Unit – III

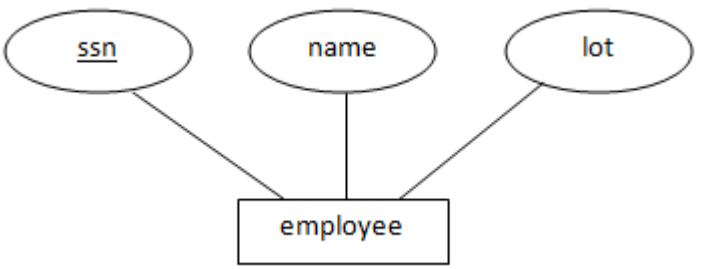
Entity Relationship Model: Introduction, Representation of entities, attributes, entity set, relationship, relationship set, constraints, sub classes, super class, inheritance, specialization, generalization using ER Diagrams.

SQL : Creating tables with relationship, implementation of key and integrity constraints, nested queries, sub queries, grouping, aggregation, ordering, implementation of different types of joins, view(updatable and non-updatable), relational set operations.

INTRODUCTION:

Entity: An entity is something which is described in the database by storing its data, it may be a concrete entity a conceptual entity.

Entity set: An entity set is a collection of similar entities. The Employees entity set with attributes ssn, name, and lot is shown in the following figure.



Attribute: An attribute describes a property associated with entities. Attribute will have a name and a value for each entity.

Domain: A domain defines a set of permitted values for a attribute

Entity Relationship Model: An ERM is a theoretical and conceptual way of showing data **relationships** in software development. It is a database **modeling** technique that generates an abstract diagram or visual representation of a system's data that can be helpful in designing a relational database.

ER model allows us to describe the data involved in a real-world enterprise in terms of objects and their relationships and is widely used to develop an initial database design.

REPRESENTATION:

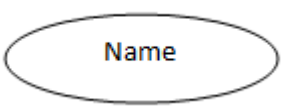
1. ENTITIES:

Entities are represented by using rectangular boxes. These are named with the entity name that they represent.

Fig: Student and Employee entities

2. ATTRIBUTES:

Attributes are the properties of entities. Attributes are represented by means of ellipses. Every ellipse represents one attribute and is directly connected to its entity.

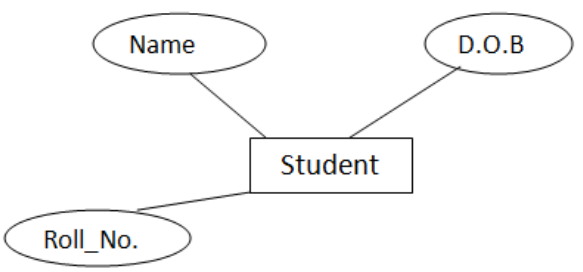


Types of attributes:

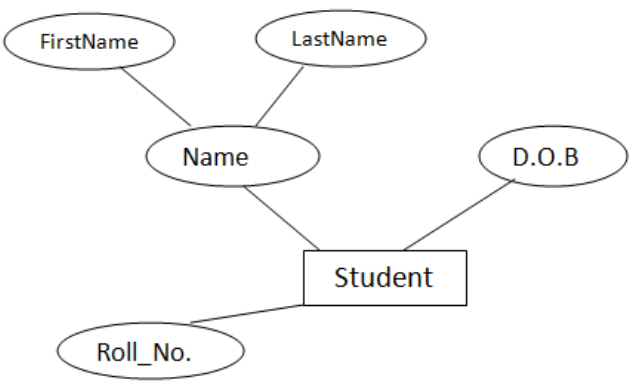
- **Simple attribute** − Simple attributes are atomic values, which cannot be divided further. For example, a student's phone number is an atomic value of 10 digits.
- **Composite attribute** − Composite attributes are made of more than one simple attribute. For example, a student's complete name may have first_name and last_name.
- **Derived attribute** − Derived attributes are the attributes that do not exist in the physical database, but their values are derived from other attributes present in the database. For example, average_salary in a department should not be saved directly in the database, instead it can be derived. For another example, age can be derived from data_of_birth.
- **Single-value attribute** − Single-value attributes contain single value. For example − Social_Security_Number.
- **Multi-value attribute** − Multi-value attributes may contain more than one values. For example, a person can have more than one phone number, email_address, etc.

ER Representation for Attributes:

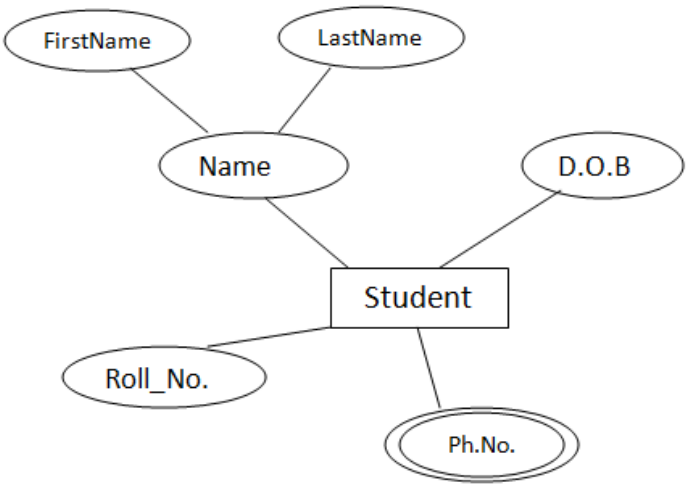
Attributes are the properties of entities. Attributes are represented by means of ellipses. Every ellipse represents one attribute and is directly connected to its entity (rectangle).



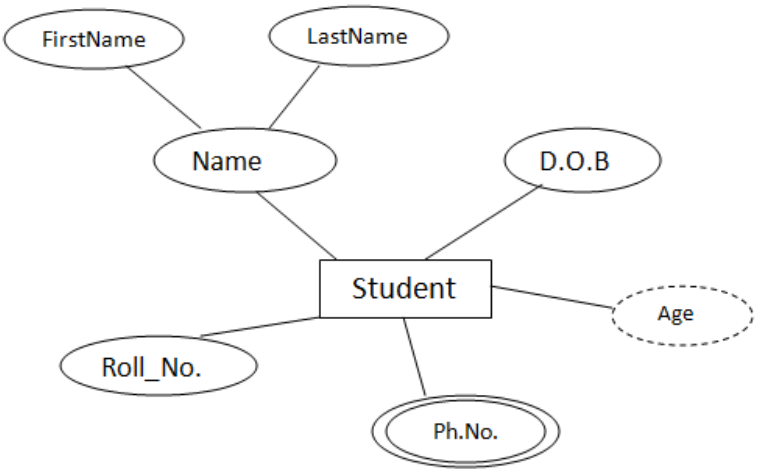
If the attributes are **composite**, they are further divided in a tree like structure. Every node is then connected to its attribute. That is, composite attributes are represented by ellipses that are connected with an ellipse.



Multi valued attributes are depicted by double ellipse.



Derived attributes are depicted by dashed ellipse.



3. RELATIONSHIP:

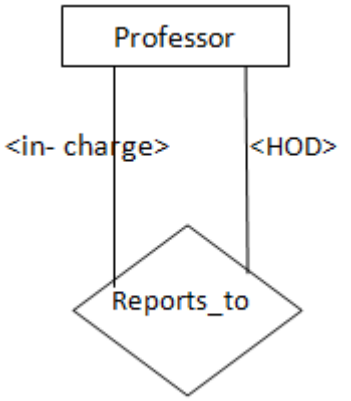
Relationships are represented by diamond-shaped box. Name of the relationship is written inside the diamond-box. All the entities (rectangles) participating in a relationship, are connected to it by a line.

Types of relationships:

Degree of Relationship is the number of participating entities in a relationship defines the degree of the relationship. Based on degree the relationships are categorized as

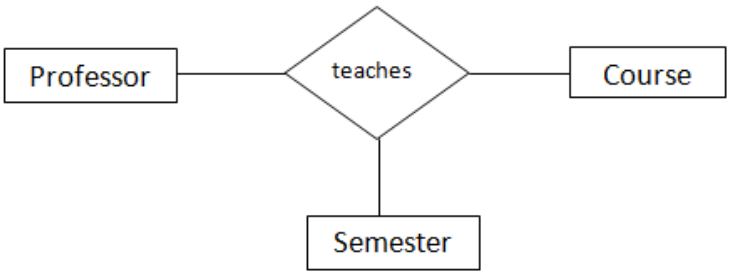
- Unary = degree 1
- Binary = degree 2
- Ternary = degree 3
- n-ary = degree

Unary Relationship: A relationship with one entity set. It is like a relationship among 2 entities of same entity set. Example: A professor (in-charge) reports to another professor (Head Of the Dept).

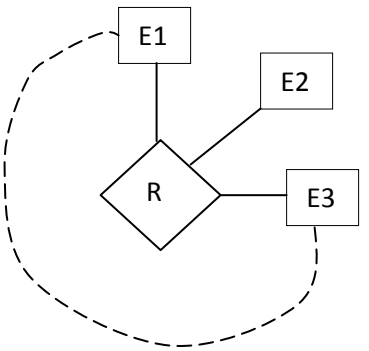


Binary Relationship: A relationship among 2 entity sets. Example: A professor teaches a course and a course is taught by a professor.

Ternary Relationship: A relationship among 3 entity sets. Example: A professor teaches a course in so and so semester.

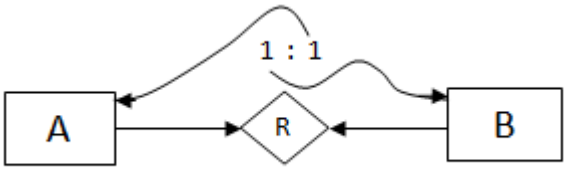


n-array Relationship: A relationship among n entity sets.



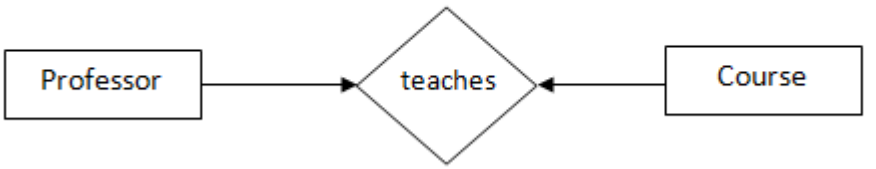
Cardinality defines the number of entities in one entity set, which can be associated with the number of entities of other set via relationship set. Cardinality ratios are categorized into 4. They are.

