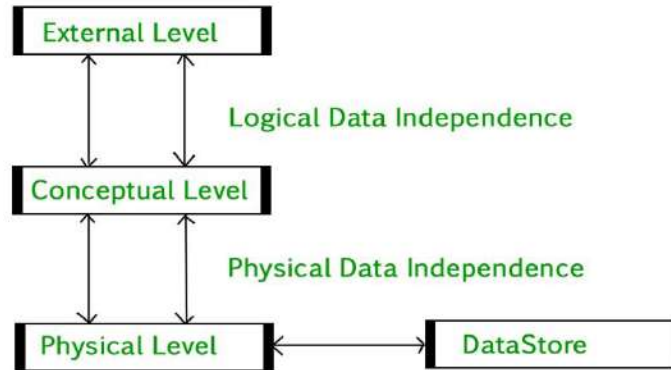


DBMS 3-tier Architecture

DBMS 3-tier architecture divides the complete system into three inter-related but independent modules as shown below:



1. **Physical Level:** At the physical level, the information about the location of database objects in the data store is kept. Various users of DBMS are unaware of the locations of these objects. In simple terms, physical level of a database describes how the data is being stored in secondary storage devices like disks and tapes and also gives insights on additional storage details.
2. **Conceptual Level:** At conceptual level, data is represented in the form of various database tables. For Example, STUDENT database may contain STUDENT and COURSE tables which will be visible to users but users are unaware of their storage. Also referred as logical schema, it describes what kind of data is to be stored in the database.
3. **External Level:** An external level specifies a view of the data in terms of conceptual level tables. Each external level view is used to cater to the needs of a particular category of users. For Example, FACULTY of a university is interested in looking course details of students; STUDENTS are interested in looking at all details related to academics, accounts, courses and hostel details

as well. So, different views can be generated for different users.
The main focus of external level is data abstraction.

Distributed Database

We define a distributed database as a collection of multiple, logically interrelated databases distributed over a computer network. A distributed database management system (distributed DBMS) is then defined as the software system that permits the management of the distributed database and makes the distribution transparent to the users. The two important terms in these definitions are “logically interrelated” and “distributed over a computer network.”

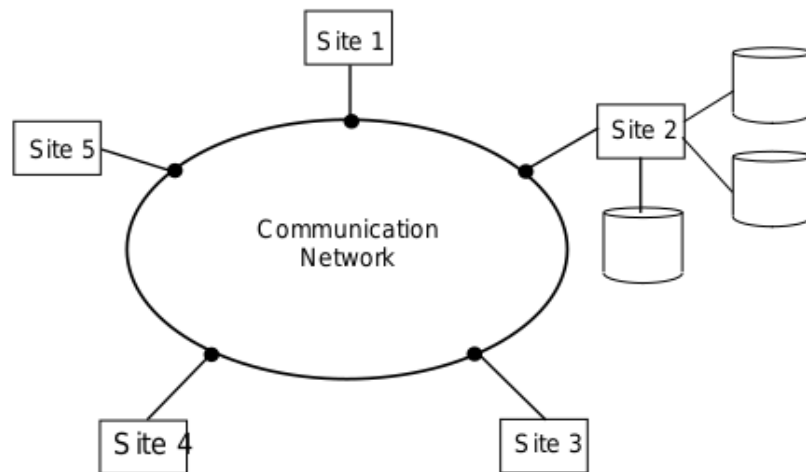


Fig. 1.3 Central Database on a Network

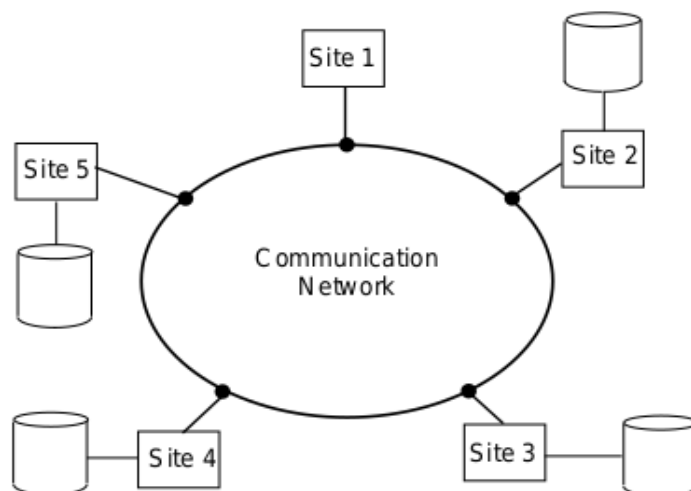


Fig. 1.4 DDBS Environment

Advantages of Distributed Database:

- Transparent Management of Distributed and Replicated Data
- Reliability Through Distributed Transactions
- Improved Performance
- Easier System Expansion

Disadvantages of Distributed Database System:

- Design Issues
- Cost of Update Replication , and Synchronization
- Cost of Security Constraints
- Recover from Failure and Synchronization

DBMS Versus File system

The following are the differences between DBMS and File System:

Basis of difference	Database Management System	File System
User requirement	It is a set of data and the user is not required to write the processes in DBMS.	It is also a collection of data but the user needs to write the procedures for handling the Database.
Protection mechanism	DBMS provides a brilliant mechanism for protection.	It is hard to protect a file.
Techniques	It has loads of different techniques to store and retrieve data.	It does not have the efficiency to store the data and retrieve it in a good way.
Locking	It uses some sort of locking to control the unauthorized access to the data.	In this type of data system, there is no control over the redirection and update of the information.
Data redundancy	It minimizes the chances of duplication of the data.	Data redundancy is more in this type of system.
Data Inconsistency	Less inconsistency in data.	More data inconsistency.

Database Architecture

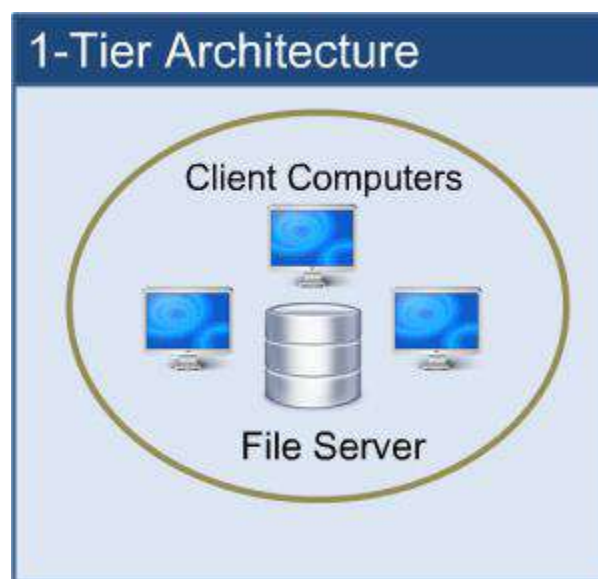
Database architecture uses programming languages to design a particular type of software for businesses or organizations. Database architecture focuses on the design, development, implementation and maintenance of computer programs that store and organize information for businesses, agencies and institutions. A database architect develops and implements software to meet the needs of users.

The design of a DBMS depends on its architecture. It can be centralized or decentralized or hierarchical. The architecture of a DBMS can be seen as either single tier or multi-tier. The tiers are classified as follows:

1. 1-tier architecture
2. 2-tier architecture
3. 3-tier architecture
4. n-tier architecture

1-tier architecture:

One-tier architecture involves putting all of the required components for a software application or technology on a single server or platform.

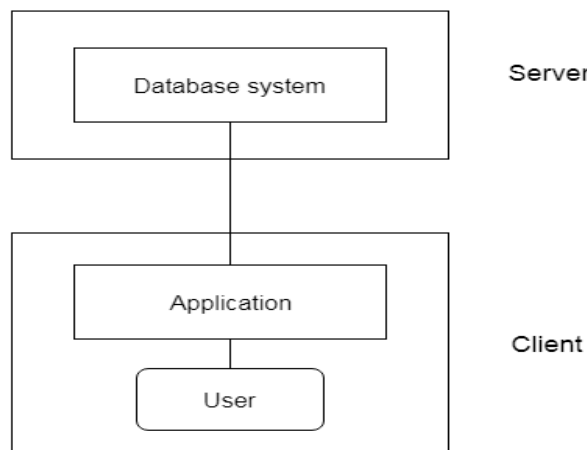


1-tier architecture

Basically, a one-tier architecture keeps all of the elements of an application, including the interface, Middleware and back-end data, in one place. Developers see these types of systems as the simplest and most direct way.

2-tier architecture:

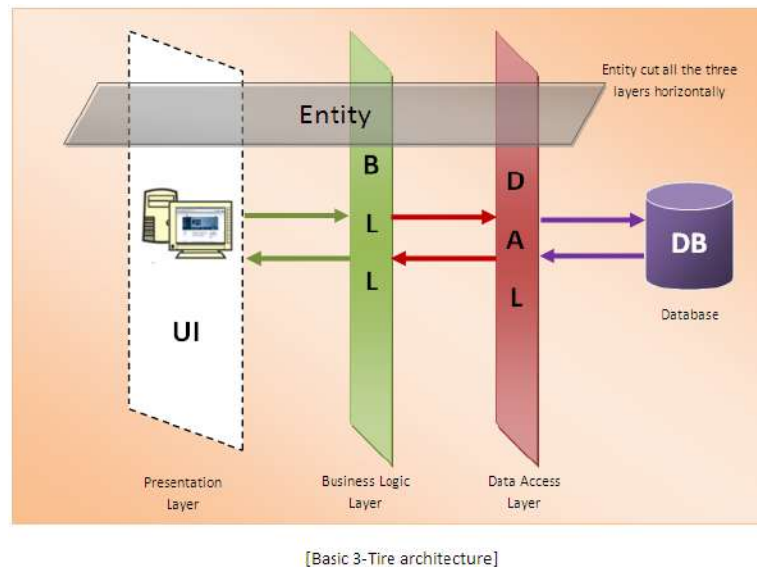
The two-tier is based on **Client Server architecture**. The two-tier architecture is like client server application. The direct communication takes place between client and server. There is no intermediate between client and server.



2-tier architecture

3-tier architecture:

The 3-tier architecture separates its tiers from each other based on the complexity of the users and how they use the data present in the database. **It is the most widely used architecture to design a DBMS.**



3-tier architecture

This architecture has different usages with different applications. It can be used in web applications and distributed applications. The strength in particular is when using this architecture over distributed systems.

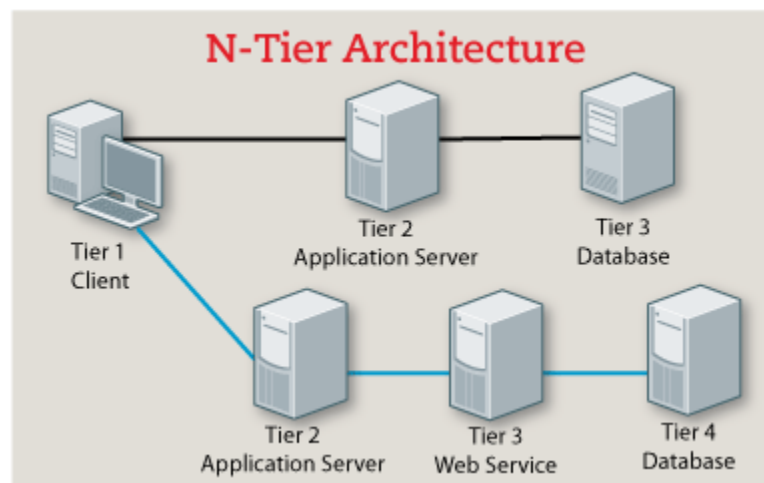
- **Database (Data) Tier** – At this tier, the database resides along with its query processing languages. We also have the relations that define the data and their constraints at this level.
- **Application (Middle) Tier** – At this tier reside the application server and the programs that access the database. For a user, this application tier presents an abstracted view of the database. End-users are unaware of any existence of the database beyond the application. At the other end, the database tier is not aware of any other user beyond the application tier. Hence, the application layer sits in the middle and acts as a mediator between the end-user and the database.
- **User (Presentation) Tier** – End-users operate on this tier and they know nothing about any existence of the database beyond this layer. At this layer, multiple views of the database can be provided by the

application. All views are generated by applications that reside in the application tier.

n-tier architecture:

N-tier architecture would involve dividing an application into [three different tiers](#). These would be the

1. Logic tier,
2. The presentation tier, and
3. The data tier.



n- Tier architecture

It is the physical separation of the different parts of the application as opposed to the usually conceptual or logical separation of the elements in the model-view-controller (MVC) framework. Another difference from the MVC framework is that n-tier layers are connected linearly, meaning all communication must go through the middle layer, which is the logic tier. In MVC, there is no actual middle layer because the interaction is triangular; the control layer has access to both the view and model layers and the model also accesses the view; the controller also creates a model based on the requirements and pushes this to the view. However, they are not mutually exclusive, as the MVC framework can be used in conjunction with the n-tier

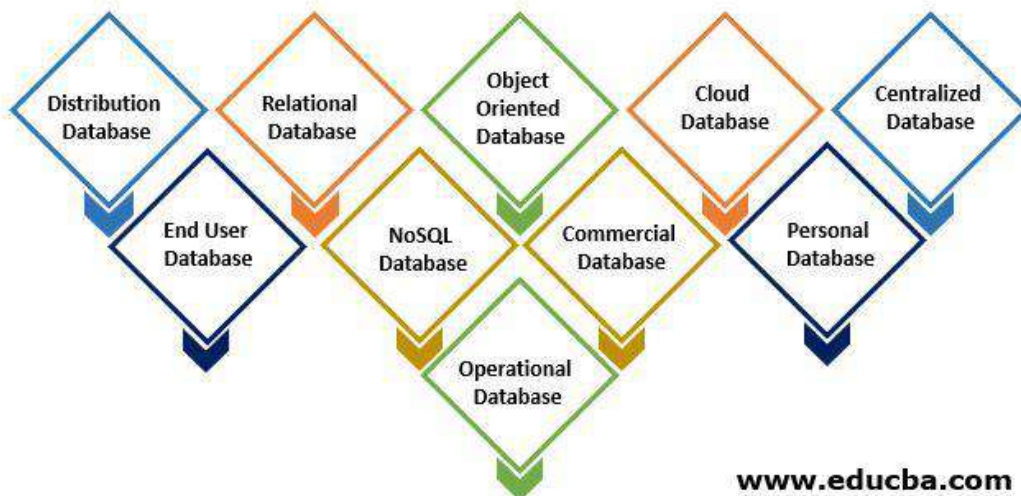
architecture, with the n-tier being the overall architecture used and MVC used as the framework for the presentation tier.

Database Types

Basically databases are data warehouses. Since we also have a book store in a public library, we can assume that a database of books is a library. However, closely defined, databases are computer frameworks which store, organize, protect and supply data. A database system is referred to as a system for the management of a database or DBM.

Different Types of Database

The following types of databases are available on the market, depending on the application requirements:



1. Distribution Database

In comparison to the centralized database idea, there are inputs from the general database and the information collected

from local computers. The data is not accessible in a single location and is distributed to various company sites. These sites are connected to each other through communication links that enable access to the data distributed.

A distributed database in which various parts of a database are located in different physical locations along with databases replicated and distributed between different points in a network can be imagined. Heterogeneous and homogenous are the two kinds of distribution database. Databases with the same base hardware and running on the same operating systems and applications are known as homogeneous DDBs. In different sites of the DDB defined as a heterogeneous DDB, operating systems, the underlying hardware and application procedures can be different.

2. Relational Database

Such databases are classified by a set of tables, in which data falls into a predefined classification. The table is made up of rows and columns with data input for a certain category and rows, with the example of the data identified by the category. The **Structured query language** is the standard interface of a relation-database user and application program. There are several basic operations which can be added to a table that enables the expansion of these databases, joining two commonly-related databases and modifying all existing applications.

3. Object Oriented Database

An object-driven database is an object-driven and relational database collection. There are different items, such as java, C++ that can be saved in a relational database using object-oriented programming languages but object-oriented databases are suitable for these components. An object-oriented database will be organized instead of actions around objects and data instead of logic. In contrast to an alphanumeric value, for example, a multimedia record in a relational database can be a definable data object.

4. Cloud Database

Now a day, data are actually stored in a public cloud, a hybrid cloud or a private cloud, also known as a virtual environment. A cloud database is an automated or built-in database for such a virtualized environment. A cloud service offers various advantages, including the ability to pay per user storage capacity and bandwidth and provides scalability on request, as well as high availability. In addition, a cloud platform allows companies to support enterprise applications in the delivery of **software as a service**.

5. Centralized Database

The data is stored centrally and users from various locations can access this data. This database includes hiring processes that help users even from a remote location to access the data.

For verification and validation of end-users, various types of authentication procedures are applied, and the application processes keeping a track and record of data utilization also provide registration numbers.

6. End User Database

The end-user is generally not worried about purchases or transactions at different levels and only understands the commodity that is a program or application. It is, therefore, a collaborative database that is designed specifically for the end-user as do the managers at various levels. This database offers a list of all the details.

7. NoSQL Database

These are used for large data sets. There are certain big data performance problems that are handled effectively by relational databases, and NoSQL databases can easily address such problems. The analysis of large-size, unstructured information can be done very efficiently on several cloud virtual servers.

8. Commercial Database

These are the paid versions of the enormous databases, designed for users who wish to access the information for assistance. These databases are specific subjects and such huge information can not be maintained. Commercial links provide access to such databases.

9. Personal Database

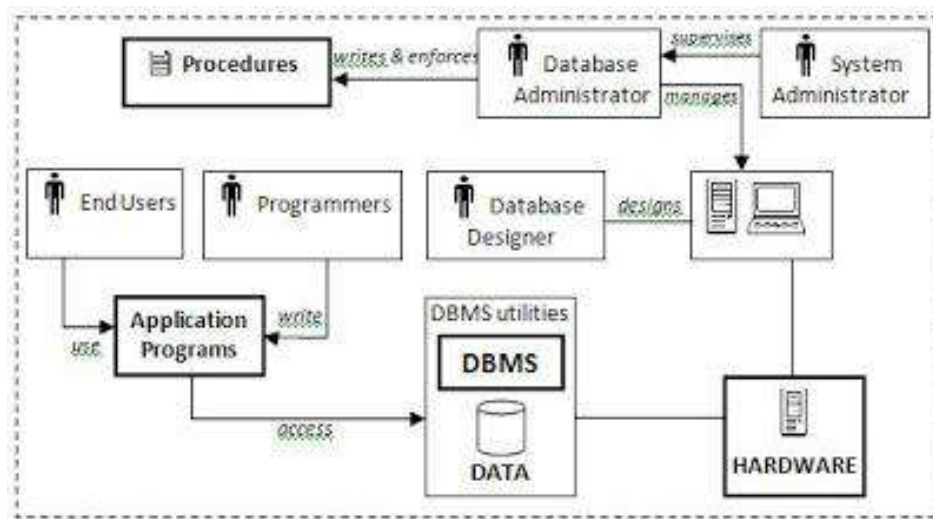
Data is collected and stored on small and easily manageable personal computers. The data are usually used by the same company department and are viewed by a small number of individuals.

10. Operational Database

In this folder, information on a company's operations is stored. These databases are needed for functional lines such as marketing, employee relationships, customer service, etc.

Database System Environment:

Database system refers to a set of components that define and control the collection, storage, management and use of data. It is composed of five major parts hardware, software, people, procedures and data.



1. Hardware: Hardware refers to all of the system's physical devices; for example, computers storage devices, printers, network devices and etc.

2. Software: To make the database system work properly, three types of software are needed: operating system, DBMS software, and application programs.

a) *Operating system:* It manages all hardware components and allows other software to run on the computers. Examples of operating system software include Windows, Linux and etc.

b) *DBMS software:* It manages the database within the database system. Some examples of DBMS software include Oracle, Access, MySql and etc.

c) *Application programs:* These are used to access and manipulate data in the DBMS and to manage the computer environment in which data access and manipulation take place. Application programs are most

commonly used to access data to generate reports. Most of the application programs provide GUI.

3. People: This component includes all users of the database system. According to the job nature, five types of users can be identified: systems administrators, database administrators, database designers, systems analysts and programmers, and end users.

a) *System administrators:* They supervise the database system's general operations.

b) *Database administrators:* They are also known as DBAs. They manage the DBMS and ensure that the database is functioning properly.

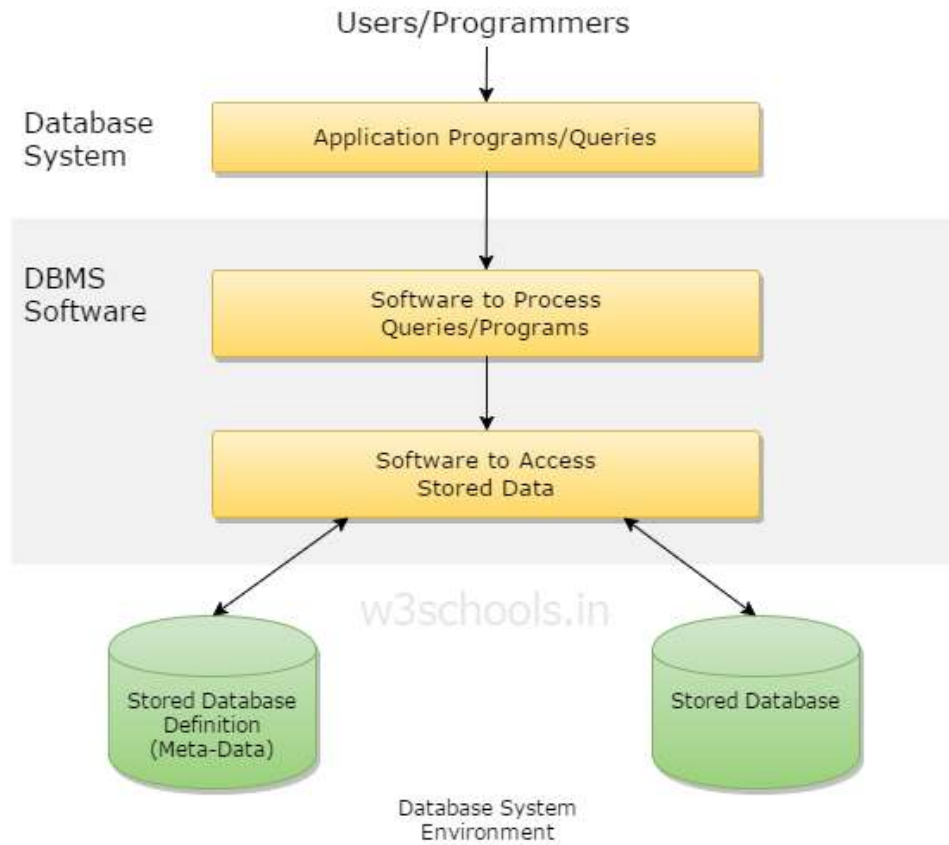
c) *Database designers:* They design the database structure. They are the database architects. As this is very critical, the designer's job responsibilities are increased.

d) *Systems analysts and programmers:* They design and implement the application programs. They design and create the data entry screens, reports, and procedures through which end users can access and manipulate the data.

e) *End users:* They are the people who use the application programs to run the organization's daily operations. For example, sales-clerks, supervisors, managers are classified as end users.

4. Procedures: Procedures are the instructions and rules that supervise the design and use of the database system. Procedures are a critical component of the system. Procedures play an important role in a company because they enforce the standards by which business is conducted in an organization

5. Data: Data refers the collection of facts stored in the database. Because data are the raw material from which information is generated, no database can exist without database.



Data Modeling: Conceptual, Logical, Physical

Data Model Types

What is Data Modeling?

Data modeling (data modeling) is the process of creating a data model for the data to be stored in a database. This data model is a conceptual representation of **Data objects**, the associations between different data objects, and the rules. Data modeling helps in the visual representation of data and enforces business rules, regulatory compliances, and government policies on the data. Data Models ensure consistency in naming conventions, default values, semantics, and security while ensuring quality of the data.

Data Model

The **Data Model** is defined as an abstract model that organizes data description, data semantics, and consistency constraints of data. The data model emphasizes on what data is needed and how it should be organized instead of what operations will be performed on data. Data Model is like an architect's building plan, which helps to build conceptual models and set a relationship between data items.

The two types of Data Modeling Techniques are

1. Entity Relationship (E-R) Model
2. UML (Unified Modelling Language)

We will discuss them in detail later.

Data Modeling concepts in detail-

- [Why use Data Model?](#)
- [Types of Data Models](#)

- [Conceptual Data Model](#)
- [Logical Data Model](#)
- [Physical Data Model](#)
- [Advantages and Disadvantages of Data Model](#)

Note:

Data object:

The data object is actually a location or region of storage that contains a collection of attributes or groups of values that act as an aspect, characteristic, quality, or descriptor of the object. A vehicle is a data object which can be defined or described with the help of a set of attributes or data.

Example: Sales databases such as customers, store items, sales.

Meta data:

Metadata in DBMS is the data ([details/schema](#)) of any other data. It can also be defined as data about data. The word 'Meta' is the prefix that is generally the technical term for self-referential. In other words, we can say that **Metadata** is the summarized data for the contextual data

Why use Data Model?

The primary goals of using data model are:

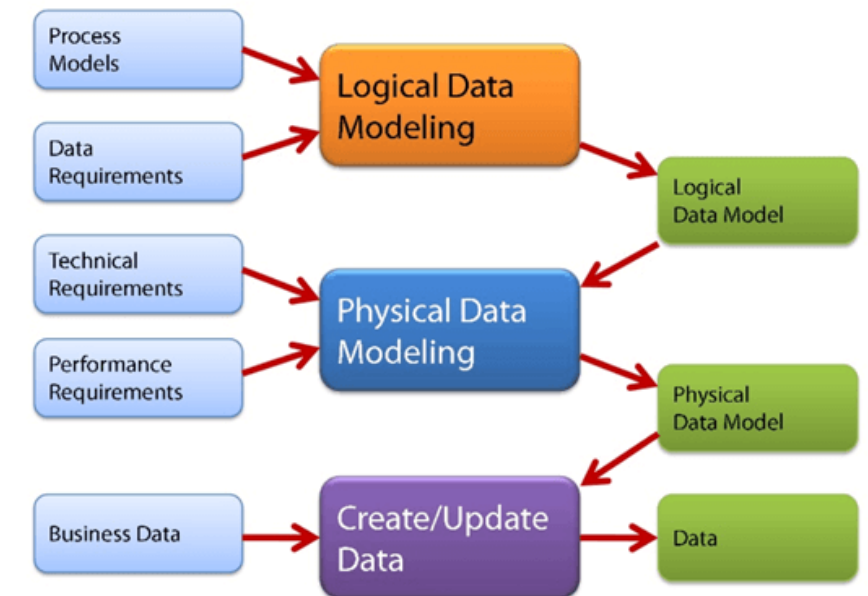
- Ensures that all data objects required by the database are accurately represented. Omission of data will lead to creation of faulty reports and produce incorrect results.
- A data model helps design the database at the conceptual, physical and logical levels.
- Data Model structure helps to define the relational tables, primary and foreign keys and stored procedures.

- It provides a clear picture of the base data and can be used by database developers to create a physical database.
- It is also helpful to identify missing and redundant data.
- Though the initial creation of data model is labor and time consuming, in the long run, it makes your IT infrastructure upgrade and maintenance cheaper and faster.

Types of Data Models

Types of Data Models: There are mainly three different types of data models: conceptual data models, logical data models, and physical data models, and each one has a specific purpose. The data models are used to represent the data and how it is stored in the database and to set the relationship between data items.

1. **Conceptual Data Model:** This Data Model defines **WHAT the system contains**. This model is typically created by Business stakeholders and Data Architects. The purpose is to organize scope and define business concepts and rules.
2. **Logical Data Model:** Defines **HOW the system should be implemented regardless of the DBMS**. This model is typically created by Data Architects and Business Analysts. The purpose is to developed technical map of rules and data structures.
3. **Physical Data Model:** This Data Model **describes HOW the system will be implemented using a specific DBMS system**. This model is typically created by DBA and developers. **The purpose is actual implementation of the database.**



Types of Data Model

Conceptual Data Model

A **Conceptual Data Model** is an organized view of database concepts and their relationships. The purpose of creating a conceptual data model is to establish entities, their attributes, and relationships. In this data modeling level, there is hardly any detail available on the actual database structure. Business stakeholders and data architects typically create a conceptual data model.

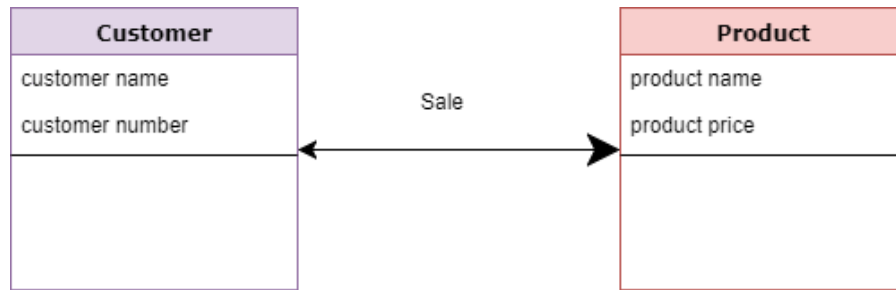
The 3 basic tenants of Conceptual Data Model are

- **Entity:** A real-world thing
- **Attribute:** Characteristics or properties of an entity
- **Relationship:** Dependency or association between two entities

Data model example:

- Customer and Product are two entities. Customer number and name are attributes of the Customer entity
- Product name and price are attributes of product entity

- Sale is the relationship between the customer and product



Conceptual Data Model

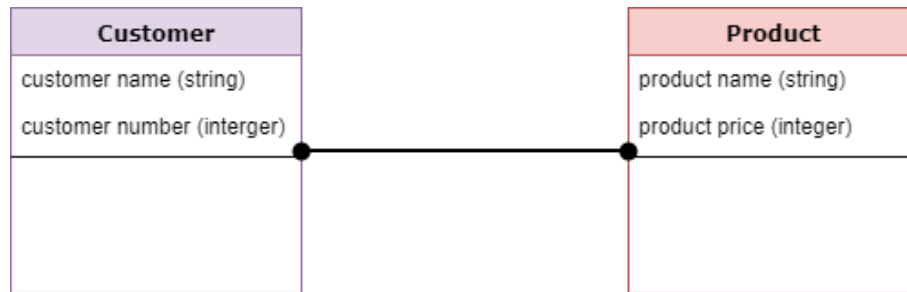
Characteristics of a conceptual data model

- Offers Organization-wide coverage of the business concepts.
- This type of Data Models are designed and developed for a business audience.
- The conceptual model is developed independently of hardware specifications like data storage capacity, location or software specifications like DBMS vendor and technology. The focus is to represent data as a user will see it in the "real world."

Conceptual data models known as Domain models create a common vocabulary for all stakeholders by establishing basic concepts and scope.

Logical Data Model

The **Logical Data Model** is used to define the structure of data elements and to set relationships between them. The logical data model adds further information to the conceptual data model elements. The advantage of using a Logical data model is to provide a foundation to form the base for the Physical model. However, the modeling structure remains generic.



Logical Data Model

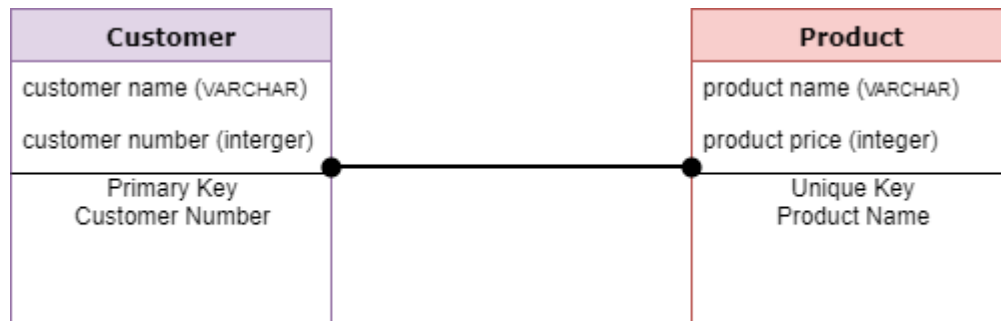
At this Data Modeling level, no primary or secondary key is defined. At this Data modeling level, you need to verify and adjust the connector details that were set earlier for relationships.

Characteristics of a Logical data model

- Describes data needs for a single project but could integrate with other logical data models based on the scope of the project.
- Designed and developed independently from the DBMS.
- Data attributes will have data types with exact precisions and length.
- Normalization processes to the model is applied typically till 3NF.

Physical Data Model

A **Physical Data Model** describes a database-specific implementation of the data model. It offers database abstraction and helps generate the schema. This is because of the richness of meta-data offered by a Physical Data Model. The physical data model also helps in visualizing database structure by replicating database column keys, constraints, indexes, triggers, and other RDBMS features.



Physical Data Model

Characteristics of a physical data model:

- The physical data model describes data need for a single project or application though it may be integrated with other physical data models based on project scope.
- Data Model contains relationships between tables that which addresses cardinality and null ability of the relationships.
- Developed for a specific version of a DBMS, location, data storage or technology to be used in the project.
- Columns should have exact datatypes, lengths assigned and default values.
- Primary and Foreign keys, views, indexes, access profiles, and authorizations, etc. are defined.

Advantages and Disadvantages of Data Model:

Advantages of Data model:

- The main goal of a designing data model is to make certain that data objects offered by the functional team are represented accurately.
- The data model should be detailed enough to be used for building the physical database.

- The information in the data model can be used for defining the relationship between tables, primary and foreign keys, and stored procedures.
- Data Model helps business to communicate the within and across organizations.
- Data model helps to documents data mappings in ETL process
- Help to recognize correct sources of data to populate the model

Disadvantages of Data model:

- To develop Data model one should know physical data stored characteristics.
- This is a navigational system produces complex application development, management. Thus, it requires a knowledge of the biographical truth.
- Even smaller change made in structure require modification in the entire application.
- There is no set data manipulation language in DBMS.

Conclusion

- Data modeling is the process of developing data model for the data to be stored in a Database.
- Data Models ensure consistency in naming conventions, default values, semantics, security while ensuring quality of the data.
- Data Model structure helps to define the relational tables, primary and foreign keys and stored procedures.
- There are three types of conceptual, logical, and physical.
- The main aim of conceptual model is to establish the entities, their attributes, and their relationships.
- Logical data model defines the structure of the data elements and set the relationships between them.

- A Physical Data Model describes the database specific implementation of the data model.
- The main goal of a designing data model is to make certain that data objects offered by the functional team are represented accurately.
- The biggest drawback is that even smaller change made in structure require modification in the entire application.
- Reading this Data Modeling tutorial, you will learn from the basic concepts such as What is Data Model? Introduction to different types of Data Model, advantages, disadvantages, and data model example.

Data Independence in DBMS: Physical & Logical with Examples

What is Data Independence of DBMS?

Data Independence is defined as a property of DBMS that helps you to change the Database schema at one level of a database system without requiring to change the schema at the next higher level. Data independence helps you to keep data separated from all programs that make use of it.

In this tutorial, you will learn:

- [What is Data Independence of DBMS?](#)
- [Types of Data Independence](#)
- [Levels of Database](#)
- [Physical Data Independence](#)
- [Logical Data Independence](#)
- [Difference between Physical and Logical Data Independence](#)
- [Importance of Data Independence](#)

Types of Data Independence

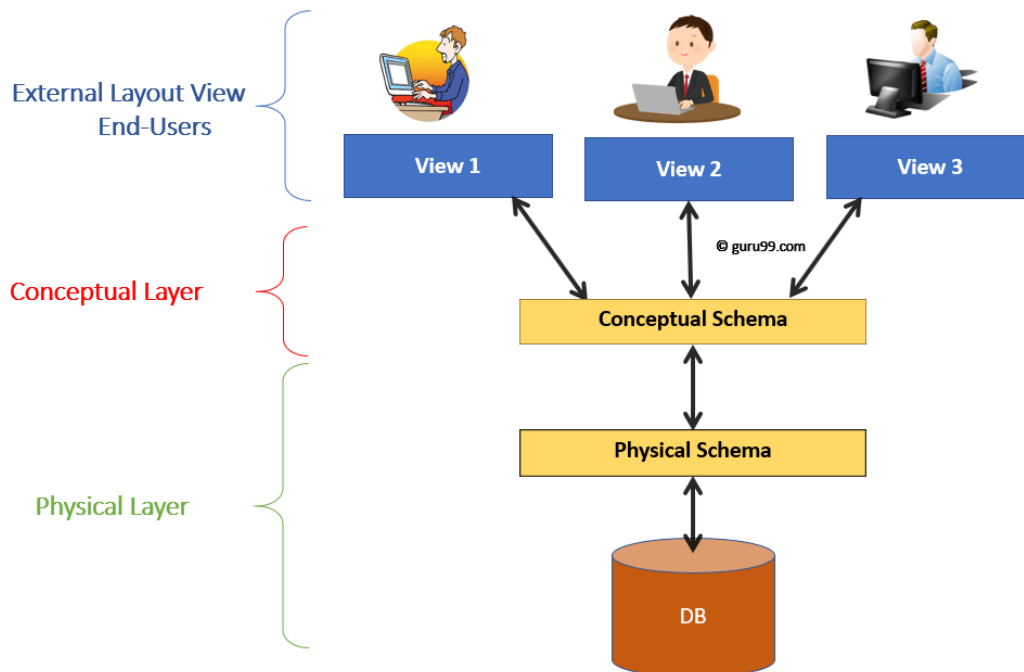
In DBMS there are two types of data independence

1. Physical data independence
2. Logical data independence.

Levels of Database

Before we learn Data Independence, a refresher on Database Levels is important. The database has 3 levels as shown in the diagram below

1. Physical/Internal
2. Conceptual/Logical
3. External/View



Levels of DBMS Architecture Diagram

Consider an Example of a University Database. At the different levels this is how the implementation will look like:

Type of Schema	Implementation
External Schema	View 1: Course info(cid:int,cname:string) View 2: studeninfo (id: int. name: string)
Conceptual Schema	Students(id: int, name: string, login: string, age: integer) Courses(id: int, cname.string, credits:integer) Enrolled(id: int, grade:string)
Physical Schema	<ul style="list-style-type: none"> • Relations stored as unordered files. • Index on the first column of Students.

Physical Data Independence

Physical data independence helps you to separate conceptual levels from the internal/physical levels. It allows you to provide a logical description of the database without the need to specify physical structures. Compared to Logical Independence, it is easy to achieve physical data independence.

With Physical independence, you can easily change the physical storage structures or devices with an effect on the conceptual schema. Any change done would be absorbed by the mapping between the conceptual and internal levels. Physical data independence is achieved by the presence of the internal level of the database and then the transformation from the conceptual level of the database to the internal level.

Examples of changes under Physical Data Independence

Due to Physical independence, any of the below change will not affect the conceptual layer.

- Using a new storage device like Hard Drive or Magnetic Tapes
- Modifying the file organization technique in the Database
- Switching to different data structures.
- Changing the access method.
- Modifying indexes.
- Changes to compression techniques or hashing algorithms.
- Change of Location of Database from say C drive to D Drive

Logical Data Independence

Logical Data Independence is the ability to change the conceptual scheme without changing

1. External views
2. External API or programs

Any change made will be absorbed by the mapping between external and conceptual levels.

When compared to Physical Data independence, it is challenging to achieve logical data independence.

Examples of changes under Logical Data Independence

Due to Logical independence, any of the below change will not affect the external layer.

