational algebra operations as a standard. | The completeness of a language is measured in the manner that it is least as powerful as calculus. That implies relation defined using some expression of the calculus is also definable by some other expression, the language is in question. |

## **What are the characteristics of SQL?**

- SQL is easy to learn.
- SQL is used to access data from relational database management systems.
- SQL can execute queries against the database.
- SQL is used to describe the data.
- SQL is used to define the data in the database and manipulate it when needed.
- SQL is used to create and drop the database and table.
- SQL is used to create a view, stored procedure, function in a database.
- SQL allows users to set permissions on tables, procedures, and views.

## **MySQL - Data Types**

Properly defining the fields in a table is important to the overall optimization of your database. You should use only the type and size of field you really need to use. For example, do not define a field 10 characters wide, if you know you are only going to use 2 characters. These type of fields (or columns) are also referred to as data types, after the type of data you will be storing in those fields.

MySQL uses many different data types broken into three categories −

- Numeric

- Date and Time
- String Types.

## Numeric Data Types

MySQL uses all the standard ANSI SQL numeric data types, so if you're coming to MySQL from a different database system, these definitions will look familiar to you.

The following list shows the common numeric data types and their descriptions −

**INT** − A normal-sized integer that can be signed or unsigned. If signed, the allowable range is from -2147483648 to 2147483647. If unsigned, the allowable range is from 0 to 4294967295. You can specify a width of up to 11 digits.

**TINYINT** − A very small integer that can be signed or unsigned. If signed, the allowable range is from -128 to 127. If unsigned, the allowable range is from 0 to 255. You can specify a width of up to 4 digits.

**SMALLINT** − A small integer that can be signed or unsigned. If signed, the allowable range is from -32768 to 32767. If unsigned, the allowable range is from 0 to 65535. You can specify a width of up to 5 digits.

**MEDIUMINT** − A medium-sized integer that can be signed or unsigned. If signed, the allowable range is from -8388608 to 8388607. If unsigned, the allowable range is from 0 to 16777215. You can specify a width of up to 9 digits.

**BIGINT** − A large integer that can be signed or unsigned. If signed, the allowable range is from -9223372036854775808 to 9223372036854775807. If unsigned, the allowable range is from 0 to 18446744073709551615. You can specify a width of up to 20 digits.

**FLOAT(M,D**) − A floating-point number that cannot be unsigned. You can define the display length (M) and the number of decimals (D). This is not required and will default to 10,2, where 2 is the number of decimals and 10 is the total number of digits (including decimals). Decimal precision can go to 24 places for a FLOAT.

**DOUBLE(M,D)** − A double precision floating-point number that cannot be unsigned. You can define the display length (M) and the number of decimals (D). This is not required and will default to 16,4, where 4 is the number of decimals. Decimal precision can go to 53 places for a DOUBLE. REAL is a synonym for DOUBLE.

**DECIMAL(M,D)** − An unpacked floating-point number that cannot be unsigned. In the unpacked decimals, each decimal corresponds to one byte. Defining the display length (M) and the number of decimals (D) is required. NUMERIC is a synonym for DECIMAL.

**Date and Time Types**

The MySQL date and time datatypes are as follows −

**DATE** − A date in YYYY-MM-DD format, between 1000-01-01 and 9999-12-31. For example, December 30th, 1973 would be stored as 1973-12-30.

**DATETIME** − A date and time combination in YYYY-MM-DD HH:MM:SS format, between 1000-01-01 00:00:00 and 9999-12-31 23:59:59. For example, 3:30 in the afternoon on December 30th, 1973 would be stored as 1973-12-30 15:30:00.

**TIMESTAMP** − A timestamp between midnight, January 1st, 1970 and sometime in 2037. This looks like the previous DATETIME format, only without the hyphens between numbers; 3:30 in the afternoon on December 30th, 1973 would be stored as 19731230153000 ( YYYYMMDDHHMMSS ).

**TIME** − Stores the time in a HH:MM:SS format.

**YEAR(M)** − Stores a year in a 2-digit or a 4-digit format. If the length is specified as 2 (for example YEAR(2)), YEAR can be between 1970 to 2069 (70 to 69). If the length is specified as 4, then YEAR can be 1901 to 2155. The default length is 4.

**String Types**

Although the numeric and date types are fun, most data you'll store will be in a string format. This list describes the common string datatypes in MySQL.

**CHAR(M)** − A fixed-length string between 1 and 255 characters in length (for example CHAR(5)), right-padded with spaces to the specified length when stored. Defining a length is not required, but the default is 1.

**VARCHAR(M)** − A variable-length string between 1 and 255 characters in length. For example, VARCHAR(25). You must define a length when creating a VARCHAR field.

**BLOB or TEXT** − A field with a maximum length of 65535 characters. BLOBs are "Binary Large Objects" and are used to store large amounts of binary data, such as images or other types of files. Fields defined as TEXT also hold large amounts of data. The difference between the two is that the sorts and comparisons on the stored data are case sensitive on BLOBs and are not case sensitive in TEXT fields. You do not specify a length with BLOB or TEXT.

**TINYBLOB or TINYTEXT** − A BLOB or TEXT column with a maximum length of 255 characters. You do not specify a length with TINYBLOB or TINYTEXT.

