DATABASE MANAGEMENT SYSTEMS

UNIT III

3.1 Introduction

The *entity-relationship (ER) data 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.
The ER model is important primarily for its role in database design. It provides useful concepts that allow us to move from an informal description of what users want from their database to a more detailed and precise, description that can be implemented in a DBMS.
Even though the ER model describes the physical database model, it is basically useful in the design and communication of the logical database model.

3.2 Overview of Database Design

Our primary focus is the design of the database. The database design process can be divided into six steps:

Requirements Analysis

The very first step in designing a database application is to understand what data is to be stored in the database, what applications must be built on the database, and what operations must be performed on the database. In other words, we must find out what the users want from the database. This process involves discussions with user groups, a study of the current operating environment, how it is expected to change an analysis of any available documentation on existing applications and so on.

Conceptual Database Design

The information gathered in the requirement analysis step is used to develop a high-level description of the data to be stored in the database, along with the conditions known to hold this data. The goal is to create a description of the data that matches both—how users and developers think of the data (and the people and processes to be represented in the data). This facilitates discussion among all the people involved in the design process i.e., developers and as well as users who have no technical background. In simple words, the conceptual database design phase is used in drawing ER model.

Logical Database Design

We must implement our database design and convert the conceptual database design into a database schema (a description of data) in the data model (a collection of high-level data description constructs that hide many low-level storage details) of the DBMS. We will consider only relational DBMSs, and therefore, the task in the logical design step is to convert the conceptual database design in the form of E-R Schema (Entity-Relationship Schema) into a relational database schema.

Schema Refinement

The fourth step in database design is to analyze the collection, of relations (tables) in our relational database schema to identify future problems, and to refine (clear) it.

Physical Database Design

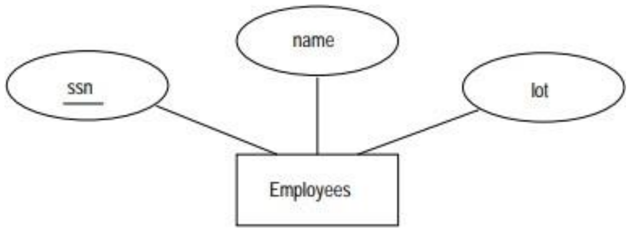
This step may simply involve building indexes on some tables and clustering some tables, or it may involve redesign of parts of the database schema obtained from the earlier design steps.

Application and Security Design

Any software project that involves a DBMS must consider applications that involve processes and identify the entities.

3.3 Entities, Attributes and Entity Sets

Entity: An entity is an object in the real world that is distinguishable from other objects.
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 an 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.

3.4 Representation of Entities and Attributes

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

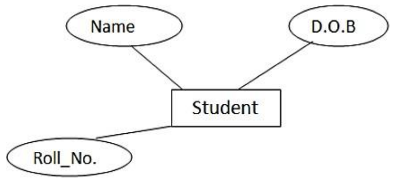
Student
Employee

Fig: Student and Employee entities

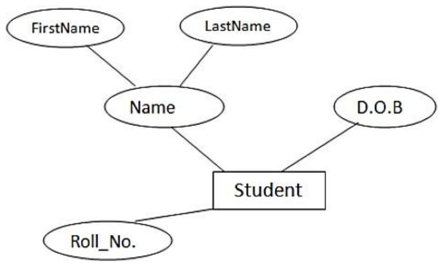
ATTRIBUTES: Attributes are the properties of entities. Attributes are represented by means of ellipses. Every ellipse represents one attribute and is directly connected to its entity.

Types of attributes:

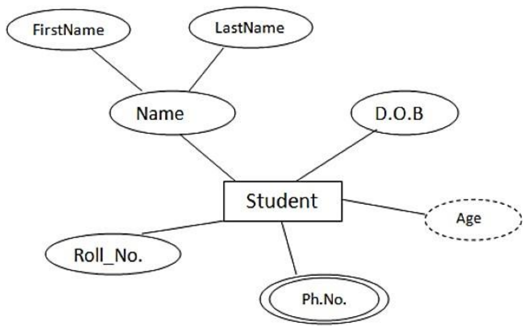
- **Simple attribute** – Simple attributes are atomic values, which cannot be divided further. For example, a student's roll number is an atomic value.



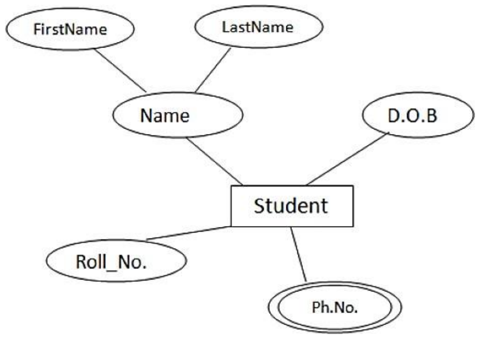
- **Composite attribute** − Composite attributes are made of more than one simple attribute. For example, a student's complete name may have first_name and last_name.



- **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.



3.5 Relationship and Relationship set

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.

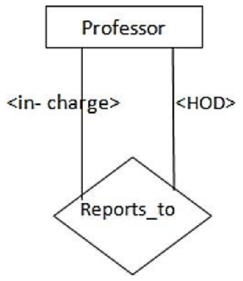
Types of relationships:

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

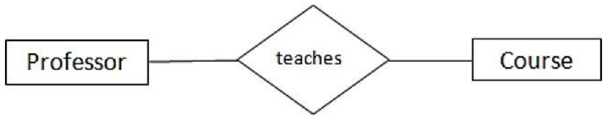
- Unary = degree 1
- Binary = degree 2

- Ternary = degree 3
- n-array = degree

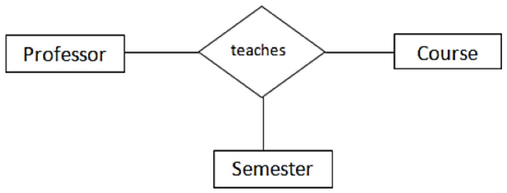
Unary Relationship: A relationship with one entity set. It is like a relationship among 2 entities of same entity set. Example: A professor (in-charge) reports to another professor (Head Of the Dept).



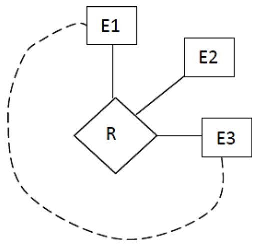
Binary Relationship: A relationship among 2 entity sets. Example: A professor teaches a course and a course is taught by a professor.



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



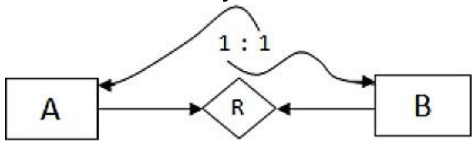
n-array Relationship: A relationship among n entity sets.



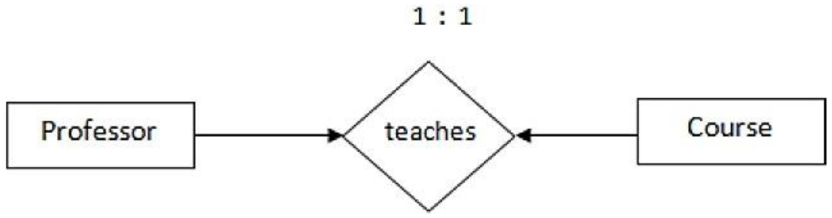
Cardinality:

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.

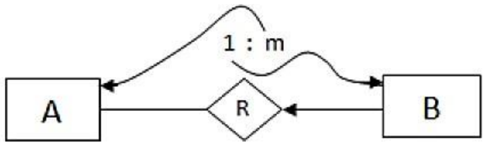
1. **One-to-One relationship:** When only one instance of entities is associated with the relationship, then the relationship is one-to-one relationship. Each entity in A is associated with at most one entity in B and each entity in B is associated with at most one entity in A.



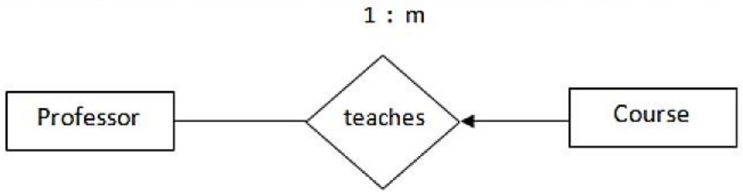
Each professor teaches one course and each course is taught by one professor.



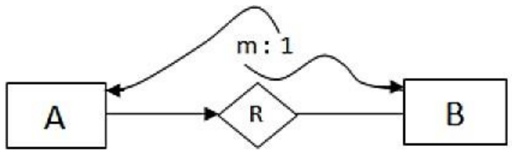
2. **One-to-many relationship**: When more than one instance of an entity is associated with a relationship, then the relationship is one-to-many relationship. Each entity in A is associated with zero or more entities in B and each entity in B is associated with at most one entity in A.



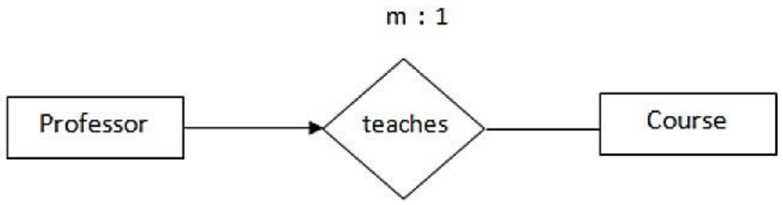
Each professor teaches 0 (or) more courses and each course is taught by at most one professor.



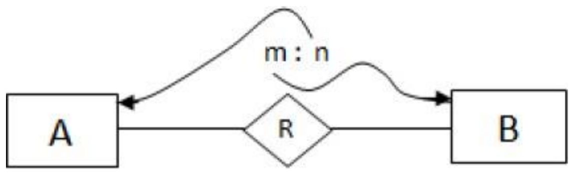
3. **Many-to-one relationship:** When more than one instance of entity is associated with the relationship, then the relationship is many-to-one relationship. Each entity in A is associated with at most one entity in B and each entity in B is associated with 0 (or) more entities in A.



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

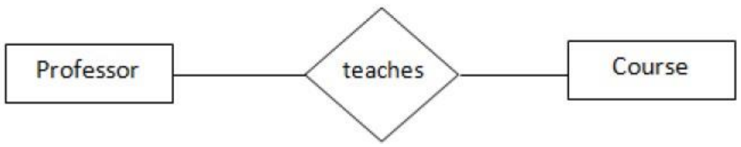


4. **Many-to-Many relationship:** If more than one instance of an entity on the left and more than one instance of an entity on the right can be associated with the relationship, then it depicts many-to-many relationship. Each entity in A is associated with 0 (or) more entities in B and each entity in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

m : n



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.

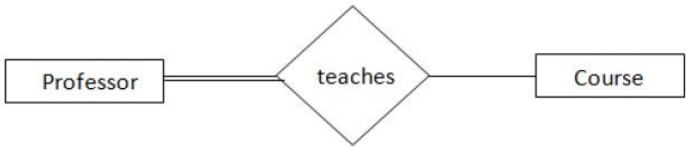
Participation Constraints:

- **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:

Each Professor teaches at least one course.

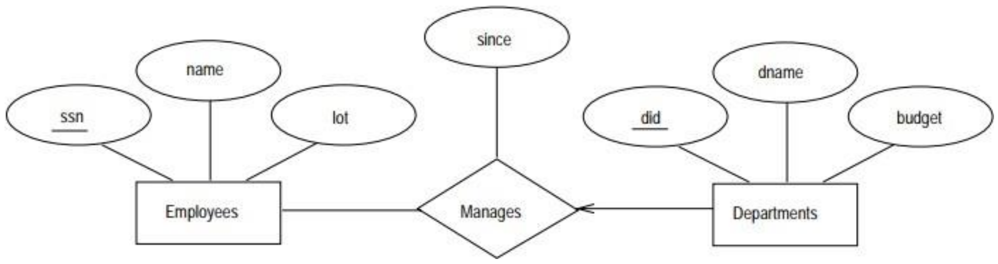
min=1	(Total Participation)
max=many	(No key)



3.6 Additional Features Of The ER Model

Key Constraints

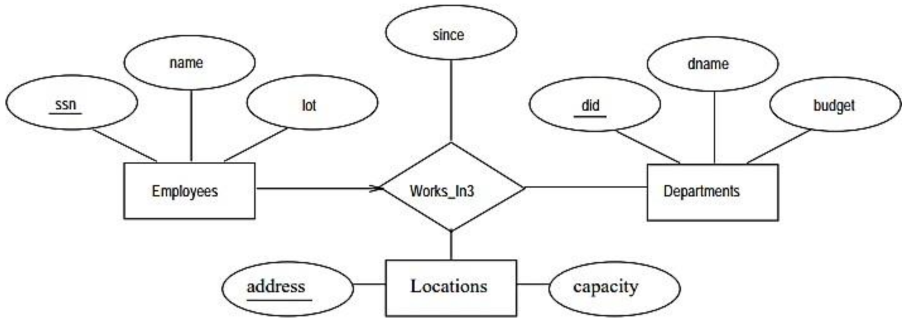
Consider a relationship set called Manages between the Employees and Departments entity sets such that each department has at most one manager, although a single employee is allowed to manage more than one department. The restriction that each department has at most one manager is an example of a **key constraint**, and it implies that each Departments entity appears in at most one Manages relationship in any allowable instance of Manages. This restriction is indicated in the ER diagram of below Figure by using an arrow from Departments to Manages. Intuitively, the arrow states that given a Departments entity, we can uniquely determine the Manages relationship in which it appears.



Key Constraints for Ternary Relationships

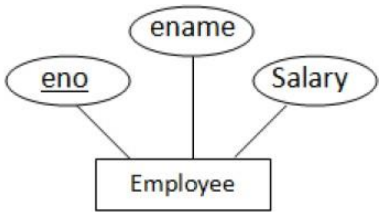
If an entity set E has a key constraint in a relationship set R, each entity in an instance of E appears in at most one relationship in (a corresponding instance of) R. To indicate a key constraint on entity set E in relationship set R, we draw an arrow from E to R.

Below figure show a ternary relationship with key constraints. Each employee works in at most one department, and at a single location.

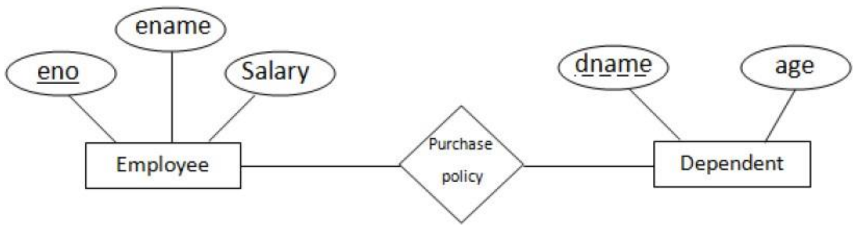


Weak Entities

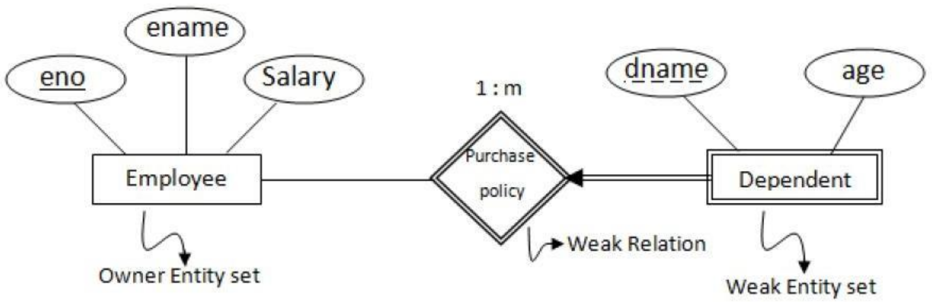
Strong Entity set: If each entity in the entity set is distinguishable or it has a key then such an entity set is known as strong entity set.



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 weak 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.

Dependents is an example of a **weak entity set.** A weak entity can be identified uniquely only by considering some of its attributes in conjunction with the primary key of another entity, which is called the identifying owner.
The following restrictions must hold:

- The owner entity set and the weak entity set must participate in a one-to-many relationship set (one owner entity is associated with one or more weak entities, but each weak entity has a single owner). This relationship set is called the **identifying relationship set** of the weak entity set.
- The weak entity set must have total participation in the identifying relationship set

3.7 E-R Diagrams Implementation

Now we are in a position to write the ER diagram for the Company database which was introduced in the beginning of this unit. The readers are strictly advised to follow the steps shown in this unit to design an ER diagram for any chosen problem.