1. Add/Modify/Delete a new attribute, entity or relationship is possible without a rewrite of existing application programs
2. Merging two records into one
3. Breaking an existing record into two or more records

Difference between Physical and Logical Data Independence

Logical Data Independence	Physical Data Independence
Logical Data Independence is mainly concerned with the structure or changing the data definition.	Mainly concerned with the storage of the data.
It is difficult as the retrieving of data is mainly dependent on the logical structure of data.	It is easy to retrieve.
Compared to Logic Physical independence it is difficult to	Compared to Logical Independence it is easy to achieve physical

achieve logical data independence.	data independence.
You need to make changes in the Application program if new fields are added or deleted from the database.	A change in the physical level usually does not need change at the Application program level.
Modification at the logical levels is significant whenever the logical structures of the database are changed.	Modifications made at the internal levels may or may not be needed to improve the performance of the structure.
Concerned with conceptual schema	Concerned with internal schema
Example: Add/Modify/Delete a new attribute	Example: change in compression techniques, hashing algorithms, storage devices, etc

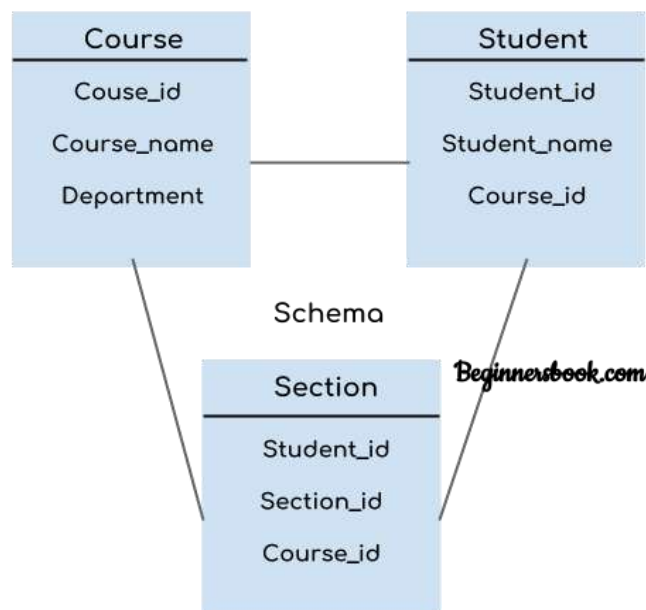
Importance of Data Independence

- Helps you to improve the quality of the data
- Database system maintenance becomes affordable
- Enforcement of standards and improvement in database security
- You don't need to alter data structure in application programs
- Permit developers to focus on the general structure of the Database rather than worrying about the internal implementation
- It allows you to improve state which is undamaged or undivided
- Database incongruity is vastly reduced.
- Easily make modifications in the physical level is needed to improve the performance of the system.

DBMS Schema

Definition of schema: Design of a database is called the schema. Schema is of three types: **Physical schema, logical schema and view schema.**

For example: In the following diagram, we have a schema that shows the relationship between three tables: Course, Student and Section. The diagram only shows the design of the database, it doesn't show the data present in those tables. **Schema is only a structural view(design) of a database** as shown in the diagram below.



The design of a database at physical level is called **physical schema**, how the data stored in blocks of storage is described at this level.

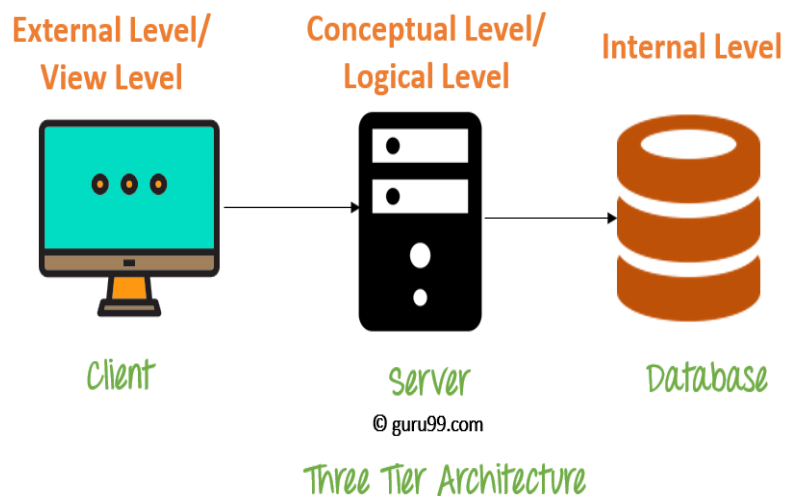
Design of database at logical level is called logical schema, programmers and database administrators work at this level, at this level data can be described as certain types of data records gets stored in data structures, however **the internal details such as**

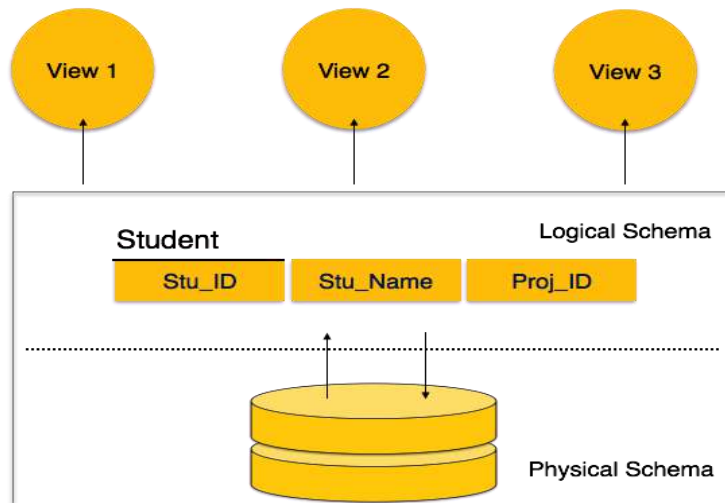
implementation of data structure is hidden at this level (available at physical level).

Design of database at view level is called **view schema or external schema**. This generally describes end user interaction with database systems.

There are mainly three levels of data abstraction:

1. **Physical schema or Internal Level:** Actual PHYSICAL storage structure and access paths.
2. **Logical schema or Conceptual or Logical Level:** Structure and constraints (like keys) for the entire database
3. **External schema or View level:** Describes various user views





Physical schema or Internal Level

The internal schema defines the physical storage structure of the database. The internal schema is a very low-level representation of the entire database. It contains multiple occurrences of multiple types of internal record. In the ANSI term, it is also called "stored record".

Facts about Internal schema:

- The internal schema is the lowest level of data abstraction
- It helps you to keep information about the actual representation of the entire database. Like the actual storage of the data on the disk in the form of records
- The internal view tells us what data is stored in the database and how
- It never deals with the physical devices. Instead, internal schema views a physical device as a collection of physical pages

Logical schema or Conceptual or Logical Level:

The conceptual schema describes the Database structure of the whole database for the community of users. This schema hides information about the physical storage structures and focuses on describing data types, entities, relationships, etc.

This logical level comes between the user level and physical storage view. However, there is only single conceptual view of a single database.

Facts about Conceptual schema:

- Defines all database entities, their attributes, and their relationships
- Security and integrity information
- In the conceptual level, the data available to a user must be contained in or derivable from the physical level

External schema or View level:

An external schema describes the part of the database which specific user is interested in. It hides the unrelated details of the database from the user. There may be "n" number of external views for each database.

Each external view is defined using an external schema, which consists of definitions of various types of external record of that specific view.

An external view is just the content of the database as it is seen by some specific particular user. For example, a user from the sales department will see only sales related data.

Facts about external schema:

- An external level is only related to the data which is viewed by specific end users.
- This level includes some external schemas.
- External schema level is nearest to the user
- The external schema describes the segment of the database which is needed for a certain user group and hides the remaining details from the database from the specific user group

Advantages Database Schema

- You can manage data independent of the physical storage
- Faster Migration to new graphical environments
- DBMS Architecture allows you to make changes on the presentation level without affecting the other two layers
- As each tier is separate, it is possible to use different sets of developers
- It is more secure as the client doesn't have direct access to the database business logic
- In case of the failure of the one-tier no data loss as you are always secure by accessing the other tier

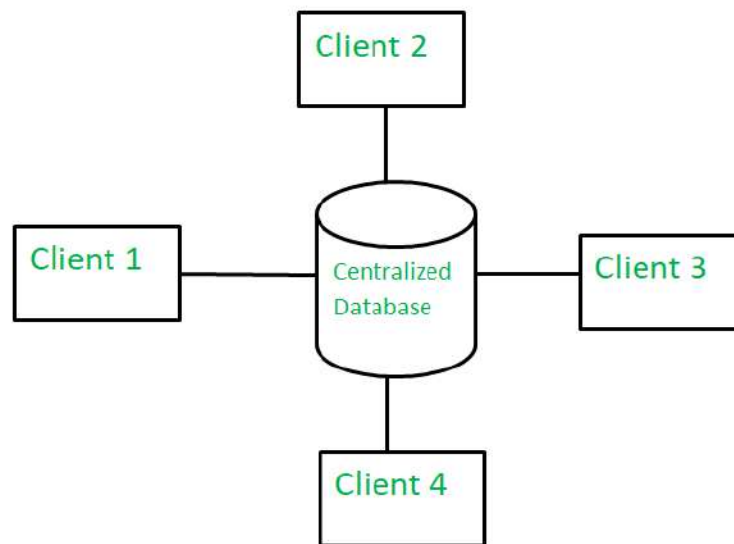
DBMS Instance

Definition of instance: The data stored in database at a particular moment of time is called instance of database. Database schema defines the variable declarations in tables that belong to a particular database; the value of these variables at a moment of time is called the instance of that database.

For example, lets say we have a single table student in the database, today the table has 100 records, so today the instance of the database has 100 records. Lets say we are going to add another 100 records in this table by tomorrow so the instance of database tomorrow will have 200 records in table. In short, at a particular moment the data stored in database is called the instance, that changes over time when we add or delete data from the database.

1. Centralized Database:

A centralized database is basically a type of database that is stored, located as well as maintained at a single location only. This type of database is modified and managed from that location itself. This location is thus mainly any database system or a centralized computer system. The centralized location is accessed via an internet connection (LAN, WAN, etc). This centralized database is mainly used by institutions or organizations.



Advantages -

- Since all data is stored at a single location only thus it is easier to access and co-ordinate data.
- The centralized database has very minimal data redundancy since all data is stored at a single place.
- It is cheaper in comparison to all other databases available.

Disadvantages -

- The data traffic in case of centralized database is more.

- If any kind of system failure occurs at centralized system then entire data will be destroyed.

- **what is a database management system(DBMS)**

"A database management system (DBMS) is a collection of programs that manages the database structure and controls access to the data stored in the database".

The DBMS serves as the intermediary between the user and the database. The database structure itself is stored as a collection of files, So, we can access the data in those files through the DBMS.

The DBMS receives all application requests and translates them into the complex operations required to fulfill those requests. The DBMS hides much of the database's internal complexity from the application programs and users.

Advantages of Database Management System (DBMS)

1. Improved data sharing

An advantage of the database management approach is, the DBMS helps to create an environment in which end users have better access to more and better-managed data.

Such access makes it possible for end users to respond quickly to changes in the environment.

2. Improved data security

The more users access the data, the greater the risks of data security breaches. Corporations invest considerable amounts of time, effort, and money to ensure that corporate data are used properly. A DBMS provides a framework for better enforcement of data privacy and security policies.

3. Better data integration

Wider access to well-managed data promotes an integrated view of the organization's operations and a clearer view of the big picture. It becomes much easier to see how actions in one segment of the company affect other segments.

4. Minimized data inconsistency

Data inconsistency exists when different versions of the same data appear in different places. For example, data inconsistency exists when a company's sales department stores a sales representative's name as "Bill Brown" and the company's personnel department stores that same person's name as "William G. Brown," or when the company's regional sales office shows the price of a product as \$45.95 and its national sales office shows the same product's price as \$43.95. The probability of data inconsistency is greatly reduced in a properly designed database.

5. Improved data access

The DBMS makes it possible to produce quick answers to ad hoc queries. From a database perspective, a query is a specific request issued to the DBMS for data manipulation—for example, to read or update the data. Simply put, a query is a question, and an ad hoc query is a spur-of-the-moment question. The DBMS sends back an answer (called the query result set) to the application. For example, end users, when dealing with large amounts of sales data, might want quick answers to questions (ad hoc queries) such as:

- What was the dollar volume of sales by product during the past six months?
- What is the sales bonus figure for each of our salespeople during

the past three months?

- How many of our customers have credit balances of 3,000 or more?

6. Improved decision making

Better-managed data and improved data access make it possible to generate better-quality information, on which better decisions are based. The quality of the information generated depends on the quality of the underlying data. Data quality is a comprehensive approach to promoting the accuracy, validity, and timeliness of the data. While the DBMS does not guarantee data quality, it provides a framework to facilitate data quality initiatives.

7. Increased end-user productivity

The availability of data, combined with the tools that transform data into usable information, empowers end users to make quick, informed decisions that can make the difference between success and failure in the global economy.

Till now we have seen different benefits of database management systems. But it has certain limitations or disadvantages.

Let's find various disadvantages of database system.

Disadvantages of Database Management System (DBMS):

Although the database system yields considerable advantages over previous data management approaches, database systems do carry significant disadvantages. For example:

1. Increased costs

one of the disadvantages of DBMS is Database systems require sophisticated hardware and software and highly skilled personnel. The cost of maintaining the hardware, software, and personnel required to

operate and manage a database system can be substantial. Training, licensing, and regulation compliance costs are often overlooked when database systems are implemented.

2. Management complexity

Database systems interface with many different technologies and have a significant impact on a company's resources and culture. The changes introduced by the adoption of a database system must be properly managed to ensure that they help advance the company's objectives. Given the fact that database systems hold crucial company data that are accessed from multiple sources, security issues must be assessed constantly.

3. Maintaining currency

To maximize the efficiency of the database system, you must keep your system current. Therefore, you must perform frequent updates and apply the latest patches and security measures to all components.

Because database technology advances rapidly, personnel training costs tend to be significant. Vendor dependence. Given the heavy investment in technology and personnel training, companies might be reluctant to change database vendors.

As a consequence, vendors are less likely to offer pricing point advantages to existing customers, and those customers might be limited in their choice of database system components.

4. Frequent upgrade/replacement cycles

DBMS vendors frequently upgrade their products by adding new functionality. Such new features often come bundled in new upgrade versions of the software. Some of these versions require hardware upgrades. Not only do the upgrades themselves cost money, but it also

costs money to train database users and administrators to properly use and manage the new features.

Relationship in DBMS-

A relationship is defined as an association among several entities.

Example-

'Enrolled in' is a relationship that exists between entities **Student** and **Course**.



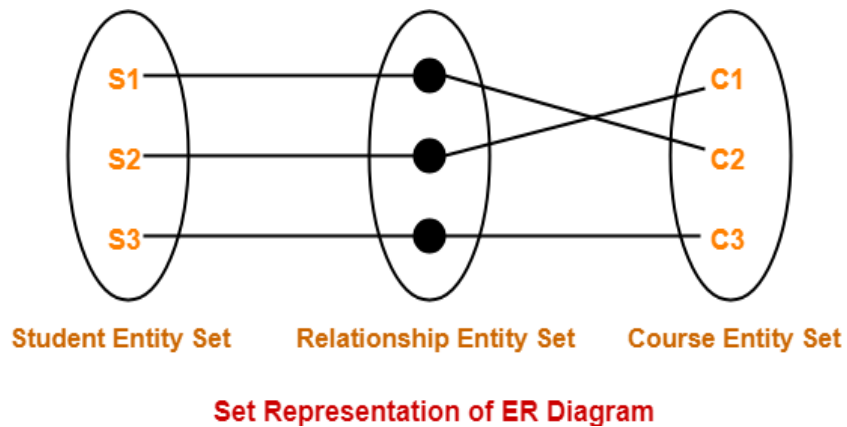
Also read- [Entity Sets in DBMS](#)

Relationship Set-

A relationship set is a set of relationships of same type.

Example-

Set representation of above ER diagram is-



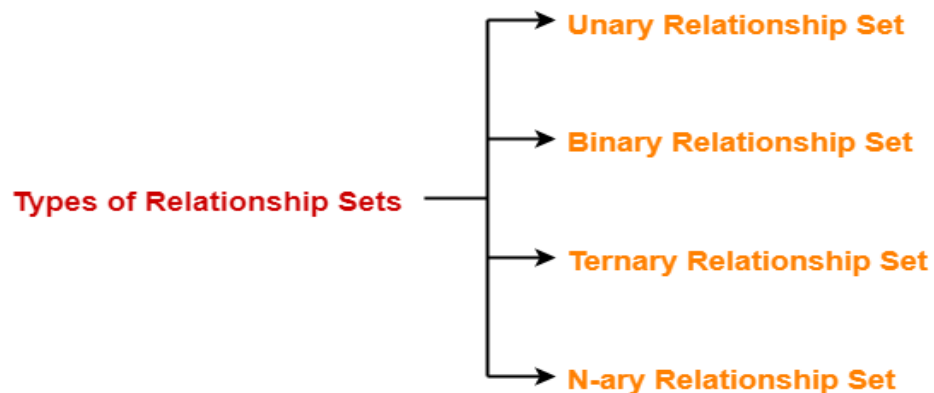
Degree of a Relationship Set-

The number of entity sets that participate in a relationship set is termed as the degree of that relationship set. Thus,

Degree of a relationship set = Number of entity sets participating in a relationship set

Types of Relationship Sets-

On the basis of degree of a relationship set, a relationship set can be classified into the following types-



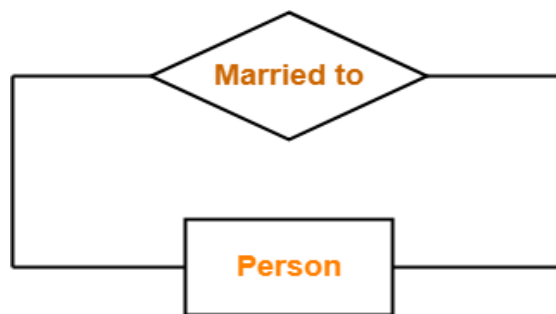
1. Unary relationship set
2. Binary relationship set
3. Ternary relationship set
4. N-ary relationship set

1. Unary Relationship Set-

Unary relationship set is a relationship set where only one entity set participates in a relationship set.

Example-

One person is married to only one person



Unary Relationship Set

2. Binary Relationship Set-

Binary relationship set is a relationship set where two entity sets participate in a relationship set.

Example-

Student is enrolled in a Course

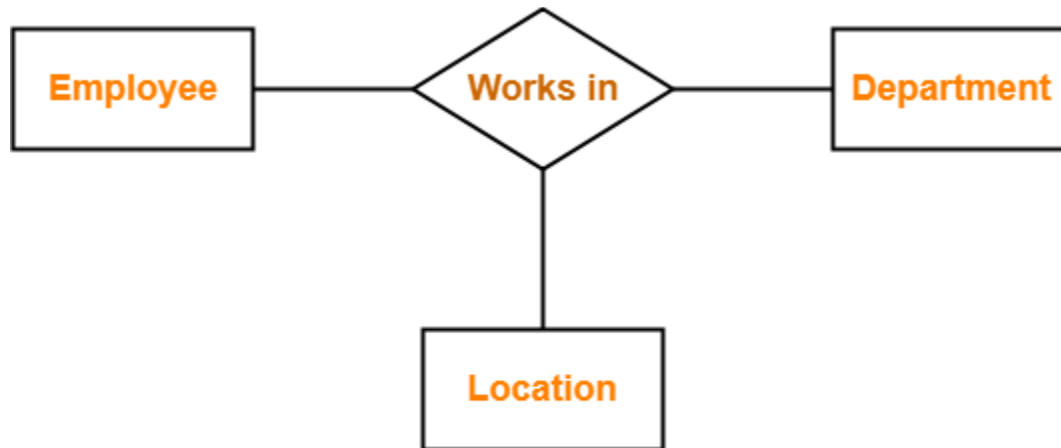


Binary Relationship Set

3. Ternary Relationship Set-

Ternary relationship set is a relationship set where three entity sets participate in a relationship set.

Example-



Ternary Relationship Set

4. N-ary Relationship Set-

N-ary relationship set is a relationship set where 'n' entity sets participate in a relationship set.

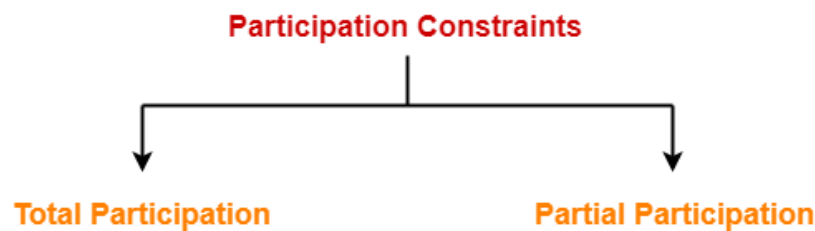
- A relationship represents the association between two or more entities
- The relationship also ***shows the different entity sets that are participating in a relationship***, these relationships are very much useful in analyzing the design process of the system

Participation Constraints-

Participation constraints define the least number of relationship instances in which an entity must compulsorily participate.

Types of Participation Constraints-

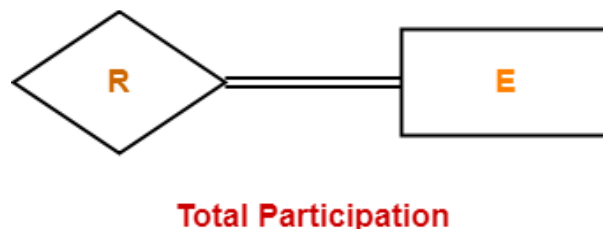
There are two types of participation constraints-



1. Total participation
2. Partial participation

1. Total Participation-

- It specifies that each entity in the entity set must compulsorily participate in at least one relationship instance in that relationship set.
- That is why, it is also called as **mandatory participation**.
- Total participation is represented using a double line between the entity set and relationship set.



Example-

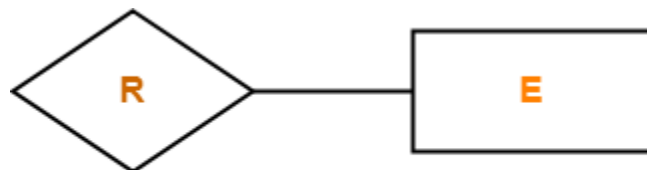


Here,

- Double line between the entity set “Student” and relationship set “Enrolled in” signifies total participation.
- It specifies that each student must be enrolled in at least one course.

2. Partial Participation-

- It specifies that each entity in the entity set may or may not participate in the relationship instance in that relationship set.
- That is why, it is also called as **optional participation**.
- Partial participation is represented using a single line between the entity set and relationship set.



Partial Participation

Example-



Here,

- Single line between the entity set “Course” and relationship set “Enrolled in” signifies partial participation.
- It specifies that there might exist some courses for which no enrollments are made.

Relationship between Cardinality and Participation Constraints-

Minimum cardinality tells whether the participation is partial or total.

- If minimum cardinality = 0, then it signifies partial participation.
- If minimum cardinality = 1, then it signifies total participation.

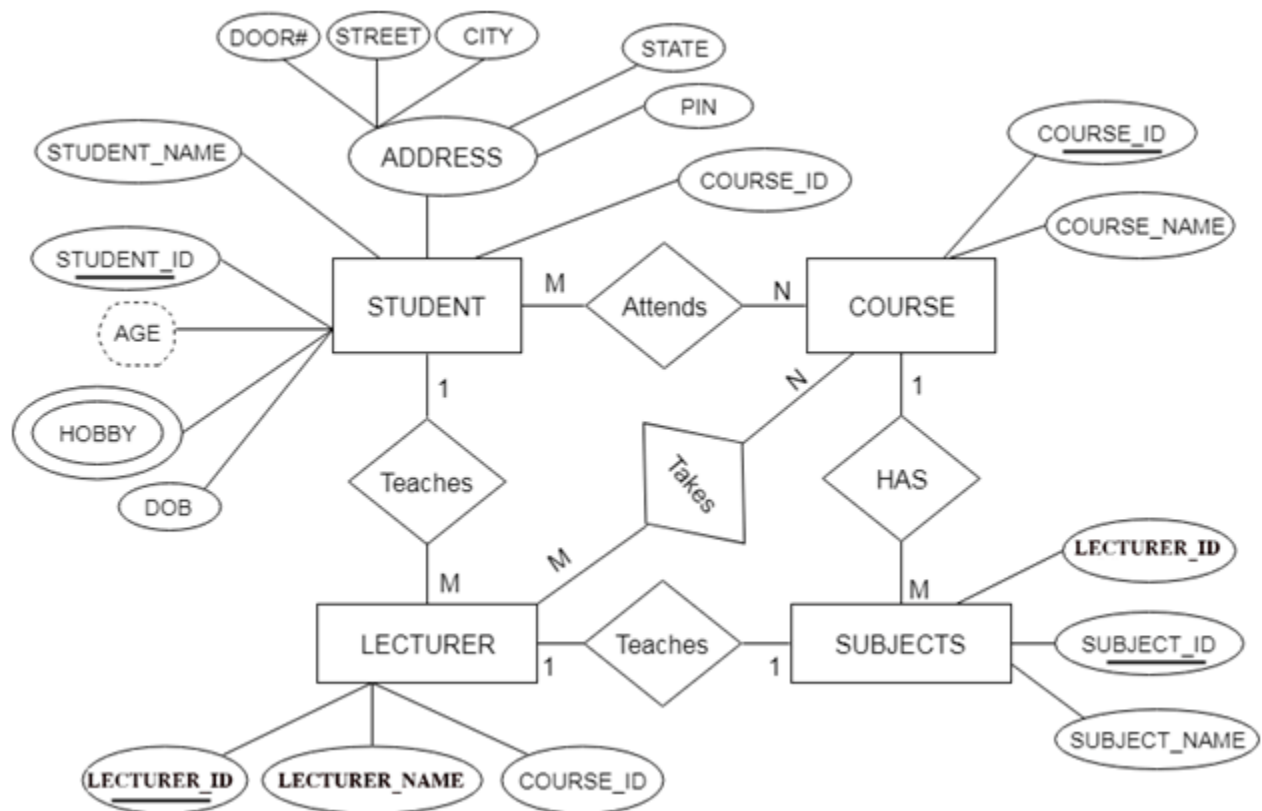
Maximum cardinality tells the maximum number of entities that participates in a relationship set.

Reduction of ER diagram to Table

The database can be represented using the notations, and these notations can be reduced to a collection of tables.

In the database, every entity set or relationship set can be represented in tabular form.

The ER diagram is given below:



There are some points for converting the ER diagram to the table:

- **Entity type becomes a table.**

In the given ER diagram, LECTURE, STUDENT, SUBJECT and COURSE forms individual tables.

- **All single-valued attribute becomes a column for the table.**

In the STUDENT entity, STUDENT_NAME and STUDENT_ID form the column of STUDENT table. Similarly, COURSE_NAME and COURSE_ID form the column of COURSE table and so on.

- **A key attribute of the entity type represented by the primary key.**

In the given ER diagram, COURSE_ID, STUDENT_ID, SUBJECT_ID, and LECTURE_ID are the key attribute of the entity.

- **The multivalued attribute is represented by a separate table.**

In the student table, a hobby is a multivalued attribute. So it is not possible to represent multiple values in a single column of STUDENT table. Hence we create a table STUD_HOBBY with column name STUDENT_ID and HOBBY. Using both the column, we create a composite key.

- **Composite attribute represented by components.**

In the given ER diagram, student address is a composite attribute. It contains CITY, PIN, DOOR#, STREET, and STATE. In the STUDENT table, these attributes can merge as an individual column.

- **Derived attributes are not considered in the table.**

In the STUDENT table, Age is the derived attribute. It can be calculated at any point of time by calculating the difference between current date and Date of Birth.

Using these rules, you can convert the ER diagram to tables and columns and assign the mapping between the tables. Table structure for the given ER diagram is as below:

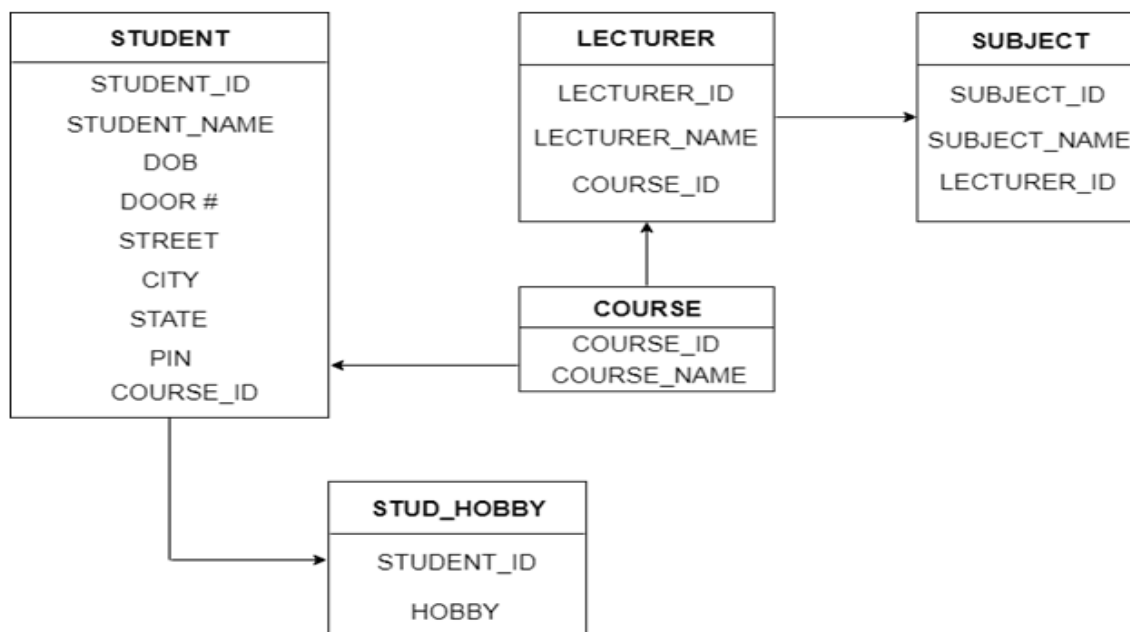


Figure: Table structure

Converting ER Diagrams to Tables-

After designing an **ER Diagram**,

- ER diagram is converted into the tables in relational model.
- This is because relational models can be easily implemented by RDBMS like MySQL , Oracle etc.

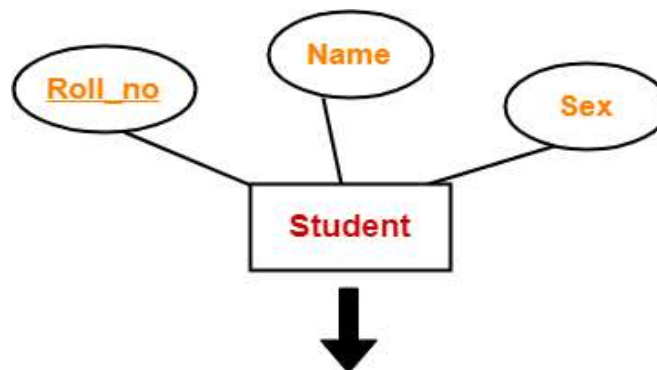
Following rules are used for converting an ER diagram into the tables-

Rule-01: For Strong Entity Set With Only Simple Attributes-

A strong entity set with only simple attributes will require only one table in relational model.

- Attributes of the table will be the attributes of the entity set.
- The primary key of the table will be the key attribute of the entity set.

Example-



<u>Roll_no</u>	Name	Sex

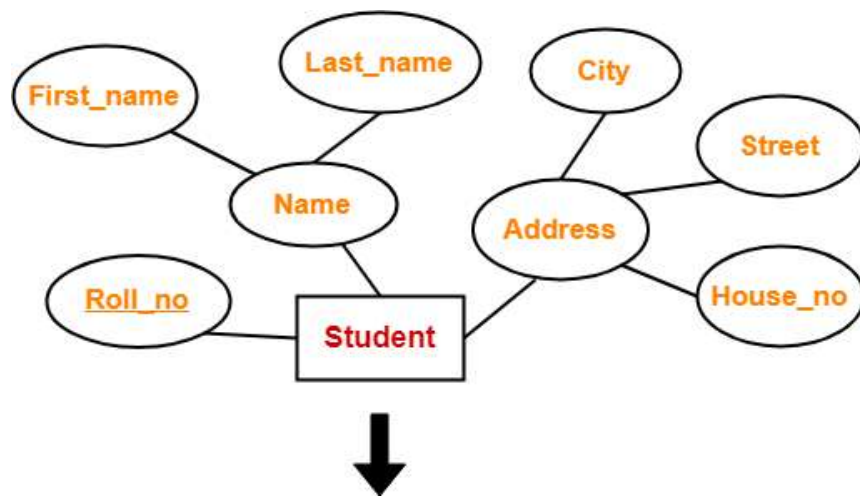
Schema : Student (Roll_no , Name , Sex)

Entity Sets in DBMS: Strong Entity set and Weak Entity set.

Rule-02: For Strong Entity Set With Composite Attributes-

- A strong entity set with any number of composite attributes will require only one table in relational model.
- While conversion, simple attributes of the composite attributes are taken into account and not the composite attribute itself.

Example-



<u>Roll_no</u>	First_name	Last_name	House_no	Street	City

Schema : Student (Roll_no , First_name , Last_name , House_no , Street , City)

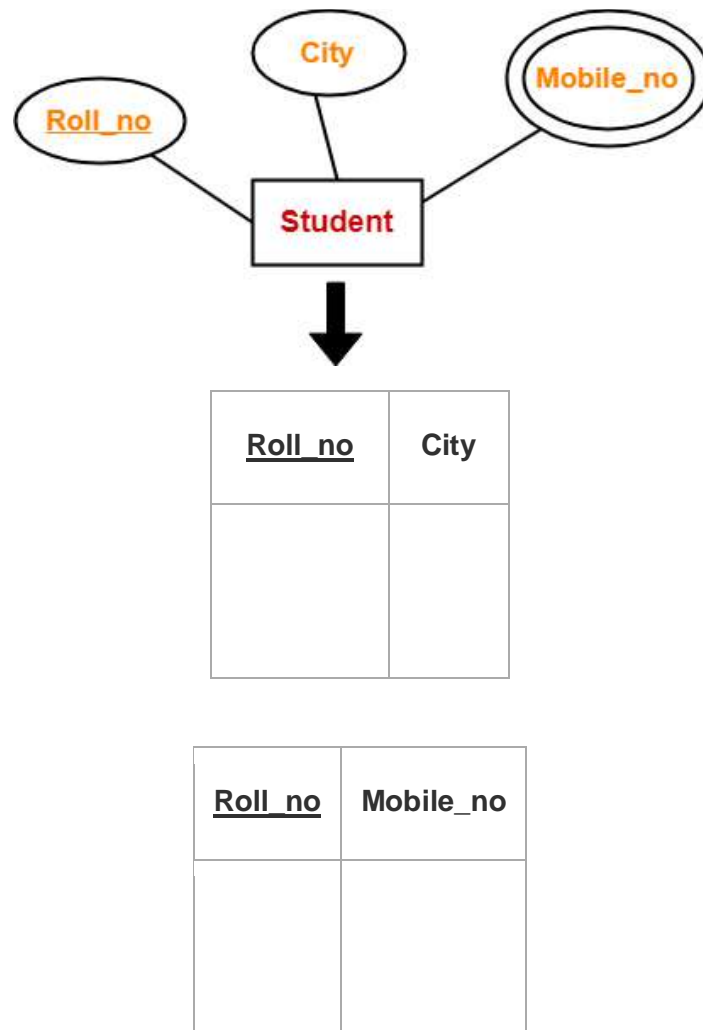
Types of Attributes in DBMS: Simple attributes, Composite attributes, Single valued attributes
Multi valued attributes, Derived attributes, Key attributes

Rule-03: For Strong Entity Set With Multi Valued Attributes-

A strong entity set with any number of multi valued attributes will require two tables in relational model.

- One table will contain all the simple attributes with the primary key.
- Other table will contain the primary key and all the multi valued attributes.

Example-



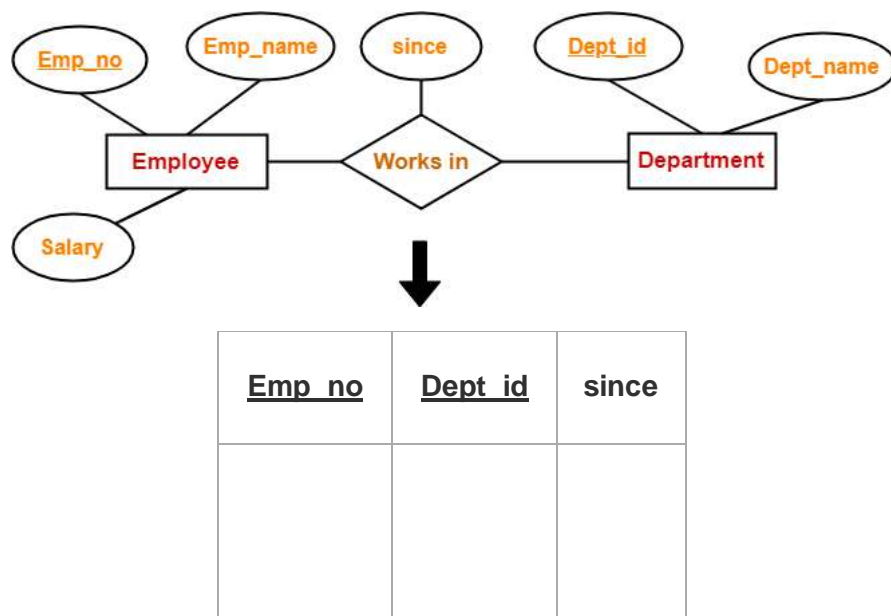
Rule-04: Translating Relationship Set into a Table-

A relationship set will require one table in the relational model.

Attributes of the table are-

- Primary key attributes of the participating entity sets
 - Its own descriptive attributes if any.
- Set of non-descriptive attributes will be the primary key.

Example-



Schema : Works in (Emp_no , Dept_id , since)

NOTE-

If we consider the overall ER diagram, three tables will be required in relational model-

- One table for the entity set "Employee"
- One table for the entity set "Department"
- One table for the relationship set "Works in"

<u>Emp_no</u>	Emp_name	Salary

<u>Dept_id</u>	Dept_name

Rule-05: For Binary Relationships With Cardinality Ratios-

The following four cases are possible-

Case-01: Binary relationship with cardinality ratio m:n

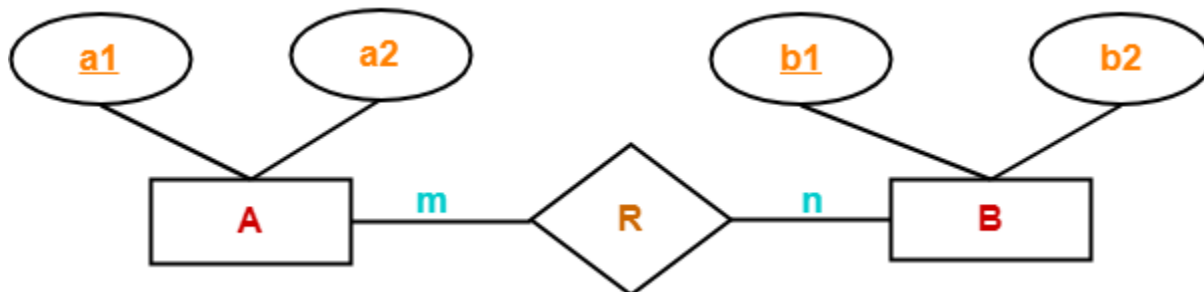
Case-02: Binary relationship with cardinality ratio 1:n

Case-03: Binary relationship with cardinality ratio m:1

Case-04: Binary relationship with cardinality ratio 1:1

Also read- **Cardinality Ratios in DBMS**

Case-01: For Binary Relationship With Cardinality Ratio m:n



Here, three tables will be required-

1. A (a1 , a2)
2. R (a1 , b1)
3. B (b1 , b2)

<u>a1</u>	a2

Table-A

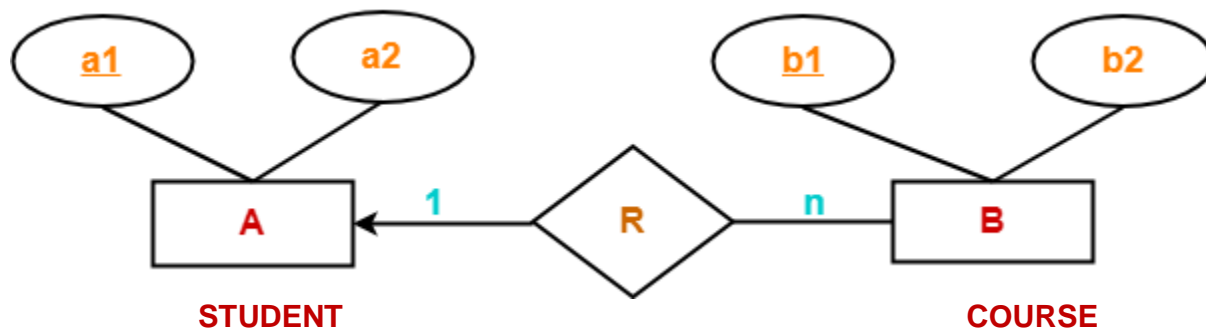
<u>a1</u>	b1

Table-R

<u>b1</u>	b2

Table-B

Case-02: For Binary Relationship With Cardinality Ratio 1:n



Here, two tables will be required-

1. A (a1 , a2)
2. BR (a1 , b1 , b2)

<u>a1</u>	a2

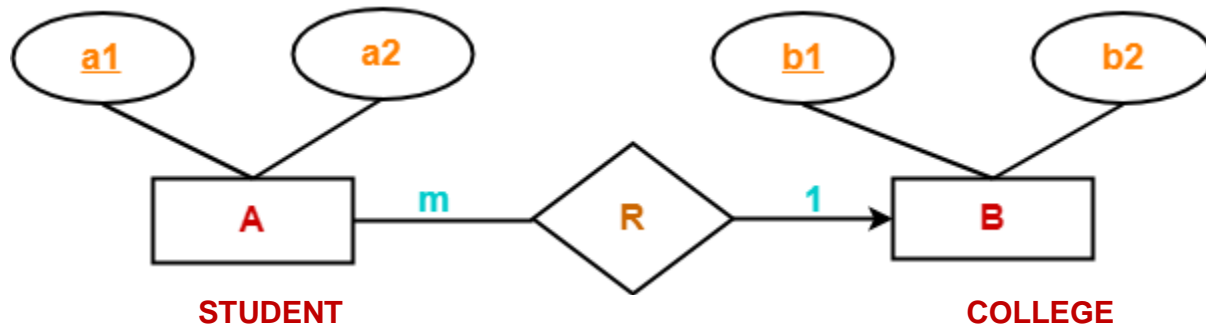
Table-A

<u>a1</u>	b1	B2

Table-BR

NOTE- Here, combined table will be drawn for the entity set B and relationship set R.