1. **One-to-One relationship:** When only one instance of an entities are associated with the relationship, then the relationship is *one-to-one relationship.* Each entity in A is associated with at most one entity in B and each entity in B is associated with at most one entity in A.

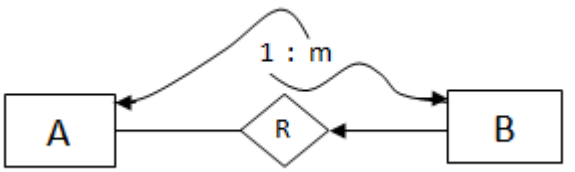


Each professor teaches one course and each course is taught by one professor.

1 : 1

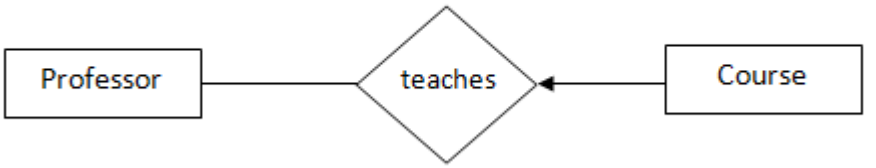


2. **One-to-many relationship**: When more than one instance of an entity is associated with a relationship, then the relationship is *one-to-many relationship.* Each entity in A is associated with zero or more entities in B and each entity in B is associated with at most one entity in A.

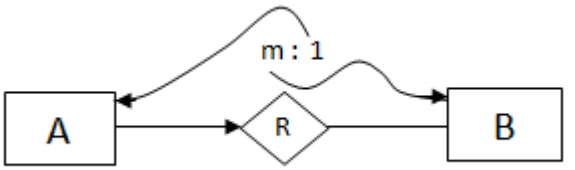


Each professor teaches 0 (or) more courses and each course is taught by at most one professor.

1 : m

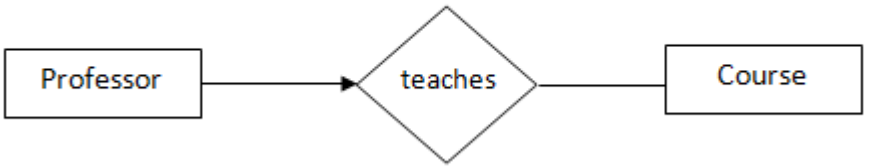


3. **Many-to-one relationship**: When more than one instance of entity is associated with the relationship, then the relationship is *many-to-one relationship.* Each entity in A is associated with at most one entity in B and each entity in B is associated with 0 (or) more entities in A.

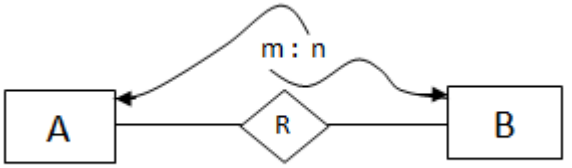


Each professor teaches at most one course and each course is taught by 0 (or) more professors.

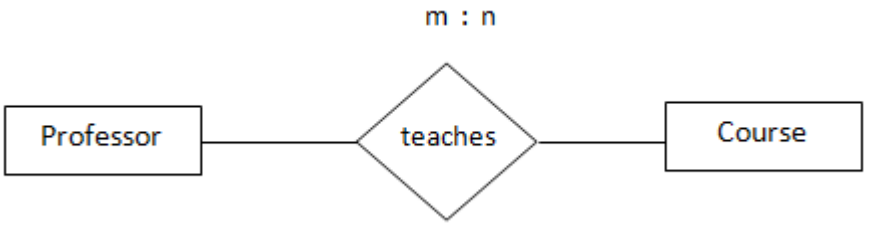
m : 1



4. **Many-to-Many relationship**: If more than one instance of an entity on the left and more than one instance of an entity on the right can be associated with the relationship, then it depicts many-to-many relationship. Each entity in A is associated with 0 (or) more entities in B and each entity in B is associated with 0 (or) more entities in A.



Each professor teaches 0 (or) more courses and each course is taught by 0 (or) more professors.

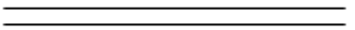


4. RELATIONSHIP SET:

A set of relationships of similar type is called a relationship set. Like entities, a relationship too can have attributes. These attributes are called **descriptive attributes**.

PARTICIPATION CONSTRAINTS:

- **Total Participation** − If Each entity in the entity set is involved in the relationship then the participation of the entity set is said to be total. Total participation is represented by double lines.

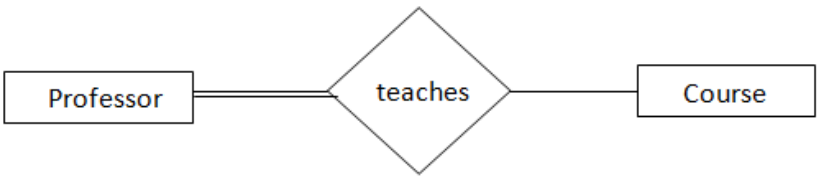


- **Partial participation** − If, Not all entities of the entity set are involved in the relationship then such a participation is said to be partial. Partial participation is represented by single lines.

Example: Participation Constraints can be explained easily with some examples. They are as follows.

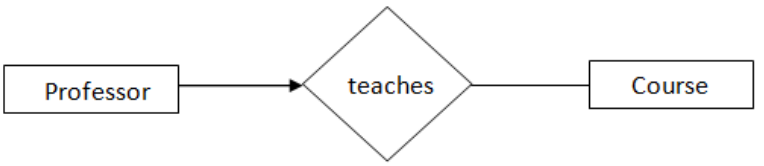
1.Each Professor teaches at least one course.

min=1 (Total Participation)
max=many (No key)



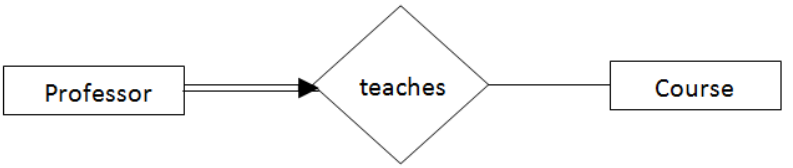
2. Each Professor teaches at most one course.

min=0 (Partial Participation)
max=many (Key)



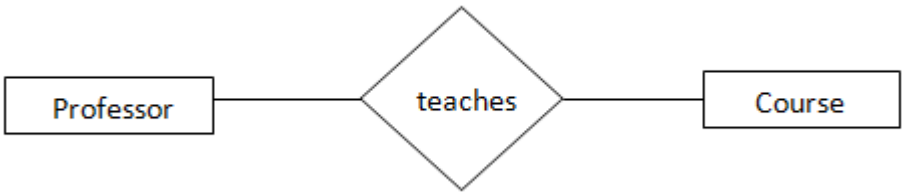
3. Each Professor teaches Exactly one course.

min=1 (Total Participation)
max=1 (Key)



4. Each Professor teaches course.

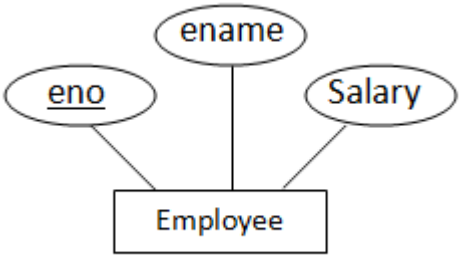
min=0 (Partial Participation)
max=many (no Key)



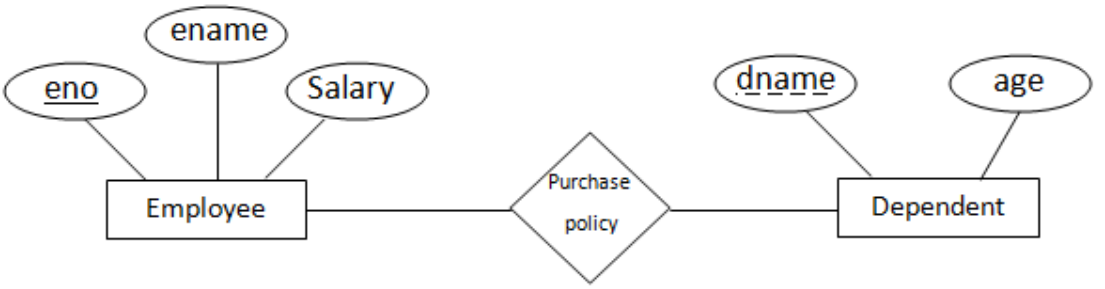
Note: Partial Participation is the default participation.

STRONG AND WEAK ENTITY SETS:

Strong Entity set: If each entity in the entity set is distinguishable or it has a key then such an entity set is known as strong entity set.



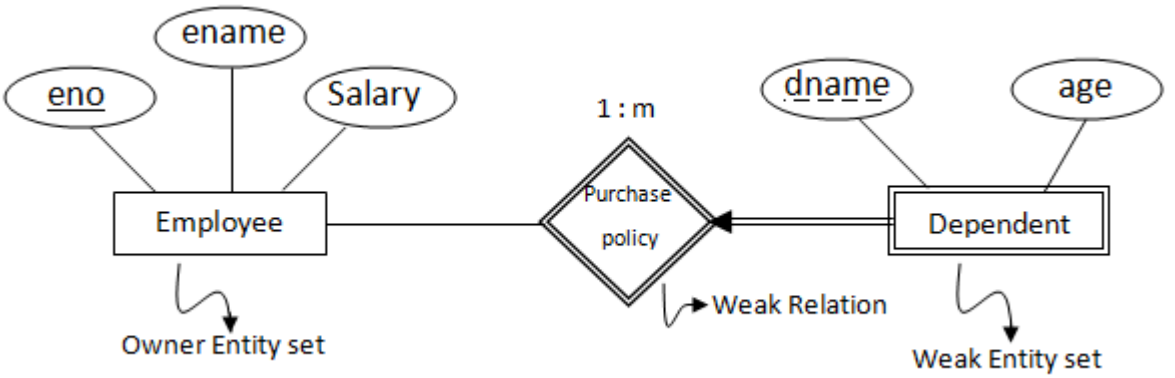
Weak Entity set: If each entity in the entity set is not distinguishable or it doesn't has a key then such an entity set is known as strong entity set.



eno is key so it is represented by solid underline. dname is partial key. It can't distinguish the tuples in the Dependent entity set. so dname is represented by dashed underline.

Weak entity set is always in total participation with the relation. If entity set is weak then the relationship is also known as weak relationship, since the dependent relation is no longer needed when the owner left.