Step 1: Identify the Strong and Weak Entity Sets

After careful analysis of the problem we come to a conclusion that there are four possible entity sets as shown below:

1. Employees	Strong Entity Set
2. Departments	Strong Entity Set
3. Projects	Strong Entity Set
4. Dependents	Weak Entity Set

Step 2: Identify the Relevant Attributes

The next step is to get all the attributes that are most applicable for each entity set. Do this work by considering each entity set in mind and also the type of attributes. Next job is to pick the primary key for strong entity sets and partial key for weak entity sets.

Example: Following are the attributes:

1. Employees	SSN. Name, Addr, DateOfBirth, Sex, Salary
2. Departments	DNo. DName, DLocation
3. Projects	PNo. PName, PLocation
4. Dependents (weak)	DepName, DateOf Birth, Sex, Relationship

The underlined attributes are the primary keys and DepName is the partial key of Dependents. Also, DLocation may be treated as a multivalued attribute.

Step 3: Identify the Relationship Sets

In this step we need to find all the meaningful relationship sets among possible entity sets. This step is very tricky, as redundant relationships may lead to complicated design and in turn a bad implementation.

Example: Let us show below what the possible relationship sets are:

1. Employees and Departments	WorksFor
2. Employees and Departments	Manages
3. Departments and Projects	Controls
4. Projects and Employees	WorksOn
5. Dependents and Employees	Has
6. Employees and Employees	Supervises

Some problems may not have recursive relationship sets but some do have. In fact, our Company database has one such relationship set called Supervises. You can complete this step adding possible descriptive attributes of the relationship sets (Manages has StartDate and WorksOn has Hours).

Step 4: Identify the Cardinality Ratio and Participation Constraints

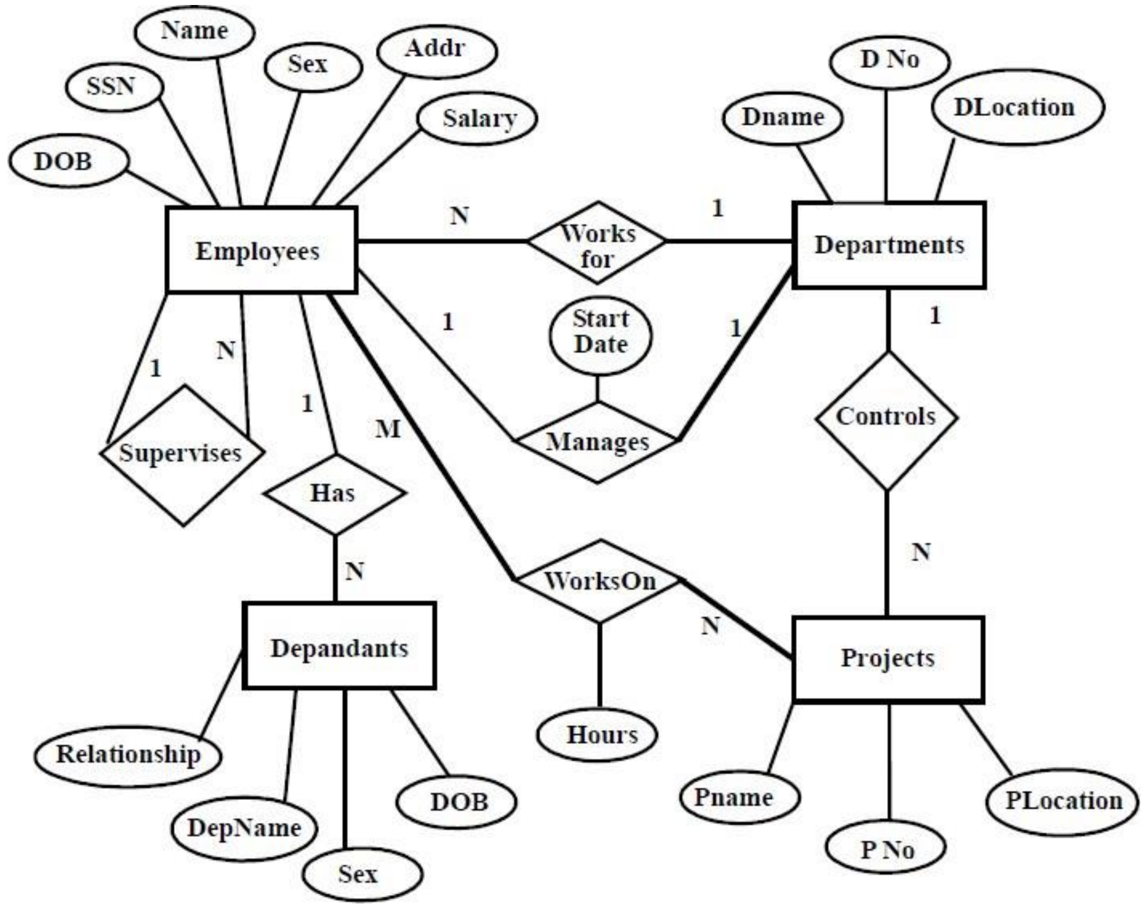
This step is relatively a simple one. Simply apply the business rules and your common sense. So, we write the structural constraints for our example as follows:

1. WorksFor	N: 1 Total on either side
2. Manages	1: 1 Total on Employees and Partial on Departments side
3. Controls	1: N Total on either side
4. WorksOn	M: N Total on either side
5. Has	1: M Total on Dependents and Partial on Employees

Step 5: Identify the IS-A and Has-A Relationship Sets

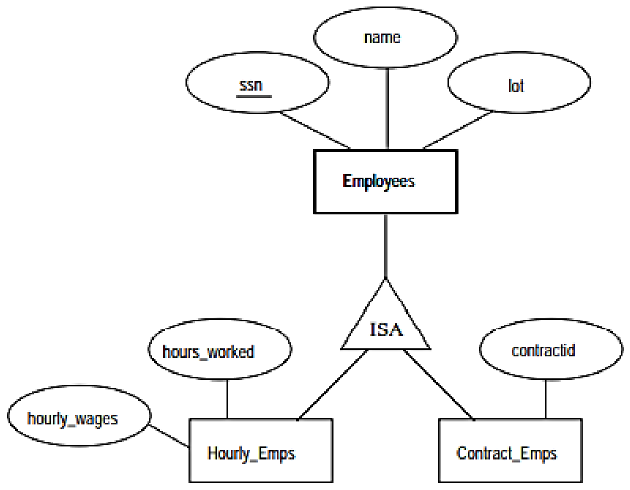
The last step is to look for “is-a” and “has-a” relationships sets for the given problem. As far as the Company database is concerned, there are no generalization and aggregation relationships in the Company database.

The complete single ER diagram by combining all the above five steps is shown in figure



3.8 Class Hierarchies

To classify the entities in an entity set into subclass entity is known as class hierarchies. Example, we might want to classify Employees entity set into subclass entities Hourly-Emps entity set and Contract-Emps entity set to distinguish the basis on which they are paid. Then the class hierarchy is illustrated as follows.



This class hierarchy illustrates the inheritance concept. Where, the subclass attributes ISA (read as : is a) super class attributes; indicating the "is a" relationship (inheritance concept).Therefore, the attributes defined for a Hourly-Emps entity set are the attributes of Hourly-Emps plus attributes of Employees (because subclass can have superclass properties). Likewise the attributes defined for a Contract-Emps entity set are the attributes of Contract-Emps plus attributes of Employees.

Class Hierarchy based on Sub-super Set

1. **Specialization:** Specialization is the process of identifying subsets (subclasses) of an entity set (superclass) that share some special distinguishable characteristic. Here, the superclass (Employee) is defined first, then the subclasses (Hourly-Emps, Contract-Emps, etc.) are defined next.

   In short, Employees is specialized into subclasses.

2. **Generalization:** Generalization is the process of identifying (defining) some generalized (common) characteristics of a collection of (two or more) entity sets and creating a new entity set that contains (possesses) these common characteristics. Here, the subclasses (Hourly-Emps, Contract-Emps, etc.) are defined first, then the Superclass (Employee) is defined, next.

   In shortly, Hourly-Emps and Contract-Emps are generalized by Employees.

Class Hierarchy based on Constraints

1. **Overlap constraints:** Overlap constraints determine whether two subclasses are allowed to contain the same entity.

Example: Can Akbar be both an Hourly-Emps entity and a Contract-Emps entity?
The answer is, No.
Other example, can Akbar be both a Contract-Emps entity and a Senior-Emps entity (among them)?
The answer is, Yes. Thus, this is a specialisation hierarchy property. We denote this by writing "Contract-Emps OVERLAPS Senior-Emps".

2. **Covering Constraints:** Covering constraints determine whether the entities in the subclasses collectively include all entities in the superclass.

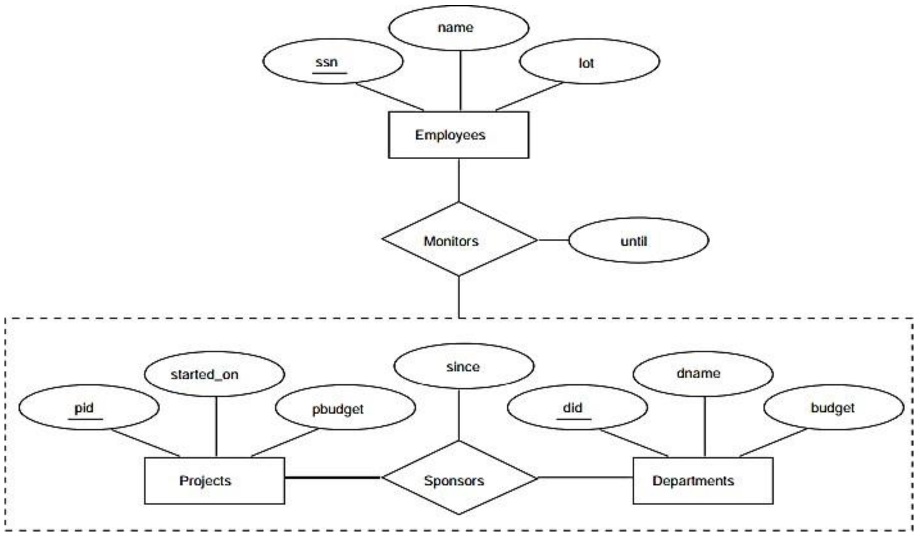
Example: Should every Employee be a Hourly-Emps or .Contract-Emps?
The Answer is, No. He can be a Daily-Emps.
Other example, should every Motor-vehicle (superclass) be a Bike (subclass) or a Car (subclass)?
The Answer is YES. Thus generalization hierarchies property is that every instance of a superclass is an instance of a subclass.
We denote this by writing " Bikes and Cars COVER Motor-vehicles".

3.9 Aggregation

Aggregation allows us to indicate that a relationship set (identified through a dashed box) participates in another relationship sets. That is, a relationship set in an association between entity sets. Sometimes we have to model a relationship between a collection of entities and relationships.

Example: Suppose that we have an entity set called Project and that each Project entity is sponsored by one or more departments. Thus, the sponsors relationship set captures this information but, a department that sponsors a project, might assign employees to monitor the sponsorship. Therefore, Monitors should be a relationship set that associates a sponsors relationship (rather than a Project or Department entity) with an Employees entity. However, again we have to define relationships to associate two or more entities.

Use of Aggregation
We use an aggregation, when we need to express a relationship among relationships. Thus, there are really two distinct relationships, Sponsors and Monitors, each with its own attributes.



3.10 Conceptual Database Design With The ER Model (ER Design Issues)

The following are the ER design issues:

1. Use entry sets attributes

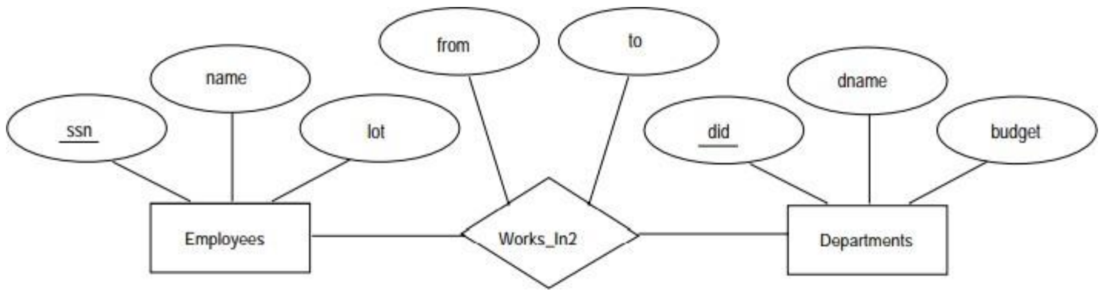
2. Use of Entity sets or relationship sets

3. Binary versus entry relationship sets

4. Aggregation versus ternary relationship.

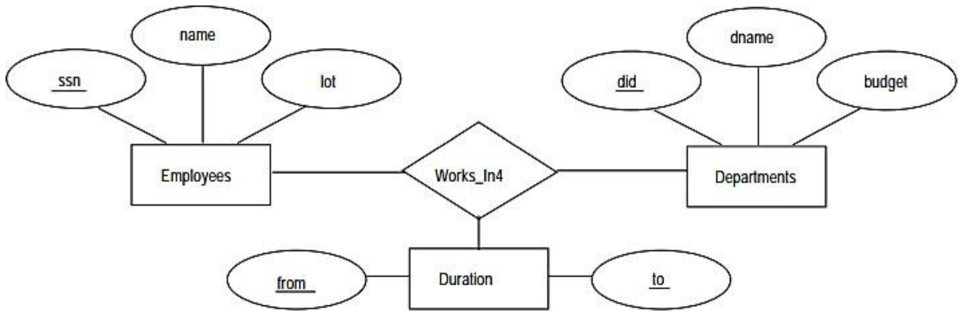
1. Use of Entity Sets versus Attributes

Consider the relationship set (called Works In2) shown in Figure



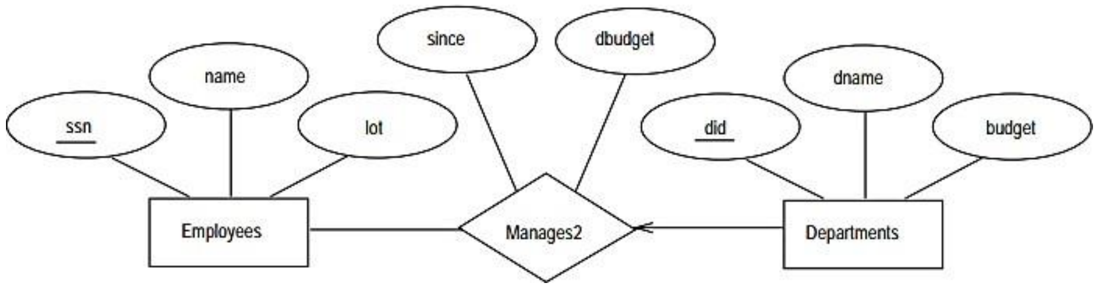
Intuitively, it records the interval during which an employee works for a department. Now suppose that it is possible for an employee to work in a given department over more than one period.