Case-03: For Binary Relationship With Cardinality Ratio m:1



Here, two tables will be required-

1. AR (a1 , a2 , b1)
2. B (b1 , b2)

<u>a1</u>	a2	<u>b1</u>

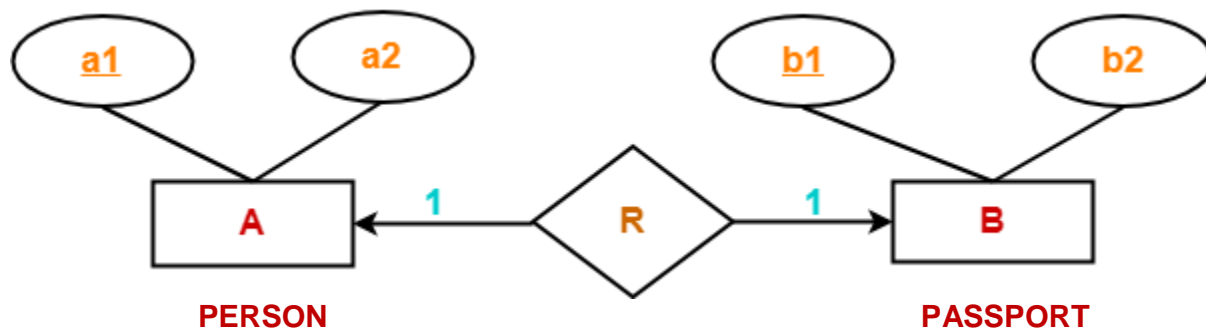
Table-AR

<u>b1</u>	b2

Table-B

NOTE- Here, combined table will be drawn for the entity set A and relationship set R.

Case-04: For Binary Relationship With Cardinality Ratio 1:1



Here, two tables will be required. Either combine 'R' with 'A' or 'B'

Way-01:

1. AR (a1 , a2 , b1)
2. B (b1 , b2)

<u>b1</u>	b2

Table-B

<u>a1</u>	a1	b1

Table-AR

Way-02:

1. A (a1 , a2)
2. BR (a1 , b1 , b2)

<u>a1</u>	a2

Table-A

Because cardinality ratio = 1 : n , so we will combine the entity set B and relationship set R.

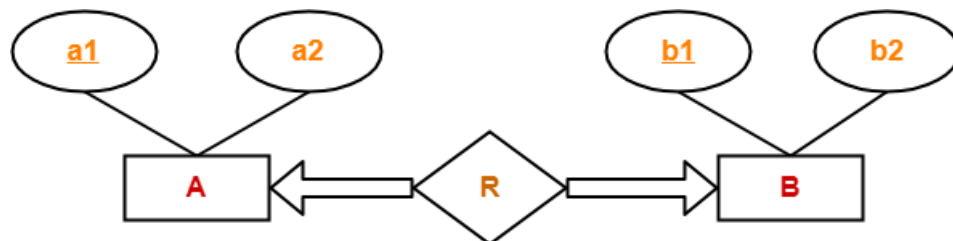
Then, two tables will be required-

1. A (a1 , a2)
2. BR (a1 , b1 , b2)

Because of total participation, foreign key a1 has acquired NOT NULL constraint, so it can't be null now.

Case-02: For Binary Relationship With Cardinality Constraint and Total Participation Constraint From Both Sides-

If there is a key constraint from both the sides of an entity set with total participation, then that binary relationship is represented using only single table.

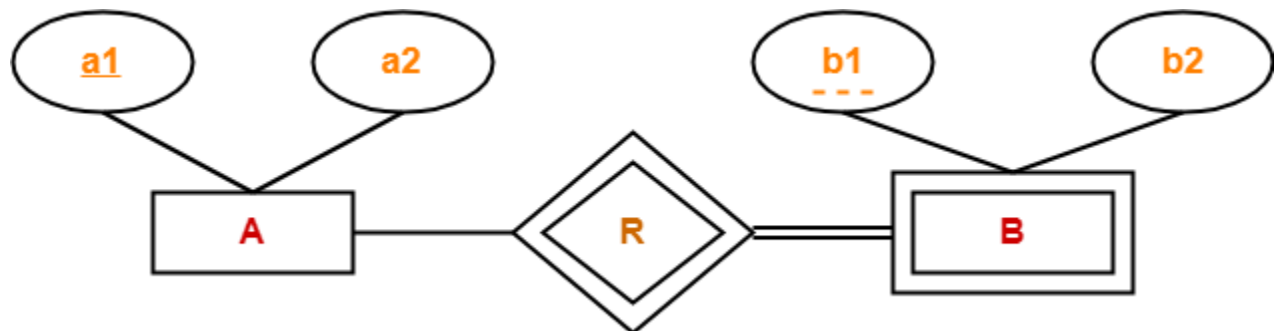


Here, Only one table is required.

- ARB (a1 , a2 , b1 , b2)

Rule-07: For Binary Relationship With Weak Entity Set-

Weak entity set always appears in association with identifying relationship with total participation constraint.



Here, two tables will be required-

1. A (a1 , a2)
2. BR (a1 , b1 , b2)

Entity Set in DBMS-

An entity set is a set of same type of entities.

An entity refers to any object having-

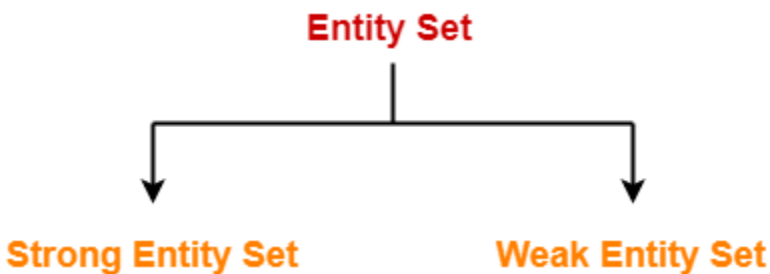
- Either a physical existence such as a particular person, office, house or car.
- Or a conceptual existence such as a school, a university, a company or a job.

In ER diagram,

- Attributes are associated with an entity set.
- Attributes describe the properties of entities in the entity set.
- Based on the values of certain attributes, an entity can be identified uniquely.

Types of Entity Sets-

An entity set may be of the following two types-



1. Strong entity set
2. Weak entity set

1. Strong Entity Set-

- A strong entity set is an entity set that contains sufficient attributes to uniquely identify all its entities.

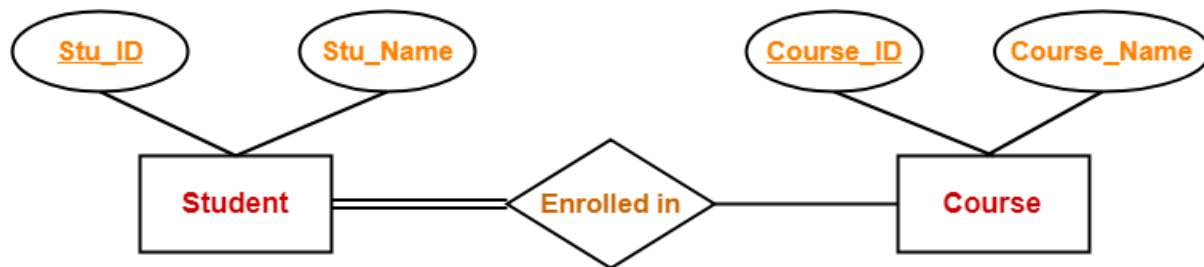
- In other words, a primary key exists for a strong entity set.
- Primary key of a strong entity set is represented by underlining it.

Symbols Used-

- A single rectangle is used for representing a strong entity set.
- A diamond symbol is used for representing the relationship that exists between two strong entity sets.
- A single line is used for representing the connection of the strong entity set with the relationship set.
- A double line is used for representing the total participation of an entity set with the relationship set.
- Total participation may or may not exist in the relationship.

Example-

Consider the following ER diagram-



In this ER diagram,

- Two strong entity sets “**Student**” and “**Course**” are related to each other.
- Student ID and Student name are the attributes of entity set “Student”.
- Student ID is the primary key using which any student can be identified uniquely.
- Course ID and Course name are the attributes of entity set “Course”.
- Course ID is the primary key using which any course can be identified uniquely.
- Double line between Student and relationship set signifies total participation.
- It suggests that each student must be enrolled in at least one course.

- Single line between Course and relationship set signifies partial participation.
- It suggests that there might exist some courses for which no enrollments are made.

2. Weak Entity Set-

- A weak entity set is an entity set that does not contain sufficient attributes to uniquely identify its entities.
- In other words, a primary key does not exist for a weak entity set.
- However, it contains a partial key called as a **discriminator**.
- Discriminator can identify a group of entities from the entity set.
- Discriminator is represented by underlining with a dashed line.

NOTE-

- The combination of discriminator and primary key of the strong entity set makes it possible to uniquely identify all entities of the weak entity set.
- Thus, this combination serves as a primary key for the weak entity set.
- Clearly, this primary key is not formed by the weak entity set completely.

Primary key of weak entity set

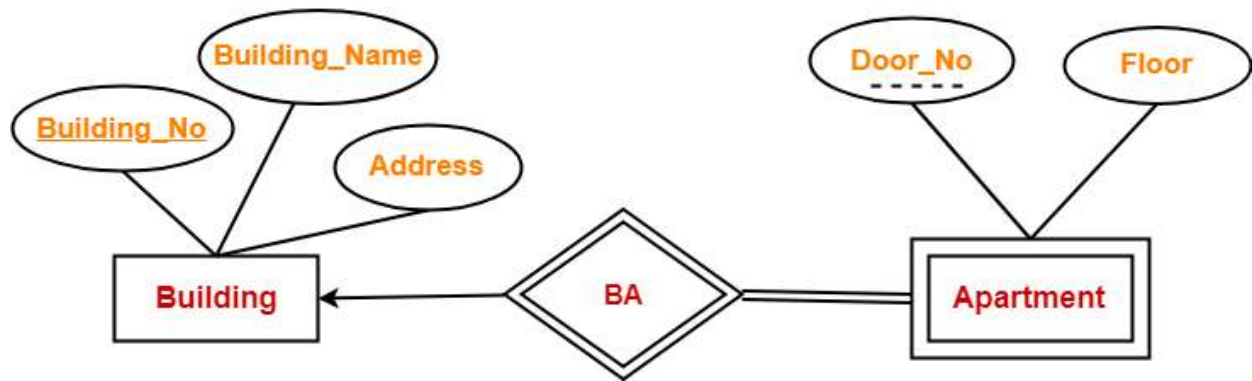
= Its own discriminator + Primary key of strong entity set

Symbols Used-

- A double rectangle is used for representing a weak entity set.
- A double diamond symbol is used for representing the relationship that exists between the strong and weak entity sets and this relationship is known as **identifying relationship**.
- A double line is used for representing the connection of the weak entity set with the relationship set.
- Total participation always exists in the identifying relationship.

Example-

Consider the following ER diagram-



In this ER diagram,

- One strong entity set “**Building**” and one weak entity set “**Apartment**” are related to each other.
- Strong entity set “Building” has building number as its primary key.
- Door number is the discriminator of the weak entity set “Apartment”.
- This is because door number alone can not identify an apartment uniquely as there may be several other buildings having the same door number.
- Double line between Apartment and relationship set signifies total participation.
- It suggests that each apartment must be present in at least one building.
- Single line between Building and relationship set signifies partial participation.
- It suggests that there might exist some buildings which has no apartment.

To uniquely identify any apartment,

- First, building number is required to identify the particular building.
- Secondly, door number of the apartment is required to uniquely identify the apartment.

Thus,

Primary key of Apartment

= Primary key of Building + Its own discriminator

= Building number + Door number

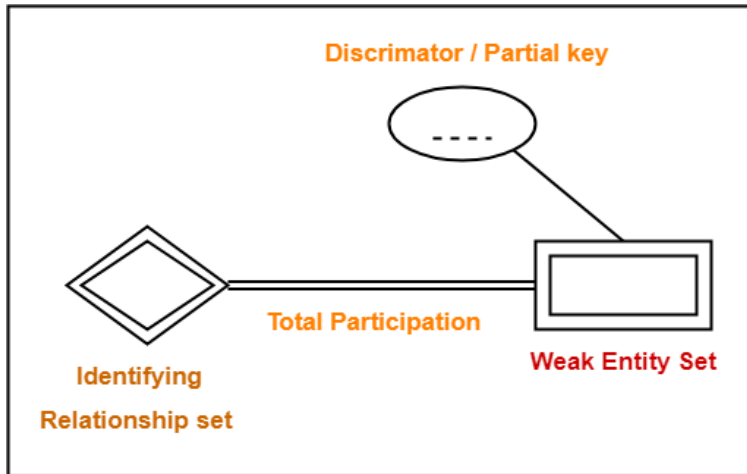
Differences between Strong entity set and Weak entity set-

Strong entity set	Weak entity set
A single rectangle is used for the representation of a strong entity set.	A double rectangle is used for the representation of a weak entity set.
It contains sufficient attributes to form its primary key.	It does not contain sufficient attributes to form its primary key.
A diamond symbol is used for the representation of the relationship that exists between the two strong entity sets.	A double diamond symbol is used for the representation of the identifying relationship that exists between the strong and weak entity set.
A single line is used for the representation of the connection between the strong entity set and the relationship.	A double line is used for the representation of the connection between the weak entity set and the relationship set.
Total participation may or may not exist in the relationship.	Total participation always exists in the identifying relationship.

Important Note-

In ER diagram, weak entity set is always present in total participation with the identifying relationship set.

So, we always have the picture like shown here-

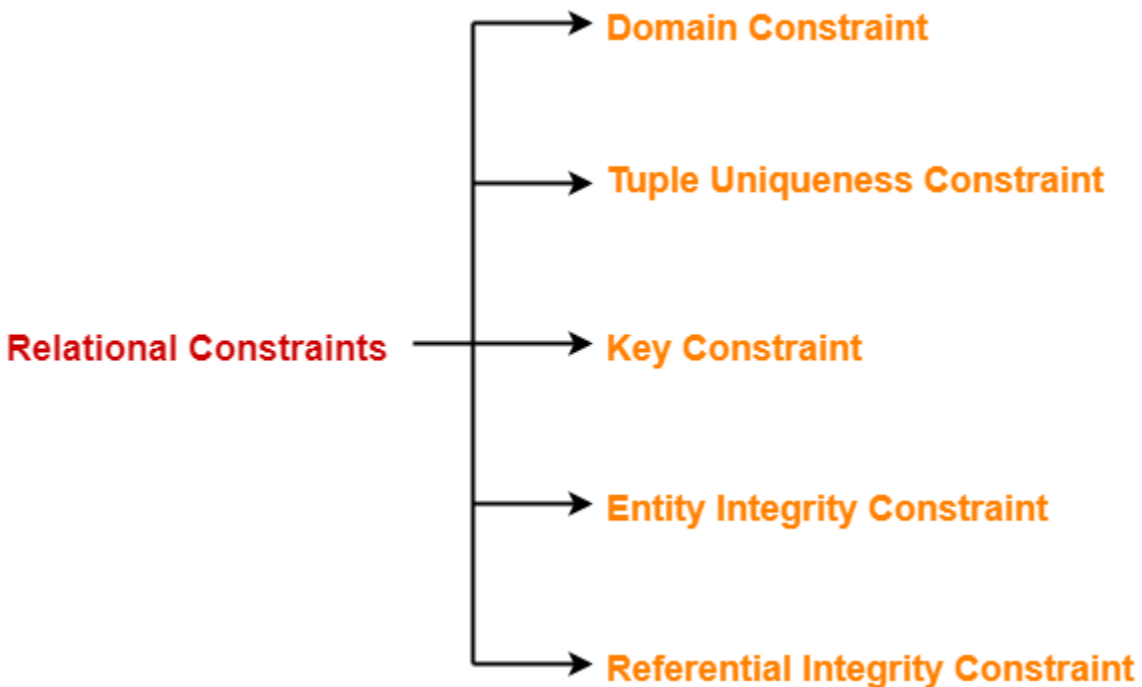


Constraints in DBMS-

- CONSTRAINT means RESTRICTION.
- Relational constraints are the restrictions imposed on the database contents and operations.
- They ensure the correctness of data in the database.

Types of Constraints in DBMS-

In DBMS, there are following 5 different types of relational constraints-



1. Domain constraint
2. Tuple Uniqueness constraint
3. Key constraint
4. Entity Integrity constraint
5. Referential Integrity constraint

1. Domain Constraint-

- Domain constraint defines the domain or set of values for an attribute.
- It specifies that the value taken by the attribute must be the atomic value from its domain.

Example-

Consider the following Student table-

STU_ID	Name	Age
S001	Akshay	20
S002	Abhishek	21
S003	Shashank	20
S004	Rahul	A

Here, value 'A' is not allowed since only integer values can be taken by the age attribute.

2. Tuple Uniqueness Constraint-

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

Tuple Uniqueness constraint specifies that all the tuples must be necessarily unique in any relation.

Example-01:

Consider the following Student table-

<u>STU_ID</u>	Name	Age
S001	Akshay	20
S002	Abhishek	21
S003	Shashank	20
S004	Rahul	20

This relation satisfies the tuple uniqueness constraint since here all the tuples are unique.

Example-02:

Consider the following Student table-

<u>STU_ID</u>	Name	Age
S001	Akshay	20
S001	Akshay	20
S003	Shashank	20
S004	Rahul	20

This relation does not satisfy the tuple uniqueness constraint since here all the tuples are not unique.

3. Key Constraint-

Key constraint specifies that in any relation-

- All the values of primary key must be unique.
- The value of primary key must not be null.

Example-

Consider the following Student table-

<u>STU_ID</u>	Name	Age
S001	Akshay	20
S001	Abhishek	21
S003	Shashank	20
S004	Rahul	20

This relation does not satisfy the key constraint as here all the values of primary key are not unique.

4. Entity Integrity Constraint-

Entity Integrity is the mechanism the system provides to maintain primary keys. The primary key serves as a unique identifier for rows in the table. **Entity Integrity** ensures two properties for primary keys: The primary key for a row is unique; it does not match the primary key of any other row in the table.

- Entity integrity constraint specifies that no attribute of primary key must contain a null value in any relation.
- This is because the presence of null value in the primary key violates the uniqueness property.

Example-

Consider the following Student table-

<u>STU_ID</u>	Name	Age
S001	Akshay	20
S002	Abhishek	21
S003	Shashank	20
	Rahul	20

This relation does not satisfy the entity integrity constraint as here the primary key contains a NULL value.

5. Referential Integrity Constraint-

Whenever two **tables** contain one or more **common columns**, **Oracle** can enforce the relationship between the two **tables** through a **referential integrity constraint**. Define a PRIMARY or UNIQUE key **constraint** on the column in the parent **table** (the one that has the complete set of column values).

- This constraint is enforced when a foreign key references the primary key of a relation.
- It specifies that all the values taken by the foreign key must either be available in the relation of the primary key or be null.

Important Results-

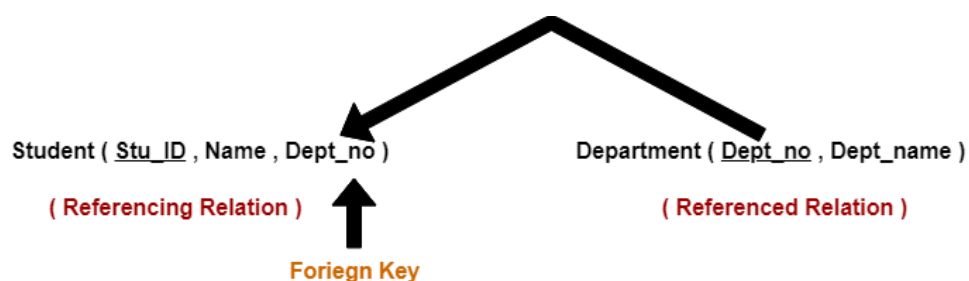
The following two important results emerges out due to referential integrity constraint-

- We can not insert a record into a referencing relation if the corresponding record does not exist in the referenced relation.
- We can not delete or update a record of the referenced relation if the corresponding record exists in the referencing relation.

Example-

Consider the following two relations- 'Student' and 'Department'.

Here, relation 'Student' references the relation 'Department'.



Student

<u>STU_ID</u>	Name	Dept_no
S001	Akshay	D10
S002	Abhishek	D10
S003	Shashank	D11
S004	Rahul	D14

Department

<u>Dept_no</u>	Dept_name
D10	ASET
D11	ALS
D12	ASFL
D13	ASHS

Here,

- The relation 'Student' does not satisfy the referential integrity constraint.
- This is because in relation 'Department', no value of primary key specifies department no. 14.
- Thus, referential integrity constraint is violated.

Referential integrity constraints **example-----2**

Referential integrity constraints is base on the concept of Foreign Keys. A foreign key is an important attribute of a relation which should be referred to in other relationships. Referential integrity constraint state happens where relation refers to a key attribute of a different or same relation. However, that key element must exist in the table.

Example:

Customer		
CustomerID	CustomerName	Status
1	Google	Active
2	Amazon	Active
3	Apple	Inactive

Billing		
InvoiceNo	CustomerID	Amount
1	1	\$100
2	1	\$200
3	2	\$150

In the above example, we have 2 relations, Customer and Billing.

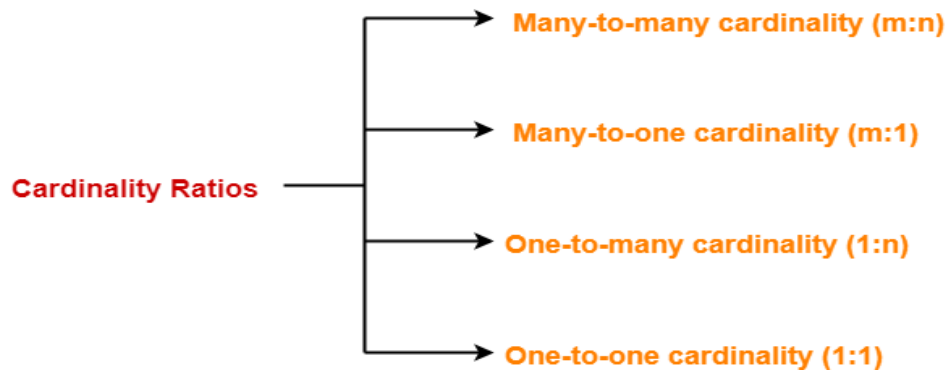
Tuple for CustomerID =1 is referenced twice in the relation Billing. So we know CustomerName=Google has billing amount \$300

Cardinality Constraint-

Cardinality constraint defines the maximum number of relationship instances in which an entity can participate.

Types of Cardinality Ratios-

There are 4 types of cardinality ratios-



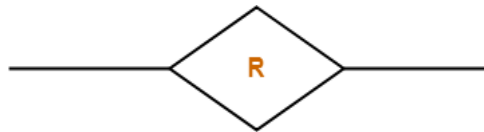
1. Many-to-Many cardinality (m:n)
2. Many-to-One cardinality (m:1)
3. One-to-Many cardinality (1:n)
4. One-to-One cardinality (1:1)

1. Many-to-Many Cardinality-

By this cardinality constraint,

- An entity in set A can be associated with any number (zero or more) of entities in set B.
- An entity in set B can be associated with any number (zero or more) of entities in set A.

Symbol Used-



Cardinality Ratio = m : n

Example-

Consider the following ER diagram-



Many to Many Relationship

Here,

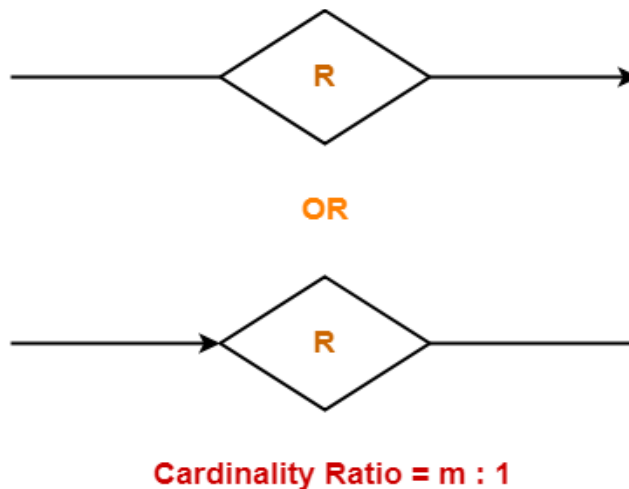
- One student can enroll in any number (zero or more) of courses.
- One course can be enrolled by any number (zero or more) of students.

2. Many-to-One Cardinality-

By this cardinality constraint,

- An entity in set A can be associated with at most one entity in set B.
- An entity in set B can be associated with any number (zero or more) of entities in set A.

Symbol Used-



Example-

Consider the following ER diagram-



Here,

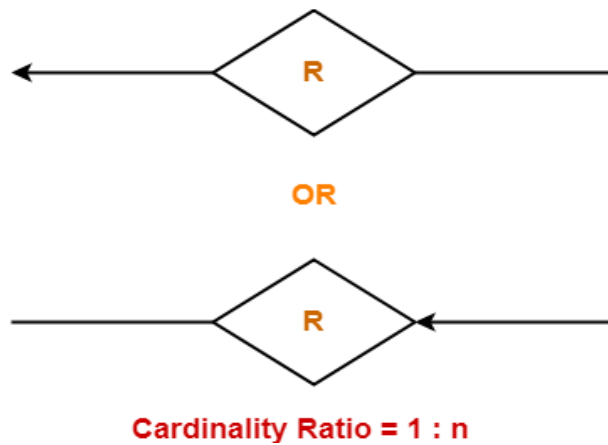
- One student can enroll in at most one course.
- One course can be enrolled by any number (zero or more) of students.S

3. One-to-Many Cardinality-

By this cardinality constraint,

- An entity in set A can be associated with any number (zero or more) of entities in set B.
- An entity in set B can be associated with at most one entity in set A.

Symbol Used-



Example-

Consider the following ER diagram-



Here,

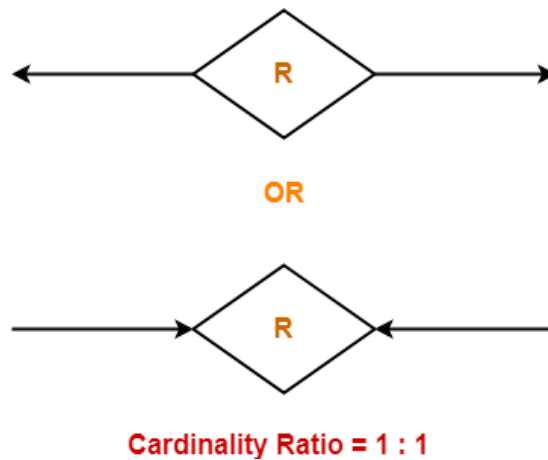
- One student can enroll in any number (zero or more) of courses.
- One course can be enrolled by at most one student.

4. One-to-One Cardinality-

By this cardinality constraint,

- An entity in set A can be associated with at most one entity in set B.
- An entity in set B can be associated with at most one entity in set A.

Symbol Used-



Example-

Consider the following ER diagram-



Here,

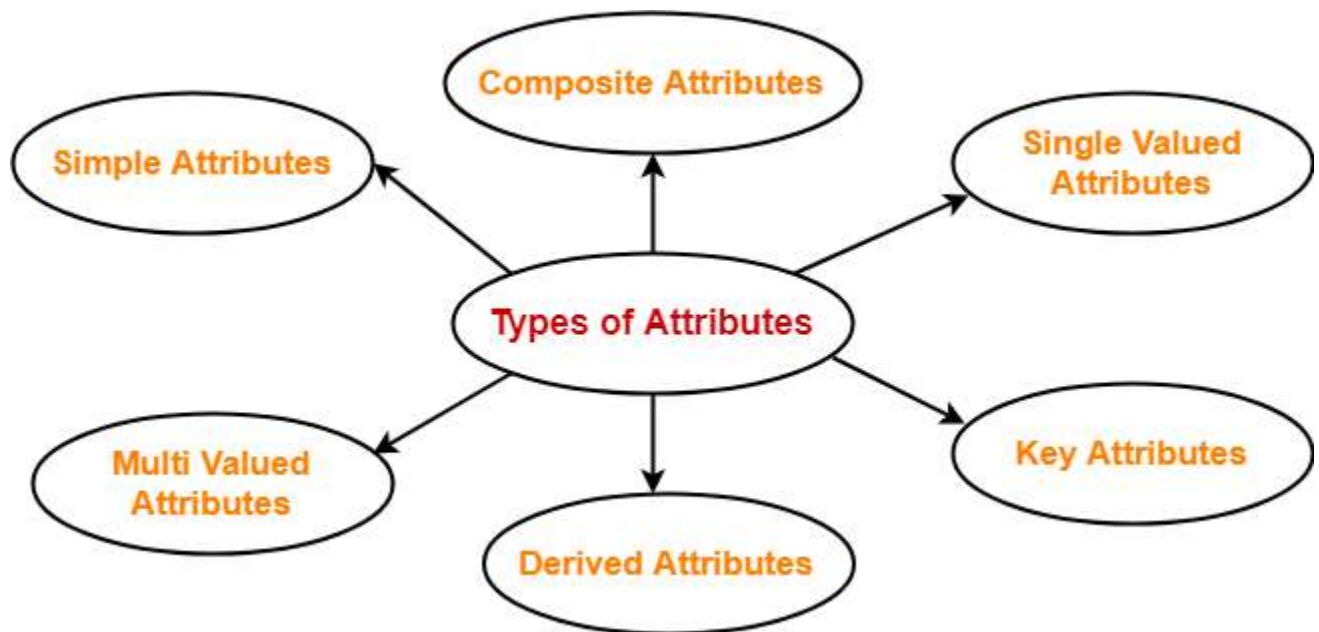
- One student can enroll in at most one course.
- One course can be enrolled by at most one student.

Attributes in ER Diagram-

- Attributes are the descriptive properties which are owned by each entity of an **Entity Set**.
- There exist a specific domain or set of values for each attribute from where the attribute can take its values.

Types of Attributes-

In ER diagram, attributes associated with an entity set may be of the following types-

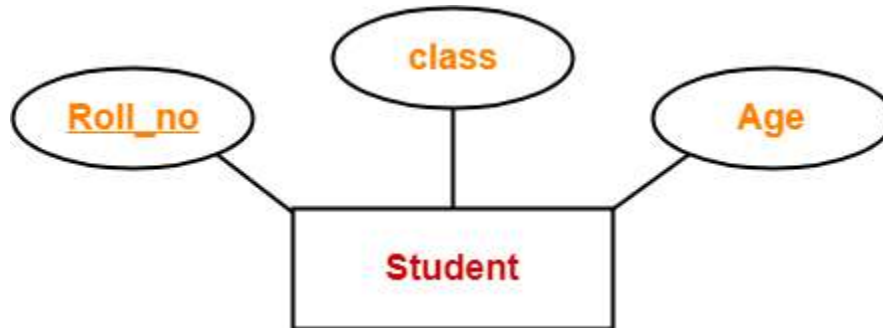


1. Simple attributes
2. Composite attributes
3. Single valued attributes
4. Multi valued attributes
5. Derived attributes
6. Key attribute

1. Simple Attributes-

Simple attributes are those attributes which cannot be divided further.

Example-

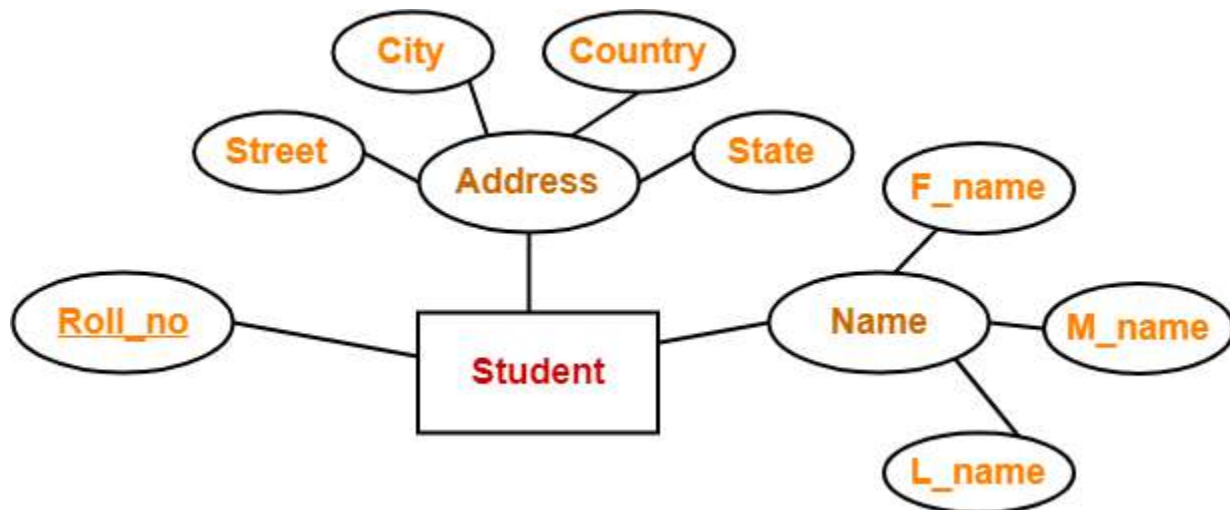


Here, all the attributes are simple attributes as they can not be divided further.

2. Composite Attributes-

Composite attributes are those attributes which are composed of many other simple attributes.

Example-

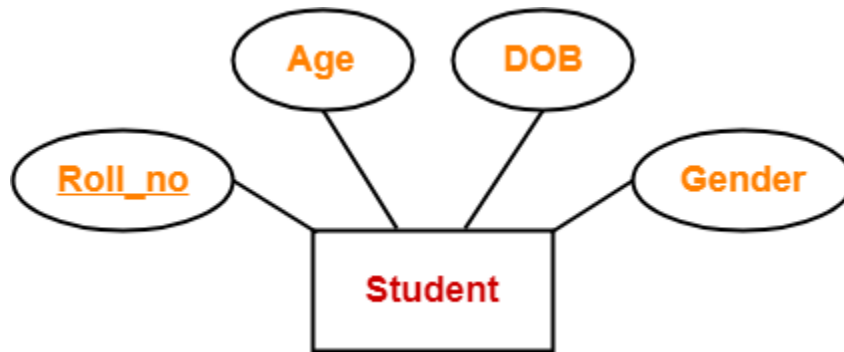


Here, the attributes "Name" and "Address" are composite attributes as they are composed of many other simple attributes.

3. Single Valued Attributes-

Single valued attributes are those attributes which can take only one value for a given entity from an entity set.

Example-

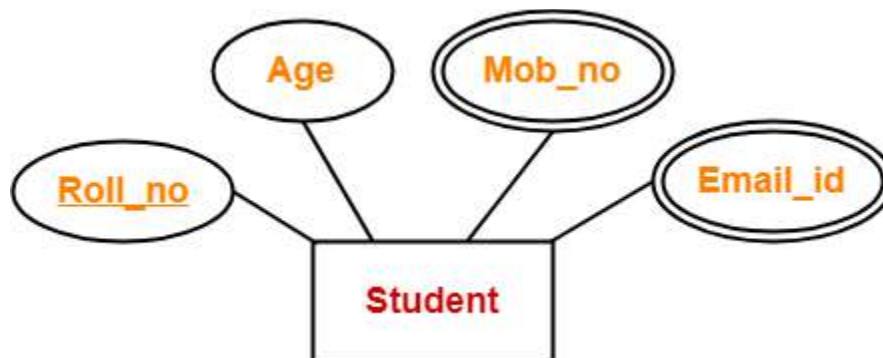


Here, all the attributes are single valued attributes as they can take only one specific value for each entity.

4. Multi Valued Attributes-

Multi valued attributes are those attributes which can take more than one value for a given entity from an entity set.

Example-

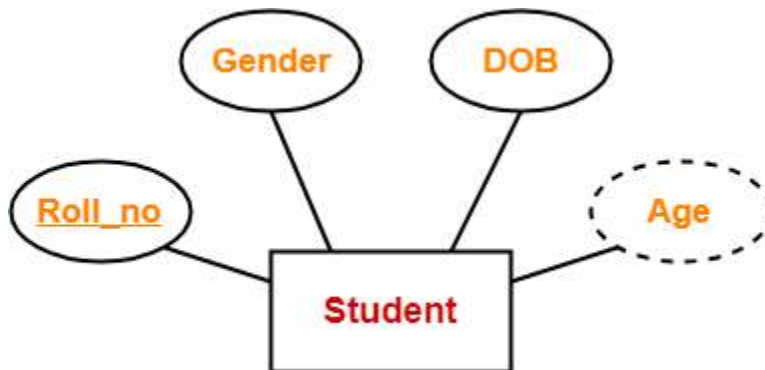


Here, the attributes "Mob_no" and "Email_id" are multi valued attributes as they can take more than one values for a given entity.

5. Derived Attributes-

Derived attributes are those attributes which can be derived from other attribute(s).

Example-

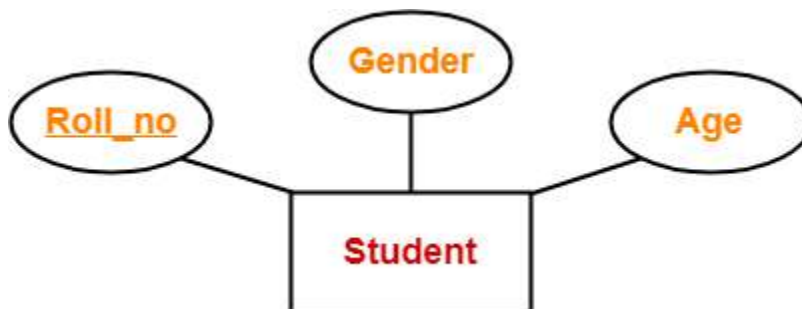


Here, the attribute “Age” is a derived attribute as it can be derived from the attribute “DOB”.

6. Key Attributes-

Key attributes are those attributes which can identify an entity uniquely in an entity set.