**MEDIUMBLOB or MEDIUMTEXT** − A BLOB or TEXT column with a maximum length of 16777215 characters. You do not specify a length with MEDIUMBLOB or MEDIUMTEXT.

**LONGBLOB or LONGTEXT** − A BLOB or TEXT column with a maximum length of 4294967295 characters. You do not specify a length with LONGBLOB or LONGTEXT.

**ENUM** − An enumeration, which is a fancy term for list. When defining an ENUM, you are creating a list of items from which the value must be selected (or it can be NULL). For example, if you wanted your field to contain "A" or "B" or "C", you would define your ENUM as ENUM ('A', 'B', 'C') and only those values (or NULL) could ever populate that field

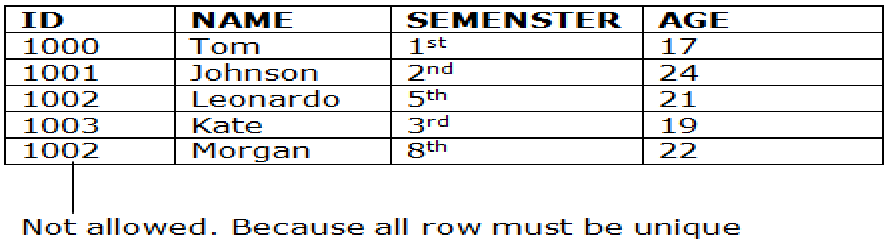
## 4. Key constraints

Keys are the entity set that is used to identify an entity within its entity set uniquely.

An entity set can have multiple keys, but out of which one key will be the primary key. A primary key can contain a unique and null value in the relational table.

Example:

Backward Skip 10sPlay VideoForward Skip 10s



| ID | NAME | SEMENSTER | AGE |
|---|---|---|---|
| 1000 | Tom | 1st | 17 |
| 1001 | Johnson | 2nd | 24 |
| 1002 | Leonardo | 5th | 21 |
| 1003 | Kate | 3rd | 19 |
| 1002 | Morgan | 8th | 22 |

Not allowed. Because all row must be unique

## KEY CONSTRAINTs

Constraints are the rules that we can apply on the type of data in a table. That is, we can specify the limit on the type of data that can be stored in a particular column in a table using constraints.

The available constraints in SQL are:

NOT NULL: This constraint tells that we cannot store a null value in a column. That is, if a column is specified as NOT NULL then we will not be able to store null in this particular column any more.

To create a NOT NULL constraint on the "Age" column when the "Persons" table is already created, use the following SQL:

Example

```
ALTER TABLE Persons
MODIFY Age int NOT NULL;
```

**UNIQUE**: This constraint when specified with a column, tells that all the values in the column must be unique. That is, the values in any row of a column must not be repeated.

The following SQL creates a UNIQUE constraint on the "ID" column when the "Persons" table is created:

```
CREATE TABLE Persons (
   ID int NOT NULL,
   LastName varchar(255) NOT NULL,
   FirstName varchar(255),
   Age int,
   UNIQUE (ID)
);
```

**PRIMARY KEY**: A primary key is a field which can uniquely identify each row in a table. And this constraint is used to specify a field in a table as primary key.

The following SQL creates a PRIMARY KEY on the "ID" column when the "Persons" table is created:

```
CREATE TABLE Persons (
   ID int NOT NULL,
   LastName varchar(255) NOT NULL,
   FirstName varchar(255),
   Age int,
   PRIMARY KEY (ID)
);
```

**FOREIGN KEY:** A Foreign key is a field which can uniquely identify each row in a another table. And this constraint is used to specify a field as Foreign key.

The following SQL creates a FOREIGN KEY on the "PersonID" column when the "Orders" table is created:

```
CREATE TABLE Orders (
   OrderID int NOT NULL,
   OrderNumber int NOT NULL,
   PersonID int,
   PRIMARY KEY (OrderID),
   FOREIGN KEY (PersonID) REFERENCES Persons(PersonID)
);
```

**CHECK:** This constraint helps to validate the values of a column to meet a particular condition. That is, it helps to ensure that the value stored in a column meets a specific condition.

The following SQL creates a CHECK constraint on the "Age" column when the "Persons" table is created. The CHECK constraint ensures that the age of a person must be 18, or older:

```
CREATE TABLE Persons (
   ID int NOT NULL,
   LastName varchar(255) NOT NULL,
   FirstName varchar(255),
   Age int,
   CHECK (Age>=18)
);
```

**DEFAULT:** This constraint specifies a default value for the column when no value is specified by the user.

The following SQL sets a DEFAULT value for the "City" column when the "Persons" table is created:

```
CREATE TABLE Persons (
   ID int NOT NULL,
   LastName varchar(255) NOT NULL,
   FirstName varchar(255),
   Age int,
   City varchar(255) DEFAULT 'Sandnes'
);
```

**SQL CREATE VIEW Statement**

In SQL, a view is a virtual table based on the result-set of an SQL statement.

A view contains rows and columns, just like a real table. The fields in a view are fields from one or more real tables in the database.

You can add SQL statements and functions to a view and present the data as if the data were coming from one single table.

A view is created with the CREATE VIEW statement.

CREATE VIEW Syntax

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

Note: A view always shows up-to-date data! The database engine recreates the view, every time a user queries it.