This possibility is ruled out by the ER diagram's semantics. The problem is that we want to record several values for the descriptive attributes for each instance of the Works_In2 relationship. (This situation is analogous to wanting to record several addresses for each employee.) We can address this problem by introducing an entity set called, say, Duration, with attributes from and to, as shown in Figure



2. Entity versus Relationship

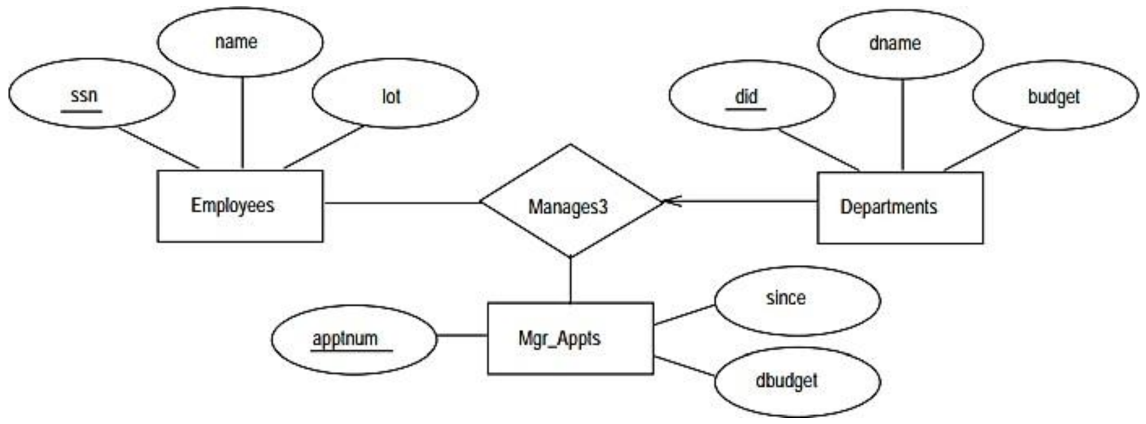
Consider the relationship set called Manages that each department manager is given a discretionary budget (dbudget), as shown in below figure, in which we have also renamed the relationship set to Manages2.



There is at most one employee managing a department, but a given employee could manage several departments; we store the starting date and discretionary budget for each manager-department pair. This approach is natural if we assume that a manager receives a separate discretionary budget for each department that he or she manages.

But what if the discretionary budget is a sum that covers all departments managed by that employee? In this case each Manages2 relationship that involves a given employee will have the same value in the dbudget field. In general such redundancy could be significant and could cause a variety of problems. Another problem with this design is that it is misleading.

We can address these problems by associating dbudget with the appointment of the employee as manager of a group of departments. In this approach, we model the appointment as an entity set, say Mgr_Appts, and use a ternary relationship, say Man ages3, to relate a manager, an appointment, and a department. The details of an appointment (such as the discretionary budget) are not repeated for each department that is included in the appointment now, although there is still one Manages3 relationship instance per such department. Further, note that each department has at most one manager, as before, because of the key constraint. This approach is illustrated in below Figure.

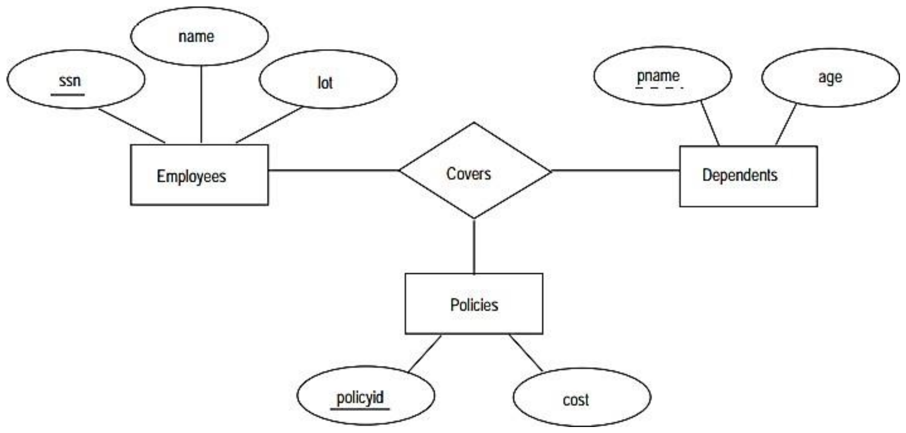


3. Binary versus Ternary Relationships

Consider the ER diagram shown in below figure. It models a situation in which an employee can own several policies, each policy can be owned by several employees, and each dependent can be covered by several policies.

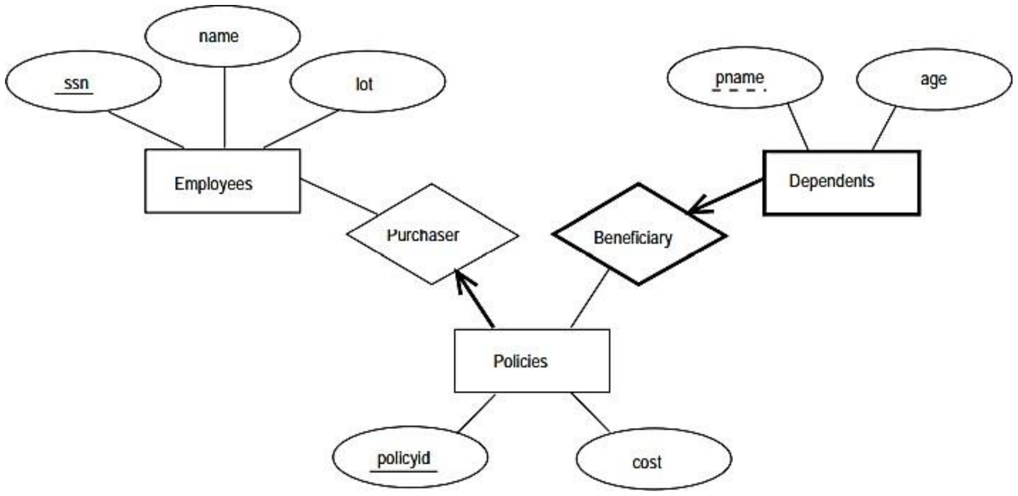
Suppose that we have the following additional requirements:

- 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 first requirement suggests that we impose a key constraint on Policies with respect to Covers, but this constraint has the unintended side effect that a policy can cover only one dependent. The second requirement suggests that we impose a total participation constraint on Policies. This solution is acceptable if each policy covers at least one dependent. The third requirement forces us to introduce an identifying relationship that is binary (in our version of ER diagrams, although there are versions in which this is not the case).

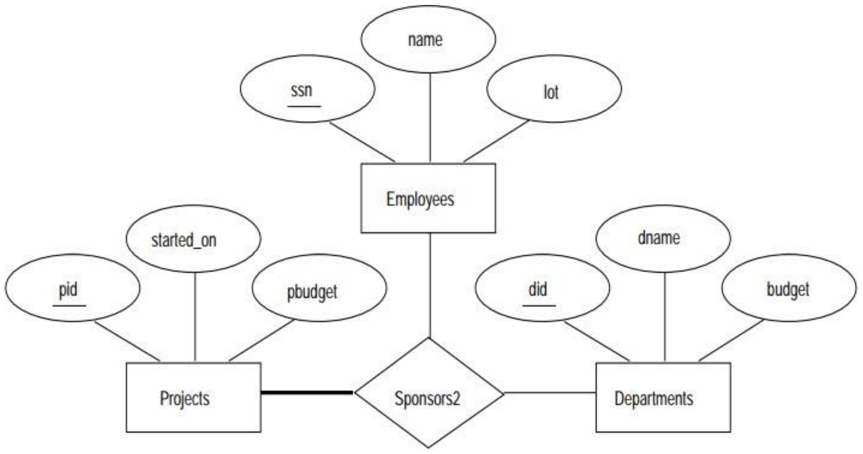
Even ignoring the third point above, the best way to model this situation is to use two binary relationships, as shown in below figure.



4. Aggregation versus Ternary Relationships

The choice between using aggregation or a ternary relationship is mainly determined by the existence of relationship that relates a relationship set to an entity set (or second relationship set). The choice may also be guided by certain integrity constraints to we want to express.

Consider the constraint that each sponsorship (of a project by a department) be monitored by at most one employee. We cannot express this constraint in terms of the Sponsors2 relationship set. Also we can express the constraint by drawing an arrow from the aggregated relationship. Sponsors to the relationship Monitors. Thus, the presence of such a constraint serves as another reason for using aggregation rather than a ternary relationship set.



Review Questions

1. Define the following terms and give examples

   (i) cardinality (ii)unary relationships (iii)aggregation (iv)specialization

2. What is Entity set? and also define Relationship set. List and explain the symbols used to draw ER Diagram.

3. Design a database for an airline. The database must keep track of customers and their

reservations, flights and their status, seat assignments on individual flights, and the schedule and routing of future flights. Your design should include an E-R diagram, a set of relational schemas, and a list of constraints, including primary-key and foreign-key constraints.

4. Discuss the representation of total participation and multivalued attribute in an E/R diagram.
5. What is an Entity Relationship diagram and why it is useful?
6. What is a weak entity in ER diagram?
7. Give the diagrammatic representation of recursive relationship in an ER diagram and also explain the importance of role names in representing a recursive relationship by taking a real time example.
8. Consider a database used to record the marks that students get in different exams of different course offerings.

   a. Construct an E-R diagram that models exams as entities, and uses a ternary relationship, for the above database.
   b. Construct an alternative E-R diagram that uses only a binary relationship between students and course-offerings. Make sure that only one relationship exists between a particular student and course-offering pair, yet you can represent the marks that a student gets in different exams of a course offering.
9. Explain about relationship sets in ER model with examples.
10. Explain about ER model design issues.

References:

- Raghurama Krishnan, Johannes Gehrke, *Database Management Systems*, 3rd Edition, Tata McGraw Hill.
- C.J. Date, *Introduction to Database Systems*, Pearson Education.
- Elmasri Navrate, *Fundamentals of Database Systems*, Pearson Education.

UNIT-3
PART- II SQL

- **Creating Tables with relationship**
- **Implementation of key and integrity constraints**
- **Relational Set Operations**
- **Implementation of different types of Joins**
- **Views(Updatable and Non Updatable)**
- **Grouping, Aggregation, Ordering**
- **Nested queries, Sub queries**

I.I NTRODUCTION TO CREATING TABLES WITH RELATIONSHIP

Creating tables using CREATE command:

This command is used to create a database and its objects such as Tables, Views, Procedures, Triggers etc. It defines each column of the table uniquely. Each column has minimum of **three** attributes, a column name , data type and size.

```
Syntax:     CREATE TABLE    <table_name>
              (
               column _name 1  DATATYPE 1 (SIZE),
                 column _name 2   DATATYPE 2 (SIZE),
                    :
               column _name n   DATATYPE N (SIZE) );
```

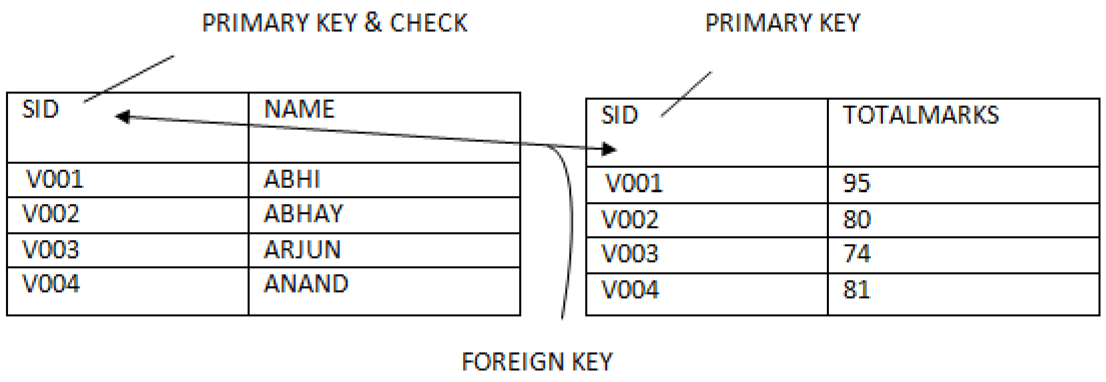
CREATING TABLES WITH RELATIONSHIP

When we want to create tables with relationship , we need to use **Referential integrity constraints**. The referential integrity constraint **enforces relationship** between tables.
-It **designates** a column or combination of columns as a **Foreign key**.
-The foreign key establish a **relationship** with a specified primary or unique key in another table called the **Referenced key**.
- When referential integrity is enforced , it prevents from..
  1) Adding records to a related table if there is no associated record in the primary table.
  2) Changing values in a primary table that result in orphaned records in a related table.
  3) Deleting records from a primary table if there are matching related records.

Note: The table containing the foreign key is called the child table and the table containing the referenced key is called the Parent table.

```
SYNTAX:   CREATE TABLE <tablename>(
                                      col_name1  datatype[size] ,
                                      col_name2  datatype[size] ,
                                                   :
                                      col_name n datatype[size],
 FOREIGN KEY(column_name)  REFERENCES   <parent_table_name>(column_name));
```

```
EX:    SQL> CREATE TABLE marks(
sid VARCHAR2(4),
 marks NUMBER(3),
PRIMARY KEY(sid),
FOREIGN KEY(sid) REFERENCES student1(sid));
```



SID	NAME
V001	ABHI
V002	ABHAY
V003	ARJUN
V004	ANAND

SID	TOTALMARKS
V001	95
V002	80
V003	74
V004	81

II.IMPLEMENTATION OF KEY AND INTEGRITY CONSTRAINTS

Data constraints: All business of the world run on business data being gathered, stored and analyzed. Business managers determine a set of business rules that must be applied to their data prior to it being stored in the database/table of ensure its integrity.
For instance , no employee in the sales department can have a salary of less than Rs.1000/- . Such rules have to be enforced on data stored. If not, inconsistent data is maintained in database.

Note: It is used to impose business rules on DBs.
It allows to enter only valid data.

Various types of Integrity Constraints:

Integrity constraints are the **rules** in real life, which are to be imposed on the data. If the data is not satisfying the constraints then it is considered as **inconsistent**. These rules are to be enforced on data because of the presence of these rules in real life. These rules are called integrity constraints. Every DBMS software must enforce integrity constraints, otherwise inconsistent data is generated.

You can use constraints to do the following:

- To **prevent invalid data entry** into tables.
- To **Enforce rules** on the data in a table whenever a row is inserted, updated, or deleted from that table. The constraint must be satisfied for the operation to succeed.
- To **Prevent the deletion** of a record from a table if there are dependencies.

<u>Example for Integrity Constraints :-</u>

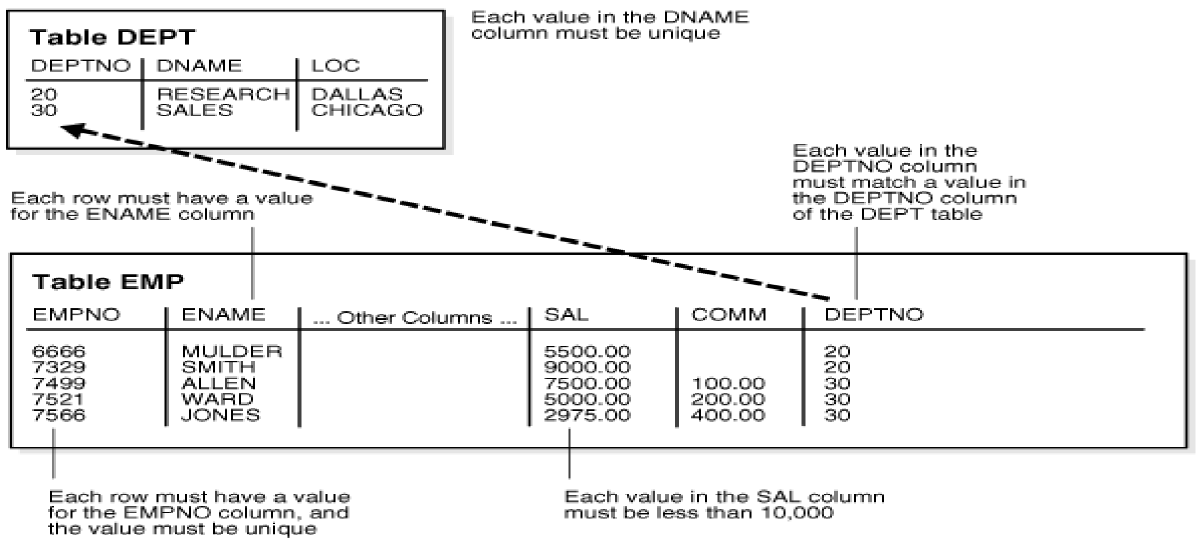


Table DEPT
DEPTNO | DNAME | LOC
20 | RESEARCH | DALLAS
30 | SALES | CHICAGO
Each value in the DNAME column must be unique
Each value in the DEPTNO column must match a value in the DEPTNO column of the DEPT table
Each row must have a value for the ENAME column
Table EMP
EMPNO | ENAME | ... Other Columns ... | SAL | COMM | DEPTNO
6666 | MULDER | | 5500.00 | | 20
7329 | SMITH | | 9000.00 | | 20
7499 | ALLEN | | 7500.00 | 100.00 | 30
7521 | WARD | | 5000.00 | 200.00 | 30
7566 | JONES | | 2975.00 | 400.00 | 30
Each row must have a value for the EMPNO column, and the value must be unique
Each value in the SAL column must be less than 10,000

Constraints are categorized as follows.