Example-

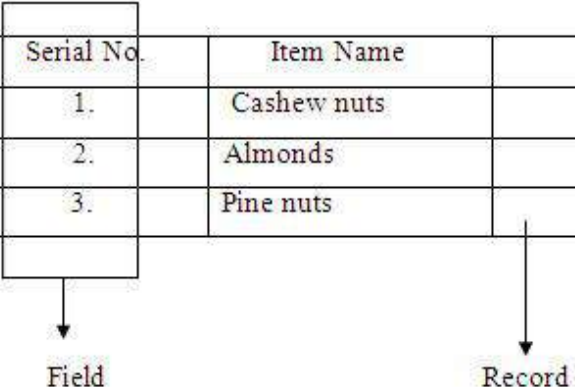


Here, the attribute “Roll_no” is a key attribute as it can identify any student uniquely.

Various Keys in Database Management System (DBMS)

Various Keys in Database Management System (DBMS): For the clarity in DBMS, keys are preferred and they are important part of the arrangement of a table. Keys make sure to uniquely identify a table's each part or record of a field or combination of fields. A [database](#) is made up of tables, which (tables) are made up of records, which (records) further made up of fields. Let us take an example to illustrate what are keys in [database management system](#). This article is about different keys in database management system (DBMS).

	Serial No.	Item Name	Quantity	Price
	1.	Cashew nuts	1 Kg	800
	2.	Almonds	1Kg	900
	3.	Pine nuts	1 Kg	1000



In the above data item, each column is a field and each row is a record.

Types of Keys in Database Management System: Each key which has the parameter of uniqueness is as follows:

1. Super key
2. Candidate key
3. Primary key
4. Composite key
5. Secondary or Alternative key
6. Non- key attribute
7. Non- prime attribute
8. Foreign key

- 9. Simple key
- 10. Compound key
- 11. Artificial key

The detailed explanation of all the mentioned keys is as follows:

1. Super key

Super Key is a set of properties within a table; it specially identifies each record in a table. Candidate key is a unique case of super key.

- For example: Roll No. of a student is unique in relation. The set of properties like roll no., name, class, age, sex, is a super key for the relation student.

REGD.NO	NAME	CLASS	AGE	SEX
1	RAM	II/IV	19	MALE
2	LAKSHMAN	II/IV	18	MALE
3	KRISHNA	II/IV	20	MALE
4	SATYA	II/IV	19	FEMALE
5	SIVA	II/IV	19	MALE

Super keys: The above table has following super keys. All of the following sets of super key are able to uniquely identify a row of the employee table.

- {REGD.NO}
- {REGD.NO, NAME}
- {REGD.NO, CLASS}
- {REGD.NO, CLASS, AGE}
- {REGD.NO, CLASS, AGE, SEX}...ETC

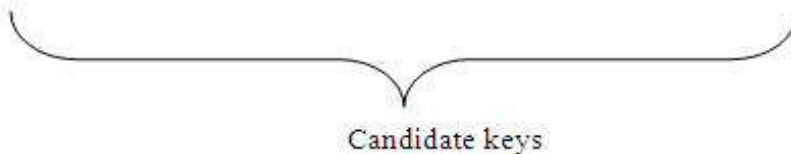
2. Candidate keys

A candidate key is a minimal super key with no redundant (NEEDLESS) attributes.

Candidate keys are the set of fields; primary key can be selected from these fields. A set of properties or attributes acts as a primary key for a table. Every table must have at least one candidate key or several candidate keys. It is a super key's subset.

Example:

Student Id	First name of student	Last name of student	Course Id
123456	Jasmine	Shaik	001
123457	Rose	Mary	002
123458	Lily	Holmes	003



- The above fields of a candidate key uniquely identify a student.
- It has the properties like - Being unique and Parameter of irreducibility.

3. Primary key

The candidate key which is very suitable to be the main key of table is a primary key.

- The primary keys are compulsory in every table.
- The properties of a primary key are:
 - Model stability
 - Occurrence of minimum fields
 - Defining value for every record i.e. being definitive
 - Feature of accessibility

- **Example**

Student Id	First name of student	Last name of student	Course Id
123456	Jasmine	Shaik	001
123457	Rose	Mary	002
123458	Lily	Holmes	003

↓
Primary key

4. Composite key

Key that consists of two or more attributes that uniquely identify any record in a table is called **Composite key**.

A composite key in DBMS is a candidate key that is composed of two or more attributes and is capable of uniquely identifying a table or a relation.

A composite key is the DBMS key having two or more attributes that together can uniquely identify a tuple in a table. Such a key is also known as **Compound Key**.

Example:

Two or more attributes together form a composite key that can uniquely identify a tuple in a table. We need to find out such table columns combination that can form a candidate key and hence a composite key.

Below is an example to understand the working of a composite key.

Consider table A having three columns or attributes, which are as follows:

Cust_Id: A customer id is provided to each customer who visits and is stored in this field.

Order_Id: Each order placed by the customer is given an order id, which is stored in this field.

Prod_code: It holds the code value for the products available.

Prod_name: An attribute holding the name of the product on the specified product code.

Below is the table that shows the diagram for the above table:

The diagram shows a table with four columns: Cust_Id, Order_Id, Prod_code, and Prod_name. The first three columns are circled in red. A line labeled 'A' points to the 'Cust_Id' and 'Prod_code' columns, with the text 'Composite Key' above it.

Cust_Id	Order_Id	Prod_code	Prod_name
001	121	P 12	P
003	123	P 10	Q
005	125	P 3	R

From the table, we found that no attribute is available that alone can identify a record in the table and can become a **primary key**. However, the combination of some attributes can form a key and can identify a tuple in the table. In the above example, **Cust_Id** and **Prod_code** can together form a primary key because they alone are not able to identify a tuple, but together they can do so.

5.Secondary or Alternative key

The rejected candidate keys as primary keys are called as secondary or alternative keys.

The candidate key which are not selected as primary key are known as secondary keys or alternative keys.

Example:

Student Id	First name of student	Last name of student	Course Id
123456	Jasmine	<u>Shaik</u>	001
123457	Rose	Mary	002
123458	Lily	Holmes	003

Secondary or Alternative keys

6.Non-key Attribute

The attributes excluding the candidate keys are called as non-key attributes.

Example:

Student Id	First name of student	Last name of student	Course Id
123456	Jasmine	<u>Shaik</u>	001
123457	Rose	Mary	002
123458	Lily	Holmes	003

Non-key attribute

7. Non-prime Attribute

Excluding primary attributes in a table are non-prime attributes.

- Example:

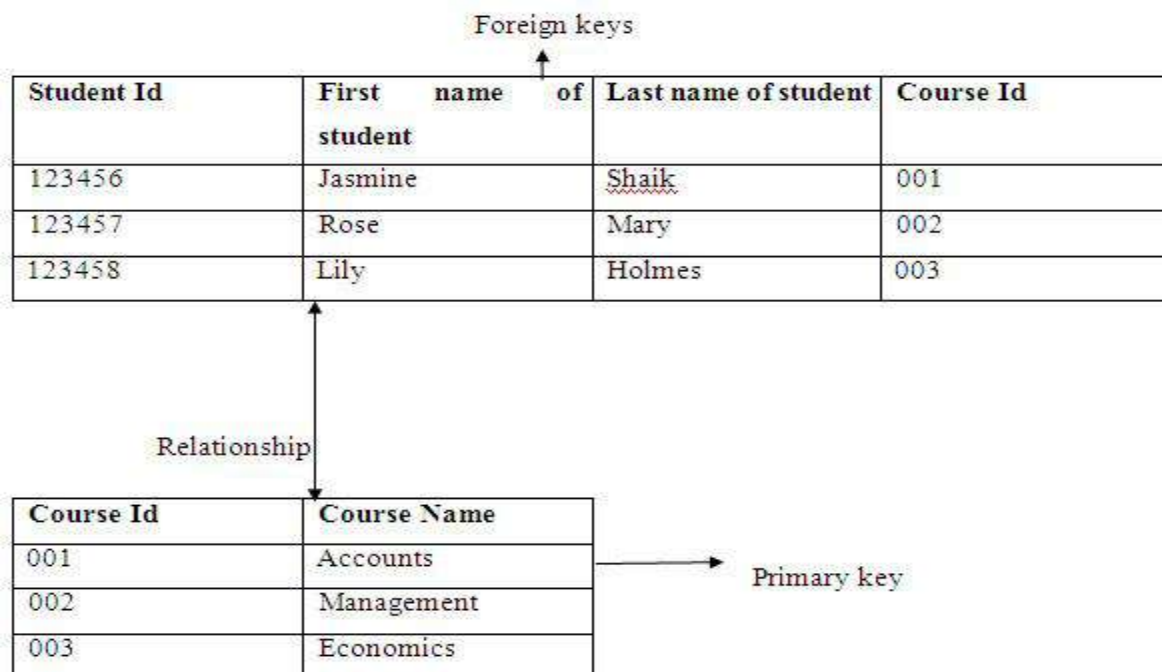
Student Id	First name of student	Last name of student	Course Id
123456	Jasmine	<u>Shaik</u>	001
123457	Rose	Mary	002
123458	Lily	Holmes	003

Non prime attributes

8. Foreign key

Generally foreign key is a primary key from one table, which has a relationship with another table.

- Example:



9. Simple key

Simple keys have a single field to specially recognize a record. The single field cannot be divided into more fields. Primary key is also a simple key.

- Example: In the below example studentId is a single field because no other student will have same Id. Therefore, it is a simple key.

Student Id	First name of student	Last name of student	Course Id
123456	Jasmine	Shaik	001
123457	Rose	Mary	002
123458	Lily	Holmes	003



Simple key

10. Compound key

Compound key has many fields to uniquely recognize a record. Compound key is different from composite key because any part of this key can be foreign key but in composite key its part may or may not be a foreign key.

11. Surrogate/Artificial key

Surrogate key is artificially generated key and its main purpose is to be the primary key of table. Artificial keys do not have meaning to the table. There are few properties of surrogate or artificial keys.

They are unique because these just created when you don't have any natural primary key.

They are integer values.

One cannot find the meaning of surrogate keys in the table.

End users cannot surrogate key.

Surrogate keys are allowed when

- No property has the parameter of primary key.
- The primary key is huge and complex.

Example: Table which has the details of the student has primary key but it is large and complex. The addition of row id column to it is the DBA's decision, where the primary key is row id.

12. Natural /Domain/ Business Key:

It is a key that is naturally declared as the Primary key. Natural keys are sometimes called as business or domain keys because these key are based on the real world observation. So it is a key whose attributes or values exists in the real world. These attributes have logical relationship with the table.

For Example: Social Security Number (SSN) is a natural key that can be declared as the primary key

What is Table ?

In Relational database model, a table is a collection of data elements organised in terms of rows and columns. A table is also considered as a convenient representation of relations. But a table can have duplicate row of data while a true relation cannot have duplicate data. Table is the most simplest form of data storage. Below is an example of an Employee table.

ID	Name	Age	Salary
1	Adam	34	13000
2	Alex	28	15000
3	Stuart	20	18000
4	Ross	42	19020

What is a tuple in the database?

A tuple is simply a row contained in a table in the tablespace. A table usually contains columns and rows in which rows stand for records while columns stand for attributes. A single row of a table that has a single record for such a relation is known as a tuple. A Tuple is, therefore, a single entry in a table; it is also called a row or record. They usually represent a set of related data, in Maths it is simply an ordered list of elements.

A single entry in a table is called a Tuple or Record or Row. A tuple in a table represents a set of related data. For example, the above Employee table has 4 tuples/records/rows.

Following is an example of single record or tuple.

1	Adam	34	13000
---	------	----	-------

What is an Attribute?

A table consists of several records (row), each record can be broken down into several smaller parts of data known as **Attributes**. The above **Employee** table consist of four attributes, **ID**, **Name**, **Age** and **Salary**.

Attribute Domain

When an attribute is defined in a relation (table), **it is defined to hold only a certain type of values**, which is known as **Attribute Domain**.

Hence, the attribute **Name** will hold the name of employee for every tuple. If we save employee's address there, it will be violation of the Relational database model.

Name
Adam
Alex
Stuart - 9/401, OC Street, Amsterdam
Ross

What is a Relation Schema?

A relation schema describes the structure of the relation, with the name of the relation (name of table), its attributes and their names and type.

What is a Relation Key?

A relation key is an attribute which can **uniquely** identify a particular tuple (row) in a relation (table).

Relational Integrity Constraints

Every relation in a relational database model should abide by or follow a few constraints to be a valid relation, these constraints are called as **Relational Integrity Constraints**.

The three main Integrity Constraints are:

1. Key Constraints
2. Domain Constraints
3. Referential integrity Constraints

Key Constraints

We store data in tables, to later access it whenever required. In every table one or more than one attributes together are used to fetch data from tables. **The Key Constraint specifies that there should be such an attribute(column) in a relation(table),** which can be used to fetch data for any tuple(row).

The Key attribute should never be **NULL** or same for two different row of data.

For example, in the **Employee** table we can use the attribute **ID** to fetch data for each of the employee. No value of **ID** is null and it is unique for every row, hence it can be our **Key attribute**.

Domain Constraint

Domain constraints refers to the rules defined for the values that can be stored for a certain attribute.

Like we explained above, we cannot store **Address** of employee in the column for **Name**.

Similarly, a mobile number cannot exceed 10 digits.

Referential Integrity Constraint

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

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

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

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

STUDENT TABLE

ROLL	NAME	COURSE
1	John	MCA
2	Smith	MTech
3	Shane	BTech
4	Ricky	MBA

The syntax of Child Table or Referencing table is:

```
CREATE TABLE Subject (Roll int references Student, SubCode int, SubName varchar(10) );
```

SUBJECT TABLE

ROLL	SubCode	SubName
1	001	DBMS
2	005	SQL
3	006	DS
4	070	OB

In the above table, column Roll is acting as **Foreign Key**, whose values are derived using the Roll value of Primary key from Master table.

Table is a collection of data, organized in terms of rows and columns. In DBMS term, table is known as relation and row as tuple.

A table has a **specified number of columns**, but can have **any number of rows**.

Table is the simple form of data storage. A table is also considered as a convenient representation of relations

SQL CREATE TABLE Statement

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

Syntax

```
CREATE TABLE table_name (  
    column1 datatype,  
    column2 datatype,  
    column3 datatype,  
    ....  
);
```

The column parameters specify the names of the columns of the table.

The datatype parameter specifies the type of data the column can hold (e.g. varchar, integer, date, etc.).

The following **example creates a table called "Persons"** that contains five columns: PersonID, LastName, FirstName, Address, and City:

```
SQL> CREATE TABLE Persons (  
    PersonID int,  
    LastName varchar(20),  
    FirstName varchar(2),  
    Address varchar(2),  
    City varchar(20)  
);
```

Table created.

The PersonID column is of type int and will hold an integer.

The LastName, FirstName, Address, and City columns are of type varchar and will hold characters, and the maximum length for these fields is 255 characters.

Create Table Using Another Table

A copy of an existing table can also be created using **CREATE TABLE**.

The new table gets the same column definitions. All columns or specific columns can be selected.

If you create a new table using an existing table, the new table will be filled with the existing values from the old table.

Syntax

```
CREATE TABLE new_table_name AS
SELECT column1,column2,...
FROM existing_table_name
WHERE ....;
```

The following SQL creates a new table called "TestTables" (which is a copy of the "Persons" table):

EXAMPLE:

```
SQL> create table testTable as
      select PersonID,
      LastName,
      FirstName
      from Persons;
```

Table created.

```
SQL> SELECT * FROM TESTTABLE;
```

no rows selected

INSERT THE VALUES INTO TABLE

- SQL> insert into Persons values(1, 'ram', 'krish', 'viz', 'viz');

1 row created.


```
SQL> select * from Persons;
```

PERSONID	LASTNAME	FIRSTNAME	ADDRESS
1	ram	krish	viz

```
SQL> COLUMN ADDRESS FORMAT A10; // SET THE COLUMN WIDTH
```

```
SQL> COLUMN CITY FORMAT A10;
```

```
SQL> /
```

PERSONID	LASTNAME	FIRSTNAME	ADDRESS	CITY
1	ram	krish	viz	viz

Note: After insert the values into Persons table, by using that table (Persons) we are creating new table, then that new table (testable) contains the parent table values by default.

```
SQL> create table testTable as
      select PersonID,
      LastName,
      FirstName
      from Persons;
Table created.
```

```
SQL> SELECT * FROM TESTTABLE;
```

PERSONID	LASTNAME	FIRSTNAME
1	ram	krish

SQL Operators | Arithmetic, Comparison & Logical Operators

What is an Operator?

An operator in SQL is a reserved keyword or a symbol (special character) that operates up on the operands (or a set of operands) to perform various operations to return a result.

What is an Operand?

An operand can be defined as the data item or an argument that is used by an operator to perform different operations like arithmetic, logical, comparison, etc.

SQL Arithmetic Operators

Arithmetic operators operate on numeric operands. These are mainly used for mathematical operations such as addition, subtraction, etc.

There are two types of arithmetic operators as shown below:

1. Unary Arithmetic Operators:

These operators act upon only one operand. The format followed by the Unary operator is 'OPERATOR OPERAND'.

2. Binary Arithmetic Operators:

These operators act upon two operands. The format followed by the Binary operator is 'OPERAND OPERATOR OPERAND'.

Unary Arithmetic Operators

	OPERATOR	OPERATION	EXPLANATION
1	+	UNARY POSITIVE	To make the operand or a value positive.
2	-	UNARY NEGATIVE	To negate the operand or make a value negative.

Binary Arithmetic Operators

OPERATOR		OPERATION	EXPLANATION
3	+	ADDITION	Adds up the operand values specified on either side of the operator.
4	-	SUBTRACTION	Performs subtraction of right hand side value from the left hand side value.
5	*	MULTIPLICATION	Multiplies the operand values present on either side of operator.
6	/	DIVISION	Performs division of left hand side value by right hand side value and returns the quotient.
7	%	MODULUS	Performs division of left hand side value by right hand side value and returns the remainder.

1. Unary Positive (+)

`Select +10 as VALUE from dual;`

Output:

VALUE
10

2. Unary Negative (-)

`Select -20 as VALUE from dual;`

Output:

VALUE
-20

3. Addition (+)

Select 10+20 as VALUE from dual;

Output:

VALUE
30

4. Subtraction (-)

Example 1:

Select 10-20 as VALUE from dual;

Output:

VALUE
-10

Example 2:

Select 20-10 as VALUE from dual;

Output:

VALUE
10

5. Multiplication (*)

Select 10*20 as VALUE from dual;

Output:

VALUE
200

6. Division (/)

Example 1:

Select 10/20 as VALUE from dual;

Output:

VALUE
.5

Example 2:

Select 20/10 as VALUE from dual;

Output:

VALUE
2

7. Modulus (%)

Example 1:

Select 10%20 as VALUE from dual;

Output:

VALUE
10

Example 2:

Select 20%10 as VALUE from dual;

Output:

VALUE
0

SQL Comparison Operators

Comparison operators compare two operand values or can also be used in conditions where one expression is compared to another that often returns a result (Result can be true or false).

OPERATOR	OPERATION	EXPLANATION
=	Equality test	Performs the test on two operand values to check if they are equal. If they are equal, condition (in the where clause) becomes true and returns the corresponding rows.
<> or != or ^=	Inequality test	Performs the test on two operand values to check if they are not equal. If they are not equal, returns true else returns false.
>	Greater than test	Tests if the left hand side operand value is greater than that of operand value on the right hand side. If yes, returns true. Another case is if you use it in where clause of a select query, then it returns all those rows having column value greater than the specified value.
<	Less than test	Tests if the left hand side operand value is less than that of operand value on the right hand side. If yes, returns true. Another case where you can use it in where clause of a select query. In such a case, it returns all those rows having column values less than the specified value.
<=	Less than or equal to test	Evaluates if the left hand side operand value is less than or equal to the operand value on the right hand side. If yes, returns true.
>=	Greater than or equal to test	Evaluates if the left hand side operand value is greater than or equal to the operand value on the right hand side. If yes, returns true.

Consider the table below which is being used as a reference for the next examples.

```
SQL> CREATE TABLE FACULTY (FID INT NOT NULL, FNAME VARCHAR(10) NOT NULL, MOBILE_NO INT, JOBID VARCHAR(5), SALARY INT);
```

Table created.

Insert values into the table:

```
SQL> INSERT INTO FACULTY VALUES (1, 'RAM', 9988776655, 'JL',10000);
```

1 row created.

```
SQL> INSERT INTO FACULTY VALUES (2, 'LAKSH', 9977665544, 'PROF', 50000);
```

1 row created.

```
SQL> INSERT INTO FACULTY VALUES (3, 'KRISH', 9977664422, 'ASO', 40000);
```

1 row created.

```
SQL> INSERT INTO FACULTY VALUES (4, 'SIVA', 9966778855, 'ASIS', 30000);
```

1 row created.

Sample Table - FACULTY_DETAILS

```
SQL> SELECT * FROM FACULTY;
```

	FID	FNAME	MOBILE_NO	JOBID	SALARY
1	RAM	9988776655	JL	10000	
2	LAKSH	9977665544	PROF	50000	
3	KRISH	9977664422	ASO	40000	
4	SIVA	9966778855	ASIS	30000	

1. Equal to (=)

Example 1: [Scenario of retrieving rows with the value having NUMBER datatype]

```
SELECT * FROM FACULTY WHERE FID = 3;
```

```
SQL> SELECT * FROM FACULTY WHERE FID=3;
```

FID	FNAME	MOBILE_NO	JOBID	SALARY
3	KRISH	9977664422	ASO	40000

Example 2: [Scenario of retrieving rows with the value having VARCHAR2 datatype]

```
SELECT * FROM FACULTY WHERE FNAME = 'SIVA';
```

```
SQL> SELECT * FROM FACULTY WHERE FNAME='SIVA';
```

FID	FNAME	MOBILE_NO	JOBID	SALARY
4	SIVA	9966778855	ASIS	30000

2. Not Equal to (<> or != or ^=)

```
SELECT * FROM FACULTY WHERE SALARY != 30000;
```

OR

```
SQL> SELECT * FROM FACULTY WHERE SALARY<>30000;
```

OR

```
SQL> SELECT * FROM FACULTY WHERE SALARY ^= 30000;
```

FID	FNAME	MOBILE_NO	JOBID	SALARY
1	RAM	9988776655	JL	10000
2	LAKSH	9977665544	PROF	50000
3	KRISH	9977664422	ASO	40000

3. Greater than (>)

```
SELECT * FROM FACULTY WHERE SALARY > 20000;
```

```
SQL> SELECT * FROM FACULTY WHERE SALARY > 20000;
```

FID	FNAME	MOBILE_NO	JOBID	SALARY
2	LAKSH	9977665544	PROF	50000
3	KRISH	9977664422	ASO	40000
4	SIVA	9966778855	ASIS	30000

4. Less than (<)

```
SELECT * FROM FACULTY WHERE SALARY < 20000;
```

```
SQL> SELECT * FROM FACULTY WHERE SALARY < 20000;
```

FID	FNAME	MOBILE_NO	JOBID	SALARY
1	RAM	9988776655	JL	10000

5. Less than or equal to (<=)-

```
SELECT * FROM FACULTY WHERE SALARY <= 30000;
```

```
SQL> SELECT * FROM FACULTY WHERE SALARY <= 30000;
```

FID	FNAME	MOBILE_NO	JOBID	SALARY
1	RAM	9988776655	JL	10000
4	SIVA	9966778855	ASIS	30000

6. Greater than or equal to (>=)

```
SELECT * FROM FACULTY WHERE SALARY >= 30000;
```

```
SQL> SELECT * FROM FACULTY WHERE SALARY >= 30000;
```

FID	FNAME	MOBILE_NO	JOBID	SALARY
2	LAKSH	9977665544	PROF	50000
3	KRISH	9977664422	ASO	40000
4	SIVA	9966778855	ASIS	30000

SQL Logical Operators

Logical operators are the special type of operators that helps us to retrieve data from the database with even more specificity.

OPERATOR	EXPLANATION
ALL	ALL is used to compare a value with all other values in the set. This set of values can either be a list or the result of the subquery. 'ALL' has to be preceded with any of the comparison operators always. Suppose, a query returns no rows then it evaluates to be TRUE.
AND	AND permits the occurrence of multiple conditions in a SQL query. Only those rows are returned that satisfy all the conditions mentioned in a SQL statement.
ANY	ANY is used to compare a value with any of the values specified either in a list or in the result of the subquery. 'ANY' has to be preceded with any of the comparison operators always. Suppose, a query returns no rows then it evaluates to be FALSE.
BETWEEN 'x' and 'y'	It returns the rows which fall into a specified range where x is the minimum value and y is the maximum value (Both maximum and minimum values are included). It is equivalent to saying the range is greater than or equal to x and less than or equal to y.
NOT BETWEEN 'x' and 'y'	It returns the rows which do not fall into a specified range where x is the minimum value and y is the maximum value. It is equivalent to saying the range is not greater than or equal to x and not less than or equal to y.

IN	IN is equivalent to '= ANY '. It compares a particular value to another set of values mentioned in list separated by commas or a subquery that returns the set of values. If matches are found, then it returns all such rows.
NOT IN	IN is equivalent to ' !=ANY '. It compares a particular value to another set of values mentioned in a list separated by commas or a subquery that returns the set of values. If matches are found, then it returns all other rows except the ones for which match has been found.
LIKE	LIKE operator uses the wildcard notations '%' and '_' to search for specific values in the column. '%' represents zero, one or multiple characters and '_' represents a single character. Note that '*' represents zero, one, or multiple characters where as '?' represents a single character in MS Access.
OR	OR permits the occurrence of multiple conditions in a SQL query. All those rows are returned which satisfy any of the conditions mentioned in a SQL statement.
SOME	SOME is used to compare a value with any of the values specified either in a list or in the result of the subquery. 'SOME' has to be preceded with any of the comparison operators always. This operator functions similar to ANY.
IS NULL	This operator is used when there is a need to deal with NULL values. We can specify this as a condition in where clause of the SQL query which returns the rows (if present) having column value equal to NULL.

IS NOT NULL	It can be specified in a where clause of a SQL query which then returns the rows that do not have nulls.
EXISTS	It compares a specified column value with that of column values present in another table or a subquery result along with the other mentioned criteria. If matches are found, it returns all such rows.
NOT EXISTS	It compares a specified column value with that of column values present in another table or a subquery result along with the other mentioned criteria. It returns all other rows except the ones that do exist as a result of the subquery.

Consider the tables “FACULTY” and “JOBS” below that is being used as a reference for the next example.

Sample Table-1 – FACULTY (TABLE CREATION):

```
SQL> CREATE TABLE FACULTY (FID INT NOT NULL, FNAME VARCHAR(10) NOT NULL, MOBILE_NO INT, JOBID VARCHAR(5), SALARY INT);
```

Table created.

Insert values into the table:

```
SQL> INSERT INTO FACULTY VALUES (1, 'RAM', 9988776655, 'JL',10000);
1 row created.
```

```
SQL> INSERT INTO FACULTY VALUES (2, 'LAKSH', 9977665544, 'PROF', 50000);
1 row created.
```

```
SQL> INSERT INTO FACULTY VALUES (3, 'KRISH', 9977664422, 'ASO', 40000);
1 row created.
```

```
SQL> INSERT INTO FACULTY VALUES (4, 'SIVA', 9966778855, 'ASIS', 30000);  
1 row created.
```

Sample Table-1 - FACULTY

```
SQL> SELECT * FROM FACULTY;
```

FID	FNAME	MOBILE_NO	JOBID	SALARY
1	RAM	9988776655	JL	10000
2	LAKSH	9977665544	PROF	50000
3	KRISH	9977664422	ASO	40000
4	SIVA	9966778855	ASIS	30000

Sample Table-2 - JOBS (TABLE CREATION):

```
SQL> CREATE TABLE JOBS(JOBID VARCHAR(5) NOT NULL, DESIGNATION  
VARCHAR(10) NOT NULL, TYPE_OF_JOB VARCHAR(10) NOT NULL, WORKING_HOURS  
INT NOT NULL);
```

Table created.

```
SQL> INSERT INTO JOBS VALUES('JL', 'JUNIOR LECTURER', 'PART TIME', 4);  
INSERT INTO JOBS VALUES('JL', 'JUNIOR LECTURER', 'PART TIME', 4)
```

*

ERROR at line 1:

ORA-12899: value too large for column "DSR"."JOBS"."DESIGNATION"
(actual: 15,
maximum: 10)

NOTE: TO ALTER THE TABLE SCHEMA

```
SQL> ALTER TABLE JOBS MODIFY ( DESIGNATION VARCHAR(15));
```

Table altered.

INSERT VALUES INTO THE TABLE:

```
SQL> INSERT INTO JOBS VALUES('JL', 'JUNIOR LECTURER', 'PART TIME', 4);
```

1 row created.

```
SQL> INSERT INTO JOBS VALUES('PROF', 'PROFESSOR', 'FULL TIME', 8);
```

1 row created.

```
SQL> INSERT INTO JOBS VALUES('ASO', 'ASSOCIATE', 'FULL TIME', 7);
```

1 row created.

```
SQL> INSERT INTO JOBS VALUES('ASSI', 'ASSISTANT', 'FULL TIME', 7);
```

1 row created.

- *Sample Table-2 - JOBS IS:*

```
SQL> SELECT * FROM JOBS;
```

JOBID	DESIGNATION	TYPE_OF_J0	WORKING_HOURS
JL	JUNIOR LECTURER	PART TIME	4
PROF	PROFESSOR	FULL TIME	8
ASO	ASSOCIATE	FULL TIME	7
ASSI	ASSISTANT	FULL TIME	7

1. ALL

Example 1:

```
SELECT * FROM FACULTY WHERE SALARY > ALL (SELECT SALARY FROM FACULTY W  
HERE FID = 4);
```

Referring to the above statement, we know that the subquery returns one record with SALARY = 30000. So, the main query returns all such records whose SALARY is greater than 30000.

```
SQL> SELECT * FROM FACULTY WHERE SALARY > ALL (SELECT SALARY FROM
FACULTY WHERE FID = 4);
```

FID	FNAME	MOBILE_NO	JOBID	SALARY
2	LAKSH	9977665544	PROF	50000
3	KRISH	9977664422	ASO	40000

SPLITTING OF ABOVE NESTED QUERY:

```
SQL> SELECT SALARY FROM FACULTY WHERE FID = 4;
```

```

SALARY
-----
30000
```

```
SQL> SELECT * FROM FACULTY WHERE SALARY > 30000;
```

FID	FNAME	MOBILE_NO	JOBID	SALARY
2	LAKSH	9977665544	PROF	50000
3	KRISH	9977664422	ASO	40000

Example 2:

```
SELECT * FROM FACULTY WHERE SALARY > ALL (SELECT SALARY FROM FACULTY W
HERE FID < 2);
```

This time the subquery returns ONE record (with SALARY = 10000). So, the main query returns all such records whose SALARY is greater than all of those returned from the subquery.


```
SQL> SELECT * FROM FACULTY WHERE SALARY > ALL (SELECT SALARY FROM
FACULTY WHERE FID < 2);
```

FID	FNAME	MOBILE_NO	JOBID	SALARY
2	LAKSH	9977665544	PROF	50000
3	KRISH	9977664422	ASO	40000
4	SIVA	9966778855	ASIS	30000

```
SQL> SELECT * FROM FACULTY WHERE SALARY > ALL (SELECT SALARY FROM
FACULTY WHERE FID < 3);
```

no rows selected

Example 3:

```
SELECT * FROM FACULTY WHERE SALARY < ALL (SELECT SALARY FROM FACULTY
WHERE FID > 3);
```

In this case, the subquery returns ONE record (with SALARY = 30000). So, the main query returns all such records whose SALARY is less than all of those returned from the subquery.

```
SQL> SELECT * FROM FACULTY WHERE SALARY < ALL (SELECT SALARY FROM
FACULTY WHERE FID > 3);
```

FID	FNAME	MOBILE_NO	JOBID	SALARY
1	RAM	9988776655	JL	10000

2.AND

```
SELECT * FROM FACULTY WHERE FID = 4 AND JOBID = 'ASIS';
```

In the above statement, there are two conditions in the where clause. **AND** Operator makes sure that only those rows are returned that satisfy both the conditions. Likewise, you can specify as many conditions as you want.

```
SQL> SELECT * FROM FACULTY WHERE FID = 4 AND JOBID = 'ASIS';
```

FID	FNAME	MOBILE_NO	JOBID	SALARY
4	SIVA	9966778855	ASIS	30000

```
SQL> SELECT * FROM FACULTY WHERE FID = 4 AND JOBID = 'ASO';
```

```
no rows selected
```

```
****
```

3.ANY

Example 1:

Here, the subquery returns **TWO** records (with SALARY = 30000, 40000). So, the main query returns all such records whose SALARY is greater than any one value of those returned from the subquery.

```
SQL> SELECT * FROM FACULTY WHERE SALARY > ANY (SELECT SALARY FROM FACULTY WHERE FID > 2);
```

FID	FNAME	MOBILE_NO	JOBID	SALARY
2	LAKSH	9977665544	PROF	50000
3	KRISH	9977664422	ASO	40000

Example 2:

```
SELECT * FROM FACULTY WHERE SALARY = ANY (SELECT SALARY FROM FACULTY
WHERE FID > 4);
```

In this scenario, the subquery returns three records (with SALARY = 30000, 43000, 60000). So, the main query returns all such records which are returned as a result of the subquery.

```
SQL> SELECT * FROM FACULTY WHERE SALARY = ANY (SELECT SALARY FROM
FACULTY WHERE FID > 4);
```

<i>FID</i>	<i>FNAME</i>	<i>MOBILE_NO</i>	<i>JOBID</i>	<i>SALARY</i>
4	SIVA	9966778855	ASIS	30000
5	SATYA	8899775566	ASIS	30000

SPLITTING OF THE ABOVE QUERY FOR BETTER UNDERSTND

```
SQL> SELECT SALARY FROM FACULTY WHERE FID > 4;
SALARY
```

```
-----
30000
```

```
SQL> SELECT * FROM FACULTY WHERE SALARY = 30000;
```

<i>FID</i>	<i>FNAME</i>	<i>MOBILE_NO</i>	<i>JOBID</i>	<i>SALARY</i>
5	SATYA	8899775566	ASIS	30000
4	SIVA	9966778855	ASIS	30000

Example 3:

```
SQL> SELECT * FROM FACULTY WHERE SALARY < ANY (SELECT SALARY FROM
FACULTY WHERE FID > 3);
```

FID	FNAME	MOBILE_NO	JOBID	SALARY
1	RAM	9988776655	JL	10000

SPLITTING OF THE ABOVE QUERY FOR BETTER UNDERSTND

SQL> SELECT SALARY FROM FACULTY WHERE FID > 3;

SALARY
30000
30000

SQL> SELECT * FROM FACULTY WHERE SALARY < 30000;

FID	FNAME	MOBILE_NO	JOBID	SALARY
1	RAM	9988776655	JL	10000

4.BETWEEN X AND Y

SQL> SELECT * FROM FACULTY WHERE SALARY BETWEEN 20000 AND 50000;

FID	FNAME	MOBILE_NO	JOBID	SALARY
5	SATYA	8899775566	ASIS	30000
2	LAKSH	9977665544	PROF	50000
3	KRISH	9977664422	ASO	40000
4	SIVA	9966778855	ASIS	30000

5. Not Between X AND Y

```
SQL> SELECT * FROM FACULTY WHERE SALARY NOT BETWEEN 20000 AND 50000;
```

FID	FNAME	MOBILE_NO	JOBID	SALARY
1	RAM	9988776655	JL	10000

6. IN

```
SELECT * FROM FACULTY WHERE FID IN (3, 5);
```

Returning the rows with the column values of NUMBER datatype specified in the list.

```
SQL> SELECT * FROM FACULTY WHERE FID IN (3, 5);
```

FID	FNAME	MOBILE_NO	JOBID	SALARY
5	SATYA	8899775566	ASIS	30000
3	KRISH	9977664422	ASO	40000

```
SQL> SELECT * FROM FACULTY WHERE FID IN (3, 4);
```

FID	FNAME	MOBILE_NO	JOBID	SALARY
3	KRISH	9977664422	ASO	40000
4	SIVA	9966778855	ASIS	30000

```
SQL> SELECT * FROM FACULTY WHERE FID IN (2, 4);
```

FID	FNAME	MOBILE_NO	JOBID	SALARY
2	LAKSH	9977665544	PROF	50000
4	SIVA	9966778855	ASIS	30000

Example 2:

```
SELECT * FROM FACULTY WHERE JOBID IN ('ASIS', 'JL');
```

Returning the rows with the column values of VARCHAR2 datatype specified in the list.

```
SQL> SELECT * FROM FACULTY WHERE JOBID IN ('ASIS','JL');
```

FID	FNAME	MOBILE_NO	JOBID	SALARY
5	SATYA	8899775566	ASIS	30000
1	RAM	9988776655	JL	10000
4	SIVA	9966778855	ASIS	30000

```
SQL> SELECT * FROM FACULTY WHERE JOBID IN ('ASIS','PROF');
```

FID	FNAME	MOBILE_NO	JOBID	SALARY
5	SATYA	8899775566	ASIS	30000
2	LAKSH	9977665544	PROF	50000
4	SIVA	9966778855	ASIS	30000

7. NOT IN

```
SELECT * FROM FACULTY WHERE FID NOT IN (3,5)
```

Notice that all those rows are returned except the ones that are specified in the argument list.

```
SQL> SELECT * FROM FACULTY WHERE FID NOT IN (3, 5);
```

FID	FNAME	MOBILE_NO	JOBID	SALARY
1	RAM	9988776655	JL	10000
2	LAKSH	9977665544	PROF	50000
4	SIVA	9966778855	ASIS	30000

8. LIKE

Example 1:

```
SELECT * FROM FACULTY WHERE FNAME LIKE 'RA%';
```

The above query returns the rows that have FNAME starting with 'RA'.

```
SQL> SELECT * FROM FACULTY WHERE FNAME LIKE 'RA%';
```

FID	FNAME	MOBILE_NO	JOBID	SALARY
1	RAM	9988776655	JL	10000

Example 2:

```
SELECT * FROM FACULTY WHERE FNAME LIKE '%H';
```

This query returns the rows that have FNAME ending with 's'.

```
SQL> SELECT * FROM FACULTY WHERE FNAME LIKE '%H';
```

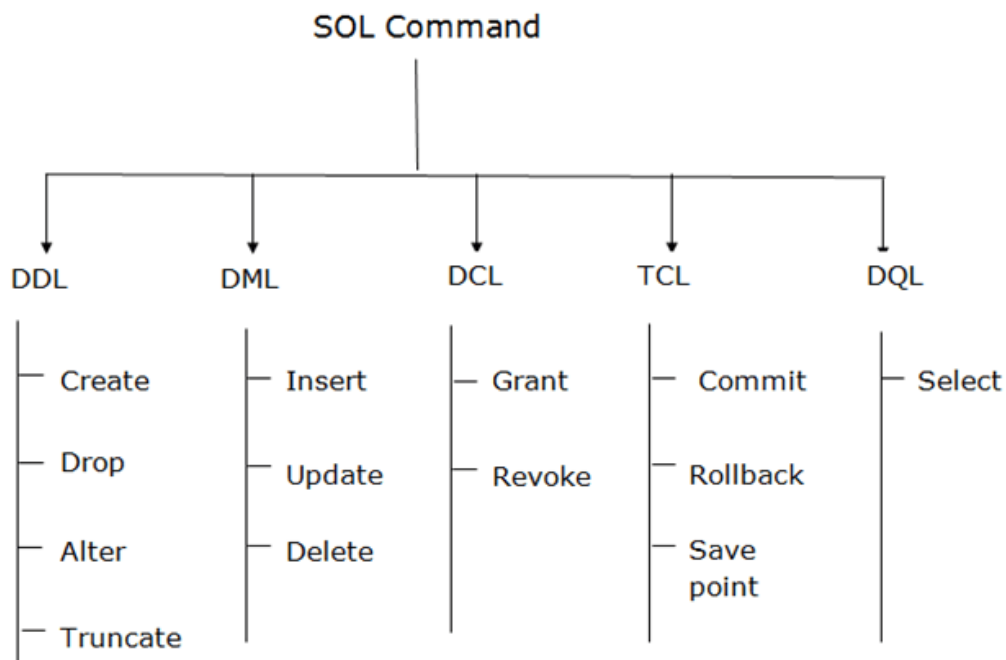
FID	FNAME	MOBILE_NO	JOBID	SALARY
2	LAKSH	9977665544	PROF	50000
3	KRISH	9977664422	ASO	40000

SQL Commands

- SQL commands are instructions. It is used to communicate with the database. It is also used to perform specific tasks, functions, and queries of data.
- SQL can perform various tasks like create a table, add data to tables, drop the table, modify the table, set permission for users.
- SQL commands are divided into five subgroups, DDL, DML, DCL, DQL and TCL.
-

Types of SQL Commands

There are five types of SQL commands: DDL, DML, DCL, TCL, and DQL.