Ex: policy dependent details are not needed when the owner (employee) of that policy left or fired from the company or expired. The detailed ER Diagram is as follows.



The cardinality of the owner entity set is with weak relationship is 1 : m. Weak entity set is uniquely identifiable by partial key and key of the owner entity set.

Dependent entity set is key to the relation because the all the tuples of weak entity set are associated with the owner entity set tuples.

GENERALIZATION AND SPECIALIZATION

One entity type might be a subtype of another, very similar to subclasses in OO programming. Relationship that is existed between these entities is known as IsA relationship. The two entities related by IsA are always descriptions of the same real-world object. These are typically used in databases to be implemented as Object Oriented Models. The upper entity type is the more abstract entity type (super type) from which the lower entities inherit its attributes.

Properties of Is A:

Inheritance:

- All attributes of the super type apply to the subtype.
- The subtype inherits all attributes of its super type.
- The key of the super type is also the key of the subtype.

Transitivity:

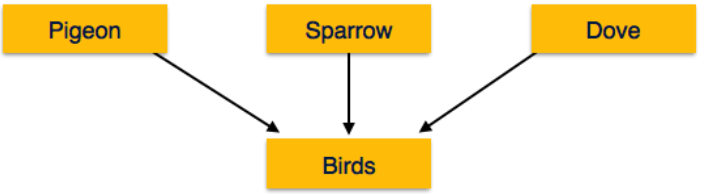
- This property creates a hierarchy of IsA relationships.

Advantages:

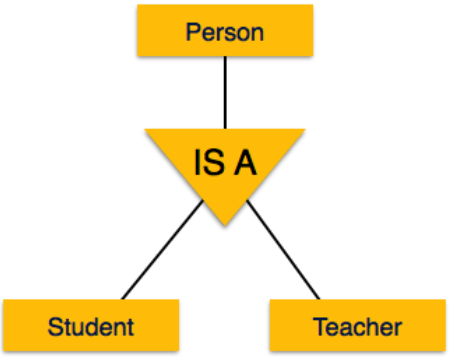
- Used to create a more concise and readable E-R diagram.
- It best maps to object oriented approaches either to databases or related applications.
- Attributes common to different entity sets need not be repeated.
- They can be grouped in one place as attributes of the supertype.
- Attributes of (sibling) subtypes are likely to be different.

The process of sub grouping with in a entity set is known as specialization or generalization. Specialization follows top down approach and generalization follows bottom-up approach. Both the speculation and generalization are depicted using a triangle component labeled as IS A.

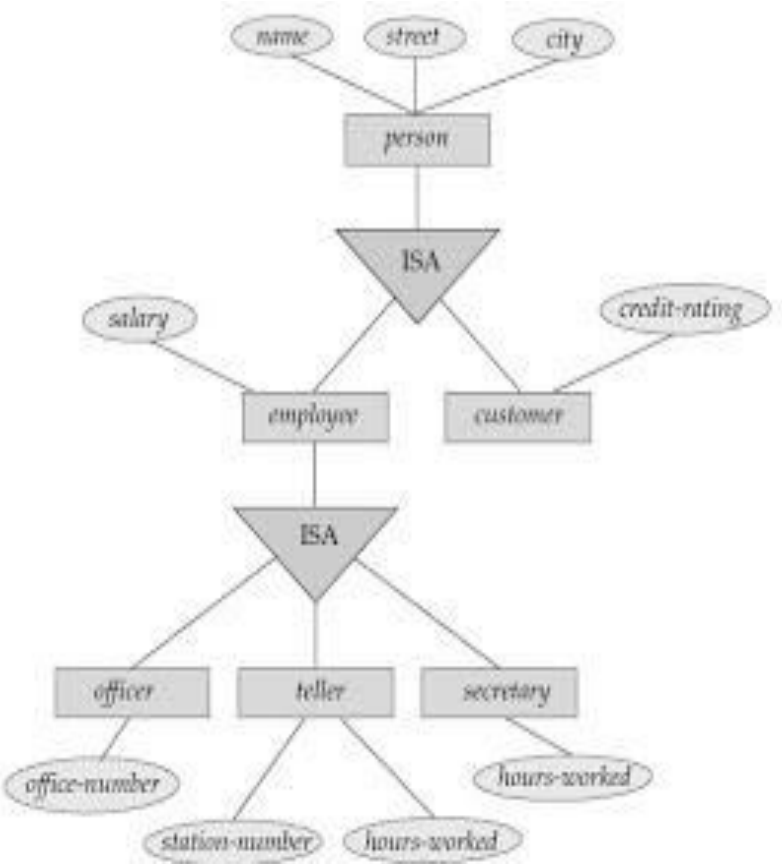
Generalization: is a bottom-up approach in which two lower level entities combine to form a higher level entity. In generalization, the higher level entity can also combine with other lower level entity to make further higher level entity. In generalization, a number of entities are brought together into one generalized entity based on their similar characteristics. For example, pigeon, house sparrow, crow and dove can all be generalized as Birds.



Specialization: is opposite to Generalization. It is a top-down approach in which one higher level entity can be broken down into two lower level entity. In specialization, some higher level entities may not have lower-level entity sets at all. In specialization, a group of entities is divided into sub-groups based on their characteristics. Take a group 'Person' for example. A person has name, date of birth, gender, etc. These properties are common in all persons, human beings. But in a company, persons can be identified as employee, employer, customer, or vendor, based on what role they play in the company.



Inheritance: We use all the above features of ER-Model in order to create classes of objects in object-oriented programming. The details of entities are generally hidden from the user; this process known as **abstraction**. Inheritance is an important feature of Generalization and Specialization. It allows lower-level entities to inherit the attributes of higher-level entities.



Attribute inheritance is a crucial property where a subclass entity set inherits all the attributes of its super class entity set. Attributes can be additionally specified which is used to give a clear representation though that same attribute is found nowhere in the hierarchy.

Employee and customer can inherit the attributes of Person entity and they have their own attributes like salary for employee and credit_rating for customer. similarly, the entities officer, teller and secretary inherit alll the attributes of employee and they can have their own attributes like office_member for officer, station_number & hours_worked for teller and hours_worked for secretary.

If an entity set has one single higher level entity set then it is termed as single inheritance. If it has multiple higher level entity sets then we can term it as multiple inheritance.

Constraints in Class Hierarchies:

Constraints that can be applied for Class Hierarchies are:

1. Condition Constraints
2. User Defined Constraints

A **Condition Defined Constraint** is imposed, while classifying the entities of a higher level entity set to be part of (or) a member of lower level entity sets based on a specified defined constraints.

Example: Every higher level entity in the entity set "Account" is checked using the attribute "acc_type" to be assigned either to the "SavingsAccount" or to the "CurrentAccount". SavingsAccount and CurrentAccount are lower level entity sets.

If no condition is specified during the process of designing the lower level entity sets, then it is called **user defined constraint.**

Disjoint Constraint: This constraint checks whether an entity belongs to only one lower level entity set or not.

Overlapping Constraint: This constraint ensures by testing out that an entity in the higher level entity set belong to more than one lower level entity sets.

Completeness Constraint: This is also called total constraint which specifies whether or not and entity in the higher level entity set must belong if at least one lower level entity set in generalization or specialization.

When we consider the completeness constraint, we come across total and partial constraints. i.e., Total Participation constraint and Partial Participation Constraint.

Total Participation forces that a higher level entity set 's entity(Every entity) must belong to at least one lower level entity set mandatorily.

Ex: An account entity set's entity set must be belong to either savings account entity set or current account entity set.

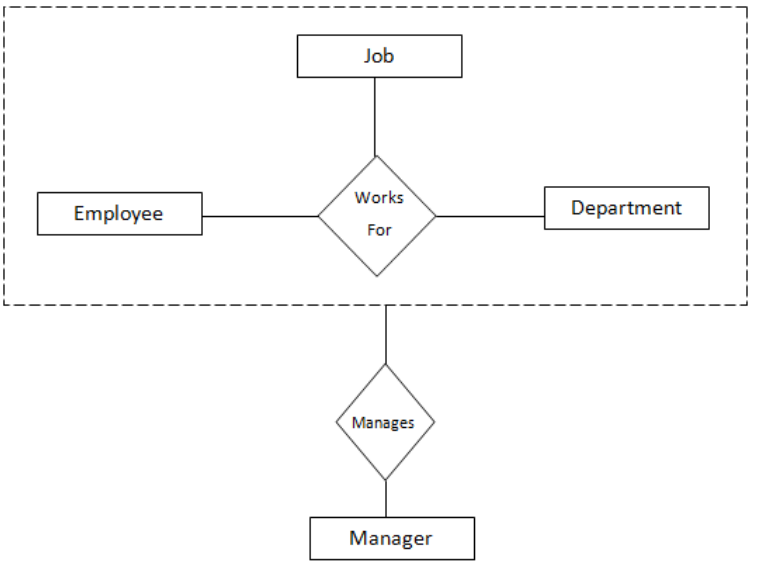
Partial Participation is rarely found with an entity set because sometimes an entity set in the higher level entity set beside being a member of that higher level entity set, doesn't belong to any of the lower level entity sets immediately until the stipulated period.

Ex: A new employer listed in the higher level entity set but not designated to any one of the available teams that belong to the lower level entity set.

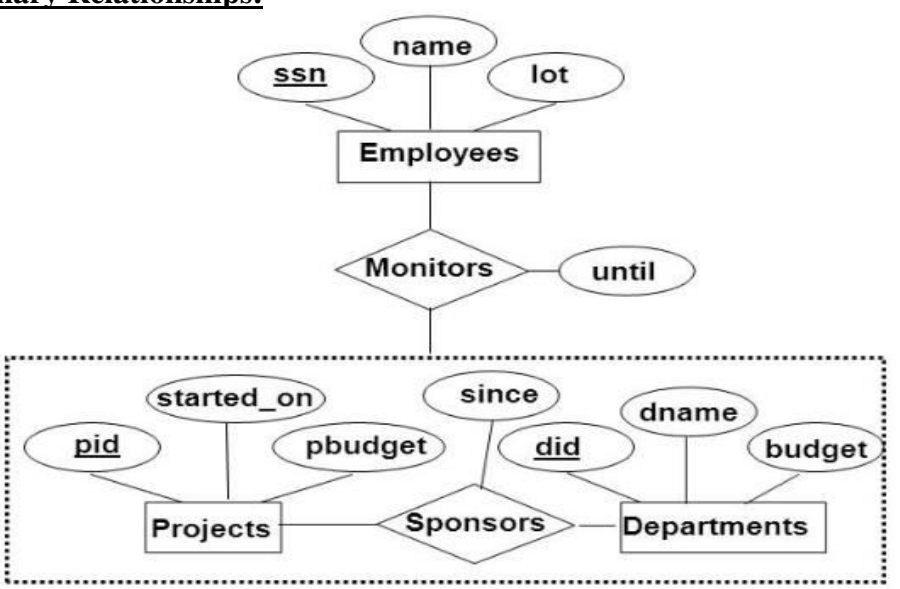
AGGREGATION:

An aggregation is not a ternary relationship but is an attempt to establish the relationship with another relationship set. It is also termed as relationship with in a relationship. Aggregation can be used over a binary, ternary or a quaternary relationship set. Aggregation is denoted using a dashed rectangle.