1.Domain integrity constraints - A domain means a set of values assigned to a column. i.e A set of permitted values. Domain constraints are handled by

→Defining proper **data type**
→Specifying **not null** constraint
→Specifying **check** constraint.
→Specifying **default** constraint

Not null –indicates that a column cannot store NULL value.
Check – Ensures that the value in column meets a specific condition.
Default- prevents **null** value in column when value is not provided in column by user. So, it assigns default value or globally assigned value.

2. **Entity integrity constraints – are TWO types**
Unique constraint –It defines a Entity or column as a UNIQUE for particular table.
And ensures that each row of a column must have a UNIQUE value or name.
Primary key constraint – This avoids **duplicate** and **null** values. It combination of a NOT NULL and UNIQUE.
3.Referential integrity constraints
Foreign key –indicates the relationship between child and parent tables.
This Constraint are always attached to a column not a table.
We can add constraints in **two** ways.

<u>Column level :-</u>

→ Constraint is declared immediately declaring column.
→ Define with each column
→ Use column level to declare constraint for single column.
→ Composite key cannot be defined at column level.

<u>Table level :-</u>

→constraint is declared after declaring all columns.
→use table level to declare constraint for combination of columns.(i.e composite key)
→ **not null** cannot be defined.

Another type is possible at **Alter level**

→ constraint is declared with ALTER command.
→When we use this , make sure that the table should not contain data.

To add these constraints , we can use constraint with label or with out label.

TWO BASIC TYPES ------- 1. Constraint WITH NAME
2.constraints WITHOUT NAME.

i) Declaring Constraint at "TABLE" level (Constraints with label)

Syntax :- **CREATE TABLE <table name>**

```
            (
                col_name1  DATATYPE(SIZE) ,
                        …..,
                col_nameN  DATATYPE(SIZE)    ,
                 CONSTRAINT  <cons_lable>    NAME _OF_ THE_CONSTRAINT [column_list] );
```

ii) Declaring Constraint at "Column" level (Constraints with label)

Syntax :-

col_name DATATYPE(SIZE) CONSTRAINT <cons_lable> NAME _OF_ THE_CONSTRAINT

iii) Declaring Constraint at "TABLE" level (Constraints without label)

Syntax :-

```
CREATE TABLE <table name>
(
col_name1  DATATYPE(SIZE) ,
                    …..,
col_nameN  DATATYPE(SIZE)    ,
NAME _OF_ THE_CONSTRAINT [column_list] );
```

iv) Declaring Constraint at "Column" level (Constraints without label)

Syntax :-

col_name DATATYPE(SIZE) NAME _OF_ THE_CONSTRAINT

v) Adding Constraint to a table at "Alter" level (Constraints with label)

A constraint can be added to a table at any time after the table was created by using ALTER TABLE statement , using **ADD** clause.

Syntax:

ALTER TABLE <table_name> ADD CONSTRAINT cont_label NAME _OF_ THE_CONSTRAINT (column);

vi) Declaring Constraint at "Alter" level (Constraints without label)

Syntax:
ALTER TABLE <table_name> ADD NAME _OF_ THE_CONSTRAINT (column);
Note:' Constraint ' clause is not required when constraints declared without a label.

1.1 NOT NULL:

- It ensures that a table column cannot be left empty.
- Column declared with NOT NULL is a mandatory column i.e data must be entered.
- The NOT NULL constraint can only be applied at column level.
- It allows DUPLICATE data.
- Used to avoid null values into columns.

NOTE: It is applicable at **COLUMN LEVEL only.**
SYNTAX: **column_name   DATATAYPE[SIZE] NOT NULL**

EX 1 : CREATE TABLE   table_notnull(
sid NUMBER(4) **NOT NULL**, // COLUMN LEVEL
sname VARCHAR2(10));

```
SQL> SELECT *FROM  table_notnull;
      SID    SNAME
- - - - - - - - - - - - - -
      501    GITA
      502    RAJU
      503
      503
      504
```

Here, SID column not allowed any null values and it can allow duplicate values , but sname can allows it.
Ex 2:

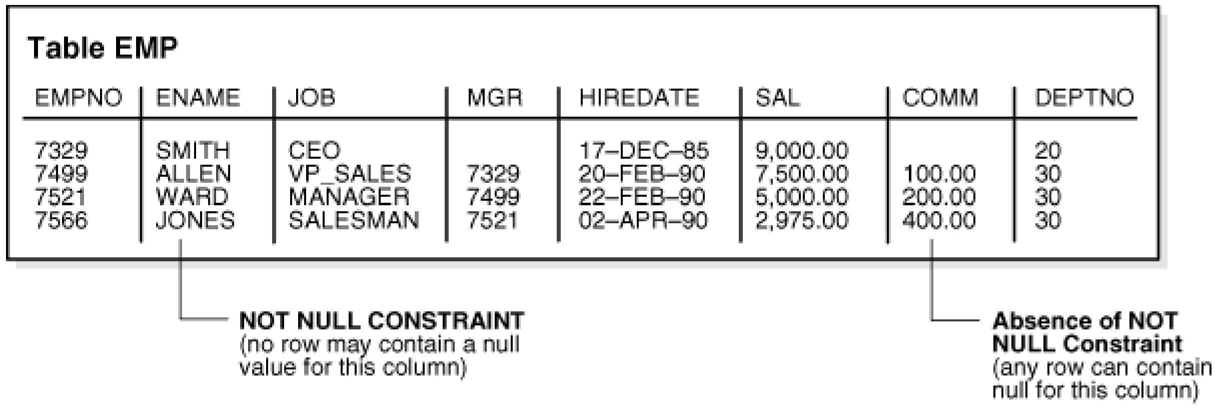


Table EMP

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7329	SMITH	CEO		17–DEC–85	9,000.00		20
7499	ALLEN	VP_SALES	7329	20–FEB–90	7,500.00	100.00	30
7521	WARD	MANAGER	7499	22–FEB–90	5,000.00	200.00	30
7566	JONES	SALESMAN	7521	02–APR–90	2,975.00	400.00	30

NOT NULL CONSTRAINT (no row may contain a null value for this column)
Absence of NOT NULL Constraint (any row can contain null for this column)

1.2 CHECK :

- Used to impose a conditional rule a table column.
- It defines a condition that each row must satisfy.
- Check constraint **validates data** based on a condition .

- Value entered in the column **should not violate the condition**.
- Check constraint allows **null** values.
- Check constraint can be declared at **table** level or **column** level.
- There is no limit to the number of CHECK constraints that can be defined on a condition.

Limitations :-

- Conditions should not contain/not applicable to pseudo columns like ROWNUM, SYSDATE etc.
- Condition should not access columns of another table

//CONSTRAINT @ COLUMN LEVEL

SYNTAX: column_name DATATYPE (SIZE) **CHECK** (condition) // without label

Here, we are creating a table with two columns such as Sid, sname.

```
Ex:    CREATE TABLE  check_column(
     sid VARCHAR2(4) CHECK (sid LIKE 'C%'  AND  LENGTH(sid)=4),     // without label
     sname VARCHAR2(10));
```

Here, **sid** should start with **'C** 'and a length of sid is exactly **4** characters.

```
SQL> SELECT *FROM  check_column;
SID  SNAME
----------
C501 MANI
C502 DHANA
C503 RAVI
C504 RAJA
```

// with label

Syntax: Column_name DATATYPE(SIZE) CONSTRAINT constaint_label CHECK(condtion)

```
CREATE TABLE  check_column (
sid VARCHAR2(4)  CONSTRAINT ck CHECK (sid LIKE 'C%'  AND  LENGTH(sid)=4),
sname VARCHAR2(10) );
```

//CHECK @TABLE LEVEL

```
CREATE TABLE    check_table (
                                        sid VARCHAR2(4) ,
                                       sname VARCHAR2(10),
                                        CHECK (sid LIKE'C%' AND LENGTH(sid)=4),
                                        CHECK(sname LIKE '%A'));
```

Here, **sid** should start with **'C** 'and a length of sid is exactly **4** characters. And **sname** should ends with letter ' A'

```
SQL> SELECT *FROM  check_table;

SID  SNAME
----------
C401 ABHILA
C401 ANITHA
C403 NANDHITHA
C522 LOHITHA
```

// with label

```
CONSTRAINT ck1 CHECK (sid LIKE'C%' AND LENGTH(sid)=4),
CONSTRAINT ck2 CHECK(sname LIKE '%A'));
```

@ ALTER LEVEL
Here, we add check constraint to new table with columns.

```
SQL> CREATE TABLE  check_alter(sid VARCHAR2(4),
                                Sname VARCHAR2(10));
```

//CHECK @ ALTER LEVEL: / / CONSTRAINT WITHOUT NAME
SQL> ALTER TABLE <table name>ADD CHECK (condition);

SQL> ALTER TABLE check_alter ADD CHECK (sid like 'C%');

SYNTAX: **ALTER TABLE <table_name> ADD CONSTRAINT cont_name CHECK(cond);**

```
SQL> ALTER TABLE check_alter  ADD CONSTRAINT ck  CHECK ( sid LIKE 'C%');
```

ANOTHER EXAMPLE FOR TABLE LEVEL CONSTRAINT

Here, We create table with THREE columns

ADD CHECK CONSTRAINT @ TABLE LEVEL (AT THE END OF TABLE DEFINITION)
MARKS IN BETWEEN 0 AND 100.

```
SQL> CREATE TABLE marks2 ( sid VARCHAR2(4),
                           sec VARCHAR2(2),
                           marks NUMBER(3),
                           CHECK(marks>0 AND marks<=100) );
```

DROP @ CHECK CONSTRAINT

SYNTAX: ALTER TABLE <table_name> DROP CONSRAINT cont_name;

```
SQL> ALTER TABLE  check_table     DROP  CONSTRAINT ck;
```

DEFAULT

-If values are not provided for table column , default will be considered.
-This prevents NULL values from entering the columns , if a row is inserted without a value for a column.
-The default value can be a literal, an expression, or a SQL function.
-The default expression must match the data type of the column.
- The DEFAULT constraint is used to provide a default value for a column.

-The default value will be added to all new records IF no other value is specified.

Syntax: Column_name datatype (size) **DEFAULT** <value/expression/function>

Ex: MIDDLENAME VARCHAR(10) **DEFAULT** 'UNAVAILABLE'

CONTACTNO NUMBER(10) **DEFAULT** 9999999999

This defines what value the column should use when no value has been supplied explicitly when inserting a record in the table.

```
CREATE TABLE  tab_default(  sid NUMBER(10),
                            contactno number(10)  DEFAULT 9999999999);
```

Add data to table:

```
Insert into tab_default (sid,contactno) values(501,9493949312);
Insert into tab_student(sid) values(502);
Insert into tab_student(sid) values(503);
Insert into tab_student(sid,sname) values(504,9393949412);
```

Select * from tab_default;

```
SID  CONTACTNO
---------- ------------------
501   9493949312
502   9999999999
503   9999999999
504   9393949412
```

2.1. UNIQUE

- Columns declared with UNIQUE constraint **does not accept duplicate values**.
- One table can have a number of unique keys.
- Unique key can be defined on more than one column i.e composite unique key
- A composite key UNIQUE key is always defined at the table level only.
- By default UNIQUE columns accept null values unless declared with NOT NULL constraint
- Oracle automatically creates UNIQUE index on the column declared with UNIQUE constraint

- UNIQUE constraint can be declared at column level and table level.

UNIQUE@ COLUMN LEVEL
SYNTAX: column_name DATA_TYPE(SIZE) UNIQUE

```
CREATE TABLE table_unique(
                                        sid NUMBER(4) UNIQUE,
                                        sname VARCHAR2(10));
```

//UNIQUE @ TABLE LEVEL
SYNTAX: UNIQUE(COLUMN_LIST);

```
CREATE TABLE table_unique2(
                    sid NUMBER(4),
                    sname VARCHAR2(10) ,
                    UNIQUE(sid,sname));
```

```
SQL> SELECT *FROM  TABLE_UNIQUE2;
     SID SNAME
--------------
     401 RAMU
     402 SITA                    // Here , these two records are distinct not the same.
     402 GITHA
     403 GITHA
     404 RAMU
```

Unique @ ALTER level:

```
Alter table table_unique ADD  UNIQUE (sid)    // with out label
Alter table table_unique ADD CONSTRAINT  uq  UNIQUE(sid) // with label
```

DROP UNIQUE @ TABLE LEVEL
SQL> ALTER TABLE table_unique2 DROP UNIQUE(sid,sname);

Now , we removed unique constraint , so now this table consists duplicate data.
//UNIQUE@ ALTER LEVEL (here, the table contains duplicates, so it is not works)
//delete data from table_unique2
SQL> DELETE FROM table_unique2;

PRIMARY KEY constraint :-

PRIMARY KEY is one of the **candidates** key , which uniquely identifies a record in a table.

-used to define key column of a table.

-it is provided with an automatic index.

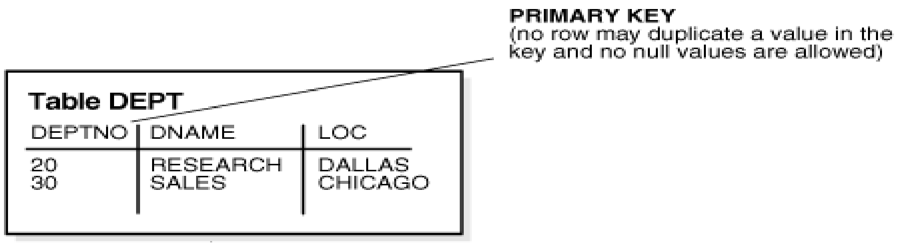
-A primary key constraint combines a NOT NULL and UNIQUE behavior in one declaration.

Characterstics of PRIMARY KEY :-

➔ There should be at the most one Primary Key or Composite primary key per table.