1. Data Definition Language (DDL)

- DDL changes the structure of the table like creating a table, deleting a table, altering a table, etc.

- All the command of DDL are auto-committed that means it permanently save all the changes in the database.

Here are some commands that come under DDL:

- CREATE
- ALTER
- DROP
- TRUNCATE

a. **CREATE** It is used to create a new table in the database.

Syntax:

```
CREATE TABLE TABLE_NAME (COLUMN_NAME DATATYPES[,....]);
```

Example:

```
CREATE TABLE EMPLOYEE(Name VARCHAR2(20), Email VARCHAR2(100), DOB DATE
);
```

b. **DROP:** It is used to delete both the structure and record stored in the table.

Syntax

```
DROP TABLE;
```

Example

```
DROP TABLE EMPLOYEE;
```

c. **ALTER:** It is used to alter the structure of the database. This change could be either to modify the characteristics of an existing attribute or probably to add a new attribute.

Syntax:

To add a new column in the table

```
ALTER TABLE table_name ADD column_name COLUMN-definition;
```

To modify existing column in the table:

```
ALTER TABLE MODIFY(COLUMN DEFINITION....);
```

EXAMPLE

```
ALTER TABLE STU_DETAILS ADD(ADDRESS VARCHAR2(20));
```

```
ALTER TABLE STU_DETAILS MODIFY (NAME VARCHAR2(20));
```

d. **TRUNCATE**: It is used to delete **all the rows from the table** and free the space containing the table.

Syntax:

```
TRUNCATE TABLE table_name;
```

Example:

```
TRUNCATE TABLE EMPLOYEE;
```

2. Data Manipulation Language

- DML commands are used to modify the database. It is responsible for all form of changes in the database.
- The command of DML is not auto-committed that means it can't permanently save all the changes in the database. They can be rollback.

Here are some commands that come under DML:

- **INSERT**
- **UPDATE**
- **DELETE**

a. **INSERT**: The INSERT statement is a SQL query. It is used to insert data into the **row of a table**.

Syntax:

```
INSERT INTO TABLE_NAME  
    (col1, col2, col3,.... col N)  
    VALUES (value1, value2, value3, .... valueN);
```

Or

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

For example:

```
INSERT INTO javatpoint (Author, Subject) VALUES ("Sonoo", "DBMS");
```

```
INSERT INTO EMP19 (EMPNO,ENAME) VALUES(1,'BHANU');
```

b. UPDATE: This command is used to update or modify the value of a column in the table.

Syntax:

```
UPDATE table_name SET [column_name1= value1,...column_nameN = valueN] [WHERE CONDITION]
```

For example:

```
UPDATE students
    SET User_Name = 'Sonoo'
    WHERE Student_Id = '3'
UPDATE EMP19 SET EMPNO=3 WHERE ENAME='AKSHAY';
***
```

c. DELETE: It is used to remove one or more row from a table.

Syntax:

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

For example:

```
DELETE FROM javatpoint
    WHERE Author="Sonoo";
```

3. Data Control Language

DCL commands are used to grant and take back authority from any database user.

Here are some commands that come under DCL:

- **Grant**
- **Revoke**

a. Grant: It is used to give user access privileges to a database.

Example

```
GRANT SELECT, UPDATE ON MY_TABLE TO SOME_USER, ANOTHER_USER;
***
```

b. Revoke: It is used to take back permissions from the user.

Example

```
REVOKE SELECT, UPDATE ON MY_TABLE FROM USER1, USER2;
```

4. Transaction Control Language

TCL commands can only use with DML commands like INSERT, DELETE and UPDATE only.

These operations are automatically committed in the database that's why they cannot be used while creating tables or dropping them.

Here are some commands that come under TCL:

- **COMMIT**
- **ROLLBACK**
- **SAVEPOINT**

a. Commit: Commit command is used to save all the transactions to the database.

Syntax:

```
COMMIT;
```

Example:

```
DELETE FROM CUSTOMERS  
WHERE AGE = 25;  
COMMIT;
```

b. Rollback: Rollback command is used to undo transactions that have not already been saved to the database.

Syntax:

```
ROLLBACK;
```

Example:

```
DELETE FROM CUSTOMERS  
WHERE AGE = 25;  
ROLLBACK;
```

c. **SAVEPOINT:** It is used to roll the transaction back to a certain point without rolling back the entire transaction.

Syntax:

```
SAVEPOINT SAVEPOINT_NAME;  
***
```

5. Data Query Language

DQL is used to fetch the data from the database.

It uses only one command:

- o **SELECT**

a. **SELECT:** This is the same as the projection operation of relational algebra. It is used to select the attribute based on the condition described by WHERE clause.

Syntax:

```
SELECT expressions  
FROM TABLES  
WHERE conditions;
```

For example:

```
SELECT emp_name  
FROM employee  
WHERE age > 20;
```

What is a Primary Key in SQL?

Primary Key Constraint is a type of key through which you can uniquely identify every tuple or a record in a table. Every table can have only one primary key but can have multiple candidate keys. Also, each primary key should be unique and must not contain any NULL values.

Primary keys are used along with the foreign keys to refer to various tables and form referential integrities. For Table A, a primary key can consist of single or multiple columns.

Rules for Primary Key

The rules of Primary Key are as follows:

1. All the values in the column chosen as the primary key must be unique.
2. Each and every table can have only one primary key
3. No value in the primary key column can be NULL
4. You cannot insert a new row with a pre-existing primary key

Primary Key Operations:

To understand the various operations present on the primary key, consider the following table:

CUSTOMER_ID	CUSTOMER_NAME	PHONE_NUMBER

Primary Key on Create Table

You can use the following syntax to create a primary key on the “customerID” column while you are creating this table:

```
CREATE TABLE Customers (  
CustomerID int NOT NULL PRIMARY KEY,  
CustomerName varchar(10) NOT NULL,  
PhoneNumber int);
```

Apply Primary Key on Multiple Columns

To apply primary key on multiple columns while [creating a table](#), refer to the following example:

```
CREATE TABLE Customers (  
customerID int NOT NULL,  
CustomerName varchar(255) NOT NULL,  
PhoneNumber int,  
CONSTRAINT PK_Customer PRIMARY KEY (CustomerID, CustomerName)  
);
```

Refer to the below image.

edureka! **Primary Key**

CustomerID	CustomerName	PhoneNumber
1	Rohit	9876543210
2	Sonal	9765434567
3	Ajay	9765234562
4	Aishwarya	9876567899
5	Akash	9876541236

Primary Key on Alter Table

You can use the following syntax to create a primary key on the “customerID” column when the “customers” table is already created and you just want to alter the table:

```
ALTER TABLE Customers  
ADD PRIMARY KEY (CustomerID);
```

If you want to add a name to the Primary Key constraint and define it on multiple columns, use the following SQL syntax:

```
ALTER TABLE Customers  
ADD CONSTRAINT PK_Customer PRIMARY KEY (CustomerID, CustomerName);
```

Delete/ Drop Primary Key

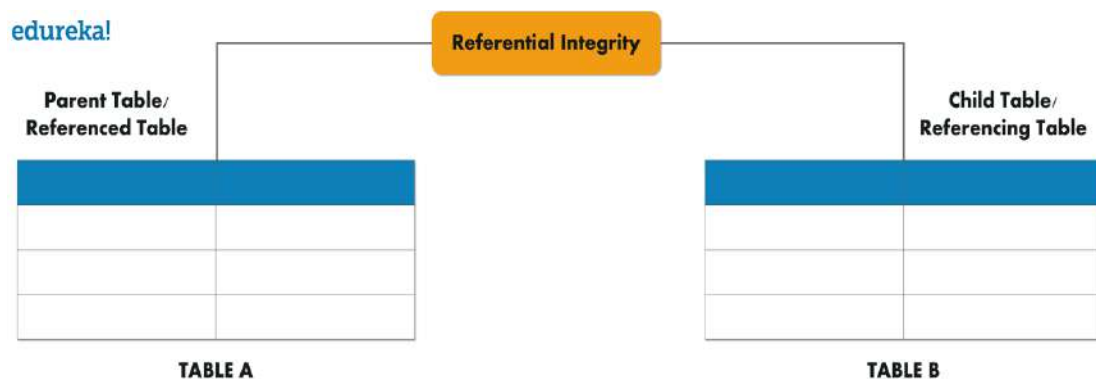
To drop the primary key, you can refer to the following example:

```
1 #For SQL Server/ MS Access/ Oracle  
2 ALTER TABLE Customers  
3 DROP CONSTRAINT PK_Customer;  
4 #For MySQL  
5 ALTER TABLE Customers  
6 DROP PRIMARY KEY;
```

What is Foreign Key(Referential integrity) constraint?

A foreign key is a type of key used to link two tables in a database. So, a foreign key is an attribute or a collection of attributes in one table that refers to the primary key in another table.

For Example, if Table A and Table B are related to each other, then if Table A consists of the primary key, this table would be called the referenced table or parent table. Similarly, if Table B consists of a foreign key, then that table is known as the referencing table or child table. Refer to the below image:



Foreign Key Operations:

To understand the various operations present on Foreign key, consider the following two tables:

Customer Table:

CustomerID	CustomerName	PhoneNumber
1	Rohan	9876543210
2	Sonali	9876567864
3	Ajay	9966448811
4	Geeta	9765432786

5	Shubham	9944888756
---	---------	------------

Courses Table:

CourseID	CourseName	CustomerID
c01	DevOps	2
c02	Machine Learning	4
c03	RPA	1
c04	Tableau	3
c05	AWS	2

Now, if you observe, the **customerID** column in the **courses** table refers to the **customerID** column in the **customers'** table. The **customerID** column from the **customers'** table is the Primary Key and the **customerID** column from the **courses** table is the Foreign Key of that table.

Starting with the first operation:

Foreign Key on Create Table

You can use the following syntax to create a foreign key on the "customerID" column when you create "courses" table:

```
CREATE TABLE courses (
courseID varchar NOT NULL PRIMARY KEY,
courseName varchar NOT NULL,
customerID int FOREIGN KEY REFERENCES customers(customerID)
);
```

Apply Foreign Key on Multiple Columns

To apply foreign key on multiple columns while [creating a table](#), refer to the following example:

```
CREATE TABLE courses (  
courseID varchar NOT NULL,  
courseName varchar NOT NULL,  
customerID int, PRIMARY KEY (courseID),  
CONSTRAINT FK_CustomerCourses FOREIGN KEY (customerID)  
REFERENCES customers(customerID)  
);
```

Foreign Key on Alter Table

You can use the following syntax to create a foreign key on the “customerID” column when the “courses” table is already created and you just want to alter the table:

```
ALTER TABLE courses  
ADD FOREIGN KEY (customerID) REFERENCES customers(customerID);
```

If you wish to add a name to the Foreign Key constraint and define it on multiple columns, use the following SQL syntax:

```
ALTER TABLE courses  
ADD CONSTRAINT FK_CustomerCourse  
FOREIGN KEY (customerID) REFERENCES Customers(customerID);
```

Drop Foreign Key:

To drop the foreign key, you can refer to the following example

```
ALTER TABLE courses
DROP CONSTRAINT FK_CustomerCourse;
```

EXAMPLE FOR PRIMARY AND FOREIGN KEYS:

HOW TO MAKE PRIMARY KEY AND FOREIGN KEY BETWEEN TWO TABLES BY QUERIES

1. CREATE TWO TABLES
2. BOTH TABLES IDS(COLUMN NAME) MUST BE PRIMARY KEY.
3. MAKE 1ST TABLE IDS(COLUMN NAME) AS FOREIGN KEY IN 2ND TABLE.

```
SQL> CREATE TABLE CUSTOMER( ID INT NOT NULL, NAME VARCHAR(10), PAYMENT
INT, PRIMARY KEY(ID));
```

Table created.

```
SQL> INSERT INTO CUSTOMER VALUES (101, 'RAM', 1000);
```

1 row created.

```
SQL> INSERT INTO CUSTOMER VALUES (102, 'LAKSHMAN', 2000);
```

1 row created.

```
SQL> INSERT INTO CUSTOMER VALUES(103, 'KRISHNA',3000);
```

1 row created.

```
SQL> INSERT INTO CUSTOMER VALUES(104, 'SIVA',3000);
```

1 row created.

```
SQL> INSERT INTO CUSTOMER VALUES(105, 'SATYA', 5000);
```

1 row created.

```
SQL> INSERT INTO CUSTOMER VALUES(103, 'SAI', 6000);
```

```
INSERT INTO CUSTOMER VALUES(103, 'SAI', 6000)
```

*

ERROR at line 1:

ORA-00001: unique constraint (DSR.SYS_C006090) violated

SQL> SELECT * FROM CUSTOMER; --TABLE--1

ID	NAME	PAYMENT
101	RAM	1000
102	LAKSHMAN	2000
103	KRISHNA	3000
104	SIVA	3000
105	SATYA	5000

```
CREATE TABLE supplier
( supplier_id numeric(10) not null,
  supplier_name varchar2(50) not null,
  contact_name varchar2(50),
  CONSTRAINT supplier_pk PRIMARY KEY (supplier_id)
);
```

SQL> select * from supplier;

SUPPLIER_ID	SUPPLIER_NAME	CONTACT_NAME
101	ADITYA	DAVID
102	BIRLA	RAHUL
103	RELIANCE	AMBANI
104	WIPRO	BASHA

```
CREATE TABLE products
( product_id numeric(10) not null,
  supplier_id numeric(10) not null,
  CONSTRAINT fk_supplier
    FOREIGN KEY (supplier_id)
    REFERENCES supplier(supplier_id)
);
```

SQL> insert into products values(1001, 101, 'mouse');

1 row created.

SQL> insert into products values(1002, 102, 'keyboard');

1 row created.

SQL> insert into products values(1003,103,'hard disk');

1 row created.

```
SQL> SELECT * FROM PRODUCTS;
```

PRODUCT_ID	SUPPLIER_ID	PRODUCT
1001	101	mouse
1002	102	keyboard
1003	103	hard disk
1004	104	LAPTOP

```
SQL> SELECT SUPPLIER_NAME FROM SUPPLIER, PRODUCTS WHERE  
SUPPLIER.SUPPLIER_ID=PRODUCTS.SUPPLIER_ID AND PRODUCTS.PRODUCT='LAPTOP';
```

SUPPLIER_NAME
WIPRO

REMAINDERS:

parent key not found

```
SQL> insert into products values(1004, 104, 'usb');
```

ERROR at line 1:

ORA-02291: integrity constraint (DSR.FK_SUPPLIER) violated - parent key not Found

Unique / primary keys in table referenced by foreign keys:

```
SQL> drop table supplier;
```

ERROR at line 1:

ORA-02449: unique/primary keys in table referenced by foreign keys

SQL | Numeric Functions

Numeric Functions are used to perform operations on numbers and return numbers.

Following are the numeric functions defined in SQL:

1. **ABS()**: It returns the absolute value of a number.

Syntax: `SELECT ABS(-243.5) FROM DUAL;`

Output:

```
ABS(-243.5)
-----
      243.5
```

SQL> `SELECT ABS(-10) FROM DUAL;`

Output:

```
ABS(-10)
-----
       10
```

2. **ACOS()**: It returns the cosine of a number.

Syntax: `SELECT ACOS(0.25) FROM DUAL;`

Output:

```
ACOS(0.25)
-----
1.31811607
```

3. **ASIN()**: It returns the arc sine of a number.

Syntax: `SELECT ASIN(0.25) FROM DUAL;`

Output:

```
ASIN(0.25)
-----
.252680255
```


4. **ATAN()**: It returns the arc tangent of a number.

Syntax: `SELECT ATAN(2.5) FROM DUAL;`

Output:

```
ATAN(2.5)
-----
1.19028995
```

5. **CEIL()**: It returns the smallest integer value that is greater than or equal to a number.

Syntax: `SELECT CEIL(25.75) FROM DUAL;`

Output:

```
CEIL(25.75)
-----
26
```

6. **CEILING()**: It returns the smallest integer value that is greater than or equal to a number.

Syntax: `SELECT CEILING(25.75) FROM DUAL; //its not working in sql plus`

Output: 26

7. **COS()**: It returns the cosine of a number.

Syntax: `SELECT COS(30) FROM DUAL;`

Output: 0.15425144988758405

```
COS(30)
-----
.15425145
```

8. **EXP()**: It returns e raised to the power of number.

Syntax: `SELECT EXP(1) FROM DUAL;`

Output: 2.718281828459045

```
EXP(1)
-----
2.71828183
```

9. **FLOOR()**: It returns the largest integer value that is less than or equal to a number.

Syntax: `SELECT FLOOR(25.75) FROM DUAL;`

Output: 25

```
FLOOR(25.75)
-----
          25
```

10. **GREATEST()**: It returns the greatest value in a list of expressions.

Syntax: `SELECT GREATEST(30, 2, 36, 81, 125) FROM DUAL;`

Output: 125

```
GREATEST(30,2,36,81,125)
-----
                   125
```

11. **LEAST()**: It returns the smallest value in a list of expressions.

Syntax: `SELECT LEAST(30, 2, 36, 81, 125) FROM DUAL;`

Output: 2

```
LEAST(30,2,36,81,125)
-----
                   2
```

12. **LN()**: It returns the natural logarithm of a number.

Syntax: `SELECT LN(2) FROM DUAL;`

Output: 0.6931471805599453

```
LN(2)
-----
.693147181
```

13. **MOD()**: It returns the remainder of n divided by m.

Syntax: `SELECT MOD(18, 4) FROM DUAL;`

Output: 2

```
MOD(18,4)
```

2

14. **ROUND()**: It returns a number rounded to a certain number of decimal places.

Syntax: SELECT ROUND(5.553) FROM DUAL;

Output: 6

ROUND(5.553)

6

15. **SIGN()**: It returns a value indicating the sign of a number.

Syntax: SELECT SIGN(255.5) FROM DUAL;

Output: 1

SIGN(255.5)

1

16. **SIN()**: It returns the sine of a number.

Syntax: SELECT SIN(20) FROM DUAL;

Output: 0.9092974268256817

SIN(20)

.912945251

17. **TAN()**: It returns the tangent of a number.

Syntax: SELECT TAN(1.75) FROM DUAL;

Output: -5.52037992250933

TAN(1.75)

-5.5203799

The SQL **NULL** is the term used to represent a missing value. A **NULL** value in a table is a value in a field that appears to be blank.

A field with a **NULL** value is a field with no value. It is very important to understand that a **NULL** value is different than a zero value or a field that contains spaces.

Syntax

The basic syntax of **NULL** while creating a table.

```
SQL> CREATE TABLE CUSTOMERS(  
    ID    INT                NOT NULL,  
    NAME  VARCHAR (20)      NOT NULL,  
    AGE   INT                NOT NULL,  
    ADDRESS CHAR (25) ,  
    SALARY DECIMAL (18, 2),  
    PRIMARY KEY (ID)  
);
```

Here, **NOT NULL** signifies that column should always accept an explicit value of the given data type. There are two columns where we did not use **NOT NULL**, which means these columns could be **NULL**.

A field with a **NULL** value is the one that has been left blank during the record creation.

Example

The **NULL** value can cause problems when selecting data. However, because when comparing an unknown value to any other value, the result is always unknown and not included in the results. You must use the **IS NULL** or **IS NOT NULL** operators to check for a **NULL** value.

Consider the following CUSTOMERS table having the records as shown below.

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

Now, following is the usage of the **IS NOT NULL** operator.

```
SQL> SELECT ID, NAME, AGE, ADDRESS, SALARY
      FROM CUSTOMERS
      WHERE SALARY IS NOT NULL;
```

This would produce the following result -

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	32	Ahmedabad	2000.00
2	Khilan	25	Delhi	1500.00
3	kaushik	23	Kota	2000.00
4	Chaitali	25	Mumbai	6500.00
5	Hardik	27	Bhopal	8500.00

Now, following is the usage of the **IS NULL** operator.

```
SQL> SELECT ID, NAME, AGE, ADDRESS, SALARY
      FROM CUSTOMERS
      WHERE SALARY IS NULL;
```

This would produce the following result -

ID	NAME	AGE	ADDRESS	SALARY
6	Komal	22	MP	
7	Muffy	24	Indore	

Example-2:

Creation of table:

```
SQL> create table cust(id int not null,
      name varchar(20) not null,
      age int not null,
      address char(25),
      primary key(id)
    );
```

Table created.

Insert values into table:

```
SQL> insert into cust values(1, 'ram', 19, 'viz');
```

1 row created.

```
SQL> insert into cust values(2, 'laksh',19, 'hyd');
```

1 row created.

- SQL> insert into cust values(3, 20, 'vizag');
insert into cust values(3, 20, 'vizag')

*

ERROR at line 1:

ORA-00947: not enough values

or

- SQL> insert into cust values(3, 'krish', 'vizag');
insert into cust values(3, 'krish', 'vizag')

*

ERROR at line 1:

ORA-00947: not enough values

or

- SQL> insert into cust (id, name, address)
values(3, 'krish', 'vizag');
insert into cust (id, name, address) values(3, 'krish', 'vizag')

*

ERROR at line 1:

ORA-01400: cannot insert NULL into ("DSR"."CUST"."AGE")

Or

- SQL> insert into cust (id, name, age, address)
values(3, 'krish', 'vizag');
insert into cust (id, name, age, address)
values(3, 'krish', 'vizag')

*

ERROR at line 1:

ORA-00947: not enough values

SQL> insert into cust (id, name, age) values(3, 'krish', 20);

1 row created.

```
SQL> select * from cust;
```

ID	NAME	AGE	ADDRESS
1	ram	19	viz
2	laksh	19	hyd
3	krish	20	

How to Test for NULL Values?

It is not possible to test for NULL values with comparison operators, such as =, <, or <>.

We will have to use the **IS NULL** and **IS NOT NULL** operators instead.

IS NULL Syntax

```
SELECT column_names  
FROM table_name  
WHERE column_name IS NULL;
```

Consider the following table for exmaple:

```
SQL> select * from emp19;
```

EMPNO	ENAME	AGE	CITY
1	BHANU	20	
2	SAMYOG	19	
3	AKSHAY	19	
	PINKY	19	HYD
	PINKY	19	VIJ

```
SQL> SELECT CITY FROM EMP19 WHERE CITY IS NULL;
```

```
CITY  
-----
```

IS NOT NULL Syntax

```
SELECT column_names
FROM table_name
WHERE column_name IS NOT NULL;
```

Consider the following table for exmaple:

```
SQL> select * from emp19;
```

EMPNO	ENAME	AGE	CITY
1	BHANU	20	
2	SAMYOG	19	
3	AKSHAY	19	
	PINKY	19	HYD
	PINKY	19	VIJ

```
EX:1 SQL> SELECT CITY FROM EMP19 WHERE CITY IS NOT NULL;
```

```
CITY
-----
HYD
VIJ
```

```
EX:2 SQL> SELECT AGE FROM EMP19 WHERE AGE IS NOT NULL;
```

```
AGE
-----
20
19
19
19
19
```

What is DUAL table?

The DUAL is special one row, one column table present by default in all Oracle databases. The owner of DUAL is SYS (SYS owns the data dictionary, therefore DUAL is part of the data dictionary.) but DUAL can be accessed by every user. The table has a single VARCHAR2(1) column called DUMMY that has a value of 'X'. MySQL allows DUAL to be specified as a table in queries that do not need data from any tables. In SQL Server DUAL table does not exist, but you could create one. The DUAL table was created by Charles Weiss of Oracle corporation to provide a table for joining in internal views.

See the following commands:

The following command displays the structure of DUAL table :

```
DESC DUAL;
```

Output:

Name	Null?	Type

DUMMY		VARCHAR2(1)

The following command displays the content of the DUAL table :

```
SELECT * FROM DUAL;
```

Output:

DUMMY

X

The following command displays the number of rows of DUAL table :

```
SELECT COUNT(*) FROM DUAL;
```

Output:

```
COUNT(*)
-----
        1
```

The following command displays the string value from the DUAL table :

```
SELECT 'ABCDEF12345' FROM DUAL;
```

Output:

```
'ABCDEF1234
-----
ABCDEF12345
```

The following command displays the numeric value from the DUAL table :

```
SELECT 123792.52 FROM DUAL;
```

Output:

```
123792.52
-----
123792.52
```

The following command tries to delete all rows from the DUAL table :

```
DELETE FROM DUAL;
```

Output:

```
DELETE FROM DUAL
          *
ERROR at line 1:
ORA-01031: insufficient privileges
```

The following command tries to remove all rows from the DUAL table:

```
TRUNCATE TABLE DUAL;
```

Note: The DELETE command is used to remove rows from a table. After performing a DELETE operation you need to COMMIT or ROLLBACK the transaction to make the change permanent or to undo it. TRUNCATE removes all rows from a table. The operation cannot be rolled back.

Output:

```
TRUNCATE TABLE DUAL
          *
ERROR at line 1:
ORA-00942: table or view does not exist
```

The following command select two rows from dual :

```
SELECT dummy FROM DUAL
```

```
UNION ALL
```

```
SELECT dummy FROM DUAL;
```

Output

```
DUMMY
```

```
-----
```

```
X
```

```
X
```

Example - 1

You can also check the system date from the DUAL table using the following statement :

```
SELECT sysdate FROM DUAL ;
```

Output:

```
SYSDATE
```

```
-----
```

```
11-DEC-10
```

Example - 2

You can also check the arithmetic calculation from the DUAL table using the following statement :

```
SELECT 15+10-5*5/5 FROM DUAL;
```

Output:

```
15+10-5*5/5
-----
          20
```

Example - 3

Following code display the numbers 1..10 from DUAL :

```
SELECT level
FROM DUAL
CONNECT BY level <=10;
```

Output:

```
      LEVEL
-----
         1
         2
         3
         4
         5
         6
         7
         8
         9
        10
```

Example - 4

In the following code, DUAL involves the use of decode with NULL.

```
SELECT decode(null,null,1,0)
FROM DUAL;
```

Output:

```
DECODE(NULL,NULL,1,0)
-----
                        1
```

DUAL table : Oracle vs MySQL

We have already learned that DUAL is a special one row one column table. For Oracle, it is useful because Oracle doesn't allow statements like :

```
SELECT 15+10-5*5/5;
```

Output:

```
SELECT 15+10-5*5/5
          *
```

ERROR at line 1:

ORA-00923: FROM keyword not found where expected

But the following command will execute (see the output of the previous example) :

```
SELECT 15+10-5*5/5 FROM DUAL;
```

In case of MySQL the following command will execute :

```
SELECT 15+10-5*5/5;
```

Output:

```
15+10-5*5/5
20.0000
```

The following table shows the uses of dummy table in standard DBMS.

DBMS	Dummy-table concept
MSSQL	No dummy-table concept.
MySQL	No dummy-table concept.
Oracle	Dummy-table : DUAL.
Informix	Since version 11.10, a dummy table has been included : sysmaster:sysdual
PostgreSQL	No dummy-table concept.
DB2	Dummy-table : SYSIBM.SYSDUMMY1

Consider the following table and adding new attribute (column DOB)

```
SQL> DESC EMP19;
```

Name	Null?	Type
-----	-----	-----
EMPNO		NUMBER(38)
ENAME		VARCHAR2(20)
AGE		NUMBER(38)
CITY		VARCHAR2(10)

```
SQL> ALTER TABLE EMP19 ADD DOB DATE;
```

Table altered.

After alter the table (adding new column) the table description is;

```
SQL> DESC EMP19;
```

Name	Null?	Type
-----	-----	-----
EMPNO		NUMBER(38)
ENAME		VARCHAR2(20)
AGE		NUMBER(38)
CITY		VARCHAR2(10)
DOB		DATE

```
SQL> INSERT INTO EMP19 VALUES(1, 'RAM', 19, 'VIZ', TO_DATE('01-01-2003',  
'DD-MM-YYYY'));
```

1 row created.

```
SQL> INSERT INTO EMP19 VALUES(2, 'LAKSH', 18, 'VIZ', TO_DATE('01-01-  
2004', 'DD-MM-YYYY'));
```

1 row created.

```
SQL> SELECT * FROM EMP19;
```

EMPNO	ENAME	AGE	CITY	DOB
1	RAM	19	VIZ	01-JAN-03
2	LAKSH	18	VIZ	01-JAN-04

ANOTHER TYPE INSERTION---IMP

```
SQL> INSERT INTO EMP19 VALUES(3,'KRISH', 17, 'VIZAG', DATE '2003-07-23');
```

1 row created.

```
SQL> SELECT * FROM EMP19;
```

EMPNO	ENAME	AGE	CITY	DOB
1	RAM	19	VIZ	01-JAN-03
2	LAKSH	18	VIZ	01-JAN-04
3	KRISH	17	VIZAG	23-JUL-03

Useful Date and Time Functions in PL/SQL

Date and Time Function formats are different various database. we are going to discuss most common functions used in Oracle database.

The function **SYSDATE** returns 7 bytes of data, which includes:

- Century
- Year
- Month
- Day
- Hour
- Minute
- Second

1. Extract():

Oracle helps you to extract Year, Month and Day from a date using Extract() Function.

Example-1: Extracting Year

```
SQL> SELECT SYSDATE AS CURRENT_DATE_TIME, EXTRACT(Year FROM  
SYSDATE) AS ONLY_CURRENT_YEAR FROM Dual;
```

CURRENT_D	ONLY_CURRENT_YEAR

25-APR-21	2021

Explanation:

Useful to retrieve only **year** from the System date/Current date or particular specified date.

Example-2: Extracting Month:

```
SQL> SELECT SYSDATE AS CURRENT_DATE_TIME, EXTRACT(Month FROM  
SYSDATE) AS ONLY_CURRENT_MONTH FROM DUAL;
```

Output:

CURRENT_D	ONLY_CURRENT_MONTH
-----	-----
25-APR-21	4

Explanation:

Useful to retrieve only **month** from the System date/Current date or particular specified date.

Example-3: Extracting Day:

```
SQL> SELECT SYSDATE AS CURRENT_DATE_TIME, EXTRACT(Day FROM  
SYSDATE) AS ONLY_CURRENT_DAY FROM Dual;
```

Output:

CURRENT_D	ONLY_CURRENT_DAY
-----	-----
25-APR-21	25

Explanation:

Useful to retrieve only **day** from the System date/Current date or particular specified date.

2. ADD_MONTHS(date,n):

Using this method in PL/SQL you can add as well as subtract number of months(n) to a date. Here 'n' can be both negative or positive.

Example-4:

```
SQL> SELECT ADD_MONTHS(SYSDATE, -1) AS PREV_MONTH,  
           SYSDATE AS CURRENT_DATE,  
           ADD_MONTHS(SYSDATE, 1) as NEXT_MONTH FROM Dual;
```

Output:

PREV_MONT	CURRENT_D	NEXT_MONT
25-MAR-21	25-APR-21	25-MAY-21

- **Explanation:**

ADD_MONTHS function have two parameters one is date, where it could be any specified/particular date or System date as current date and second is 'n', it is an integer value could be positive or negative to get upcoming date or previous date.

3. LAST_DAY (date):

Using this method in PL/SQL you can get the last day in the month of specified date.

Example-5:

```
SQL> SELECT SYSDATE AS CURRENT_DATE, LAST_DAY(SYSDATE) AS  
LAST_DAY_OF_MONTH, LAST_DAY(SYSDATE)+1 AS FIRST_DAY_OF_NEXT_MONTH  
FROM Dual;
```

Output:

CURRENT_D	LAST_DAY_	FIRST_DAY
25-APR-21	30-APR-21	01-MAY-21

Explanation:

In above example, we are getting current date using SYSDATE function and last date of the month would be retrieved using LAST_DAY function and this function be also helpful for retrieving the first day of the next month.

Example-6: Number of Days left in the month

```
SQL> SELECT SYSDATE AS CURRENT_DATE, LAST_DAY(SYSDATE) - SYSDATE AS  
DAYS_LEFT_IN_MONTH FROM Dual;
```

Output:

CURRENT_D	DAYS_LEFT_IN_MONTH
25-APR-21	5

4. MONTHS_BETWEEN(date1,date2):

Using this method in PL/SQL you can calculate the number of months between two entered dates date1 and date2. if date1 is later than date2 then the result would be positive and if date1 is earlier than date2 then result is negative.

Note:

If a fractional month is calculated, the MONTHS_BETWEEN function calculates the fraction based on a 31-day month.

Example-7:

```
SQL> SELECT MONTHS_BETWEEN (TO_DATE ('01-07-2003', 'dd-mm-yyyy'),  
TO_DATE ('14-03-2003', 'dd-mm-yyyy')) AS NUMBER_OF_MONTHS FROM Dual;
```

Output:

NUMBER_OF_MONTHS
3.58064516

Explanation:

Here date1 and date2 are not on the same day of the month that's why we are getting the value in fractions, as well as date1 is later than date2 so the resulting value is in integers. Entered date should be in particular date format, that is the reason of using TO_DATE function while comparison within MONTHS_BETWEEN functions.

Let's select the number of months an employee has worked for the company.

5. **NEXT_DAY(date,day_of_week):**

It will return the upcoming date of the first weekday that is later than the entered date. It has two parameters first date where, system date or specified date can be entered; second day of week which should be in character form.

Example-9:

```
SELECT NEXT_DAY (SYSDATE, 'SUNDAY') AS NEXT_SUNDAY FROM Dual;
```

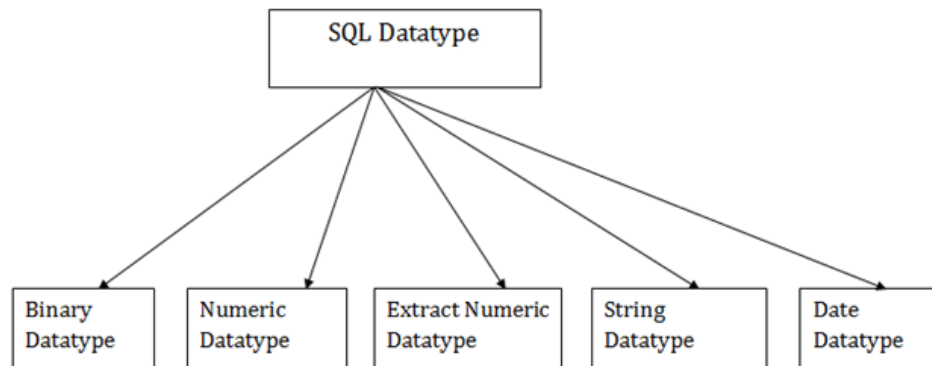
Output:

```
NEXT_SUND  
-----  
02-MAY-21
```

SQL Datatype

- SQL Datatype is used to define the values that a column can contain.
- Every column is required to have a name and data type in the database table.

Datatype of SQL:



1. Binary Datatypes

There are Three types of binary Datatypes which are given below:

Data Type	Description
binary	It has a maximum length of 8000 bytes. It contains fixed-length binary data.
varbinary	It has a maximum length of 8000 bytes. It contains variable-length binary data.
image	It has a maximum length of 2,147,483,647 bytes. It contains variable-length binary data.

2. Approximate Numeric Datatype :

The subtypes are given below:

Data type	From	To	Description
float	-1.79E + 308	1.79E + 308	It is used to specify a floating-point value e.g. 6.2, 2.9 etc.
real	-3.40e + 38	3.40E + 38	It specifies a single precision floating point number

3. Exact Numeric Datatype

The subtypes are given below:

Data type	Description
int	It is used to specify an integer value.
smallint	It is used to specify small integer value.
bit	It has the number of bits to store.
decimal	It specifies a numeric value that can have a decimal number.
numeric	It is used to specify a numeric value.

4. Character String Datatype

The subtypes are given below:

Data type	Description
char	It has a maximum length of 8000 characters. It contains Fixed-length non-unicode characters.
varchar	It has a maximum length of 8000 characters. It contains variable-length non-unicode characters.
text	It has a maximum length of 2,147,483,647 characters. It contains variable-length non-unicode characters.

5. Date and time Datatypes

The subtypes are given below:

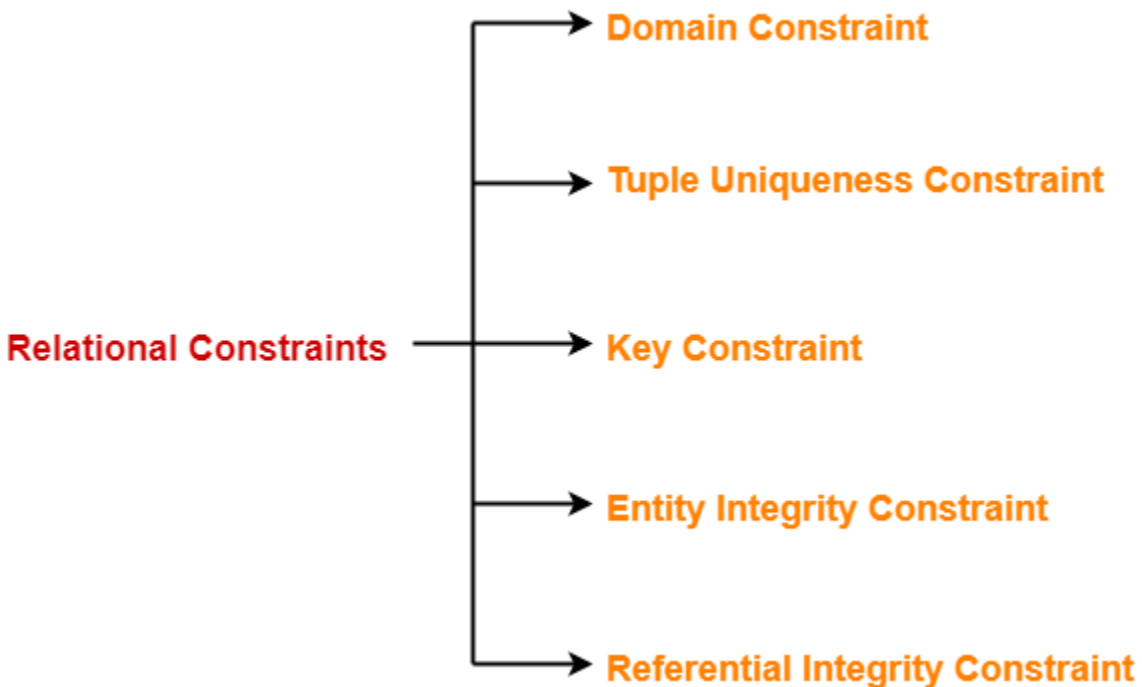
Datatype	Description
date	It is used to store the year, month, and days value.
time	It is used to store the hour, minute, and second values.
timestamp	It stores the year, month, day, hour, minute, and the second value.

Constraints in DBMS-

- CONSTRAINT means RESTRICTION.
- Relational constraints are the restrictions imposed on the database **contents and operations**.
- They ensure the correctness of data in the database.

Types of Constraints in DBMS-

In DBMS, there are following 5 different types of relational constraints-



1. Domain constraint
2. Tuple Uniqueness constraint
3. Key constraint
4. Entity Integrity constraint
5. Referential Integrity constraint

1. Domain Constraint-

- Domain constraint defines the domain or set of values for an attribute.
- It specifies that the value taken by the attribute must be the atomic value from its domain.

Example-

Consider the following Student table-

STU_ID	Name	Age
S001	Akshay	20
S002	Abhishek	21
S003	Shashank	20
S004	Rahul	A

Here, value 'A' is not allowed since only integer values can be taken by the age attribute.