---

**SQL CREATE VIEW Examples**

The following SQL creates a view that shows all customers from Brazil:

ExampleGet your own SQL Server

```
CREATE VIEW [Brazil Customers] AS
SELECT CustomerName, ContactName
FROM Customers
WHERE Country = 'Brazil';
```

**MySQL CREATE VIEW Statement**

In SQL, a view is a virtual table based on the result-set of an SQL statement.

A view contains rows and columns, just like a real table. The fields in a view are fields from one or more real tables in the database.

You can add SQL statements and functions to a view and present the data as if the data were coming from one single table.

A view is created with the CREATE VIEW statement.

CREATE VIEW Syntax

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

Note: A view always shows up-to-date data! The database engine recreates the view, every time a user queries it.

---

**MySQL CREATE VIEW Examples**

The following SQL creates a view that shows all customers from Brazil:

ExampleGet your own SQL Server

```
CREATE VIEW [Brazil Customers] AS
SELECT CustomerName, ContactName
FROM Customers
WHERE Country = 'Brazil';
```

**<u>Subquery in SQL?</u>**

A subquery is a SQL query nested inside a larger query.

● A subquery may occur in : o - A SELECT clause o - A FROM clause o - A WHERE clause

● The subquery can be nested inside a SELECT, INSERT, UPDATE, or DELETE statement or inside another subquery.

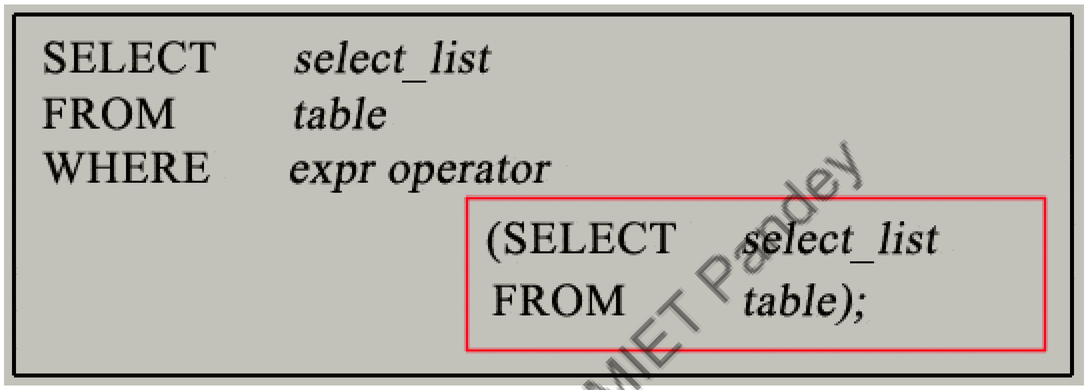
● A subquery is usually added within the WHERE Clause of another SQL SELECT statement

- You can use the comparison operators, such as >, <, or =. The comparison operator can also be a multiple-row operator, such as IN, ANY, or ALL.
- A subquery is also called an inner query or inner select, while the statement containing a subquery is also called an outer query or outer select.
- The inner query executes first before its parent query so that the results of an inner query can be passed to the outer query.

You can use a subquery in a SELECT, INSERT, DELETE, or UPDATE statement to perform the following tasks:

- Compare an expression to the result of the query.
- Determine if an expression is included in the results of the query.
- Check whether the query selects any rows.

**Syntax :**



```
SELECT    select_list
FROM      table
WHERE     expr operator
                      (SELECT   select_list
                       FROM     table);
```

- The subquery (inner query) executes once before the main query (outer query) executes.
- The main query (outer query) use the subquery result.

## SQL Aggregate Functions

SQL aggregate functions return a single value, calculated from values in a column.

Useful aggregate functions:

- AVG() - Returns the average value
- COUNT() - Returns the number of rows
- FIRST() - Returns the first value
- LAST() - Returns the last value
- MAX() - Returns the largest value
- MIN() - Returns the smallest value
- SUM() - Returns the sum

# SQL Queries (Insert, Delete, & Update Operations)

Creating a basic table involves naming the table and defining its columns and each column's data type.

The SQL **CREATE TABLE** statement is used to create a new table.

Syntax:

Basic syntax of CREATE TABLE statement is as follows:

```
CREATE TABLE table_name(
  column1 datatype,
  column2 datatype,
  column3 datatype,
  .....
  columnN datatype,
  PRIMARY KEY( one or more columns )
);
```

CREATE TABLE is the keyword telling the database system what you want to do. In this case, you want to create a new table. The unique name or identifier for the table follows the CREATE TABLE statement.

Then in brackets comes the list defining each column in the table and what sort of data type it is. The syntax becomes clearer with an example below.

A copy of an existing table can be created using a combination of the CREATE TABLE statement and the SELECT statement. You can check complete details atCreate Table Using another Table

**SQL - INSERT Query**

The SQL **INSERT INTO** Statement is used to add new rows of data to a table in the database.

Syntax:

There are two basic syntaxes of INSERT INTO statement as follows:

```
INSERT INTO TABLE_NAME (column1, column2, column3,...columnN)]
VALUES (value1, value2, value3,...valueN);
```

Here, column1, column2,...columnN are the names of the columns in the table into which you want to insert data.