- ➔ PK column do not accept null values.
- ➔ PK column do not accept duplicate values.
- ➔ RAW,LONG RAW,VARRAY,NESTED TABLE,BFILE columns cannot be declared with PK
- ➔ If PK is composite then uniqueness is determined by the combination of columns.
- ➔ A composite primary key cannot have more than 32 columns
- ➔ It is recommended that PK column should be short and numeric.
- ➔ Oracle automatically creates Unique Index on PK column

EX:


PRIMARY KEY
(no row may duplicate a value in the key and no null values are allowed)
Table DEPT
DEPTNO
DNAME
LOC
20
30
RESEARCH
SALES
DALLAS
CHICAGO

// PRMARY KEY @ COLUMN LEVEL
SYNTAX : column_name DATA_TYPE(SIZE) PRIMARY KEY

```
SQL> CREATE TABLE  student1 (
                              sid VARCHAR2(4) PRIMARY KEY
                                                  CHECK (sid LIKE 'V%' AND LENGTH(sid)=4 ) ,
                               name VARCHAR2(10));

SQL> DESC  student1;
 Name                          Null?    Type
 -----------------------------------------------------------
 SID                          NOT NULL VARCHAR2(4)
 NAME                                   VARCHAR2(10)
```

CASE 2: ADD PRIMARY KEY @ ALTER LEVEL

```
SQL> CREATE TABLE student2( sid VARCHAR2(4),
                                     name VARCHAR2(10));
```

SYNTAX: ALTER TABLE <tablename> ADD PRIMARY KEY (col_name);

SQL> ALTER TABLE student2 ADD PRIMARY KEY(sid);

Table altered.

```
SQL> DESC student2;
 Name                          Null?    Type
 ----------------------------------------- -------- ----------------------------
 SID                          NOT NULL VARCHAR2(4)
 NAME                                   VARCHAR2(10)
```

CASE 3 : ADD PRIMARY KEY @ TABLE LEVEL
here, we can create a simple and composite primary keys;

```
SYNTAX:   CREATE TABLE  < tablename>( col_name1    datatype[size],
                                      col_name2  datatype[size],
                                                :
                                       col_namen datatype[size],
                                      PRIMARY KEY (col_name);
```

//SIMPLE PRIMARY KEY @ TABLE LEVEL

```
          SQL> CREATE TABLE student3(
                                      sid VARCHAR2(4),
                                      name VARCHAR2(10),
                                      marks NUMBER(3),
                                      PRIMARY KEY(sid) );
```

SQL> DESC student3;

```
 Name                          Null?    Type
 ---------------------------------------------------
 SID                           NOT NULL VARCHAR2(4)
 NAME                                   VARCHAR2(10)
 MARKS                                  NUMBER(3)
```

// COMPOSITE PRIMARY KEY @ TABLE LEVEL

SYNTAX:

```
CREATE TABLE  < tablename>( col_name1    datatype[size],
                            col_name2   datatype[size],
                                                :
                            col_namen  datatype[size],
                   PRIMARY KEY (col_name1,col_name2….colmn_name n);
```

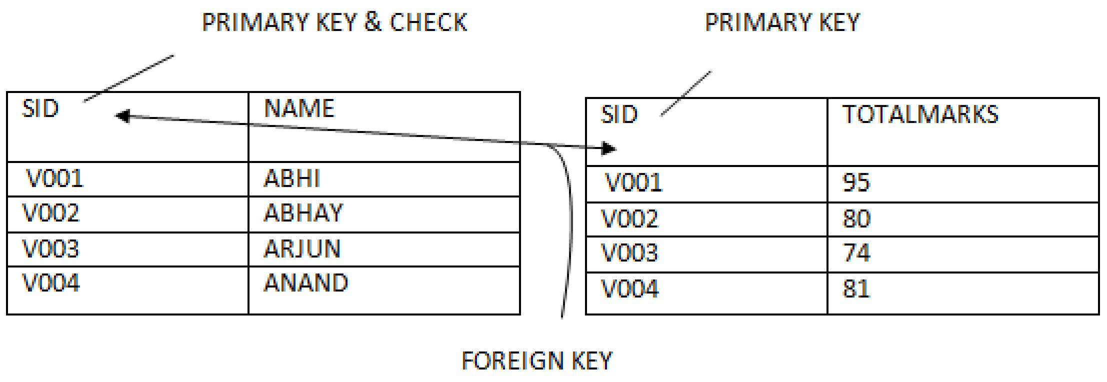
```
SQL> CREATE TABLE student4(
                            sid VARCHAR2(4),
                             name VARCHAR2(10),
                             marks NUMBER(3),
                             PRIMARY KEY(sid,name) ); // WITH OUT LABEL


    ( CONSTRAINT  pk PRIMARY KEY(sid,name)    // WITH LABEL)
```

```
SQL> DESC STUDENT4;
 Name                          Null?        Type
 ----------------------------------------- -------- ----------------------------
 SID                           NOT NULL VARCHAR2(4)
 NAME                          NOT NULL VARCHAR2(10)
 MARKS                                  NUMBER(3)
```

FOREIGN KEY Constraint:-

- Foreign key is used to establish relationship between tables.
- Foreign key is a column in one table that refers primary key/unique columns of another or same table.
- Values of foreign key should match with values of primary key/unique or foreign key can be null.
- Foreign key column allows null values unless it is declared with NOT NULL.
- Foregin key column allows duplicates unless it is declared with UNIQUE
- By default oracle establish 1:M relationship between two tables.
- To establish 1:1 relationship between two tables declare foreign key with unique constraint
- Foreign key can be declared at column level or table level.
- Composite foreign key must refer composite primary key or Composite unique key.

EX: **TABLES: STYDENT 1 & MARKS 1**



SID	NAME
V001	ABHI
V002	ABHAY
V003	ARJUN
V004	ANAND

SID	TOTALMARKS
V001	95
V002	80
V003	74
V004	81

<u>CHILD TABLE ‘ MARKS1’ :</u>

ADDING PRIMARY AND CHECK CONSTRAINT @ CREATE LEVEL

```
SQL> CREATE TABLE marks1(
          sid  VARCHAR2(4) PRIMARY KEY CHECK( sid LIKE 'V%' AND LENGTH(sid)=4),
                                   marks NUMBER(3) );
```

// **ADDING PRIMARY AND FOREIGN KEY @ TABLE/ CREATE LEVEL**

SYNTAX: CREATE TABLE <tablename>(

```
                                    col_name1  datatype[size] ,
                                    col_name2  datatype[size] ,
                                                         :
                                    col_name n datatype[size],
FOREIGN KEY(column_name) REFERENCES    <parent_table_name>(column_name));
```

```
EX:     SQL> CREATE TABLE marks3(
                         sid VARCHAR2(4),
                          marks NUMBER(3),
                         PRIMARY KEY(sid),
                         FOREIGN KEY(sid) REFERENCES student1(sid));
```

```
SQL> DESC MARKS3;
 Name                                      Null?    Type
 ----------------------------------------- -------- ----------------------------
 SID                                       NOT NULL VARCHAR2(4)
 MARKS                                              NUMBER(3)
```

Query : ADD CHECK CONSTRAINT @ ALTER LEVEL (ON EXISTING TABLE)
MARKS IN BETWEEN 0 AND 100.

```
SQL> ALTER TABLE marks3  ADD  CHECK ( marks>0 AND marks< =100 );
```

ADD CHECK CONSTRAINT @ TABLE LEVEL (AT THE END OF TABLE DEFINITION)
MARKS IN BETWEEN 0 AND 100.

```
SQL> CREATE TABLE marks3 ( sid VARCHAR2(4),
                          sec VARCHAR2(2),
                          marks NUMBER(3),
                          CHECK(marks>0 AND marks<=100) );
```

//ADDING FOREIGN KEY @ ALTER LEVEL

```
SQL> ALTER TABLE marks1 ADD FOREIGN KEY (sid) REFERENCES
STUDENT1(sid);
SQL> desc marks1;
 Name                                      Null?    Type
 ----------------------------------------- -------- ----------------------------
 SID                                       NOT NULL VARCHAR2(4)
 MARKS                                              NUMBER(3)
```

<u>Note :-</u>

→ PRIMARY KEY cannot be dropped if it referenced by any FOREIGN KEY constraint.

→If PRIMARY KEY is dropped with CASCADE option then along with PRIMARY KEY referencing FOREING KEY is also dropped.

→PRIMARY KEY column cannot be dropped if it is referenced by some FOREIGN KEY.

→PRIMARY KEY table cannot be dropped if it is referenced by some FOREIGN KEY.

→PRIMARY KEY table cannot be truncated if it is referenced by some FOREIGN KEY.

Note:: Once the primary key and foreign key relationship has been created then you can not remove any parent record if the dependent childs exists.

USING ON DELETE CASCADE

By using this clause you can remove the parent record even if childs exists. Because when ever you remove parent record oracle automatically removes all its dependent records from child table, if this clause is present while creating foreign key constraint.

Ex: Consider twe tables dept(parent) and emp(child) tables.
TABLE LEVEL

SQL> create table emp(empno number(2), ename varchar(10), deptno number(2), primary key(empno), foreign key(deptno) references dept(deptno) on delete cascade); // without label

SQL> create table emp(empno number(2), ename varchar(10), deptno number(2), constraint pk primary key(empno), constraint fk foreign key(deptno) references dept(deptno) on delete cascade); // with label

ALTER LEVEL

SQL> alter table emp add foreign key(deptno) references dept(deptno) on delete cascade;

SQL> alter table emp add constraint fk foreign key(deptno) references dept(deptno) on delete cascade;

Enabling/Disabling a Constraint:

If the constraints are present, then for each DML operation constraints are checked by executing certain codes internally. It may slow down the DML operation marginally. For massive DML operations, such as transferring data from one table to another because of the presence of constraint, the speed will be considered slower. To improve the speed in such cases, the following methods are adopted:

→Disable constraint
→Performing the DML operation DML operation
→Enable constraint

Disabling Constraint:-
Syntax :-

ALTER TABLE <tabname>   DISABLE CONSTRAINT <constraint_name> ;

Example :-

SQL>ALTER TABLE student1 DISABLE CONSTRAINT ck ;

SQL>ALTER TABLE mark1 DISABLE PRIMARY KEY CASCADE;

NOTE:-

If constraint is disabled with CASCADE then PK is disabled with FK.

Enabling Constraint :-

Syntax :-

ALTER TABLE <tabname> ENABLE CONSTRAINT <name>

Example :-

SQL>ALTER TABLE student1 ENABLE CONSTRAINT ck;

III. SET OPERATIONS IN SQL

SQL supports few Set operations to be performed on table data. These are used to get meaningful results from data, under different special conditions. The SET operators combine the results of two or more component queries into one result. Queries containing SET operators are called Compound Queries.

The number of columns and data types of the columns being selected must be identical in all the SELECT statements used in the query. The names of the columns need not be identical.

All SET operators have equal precedence. If a SQL statement contains multiple SET operators, the oracle server evaluates them from left (top) to right (bottom) if no parentheses explicitly specify another order.

Introduction

SQL set operators allows combine results from two or more SELECT statements. At first sight this looks similar to SQL joins although there is a big difference. SQL joins tends to combine columns i.e. with each additionally joined table it is possible to select more and more columns. SQL set operators on the other hand combine rows from different queries with strong preconditions .

- Retrieve the same number of columns and
- The data types of corresponding columns in each involved SELECT must be compatible (either the same or with possibility implicitly convert to the data types of the first SELECT statement).

Set operator types

According to SQL Standard there are following Set operator types:

- UNION ---returns all rows selected by either query. To return all rows from multiple tables and eliminates any duplicate rows.
- UNION ALL-- returns all rows from multiple tables including duplicates.
- INTERSECT – returns all rows common to multiple queries.
- MINUS—returns rows from the first query that are not present in second query.

Note: Whenever these operators used select statement must have

- Equal no. of columns.
- Similar data type columns.

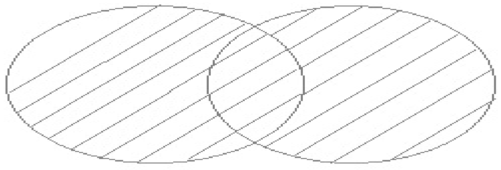
Syntax :-

SELECT statement 1
UNION / UNION ALL / INTERSECT / MINUS
SELECT statement 2 ;
Rules :-

1 No of columns returned by first query must be equal to no of columns returned by second query
2 Corresponding columns datatype type must be same.

1. UNION

- UNION operator combines data returned by two SELECT statement.
- Eliminates duplicates.
- Sorts result.
- This will combine the records of multiple tables having the same structure.



Example :-

```
1       SQL>SELECT job FROM emp WHERE deptno=10
                        UNION
                SELECT job FROM emp WHERE deptno=20 ;

2       SQL>SELECT job,sal FROM emp WHERE deptno=10
```

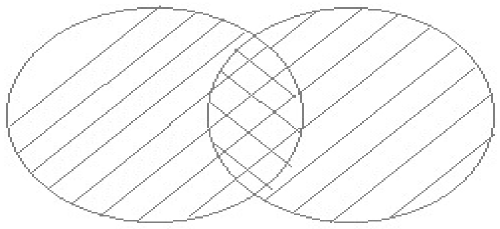
```
        UNION
      SELECT job,sal FROM emp WHERE deptno=20 ORDER BY sal ;
```

NOTE:- ORDER BY clause must be used with last query.

3 SQL> select * from student1 **union** select * from student2;

2. UNION ALL

This will combine the records of multiple tables having the same structure but including duplicates. IT is similar to UNION but it includes duplicates.



Example :-

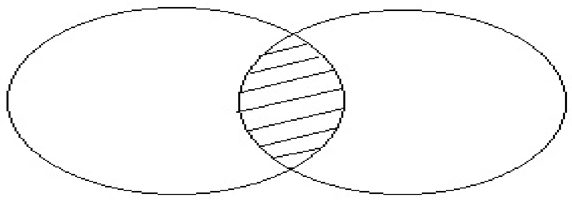
```
SQL>SELECT job FROM emp WHERE deptno=10
      UNION ALL
      SELECT job FROM emp WHERE deptno=20  ;
```

SQL> select * from student1 **union all** select * from student2;

3. INTERSECT

This will give the common records of multiple tables having the same structure.

INTERSECT operator returns common values from the result of two SELECT statements.



Example:-

Display common jobs belongs to 10th and 20th departments ?

EX 1: SQL>SELECT job FROM emp WHERE deptno=10

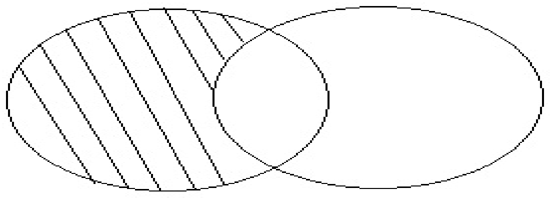
```
INTERSECT
 SELECT job FROM emp WHERE deptno=20;
```

EX2: SQL> select * from student1 intersect select * from student2;

4. MINUS

This will give the records of a table whose records are not in other tables having the same structure.

MINUS operator returns values present in the result of first SELECT statement and not present in the result of second SELECT statement.



Example:-

Display jobs in 10th dept and not in 20th dept ?

```
EX1:  SQL>SELECT job FROM emp WHERE deptno=10
            MINUS
             SELECT job FROM emp WHERE deptno=20;
```

E**x2: SQL> select * from student1 minus select * from student2;**

UNION vs JOIN :-

UNION	JOIN
Union combines data	Join relates data
Union is performed on similar structures on	Join can be performed also be performed dissimilar structures also

V. SQL JOINS

A SQL JOIN is an Operation , used to retrieve data from multiple tables. It is performed whenever two or more tables are joined in a SQL statement. so, SQL Join clause is used to combine records from two or more tables in a database. A JOIN is a means for combining fields from two tables by using values common to each. Several operators can be used to join tables, such as =, <>, <=, >=, !=, BETWEEN, LIKE, and NOT; these all to be used to join tables. However, the most common operator is the equal symbol.

SQL Join Types:

There are different types of joins available in SQL:

- **INNER JOIN**: Returns rows when there is a match in both tables.
- **OUTER JOIN** : Returns all rows even there is a match or no match in tables.
  - **LEFT JOIN/LEFT OUTER JOIN**: Returns all rows from the left table, even if there are no matches in the right table.
  - **RIGHT JOIN/RIGHT OUTER JOIN** : Returns all rows from the right table, even if there are no matches in the left table.
  - **FULL JOIN/FULL OUTER JOIN** : Returns rows when there is a match in one of the tables.
- **SELF JOIN**: It is used to join a table to itself as if the table were two tables, temporarily renaming at least one table in the SQL statement.
- **CARTESIAN JOIN or CROSS JOIN** : It returns the Cartesian product of the sets of records from the two or more joined tables.

Based on **Operators**, **The Join** can be classified as

- **Inner join or Equi Join**
- **Non-Equi Join**

- **NATURAL JOIN:** It is performed only when common column name is same. In this,no need to specify join condition explicitly , ORACLE automatically performs join operation on the column with same name.

1. <u>SQL INNER JOIN (simple join)</u>

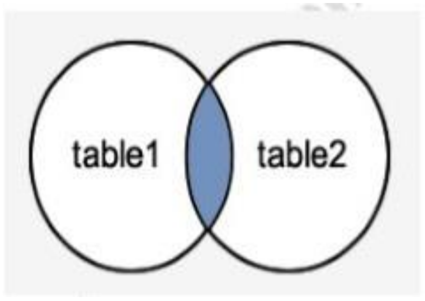
It is the most common type of SQL join. SQL INNER JOINS return all rows from multiple tables where the join condition is met.

<u>Syntax</u>

SELECT columns FROM table1 INNER JOIN table2 ON table1.column = Table2.column;

<u>Visual Illustration</u>

In this visual diagram, the SQL INNER JOIN returns the shaded area:



The SQL INNER JOIN would return the records where table1 and table2 intersect.

Let's look at some data to explain how the INNER JOINS work with example.

We have a table called SUPPLIERS with two fields (supplier_id and supplier_name).

It contains the following data:

supplier_id	supplier_name
10000	ibm
10001	hewlett packard
10002	microsoft
10003	nvidia

We have another table called ORDERS with three fields (order_id, supplier_id, and order_date).

It contains the following data:

order_id	supplier_id	order_date
500125	10000	2003/05/12
500126	10001	2003/05/13
500127	10004	2003/05/14

<u>Example of INNER JOIN:</u>

Q: List supplier id, name and order id of supplier.