2. Tuple Uniqueness Constraint-

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

Tuple Uniqueness constraint specifies that all the tuples must be necessarily unique in any relation.

Example-01:

Consider the following Student table-

<u>STU_ID</u>	Name	Age
S001	Akshay	20
S002	Abhishek	21
S003	Shashank	20
S004	Rahul	20

This relation satisfies the tuple uniqueness constraint since here all the tuples are unique.

Example-02:

Consider the following Student table-

<u>STU_ID</u>	Name	Age
S001	Akshay	20
S001	Akshay	20
S003	Shashank	20
S004	Rahul	20

This relation does not satisfy the tuple uniqueness constraint since here all the tuples are not unique.

3. Key Constraint-

Key constraint specifies that in any relation-

- All the values of primary key must be unique.
- The value of primary key **must not be null**.

Example-

Consider the following Student table-

<u>STU_ID</u>	Name	Age
S001	Akshay	20
S001	Abhishek	21
S003	Shashank	20
S004	Rahul	20

This relation does not satisfy the key constraint as here all the values of primary key are not unique.

4. Entity Integrity Constraint-

Entity Integrity is the mechanism the system provides to **maintain primary keys**. The primary key serves as a unique identifier for rows in the table. **Entity Integrity** ensures two properties for primary keys: **The primary key for a row is unique; it does not match the primary key of any other row in the table.**

- Entity integrity constraint specifies that no attribute of primary key must contain a null value in any relation.
- This is because the presence of null value in the primary key violates the uniqueness property.

Example-

Consider the following Student table-

<u>STU_ID</u>	Name	Age
S001	Akshay	20
S002	Abhishek	21
S003	Shashank	20
	Rahul	20

This relation does not satisfy the entity integrity constraint as here the primary key contains a NULL value.

5. Referential Integrity Constraint-

Whenever two **tables** contain one or more **common columns**, **Oracle** can enforce the relationship between the two **tables** through a **referential integrity constraint**. Define a PRIMARY or UNIQUE key **constraint** on the column in the parent **table** (the one that has the complete set of column values).

- This constraint is enforced when a foreign key references the primary key of a relation.
- It specifies that all the values taken by the foreign key must either be available in the relation of the primary key or be null.

Important Results-

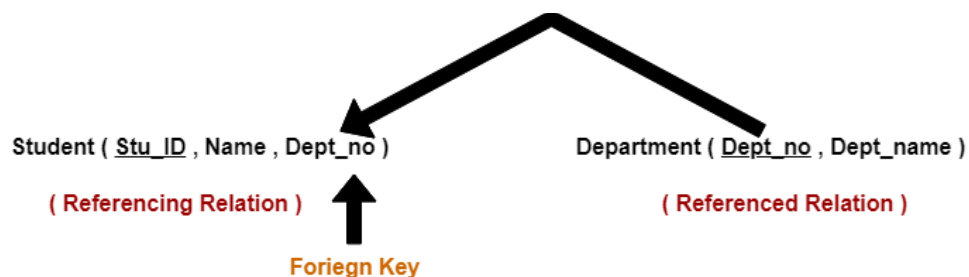
The following two important results emerges out due to referential integrity constraint-

- We can not insert a record into a referencing relation if the corresponding record does not exist in the referenced relation.
- We can not delete or update a record of the referenced relation if the corresponding record exists in the referencing relation.

Example-

Consider the following two relations- 'Student' and 'Department'.

Here, relation 'Student' references the relation 'Department'.



Student

<u>STU_ID</u>	Name	Dept_no
S001	Akshay	D10
S002	Abhishek	D10
S003	Shashank	D11
S004	Rahul	D14

Department

<u>Dept_no</u>	Dept_name
D10	ASET
D11	ALS
D12	ASFL
D13	ASHS

Here,

- The relation 'Student' does not satisfy the referential integrity constraint.
- This is because in relation 'Department', no value of primary key specifies department no. 14.
- Thus, referential integrity constraint is violated.

Referential integrity constraints **example-----2**

Referential integrity constraints is base on the concept of Foreign Keys. A foreign key is an important attribute of a relation which should be referred to in other relationships. Referential integrity constraint state happens where relation refers to a key attribute of a different or same relation. However, that key element must exist in the table.

Example:

Customer		
CustomerID	CustomerName	Status
1	Google	Active
2	Amazon	Active
3	Apple	Inactive

Billing		
InvoiceNo	CustomerID	Amount
1	1	\$100
2	1	\$200
3	2	\$150

In the above example, we have 2 relations, Customer and Billing.

Tuple for CustomerID =1 is referenced twice in the relation Billing. So we know CustomerName=Google has billing amount \$300

SQL TRUNCATE TABLE

SQL TRUNCATE TABLE command used to completely remove all table records. Not supporting to a WHERE clause.

SQL TRUNCATE command is faster and use some transaction log resources.

SQL TRUNCATE command logically equivalent to a DELETE command that deletes all rows, but they are practically different under some rules.

SQL TRUNCATE command Rules

- TRUNCATE operation use for dropping or re-create table, which is much faster than deleting rows one by one.
- TRUNCATE operation not rollback, It means truncated can not be returned.
- TRUNCATE operation is not a safe.

Syntax

Considering following syntax that help you to understanding TRUNCATE,

```
TRUNCATE TABLE table_name;
```

Example

```
SQL> TRUNCATE TABLE emp_data;
```

```
Table Truncated.
```

Different between Delete and Truncate Commands

	DELETE	TRUNCATE
1	<p>If we want to delete all the records from the table, then our SQL query is:</p> <pre>SQL> DELETE FROM table_name;</pre>	<p>Whereas Truncate command is used to delete all records from the table. Our SQL query is:</p> <pre>SQL> TRUNCATE TABLE table_name;</pre>
2	<p>DELETE statement with WHERE clause you can remove specific record in a table.</p> <pre>SQL> DELETE FROM table_name WHERE condition;</pre>	<p>Whereas TRUNCATE statement use for remove all record in a table.</p>
3	<p>DELETE statement is a DML (Data Manipulation Language) command.</p>	<p>Whereas TRUNCATE statement is a DDL (Data Definition Language) command.</p>
4	<p>Executed DELETE statement you can UNDO the changes and return back to deleted data.</p>	<p>Whereas executed TRUNCATE statement you can't return back.</p>

SQL INDEX

What is an index in SQL? SQL INDEX are used to quickly find data without searching every rows in a database table.

SQL INDEX is improving the speed of search operation on a database table. But additional you need more storage space to maintain duplicate copy of the database.

INDEX is a copy of the selected column of the database table to store additionally duplicate copy of the data.

End users does not know for indexes is created on table, only they are searching data more quickly and efficiently.

Type of SQL INDEX

CREATE INDEX statement to create indexes on a table. following 3 type indexes you are create on a table.

- **Simple INDEX** : Create INDEX on one column.
- **Composite INDEX** : Create INDEX on multiple columns.
- **Unique INDEX** : Create INDEX on column for restrict duplicate values on INDEX column.

Simple INDEX

Simple INDEX create only **one selected column of the database table**.

Syntax

```
CREATE INDEX index_name  
    ON table_name (column_name)
```

Storage setting specifies the table space explicitly. This are the optional storage setting if you are not specifies automatically default storage setting used.

Consider the following employee table

```
SQL> SELECT * FROM EMPLOYEE;
```

ID	NAME	AGE	ADDRESS	SALARY
111	RAM	22	DELHI	13000
222	LAKSH	20	DELHI	15000
333	VENKAT	23	MUMBAI	14000
444	KRISH	21	HYD	17000
555	SIVA	25	BANGLORE	16000
666	SATYA	24	CHENNAI	18000

Example:

```
SQL> CREATE INDEX EMP_IND ON EMPLOYEE (AGE);
```

Index created.

We are creating simple index on `age` column of the `employee` table. In this column allow duplicate values of the column.

Composite INDEX

Composite INDEX creates on multiple selected column of the database table.

Syntax

```
CREATE INDEX index_name
ON table_name (column_name, column_name)
```

Example

```
SQL> CREATE INDEX EMP_IND ON EMPLOYEE(ID, AGE);
```

Index created.

We are creating composite index on `ID` and `AGE` column of the `EMPLOYEE` table. Duplicate values are allowing for creating indexes.

Unique INDEX

Unique INDEX creates on selected column of the database table and **does not allow duplicate values of that indexes column.**

Syntax

```
CREATE UNIQUE INDEX index_name  
  ON table_name (column_name)  
  [ storage_setting ];
```

We are create unique index on **ID** column of the **EMPLOYEE** table. Duplicate name value are does not allow again for creating indexes.

Consider the following table for unique index

SQL> SELECT * FROM EMPLOYEE;

ID	NAME	AGE	ADDRESS	SALARY
111	RAM	22	DELHI	13000
222	LAKSH	20	DELHI	15000
333	VENKAT	23	MUMBAI	14000
444	KRISH	21	HYD	17000
555	SIVA	25	BANGLORE	16000
666	SATYA	24	CHENNAI	18000
777	GANESH	22	PUNE	20000

7 rows selected.

Example

SQL> CREATE UNIQUE INDEX EMP_INDEX ON EMPLOYEE (NAME);

Index created

```
SQL> INSERT INTO EMPLOYEE VALUES(777, 'GANESH', 23, 'PUNE', 15000);
```

```
INSERT INTO EMPLOYEE VALUES(777, 'GANESH', 23, 'PUNE', 15000)
```

*

ERROR at line 1:

ORA-00001: unique constraint (DSR.SYS_C006091) violated

```
SQL> INSERT INTO EMPLOYEE VALUES(888, 'GANESH', 23, 'PUNE', 15000);
```

1 row created.

```
SQL> SELECT * FROM EMPLOYEE;
```

ID	NAME	AGE	ADDRESS	SALARY
111	RAM	22	DELHI	13000
222	LAKSH	20	DELHI	15000
333	VENKAT	23	MUMBAI	14000
444	KRISH	21	HYD	17000
555	SIVA	25	BANGLORE	16000
666	SATYA	24	CHENNAI	18000
777	GANESH	22	PUNE	20000
888	GANESH	23	PUNE	15000

8 rows selected.

```
SQL> CREATE UNIQUE INDEX UNQ_IND1 ON EMPLOYEE(NAME);
```

```
CREATE UNIQUE INDEX UNQ_IND1 ON EMPLOYEE(NAME)
```

*

ERROR at line 1:

ORA-01452: cannot CREATE UNIQUE INDEX; duplicate keys found

RENAME INDEX

Syntax

```
ALTER INDEX index_name  
    RENAME TO new_index_name;
```

Example

```
SQL> ALTER INDEX EMP_IND  
    RENAME TO EMP_INDEX;
```

We are renaming the above created index name `EMP_IND` to a new index name `EMP_INDEX`.

DROP INDEX

Syntax

```
DROP INDEX index_name;
```

Example

```
SQL> DROP INDEX EMP_INDEX;
```

In this statement we are dropping `EMP_INDEX` INDEX.

Subclasses, Superclasses, and Inheritance

An entity type is used to represent both a *type of entity* and the *entity set* or *collection of entities of that type* that exist in the database.

For example, the entity type EMPLOYEE describes the type (that is, the attributes and relationships) of each employee entity, and also refers to the current set of EMPLOYEE entities in the COMPANY database.

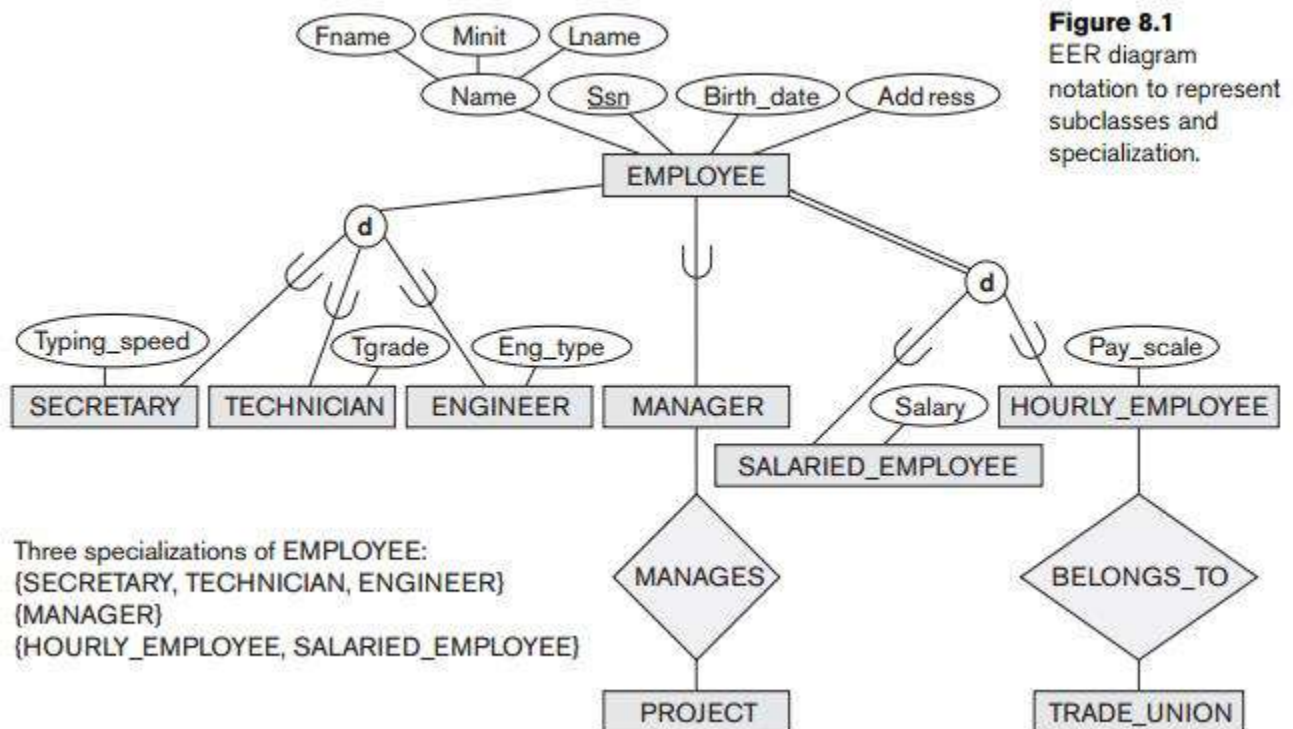
In many cases an entity type has numerous subgroupings or subtypes of its entities that are meaning-full and need to be represented explicitly because of their significance to the database application.

For example, the entities that are members of the EMPLOYEE entity type may be distinguished further into SECRETARY, ENGINEER, MANAGER, TECHNICIAN, SALARIED_EMPLOYEE, HOURLY_EMPLOYEE, and so on.

The set of entities in each of the latter groupings is a subset of the entities that belong to the EMPLOYEE entity set, meaning that every entity that is a member of one of these subgroupings is also an employee. We call each of these subgroupings a **subclass** or **subtype** of the EMPLOYEE entity type, and the EMPLOYEE entity type is called the **superclass** or **supertype** for each of these subclasses. Figure 8.1 shows how to represent these concepts diagrammatically in EER diagrams. (The circle notation in Figure 8.1 will be explained in Section 8.2.)

We call the relationship between a superclass and any one of its subclasses a **superclass/subclass** or **supertype / subtype** or simply **class / subclass relationship**.

In our previous example, **EMPLOYEE / SECRETARY** and **EMPLOYEE / TECHNICIAN** are two class / subclass relationships. Notice that a member entity of the subclass represents 'the same real-world entity' as some member of the superclass; for example, a **SECRETARY** entity 'Joan' is also the **EMPLOYEE** 'Joan.' Hence, the subclass member is the same as the entity in the superclass, but in a distinct *specific role*. When we implement a superclass / subclass relationship in the database system, however, we may represent a member of the subclass as a distinct database object—say, a distinct record that is related via the key attribute to its superclass entity. In Section 9.2, we discuss various options for representing superclass/subclass relationships in relational databases.



the **d** symbol in the circles in Figure 8.1 and additional EER diagram notation shortly.

An entity cannot exist in the database merely by being a member of a subclass; it must also be a member of the superclass. Such an entity can be included optionally as a member of any number of subclasses. For example, a salaried employee who is also an engineer belongs to the two subclasses ENGINEER and SALARIED_EMPLOYEE of the EMPLOYEE entity type. However, it is not necessary that every entity in a superclass is a member of some subclass.

An important concept associated with subclasses (subtypes) is that of **type inheritance**. Recall that the *type* of an entity is defined by the attributes it possesses and the relationship types in which it participates. Because an entity in the subclass represents the same real-world entity from the superclass, it should possess values for its specific attributes as well as values of its attributes as a member of the superclass.

An entity that is a member of a subclass inherits all the attributes of the entity as a member of the superclass.

The entity also inherits all the relationships in which the superclass participates.

Notice that a subclass, with its own specific (or local) attributes and relationships together with all the attributes and relationships it inherits from the superclass.

Understanding Object Inheritance

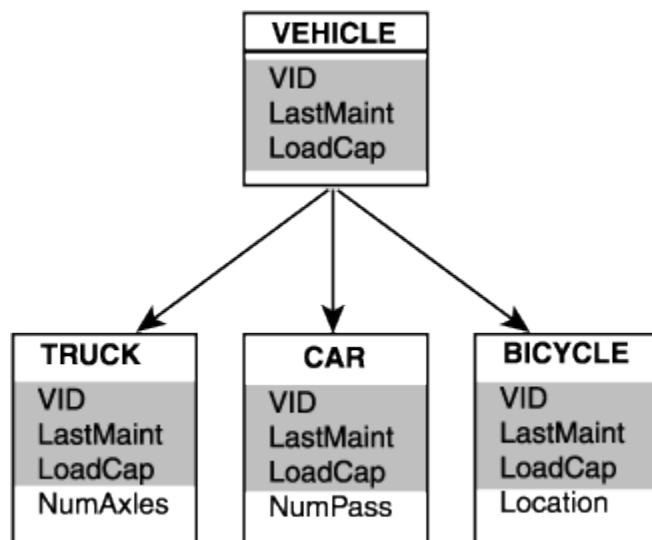
Consider a simple database used by a courier company. It contains registration information for three types of vehicles: trucks, cars,

and bicycles. For each vehicle type, your application requires the following information:

- VID (Vehicle Identification)
- LastMaint (mileage since last maintenance)
- LoadCap (load capacity)

If these are all the attributes shared by all vehicles in the application, then these attributes must all appear in the super class, Vehicle. You can then build subclasses for each of the vehicle types that reflects their differences. For example, the Truck class may have an attribute indicating whether the local department of transportation considers it to be a commercial vehicle (NumAxles), the Car class may require a NumPass (number of passengers) attribute, and the Bicycle class, by virtue of its more limited range, may require a Location attribute. Through inheritance, each vehicle automatically inherits the basic vehicle information, but by being separate subclasses, also have unique characteristics.

Figure 3-6 Inheritance in a Courier Application



Representing Inheritance in the Database

You can represent inheritance in the database in one of two ways:

- Multiple tables that represent the parent class and each child class
- A single table that comprises the parent and all child classes

Figure 3-7 Inheritance in the Database in Individual Tables

VEHICLE Table

VID	LastMaint	LoadCap
1	2002	850
2	2000	30
3	2001	920
4	1998	1700
5	2003	35
6	2001	2250

TRUCK Table

VID	LastMaint	LoadCap	NumPass
4	1998	1700	5
6	2001	2250	3

CAR Table

VID	LastMaint	LoadCap	NumPass
1	2002	850	5
3	2001	920	7

BICYCLE Table

VID	LastMaint	LoadCap	NumPass
2	2000	30	1
5	2003	35	1

Specialization and Generalization

1. Specialization

Specialization is the process of *defining a set of subclasses of an entity type*; this entity type is called *the superclass of the specialization*. The set of subclasses that forms a specialization is defined on the basis of some distinguishing characteristic of the entities in the superclass. For example, the set of subclasses {SECRETARY, ENGINEER, TECHNICIAN} is a specialization of the superclass EMPLOYEE that distinguishes among employee entities based on the *job type* of each employee entity. We may have several specializations of the same entity type based on different distinguishing characteristics. For example, another specialization of the EMPLOYEE entity type may yield the set of subclasses {SALARIED_EMPLOYEE, HOURLY_EMPLOYEE}; this specialization distinguishes among employees based on the *method of pay*.

Figure 8.1 shows how we represent a specialization diagrammatically in an EER diagram. The subclasses that define a specialization are attached by lines to a circle that represents the specialization, which is connected in turn to the superclass. The *subset symbol* on each line

connecting a subclass to the circle indicates the direction of the superclass / subclass relationship. Attributes that apply only to entities of a particular subclass – such as TypingSpeed of SECRETARY – are attached to the rec - tangle representing that subclass. These are called **specific attributes** (or **local attributes**) of the subclass. Similarly, a subclass can participate in **specific relation-ship types**, such as the HOURLY_EMPLOYEE subclass participating in the BELONGS TO relationship in Figure 8.1. We will explain **the d symbol in the circles in Figure 8.1 and additional EER diagram notation shortly.**

Figure 8.2 shows a few entity instances that belong to subclasses of the {SECRETARY, ENGINEER, TECHNICIAN} specialization. Again, notice that an entity that belongs to a subclass represents *the same real-world entity* as the entity connected to it in the EMPLOYEE superclass, even though the same entity is shown twice; for example, *e1* is shown in both EMPLOYEE and SECRETARY in Figure 8.2. As the figure suggests, a superclass / subclass relationship such as EMPLOYEE/ SECRETARY somewhat resembles a 1:1 relationship *at the instance level* (see Figure 7.12). The main difference is that in a 1:1 relationship two *distinct entities* are related, whereas in a superclass / subclass relationship the

entity in the subclass is the same real-world entity as the entity in the superclass but is playing a *specialized role*—for example, an EMPLOYEE specialized in the role of SECRETARY, or an EMPLOYEE specialized in the role of TECHNICIAN.

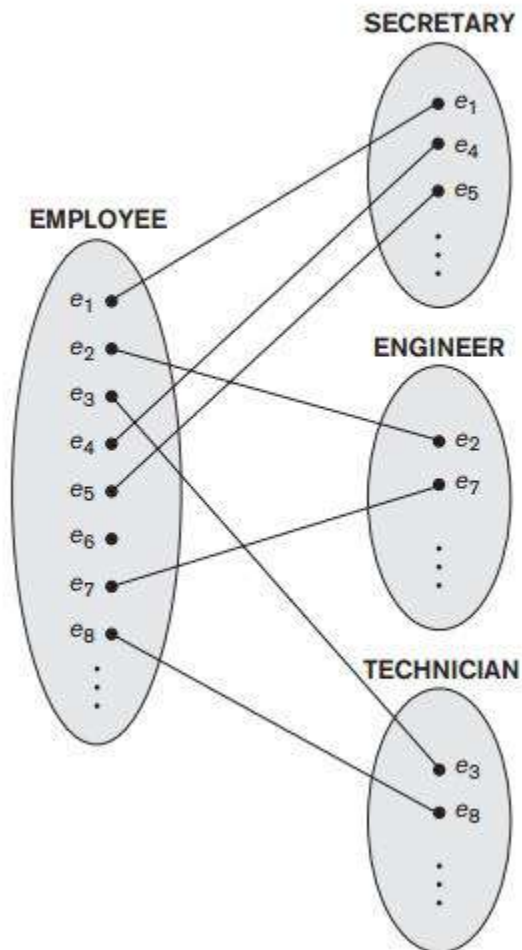


Figure 8.2
Instances of a specialization.

There are two main reasons for including class/subclass relationships and specializations in a data model. The first is that certain attributes may apply to some but not all entities of the superclass. A subclass is defined in order to group the entities to which these attributes apply. The members of the subclass may still share the majority of

their attributes with the other members of the superclass. For example, in Figure 8.1 the SECRETARY subclass has the specific attribute Typing_speed, whereas the ENGINEER subclass has the specific attribute Eng_type, but SECRETARY and ENGINEER share their other inherited attributes from the EMPLOYEE entity type.

The second reason for using subclasses is that some relationship types may be participated in only by entities that are members of the subclass. For example, if only HOURLY_EMPLOYEES can belong to a trade union, we can represent that fact by creating the subclass HOURLY_EMPLOYEE of EMPLOYEE and relating the subclass to an entity type TRADE_UNION via the BELONGS_TO relationship type, as illustrated in Figure 8.1.

In summary, the specialization process allows us to do the following:

- Define a set of subclasses of an entity type

- Establish additional specific attributes with each subclass

- Establish additional specific relationship types between each subclass and other entity types or other subclasses

2. Generalization

We can think of a *reverse process* of abstraction in which we suppress the differences among several entity types, identify their common features, and **generalize** them into a single **superclass** of which the original entity types are special **subclasses**. For example, consider the entity types CAR and TRUCK shown in Figure 8.3(a). Because they have several common attributes, they can be generalized into the entity type VEHICLE, as shown in Figure 8.3(b). Both CAR and TRUCK are now subclasses of the **generalized superclass** VEHICLE. We use the term **generalization** to refer to the process of defining a generalized entity type from the given entity types.

Notice that the generalization process can be viewed as being functionally the inverse of the specialization process. Hence, in Figure 8.3 we can view {CAR, TRUCK} as a specialization of VEHICLE, rather than viewing VEHICLE as a generalization of CAR and TRUCK. Similarly, in Figure 8.1 we can view EMPLOYEE as a generalization of SECRETARY, TECHNICIAN, and ENGINEER. A diagrammatic notation to distinguish between generalization and specialization is used in some design methodologies. An arrow pointing to the generalized superclass represents a generalization, whereas arrows pointing to the specialized subclasses represent a

specialization. We will *not* use this notation because the decision as to which process is followed in a particular situation is often subjective. Appendix A gives some of the suggested alternative dia-grammatic notations for schema diagrams and class diagrams.

So far we have introduced the concepts of subclasses and superclass/subclass relationships, as well as the specialization and generalization processes. In general, a

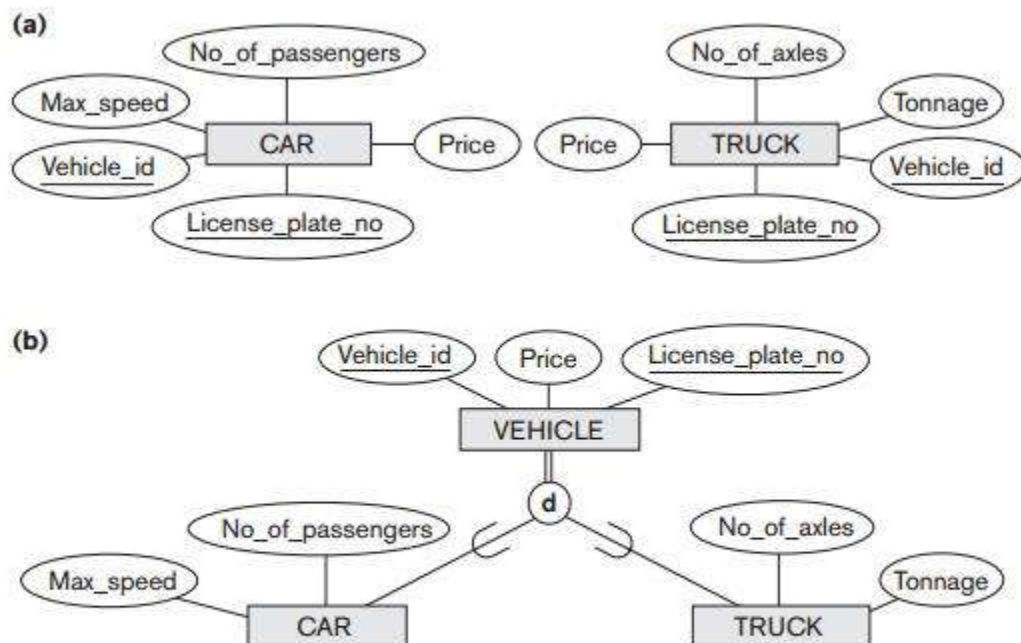


Figure 8.3
Generalization. (a) Two entity types, CAR and TRUCK. (b)
Generalizing CAR and TRUCK into the superclass VEHICLE.

superclass or subclass represents a collection of entities of the same type and hence also describes an *entity type*; that is why superclasses and subclasses are all shown in rectangles in EER diagrams, like entity types. Next, we

discuss the properties of specializations and generalizations in more detail.

Commit, Rollback and Savepoint SQL commands

Transaction Control Language(TCL) commands are used to manage transactions in the [database](#).

These are used to manage the changes made to the data in a table by DML statements. It also allows statements to be grouped together into logical transactions.

COMMIT command

COMMIT command is used to permanently save any transaction into the database.

When we use any DML command like **INSERT**, **UPDATE** or **DELETE**, the changes made by these commands are not permanent, until the current session is closed, the changes made by these commands can be rolled back.

To avoid that, we use the **COMMIT** command to mark the changes as permanent.

Following is commit command's syntax,

```
COMMIT;
```

ROLLBACK command

This command restores the database to last committed state. It is also used with **SAVEPOINT** command to jump to a savepoint in an ongoing transaction.

If we have used the **UPDATE** command to make some changes into the database, and realise that those changes were not required, then we can use the **ROLLBACK** command to rollback those changes, if they were not committed using the **COMMIT** command.

Following is rollback command's syntax,

```
ROLLBACK TO savepoint_name;
```

SAVEPOINT command

SAVEPOINT command is used to temporarily save a transaction so that you can rollback to that point whenever required.

Following is savepoint command's syntax,

```
SAVEPOINT savepoint_name;
```

In short, using this command we can **name** the different states of our data in any table and then rollback to that state using the **ROLLBACK** command whenever required.

Consider the following table:

```
SQL> CREATE TABLE CLASS(SID NUMBER, SNAME VARCHAR2(10));
```

Table created.

```
SQL> INSERT INTO CLASS VALUES(1, 'RAM');
```

1 row created.

```
SQL> INSERT INTO CLASS VALUES(2, 'KRSHNA');
```

1 row created.

```
SQL> INSERT INTO CLASS VALUES(3, 'SIVA');
```

1 row created.

```
SQL> SELECT * FROM CLASS;
```

```
      SID SNAME
-----
1      RAM
2 KRSHNA
3  SIVA
```

Lets use some SQL queries on the above table and see the results.

Using Savepoint and Rollback

```
SQL> INSERT INTO CLASS VALUES (4, 'LAKSHMAN');  
1 row created.
```

```
SQL> COMMIT;
```

Commit complete.

```
SQL> SELECT * FROM CLASS;
```

SID	SNAME
1	RAM
2	KRSHNA
3	SIVA
4	LAKSHMAN

Now we are update the value of SNAME= LAKSHMAN to GANESH where SID=4.

```
SQL> UPDATE CLASS SET SNAME='GANESH' WHERE SID=4;  
1 row updated.
```

After update the value in CLASS table we are creating a SAVEPOINT SP1.

```
SQL> SAVEPOINT SP1;  
Savepoint created.
```

Insert new values into CLASS table

```
SQL> INSERT INTO CLASS VALUES(6, 'HARI');  
1 row created.
```

Now we are creating another save point SP2 after inserting new value ie SID=6.

```
SQL> SAVEPOINT SP2;
```

Savepoint created.

Insert another value into CLASS table

```
SQL> INSERT INTO CLASS VALUES(5, 'SATYA');  
1 row created.
```

Now we are creating another save point SP3 after inserting new value ie SID=5.

```
SQL> SAVEPOINT SP3;
```

Savepoint created.

NOTE: **SELECT** statement is used to show the data stored in the table.
The resultant table will look like,

```
SQL> SELECT * FROM CLASS;
```

SID	SNAME
1	RAM
2	KRSHNA
3	SIVA
4	GANESH
6	HARI
5	SATYA

6 rows selected.

Now let's use the **ROLLBACK** command to roll back the state of data to the **SAVEPOINT SP2**.

```
SQL> ROLLBACK TO SP2;  
Rollback complete.
```

Now our class table will look like..

```
SQL> SELECT * FROM CLASS;
```

SID	SNAME
1	RAM
2	KRSHNA
3	SIVA
4	GANESH
6	HARI

Now let's again use the **ROLLBACK** command to roll back the state of data to the **SAVEPOINT SP1**;

```
SQL> ROLLBACK TO SP1;  
Rollback complete.
```

Now our class table will look like..

```
SQL> SELECT * FROM CLASS;
```

SID	SNAME
1	RAM
2	KRSHNA
3	SIVA
4	GANESH

Oracle GROUP BY Clause

In Oracle **GROUP BY** clause is used with **SELECT** statement to *collect data from multiple records* and group the results by one or more columns.

Syntax: **SELECT** expression1, expression2, ... expression_n,
 aggregate_function (aggregate_expression)
FROM tables
WHERE conditions
GROUP BY expression1, expression2, ... expression_n;

Parameters:

- **expression1, expression2, ... expression_n:** It specifies the expressions that are not encapsulated within aggregate function. These expressions must be included in GROUP BY clause.
- **aggregate_function:** It specifies the aggregate functions i.e. SUM, COUNT, MIN, MAX or AVG functions.
- **aggregate_expression:** It specifies the column or expression on that the aggregate function is based on.
- **tables:** It specifies the table from where you want to retrieve records.
- **conditions:** It specifies the conditions that must be fulfilled for the record to be selected.

Oracle GROUP BY Example: (with SUM function)

Let's take a table "salesdeptT"

```
SQL> CREATE TABLE "SALESDEPT "(  
      ITEM VARCHAR(15),  
      SALE INT,  
      BILLING_ADDRESS VARCHAR(15)  
    )  
    /
```

Table created.

```

SQL> INSERT INTO SALESDEPT VALUES('COMPUTER', 10, 'VIJ');
      1 row created.
SQL> INSERT INTO SALESDEPT VALUES('MOBILE', 5, 'ELR');
      1 row created.
SQL> INSERT INTO SALESDEPT VALUES('MEDICINES', 100, 'GUNTUR');
      1 row created.
SQL> INSERT INTO SALESDEPT VALUES('LAPTOP', 7, 'VIZAG');
      1 row created.
SQL> INSERT INTO SALESDEPT VALUES('MOUSE', 6, 'TIRUPATI');
      1 row created.
SQL> INSERT INTO SALESDEPT VALUES('MOBILE', 10, 'RJY');
      1 row created.
SQL> INSERT INTO SALESDEPT VALUES('LAPTOP', 5, 'HYD');
      1 row created.
SQL> INSERT INTO SALESDEPT VALUES('LAPTOP',10, 'VIJ');
      1 row created.

```

```

SQL> SELECT * FROM SALESDEPT;

```

ITEM	SALE BILLING_ADDRESS
COMPUTER	10 VIJ
MOBILE	5 ELR
MEDICINES	100 GUNTUR
LAPTOP	7 VIZAG
MOUSE	6 TIRUPATI
MOBILE	10 RJY
LAPTOP	5 HYD
LAPTOP	10 VIJ

```

8 rows selected.

```

```
SQL> SELECT ITEM, SUM(SALE) FROM SALESDEPT GROUP BY ITEM;
```

ITEM	SUM(SALE)
COMPUTER	10
MEDICINES	100
MOBILE	15
MOUSE	6
LAPTOP	22

EXAMPLE-2

Consider the bellow table of data

```
SQL> CREATE TABLE "EMPLOYEES"(  
    EID INT NOT NULL,  
    ENAME VARCHAR(10) NOT NULL,  
    AGE INT,  
    DEPT VARCHAR(10),  
    SALARY INT  
)  
/  
Table created.
```

Inset the values into table

```
SQL> INSERT INTO EMPLOYEES VALUES(101, 'RAM', 20, 'SOFTWARE',  
    30000);
```

1 row created.

```
SQL> INSERT INTO EMPLOYEES VALUES(102, 'KRISH', 22, 'SOFTWARE',  
    35000);
```

1 row created.

```
SQL> INSERT INTO EMPLOYEES VALUES(103, 'LAKSH', 23, 'SOFTWARE',  
    40000);
```

1 row created.

```
SQL> INSERT INTO EMPLOYEES VALUES(104, 'SIVA', 22, 'MECHANICAL',  
20000);
```

1 row created.

```
SQL> INSERT INTO EMPLOYEES VALUES(105, 'VENKAT', 23, 'HARDWARE',  
25000);
```

1 row created.

```
SQL> INSERT INTO EMPLOYEES VALUES(106, 'SATYA', 22, 'CIVIL',  
30000);
```

1 row created.

```
SQL> SELECT * FROM EMPLOYEES;
```

EID	ENAME	AGE	DEPT	SALARY
101	RAM	20	SOFTWARE	30000
102	KRISH	22	SOFTWARE	35000
103	LAKSH	23	SOFTWARE	40000
104	SIVA	22	MECHANICAL	20000
105	VENKAT	23	HARDWARE	25000
106	SATYA	22	CIVIL	30000

6 rows selected.

- **Group By example with MIN() function:**

```
SQL> SELECT DEPT, MIN (SALARY) FROM EMPLOYEES GROUP BY DEPT;
```

DEPT	MIN(SALARY)
HARDWARE	25000
SOFTWARE	30000
CIVIL	30000
MECHANICAL	20000

- Group By example with MAX() function:

```
SQL> SELECT DEPT, MAX(SALARY) FROM EMPLOYEES GROUP BY DEPT;
```

DEPT	MAX(SALARY)
HARDWARE	25000
SOFTWARE	40000
CIVIL	30000
MECHANICAL	20000

- Group By example with COUNT() FUNCTION:

```
SQL> SELECT DEPT, COUNT (*) FROM EMPLOYEES WHERE SALARY > 20000
GROUP BY DEPT;
```

DEPT	COUNT(*)
HARDWARE	1
SOFTWARE	3
CIVIL	1

```
SQL> SELECT DEPT, EID, COUNT(*) AS TOTALEMLOYEE FROM EMPLOYEES
GROUP BY DEPT, EID;
```

DEPT	EID	TOTALEMLOYEE
MECHANICAL	104	1
HARDWARE	105	1
SOFTWARE	102	1
CIVIL	106	1
SOFTWARE	101	1
SOFTWARE	103	1

```
SQL> SELECT DEPT, COUNT(*) AS TOTALEMLOYEE FROM EMPLOYEES GROUP
      BY DEPT;
```

DEPT	TOTALEMLOYEE
HARDWARE	1
SOFTWARE	3
CIVIL	1
MECHANICAL	1

EXAMPLE-3-GROUP BY USING MULTIPLE TABLES

Consider the following tables

```
SQL> SELECT * FROM STUDENT;
```

S_ID	SNAME	AGE	ADDRESS
101	RAM	19	VIJ
102	LAKSHMAN	18	HYD
103	KRISH	20	VIZAG
104	VENKAT	21	vi j
105	SIVA	22	VIZAG
106	SATYA	20	TIRUPATI

```
SQL> SELECT * FROM COURSE ORDER BY S_ID;
```

COURSE_ID	S_ID
2	101
1	101
2	102
2	103
3	103
2	104
3	104
1	105

QUESTION:

Display S_ID, SNAME of the student and total no. of courses selected by each student (from course table) using GROUP BY Clause

```
SQL> SELECT S.S_ID, S.SNAME, COUNT (C.COURSE_ID) AS TOTALCOURSES FROM  
STUDENT S LEFT JOIN COURSE C ON S.S_ID = C.S_ID GROUP BY S.S_ID,  
S.SNAME;
```

S_ID	SNAME	TOTALCOURSES
102	LAKSHMAN	1
104	VENKAT	2
105	SIVA	1
101	RAM	2
103	KRISH	2
106	SATYA	0

6 rows selected.

```
SQL> SELECT S.S_ID, COUNT (C.COURSE_ID) AS TOTALCOURSES FROM STUDENT S  
LEFT JOIN COURSE C ON S.S_ID = C.S_ID GROUP BY S.S_ID;
```

S_ID	TOTALCOURSES
102	1
101	2
104	2
105	1
103	2
106	0

6 rows selected.


```
SQL> SELECT S.S_ID, COUNT (*) AS TOTALCOURSES FROM STUDENT S LEFT JOIN
COURSE C ON S.S_ID= C.S_ID GROUP BY S.S_ID;
```

S_ID	TOTALCOURSES
102	1
101	2
104	2
105	1
103	2
106	1

6 rows selected.

```
SQL> SELECT S.S_ID, COUNT(*)AS TOTALCOURSES FROM STUDENT S LEFT JOIN
COURSE C ON S.S_ID= C.S_ID GROUP BY S.S_ID ORDER BY S.S_ID;
```

S_ID	TOTALCOURSES
101	2
102	1
103	2
104	2
105	1
106	1

6 rows selected.