You may not need to specify the column(s) name in the SQL query if you are adding values for all the columns of the table. But make sure the order of the values is in the same order as the columns in the table. The SQL INSERT INTO syntax would be as follows:

```
INSERT INTO TABLE_NAME VALUES (value1,value2,value3,...valueN);
```

Example:

Following statements would create six records in CUSTOMERS table:

```
INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (1, 'Ramesh', 32, 'Ahmedabad', 2000.00 );
```

## SQL - SELECT Query

SQL **SELECT** statement is used to fetch the data from a database table which returns data in the form of result table. These result tables are called result-sets.

Syntax:

The basic syntax of SELECT statement is as follows:

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

Here, column1, column2...are the fields of a table whose values you want to fetch. If you want to fetch all the fields available in the field, then you can use the following syntax:

```
SELECT * FROM table_name;
```

Example:

Consider the CUSTOMERS table having the following records:

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
|  7 | Muffy    |  24 | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
```

Following is an example, which would fetch ID, Name and Salary fields of the customers available in CUSTOMERS table:

```
SQL> SELECT ID, NAME, SALARY FROM CUSTOMERS;
```

This would produce the following result:

```
+----+----------+----------+
| ID | NAME     | SALARY   |
+----+----------+----------+
|  1 | Ramesh   |  2000.00 |
|  2 | Khilan   |  1500.00 |
|  3 | kaushik  |  2000.00 |
|  4 | Chaitali |  6500.00 |
|  5 | Hardik   |  8500.00 |
|  6 | Komal    |  4500.00 |
|  7 | Muffy    | 10000.00 |
+----+----------+----------+
```

If you want to fetch all the fields of CUSTOMERS table, then use the following query:

```
SQL> SELECT * FROM CUSTOMERS;
```

This would produce the following result:

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
|  7 | Muffy    |  24 | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
```

## SQL - WHERE Clause

The SQL **WHERE** clause is used to specify a condition while fetching the data from single table or joining with multiple tables.

If the given condition is satisfied then only it returns specific value from the table. You would use WHERE clause to filter the records and fetching only necessary records.

The WHERE clause is not only used in SELECT statement, but it is also used in UPDATE, DELETE statement, etc., which we would examine in subsequent chapters.

Syntax:

The basic syntax of SELECT statement with WHERE clause is as follows:

```
SELECT column1, column2, columnN
FROM table_name
WHERE [condition]
```

You can specify a condition using comparison or logical operators like >, <, =, LIKE, NOT, etc. Below examples would make this concept clear.

Example:

Consider the CUSTOMERS table having the following records:

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
|  7 | Muffy    |  24 | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
```

Following is an example which would fetch ID, Name and Salary fields from the CUSTOMERS table where salary is greater than 2000:

```
SQL> SELECT ID, NAME, SALARY
FROM CUSTOMERS
WHERE SALARY > 2000;
```

This would produce the following result:

```
+----+----------+----------+
| ID | NAME     | SALARY   |
+----+----------+----------+
|  4 | Chaitali |  6500.00 |
|  5 | Hardik   |  8500.00 |
|  6 | Komal    |  4500.00 |
|  7 | Muffy    | 10000.00 |
+----+----------+----------+
```

## SQL - UPDATE Query

The SQL **UPDATE** Query is used to modify the existing records in a table.

You can use WHERE clause with UPDATE query to update selected rows otherwise all the rows would be affected.

Syntax:

The basic syntax of UPDATE query with WHERE clause is as follows:

```
UPDATE table_name
SET column1 = value1, column2 = value2...., columnN = valueN
WHERE [condition];
```

You can combine N number of conditions using AND or OR operators.

Example:

Consider the CUSTOMERS table having the following records:

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
|  7 | Muffy    |  24 | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
```

Following is an example, which would update ADDRESS for a customer whose ID is 6:

```
SQL> UPDATE CUSTOMERS
SET ADDRESS = 'Pune'
WHERE ID = 6;
```

Now, CUSTOMERS table would have the following records:

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  6 | Komal    |  22 | Pune      |  4500.00 |
|  7 | Muffy    |  24 | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
```

If you want to modify all ADDRESS and SALARY column values in CUSTOMERS table, you do not need to use WHERE clause and UPDATE query would be as follows:

```
SQL> UPDATE CUSTOMERS
SET ADDRESS = 'Pune', SALARY = 1000.00;
```

Now, CUSTOMERS table would have the following records:

```
+----+----------+-----+---------+---------+
| ID | NAME     | AGE | ADDRESS | SALARY  |
+----+----------+-----+---------+---------+
|  1 | Ramesh   |  32 | Pune    | 1000.00 |
|  2 | Khilan   |  25 | Pune    | 1000.00 |
|  3 | kaushik  |  23 | Pune    | 1000.00 |
|  4 | Chaitali |  25 | Pune    | 1000.00 |
|  5 | Hardik   |  27 | Pune    | 1000.00 |
|  6 | Komal    |  22 | Pune    | 1000.00 |
|  7 | Muffy    |  24 | Pune    | 1000.00 |
+----+----------+-----+---------+---------+
```

# SQL - DELETE Query

The SQL **DELETE** Query is used to delete the existing records from a table.

You can use WHERE clause with DELETE query to delete selected rows, otherwise all the records would be deleted.

Syntax:

The basic syntax of DELETE query with WHERE clause is as follows:

```
DELETE FROM table_name
WHERE [condition];
```

You can combine N number of conditions using AND or OR operators.

Example:

Consider the CUSTOMERS table having the following records:

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
|  7 | Muffy    |  24 | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
```