Aggregation over ternary relationship:



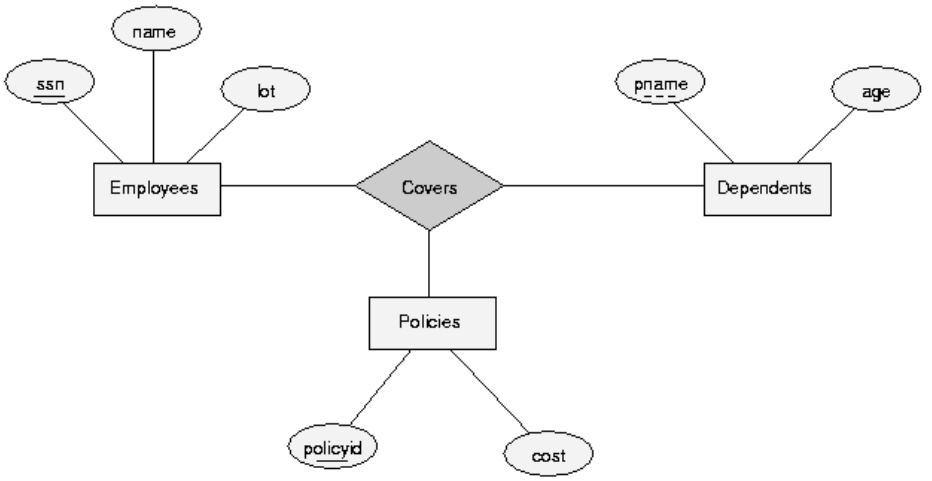
Aggregation over binary Relationships:



In the examples shown above, we treated the already existed relationship sets "WorksFor" and "Sponsors" as an entity set for defining the new relationship sets "Manages" and "Monitors". A relationship set is participating in another relationship. So it can be termed as aggregation.

TERNARY RELATIONSHIP DECOMPOSED INTO BINARY:

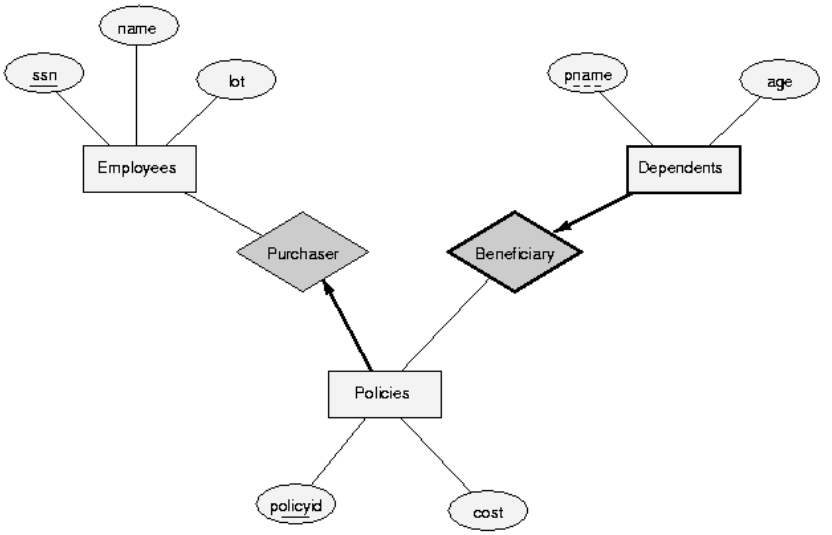
Consider the following ER diagram, representing insurance policies owned by employees at a company. It depicts 3 entity sets Employee, policy and Dependents. The 3 entity sets are associated with a ternary relationship set called Covers. Each employee can own several polices, each policy can be owned by several employees, and each dependent can be covered by several policies.



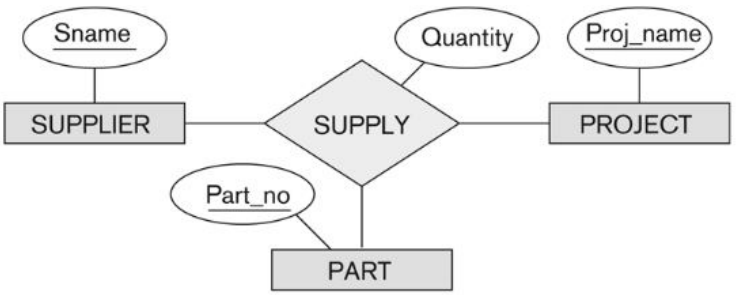
if we wish to model the following additional requirements:

- A policy cannot be owned jointly by two or more employees.
- Every policy must be owned by some employee.
- Dependents is a weak entity set, and each dependent entity is uniquely identified by taking pname in conjunction with the policyid of a policy entity (which, intuitively, covers the given dependent).

The best way to model this is to switch away from the ternary relationship set, and instead use two distinct binary relationship sets.

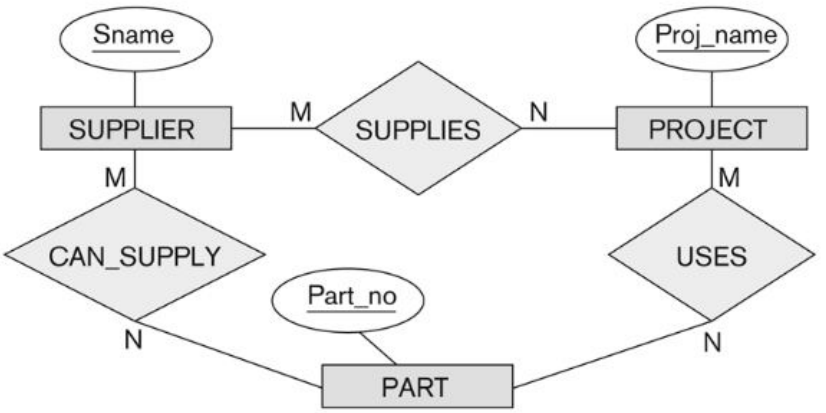


TERNARY VS BINARY: Generally the degree of a relationship set is assessed by counting the no. of nodes or edges that emanate from that relationship set.



Supply in the ternary relationship set from the first figure, which has a set of relationship instances (s,j,p) which means 's' is a supplier who is supplying part 'p' to a project 'j'.

A ternary relationship represent different information than 3 binary relationship sets do. Here the relationship sets canSupply, uses and supplies substitute the ternary relationship set "supply".

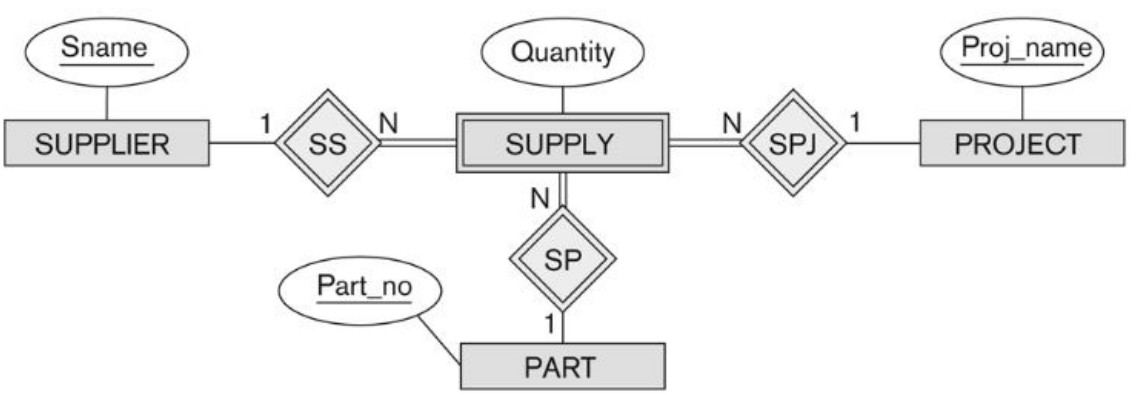


"CANSUPPLY", "USES" and "SUPPLIES" are the three binary relationship sets established where

- Supplier and part which have "CANSUPPLY" binary relationship include an instance (s,p) which says supplier 's' can supply part 'p' to any project.
- "USES" relationship between project and part includes an instance (j,p) which says project 'j' uses part 'p'.
- "SUPPLIES" binary relationship between supplier and project includes an instance (s,j) which says supplier 's' supplies some part to project 'j'.

No combination of binary relationships is an adequate substitute. because there is question "where to add quantity attribute?". Is it to the can-supply or to the uses or to the supplies??

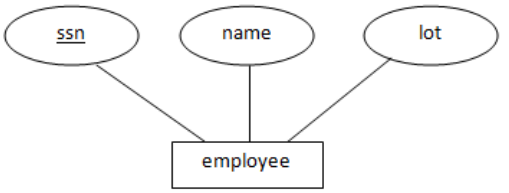
The solution for this is to maintain the same ternary relationship with a weak entity set Supply which has attribute Qty.



SQL

CREATING TABLES WITH RELATIONSHIPS:

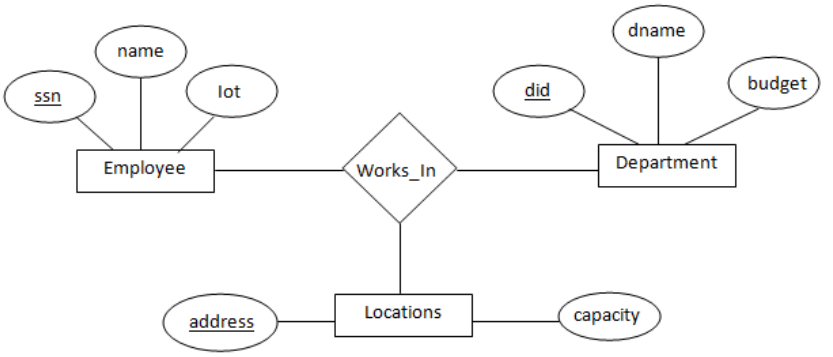
Entity Set: An entity set is mapped to a relation in a straightforward way: Each attribute of the entity set becomes an attribute of the table. Note that we know both the domain of each attribute and the (primary) key of an entity set.