Question:

Display COURSE_ID and total no. of Students selected a single course by considering student and course tables using GROUP BY Clause.

```
SQL> SELECT C.COURSE_ID, COUNT(*) AS TOTALSTUDENTS FROM STUDENT S
      RIGHT JOIN COURSE C ON S.S_ID = C.S_ID GROUP BY C.COURSE_ID ORDER
      BY COURSE_ID;
```

COURSE_ID	TOTALSTUDENTS
1	2
2	4
3	2

```
SQL> SELECT C.COURSE_ID, COUNT(*) AS TOTALSTUDENTS FROM STUDENT S LEFT
      JOIN COURSE C ON S.S_ID = C.S_ID GROUP BY C.COURSE_ID ORDER BY
      COURSE_ID;
```

COURSE_ID	TOTALSTUDENTS
1	2
2	4
3	2
	1

HAVING Clause

The **HAVING** clause was added to SQL because the **WHERE** keyword cannot be used with aggregate functions.

The **HAVING Clause** enables you to specify conditions that filter which group results appear in the results.

The **WHERE** clause places conditions on the selected columns, whereas the **HAVING** clause places conditions on groups created by the **GROUP BY** clause.

The **HAVING** clause must follow the **GROUP BY** clause in a query and must also precedes the **ORDER BY** clause if used. The following code block has the syntax of the **SELECT** statement including the **HAVING** clause -

Syntax

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

Consider the following **STUDENT** table:

```
SQL> SELECT * FROM STUDENT;
```

S_ID	SNAME	AGE	ADDRESS
101	RAM	19	VIJ
102	LAKSHMAN	18	HYD
103	KRISH	20	VIZAG
104	VENKAT	21	VIJ
105	SIVA	22	VIZAG
106	SATYA	20	TIRUPATI

QUESTION:

Select the S_ID and ADDRESS from student table by grouping the S_Id based on the ADDRESS.

```
SQL> SELECT COUNT(S_ID), ADDRESS FROM STUDENT GROUP BY ADDRESS HAVING  
COUNT(S_ID) > 5;
```

no rows selected

```
SQL> SELECT COUNT (S_ID), ADDRESS FROM STUDENT GROUP BY ADDRESS HAVING  
COUNT(S_ID) < 3;
```

COUNT(S_ID)	ADDRESS
1	HYD
1	TIRUPATI
2	VIJ
2	VIZAG

```
SQL> SELECT COUNT (S_ID), ADDRESS FROM STUDENT GROUP BY ADDRESS HAVING  
COUNT(S_ID) < 2;
```

COUNT(S_ID)	ADDRESS
1	HYD
1	TIRUPATI

Oracle GROUP BY Clause

In Oracle **GROUP BY** clause is used with **SELECT** statement to *collect data from multiple records* and group the results by one or more columns.

Syntax: **SELECT** expression1, expression2, ... expression_n,
 aggregate_function (aggregate_expression)
FROM tables
WHERE conditions
GROUP BY expression1, expression2, ... expression_n;

Parameters:

- **expression1, expression2, ... expression_n:** It specifies the expressions that are not encapsulated within aggregate function. These expressions must be included in GROUP BY clause.
- **aggregate_function:** It specifies the aggregate functions i.e. SUM, COUNT, MIN, MAX or AVG functions.
- **aggregate_expression:** It specifies the column or expression on that the aggregate function is based on.
- **tables:** It specifies the table from where you want to retrieve records.
- **conditions:** It specifies the conditions that must be fulfilled for the record to be selected.

Oracle GROUP BY Example: (with SUM function)

Let's take a table "salesdeptT"

```
SQL> CREATE TABLE "SALESDEPT "(  
      ITEM VARCHAR(15),  
      SALE INT,  
      BILLING_ADDRESS VARCHAR(15)  
    )  
    /
```

Table created.

```

SQL> INSERT INTO SALESDEPT VALUES('COMPUTER', 10, 'VIJ');
      1 row created.
SQL> INSERT INTO SALESDEPT VALUES('MOBILE', 5, 'ELR');
      1 row created.
SQL> INSERT INTO SALESDEPT VALUES('MEDICINES', 100, 'GUNTUR');
      1 row created.
SQL> INSERT INTO SALESDEPT VALUES('LAPTOP', 7, 'VIZAG');
      1 row created.
SQL> INSERT INTO SALESDEPT VALUES('MOUSE', 6, 'TIRUPATI');
      1 row created.
SQL> INSERT INTO SALESDEPT VALUES('MOBILE', 10, 'RJY');
      1 row created.
SQL> INSERT INTO SALESDEPT VALUES('LAPTOP', 5, 'HYD');
      1 row created.
SQL> INSERT INTO SALESDEPT VALUES('LAPTOP',10, 'VIJ');
      1 row created.

```

```
SQL> SELECT * FROM SALESDEPT;
```

ITEM	SALE BILLING_ADDRESS
COMPUTER	10 VIJ
MOBILE	5 ELR
MEDICINES	100 GUNTUR
LAPTOP	7 VIZAG
MOUSE	6 TIRUPATI
MOBILE	10 RJY
LAPTOP	5 HYD
LAPTOP	10 VIJ

8 rows selected.

```
SQL> SELECT ITEM, SUM(SALE) FROM SALESDEPT GROUP BY ITEM;
```

ITEM	SUM(SALE)
COMPUTER	10
MEDICINES	100
MOBILE	15
MOUSE	6
LAPTOP	22

EXAMPLE-2

Consider the bellow table of data

```
SQL> CREATE TABLE "EMPLOYEES"(  
    EID INT NOT NULL,  
    ENAME VARCHAR(10) NOT NULL,  
    AGE INT,  
    DEPT VARCHAR(10),  
    SALARY INT  
)  
/  
Table created.
```

Inset the values into table

```
SQL> INSERT INTO EMPLOYEES VALUES(101, 'RAM', 20, 'SOFTWARE',  
    30000);
```

1 row created.

```
SQL> INSERT INTO EMPLOYEES VALUES(102, 'KRISH', 22, 'SOFTWARE',  
    35000);
```

1 row created.

```
SQL> INSERT INTO EMPLOYEES VALUES(103, 'LAKSH', 23, 'SOFTWARE',  
    40000);
```

1 row created.

```
SQL> INSERT INTO EMPLOYEES VALUES(104, 'SIVA', 22, 'MECHANICAL',  
20000);
```

1 row created.

```
SQL> INSERT INTO EMPLOYEES VALUES(105, 'VENKAT', 23, 'HARDWARE',  
25000);
```

1 row created.

```
SQL> INSERT INTO EMPLOYEES VALUES(106, 'SATYA', 22, 'CIVIL',  
30000);
```

1 row created.

```
SQL> SELECT * FROM EMPLOYEES;
```

EID	ENAME	AGE	DEPT	SALARY
101	RAM	20	SOFTWARE	30000
102	KRISH	22	SOFTWARE	35000
103	LAKSH	23	SOFTWARE	40000
104	SIVA	22	MECHANICAL	20000
105	VENKAT	23	HARDWARE	25000
106	SATYA	22	CIVIL	30000

6 rows selected.

- **Group By example with MIN() function:**

```
SQL> SELECT DEPT, MIN (SALARY) FROM EMPLOYEES GROUP BY DEPT;
```

DEPT	MIN(SALARY)
HARDWARE	25000
SOFTWARE	30000
CIVIL	30000
MECHANICAL	20000

- Group By example with MAX() function:

```
SQL> SELECT DEPT, MAX(SALARY) FROM EMPLOYEES GROUP BY DEPT;
```

DEPT	MAX(SALARY)
HARDWARE	25000
SOFTWARE	40000
CIVIL	30000
MECHANICAL	20000

- Group By example with COUNT() FUNCTION:

```
SQL> SELECT DEPT, COUNT(*) FROM EMPLOYEES WHERE SALARY > 20000
GROUP BY DEPT;
```

DEPT	COUNT(*)
HARDWARE	1
SOFTWARE	3
CIVIL	1

```
SQL> SELECT DEPT, EID, COUNT(*) AS TOTALEMLOYEE FROM EMPLOYEES
GROUP BY DEPT, EID;
```

DEPT	EID	TOTALEMLOYEE
MECHANICAL	104	1
HARDWARE	105	1
SOFTWARE	102	1
CIVIL	106	1
SOFTWARE	101	1
SOFTWARE	103	1

```
SQL> SELECT DEPT, COUNT(*) AS TOTALEMLOYEE FROM EMPLOYEES GROUP
      BY DEPT;
```

DEPT	TOTALEMLOYEE
HARDWARE	1
SOFTWARE	3
CIVIL	1
MECHANICAL	1

EXAMPLE-3-GROUP BY USING MULTIPLE TABLES

Consider the following tables

```
SQL> SELECT * FROM STUDENT;
```

S_ID	SNAME	AGE	ADDRESS
101	RAM	19	VIJ
102	LAKSHMAN	18	HYD
103	KRISH	20	VIZAG
104	VENKAT	21	vi j
105	SIVA	22	VIZAG
106	SATYA	20	TIRUPATI

```
SQL> SELECT * FROM COURSE ORDER BY S_ID;
```

COURSE_ID	S_ID
2	101
1	101
2	102
2	103
3	103
2	104
3	104
1	105

QUESTION:

Display S_ID, SNAME of the student and total no. of courses selected by each student (from course table) using GROUP BY Clause

```
SQL> SELECT S.S_ID, S.SNAME, COUNT (C.COURSE_ID) AS TOTALCOURSES FROM  
STUDENT S LEFT JOIN COURSE C ON S.S_ID = C.S_ID GROUP BY S.S_ID,  
S.SNAME;
```

S_ID	SNAME	TOTALCOURSES
102	LAKSHMAN	1
104	VENKAT	2
105	SIVA	1
101	RAM	2
103	KRISH	2
106	SATYA	0

6 rows selected.

```
SQL> SELECT S.S_ID, COUNT (C.COURSE_ID) AS TOTALCOURSES FROM STUDENT S  
LEFT JOIN COURSE C ON S.S_ID = C.S_ID GROUP BY S.S_ID;
```

S_ID	TOTALCOURSES
102	1
101	2
104	2
105	1
103	2
106	0

6 rows selected.

```
SQL> SELECT S.S_ID, COUNT (*) AS TOTALCOURSES FROM STUDENT S LEFT JOIN
COURSE C ON S.S_ID= C.S_ID GROUP BY S.S_ID;
```

S_ID	TOTALCOURSES
102	1
101	2
104	2
105	1
103	2
106	1

6 rows selected.

```
SQL> SELECT S.S_ID, COUNT(*)AS TOTALCOURSES FROM STUDENT S LEFT JOIN
COURSE C ON S.S_ID= C.S_ID GROUP BY S.S_ID ORDER BY S.S_ID;
```

S_ID	TOTALCOURSES
101	2
102	1
103	2
104	2
105	1
106	1

6 rows selected.

Question:

Display COURSE_ID and total no. of Students selected a single course by considering student and course tables using GROUP BY Clause.

```
SQL> SELECT C.COURSE_ID, COUNT(*) AS TOTALSTUDENTS FROM STUDENT S
      RIGHT JOIN COURSE C ON S.S_ID = C.S_ID GROUP BY C.COURSE_ID ORDER
      BY COURSE_ID;
```

COURSE_ID	TOTALSTUDENTS
1	2
2	4
3	2

```
SQL> SELECT C.COURSE_ID, COUNT(*) AS TOTALSTUDENTS FROM STUDENT S LEFT
      JOIN COURSE C ON S.S_ID = C.S_ID GROUP BY C.COURSE_ID ORDER BY
      COURSE_ID;
```

COURSE_ID	TOTALSTUDENTS
1	2
2	4
3	2
	1

HAVING Clause

The **HAVING** clause was added to SQL because the **WHERE** keyword cannot be used with aggregate functions.

The **HAVING Clause** enables you to specify conditions that filter which group results appear in the results.

The **WHERE** clause places conditions on the selected columns, whereas the **HAVING** clause places conditions on groups created by the **GROUP BY** clause.

The **HAVING** clause must follow the **GROUP BY** clause in a query and must also precede the **ORDER BY** clause if used. The following code block has the syntax of the **SELECT** statement including the **HAVING** clause -

Syntax

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

Consider the following **STUDENT** table:

```
SQL> SELECT * FROM STUDENT;
```

S_ID	SNAME	AGE	ADDRESS
101	RAM	19	VIJ
102	LAKSHMAN	18	HYD
103	KRISH	20	VIZAG
104	VENKAT	21	VIJ
105	SIVA	22	VIZAG
106	SATYA	20	TIRUPATI

QUESTION:

Select the S_ID and ADDRESS from student table by grouping the S_Id based on the ADDRESS.

```
SQL> SELECT COUNT(S_ID), ADDRESS FROM STUDENT GROUP BY ADDRESS HAVING  
COUNT(S_ID) > 5;
```

no rows selected

```
SQL> SELECT COUNT (S_ID), ADDRESS FROM STUDENT GROUP BY ADDRESS HAVING  
COUNT(S_ID) < 3;
```

COUNT(S_ID)	ADDRESS
1	HYD
1	TIRUPATI
2	VIJ
2	VIZAG

```
SQL> SELECT COUNT (S_ID), ADDRESS FROM STUDENT GROUP BY ADDRESS HAVING  
COUNT(S_ID) < 2;
```

COUNT(S_ID)	ADDRESS
1	HYD
1	TIRUPATI

Views in SQL

- Views in SQL are considered as a **virtual table**. A view also contains rows and columns.
- To create the view, we can select the fields from one or more tables present in the database.
- A view can either have specific rows based on certain condition or all the rows of a table.
- **View is used to restrict data access.**

1. Creating view

A view can be created using the **CREATE VIEW** statement.

We can create a view from a single table or multiple tables.

Syntax:

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

OR

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

The data fetched from **SELECT** statement will be stored in another object called **view_name**(user defined view name).

We can use **CREATE** and **REPLACE** separately too, but using both together works better, as if any view with the specified name exists, this query will replace it with fresh data.

2. Creating View from a single table

In this example, we create a View named **STU_VIEW** from the table **STUDENT**.

```
SQL> CREATE OR REPLACE VIEW STU_VIEW AS
      SELECT SNAME, AGE FROM STUDENT
      WHERE AGE=20;
```

View created.

- Just like table query, we can query the view to view the data.

```
SQL> SELECT SNAME, AGE FROM STUDENT WHERE AGE=20;
```

SNAME	AGE
-----	-----
KRISH	20
SATYA	20

- **Displaying a VIEW**

The syntax for displaying the data in a view is similar to fetching data from a table using a **SELECT** statement.

```
SQL> SELECT * FROM STU_VIEW;
```

SNAME	AGE
-----	-----
KRISH	20
SATYA	20

EXAMPLE-2

```
SQL> CREATE OR REPLACE VIEW STU_VIEW AS SELECT SNAME, AGE FROM STUDENT
      WHERE S_ID < 105;
```

View created.

```
SQL> SELECT * FROM STU_VIEW;
```

SNAME	AGE
RAM	19
LAKSHMAN	18
KRISH	20
VENKAT	21

➤ Just like table query, we can query the view to view the data.

```
SQL> SELECT SNAME, AGE FROM STUDENT WHERE S_ID < 105;
```

SNAME	AGE
RAM	19
LAKSHMAN	18
KRISH	20
VENKAT	21

3. Creating View from multiple tables

View from multiple tables can be created by simply include multiple tables in the SELECT statement.

In the given example, a view is created named **MarksView** from two tables **Student** and **SMarks**.

Consider the following tables for creating view from multiple tables.

Table: 1 SQL> SELECT * FROM STUDENT;

S_ID	SNAME	AGE	ADDRESS
101	RAM	19	VIJ
102	LAKSHMAN	18	HYD
103	KRISH	20	VIZAG
104	VENKAT	21	vij

105 SIVA	22 VIZAG
106 SATYA	20 TIRUPATI

6 rows selected.

Table: 2

SQL> SELECT * FROM SMARKS;

S_ID	SNAME	MARKS
101	RAM	95
102	LAKSHMAN	85
103	KRISH	80
104	VENKAT	70
105	SIVA	75
106	SATYA	72

6 rows selected.

Query: SQL> CREATE OR REPLACE VIEW MARKS_VIEW AS
SELECT STUDENT.S_ID, STUDENT.SNAME, SMARKS.MARKS
FROM STUDENT, SMARKS
WHERE STUDENT.SNAME = SMARKS.SNAME;

View created.

To display the above created **view** the query is:

SQL> SELECT * FROM MARKS_VIEW;

S_ID	SNAME	MARKS
101	RAM	95
102	LAKSHMAN	85
103	KRISH	80
104	VENKAT	70
105	SIVA	75
106	SATYA	72

4. UPDATING VIEWS:

There are certain conditions needed to be satisfied to update a view. If any one of these conditions is **not** met, then we will not be allowed to update the view.

1. The SELECT statement which is used to create the view should not include **GROUP BY clause or ORDER BY clause**.
2. The SELECT statement should not have the DISTINCT keyword.
3. The View should have all NOT NULL values.
4. The view should not be created using nested queries or complex queries.
5. **The view should be created from a single table. If the view is created using multiple tables then we will not be allowed to update the view.**

We can use the CREATE OR REPLACE VIEW statement to add or remove from a view.

SYNTAX:

```
CREATE OR REPLACE VIEW view_name AS
SELECT column1, column2,...
FROM Table_Name
WHERE Condition;
```

View_name: Name of the view

For example, if we want to update the view **MARKS_VIEW** and add the field AGE to this View from **STUDENT** Table, we can do this as:

Before update **MARKS_VIEW** table the data in the table is”

```
SQL> select * from marks_view;

      S_ID SNAME          MARKS
-----
      101 RAM              95
      102 LAKSHMAN         85
      103 KRISH             80
      104 VENKAT            70
      105 SIVA              75
      106 SATYA             72
```

6 rows selected.

```
SQL> CREATE OR REPLACE VIEW MARKS_VIEW AS
      SELECT STUDENT.SNAME, STUDENT.ADDRESS, SMARKS.MARKS, STUDENT.AGE
      FROM STUDENT, SMARKS
      WHERE STUDENT.SNAME = SMARKS.SNAME;
```

View created.

➤ AFTER UPDATED THE VIEW WITH **AGE** COLUMN

```
SQL> SELECT * FROM MARKS_VIEW;
```

SNAME	ADDRESS	MARKS	AGE
RAM	VIJ	95	19
LAKSHMAN	HYD	85	18
KRISH	VIZAG	80	20
VENKAT	VIJ	70	21
SIVA	VIZAG	75	22
SATYA	TIRUPATI	72	20

6 rows selected.

When can a view be updated using update statement?

1. The view is defined based on one and only one table.
2. The view must include the PRIMARY KEY of the table based upon which the view has been created.
3. The view should not have any field made out of aggregate functions.
4. The view must not have any DISTINCT clause in its definition.
5. The view must not have any GROUP BY or HAVING clause in its definition.
6. The view must not have any SUBQUERIES in its definitions.
7. If the view you want to update is based upon another view, the later should be updatable.
8. Any of the selected output fields (of the view) must not use constants, strings or value

```
SQL> UPDATE MARKS_VIEW SET MARKS = 95 WHERE S_ID = 104;
```

```
      update marks_view set marks = 95 where s_id = 104
```

```
      *
```

```
      ERROR at line 1:
```

```
ORA-01779: cannot modify a column which maps to a non key-preserved table
```

```
NOTE: Here MARKS_VIEW is created by using 2 table STUDENT and SMARKS.
```

❖ Now consider the STU_VIEW (This view is created using single table STUDENT)

```
SQL> SELECT * FROM STU_VIEW;
```

S_ID	SNAME	AGE
101	RAM	19
102	LAKSHMAN	18
103	KRISH	20
104	VENKAT	21
105	SIVA	22
106	SATYA	20

6 rows selected.

```
SQL> UPDATE STU_VIEW SET AGE=20 WHERE S_ID=102;
```

1 row updated.

After updating the record in STU_VIEW the data in the STU_VIEW is:

```
SQL> SELECT * FROM STU_VIEW;
```

S_ID	SNAME	AGE
101	RAM	19
102	LAKSHMAN	20
103	KRISH	20
104	VENKAT	21
105	SIVA	22
106	SATYA	20

6 rows selected.

After updating the record in the STU_VIEW the data in the table STUDENT is:

```
SQL> SELECT * FROM STUDENT;
```

S_ID	SNAME	AGE	ADDRESS
101	RAM	19	VIJ
102	LAKSHMAN	20	HYD
103	KRISH	20	VIZAG
104	VENKAT	21	VIJ
105	SIVA	22	VIZAG
106	SATYA	20	TIRUPATI

6 rows selected.

➤ Non Updatable Views:

- Non-Updateable Views may affect **INSERT, UPDATE, and DELETE** operations.
- One of the main reasons, why the views become non-updateable is because of inclusion of aggregate functions (which also includes DISTINCT), Group By, and Join.
- Also in the cases of Nested Views, which includes those views that is non-updateable, will cause the final view also to be non-updateable.

As shown below, I have used a table called Student.

```
SQL> SELECT * FROM STUDENT;
```

S_ID	SNAME	AGE	ADDRESS
101	RAM	19	VIJ
102	LAKSHMAN	18	HYD
103	KRISH	20	VIZAG
104	VENKAT	21	VIJ
105	SIVA	22	VIZAG
106	SATYA	20	TIRUPATI

6 rows selected.

Now, we will create a view name "VSTU" which will also include aggregate function like DISTINCT, and see what happens:

```
SQL> CREATE OR REPLACE VIEW VSTU AS  
      SELECT DISTINCT AGE FROM STUDENT;
```

View created.

Display the view VSTU--

```
SQL> SELECT * FROM VSTU;
```

```
    AGE
```

```
-----
```

```
    22
```

```
    20
```

```
    21
```

```
    18
```

```
    19
```

Therefore, to test the updateability of the view, the next query attempts to perform an INSERT command through the view:

```
SQL> INSERT INTO VSTU (AGE) VALUES (25);
```

```
INSERT INTO VSTU (AGE) VALUES (25)
```

```
*
```

```
ERROR at line 1:
```

```
ORA-01732: data manipulation operation not legal on this view
```

Now, instead of result an error arouses;

Cannot update the view or function 'VSTU' because it contains aggregates, or a DISTINCT or GROUP BY clause.

```
SQL> DELETE FROM VSTU WHERE AGE = 22;
```

```
DELETE FROM VSTU WHERE AGE = 22
```

```
*
```

```
ERROR at line 1:
```

```
ORA-01732: data manipulation operation not legal on this view
```

➤ INSERTING ROW IN A VIEW:

We can insert a row in a View in a same way as we do in a table. We can use the INSERT INTO statement of SQL to insert a row in a View.

Syntax:

```
INSERT INTO VIEW_NAME(COLUMN1, COLUMN2,..._)
VALUES (COLUMN1, COLUMN2,..._);
```

VIEW_NAME: NAME OF THE VIEW

Example:

In the below example we will insert a new row(RECORD) in the View **STU_VIEW** which we have created above in the example of “creating views from a single table”.

```
SQL> SELECT * FROM STU_VIEW;
```

S_ID	SNAME	AGE
101	RAM	19
102	LAKSHMAN	18
103	KRISH	20
104	VENKAT	21

➤ Insert a new record into student table using **VIEW (STU_VIEW)**.

```
SQL> INSERT INTO STU_VIEW VALUES (107, 'GANESH', 22);
```

1 row created.

```
SQL> SELECT * FROM STU_VIEW;
```

S_ID	SNAME	AGE
101	RAM	19
102	LAKSHMAN	18
103	KRISH	20
104	VENKAT	21

```
SQL> SELECT * FROM STUDENT;
```

S_ID	SNAME	AGE	ADDRESS
101	RAM	19	VIJ
102	LAKSHMAN	18	HYD
103	KRISH	20	VIZAG
104	VENKAT	21	VIJ
105	SIVA	22	VIZAG
106	SATYA	20	TIRUPATI
107	GANESH	22	-----

7 rows selected.

NOTE: If NOT NULL constraint assign to any column in the table that column **must be initialized** when you are **inserting a new record into table using view**. Otherwise it throws an error. The error is shown bellow.

```
SQL> INSERT INTO STU_VIEW VALUES ('SIVA', 22);
```

```
INSERT INTO STU_VIEW VALUES ('SIVA' ,22 )
```

*

ERROR at line 1:

ORA-01400: cannot insert NULL into ("DSR"."STUDENT"."S_ID")

Deleting a row from a View:

Deleting rows from a view is also as simple as deleting rows from a table. We can use the DELETE statement of SQL to delete rows from a view. Also **deleting a row from a view first delete the row from the actual table and the change is then reflected in the view.**

Syntax:

```
DELETE FROM View_Name WHERE Condition;
```

View_Name: Name of the view from where we want to delete rows

Condition: condition to select the rows.

In this example we will delete the last row from the view **STU_VIEW** which we just added in the above example of inserting rows. Before doing this we create new view;

```
SQL> CREATE OR REPLACE VIEW STU_VIEW AS SELECT S_ID, SNAME FROM
STUDENT WHERE AGE < 23;
```

View created.

```
SQL> SELECT * FROM STU_VIEW;
```

S_ID	SNAME
101	RAM
102	LAKSHMAN
103	KRISH
104	VENKAT
105	SIVA
106	SATYA
107	GANESH

7 rows selected.

Now we are **DELETE** a record (row) from the table using the **STU_VIEW** created above.

```
SQL> DELETE FROM STU_VIEW WHERE SNAME = 'GANESH';
```

1 row deleted.

```
SQL> SELECT * FROM STU_VIEW;
```

S_ID	SNAME
------	-------

```
101 RAM
102 LAKSHMAN
103 KRISH
104 VENKAT
105 SIVA
106 SATYA
```

```
-----
```

6 rows selected.

SQL> SELECT * FROM STUDENT;

S_ID	SNAME	AGE	ADDRESS
101	RAM	19	VIJ
102	LAKSHMAN	18	HYD
103	KRISH	20	VIZAG
104	VENKAT	21	VIJ
105	SIVA	22	VIZAG
106	SATYA	20	TIRUPATI

```
-----
```

6 rows selected.

DELETING(DROP) VIEWS

We have learned about creating a View, but what if a created View is not needed anymore? Obviously we will want to delete it. SQL allows us to delete an existing View. We can delete or drop a View using the DROP statement.

SYNTAX:

DROP VIEW View_name;

View_name: Name of the view which we want to delete.

For Example if we want to delete the view STU_VIEW, We can do this as:

```
SQL> DROP VIEW STU_VIEW;
```

View dropped.

```
SQL> SELECT * FROM STU_VIEW;
```

```
SELECT * FROM STU_VIEW
```

```
*
```

ERROR at line 1:

ORA-00942: table or view does not exist

USES OF A VIEW

A good database should contain views due to the given reasons:

1. Restricting data access -

Views provide an additional level of table security by restricting access to a predetermined set of rows and columns of a table.

2. Hiding data complexity -

A view can hide the complexity that exists in a multiple table join.

3. Simplify commands for the user -

Views allows the user to select information from multiple tables without requiring the users to actually know how to perform a join.

4. Store complex queries -

Views can be used to store complex queries.

5. Rename Columns -

Views can also be used to rename the columns without affecting the base tables provided the number of columns in view must match the number of columns specified in select statement. Thus, renaming helps to to hide the names of the columns of the base tables.

6. Multiple view facility -

Different views can be created on the same table for different users.

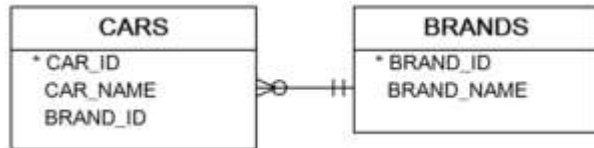
FORCE keyword is used while creating a view, forcefully. This keyword is used to create a View even if the table does not exist. After creating a force View if we create the base table and enter values in it, the view will be automatically updated.

Syntax for forced View is,

```
CREATE OR REPLACE FORCE VIEW VIEW_NAME AS  
SELECT COLUMN_NAME(S)  
FROM TABLE_NAME  
WHERE CONDITION;
```

A view behaves like a table because you can [query](#) data from it. However, you cannot always manipulate data via views.

Let's consider the following database tables:



In database diagram, a car belongs to one brand while a brand has one or many cars. The relationship between brand and car is a one-to-many. The following SQL statements [create](#) the CARS and BRANDS tables; and also [insert sample data](#) into these tables.

```
SQL> CREATE TABLE BRANDS (BRAND_ID NUMBER NOT NULL, BRAND_NAME  
VARCHAR(10) NOT NULL, PRIMARY KEY(BRAND_ID));
```

Table created.

```
SQL> CREATE TABLE CARS(  
    CAR_ID NUMBER NOT NULL,  
    CAR_NAME VARCHAR (15) NOT NULL,  
    BRAND_ID NUMBER NOT NULL,  
    PRIMARY KEY (CAR_ID),  
    FOREIGN KEY (BRAND_ID) REFERENCES BRANDS (BRAND_ID)  
);
```

Table created.

➤ INSERT RECORDS INTO BRANDS TABLE:

```
SQL> INSERT INTO BRANDS VALUES(1, 'AUDI');  
1 row created.
```

```
SQL> INSERT INTO BRANDS VALUES(2, 'BMW');  
1 row created.
```

```
SQL> INSERT INTO BRANDS VALUES(3, 'FORD');  
1 row created.
```

```
SQL> INSERT INTO BRANDS VALUES(4, 'HONDA');
```


1 row created.

```
SQL> INSERT INTO BRANDS VALUES(5, 'TOYOTA');
```

1 row created.

➤ **INSERT RECORDS INTO CARS TABLE:**

```
SQL> INSERT INTO CARS VALUES(1, 'Audi R8 Coupe', 1);
```

1 row created.

```
SQL> INSERT INTO CARS VALUES(2, 'Audi Q2',1);
```

1 row created.

```
SQL> INSERT INTO CARS VALUES(3, 'Audi S1', 1);
```

1 row created.

```
SQL> INSERT INTO CARS VALUES(4, 'BMW 2-serie',2);
```

1 row created.

```
SQL> INSERT INTO CARS VALUES(5, 'BMW i8', 2);
```

1 row created.

```
SQL> INSERT INTO CARS VALUES(6, 'Ford Edge', 3);
```

1 row created.

```
SQL> INSERT INTO CARS VALUES(7, 'Ford Mustang', 3);
```

1 row created.

```
SQL> INSERT INTO CARS VALUES(8, 'Honda S2000', 4);
```

1 row created.

```
SQL> INSERT INTO CARS VALUES(9, 'Honda Legend', 4);
```

1 row created.

```
SQL> INSERT INTO CARS VALUES(10, 'Toyota GT86', 5);
```

1 row created.

```
SQL> INSERT INTO CARS VALUES(11, 'Toyota C-HR', 5);
```

1 row created.

The data tables are:

```
SQL> SELECT * FROM BRANDS;
```

BRAND_ID BRAND_NAME

1 AUDI
2 BMW
3 FORD
4 HONDA
5 TOYOTA

SQL> SELECT * FROM CARS;

CAR_ID CAR_NAME BRAND_ID

1	Audi R8 Coupe	1
2	Audi Q2	1
3	Audi S1	1
4	BMW 2-serie	2
5	BMW i8	2
6	Ford Edge	3
7	Ford Mustang	3
8	Honda S2000	4
9	Honda Legend	4
10	Toyota GT86	5
11	Toyota C-HR	5

11 rows selected.

- Create a view using cars table with CAR_ID and CAR_NAME are the attributes.

SQL> CREATE OR REPLACE VIEW CAR_MASTER AS
SELECT CAR_ID, CAR_NAME FROM CARS;

View created.

```
SQL> SELECT * FROM CAR_MASTER;
```

```
CAR_ID CAR_NAME
```

```
-----
```

```
1 Audi R8 Coupe
```

```
2 Audi Q2
```

```
3 Audi S1
```

```
4 BMW 2-serie
```

```
5 BMW i8
```

```
6 Ford Edge
```

```
7 Ford Mustang
```

```
8 Honda S2000
```

```
9 Honda Legend
```

```
10 Toyota GT86
```

```
11 Toyota C-HR
```

```
11 rows selected.
```

➤ Update the cars table using **CAR_MASTER** view:

```
SQL> UPDATE CAR_MASTER SET CAR_NAME = 'AUDI Q2' WHERE CAR_ID = 2;
```

```
1 row updated.
```

```
SQL> SELECT * FROM CAR_MASTER;
```

```
CAR_ID CAR_NAME
```

```
-----
```

```
1 Audi R8 Coupe
```

```
2 AUDI RS7
```

```
3 Audi S1
```

```
4 BMW 2-serie
```

```
5 BMW i8
```

```
6 Ford Edge
```

```
7 Ford Mustang
```

```
8 Honda S2000
```

```
9 Honda Legend
```

```
10 Toyota GT86
11 Toyota C-HR
11 rows selected.
```

```
SQL> SELECT * FROM CARS;
```

CAR_ID	CAR_NAME	BRAND_ID
1	Audi R8 Coupe	1
2	AUDI RS7	1
3	Audi S1	1
4	BMW 2-serie	2
5	BMW i8	2
6	Ford Edge	3
7	Ford Mustang	3
8	Honda S2000	4
9	Honda Legend	4
10	Toyota GT86	5
11	Toyota C-HR	5

```
11 rows selected.
```

```
*****
```

➤ Update view and the view created by two tables using inner join:

```
SQL> INSERT INTO ALL_CARS VALUES(12, 'Audi A5 Cabrio', 1);
```

1 row created.

```
SQL> SELECT * FROM ALL_CARS;
```

CAR_ID	CAR_NAME	BRAND_ID
1	Audi R8 Coupe	1
2	AUDI Q2	1
3	Audi S1	1
4	BMW 2-serie	2
5	BMW i8	2
6	Ford Edge	3
7	Ford Mustang	3
8	Honda S2000	4
9	Honda Legend	4
10	Toyota GT86	5
11	Toyota C-HR	5
12	Audi A5 Cabrio	1

12 rows selected.

A new row has been inserted into the cars table. This INSERT statement works because Oracle can decompose it to an INSERT statement against the cars table.

The following statement deletes all Honda cars from the cars table via the ALL_CARS view:

```
SQL> DELETE FROM ALL_CARS WHERE BRAND_NAME='HONDA';
```

2 rows deleted.

Oracle has some rules and restrictions that apply to updatable join views. One of them is the concept of key-preserved tables.

A key-preserved table is a base table with a one-to-one row relationship with the rows in the view, via either the primary key or a unique key. In the example above, the cars table is a key-preserved table.

Here are some examples of updatable join view restrictions:

- The SQL statement e.g., **INSERT**, **UPDATE**, and **DELETE**, is only allowed to modify data from a single base table.
- For an INSERT statement, all columns listed in the INTO clause must belong to a key-preserved table.
- For an UPDATE statement, all columns in the SET clause must belong to a key-preserved table.
- For a DELETE statement, if the join results in more than one key-preserved table, the Oracle deletes from the first table in the FROM clause.

Besides these restrictions, Oracle also requires that the defining-query does not contain any of the following elements:

- Aggregate functions e.g., **AVG**, **COUNT**, **MAX**, **MIN**, and **SUM**.
- **DISTINCT** operator.
- **GROUP BY** clause.
- **HAVING** clause.
- Set operators e.g., **UNION**, **UNION ALL**, **INTERSECT**, and **MINUS**.

Find updatable columns of a join view:

To find which column can be updated, inserted, or deleted, you use the `user_updatable_columns` view. The following example shows which column of the `all_cars` view is updatable, insertable, and deletable:

```
SQL> SET LINESIZE 200; //TO SET THE LINE SIZE
```

```
SQL> SELECT * FROM USER_UPDATABLE_COLUMNS WHERE TABLE_NAME =  
'ALL_CARS';
```

OWNER	TABLE_NAME	COLUMN_NAME	UPD	INS	DEL
-----	-----	-----	---	---	---
DSR	ALL_CARS	CAR_ID	YES	YES	YES
DSR	ALL_CARS	CAR_NAME	YES	YES	YES
DSR	ALL_CARS	BRAND_ID	YES	YES	YES

Subqueries with the INSERT Statement

- SQL subquery can also be used with the Insert statement. In the insert statement, data returned from the subquery is used to insert into another table.
- In the subquery, the selected data can be modified with any of the character, date functions.

Syntax:

```
INSERT INTO table_name (column1, column2, column3....)
SELECT * FROM table_name
WHERE VALUE OPERATOR
```

CONSIDER THE FOLLOWING TABLES:

```
SQL> CREATE TABLE PERSONS1 (ID INT NOT NULL, LASTNAME VARCHAR (15),
    FIRSTNAME VARCHAR(15), AGE INT);
```

```
SQL> INSERT INTO PERSONS1 VALUES ( 101, 'ABC', 'DEF',10);
```

1 row created.

```
SQL> INSERT INTO PERSONS1 VALUES(102, 'UVW', 'XYZ', 15);
```

1 row created.

```
SQL> INSERT INTO PERSONS1 VALUES(103, 'PQR', 'STU', 16);
```

1 row created.

```
SQL> SELECT * FROM PERSONS1;
```

ID	LASTNAME	FIRSTNAME	AGE
101	ABC	DEF	10
102	UVW	XYZ	15
103	PQR	STU	16

If we want to insert those orders from 'PERSONS1' table which have the ID LESS THAN 103 into 'PERSON_BKP' table the following SQL can be used:

CREATION OF TABLE PERSON_BKP:

```
SQL> CREATE TABLE PERSON_BKP(ID INT NOT NULL, LASTNAME VARCHAR(15), FIRSTNAME  
    VARCHAR(15), AGE INT);
```

Table created.

```
SQL> SELECT * FROM PERSON_BKP;
```

No rows selected

SQL Code:

```
SQL> INSERT INTO PERSON_BKP SELECT * FROM PERSONS1 WHERE ID < 103;
```

2 rows created.

```
SQL> SELECT * FROM PERSON_BKP;
```

ID	LASTNAME	FIRSTNAME	AGE
101	ABC	DEF	10
102	UVW	XYZ	15

SQL Sub Query

A Subquery is a query within another SQL query and embedded **within the WHERE clause**.

Important Rule:

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 an inner query can be passed to the outer query.**

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

- Compare an expression to the result of the query.
- Determine if an expression is included in the results of the query.

- Check whether the query selects any rows.

1. Subqueries with the Select Statement

SQL subqueries are most frequently used with the Select statement.

Syntax

```
SELECT column_name
FROM table_name
WHERE column_name expression operator
( SELECT column_name from table_name WHERE ... );
```

Consider the following EMP table:

```
SQL> INSERT INTO EMP VALUES (1001, 'RAM', 20, 10000);
```

1 row created.

```
SQL> INSERT INTO EMP VALUES (1002, 'LAKSH', 21, 9000);
```

1 row created.

```
SQL> INSERT INTO EMP VALUES (1003, 'KRISH', 22, 11000);
```

1 row created.

```
SQL> INSERT INTO EMP VALUES (1004, 'VENKAT', 19, 12000);
```

1 row created.

```
SQL> INSERT INTO EMP VALUES (1005, 'SIVA', 23, 9000);
```

1 row created.

```
SQL> INSERT INTO EMP VALUES (1006, 'SATYA', 22, 10000);
```

1 row created

```
SQL> select * from emp;
```

EID	NAME	AGE	SALARY
1001	RAM	20	10000
1002	LAKSH	21	9000
1003	KRISH	22	11000
1004	VENKAT	19	12000
1005	SIVA	23	9000

1006 SATYA	22	10000
------------	----	-------

```
SQL> SELECT * FROM EMP WHERE SALARY>10000;
```

EID	NAME	AGE	SALARY
1003	KRISH	22	11000
1004	VENKAT	19	12000

```
SQL> SELECT EID FROM EMP WHERE SALARY > 10000;
```

EID
1003
1004

```
SQL> SELECT * FROM EMP WHERE EID IN (SELECT EID FROM EMP WHERE SALARY>10000);
```

EID	NAME	AGE	SALARY
1003	KRISH	22	11000
1004	VENKAT	19	12000

EXAMPLE -2:

In this section, you will learn the requirements of using subqueries. We have the following two tables 'student1' and 'marks' with common field 'SID'.