SELECT s.supplier_id, s.supplier_name, od.order_date FROM suppliers s INNER JOIN orders od ON s.supplier_id = od.supplier_id;

This SQL INNER JOIN example would return all rows from the suppliers and orders tables where there is a matching supplier_id value in both the suppliers and orders tables.

Our result set would look like this:

supplier_id	name	order_date
10000	ibm	2003/05/12
10001	hewlett packard	2003/05/13

The rows for Microsoft and NVIDIA from the supplier table would be omitted, since the supplier_id's 10002 and 10003 do not exist in both tables.

The row for 500127 (order_id) from the orders table would be omitted, since the supplier_id 10004 does not exist in the suppliers table.

2. OUTER JOIN:

Inner / Equi join returns only matching records from both the tables but not unmatched record, An Outer join retrieves all row even when one of the column met join condition.

Types of outer join:

1. LEFT JOIN/LEFT OUTER JOIN
2. RIGHT JOIN/RIGHT OUTER JOIN
3. FULL JOIN/FULL OUTER JOIN

2.1.LEFT OUTER JOIN

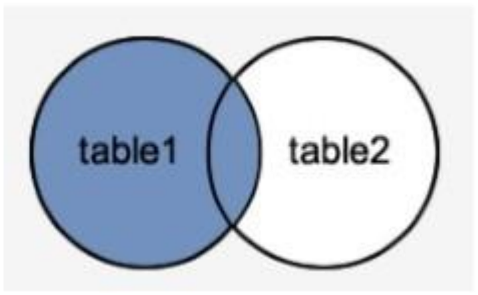
This type of join returns all rows from the LEFT-hand table specified in the ON condition and only those rows from the other table where the joined fields are equal (join condition is met).

Syntax

```
SELECT columns FROM  table1  LEFT [OUTER]  JOIN  table2
ON  table1.column = table2.column;
```

Visual Illustration

In this visual diagram, the SQL LEFT OUTER JOIN returns the shaded area:



The SQL LEFT OUTER JOIN would return the all records from table1 and only those records from table2 that intersect with table1.

Example

```
SELECT suppliers.supplier_id, suppliers.supplier_name, orders.order_date FROM
suppliers LEFT OUTER JOIN orders ON suppliers.supplier_id = orders.supplier_id;
```

This LEFT OUTER JOIN example would return all rows from the suppliers table and only those rows from the orders table where the joined fields are equal.

```
supplier_id          supplier_name             order_date
--------------     ----------------------      -----------------------
10000                       ibm                        2003/05/12
10001                       hewlett packard  2003/05/13
10002                       microsoft               <null>
10003                       nvidia                   <null>
```

The rows for Microsoft and NVIDIA would be included because a LEFT OUTER JOIN was used. However, you will notice that the order_date field for those records contains a <null> value.

2.2 SQL RIGHT OUTER JOIN

This type of join returns all rows from the RIGHT-hand table specified in the ON condition and only those rows from the other table where the joined fields are equal (join condition is met).

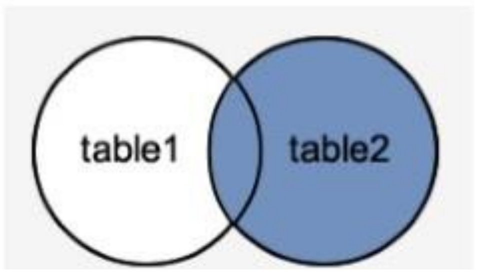
Syntax

SELECT columns FROM table1 RIGHT [OUTER] JOIN table2 ON table1.column = table2.column;

In some databases, the RIGHT OUTER JOIN keywords are replaced with RIGHT JOIN.

Visual Illustration

In this visual diagram, the SQL RIGHT OUTER JOIN returns the shaded area:



The SQL RIGHT OUTER JOIN would return the all records from table2 and only those records from table1 that intersect with table2.

Example

SELECT orders.order_id, orders.order_date, suppliers.supplier_name FROM suppliers RIGHT OUTER JOIN orders ON suppliers.supplier_id = orders.supplier_id;

This RIGHT OUTER JOIN example would return all rows from the orders table and only those rows from the suppliers table where the joined fields are equal.

If a supplier_id value in the orders table does not exist in the suppliers table, all fields in the suppliers table will display as <null> in the result set.

```
order_id          order_date          supplier_name
------------      ---------------     -----------------
500125            2013/05/12          ibm
500126            2013/05/13          hewlett packard
500127            2013/05/14          <null>
```

The row for 500127 (order_id) would be included because a RIGHT OUTER JOIN was used. However, you will notice that the supplier_name field for that record contains a <null> value.

2.3. SQL FULL OUTER JOIN

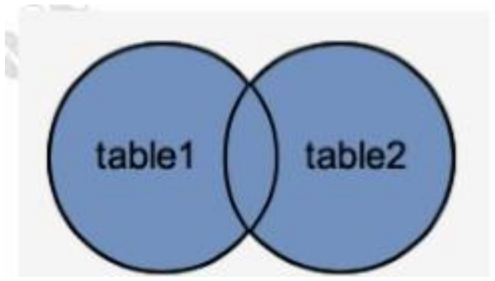
This type of join returns all rows from the LEFT-hand table and RIGHT-hand table with nulls in place where the join condition is not met.

<u>Syntax</u>

SELECT columns FROM table1 FULL [OUTER] JOIN table2 ON table1.column = table2.column; In some databases, the FULL OUTER JOIN keywords are replaced with FULL JOIN.

<u>Visual Illustration</u>

In this visual diagram, the SQL FULL OUTER JOIN returns the shaded area:



The SQL FULL OUTER JOIN would return the all records from both table1 and table2.

<u>Example</u>

Here is an example of a SQL FULL OUTER JOIN:

Query : Find supplier id, supplier name and order date of suppliers who have ordered.

SELECT suppliers.supplier_id, suppliers.supplier_name, orders.order_date FROM suppliers FULL OUTER JOIN orders ON suppliers.supplier_id = orders.supplier_id;

This FULL OUTER JOIN example would return all rows from the suppliers table and all rows from the orders table and whenever the join condition is not met, <nulls> would be extended to those fields in the result set.

If a supplier_id value in the suppliers table does not exist in the orders table, all fields in the orders table will display as <null> in the result set. If a supplier_id value in the orders table does not exist in the suppliers table, all fields in the suppliers table will display as <null> in the result set.

```
supplier_id          supplier_name        order_date
-----------------    --------------------  ---------------------
10000                ibm                   2013/05/12
10001                hewlett packard       2013/05/13
10002                microsoft             <null>
10003                nvidia                <null>
<null>               <null>                2013/05/14
```

The rows for Microsoft and NVIDIA would be included because a FULL OUTER JOIN was used. However, you will notice that the order_date field for those records contains a <null> value.

The row for supplier_id 10004 would be also included because a FULL OUTER JOIN was used. However, you will notice that the supplier_id and supplier_name field for those records contain a <null> value.

<u>Equi join :</u>

When the Join Condition is based on EQUALITY (=) operator, the join is said to be an Equi join. It is also called as **Inner Join.**

<u>Syntax</u>

Select col1,col2,…From <table 1>,<table 2>Where <join condition with '=' > .

Ex.Query : Find supplier id, supplier name and order date of suppliers who have ordered .

select s.supplierid, s.uppliername ,o.order_date from suppliers s, orders o where s.supplierid =o.supplierid.

supplier_id	name	order_date
10000	**ibm**	**2003/05/12**
10001	**hewlett packard**	**2003/05/13**

Non Equi Join :-

When the join condition based on other than equality operator , the join is said to be a Non-Equi join.

Syntax:-

Select col1,col2,…….

From <table 1>,<table 2>

Where <join condition > [AND <join cond> AND <cond> ----]

→In NON- EQUI JOIN, JOIN COND is not based on = operator. It is based on other than = operator, usually BETWEEN or > or < operators.

Query : Find supplier id,supplier name and order date in between 50025 and 500127.

```
sql> select s.supplier_id,s.supplier_name,o.order_date  from suppliers s , orders o  where
o.order_id between 500125 and 500127;

SUPPLIER_ID SUPPLIER_N ORDER_DAT
------------- ------------ -----------
10000 ibm       12-may-03
10000 ibm       13-may-03
10000 ibm       14-may-03
10001 hewlett   12-may-03
10001 hewlett   13-may-03
10001 hewlett   14-may-03
10002 microsoft 12-may-03
10002 microsoft 13-may-03
10002 microsoft 14-may-03
10003 nvidia    12-may-03
10003 nvidia    13-may-03
10003 nvidia    14-may-03
```

Query : Find supplier id,supplier name and order date above 500126.

```
sql> select s.supplier_id,s.supplier_name,o.order_date  from suppliers s , orders o  where
o.order_id >500126;

SUPPLIER_ID  SUPPLIER_NO ORDER_DAT
------------- ------------ -----------
10000    ibm           14-may-03
10001    hewlett       14-may-03
10002    microsoft     14-may-03
10003    nvidia        14-may-03
```

<u>Self Join :-</u>

Joining a table to itself is called Self Join.

- Self Join is performed when tables having self refrential integrity.
- To perform Self Join same table must be listed twice with different alias.
- Self Join is Equi Join within the table.

It is used to join a table to itself as if the table were two tables, temporarily renaming at least one table in the SQL statement.

<u>Syntax :</u>

(Here T1 and T2 refers same table)

```
SELECT <collist>  From Table1 T1, Table1 T2
        Where T1.Column1=T2.Column2;
```

<u>Example</u>:

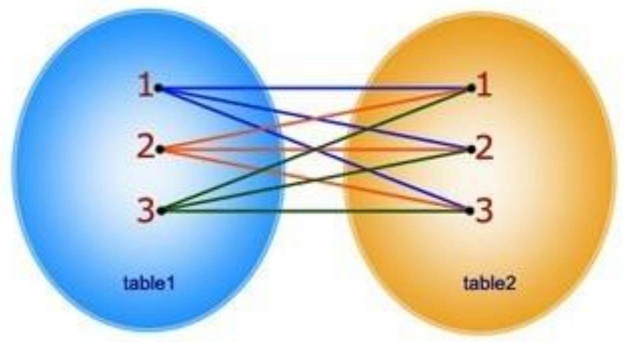
select s1.supplier_id ,s1.supplier_name ,s2.supplier_id from suppliers s1, suppliers s2 where s1.supplier_id=s2.supplier_id ;

```
     supplier_id          supplier_name    supplier_id
-----------------         ------------------   ---------------
10000                 ibm                    10000
10001                 hewlett packard        10001
10002                 microsoft              10002
10003                 nvidia                 10003
```

CROSS JOIN:

It returns the Cartesian product of the sets of records from the two or more joined tables. In Cartesian product, each element of one set is combined with every element of another set to form the resultant elements of Cartesian product.

Sytax: SELECT * FROM <tablename1> CROSS JOIN <tablename2>



In CROSS JOIN, each row from 1st table joins with all the rows of another table.
If 1st table contain x rows and y rows in 2nd one the result set will be x * y rows.

- CROSS JOIN returns cross product of two tables.
- Each record of one table is joined to each and every record of another table.
- If table1 contains 10 records and table2 contains 5 records then CROSS JOIN between table1 and table2 returns 50 records.
- ORACLE performs CROSS JOIN when we submit query without JOIN COND.

Example: sql> SELECT * FROM suppliers CROSS JOIN orders;

```
supplier_id supplier_n  order_id supplier_id order_dat
------------- ------------ ------------ ------------- -----------
      10000 ibm         500125      10000 12-may-03
      10000 ibm         500126      10001 13-may-03
      10000 ibm         500127      10003 14-may-03
      10001 hewlett     500125      10000 12-may-03
      10001 hewlett     500126      10001 13-may-03
      10001 hewlett     500127      10003 14-may-03
      10002 microsoft   500125      10000 12-may-03
      10002 microsoft   500126      10001 13-may-03
      10002 microsoft   500127      10003 14-may-03
      10003 nvidia      500125      10000 12-may-03
      10003 nvidia      500126      10001 13-may-03
```

NATURAL JOIN:

- NATURAL JOIN is possible in ANSI SQL/92 standard.
- NATURAL JOIN is similar to EQUI JOIN.
- NATURAL JOIN is performed only when common column name is same.
- In NATURAL JOIN no need to specify join condition explicitly , ORACLE automatically performs join operation on the column with same name.

Syntax: SELECT <column list> FROM table1 NATURAL JOIN table2;

Example: (Sailors table)

SELECT sid,sname,sid FROM sailors NATURAL JOIN reserves ; //both tables have same column name.

```
       SID SNAME             SID
---------- ---------- ----------
        22 DUSTIN             22
        22 DUSTIN             22
        22 DUSTIN             22
        22 DUSTIN             22
        31 LUBBER             31
        31 LUBBER             31
        31 LUBBER             31
        64 HORTIO             64
        64 HORTIO             64
        74 HORTIO             74
```

VI. VIEWS

A view in SQL is a logical subset of data from one or more tables. View is used to restrict data access.Data abstraction is usually required after a table is created and populated with data. Data held by some tables might require restricted access to prevent all users from accessing all columns of a table, for data security reasons. Such a security issue can be solved by creating several tables with appropriate columns and assigning specific users to each such table, as required. This answers data security requirements very well but gives rise to a great deal of redundant data being resident in tables, in the database.To reduce redundant data to the minimum possible, Oracle provides Virtual tables which are Views.

View Definition :-

→A View is a virtual table based on the result returned by a SELECT query.

→The most basic purpose of a view is restricting access to specific column/rows from a table thus allowing different users to see only certain rows or columns of a table.

Composition Of View:-

→A view is composed of rows and columns, very similar to table. The fields in a view are fields from one or more database tables in the database.

→SQL functions, WHERE clauses and JOIN statements can be applied to a view in the same manner as they are applied to a table.

View storage:-

→Oracle does not store the view data. It recreates the data, using the view's SELECT statement, every time a user queries a view.

→A view is stored only as a definition in Oracle's system catalog.

→When a reference is made to a view, its definition is scanned, the base table is opened and the view is created on top of the base table.This, therefore, means that a view never holds data, until a specific call to the view is made. This reduces redundant data on the HDD to a very large extent.

Advantages Of View:-

Security:- Each user can be given permission to access only a set of views that contain specific data.

Query simplicity:- A view can drawn from several different tables and present it as a single table turning multiple table queries into single table queries against the view.

Data Integrity:- If data is accessed and entered through a view, the DBMS can automatically check the data to ensure that it meets specified integrity constraints.

Disadvantage of View:-

Performance:- Views only create the appearance of the table but the RDBMS must still translate queries against the views into the queries against the underlined source tables. If the view is defined on a complex multiple table query then even a simple query against the view becomes a complicated join and takes a long time to execute.
Types of Views :-

- Simple Views
- Complex Views

Simple Views :-
a View based on single table is called simple view.

Syntax:-
```
CREATE VIEW <View Name>
AS
SELECT<ColumnName1>,<ColumnName2>..
FROM  <TableName>
[WHERE <COND>]
[WITH CHECK OPTION]
[WITH READ ONLY]
```

Example :-

```
SQL>CREATE VIEW emp_v
     AS
     SELECT empno,ename,sal FROM emp ;
```

→Views can also be used for manipulating the data that is available in the base tables[i.e. the user can perform the Insert, Update and Delete operations through view.

→Views on which data manipulation can be done are called Updateable Views.

→If an Insert, Update or Delete SQL statement is fired on a view, modifications to data in the view are passed to the underlying base table.

→For a view to be updatable,it should meet the following criteria:

→Views defined from Single table.

→If the user wants to INSERT records with the help of a view, then the PRIMARY KEY column(s) and all the NOT NULL columns must be included in the view.

Inserting record through view :-

SQL>INSERT INTO emp_v VALUES(1,'A',5000,200) ;

Updating record throught view :-

Updating a View:

A view can be updated under certain conditions:

- 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 a view.

EX: SQL>UPDATE emp_v SET sal=2000 WHERE empno=1;

Deleting record throught view :-

SQL>DELETE FROM emp_v WHERE empno=1;

With Check Option :-

If VIEW created with WITH CHECK OPTION then any DML operation through that view violates where condition then that DML operation returns error.