SQL> CREATE TABLE Employees (ssn CHAR(11), name CHAR(30) , lot INTEGER,
PRIMARY KEY (ssn));

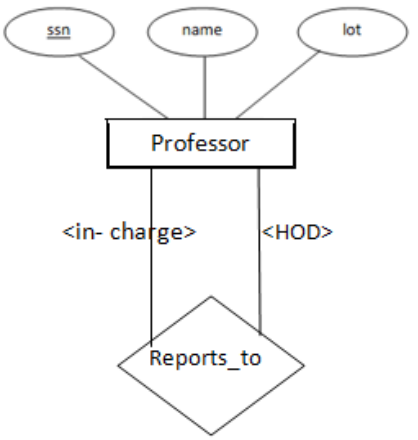
Relationship sets without constraints: To represent a relationship, we must be able to identify each participating entity and give values to the descriptive attributes of the relationship. Thus, the attributes of the relation include:

- The primary key attributes of each participating entity set, as foreign key fields.
- The descriptive attributes of the relationship set. The set of non descriptive attributes is a super key for the relation. If there are no key constraints, this set of attributes is a candidate key.



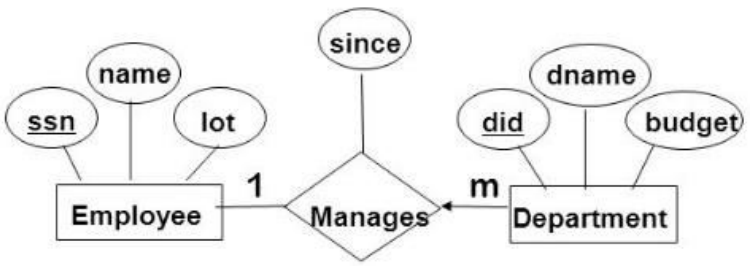
SQL> CREATE TABLE Works_In (ssn CHAR(11),did INTEGER,
address CHAR(20) ,since DATE, PRIMARY KEY (ssn, did, address),
FOREIGN KEY (ssn) REFERENCES Employees,
FOREIGN KEY (address) REFERENCES Locations,
FOREIGN KEY (did) REFERENCES Departments);

Another Example:



```
SQL>  CREATE TABLE Reports_To ( in_charge_ssn CHAR (11),
      hod_ssn CHAR (11) ,
      PRIMARY KEY (in_charge_ssn,hod_ssn),
      FOREIGN KEY (in_charge_ssn) REFERENCES Professor(ssn),
      FOREIGN KEY (hod_ssn) REFERENCES Professor(ssn) );
```

Relationship sets with key constraints: If a relationship set involves n entity sets and some of them are linked via arrows in the ER diagram, the key for anyone of these m entity sets constitutes a key for the relation to which the relationship set is mapped. Hence we have m candidate keys, and one of these should be designated as the primary key.



The table corresponding to Manages has the attributes *ssn, did, since.* However, because each department has at most one manager, no two tuples can have the same *did* value but differ on the *ssn* value. A consequence of this observation is that *did* is itself a key for Manages; indeed, the set *did, ssn* is not a key (because it is not minimal). The Manages relation can be defined using the following SQL statement:

```
SQL>  CREATE TABLE Manages (ssn CHAR (11) , did INTEGER, since DATE,
      PRIMARY KEY (did),
      FOREIGN KEY (ssn) REFERENCES Employees,
      FOREIGN KEY (did) REFERENCES Departments)
```

A second approach to translating a relationship set with key constraints is often superior because it avoids creating a distinct table for the relationship set. The idea is to include the information about the relationship set in the table corresponding to the entity set with the key, taking advantage of the key constraint. In the Manages example, because a department has at most one manager, we can add the key fields of the Employees tuple denoting the manager and the *since* attribute to the Departments tuple.

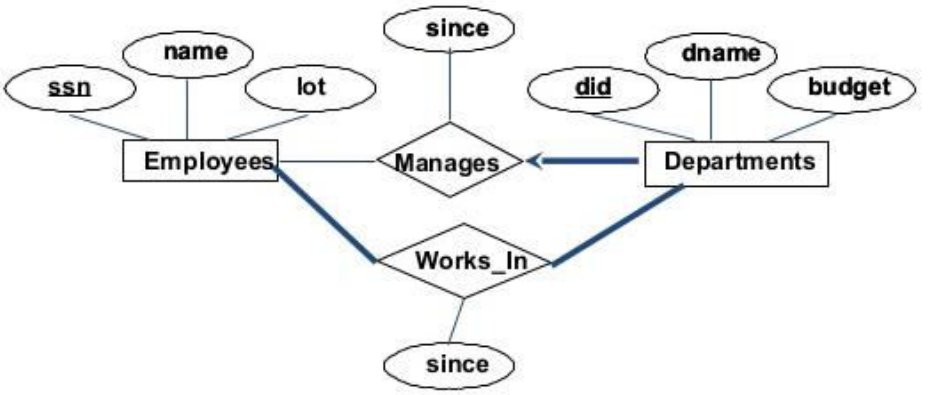
This approach eliminates the need for a separate Manages relation, and queries asking for a department's manager can be answered without combining information from two relations. The only drawback to this approach is that space could be wasted if several departments have no managers. In this case the added fields would have to be filled with *null* values. The first translation (using a separate table for Manages) avoids this inefficiency, but some important queries require us to combine information from two relations, which can be a slow operation.

The following SQL statement, defining a DepLMgr relation that captures the information in both Departments and Manages, illustrates the second approach to translating relationship sets with key constraints:

```
SQL>  CREATE TABLE DepLMgr ( did INTEGER, dname CHAR(20),
       budget REAL, ssn CHAR (11) , since DATE,
      PRIMARY KEY (did),
      FOREIGN KEY (ssn) REFERENCES Employees);
```

Relationship Sets with Participation Constraints

Every department is required to have a manager, due to the participation constraint, and at most one manager, due to the key constraint.



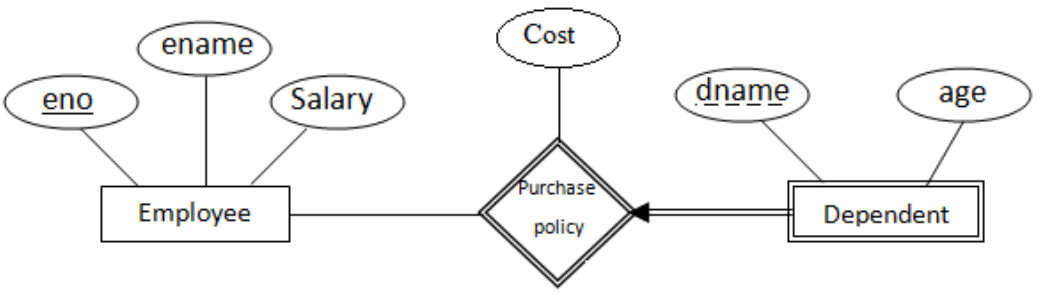

```
SQL>  CREATE TABLE DepLMgr ( did INTEGER, dname CHAR(20) , budget REAL,
      ssn CHAR(11) NOT NULL, since DATE,
      PRIMARY KEY (did),
      FOREIGN KEY (ssn) REFERENCES Employees ON DELETE NO ACTION);
```

It also captures the participation constraint that every department must have a manager: Because *ssn* cannot take on *null* values, each tuple of DepLMgr identifies a tuple in Employees (who is the manager). The NO ACTION specification, which is the default and need not be explicitly specified, ensures that an Employees tuple cannot be deleted while it is pointed to by a Dept-Mgr tuple. If we wish to delete such an Employees tuple, we must first change the DepLMgr tuple to have a new employee as manager.

To ensure total participation of Departments in Works_In using SQL, we need an assertion. We have to guarantee that every *did* value in Departments appears in a tuple of Works_In; further, this tuple of Works_In must also have *non-null* values in the fields that are foreign keys referencing other entity sets involved in the relationship (in this example, the *ssn* field). We can ensure the second part of this constraint by imposing the stronger requirement that *ssn* in Works-ln cannot contain *null* values.

Weak Entity Sets: A weak entity set always participates in a one-to-many binary relationship and has a key constraint and total participation. The second translation approach is ideal in this case, but we must take into account that the weak entity has only a partial key. Also, when an owner entity is deleted, we want all owned weak entities to be deleted.

Consider the Dependents weak entity set shown in Figure , with partial key *pname.* A Dependents entity can be identified uniquely only if we take the key of the *owning* Employees entity and the *pname* of the Dependents entity, and the Dependents entity must be deleted if the owning Employees entity is deleted.



We can capture the desired semantics with the following definition of the Dep_Policy relation:

```
SQL>  CREATE TABLE Dep_Policy (pname CHAR(20) , age INTEGER,
      cost REAL, eno CHAR (11) ,
      PRIMARY KEY (pname, eno),
      FOREIGN KEY (eno) REFERENCES Employees ON DELETE CASCADE );
```

Observe that the primary key is *(pname, eno)* , since Dependents is a weak entity. We have to ensure that every Dependents entity is associated with an Employees entity (the owner), as per the total participation constraint on Dependents. That is, *eno* cannot be *null.* This is ensured because *eno* , is part of the primary key. The CASCADE option ensures that information about an employee's policy and dependents is deleted if the corresponding Employees tuple is deleted.

IMPLEMENTATION OF KEY AND INTEGRITY CONSTRAINTS

1.NOT NULL: When a column is defined as NOTNULL, then that column becomes a mandatory column. It implies that a value must be entered into the column if the record is to be accepted for storage in the table.

Syntax: CREATE TABLE Table_Name(column_name data_type(*size*) **NOT NULL,**);

Example:

SQL> CREATE Table emp2(eno number(5) not null,ename varchar2(10));

Table created.

SQL> desc emp2;

Name	Null?	Type
ENO	NOT NULL	NUMBER(5)
ENAME		VARCHAR2(10)

2. UNIQUE: The purpose of a unique key is to ensure that information in the column(s) is unique i.e. a value entered in column(s) defined in the unique constraint must not be repeated across the column(s). A table may have many unique keys.