Following is an example, which would DELETE a customer, whose ID is 6:

```
SQL> DELETE FROM CUSTOMERS
WHERE ID = 6;
```

Now, CUSTOMERS table would have the following records:

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  7 | Muffy    |  24 | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
```

If you want to DELETE all the records from CUSTOMERS table, you do not need to use WHERE clause and DELETE query would be as follows:

```
SQL> DELETE FROM CUSTOMERS;
```

Now, CUSTOMERS table would not have any record.

# DBMS - Joins

We understand the benefits of taking a Cartesian product of two relations, which gives us all the possible tuples that are paired together. But it might not be feasible for us in certain cases to take a Cartesian product where we encounter huge relations with thousands of tuples having a considerable large number of attributes.

**Join** is a combination of a Cartesian product followed by a selection process. A Join operation pairs two tuples from different relations, if and only if a given join condition is satisfied.

We will briefly describe various join types in the following sections.

## Theta (θ) Join

Theta join combines tuples from different relations provided they satisfy the theta condition. The join condition is denoted by the symbol **θ**.

### Notation

R1 $_{\theta}$ R2

R1 and R2 are relations having attributes (A1, A2, .., An) and (B1, B2,.. ,Bn) such that the attributes don't have anything in common, that is R1 ∩ R2 = Φ.

Theta join can use all kinds of comparison operators.

**Student**

| SID | Name | Std |
|---|---|---|
| 101 | Alex | 10 |
| 102 | Maria | 11 |

**Subjects**

| Class | Subject |
|---|---|
| 10 | Math |
| 10 | English |
| 11 | Music |
| 11 | Sports |

`Student_Detail –`

STUDENT $_{\text{Student.Std = Subject.Class}}$ SUBJECT

**Student_detail**

| SID | Name | Std | Class | Subject |
|---|---|---|---|---|
| 101 | Alex | 10 | 10 | Math |
| 101 | Alex | 10 | 10 | English |
| 102 | Maria | 11 | 11 | Music |
| 102 | Maria | 11 | 11 | Sports |

## Equijoin

When Theta join uses only **equality** comparison operator, it is said to be equijoin. The above example corresponds to equijoin.

## Natural Join (⋈)

Natural join does not use any comparison operator. It does not concatenate the way a Cartesian product does. We can perform a Natural Join only if there is at least one common attribute that exists between two relations. In addition, the attributes must have the same name and domain.

Natural join acts on those matching attributes where the values of attributes in both the relations are same.

**Courses**

| CID | Course | Dept |
|---|---|---|
| CS01 | Database | CS |
| ME01 | Mechanics | ME |
| EE01 | Electronics | EE |

**HoD**

| Dept | Head |
|---|---|
| CS | Alex |
| ME | Maya |
| EE | Mira |

**Courses HoD**

| Dept | CID | Course | Head |
|---|---|---|---|
| CS | CS01 | Database | Alex |
| ME | ME01 | Mechanics | Maya |
| EE | EE01 | Electronics | Mira |

## Outer Joins

Theta Join, Equijoin, and Natural Join are called inner joins. An inner join includes only those tuples with matching attributes and the rest are discarded in the resulting relation. Therefore, we need to use outer joins to include all the tuples from the participating relations in the resulting relation. There are three kinds of outer joins − left outer join, right outer join, and full outer join.

## Left Outer Join(R ⟕S)

All the tuples from the Left relation, R, are included in the resulting relation. If there are tuples in R without any matching tuple in the Right relation S, then the S-attributes of the resulting relation are made NULL.

**Left**

| **A** | **B** |
|---|---|
| 100 | Database |
| 101 | Mechanics |
| 102 | Electronics |

**Right**

| **A** | **B** |
|---|---|
| 100 | Alex |
| 102 | Maya |
| 104 | Mira |

**Courses ⟕HoD**

| **A** | **B** | **C** | **D** |
|---|---|---|---|
| 100 | Database | 100 | Alex |
| 101 | Mechanics | --- | --- |
| 102 | Electronics | 102 | Maya |

## Right Outer Join: ( R ⟖S )

All the tuples from the Right relation, S, are included in the resulting relation. If there are tuples in S without any matching tuple in R, then the R-attributes of resulting relation are made NULL.

**Courses ⟖HoD**

| **A** | **B** | **C** | **D** |
|---|---|---|---|
| 100 | Database | 100 | Alex |
| 102 | Electronics | 102 | Maya |
| --- | --- | 104 | Mira |

## Full Outer Join: ( R ⟗S)

All the tuples from both participating relations are included in the resulting relation. If there are no matching tuples for both relations, their respective unmatched attributes are made NULL.

**Courses ⟗ HoD**

| A | B | C | D |
|---|---|---|---|
| 100 | Database | 100 | Alex |
| 101 | Mechanics | --- | --- |
| 102 | Electronics | 102 | Maya |
| --- | --- | 104 | Mira |

## UNION operator

In SQL the **UNION** clause combines the results of two SQL queries into a single table of all matching rows. The two queries must result in the same number ofcolumns and compatible data types in order to unite. Any duplicate records are automatically removed unless UNION ALL is used.

UNION can be useful in data warehouse applications where tables aren't perfectly normalized. A simple example would be a database having tables sales2005and sales2006 that have identical structures but are separated because of performance considerations. A UNION query could combine results from both tables.

Note that UNION does not guarantee the order of rows. Rows from the second operand may appear before, after, or mixed with rows from the first operand. In situations where a specific order is desired, ORDER BY must be used.

Note that UNION ALL may be much faster than plain UNION.

### Examples[edit]

Given these two tables:

**sales2005**

| person | amount |
|---|---|
| Jeno | 1000 |
| Alex | 2000 |
| Bob | 5000 |

**sales2006**

| person | amount |
|---|---|
| Joe | 2000 |
| Alex | 2000 |
| Zach | 35000 |

Executing this statement:

```
SELECT * FROM sales2005
UNION
SELECT * FROM sales2006;
```

yields this result set, though the order of the rows can vary because no ORDER BY clause was supplied:

| person | amount |
|---|---|

| | |
|---|---|
| Joe | 1000 |
| Alex | 2000 |
| Bob | 5000 |
| Joe | 2000 |
| Zach | 35000 |

Note that there are two rows for Joe because those rows are distinct across their columns. There is only one row for Alex because those rows are not distinct for both columns.

UNION ALL gives different results, because it will not eliminate duplicates. Executing this statement:

```
SELECT * FROM sales2005
UNION ALL
SELECT * FROM sales2006;
```

would give these results, again allowing variance for the lack of an ORDER BY statement:

| person | amount |
|---|---|
| Joe | 1000 |
| Joe | 2000 |
| Alex | 2000 |
| Alex | 2000 |
| Bob | 5000 |
| Zach | 35000 |

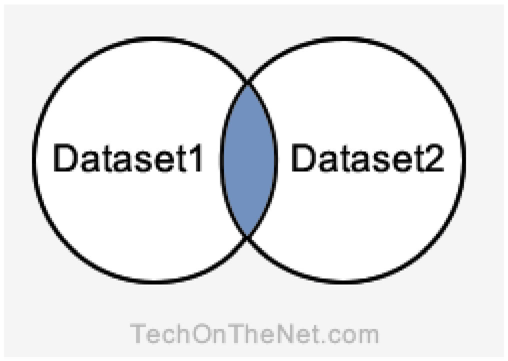
The discussion of full outer joins also has an example that uses UNIOn

## SQL INTERSECT

**DESCRIPTION**

The SQL INTERSECT operator is used to return the results of 2 or more SELECT statements. However, it only returns the rows selected by all queries or data sets. If a record exists in one query and not in the other, it will be omitted from the INTERSECT results.

## Intersect Query



**Explanation:** The INTERSECT query will return the records in the blue shaded area. These are the records that exist in both Dataset1 and Dataset2.

Each SQL statement within the SQL INTERSECT must have the same number of fields in the result sets with similar data types

**SQL INTERSECT** is query that allows you to select related information from 2 tables, this is combine 2 SELECT statement into 1 and display it out.

**INTERSECT** produces rows that appear in both queries.that means **INTERSECT** command acts as an **AND**operator (value is selected only if it appears in both statements).

The syntax is as follows:

```
SELECT [COLUMN NAME 1], [COLUMN NAME 2],… FROM [TABLE NAME 1]
INTERSECT
SELECT [COLUMN NAME 1], [COLUMN NAME 2],… FROM [TABLE NAME 2]
```

**EXAMPLE :**

We have 2 table name GamesScores, GameScores_new.

Table ***GameScores***

| **PlayerName** | **Department** | **Scores** |
|---|---|---|
| Jason | IT | 3000 |
| Irene | IT | 1500 |
| Jane | Marketing | 1000 |
| David | Marketing | 2500 |
| Paul | HR | 2000 |
| James | HR | 2000 |

Table ***GameScores_new***

| **PlayerName** | **Department** | **Scores** |
|---|---|---|
| Jason | IT | 3000 |
| David | Marketing | 2500 |
| Paul | HR | 2000 |
| James | HR | 2000 |

**SQL statement :**

```
SELECT PlayerName FROM GameScores
INTERSECT
SELECT PlayerName FROM GameScores_new
```

***Result:***

| PlayerName |
|---|
| David |
| James |
| Jason |
| Paul |

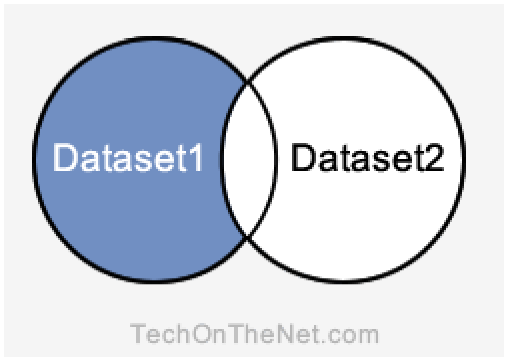
**SQL: MINUS OPERATOR**

This SQL tutorial explains how to use the SQL **MINUS operator** with syntax and examples.

**DESCRIPTION**

The SQL MINUS operator is used to return all rows in the first SELECT statement that are not returned by the second SELECT statement. Each SELECT statement will define a dataset. The MINUS operator will retrieve all records from the first dataset and then remove from the results all records from the second dataset.

## Minus Query



**Explanation:** The MINUS query will return the records in the blue shaded area. These are the records that exist in Dataset1 and not in Dataset2.

Each SELECT statement within the MINUS query must have the same number of fields in the result sets with similar data types.

The MINUS operator is not supported in all SQL databases. It can used in databases such as Oracle.

For databases such as SQL Server, PostgreSQL, and SQLite, use the EXCEPT operator to perform this type of query.

**SYNTAX**

The syntax for the SQL MINUS operator is:

```
SELECT expression1, expression2, ... expression_n

FROM tables

WHERE conditions

MINUS

SELECT expression1, expression2, ... expression_n

FROM tables
```

```
WHERE conditions;
```

## Parameters or Arguments

**expression1, expression2, expression_n**

The columns or calculations that you wish to retrieve.

**tables**

The tables that you wish to retrieve records from. There must be at least one table listed in the FROM clause.

**conditions**

These are conditions that must be met for the records to be selected.

**Note:**

- There must be same number of expressions in both SELECT statements.
- The corresponding expressions must have the same data type in the SELECT statements. For example:*expression1* must be the same data type in both the first and second SELECT statement.

The **MINUS** command operates on two SQL statements. It takes all the results from the first SQL statement, and then subtract out the ones that are present in the second SQL statement to get the final answer. If the second SQL statement includes results not present in the first SQL statement, such results are ignored.

The syntax is as follows:

**[SQL Statement 1]**
**MINUS**
**[SQL Statement 2];**

Let's continue with the same example:

Table ***Store_Information***

| | | |
|---|---|---|
| Los Angeles | 1500 | Jan-05-1999 |
| San Diego | 250 | Jan-07-1999 |
| Los Angeles | 300 | Jan-08-1999 |
| Boston | 700 | Jan-08-1999 |

Table ***Internet_Sales***

| | |
|---|---|
| Jan-07-1999 | 250 |
| Jan-10-1999 | 535 |
| Jan-11-1999 | 320 |
| Jan-12-1999 | 750 |

and we want to find out all the dates where there are store sales, but no internet sales. To do so, we use the following SQL statement:

**SELECT Txn_Date FROM Store_Information**
**MINUS**
**SELECT Txn_Date FROM Internet_Sales;**

Result:

**Txn_Date**

**Jan-05-1999**

**Jan-08-1999**

'Jan-05-1999', 'Jan-07-1999';, and 'Jan-08-1999' are the distinct values returned from **SELECT Txn_Date FROM Store_Information**. 'Jan-07-1999' is also returned from the second SQL statement, **SELECT Txn_Date FROM Internet_Sales**, so it is excluded from the final result set.

Please note that the **MINUS** command will only return distinct values.

Some databases may use **EXCEPT** instead of **MINUS**. Please check the documentation for your specific database for the correct usage.

## Cursor in PL/SQL :

A cursor can be basically referred to as a pointer to the context area.Context area is a memory area that is created by Oracle when SQL statement is processed.The cursor is thus responsible for holding the rows that have been returned by a SQL statement.Thus the PL/SQL controls the context area by the help of cursor.An Active set is basically the set of rows that the cursor holds. The cursor can be of two types: Implicit Cursor, and Explicit Cursor.

### Advantages of Cursor:

- They are helpful in performing the row by row processing and also row wise validation on each row.
- Better concurrency control can be achieved by using cursors.
- Cursors are faster than while loops.

### Disadvantages of Cursor:

- They use more resources each time and thus may result in network round trip.
- More number of network round trips can degrade the performance and reduce the speed.

## 2. Trigger in PL/SQL :

A Trigger is basically a program which gets automatically executed in response to some events such as modification in the database.Some of the events for their execution are DDL statement, DML statement or any Database operation.Triggers are thus stored within the database and come into action when specific conditions match.Hence, they can be defined on any schema, table, view etc. There are six types of triggers: BEFORE INSERT, AFTER INSERT, BEFORE UPDATE, AFTER UPDATE, BEFORE DELETE, and AFTER DELETE.

## Advantages of Trigger:

- They are helpful in keeping the track of all the changes within the database.
- They also help in maintaining the integrity constraints.

**Disadvantages of Trigger:**

- They are very difficult to view which makes the debugging also difficult.
- Too much use of the triggers or writing complex codes within a trigger can slow down the performance.

**Difference between Cursor and Trigger:**

| S.NO | CURSOR | TRIGGER |
|---|---|---|
| 1. | It is a pointer which is used to control the context area and also to go through the records in the database. | It is a program which gets executed in response to occurrence of some events. |
| 2. | A cursor can be created within a trigger by writing the declare statement inside the trigger. | A trigger cannot be created within a cursor. |
| 3. | It gets created in response to execution of SQL statement thus it is not previously stored. | It is a previously stored program. |
| 4. | The main function of the cursor is retrieval of rows from the result set | The main function of trigger is to maintain the integrity of the |

| | | |
|---|---|---|
| | one at a time (row by row). | database. |
| 5. | A cursor is activated and thus created in response to any SQL statement. | A trigger is executed in response to a DDL statement, DML statement or any database operation. |
| 6. | The main disadvantage of cursor is that it uses more resources each time and thus results in network round trip. | The main disadvantage of trigger is that they are hard to view which makes the debugging really difficult. |

## PL/SQL Procedure

The PL/SQL stored procedure or simply a procedure is a PL/SQL block which performs one or more specific tasks. It is just like procedures in other programming languages.

The procedure contains a header and a body.

- **Header:** The header contains the name of the procedure and the parameters or variables passed to the procedure.
- **Body:** The body contains a declaration section, execution section and exception section similar to a general PL/SQL block.

## How to pass parameters in procedure:

When you want to create a procedure or function, you have to define parameters .There is three ways to pass parameters in procedure:

1. **IN parameters:** The IN parameter can be referenced by the procedure or function. The value of the parameter cannot be overwritten by the procedure or the function.
2. **OUT parameters:** The OUT parameter cannot be referenced by the procedure or function, but the value of the parameter can be overwritten by the procedure or function.
3. **INOUT parameters:** The INOUT parameter can be referenced by the procedure or function and the value of the parameter can be overwritten by the procedure or function.

## PL/SQL Create Procedure

**Syntax for creating procedure:**