Example :-

```
SQL>CREATE VIEW V2
        AS
        SELECT empno,ename,sal,deptno FROM emp
        WHERE deptno=10
        WITH CHECK OPTION ;
```

Then insert the record into emp table through view V2

SQL>INSERT INTO V2 VALUES(2323,'RAJU',4000,20) ;

The above INSERT returns error because DML operation violating WHERE clause.

Complex Views :-

A view is said to complex view

→If it based on more than one table

→Query contains

AGGREGATE functions
DISTINCT clause
GROUP BY clause
HAVING clause
Sub-queries
Constants
Strings or Values Expressions
UNION,INTERSECT,MINUS operators.

Example 1 :-

```
SQL>CREATE VIEW V3
AS
SELECT E.empno,E.ename,E.sal,D.dname,D.loc
FROM emp E JOIN dept D
USING(deptno) ;
```

NON- UPDATABLE VIEWS:

we cannot perform insert or update or delete operations on base table through complex views. Complex views are not updatable views.

Example 2 :-

```
SQL>CREATE VIEW V2
AS
SELECT deptno,SUM(sal) AS sumsal
FROM EMP
GROUP BY deptno;
```

Destroying a View:-

The DROP VIEW command is used to destroy a view from the database.

Syntax:-

```
DROP VIEW<viewName>
```

Example :-

```
SQL>DROP VIEW emp_v;
```

DIFFERENCES BETWEEN SIMPLE AND COMPLEX VIEWS:

SIMPLE	COMPLEX
Created from one table	Created from one or more tables
Does not contain functions	Conations functions
Does not contain groups of data	Contain groups of data

MATERIALIZED VIEW: @ DATAWAREHOUSE SYSTEMS

A materialized view in Oracle is a database object that contains the results of a query. They are local copies of data located remotely, or are used to create summary tables based on aggregations of a table's data. Materialized views, which store data based on remote tables are also, know as snapshots.

A materialized view can query tables, views, and other materialized views. Collectively these are called master tables (a replication term) or detail tables (a data warehouse term).

For replication purposes, materialized views allow you to maintain copies of remote data on your local node. These copies are read-only. If you want to update the local copies, you have to use the Advanced Replication feature. You can select data from a materialized view as you would from a table or view.

For data warehousing purposes, the materialized views commonly created are aggregate views, single-table aggregate views, and join views.

In replication environments, the materialized views commonly created are primary key, rowid, and subquery materialized views.

SYNTAX:

```
CREATE MATERIALIZED VIEW view-name

BUILD [IMMEDIATE | DEFERRED]

REFRESH [FAST | COMPLETE | FORCE ]

ON [COMMIT | DEMAND ]

[[ENABLE | DISABLE] QUERY REWRITE]

[ON PREBUILT TABLE]

AS

SELECT  COLUMN_LIST FROM TABLE_NAME;
```

The BUILD clause options are shown below.

- IMMEDIATE : The materialized view is populated immediately.
- DEFERRED : The materialized view is populated on the first requested refresh.

The following refresh types are available.

- FAST : A fast refresh is attempted. If materialized view logs are not present against the source tables in advance, the creation fails.
- COMPLETE : The table segment supporting the materialized view is truncated and repopulated completely using the associated query.
- FORCE : A fast refresh is attempted. If one is not possible a complete refresh is performed.

A refresh can be triggered in one of two ways.

- ON COMMIT : The refresh is triggered by a committed data change in one of the dependent tables.
- ON DEMAND : The refresh is initiated by a manual request or a scheduled task.

The QUERY REWRITE clause tells the optimizer if the materialized view should be consider for query rewrite operations. An example of the query rewrite functionality is shown below.
The ON PREBUILT TABLE clause tells the database to use an existing table segment, which must have the same name as the materialized view and support the same column structure as the query.

Example:
The following statement creates the rowid materialized view on table emp located on a remote database:

```
SQL>    CREATE MATERIALIZED VIEW mv_emp_rowid
        REFRESH WITH ROWID
        AS SELECT * FROM emp@remote_db;

Materialized view log created.
```

VII. ORDERING

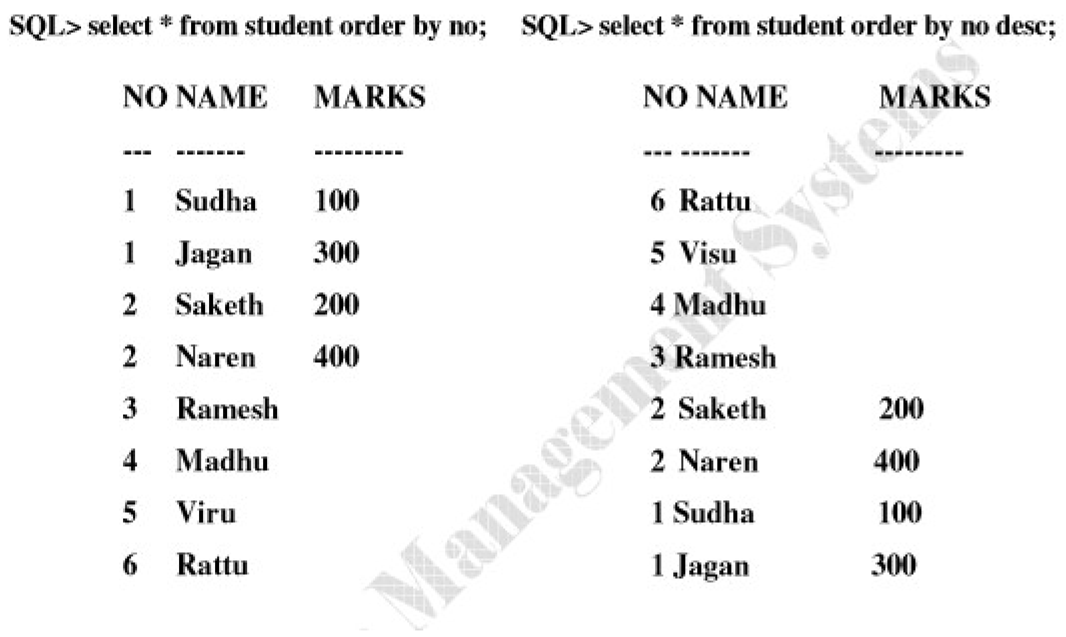
USING “ ORDER BY” clause:

This will be used to ordering the columns data (ascending or descending).
Syntax1: (simple form)
select * from <table_name> order by <col> desc;

Note: By default oracle will use ascending order.

If you want output in descending order you have to use desc keyword after the column.
Ex:

```
SQL> select * from student order by no;       SQL> select * from student order by no desc;
```



```
SQL> select * from student order by no;     SQL> select * from student order by no desc;

          NO NAME     MARKS                           NO NAME        MARKS
          ---  -------    ---------                            --- -------        ---------
          1    Sudha     100                                  6  Rattu
          1    Jagan     300                                  5  Visu
          2    Saketh    200                                  4 Madhu
          2    Naren     400                                  3 Ramesh
          3    Ramesh                                         2  Saketh        200
          4    Madhu                                          2  Naren         400
          5    Viru                                           1 Sudha          100
          6    Rattu                                          1 Jagan          300
```

The order of rows returned in a query result is undefined. The ORDER BY clause can be used to sort the rows. If you use the ORDER BY clause, it must be the last clause of the SQL statement. You can specify an expression, or an alias, or column position in ORDER BY clause.

Syntax2 : (complex form)

SELECT *expr* FROM *table*
[WHERE *condition(s)*]
[ORDER BY {*column*, *expr*} [ASC|DESC]];

In the syntax,

ORDER BY :specifies the order in which the retrieved rows are displayed.
orders the rows in ascending order (default order)
orders the rows in descending order

Ordering of Data :-

- Numeric values are displayed with the lowest values firs   for example   1–999.
- Date values are displayed with the earliest value first for example 01-JAN-92 before 01-JAN-95.
- Character values are displayed in alphabetical order—for example, A first and *Z* last.
- Null values are displayed last for ascending sequences and first for descending sequences.

Examples :-

Arrange employee records in ascending order of their sal ?

SQL>SELECT * FROM emp ORDER BY sal ;

Arrange employee records in descending order of their sal ?

SQL>SELECT * FROM emp ORDER BY sal DESC ;

Display employee records working for 10th dept and arrange the result in ascending order of their sal ?

SQL>SELECT * FROM emp WHERE deptno=10 ORDER BY sal ;

Arrange employee records in ascending of their deptno and with in dept arrange records in descending order of their sal ?

SQL>SELECT * FROM emp ORDER BY deptno,sal DESC ;

In ORDER BY clause we can use column name or column position , for example

SQL>SELECT * FROM emp ORDER BY 5 DESC ;

In the above example records are sorted based on the fifth column in emp table.

Arrange employee records in descending order of their comm. If comm. Is null then arrange those records last ?

SQL>SELECT * FROM emp ORDER BY comm DESC NULLS LAST ;

VIII.GROUP BY AND HAVING CLAUSE

GROUP BY clause

Using group by, we can create groups of related information. Columns used in select must be used with group by, otherwise it was not a group by expression.

```
SELECT      [DISTINCT]  select-list
FROM  from-list
WHERE    qualification
GROUP BY  grouping-list
HAVING      group-qualification
```

- The select list in the SELECT clause contain
  1. A list of column names
  2. A list of terms having the form aggop(aggregate operators)

  Every column that appear in (1) must also appear in **grouping-list**

- The expression appearing in the **group-qualification** in the HAVING clause must have a single value per group.

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

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

DEPTNO    SUM(SAL)

---------- ----------

     10            8750

      20            10875

      30             9400
```

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

Sql> Find the age of the youngest sailor for each rating level.

SQL> **Select s.rating, MIN(s.age) from sailors s GROUP BY s.rating;**

Find the age of the youngest sailor who is eligible to vote for each rating level with at least two such sailors ?

```
SQL> select s.rating, MIN(s.age) as minage from sailors s where s.age>=18
                                    GROUP BY s.rating
                                    HAVING COUNT(*) > 1;
```

For each red boat find the number of reservations for this boat?

```
SQL> Select b.bid, COUNT(*) AS reservationcount from boats b,
          reserves r where r.bid=b.bid and b.color='red'
               GROUP BY b.bid;
```

Find the average age of sailors for each rating level that has at least two sailors ?

```
SQL> Select s.rating, AVG(s.age) AS avgage from sailors s
               GROUP BY s.rating
               HAVING COUNT(*) > 1;
```

IX. AGGREGATION

It is a group operation, which will be works on all records of a table. To do this, Group functions required to process group of rows and Returns one value from that group.

→These functions are also called **AGGREGATE** functions or **GROUP** functions

- **Aggregate functions - max(),min(),sum(),avg(),count(),count(*).**

  Group functions will be applied on all the rows but produces single output.

  a) SUM

  This will give the sum of the values of the specified column.

  Syntax: sum (column)

  Ex: SQL> select sum(sal) from emp;

  b) AVG

  This will give the average of the values of the specified column.

  Syntax: avg (column)

  Ex: SQL> select avg(sal) from emp;

  c) MAX

  This will give the maximum of the values of the specified column.

  Syntax: max (column)

  Ex: SQL> select max(sal) from emp;

  d) MIN

  This will give the minimum of the values of the specified column.

  Syntax: min (column)

  Ex: SQL> select min(sal) from emp;

  e) COUNT

  This will give the count of the values of the specified column.

  Syntax: count (column)

  Ex: SQL> select count(sal),count(*) from emp;

<u>X.</u> <u>SUB QUERIES</u>

What is subquery in SQL?

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

- A subquery may occur in :

  - A SELECT clause
  - A FROM clause

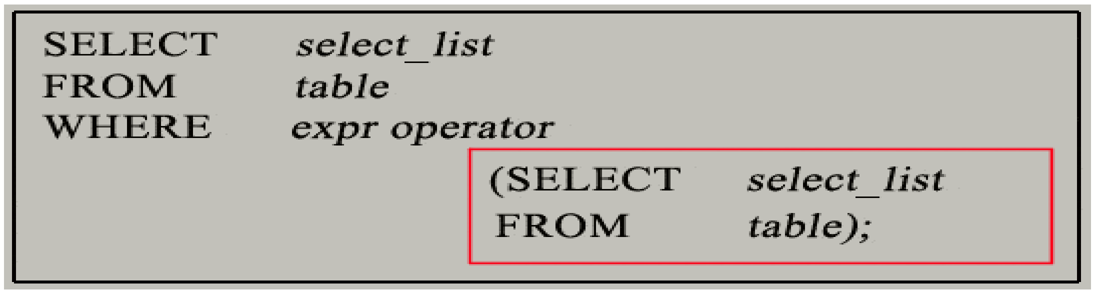
- A WHERE clause

- 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 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 Subqueries Example :

In this section, you will learn the requirements of using subqueries. We have the following two tables **'student'** and **'marks'** with common field 'SID'.

SQL> select *from student1;

```
SID    NAME
----------
```

```
v001 abhi
v002 abhay
v003 arjun
v004  anand
```

SQL> select *from marks;

```
SID       TOTALMARKS
- - - - - - - - - - - - - -
v001            95
v002            80
v003            74
v004            81
```

Now we want to write a query **to identify all students who get better marks than that of the student who's StudentID is 'V002',** but we do not know the marks of **'V002'**.

- To solve this problem, we require **two** queries.

One query returns the marks (stored in Totalmarks field) of 'V002' and a second query identifies the students who get better marks than the result of the first query.

SQL> select *from marks where sid='v002';

Query Result:

```
SID       TOTALMARKS
---------- ----------
v002            80
```

The result of query is **80**.

- Using the result of this query, here we have written another query to identify the students who get better marks than 80. Here is the query :

Second query :

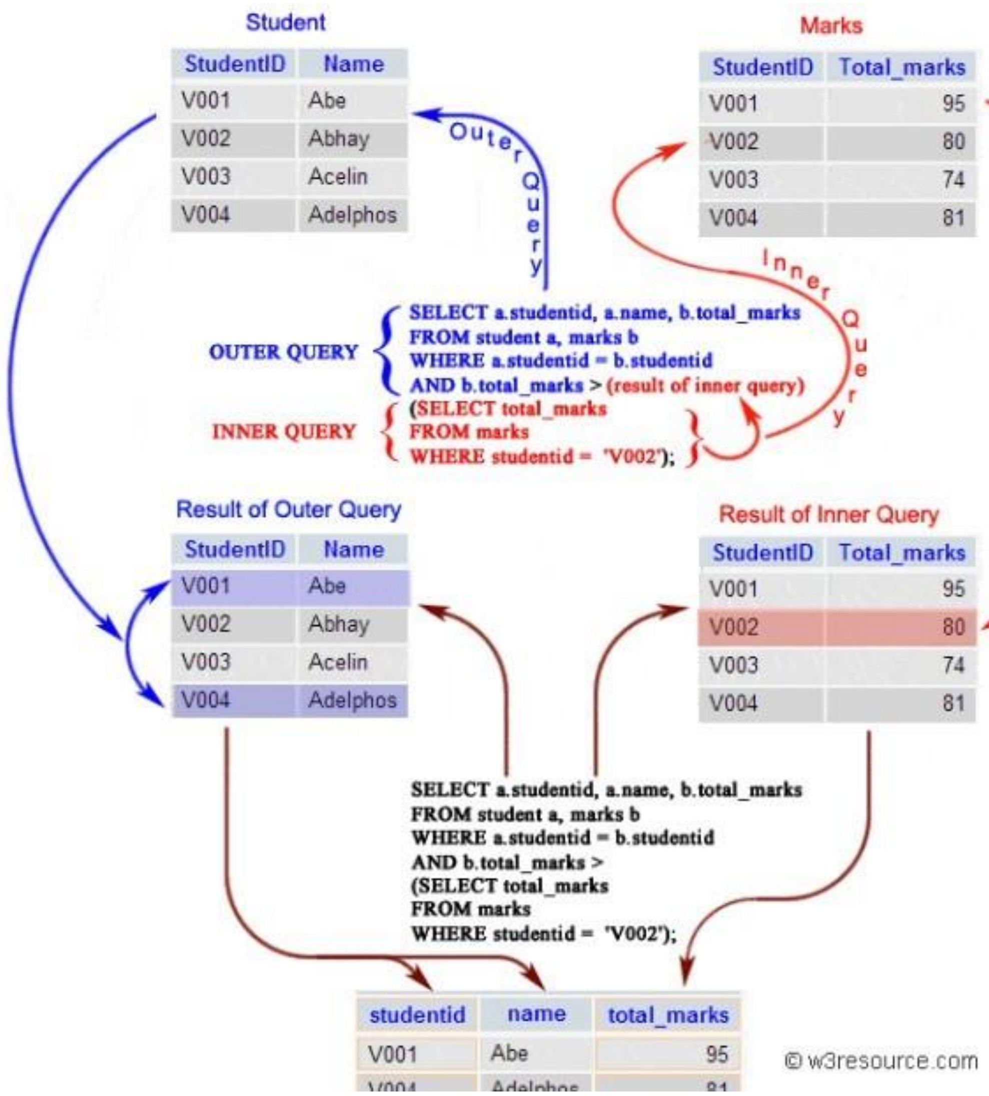
SQL> select s.sid,s.name,m.totalmarks from student1 s, marks m where s.sid=m.sid and m.totalmarks>80;