Syntax: CREATE TABLE Table_Name(column_name data_type(*size*) **UNIQUE, ….**);

Example:

SQL> CREATE Table emp3(eno number(5) unique,ename varchar2(10));

Table created.

SQL> desc emp3;

Name	Null?	Type
ENO		NUMBER(5)
ENAME		VARCHAR2(10)

SQL> insert into emp3 values(&eno,'&ename');

```
Enter value for eno: 1
Enter value for ename: sss
old 1: insert into emp3 values(&eno,'&ename')
new 1: insert into emp3 values(1,'sss')
```

1 row created.

SQL> /

Enter value for eno: 1
Enter value for ename: sas
old 1: insert into emp3 values(&eno,'&ename')
new 1: insert into emp3 values(1,'sas')
insert into emp3 values(1,'sas')
ERROR at line 1:
ORA-00001: unique constraint (SCOTT.SYS_C003006) violated

3. CHECK: Specifies a condition that each row in the table must satisfy. To satisfy the constraint, each row in the table must make the condition either TRUE or unknown (due to a null).

Syntax: CREATE TABLE Table_Name(column_name data_type(*size*) CHECK(*logical expression*), ….);

Example:

SQL> CREATE TABLE student (sno NUMBER (3), name CHAR(10),class CHAR(5),CHECK(class IN('CSE','CAD','VLSI'));

4. PRIMARY KEY: A field which is used to identify a record uniquely. A column or combination of columns can be created as primary key, which can be used as a reference from other tables. A table contains primary key is known as Master Table.

- It must uniquely identify each record in a table.
- It must contain unique values.
- It cannot be a null field.
- It cannot be multi port field.
- It should contain a minimum no. of fields necessary to be called unique.

Syntax: CREATE TABLE Table_Name(column_name data_type(*size*) PRIMARY KEY, **….**);

Example:

SQL> CREATE TABLE faculty (fcode NUMBER(3) PRIMARY KEY, fname CHAR(10));

5. FOREIGN KEY: It is a table level constraint. We cannot add this at column level. To reference any primary key column from other table this constraint can be used. The table in which the foreign key is defined is called a **detail table**. The table that defines the primary key and is referenced by the foreign key is called the **master table**.

Syntax: CREATE TABLE Table_Name (col_name type(*size*)
FOREIGN KEY(col_name) **REFERENCES** table_name
);

Example:

SQL> CREATE TABLE subject (

scode **NUMBER (3) PRIMARY KEY,**

subname **CHAR(10)**,fcode **NUMBER(3),**

FOREIGN KEY(fcode) **REFERENCE** faculty);

SUB QUERIES

A Subquery or Inner query or a Nested query is a query within another SQL query and embedded within the WHERE clause. Subquery is used to return data that will be used in the main query as a condition to further restrict the data to be retrieved. Subqueries can be used with the SELECT, INSERT, UPDATE, and DELETE statements along with the operators like =, <, >, >=, <=, IN, BETWEEN, etc.

There are a few rules that subqueries must follow −

- Subqueries must be enclosed within parentheses.
- A subquery can have only one column in the SELECT clause, unless multiple columns are in the main query for the subquery to compare its selected columns.
- An ORDER BY command cannot be used in a subquery, although the main query can use an ORDER BY. The GROUP BY command can be used to perform the same function as the ORDER BY in a subquery.
- Subqueries that return more than one row can only be used with multiple value operators such as the IN operator.
- The SELECT list cannot include any references to values that evaluate to a BLOB, ARRAY, CLOB, or NCLOB.
- A subquery cannot be immediately enclosed in a set function.
- The BETWEEN operator cannot be used with a subquery. However, the BETWEEN operator can be used within the subquery.

Subqueries with the SELECT Statement: Subqueries are most frequently used with the SELECT statement.

Syntax:

```
SELECT column_name [, column_name ] FROM   table1 [, table2 ]
WHERE  column_name OPERATOR
  (SELECT column_name [, column_name ]  FROM table1 [, table2 ]  [WHERE])
```

Example:

Consider the CUSTOMERS table having the following records −

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	35	Ahmadabad	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

Now, let us check the following subquery with a SELECT statement.

```
SQL>    SELECT *  FROM CUSTOMERS
        WHERE ID IN (SELECT ID FROM CUSTOMERS WHERE SALARY > 4500) ;
```

This would produce the following result.

ID	NAME	AGE	ADDRESS	SALARY
4	Chaitali	25	Mumbai	6500.00
5	Hardik	27	Bhopal	8500.00
7	Muffy	24	Indore	10000.00

Subqueries with the INSERT Statement: Subqueries also can be used with INSERT statements. The INSERT statement uses the data returned from the subquery to insert into another table. The selected data in the subquery can be modified with any of the character, date or number functions.

Syntax:

INSERT INTO table_name [(column1 [, column2])]
SELECT [*|column1 [, column2] FROM table1 [, table2] [WHERE VALUE OPERATOR]

Example: Consider a table CUSTOMERS_BKP with similar structure as CUSTOMERS table. Now to copy the complete CUSTOMERS table into the CUSTOMERS_BKP table, you can use the following syntax.

```
SQL>    INSERT INTO CUSTOMERS_BKP
        SELECT * FROM CUSTOMERS WHERE ID IN
                (SELECT ID  FROM CUSTOMERS) ;
```

Subqueries with the UPDATE Statement: The subquery can be used in conjunction with the UPDATE statement. Either single or multiple columns in a table can be updated when using a subquery with the UPDATE statement.

Syntax:

UPDATE table SET column_name = new_value [WHERE OPERATOR [VALUE]
(SELECT COLUMN_NAME FROM TABLE_NAME [WHERE)])

Example: Assuming, we have CUSTOMERS_BKP table available which is backup of CUSTOMERS table. The following example updates SALARY by 0.25 times in the CUSTOMERS table for all the customers whose AGE is greater than or equal to 27.

```
SQL>    UPDATE CUSTOMERS SET SALARY = SALARY * 0.25
        WHERE AGE IN (SELECT AGE FROM CUSTOMERS_BKP WHERE AGE >= 27 );
```

This would impact two rows and finally CUSTOMERS table would have the following records.

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	35	Ahmadabad	125.00
2	Khilan	25	Delhi	1500.00
3	Kaushik	23	Kota	2000.00
4	Chaitali	25	Mumbai	6500.00
5	Hardik	27	Bhopal	2125.00
6	Komal	22	MP	4500.00
7	Muffy	24	Indore	10000.00

Subqueries with the DELETE Statement: The subquery can be used in conjunction with the DELETE statement like with any other statements mentioned above.

Syntax:

DELETE FROM TABLE_NAME [WHERE OPERATOR [VALUE]
(SELECT COLUMN_NAME FROM TABLE_NAME [WHERE)])

Example: Assuming, we have a CUSTOMERS_BKP table available which is a backup of the CUSTOMERS table. The following example deletes the records from the CUSTOMERS table for all the customers whose AGE is greater than or equal to 27.

```
SQL>    DELETE FROM CUSTOMERS WHERE AGE IN
        (SELECT AGE FROM CUSTOMERS_BKP  WHERE AGE >= 27 );
```

This would impact two rows and finally the CUSTOMERS table would have the following records.

ID	NAME	AGE	ADDRESS	SALARY
2	Khilan	25	Delhi	1500.00
3	Kaushik	23	Kota	2000.00
4	Chaitali	25	Mumbai	6500.00
6	Komal	22	MP	4500.00
7	Muffy	24	Indore	10000.00

GROUPING

The SQL **GROUP BY** clause is used in collaboration with the SELECT statement to arrange identical data into groups. This GROUP BY clause follows the WHERE clause in a SELECT statement and precedes the ORDER BY clause.

Syntax: The basic syntax of a GROUP BY clause is shown in the following code block. The GROUP BY clause must follow the conditions in the WHERE clause and must precede the ORDER BY clause if one is used.

```
SELECT column1, column2  FROM table_name WHERE [ conditions ]
GROUP BY column1, column2 ORDER BY column1, column2
```

Guidelines to use Group By Clause

- If the group function is included in the select clause., we should not use individual result columns.
- The extra non-group functional columns should be declared in the Group By clause.
- Using WHERE clause, rows can be pre executed before dividing them into groups.
- Column aliases can't be used in the Group by clause.
- By Default, rows are sorted by ascending order of columns included in the Group By list.

Examples:

Display the average salary of the departments from Emp table.

```
SQL> select deptno,AVG(sal) from emp group by deptno;
```

Display the minimum and maximum salaries of employees working as clerks in each department.

```
SQL> select deptno,min(sal),max(sal) from emp where job='CLERK' Group by deptno;
```

Excluding Groups of Results: While using Group By clause, there is a provision to exclude some group results using HAVING clause. HAVING clause is used to specify which groups can be specified. It is used to filter the data which is associated with the group functions.

Syntax:

```
SELECT column1, column2 FROM table1, table2 WHERE [ conditions ]
GROUP BY column1, column2 HAVING [ conditions ]
```

Sequence of steps:

- First rows are grouped.
- Group functions are applied to that identifies groups.
- Groups that match with the criteria in having clause are displayed.

The HAVING clause can predict Group By clause but it is more logical to declare it after Group By clause. Group By clause can be used without a group function in the SELECT list. If rows need to be restricted based on the result of a Group function, we must have a group by clause as well as Having clause. Existence of Group by clause does not guarantee the existence of HAVING clause but the existence of HAVING clause demands the existence of Group By clause.

Example:

Display the Departments having the min salary of clerks is > 1000

```
SQL>  select deptno, min(sal) from emp where job='CLERK'
        group by deptno HAVING min(sal)> 1000;
```

Display the sum of the salaries of the departments.

```
SQL> select deptno, sum(sal) from emp group by deptno;
```

AGGREGATION: Aggregation Functions or Group Functions

These function return a single row based on group of rows. These can appear in SELECT list and HAVING clauses only. These operate on sets of rows to give one result per group. The set may be whole table or table split into group.

Guidelines to use Aggregate Functions:

Distinct makes the functions to consider only non duplicate value.
All makes the function to consider every value including duplicates.