```
CREATE [OR REPLACE] PROCEDURE procedure_name
    [ (parameter [,parameter]) ]
IS
    [declaration_section]
BEGIN
    executable_section
```

```
[EXCEPTION
   exception_section]
END [procedure_name];
```

## Create procedure example

In this example, we are going to insert record in user table. So you need to create user table first.

**Table creation:**

```
create table user(id number(10) primary key,name varchar2(100));
```

Now write the procedure code to insert record in user table.

**Procedure Code:**

```
create or replace procedure "INSERTUSER"
(id IN NUMBER,
name IN VARCHAR2)
is
begin
insert into user values(id,name);
end;
/
```

Output:

```
Procedure created.
```

**Mapping constraints:** Mapping constraints of mapping cardinalities or cardinality ratio, express the number of entities to which another entity can be associated via a relationship set. Mapping cardinalities are most useful in describing binary relationship sets that involve more than two entity sets. For a binary relationship set R between entities sets A and B the mapping cardinality must be one of the following:

**One to One:** An entity in A is associated with at most one entity in B and an entity in B is associated with at most one entity in A(see figure). **One to Many:** An entity in A is associated with any number (Zero to more) of entities in B. An entity in B, however can be associated with at most one entity in A (see figure).**Many to One:** An entity in A is associated with at most one entity in B. An entity in B, however, can be associated with any number (zero of more) of entity in A (see figure) **Many to Many :** An entity in A is associated with any number of entities in B, and an entity cardinality for a particular relationship set obviously depends on the real word situation that the relationship set is modeling. As an illustration, consider the borrower relationship set. If in a particular bank a loan can belong to only one customer and customer can have several loans, then the relationship set from customer to loan is one to many. If a loan can belong to several customers the relationship set is many to many.

## Mapping Cardinality

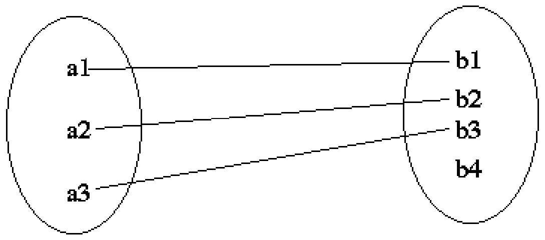
A **mapping cardinality** is a data constraint that specifies how many entities an entity can be related to in a relationship set.

Example: A student can only work on two projects, the number of students that work on one project is not limited.

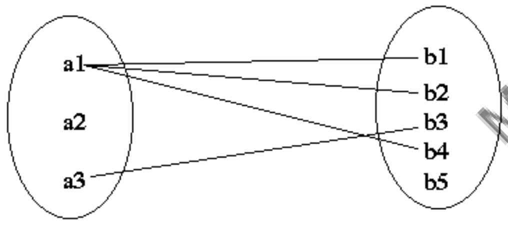
A **binary relationship set** is a relationship set on two entity sets. Mapping cardinalities on binary relationship sets are simplest.

Consider a binary relationship set R on entity sets A and B. There are four possible mapping cardinalities in this case:

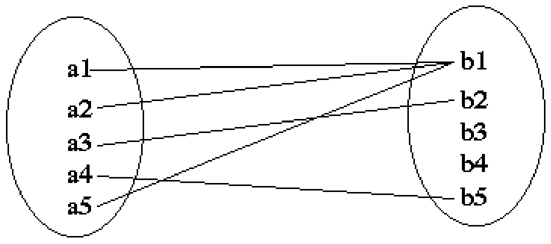
1. **one-to-one** - an entity in A is related to at most one entity in B, and an entity in B is related to at most one entity in A.



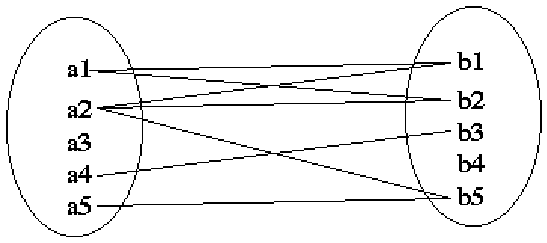
2. **one-to-many** - an entity in A is related to any number of entities in B, but an entity in B is related to at most one entity in A.



3. **many-to-one** - an entity in A is related to at most one entity in B, but an entity in B is related to any number of entities in A.



4. **many-to-many** - an entity in A is related to any number of entities in B, but an entity in B is related to any number of entities in A.


a1
a2
a3
a4
a5
b1
b2
b3
b4
b5