```
SID  NAME      TOTALMARKS
---- ---------- ----------
v001 abhi              95
v004 anand             81
```

Above two queries identified students who get better number than the student who's StudentID is 'V002' (Abhi).

You can combine the above two queries by placing one query inside the other. The subquery (also called the 'inner query') is the query inside the parentheses. See the following code and query result :

```
SQL> select s.sid,s.name,m.totalmarks from student1 s,marks m where s.sid=m.sid and
m.totalmarks >(select totalmarks from marks where sid='v002');
SID  NAME       TOTALMARKS
---- ---------- ----------
v001 abhi              95
v004 anand              81
```



Student

StudentID	Name
V001	Abe
V002	Abhay
V003	Acelin
V004	Adelphos

Marks

StudentID	Total_marks
V001	95
V002	80
V003	74
V004	81

```
SELECT a.studentid, a.name, b.total_marks
FROM student a, marks b
WHERE a.studentid = b.studentid
AND b.total_marks > (result of inner query)
(SELECT total_marks
FROM marks
WHERE studentid = 'V002');
```

Result of Outer Query

StudentID	Name
V001	Abe
V002	Abhay
V003	Acelin
V004	Adelphos

Result of Inner Query

StudentID	Total_marks
V001	95
V002	80
V003	74
V004	81

```
SELECT a.studentid, a.name, b.total_marks
FROM student a, marks b
WHERE a.studentid = b.studentid
AND b.total_marks >
(SELECT total_marks
FROM marks
WHERE studentid = 'V002');
```

studentid	name	total_marks
V001	Abe	95
V004	Adelphos	81

Subqueries: Guidelines

There are some guidelines to consider when using subqueries :

-A subquery must be enclosed in parentheses.

-A subquery must be placed on the right side of the comparison operator.

-Subqueries cannot manipulate their results internally, therefore ORDER BY clause cannot be added in to a subquery.You can use a ORDER BY clause in the main SELECT statement (outer query) which will be last clause.

-Use single-row operators with single-row subqueries.

-If a subquery (inner query) returns a null value to the outer query, the outer query will not return any rows when using certain comparison operators in a WHERE clause.

Type of Subqueries

- **Single row subquery** : Returns zero or one row.
- **Multiple row subquery** : Returns one or more rows.
- **Multiple column subquery** : Returns one or more columns.
- **Correlated subqueries** : Reference one or more columns in the outer SQL statement.

The subquery is known as a correlated subquery because the subquery is related to the outer SQL statement.

- **Nested subqueries** : Subqueries are placed within another subqueries.

1)SINGLE ROW SUBQUERIES:- Returns zero or one row.

If inner query returns only one row then it is called single row subquery.

Syntax :-

```
SELECT <collist> FROM <tabname>
            WHERE  colname  OPERATOR  (SELECT statement)
```

Operator can be < > <= >= = <>

Examples:- (on Emp Table)

Q: Display employee records whose job equals to job of SMITH?

```
SQL>SELECT * FROM emp
         WHERE job = (SELECT job FROM emp WHERE  ename='SMITH') ;
```

Q: Display employee name earning maximum salary ?

```
SQL>SELECT ename FROM emp
   WHERE sal = (SELECT MAX(sal) FROM emp) ;
```

Example2: (on SAILORS _BOAT_RESERVATION DATABASE)

```
SQL> SELECT * FROM SAILORS;
        SID   SNAME         RATING       AGE
        ---------- ---------- ---------- ----------
        22    DUSTIN              7         45
        29    BRUTUS              1         33
        31    LUBBER              8         55.5
        32    CANDY               8         25.5
        58    RUSTY              10         35
        64    HORATIO             7         35
        71   ZOBRA               10         16
        74   HORATIO              9         35
        85   ART                  3         25.5
        95   BOB                  3         63.5
        SQL> SELECT * FROM BOATS;
             BID BNAME      COLOR
        ---------- ---------- ----------
             101   INTERLAKE    BLUE
             102   INTERLAKE  RED
             103   CLIPPER        GREEN
             104   MARINE         RED
        SQL> SELECT * FROM RESERVES;
             SID       BID     DAY
        ---------- ---------- ---------
              22       101     10-OCT-17
              22       102     10-OCT-17
              22       103     10-OCT-17
              22       104     10-JUL-17
              31       102     11-OCT-17
              31       103     11-JUN-17
              31       104     11-DEC-17
              64       101     09-MAY-17
              64       102     09-AUG-17
              74       103     09-AUG-17
```

Q: Find the sailor's ID whose name is equal to 'DUSTIN'

```
SQL> SELECT SID FROM  SAILORS  WHERE SID = (SELECT SID FROM
SAILORS WHERE SNAME='DUSTIN');
     SID
----------
      22
```

Q:Find sailors records whose name equals to ' DUSTIN'?

```
SQL> SELECT *FROM  SAILORS  WHERE SID = (SELECT SID FROM
SAILORS WHERE SNAME='DUSTIN');
```

```
          SID    SNAME         RATING       AGE
          ----------   ----------      ----------        ----------
           22      DUSTIN            7                 45
```

Q:Find the rating of a sailor whose name is 'DUSTIN'.

SQL> SELECT RATING FROM SAILORS WHERE SID = (SELECT SID FROM SAILORS WHERE

SNAME='DUSTIN');

```
   RATING
------------
       7
```

Q: Find the sailors records whose sid is geater than 'dustin'?

SQL> SELECT *FROM SAILORS WHERE SID > (SELECT SID FROM

SAILORS WHERE SNAME='DUSTIN');

```
       SID SNAME         RATING       AGE
---------- ---------- ---------- ----------
        29 BRUTUS             1        33
        31 LUBBER             8       55.5
        32 CANDY              8       25.5
        58 RUSTY             10        35
        64 HORATIO            7        35
        71 ZOBRA             10        16
        74 HORATIO            9        35
        85 ART                3       25.5
        95 BOB                3       63.5
```

Q:Find the sailors records ,whose sailors' having maximum rating .

SQL> SELECT *FROM SAILORS WHERE RATING = (SELECT

MAX(RATING) FROM SAILORS);

```
       SID  SNAME         RATING       AGE
---------- ---------- ---------- ----------
        58    RUSTY           10           35
        71    ZOBRA           10           16
```

Q:Find the records of sailors whose rating is same as 'DUSTIN'

SQL> SELECT *FROM SAILORS WHERE RATING = (SELECT RATING

FROM SAILORS WHERE SNAME='DUSTIN');

```
       SID     SNAME         RATING       AGE
---------- ---------- ---------- ----------
        22     DUSTIN              7           45
```

```
      64      HORATIO            7           35
```

Q:Find the records of sailors whose rating is higher than 'DUSTIN'

```
SQL> SELECT *FROM  SAILORS  WHERE RATING > (SELECT
MAX(RATING) FROM SAILORS WHERE SNAME='DUSTIN');

       SID SNAME          RATING        AGE
---------- ---------- ---------- ----------
        31 LUBBER              8       55.5
        32 CANDY               8       25.5
        58 RUSTY              10         35
        71 ZOBRA              10         16
        74 HORATIO             9         35
```

MULTI ROW SUBQUERIES :

if inner query returns more than one row then it is called multi row subquery.

Syntax :-

```
SQL>SELECT <collist> FROM <tabname>
   WHERE colname OPERATOR  (SELECT statement) ;
```

Here, OPERATOR must be IN ,   NOT IN,   ANY, ALL

IN operator :-

To test for values in a specified list of values, use IN operator. The IN operator can be used with any data type. If characters or dates are used in the list, they must be enclosed in single quotation marks ('').

Syntax:-

```
IN (V1,V2,V3 ----------);
```

Note :-

IN (...) is actually translated by Oracle server to a set of 'OR' conditions: a =value1 OR a = value2 OR a = value3. So using IN (...) has no performance benefits, and it is used for logical simplicity.

Example :-

Q:Display employee records working as CLERK OR MANAGER ?

```
SQL>SELECT * FROM emp  WHERE  job  IN  ('CLERK','MANAGER') ;
```

Q:Find the name of sailors who have reserved boat 103

```
SQL> SELECT S.SNAME FROM SAILORS S WHERE S.SID IN (SELECT R.SID
FROM RESERVES R WHERE R.BID=103);
```

```
        SNAME
        ----------
        DUSTIN
        LUBBER
        HORATIO
```

Q:Find the name of sailors who have reserved a red boat

```
SQL> SELECT S.SNAME FROM SAILORS S WHERE S.SID IN (SELECT R.SID
FROM RESERVES R WHERE R.BID IN (SELECT B.BID FROM BOATS B WHERE
B.COLOR='RED'));
```

```
        SNAME
        ----------
        DUSTIN
        LUBBER
        HORATIO
```

Q:Find the names of sailors who have not reserved a red boat.

```
 SELECT S.SNAME FROM SAILORS S WHERE S.SID NOT IN (SELECT R.SID FROM
RESERVES R WHERE R.BID IN (SELECT B.BID FROM BOATS B WHERE B.COLOR
= 'RED'));
```

```
        SNAME
        ----------
        BRUTUS
        CANDY
        RUSTY
        ZOBRA
        HORATIO
        ART
        BOB
```

Using EXISTS operator :-

→EXISTS operator returns TRUE or FALSE.

→If inner query returns at least one record then EXISTS returns TRUE otherwise returns FALSE.

→ORACLE recommends EXISTS and NOT EXISTS operators instead of IN and NOT IN.

Q: Find the name of sailors who have reserved boat 103

```
SQL> SELECT  S.SNAME FROM SAILORS S WHERE EXISTS (SELECT * FROM
RESERVES R WHERE R.BID=103 AND R.SID = S.SID)  ;
```

```
SNAME
----------
DUSTIN
LUBBER
HORATIO
```

Q:Find the name of sailors who have not reserved boat 103

SQL> SELECT S.SNAME FROM SAILORS S WHERE NOT EXISTS (SELECT * FROM RESERVES R WHERE R.BID=103 AND R.SID = S.SID) ;

```
SNAME
----------
BRUTUS
CANDY
RUSTY
HORATIO
ZOBRA
ART
BOB
```

ANY operator:-

Compares a value to each value in a list or returned by a query. Must be preceded by =, !=, >, <, <=, >=. Evaluates to FALSE if the query returns no rows.

Select employees whose salary is greater than any salesman's salary ?

```
SQL>SELECT ename  FROM emp
        WHERE SAL > ANY ( SELECT sal FROM emp  WHERE  job = 'SALESMAN');
```

Q:Find sailors whose rating is better than some sailor called Horatio?

SQL> SELECT S.SID FROM SAILORS S WHERE S.RATING > ANY (SELECT S2.RATING FROM SAILORS S2 WHERE S2.SNAME='HORATIO') ;

```
    SID
----------
     58
     71
     74
     31
     32
```

ALL operator :-

Compares a value to every value in a list or returned by a query. Must be preceded by =, !=, >, <, <=, >=.   evaluates to TRUE if the query returns no rows.

Example:-

Select employees whose salary is greater than every salesman's salary ?

```
SQL>SELECT ename  FROM emp
            WHERE SAL > ALL ( SELECT sal FROM emp  WHERE  job = 'SALESMAN');
```

Q:Find sailors whose rating is better than every sailor called Horation?

```
SQL> SELECT S.SID FROM SAILORS S WHERE S.RATING > ALL ( SELECT
S2.RATING FROM SAILORS S2 WHERE S2.SNAME='HORATIO') ;

       SID
----------
        58
        71
```

Multi Column Subqueries:-

If inner query returns more than one column value then it is called MULTI COLUMN subquery.

Example :-

Display employee names earning maximum salaries in their dept ?

```
SQL>SELECT ename FROM emp
        WHERE (deptno,sal) IN (SELECT deptno,MAX(sal)
                                FROM emp
                                GROUP BY deptno) ;
SQL> SELECT SNAME FROM SAILORS WHERE (RATING,AGE) IN (SELECT
RATING,MAX(AGE) FROM SAILORS GROUP BY RATING);

        SNAME
        ----------
        DUSTIN
        BRUTUS
        LUBBER
        RUSTY
        HORATIO
        BOB
SQL> SELECT SID,SNAME FROM SAILORS WHERE (RATING,AGE) IN (SELECT
RATING,MAX(AGE) FROM SAILORS GROUP BY RATING);

       SID SNAME
---------- ----------
        22 DUSTIN
        29 BRUTUS
        31 LUBBER
        58 RUSTY
        74 HORATIO
        95 BOB
```

Nested Queries:-

→A subquery embedded in another subquery is called NESTED QUERY.

→Queries can be nested upto 255 level.

Example :-
Display employee name earning second maximum salary ?

```
SQL>SELECT ename FROM emp
    WHERE sal = (SELECT MAX(sal) FROM EMP
              WHERE sal < (SELECT MAX(sal) FROM emp)) ;
```

Q:Find the names of sailors who have not reserved a red boat.

SELECT S.SNAME FROM SAILORS S WHERE S.SID NOT IN (SELECT R.SID FROM RESERVES R WHERE R.BID IN (SELECT B.BID FROM BOATS B WHERE B.COLOR = 'RED'));

```
SNAME
----------
BRUTUS
CANDY
RUSTY
ZOBRA
HORATIO
ART
BOB
```

CORRELATED SUB QUERIES:

In the Co-Related sub query a parent query will be executed first and based on the output of outer query the inner query execute.

If parent query returns N rows ,inner query executed for N times.

If a subquery references one or more columns of parent query is called CO-RELATED subquery because it is related to outer query. This subquery executes once for each and every row of main query.

Example1 :-

→Display employee names earning more than avg(sal) of their dept ?

```
SQL>SELECT ename FROM emp x
  WHERE sal > (SELECT AVG(sal) FROM emp
                    WHERE deptno=x.deptno);
```

Example2: Find sailors whose rating more than avg(rating) of their id.
SQL> SELECT S.SNAME FROM SAILORS S WHERE RATING > (SELECT AVG(RATING) FROM SAILORS WHERE SID=S.SID);

no rows selected.

SUB QUERIES WITH SET OPERATORS:

Q1) Find the names of sailors who have reserved a red or a green boat?

SQL> **Select s.sname from sailors s, reserves r, boats b where s.sid=r.sid and r.bid=b.bid and (b.color = ‘red’ or b.color= ‘green’);**

Or

SQL> Select s.sname from sailors s, reserves r, boats b where s.sid=r.sid and r.bid=b.bid and b.color=’red’

UNION

Select s.sname from sailors s, reserves r, boats b where s.sid=r.sid and r.bid=b.bid and b.color=’green’;

SNAME
Dustin
Lubber
Horatio

Q2) Find the names of sailors who have reserved a red and a green boat?

SQL> Select s.sname from sailors s, reserves r, boats b where s.sid=r.sid and r.bid=b.bid and b.color=’red’

INTERSECT

Select s.sname from sailors s, reserves r, boats b where s.sid=r.sid and r.bid=b.bid and b.color=’green’;

SNAME
Dustin
Lubber

Horatio

Q3) Find the names of sailors who have reserved a red boat but not green boat?

SQL> Select s.sname from sailors s, reserves r, boats b where s.sid=r.sid and r.bid=b.bid and b.color='red'

MINUS

Select s.sname from sailors s, reserves r, boats b where s.sid=r.sid and r.bid=b.bid and b.color='green';

NO ROWS SELECTED

Q4) Find all sids of sailors who have a rating of 10 or reserved boat 104?

SQL>**select s.sid from sailors s where s.rating=10**

UNION

Select r.sid from reserves r where r.bid=104;

SID
22
31
58
71