Syntax:

GroupFunctionName (Distinct/ All columns)

The data types for arguments may be char,varchar, number or Date. All group functions except count(*) ignore NULL values. To substitute a value for NULL value, use the NVL() function. When a group function is declared in a select List, no single row columns should be declared. other columns can be declared but they should be declared in the group by clause.
The list of Aggregate Functions are:

MIN	returns the smallest value in a given column
MAX	returns the largest value in a given column
SUM	returns the sum of the numeric values in a given column
AVG	returns the average value of a given column
COUNT	returns the total number of values in a given column
COUNT(*)	returns the number of rows in a table

Consider the following table:

Employees Table

IdNum	LName	FName	JobCode	Salary	Phone
1876	CHIN	JACK	TA1	42400	212/588-5634
1114	GREENWALD	JANICE	ME3	38000	212/588-1092
1556	PENNINGTON	MICHAEL	ME1	29860	718/383-5681
1354	PARKER	MARY	FA3	65800	914/455-2337
1130	WOOD	DEBORAH	PT2	36514	212/587-0013

MIN Function:

SQL> select min(Salary) from Employees;
OUTPUT:

MIN(SALARY)
29860

MAX Function:

SQL> select max(Salary) from Employees;
OUTPUT:

MAX(SALARY)
65800

SUM Function:

SQL> select sum(Salary) from Employees;
OUTPUT:

SUM(SALARY)
212574

AVG Function:

SQL> select avg(Salary) from Employees;
OUTPUT:

AVG(SALARY)
42514.8

COUNT Function:

SQL> select count(IdNum) from Employees;
OUTPUT:

COUNT(IDNUM)
5

COUNT(*) Function:

SQL> select count(*) from Employees;
OUTPUT:

COUNT(*)
5

ORDERING: (ORDER BY CLAUSE)

The SQL **ORDER BY** clause is used to sort the data in ascending or descending order, based on one or more columns. Some databases sort the query results in an ascending order by default.

Syntax:

select column-list from table_name[where condition]
[order by column1, column2,..columnn][asc|desc];

Example:

Consider the following table:

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	35	Ahmadabad	2000
2	Khilan	25	Delhi	1500
3	Kowshik	23	Kota	2000
4	Chaitali	25	Mumbai	6500
5	Hardhik	27	Bhopal	8500
6	Komal	22	MP	4500
7	Muffy	24	Indore	10000

SQL > select * from customers order by name, salary;
OUTPUT:

ID	NAME	AGE	ADDRESS	SALARY
4	Chaitali	25	Mumbai	6500
5	Hardhik	27	Bhopal	8500
3	Kowshik	23	Kota	2000
2	Khilan	25	Delhi	1500
6	Komal	22	MP	4500
7	Muffy	24	Indore	10000
1	Ramesh	35	Ahmadabad	2000

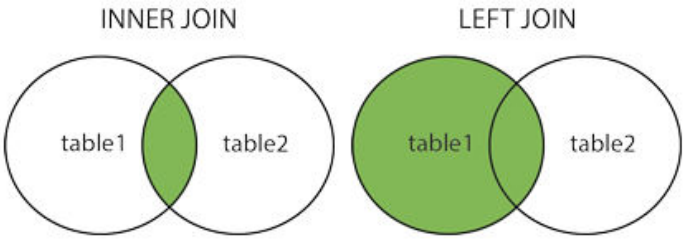
IMPLEMENTATION OF JOINS

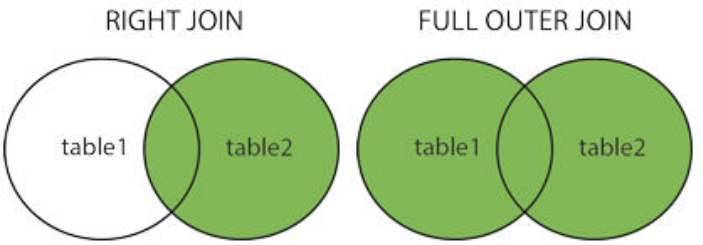
A JOIN clause is used to combine rows from two or more tables, based on a related column between them.

Types of SQL JOINS:

There are 4 different types of SQL joins.

- (INNER) JOIN: Returns records that have matching values in both tables
- LEFT (OUTER) JOIN: Return all records from the left table, and the matched records from the right table
- RIGHT (OUTER) JOIN: Return all records from the right table, and the matched records from the left table
- FULL (OUTER) JOIN: Return all records when there is a match in either left or right table





1. INNER JOIN: The INNER JOIN keyword selects records that have matching values in both tables

Syntax:

select *column_name(s)* from *table1* inner join *table2*
on *table1.column_name = table2.column_name*;

2. LEFT JOIN: The LEFT JOIN keyword returns all records from the left table (table1), and the matched records from the right table (table2). The result is NULL from the right side, if there is no match.

Syntax:

select *column_name(s)*from *table1*left join *table2*
on *table1.column_name = table2.column_name*;

3. RIGHT JOIN: The RIGHT JOIN keyword returns all records from the right table (table2), and the matched records from the left table (table1). The result is NULL from the left side, when there is no match.

Syntax:

select *column_name(s)*from *table1*right join *table2*
on *table1.column_name table2.column_name*;

4. FULL JOIN: The FULL OUTER JOIN keyword return all records when there is a match in either left (table1) or right (table2) table records.

Syntax:

select *column_name(s)*from *table1*full outer join *table2*
on *table1.column_name = table2.column_name*;

5. SQL SELF JOIN: A self JOIN is a regular join, but the table is joined with itself.

Syntax:

select *column_name(s)* from *table1 t1, table1 t2* where *condition*;

EXAMPLES:

Consider the following two tables.

Table 1 − CUSTOMERS Table is as follows.

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	31	Ahmadabad	2000.00
2	Khilan	25	Delhi	1500.00
3	Kaushik	23	Kota	2000.00
5	Hardik	27	Mumbai	6500.00
6	Komal	22	MP	4500.00
7	Muffy	24	Indore	10000.00

Table 2 − ORDERS Table is as follows.

OID	DATE	CUSTOMER_ID	AMOUNT
102	2009-10-08 00:00:00	3	3000
100	2009-10-08 00:00:00	3	1500
101	2009-11-20 00:00:00	2	1560
103	2008-05-20 00:00:00	4	2060

Inner Join:

```
SQL>  SELECT  id, name, amount, date FROM customers
      INNER JOIN orders
      ON customers.id = orders.customer_id;
```

This would produce the following result −

ID	NAME	AMOUNT	DATE
3	Kaushik	3000	2009-10-08-00:00:00
3	Kaushik	1500	2009-10-08 00:00:00
2	Khilan	1560	2009-11-20 00:00:00
4	Chaitali	2060	2008-05-20 00:00:00

Left Join:

```
SQL>  SELECT  id, name, amount, date FROM customers
      LEFT JOIN orders
      ON customers.id = orders.customer_id;
```

This would produce the following result −

ID	NAME	AMOUNT	DATE
1	Ramesh	NULL	NULL
2	Khilan	1560	2009-11-20 00:00:00
3	Kaushik	3000	2009-10-08-00:00:00
3	Kaushik	1500	2009-10-08 00:00:00
4	Chaitali	2060	2008-05-20 00:00:00
5	Hardik	NULL	NULL
6	Komal	NULL	NULL
7	Muffy	NULL	NULL

Right Join:

```
SQL>  SELECT  id, name, amount, date FROM customers
      RIGHT JOIN orders
      ON customers.id = orders.customer_id;
```

This would produce the following result −

ID	NAME	AMOUNT	DATE
3	Kaushik	3000	2009-10-08-00:00:00
3	Kaushik	1500	2009-10-08 00:00:00
2	Khilan	1560	2009-11-20 00:00:00
4	Chaitali	2060	2008-05-20 00:00:00

Full Join:

```
SQL>  SELECT  id, name, amount, date FROM customers
      FULL JOIN orders
      ON customers.id = orders.customer_id;
```

This would produce the following result −

ID	NAME	AMOUNT	DATE
1	Ramesh	NULL	NULL
2	Khilan	1560	2009-11-20 00:00:00
3	Kaushik	3000	2009-10-08-00:00:00
3	Kaushik	1500	2009-10-08 00:00:00
4	Chaitali	2060	2008-05-20 00:00:00
5	Hardik	NULL	NULL
6	Komal	NULL	NULL
7	Muffy	NULL	NULL
3	Kaushik	3000	2009-10-08-00:00:00
3	Kaushik	1500	2009-10-08 00:00:00
2	Khilan	1560	2009-11-20 00:00:00
4	Chaitali	2060	2008-05-20 00:00:00

Self Join:

```
SQL>  SELECT  a.id, b.name, a.salary FROM customers a, customers b
      WHERE a.salary<b.salary
```

This would produce the following result −

ID	NAME	SALARY
2	Ramesh	1500.00
2	Kaushik	1500.00
1	Chaitail	2000.00
2	Chaitail	1500.00
3	Chaitail	2000.00
6	Chaitail	4500.00

1	Hardik	2000.00
2	Hardik	1500.00
3	Hardik	2000.00
4	Hardik	6500.00
6	Hardik	4500.00
1	Komal	2000.00
2	Komal	1500.00
3	Komal	2000.00
1	Muffy	2000.00
2	Muffy	1500.00
3	Muffy	2000.00
4	Muffy	6500.00
5	Muffy	8500.00
6	Muffy	4500.00

VIEWS:

A view is nothing more than a SQL statement that is stored in the database with an associated name. A view is actually a composition of a table in the form of a predefined SQL query. A view can contain all rows of a table or select rows from a table. A view can be created from one or many tables which depends on the written SQL query to create a view.

Views, which are a type of virtual tables allow users to do the following −

- Structure data in a way that users or classes of users find natural or intuitive.
- Restrict access to the data in such a way that a user can see and (sometimes) modify exactly what they need and no more.
- Summarize data from various tables which can be used to generate reports.

2 Types of Views Updatable and Read-only -views

Unlike base tables, VIEWs are either updatable or read-only, but not both. INSERT, UPDATE, and DELETE operations are allowed on updatable VIEWs and base tables, subject to any other constraints. INSERT, UPDATE, and DELETE are not allowed on read-only VIEWs, but you can change their base tables, as you would expect. An updatable VIEW is one that can have each of its rows associated with exactly one row in an underlying base table.

When the VIEW is changed, the changes pass unambiguously through the VIEW to that underlying base table. Updatable VIEWs in Standard SQL are defined only for queries that meet these criteria:

1. Built on only one table
2. No GROUP BY clause
3. No HAVING clause
4. No aggregate functions
5. No calculated columns
6. No UNION, INTERSECT, or EXCEPT

7.No SELECT DISTINCT clause

8.Any columns excluded from the VIEW must be NULL-able or have a DEFAULT in the base table, so that a whole row can be constructed for insertion By implication, the VIEW must also contain a key of the table.

In short, we are absolutely sure that each row in the VIEW maps back to one and only one row in the base table. Some updating is handled by the CASCADE option in the referential integrity constraints on the base tables, not by the VIEW declaration.

The definition of updatability in Standard SQL is actually fairly limited, but very safe. The database system could look at information it has in the referential integrity constraints to widen the set of allowed updatable VIEWs. You will find that some implementations are now doing just that, but it is not common yet.

The SQL Standard definition of an updatable VIEW is actually a subset of the possible updatable VIEWs, and a very small subset at that. The major advantage of this definition is that it is based on syntax and not semantics.

Examples of Updatable and Non-updatable View.

```
CREATE VIEW view_1 AS SELECT * FROM Table1 WHERE x IN (1,2);
        -- updatable, has a key!


CREATE VIEW view_2 AS SELECT * FROM Table1 WHERE x = 1 UNION ALL SELECT * FROM Table1
WHERE x = 2;
        -- not updatable!
```

More about Views:

A view takes up **no storage space** other than for the definition of the view in the data dictionary.

A view contains **no data**. All the data it shows comes from the base tables.

A view can provide an additional level of **table security** by restricting access to a set of rows or columns of a table.

A view **hides implementation complexity**. The user can select from the view with a simple SQL, unaware that the view is based internally on a join between multiple tables.

A view lets you **change the data** you can access, applying operators, aggregation functions, filters etc. on the base table.

A view **isolates applications from changes** in definitions of base tables. Suppose a view uses two columns of a base table, it makes no difference to the view if other columns are added, modified or removed from the base table.

To know about the views in your own schema, look up **user_views**.

The underlying SQL definition of the view can be read via **select text from user_views** for the view.

Oracle does not enforce **constraints on views**. Instead, views are subject to the constraints of their base tables.

Creating Views

Database views are created using the **CREATE VIEW** statement. Views can be created from a single table, multiple tables or another view. To create a view, a user must have the appropriate system privilege according to the specific implementation.

Syntax:

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

we can include multiple tables in your SELECT statement in a similar way as we use them in a normal SQL SELECT query.

Example:

Consider the CUSTOMERS table having the following records −

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	32	Ahmadabad	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 to create a view from the CUSTOMERS table. This view would be used to have customer name and age from the CUSTOMERS table.

SQL > CREATE VIEW CUSTOMERS_VIEW AS SELECT name, age FROM CUSTOMERS;

Now, you can query CUSTOMERS_VIEW in a similar way as you query an actual table. Following is an example for the same.

*SQL > SELECT * FROM CUSTOMERS_VIEW;*

This would produce the following result.

NAME	AGE
Ramesh	32
Khilan	25
Kaushik	23
Chaitali	25
Hardik	27
Komal	22
Muffy	24

The With Check Option:

The WITH CHECK OPTION is a CREATE VIEW statement option. The purpose of the WITH CHECK OPTION is to ensure that all UPDATE and INSERTs satisfy the condition(s) in the view definition. If they do not satisfy the condition(s), the UPDATE or INSERT returns an error.

Example:

```
SQL>  CREATE VIEW CUSTOMERS_VIEW AS
      SELECT name, age FROM  CUSTOMERS
      WHERE age IS NOT NULL WITH CHECK OPTION;
```

The WITH CHECK OPTION in this case should deny the entry of any NULL values in the view's AGE column, because the view is defined by data that does not have a NULL value in the AGE column.

Updating a View

A view can be updated under certain conditions which are given below −

- The SELECT clause may not contain the keyword DISTINCT.
- The SELECT clause may not contain summary functions.
- The SELECT clause may not contain set functions.
- The SELECT clause may not contain set operators.
- The SELECT clause may not contain an ORDER BY clause.
- The FROM clause may not contain multiple tables.
- The WHERE clause may not contain subqueries.
- The query may not contain GROUP BY or HAVING.
- Calculated columns may not be updated.
- All NOT NULL columns from the base table must be included in the view in order for the INSERT query to function.

So, if a view satisfies all the above-mentioned rules then you can update that view. The following code block has an example to update the age of Ramesh.

```
SQL > UPDATE CUSTOMERS_VIEW SET AGE = 35 WHERE name = 'Ramesh';
```

This would ultimately update the base table CUSTOMERS and the same would reflect in the view itself. Now, try to query the base table and the SELECT statement would produce the following result.

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	35	Ahmadabad	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

Inserting Rows into a View

Rows of data can be inserted into a view. The same rules that apply to the UPDATE command also apply to the INSERT command. Here, we cannot insert rows in the CUSTOMERS_VIEW because we have not included all the NOT NULL columns in this view, otherwise you can insert rows in a view in a similar way as you insert them in a table.

Deleting Rows from a View

Rows of data can be deleted from a view. The same rules that apply to the UPDATE and INSERT commands apply to the DELETE command.

Following is an example to delete a record having AGE = 22.

SQL > DELETE FROM CUSTOMERS_VIEW WHERE age = 22;

This would ultimately delete a row from the base table CUSTOMERS and the same would reflect in the view itself. Now, try to query the base table and the SELECT statement would produce the following result.

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	35	Ahmadabad	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

Dropping Views

Obviously, where you have a view, you need a way to drop the view if it is no longer needed.

Syntax:

DROP VIEW view_name;

Example:

SQL> DROP VIEW CUSTOMERS_VIEW;

SET OPERATIONS

- These operators are used to combine information of similar datatype from one or more than one table.
- Datatype of the corresponding columns in all the select statement should be same.
- Different types of set commands are
  - UNION
  - UNION ALL
  - INTERSECT
  - MINUS
- Set operators are combine 2 or more queries into one result .
- The result of each SELECT statement can be treated as a set and SQL set operators can be applied on those sets to arrive at a final result.
- SQL statements containing set operators are referred to as compound queries, and each SELECT statements in a command query in referred to as a compound query.
- Set operations are often called vertical joins, as a result combines data from 2 or more SELECT based on columns instead of rows.

Syntax:

<compound query>
{ UNION | UNION ALL | MINUS | INTERSECT }
<compound query>

UNION: Combines the result of 2 select statements into one result set, and then estimates any duplicate rows from that result set.

UNION ALL: Combines the result of 2 SELECT statements into one result set including the duplicates.

INTERSECT: Returns only the rows that are returned by each of two SELECT statements.

MINUS: Takes the result set of each SELECT statement, and removes those rows that are also recommended by a second SELECT statement.

Point Of Concentration:

- The queries are all executed independently but their output is merged.
- Only final queries ends with a semicolon(;).

Rules And Restrictions:

- The result set of both the queries must have same number of columns.
- The datatype of each column in the second result set must match the datatype of the corresponding column in the first result set.
- The 2 SELECT statements may not contain an ORDER BY clause. The final result of the entire set operations can be ordered.
- The columns used for ordering must be defined through the column number.

Examples:

Display the employees who work in departments 10 and 30with out duplicates.

```
SQL>  SELECT empno, ename from emp where deptno=10
      UNION
      SELECT empno, ename from emp where deptno=30;
```

Display the employees who work in departments 10 and 30.

```
SQL>  SELECT empo,ename from emp where deptno=10
      UNION ALL
      SELECT empno, ename from emp where deptno=30 ;
```

Display the employees who work in both the departments with deptno 10 and 30.

```
SQL>   SELECT empno, ename from emp where deptno=10
       INTERSECT
       SELECT empno, ename from emp where deptno=30 ;
```

Display the employees whose row number is less than 7 but not less than 6.

```
SQL>  SELECT rownum , ename from emp where rownum<7
      MINUS
      SELECT rownum , ename from emp where rownum<6;
```

RELATIONAL OPERATIONS

Given this simple and restricted data structure, it is possible to define some very powerful relational operators which, from the users' point of view, act in parallel' on all entries in a table simultaneously, although their implementation may require conventional processing.

Codd originally defined eight relational operators.

1. SELECT originally called RESTRICT
2. PROJECT
3. JOIN
4. PRODUCT
5. UNION
6. INTERSECT
7. DIFFERENCE
8. DIVIDE

The most important of these are (1), (2), (3) and (8), which, together with some other aggregate functions, are powerful enough to answer a wide range of queries. The eight operators will be described as general procedures - i.e. not in the syntax of SQL or any other relational language. The important point is that they define the result required rather than the detailed process of obtaining it - what but not how.

SELECT: RESTRICTS the rows chosen from a table to those entries with specified attribute values.

SELECT item FROM stock_level WHERE quantity > 100

constructs a new, logical table - an unnamed relation - with one column per row (i.e. item) containing all rows from stock_level that satisfy the WHERE clause.

PROJECT: Selects rows made up of a sub-set of columns from a table.

PROJECT stock_item OVER item AND description

produces a new logical table where each row contains only two columns - item and description. The new table will only contain distinct rows from stock_item; i.e. any duplicate rows so formed will be eliminated.

JOIN: Associates entries from two tables on the basis of matching column values.

JOIN stock_item WITH stock_level OVER item

It is not necessary for there to be a one-to-one relationship between entries in two tables to be joined - entries which do not match anything will be eliminated from the result, and entries from one table which match several entries in the other will be duplicated the required number of times.

PRODUCT: Builds a relation from two specified relations consisting of all possible combinations of rows, one from each of the two relations. For example, consider two relations, A and B, consisting of rows:

A: a	B: d	=>	A product B: a	d
b	e		a	e
c			b	d
			b	e
			c	d
			c	e

UNION: Builds a relation consisting of all rows appearing in either or both of the two relations. For example, consider two relations, A and B, consisting of rows:

A: a	B: a	=>	A union B: a
b	e		b
c			c
			e

INTERSECT: Builds a relation consisting of all rows appearing in both of the two relations.
For example, consider two relations, A and B, consisting of rows:

A: a	B: a	=>	A intersect B: a
b	e		
c			

DIFFERENCE: Builds a relation consisting of all rows appearing in the first and not in the second of the two relations. For example, consider two relations, A and B, consisting of rows:

A: a	B: a	=>	A - B: b	and	B - A: e
b	e		c		
c					

DIVIDE: Takes two relations, one binary and one unary, and builds a relation consisting of all values of one column of the binary relation that match, in the other column, all values in the unary relation.

A: a	x	B: x	=>	A divide B: a
a	y	y		
a	z			
b	x			
c	y			

Of the relational operators 3.2.4. to 3.2.8.defined by Codd, the most important is DIVISION. For example, suppose table A contains a list of suppliers and commodities, table B a list of all commodities bought by a company. Dividing A by B produces a table listing suppliers who sell all commodities