CONSIDER THE TABLES:

```
SQL> CREATE TABLE STUDENT1 (SID INT NOT NULL, NAME VARCHAR (15))
```

Table created.

```
SQL> INSERT INTO STUDENT1 VALUES (101, 'RAM');
```

1 row created.

```
SQL> INSERT INTO STUDENT1 VALUES (102, 'LAKSH');
```

1 row created.

```
SQL> INSERT INTO STUDENT1 VALUES (103, 'KRISH');
```

1 row created.

```
SQL> INSERT INTO STUDENT1 VALUES (104, 'VENKAT');
```

1 row created.

```
SQL> INSERT INTO STUDENT1 VALUES (105, 'SIVA');
```

1 row created.

```
SQL> INSERT INTO STUDENT1 VALUES (106, 'SATYA');
```

1 row created.

```
SQL> SELECT * FROM STUDENT1;
```

SID	NAME
101	RAM
102	LAKSH
103	KRISH
104	VENKAT
105	SIVA
106	SATYA

6 rows selected.

```
SQL> CREATE TABLE MARKS (SID INT NOT NULL, TOTAL_MARKS INT NOT NULL);
```

Table created.

```
SQL> INSERT INTO MARKS VALUES (101, 95);
```

1 row created.

```
SQL> INSERT INTO MARKS VALUES (102, 85);
```

1 row created.

```
SQL> INSERT INTO MARKS VALUES (103, 80);
```

1 row created.

```
SQL> INSERT INTO MARKS VALUES (104, 70);
```

```
1 row created.
```

```
SQL> INSERT INTO MARKS VALUES (105, 75);
```

```
1 row created.
```

```
SQL> INSERT INTO MARKS VALUES (106, 72);
```

```
1 row created.
```

```
SQL> SELECT * FROM MARKS;
```

SID	TOTAL_MARKS
101	95
102	85
103	80
104	70
105	75
106	72

Now we want to write a query to identify all students who get better marks than that of the student who's SID is '102', but we do not know the marks of '102'.

---- To solve the problem, we require two queries. One query returns the marks (stored in Total_marks field) of '102' and a second query identifies the students who get better marks than the result of the first query.

First query:

```
SQL> SELECT * FROM MARKS WHERE SID=102;
```

Query result:

SID	TOTAL_MARKS
102	85

The result of the query is 85.

- Using the result of this query, here we have written another query to identify the students who get better marks than 85. Here is the query:

Second query:

```
SQL> SELECT STUDENT1.SID, STUDENT1.NAME, MARKS.TOTAL_MARKS FROM STUDENT1,  
MARKS WHERE STUDENT1.SID = MARKS.SID AND MARKS.TOTAL_MARKS > 85;
```

SID	NAME	TOTAL_MARKS
101	RAM	95

OR

```
SQL> SELECT A.SID, A.NAME, B.TOTAL_MARKS FROM STUDENT1 A, MARKS B WHERE A.SID  
= B.SID AND B.TOTAL_MARKS>85;
```

Query result:

SID	NAME	TOTAL_MARKS
101	RAM	95

Above two queries identified students who get the better **MARKS** than the student who's SID is '102' (RAM).

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 Code:

```
SQL> SELECT A.SID, A.NAME, B.TOTAL_MARKS FROM STUDENT1 A, MARKS B WHERE A.SID  
= B.SID AND B.TOTAL_MARKS > (SELECT TOTAL_MARKS FROM MARKS WHERE SID = 102);
```

Query result:

SID	NAME	TOTAL_MARKS
101	RAM	95

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 into a subquery. You can use an ORDER BY clause in the main SELECT statement (outer query) which will be the 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.

Scalar Functions

Scalar functions return a single value from an input value. Following are some frequently used Scalar Functions in SQL.

Consider the following EMP table

```
SQL> select * from emp;  
no rows selected
```

```
SQL> INSERT INTO EMP VALUES(1001, 'RAM', 20, 10000);  
1 row created.
```

```
SQL> INSERT INTO EMP VALUES(1002, 'LAKSH', 21, 9000);  
1 row created.
```

```
SQL> INSERT INTO EMP VALUES(1003, 'KRISH',22, 11000);  
1 row created.
```

```
SQL> INSERT INTO EMP VALUES(1004, 'VENKAT', 19, 12000);  
1 row created.
```

```
SQL> INSERT INTO EMP VALUES(1005, 'SIVA', 23, 9000);  
1 row created.
```

```
SQL> insert into emp values(1006, 'satya',22, 12345);  
1 row created.
```

```
SQL> select * from emp;
```

EID	NAME	AGE	SALARY
1001	RAM	20	10000
1002	LAKSH	21	9000
1003	KRISH	22	11000
1004	VENKAT	19	12000
1005	SIVA	23	9000
1006	satya	22	12345

```
6 rows selected.
```

1. UPPER() Function:

UPPER function is used to convert value of string column to Uppercase characters.

Syntax of UPPER,

```
SELECT UPPER(COLUMN_NAME) FROM TABLE_NAME;
```

Using UPPER() function

CONSIDER THE FOLLOWING TABLE:

```
SQL> select * from emp;
```

EID	NAME	AGE	SALARY
1001	RAM	20	10000
1002	LAKSH	21	9000
1003	KRISH	22	11000
1004	VENKAT	19	12000
1005	SIVA	23	9000
1006	satya	22	12345

SQL query for using UPPER is,

```
SQL> SELECT UPPER(NAME) FROM EMP;
```

Result is,

```
UPPER(NAME)
-----
RAM
LAKSH
KRISH
VENKAT
SIVA
SATYA
```

2. LOWER() Function:

LOWER function is used to convert value of string columns to Lowercase characters.

Syntax for LOWER is,

```
SELECT LOWER(COLUMN_NAME) FROM TABLE_NAME;
```

Using LOWER() function

Consider the following EMP table

```
SQL> SELECT * FROM EMP;
```

EID	NAME	AGE	SALARY
1001	RAM	20	10000
1002	LAKSH	21	9000
1003	KRISH	22	11000
1004	VENKAT	19	12000
1005	SIVA	23	9000
1006	satya	22	12345

SQL query for using UPPER is,

```
SQL> SELECT LOWER(NAME) FROM EMP;
```

```
LOWER(NAME)
-----
ram
laksh
krish
venkat
siva
satya
```

3. SUBSTR() Function: Return part of the string.

Syntax for substr() is :

```
SELECT SUBSTR(COLUMN_NAME, STARTING CHARACTER, NO. OF CAHRECTORS) FROM
TABLE_NAME;
```

(OR)

```
SELECT SUBSTR(COLUMN_NAME, START, LENGTH) FROM TABLE_NAME;
```

Consider the following EMP table

```
SQL> SELECT * FROM EMP;
```

EID	NAME	AGE	SALARY
1001	RAM	20	10000
1002	LAKSH	21	9000
1003	KRISH	22	11000
1004	VENKAT	19	12000
1005	SIVA	23	9000
1006	satya	22	12345

SQL query for using **SUBSTR** is,

```
SQL> SELECT SUBSTR(NAME, 2,3) FROM EMP;
```

```
SUB
---
AM
AKS
RIS
ENK
IVA
aty
```

```
SQL> SELECT SUBSTR(NAME, 2,2) FROM EMP;
```

```
SU
--
AM
AK
RI
EN
IV
at
```

```
SQL> SELECT SUBSTR(NAME, 2,4)FROM EMP;
```

```
SUBS
----
AM
AKSH
RISH
ENKA
IVA
atya
```

```
SQL> SELECT SUBSTR(NAME, 1, 3) FROM EMP;
```

```
SUB
---
RAM
LAK
KRI
VEN
SIV
Sat
```

4. REVERSE() Function: RETURNS THE REVERSE ORDER OF A STRING VALUE.

Syntax for REVERSE() is :

SELECT REVERSE(COLUMN_NAME) FROM TABLE_NAME;

Consider the following EMP table

SQL> SELECT * FROM EMP;

EID	NAME	AGE	SALARY
1001	RAM	20	10000
1002	LAKSH	21	9000
1003	KRISH	22	11000
1004	VENKAT	19	12000
1005	SIVA	23	9000
1006	satya	22	12345

SQL query for using REVERSE is,

SQL> SELECT REVERSE(NAME) FROM EMP;

```
REVERSE(NAME)
-----
MAR
HSKAL
HSIRK
TAKNEV
AVIS
aytas
```

5. INITCAP() Function: RETURNS A CHARACTER EXPRESSION, WITH THE FIRST LETTER OF EACH WORD IN UPPERCASE, ALL OTHER LETTERS IN LOWERCASE.

Syntax for INITCAP() is:

SELECT INITCAP(COLUMN_NAME) FROM TABLE_NAME;

Consider the following EMP table

SQL> SELECT * FROM EMP;

EID	NAME	AGE	SALARY
1001	RAM	20	10000
1002	LAKSH	21	9000
1003	KRISH	22	11000
1004	VENKAT	19	12000
1005	SIVA	23	9000
1006	satya	22	12345

SQL query for using INITCAP() is,

```
SQL> SELECT INITCAP(NAME) FROM EMP;
```

```
INITCAP(NAME)
-----
Ram
Laksh
Krish
Venkat
Siva
Satya
```

6. LENGTH() Function: Returns the number of characters of the specified string expression.

Syntax for LENGTH() is:

```
SELECT LENGTH(COLUMN_NAME) FROM TABLE_NAME;
```

Consider the following EMP table

```
SQL> SELECT * FROM EMP;
```

EID	NAME	AGE	SALARY
1001	RAM	20	10000
1002	LAKSH	21	9000
1003	KRISH	22	11000
1004	VENKAT	19	12000
1005	SIVA	23	9000
1006	satya	22	12345

SQL query for using LENGTH() is,

```
SQL> SELECT LENGTH(NAME) FROM EMP;
```

```
LENGTH(NAME)
-----
3
5
5
6
4
5
```

7. RTRIM() Function: Returns a character string after truncating all trailing blanks.

Syntax for RTRIM() is:

```
SELECT RTRIM(COLUMN_NAME) FROM TABLE_NAME;
```

Consider the following EMP table

```
SQL> SELECT * FROM EMP;
```

EID	NAME	AGE	SALARY
1001	RAM	20	10000
1002	LAKSH	21	9000
1003	KRISH	22	11000
1004	VENKAT	19	12000
1005	SIVA	23	9000
1006	satya	22	12345

SQL query for using **RTRIM()** is,

```
SQL> SELECT RTRIM(NAME) FROM EMP;
```

```
RTRIM(NAME)
-----
RAM
LAKSH
KRISH
VENKAT
SIVA
Satya
```

8. LTRIM() Function: Returns a character expression after it removes leading blanks.

Syntax for **LTRIM()** is:

```
SELECT LTRIM(COLUMN_NAME) FROM TABLE_NAME;
```

Consider the following EMP table

```
SQL> SELECT * FROM EMP;
```

EID	NAME	AGE	SALARY
1001	RAM	20	10000
1002	LAKSH	21	9000
1003	KRISH	22	11000
1004	VENKAT	19	12000
1005	SIVA	23	9000
1006	satya	22	12345

Without using LTRIM() the age column displayed in the below format

```
SQL> SELECT AGE FROM EMP;
```

AGE
20
21
22
19
23
22

SQL query for using **LTRIM()** is,

```
SQL> SELECT LTRIM(AGE) FROM EMP;
```

LTRIM(AGE)
20
21
22
19
23
22

```
SQL> SELECT NAME, AGE FROM EMP;
```

NAME	AGE
RAM	20
LAKSH	21
KRISH	22
VENKAT	19
SIVA	23
RAM	24

6 rows selected.

```
SQL> SELECT NAME, RTRIM(AGE) FROM EMP;
```

NAME	RTRIM(AGE)
RAM	20
LAKSH	21
KRISH	22
VENKAT	19
SIVA	23
RAM	24

6 rows selected.

9. **CONCAT()** Function: Returns text string concatenated.

Syntax for **CONCAT()** is:

SELECT CONCAT(COLUMN1, column2) FROM TABLE_NAME;

Consider the following **EMP** table

SQL> SELECT * FROM EMP;

EID	NAME	AGE	SALARY
1001	RAM	20	10000
1002	LAKSH	21	9000
1003	KRISH	22	11000
1004	VENKAT	19	12000
1005	SIVA	23	9000
1006	satya	22	12345

SQL query for using **CONCAT()** is,

SQL> select concat(name, eid) from emp;

CONCAT(NAME,EID)

RAM1001
LAKSH1002
KRISH1003
VENKAT1004
SIVA1005
satya1006

10. **INSTR()** Function: Returns **Location** of a sub-string in a string.

Syntax for **INSTR()** is:

SELECT INSTR(COLUMN_NAME, 'SUB-STRING') FROM TABLE_NAME;

OR

SELECT INSTR (STRING-1, STRING-2) FROM TABLE_NAME;

STRING-1: The string that will be searched

STRING-2: The string to search in STRING-1, if STRING-2 is not found ,
this function return 0

Consider the following EMP table

```
SQL> SELECT * FROM EMP;
```

EID	NAME	AGE	SALARY
1001	RAM	20	10000
1002	LAKSH	21	9000
1003	KRISH	22	11000
1004	VENKAT	19	12000
1005	SIVA	23	9000
1006	satya	22	12345

SQL query for using **INSTR()** is,

```
SQL> select instr(name,'a') from emp;
```

```
INSTR(NAME,'A')
-----
0
0
0
0
0
0
2
```

```
SQL> SELECT INSTR(NAME, 'A') FROM EMP;
```

```
INSTR(NAME,'A')
-----
2
2
0
5
4
0
```

SQL Functions

SQL provides many **built-in functions** to perform operations on data. These functions are useful while performing **mathematical calculations**, string concatenations, sub-strings etc. SQL functions are divided into two categories,

1. Aggregate Functions
2. Scalar Functions

Aggregate Functions

These functions return a single value after performing calculations on a group of values. Following are some of the frequently used Aggregate functions.

Consider the following EMP table:

```
SQL> INSERT INTO EMP VALUES(1001, 'RAM', 20, 10000);
```

```
1 row created.
```

```
SQL> INSERT INTO EMP VALUES(1002, 'LAKSH', 21, 9000);
```

```
1 row created.
```

```
SQL> INSERT INTO EMP VALUES(1003, 'KRISH', 22, 11000);
```

```
1 row created.
```

```
SQL> INSERT INTO EMP VALUES(1004, 'VENKAT', 19, 12000);
```

```
1 row created.
```

```
SQL> INSERT INTO EMP VALUES(1005, 'SIVA', 23, 9000);
```

```
1 row created.
```

```
SQL> SELECT * FROM EMP;
```

EID	NAME	AGE	SALARY
1001	RAM	20	10000
1002	LAKSH	21	9000
1003	KRISH	22	11000
1004	VENKAT	19	12000
1005	SIVA	23	9000

AVG() Function:

Average returns average value after calculating it from values in a **numeric column**.

Its general **syntax** is,

```
SELECT AVG (COLUMN_NAME) FROM TABLE_NAME;
```

CONSIDER THE FOLLOWING EMPLOYEE TABLE:

```
SQL> SELECT * FROM EMP;
```

EID	NAME	AGE	SALARY
1001	RAM	20	10000
1002	LAKSH	21	9000
1003	KRISH	22	11000
1004	VENKAT	19	12000
1005	SIVA	23	9000

```
SQL> SELECT AVG(SALARY) FROM EMP;
```

```
AVG(SALARY)
-----
10200
```

```
SQL> SELECT AVG(AGE) FROM EMP;
```

```
AVG(AGE)
-----
21
```

```
SQL> SELECT AVG(NAME) FROM EMP;
```

```
SELECT AVG(NAME) FROM EMP
*
```

```
ERROR at line 1:
ORA-01722: invalid number
```

COUNT() Function:

Count returns the number of rows present in the table either **based on some condition or without condition**.

Its general **syntax** is,

```
SELECT COUNT (COLUMN_NAME) FROM TABLE_NAME;
```

SQL query to count employees, satisfying specified condition is..

- SQL> SELECT COUNT (NAME) FROM EMP WHERE SALARY=9000;

Result of the above query will be,

```

COUNT(NAME)
-----
                2

```

- SQL> SELECT COUNT(EID) FROM EMP WHERE SALARY=9000;

```

COUNT(EID)
-----
                2

```

- SQL> SELECT COUNT(AGE) FROM EMP WHERE SALARY=9000;

```

COUNT(AGE)
-----
                2

```

- SQL> SELECT COUNT(NAME) FROM EMP WHERE AGE=19;

```

COUNT(NAME)
-----
                1

```

- SQL> SELECT COUNT (NAME) FROM EMP; // COUNT WITHOUT CONDITION

```

COUNT(NAME)
-----
                5

```

Example of COUNT(distinct):

- SQL> SELECT COUNT(DISTINCT SALARY) FROM EMP;

```

COUNT(DISTINCTSALARY)
-----
                        4

```

- SQL> SELECT COUNT(DISTINCT NAME) FROM EMP;

```

COUNT(DISTINCTNAME)
-----
                        5

```

- SQL> SELECT COUNT(DISTINCT AGE) FROM EMP;

```

COUNT(DISTINCTAGE)
-----
                        5

```

- `SQL> SELECT COUNT(*) FROM EMP; //RETURNS TOTAL NUMBER OF RECORDS IN A TABLE`

```

COUNT(*)
-----
        6

```

MAX() Function

MAX function returns maximum value from selected column of the table.

Syntax of MAX function is,

```
SELECT MAX(COLUMN_NAME) FROM TABLE_NAME;
```

Using MAX() function

Consider the ABOVE EMP table

SQL query to find the Maximum salary will be,

- `SQL> SELECT MAX(SALARY) FROM EMP;`

Result of the above query will be,

```

MAX(SALARY)
-----
      12000

```

- `SQL> SELECT MAX(AGE) FROM EMP;`

```

MAX(AGE)
-----
       23

```

- `SQL> SELECT MAX(EID) FROM EMP;`

```

MAX(EID)
-----
      1005

```

- `SQL> SELECT MAX(NAME) FROM EMP;`

```

MAX(NAME)
-----
    VENKAT

```

MIN() Function

MIN function returns minimum value from a selected column of the table.

Syntax for MIN function is,

```
SELECT MIN (COLUMN_NAME) FROM TABLE_NAME;
```

Using MIN() function

Consider the ABOVE EMP table,

SQL query to find minimum salary is,

```
SQL> SELECT MIN(SALARY) FROM EMP;
```

```
MIN(SALARY)
-----
          9000
```

```
SQL> SELECT MIN(AGE) FROM EMP;
```

```
MIN(AGE)
-----
        19
```

```
SQL> SELECT MIN(NAME) FROM EMP;
```

```
MIN(NAME)
-----
      KRISH
```

```
SQL> SELECT MIN (EID) FROM EMP;
```

```
MIN(EID)
-----
      1001
```

SUM() Function

SUM function returns total sum of a selected columns **numeric values**.

Syntax for SUM is,

```
SELECT SUM (COLUMN_NAME) FROM TABLE_NAME;
```

Using SUM() function

Consider the ABOVE EMP table

SQL query to find sum of salaries will be,

```
SQL> SELECT SUM (SALARY) FROM EMP;
```

```
SUM(SALARY)
-----
      51000
```

```
SQL> SELECT SUM (NAME) FROM EMP;
```

```
SELECT SUM (NAME) FROM EMP
      *
```

```
ERROR at line 1:ORA-01722: invalid number
```

DEFINE NEW NAME TO COLUMN:

```
SQL> SELECT MAX(SALARY) AS MAXIMUMPAY FROM EMP;
```

```
MAXIMUMPAY  
-----  
      12000
```

```
SQL> SELECT MIN(SALARY) AS MINIMUMPAY FROM EMP;
```

```
MINIMUMPAY  
-----  
      9000
```


SQL Set Operation

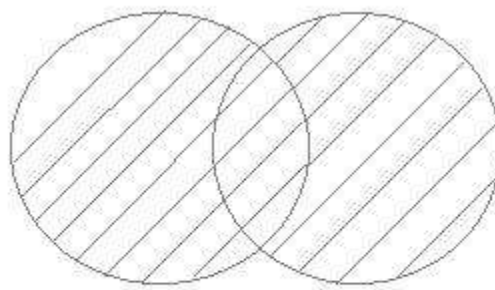
The SQL Set operation is used to combine the two or more SQL **SELECT** statements.

Types of Set Operation

1. Union
2. UnionAll
3. Intersect
4. Minus

1. Union

- The SQL Union operation is used to combine the result of **two or more SQL SELECT queries**.
- In the union operation, all the number of datatype and **columns must be same** in both the tables on which UNION operation is being applied.
- The union operation eliminates the **duplicate rows** from its **resultset**.
- **UNION** is used to combine the results of two or more **SELECT** statements. However it will eliminate duplicate rows from its resultset. In case of union, number of columns and datatype must be same in both the tables, on which UNION operation is being applied.



Syntax:

```
SELECT column_name FROM table1
UNION
SELECT column_name FROM table2;
```

CONSIDER THE FOLLOWING TABLES:

```
SQL> CREATE TABLE TABLE1(ID INT, NAME VARCHAR(20));
```

Table created.

```
SQL> INSERT INTO TABLE1 VALUES(1, 'RAM');
```

1 row created.

```
SQL> INSERT INTO TABLE1 VALUES(2, 'LAKSH');
```

1 row created.

```
SQL> INSERT INTO TABLE1 VALUES(3, 'KRISH');
```

1 row created.

```
SQL> SELECT * FROM TABLE1;
```

ID	NAME
1	RAM
2	LAKSH
3	KRISH

```
SQL> CREATE TABLE TABLE2 (ID INT, NAME VARCHAR(15));
```

Table created.

```
SQL> INSERT INTO TABLE2 VALUES (3, 'KRISH');
```

1 row created.

```
SQL> INSERT INTO TABLE2 VALUES (4, 'SIVA');
```

1 row created.

```
SQL> INSERT INTO TABLE2 VALUES (5, 'VENKAT');
```

1 row created.

```
SQL> SELECT * FROM TABLE2; //TABLE--2
```

ID	NAME
3	KRISH
4	SIVA

5 VENKAT

```
SQL> SELECT * FROM TABLE1; //TABLE--1
```

ID	NAME
1	RAM
2	LAKSH
3	KRISH

Union SQL query will be:

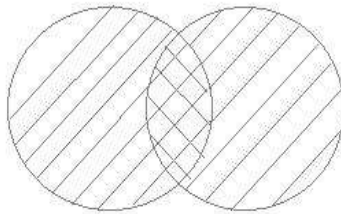
```
SQL> SELECT * FROM TABLE1 UNION SELECT * FROM TABLE2;
```

ID	NAME
1	RAM
2	LAKSH
3	KRISH
4	SIVA
5	VENKAT

2. Union All

Union All operation is equal to the Union operation. It returns the set without removing duplication and sorting the data.

This operation is similar to Union. But it also shows the duplicate rows.



Syntax:

```
SELECT column_name FROM table1
UNION ALL
SELECT column_name FROM table2;
```

Example: Using the above TABLE1 and TABLE2 table.

```
SQL> SELECT * FROM TABLE1;
```

ID	NAME
1	RAM
2	LAKSH
3	KRISH

```
SQL> SELECT * FROM TABLE2;
```

ID	NAME
3	KRISH
4	SIVA
5	VENKAT

Union All query will be like:

```
SQL> SELECT * FROM TABLE1  
      UNION ALL  
      SELECT * FROM TABLE2;
```

The resultset table will look like:

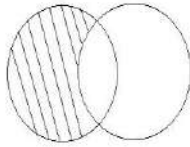
ID	NAME
1	RAM
2	LAKSH
3	KRISH
3	KRISH
4	SIVA
5	VENKAT

6 rows selected.

```
RENAME THE TABLE NAME: SQL> ALTER TABLE FIRST RENAME TO TABLE1;
```

MINUS

The Minus operation combines results of two **SELECT** statements and return only those in the final result, which belongs to the first set of the result.



- It combines the result of two **SELECT** statements. Minus operator is used to display the rows which are present in the first query but absent in the second query.
- It has no duplicates and data arranged in ascending order by default.

Syntax:

```
SELECT column_name FROM table1  
  
MINUS  
  
SELECT column_name FROM table2;
```

CONSIDER THE FOLLOWING TABLES:

```
SQL> CREATE TABLE FIRST(ID INT, NAME VARCHAR(20));
```

Table created.

```
SQL> INSERT INTO FIRST VALUES(1, 'RAM');
```

1 row created.

```
SQL> INSERT INTO FIRST VALUES(2, 'LAKSH');
```

1 row created.

```
SQL> INSERT INTO FIRST VALUES(3, 'KRISH');
```

1 row created.

```
SQL> SELECT * FROM FIRST;
```

```
   ID NAME
```

```
-----
```

```
1 RAM
2 LAKSH
3 KRISH
```

```
SQL> CREATE TABLE SECOND (ID INT, NAME VARCHAR(15));
```

Table created.

```
SQL> INSERT INTO SECOND VALUES (3, 'KRISH');
```

1 row created.

```
SQL> INSERT INTO SECOND VALUES (4, 'SIVA');
```

1 row created.

```
SQL> INSERT INTO SECOND VALUES (5, 'VENKAT');
```

1 row created.

```
SQL> SELECT * FROM SECOND; //TABLE--2
```

```
      ID NAME
-----
      3 KRISH
      4 SIVA
      5 VENKAT
```

```
SQL> SELECT * FROM FIRST; //TABLE--1
```

```
      ID NAME
-----
      1 RAM
      2 LAKSH
      3 KRISH
```

Example

Using the above First and Second table.

Minus query will be:

```
SQL> SELECT * FROM FIRST
      MINUS
      SELECT * FROM SECOND;
```

ID NAME

1 RAM

2 LAKSH

```
SQL> SELECT * FROM SECOND  
MINUS
```

```
SELECT * FROM FIRST;
```

The resultset table will look like:

ID NAME

4 SIVA

5 VENKAT

INTERSECT

- It is used to **combine two SELECT statements**. The Intersect operation **returns the common rows** from both the SELECT statements.
- In the Intersect operation, **the number of datatype and columns must be the same**.
- It **has no duplicates** and it **arranges the data in ascending order by default**.
- Intersect operation is used to combine two **SELECT statements**, but it **only returns the records which are common from both SELECT statements**. In case of **Intersect** the number of columns and datatype **must be SAME**.

Syntax

```
SELECT column_name FROM FIRST
INTERSECT
SELECT column_name FROM SECOND;
```

CONSIDER THE FOLLOWING TABLES:

```
SQL> CREATE TABLE FIRST (ID INT, NAME VARCHAR(20));
```

Table created.

```
SQL> INSERT INTO FIRST VALUES (1, 'RAM');
```

1 row created.

```
SQL> INSERT INTO FIRST VALUES (2, 'LAKSH');
```

1 row created.

```
SQL> INSERT INTO FIRST VALUES (3, 'KRISH');
```

1 row created.

```
SQL> SELECT * FROM FIRST;
```

```

ID NAME
-----
1  RAM
2  LAKSH
3  KRISH
```



```
SQL> CREATE TABLE SECOND (ID INT, NAME VARCHAR (15));
```

Table created.

```
SQL> INSERT INTO SECOND VALUES (3, 'KRISH');
```

1 row created.

```
SQL> INSERT INTO SECOND VALUES (4, 'SIVA');
```

1 row created.

```
SQL> INSERT INTO SECOND VALUES (5, 'VENKAT');
```

1 row created.

```
SQL> SELECT * FROM SECOND; //TABLE--2
```

ID	NAME
3	KRISH
4	SIVA
5	VENKAT

```
SQL> SELECT * FROM FIRST; //TABLE--1
```

ID	NAME
1	RAM
2	LAKSH
3	KRISH

Example:

Using the above First and Second table.

Intersect query will be:

```
SQL> SELECT * FROM FIRST  
INTERSECT  
SELECT * FROM SECOND;
```

ID	NAME
3	KRISH

PL/SQL Trigger

Triggers are the SQL codes that are automatically executed in response to certain events on a particular table.

Trigger is invoked by Oracle engine automatically whenever a specified event occurs. Trigger is stored into database and invoked repeatedly, when specific condition match.

Triggers are stored programs, which are automatically executed or fired when some event occurs.

Triggers are written to be executed in response to any of the following events.

- A database manipulation (DML) statement (DELETE, INSERT, or UPDATE).
- A database definition (DDL) statement (CREATE, ALTER, or DROP).
- A database operation (SERVERERROR, LOGON, LOGOFF, STARTUP, or SHUTDOWN).

Triggers could be defined on the table, view, schema, or database with which the event is associated.

Advantages of Triggers

These are the following advantages of Triggers:

- Trigger generates some derived column values automatically
- Enforces referential integrity
- Event logging and storing information on table access
- Synchronous (existing) replication of tables
- Imposing security authorizations
- Preventing invalid transactions

Creating a trigger:

Syntax for creating trigger:

```
CREATE [OR REPLACE ] TRIGGER trigger_name
{BEFORE | AFTER | INSTEAD OF }
{INSERT [OR] | UPDATE [OR] | DELETE}
[OF col_name]
ON table_name
[REFERENCING OLD AS o NEW AS n]
[FOR EACH ROW]
WHEN (condition)
DECLARE
    Declaration-statements
BEGIN
    Executable-statements
EXCEPTION
    Exception-handling-statements
END; /
```

Here,

- **CREATE [OR REPLACE] TRIGGER trigger_name:** It creates or replaces an existing trigger with the trigger_name.
- **{BEFORE | AFTER | INSTEAD OF}:** This specifies when the trigger would be executed. The INSTEAD OF clause is used for creating trigger on a view.
- **{INSERT [OR] | UPDATE [OR] | DELETE}:** This specifies the DML operation.
- **[OF col_name]:** This specifies the column name that would be updated.
- **[ON table_name]:** This specifies the name of the table associated with the trigger.
- **[REFERENCING OLD AS o NEW AS n]:** This allows you to refer new and old values for various DML statements, like INSERT, UPDATE, and DELETE.
- **[FOR EACH ROW]:** This specifies a row level trigger, i.e., the trigger would be executed for each row being affected. Otherwise the trigger

will execute just once when the SQL statement is executed, which is called a table level trigger.

- **WHEN (condition):** This provides a condition for rows for which the trigger would fire. This clause is valid only for row level triggers.

```
DECLARE --optional
    <declarations>

BEGIN    --mandatory
    <executable statements. At least one executable statement is mandatory>

EXCEPTION --optional
    <exception handles>

END;    --mandatory
/
```

PL/SQL Trigger Example

Let's take a simple example to demonstrate the trigger. In this example, we are using the following **EMPLOYEE** table:

Create table and have records:

```
SQL> CREATE TABLE EMPLOYEE (ID INT, NAME VARCHAR(15), AGE INT, ADDRESS
VARCHAR (15), SALARY INT);
```

Table created.

```
SQL> INSERT INTO EMPLOYEE VALUES (111, 'RAM', 22, 'DELHI', 10000);
```

1 row created.

```
SQL> INSERT INTO EMPLOYEE VALUES (222, 'LAKSH', 20, 'DELHI', 12000);
```

1 row created.

```
SQL> INSERT INTO EMPLOYEE VALUES (333, 'VENKAT', 23, 'MUMBAI', 11000);
```

1 row created.

```
SQL> INSERT INTO EMPLOYEE VALUES (444, 'KRISH', 21, 'HYD', 14000);
```

1 row created.

```
SQL> INSERT INTO EMPLOYEE VALUES (555, 'SIVA', 25, 'BANGLORE', 13000);
```

1 row created.

```
SQL> INSERT INTO EMPLOYEE VALUES (666, 'SATYA', 24, 'CHENNAI', 15000);
```

1 row created.

```
SQL> SELECT * FROM EMPLOYEE;
```

ID	NAME	AGE	ADDRESS	SALARY
111	RAM	22	DELHI	10000
222	LAKSH	20	DELHI	12000
333	VENKAT	23	MUMBAI	11000
444	KRISH	21	HYD	14000
555	SIVA	25	BANGLORE	13000
666	SATYA	24	CHENNAI	15000

➤ Create trigger:

Let's take a program to create a row level trigger for the EMPLOYEE table that would fire for INSERT or UPDATE or DELETE operations performed on the EMPLOYEE table. This trigger will display the salary difference between the old values and new values:

```
SQL> CREATE OR REPLACE TRIGGER display_salary_changes
      BEFORE DELETE OR INSERT OR UPDATE ON EMPLOYEE
      FOR EACH ROW
      WHEN (NEW.ID > 0)
      DECLARE
          sal_diff number;
      BEGIN
          sal_diff := :NEW.salary - :OLD.salary;
          dbms_output.put_line('Old salary: ' || :OLD.salary);
```

```

        dbms_output.put_line('New salary: ' || :NEW.salary);
        dbms_output.put_line('Salary difference: ' || sal_diff);
    END;
/

```

After the execution of the above code at SQL Prompt, it produces the following result.

Trigger created.

Check the salary difference by procedure:

Use the following code to get the old salary, new salary and salary difference after the trigger created.

```

SQL> SET SERVEROUTPUT ON;
SQL> DECLARE
    total_rows number(2);
BEGIN
    UPDATE employee SET salary=salary+500;
    IF sql%notfound THEN
        dbms_output.put_line('no employee updated');
    ELSIF sql%found THEN
        total_rows:=sql%rowcount;
        dbms_output.put_line(total_rows || 'employee updated');
    END IF;
END;
/

```

Old salary: 10500

New salary: 11000

Salary difference: 500

Old salary: 12500

New salary: 13000

Salary difference: 500

Old salary: 11500
New salary: 12000
Salary difference: 500
Old salary: 14500
New salary: 15000
Salary difference: 500
Old salary: 13500
New salary: 14000
Salary difference: 500
Old salary: 15500
New salary: 16000
Salary difference: 500
6 employee updated

PL/SQL procedure successfully completed.

Note: As many times you executed this code, the old and new both salary is incremented by 500 and hence the salary difference is always 500.

After the execution of above code again, you will get the following result.

```
SQL> SET SERVEROUTPUT ON;
```

```
SQL> DECLARE
```

```
    total_rows number(2);
```

```
    BEGIN
```

```
        UPDATE employee
```

```
        SET salary=salary+500;
```

```
        IF sql%notfound THEN
```

```
            dbms_output.put_line('no employee updated');
```

```
        ELSIF sql%found THEN
```

```
            total_rows:=sql%rowcount;
```

```
            dbms_output.put_line(total_rows || 'employees updated');
```

```
        END IF;
```

```
    END;
```

```
    /
```

Old salary: 11000
New salary: 11500
Salary difference: 500
Old salary: 13000
New salary: 13500
Salary difference: 500
Old salary: 12000
New salary: 12500
Salary difference: 500
Old salary: 15000
New salary: 15500
Salary difference: 500
Old salary: 14000
New salary: 14500
Salary difference: 500
Old salary: 16000
New salary: 16500
Salary difference: 500
6 employees updated

PL/SQL procedure successfully completed.

Oracle AFTER INSERT trigger

In this tutorial you will learn how to create an **AFTER INSERT trigger** (after inserting) in Oracle with syntax and examples.

Oracle executes (excites) the AFTER INSERT trigger after executing the INSERT operator.

SYNTAX OF THE AFTER INSERT TRIGGER

```
CREATE [ OR REPLACE ] TRIGGER name_trigger
AFTER INSERT
ON table_name
[ FOR EACH ROW ]
DECLARE
- variable declaration
BEGIN
- trigger code
EXCEPTION
WHEN ...
- emergency handling
END;
```

Parameters or arguments

- **Trigger_name** - The name of the trigger to be created.
- **AFTER INSERT** - indicates that the trigger is triggered after the INSERT operator is executed.
- **table_name** - The name of the table for which the Trigger was created.

Restrictions

- You cannot create a trigger in views.
- You cannot update :NEW (new) values.
- You cannot update :OLD (old) values.

Example

Here we are considering two tables SALES and PRODUCTSTOCK

SALES TABLE:

```
SQL> CREATE TABLE SALES (PROD_ID INT, CUSTOMERNAME VARCHAR(15));
```

Table created.

```
SQL> INSERT INTO SALES VALUES (1, 'RAM');
```

1 row created.

```
SQL> SELECT * FROM SALES;
```

```
PROD_ID CUSTOMERNAME
```

```
-----
```

```
1 RAM
```

PRODUCTSTOCK TABLE:

```
SQL> CREATE TABLE PRODUCTSTOCK (PROD_ID INT, PRODUCTNAME VARCHAR (15),  
TOTAL INT);
```

Table created.

```
SQL> INSERT INTO PRODUCTSTOCK VALUES(1, 'LUX', 4);
```

1 row created.

```
SQL> INSERT INTO PRODUCTSTOCK VALUES(2, 'DOVE', 7);
```

1 row created.

```
SQL> INSERT INTO PRODUCTSTOCK VALUES(3, 'WILD STONE', 4);
```

1 row created.

```
SQL> INSERT INTO PRODUCTSTOCK VALUES(4, 'HIMALAYA', 9);
```

1 row created.

```
SQL> SELECT * FROM PRODUCTSTOCK;
```

```
PROD_ID PRODUCTNAME      TOTAL
```

```
-----
```

```
1 LUX 4
```

```
2 DOVE 7
```

```
3 WILD STONE 4
```

```
4 HIMALAYA 9
```

```
SQL> CREATE OR REPLACE TRIGGER UPDATEPRODUCTSTOCK
      AFTER INSERT ON SALES
      FOR EACH ROW
      BEGIN
      UPDATE PRODUCTSTOCK SET TOTAL = TOTAL - 1 WHERE PROD_ID =:
      NEW.PROD_ID;
      END;
```

PL/SQL WHILE Loop

The WHILE loop syntax

Here is the syntax for the WHILE loop statement:

```
WHILE condition
LOOP
    statements;
END LOOP;
```

The condition in the WHILE is a Boolean expression that evaluates to TRUE, FALSE or NULL.

The WHILE loop statement continues to execute the statements between the LOOP and END LOOP as long as the condition in the WHILE clause evaluates to TRUE.

PL/SQL evaluates the condition in the WHILE clause before each loop iteration. If the condition is TRUE, then the loop body executes. In case it is FALSE or NULL, the loop terminates.

If the condition is FALSE before entering the loop, the WHILE loop does not execute at all. This behavior is different from the [LOOP](#) statement whose loop body always executes once.

To terminate the loop prematurely, you use an EXIT or EXIT WHEN statement.

PL/SQL WHILE loop examples

Let's take some examples of using the WHILE loop statement to see how it works.

A) Simple WHILE loop example

The following example illustrates how to use the WHILE loop statement:

```
DECLARE
    n_counter NUMBER := 1;
BEGIN
    WHILE n_counter <= 5
```

```

LOOP
    DBMS_OUTPUT.PUT_LINE( 'Counter : ' || n_counter );
    --n_counter := n_counter + 1; -- like update expression
END LOOP;
END;

```

Here is the output:

```

Counter : 1
Counter : 2
Counter : 3
Counter : 4
Counter : 5

```

In this example:

- First, the counter was initialized to zero.
- Second, the condition in the WHILE clause was evaluated before each loop iteration.
- Third, inside the loop body, the counter was increased by one in each loop iteration. After five iterations, the condition was FALSE that caused the loop terminated.

B) WHILE loop example terminated by EXIT WHEN statement

The following example is the same as the one above except that it has an additional EXITWHEN statement.

```

DECLARE
    n_counter NUMBER := 1;
BEGIN
    WHILE n_counter <= 5
    LOOP
        DBMS_OUTPUT.PUT_LINE( 'Counter : ' || n_counter );
        n_counter := n_counter + 1;
        EXIT WHEN n_counter = 3;
    END LOOP;

```

END;

The following is the output:

Counter : 1

Counter : 2

The condition in the EXIT WHEN clause evaluated to true when the counter is three. Therefore, the loop body only executed two times before it terminated.

```
Int I=1;
While(i<=5)
{
    Printf("%d", i);
    I++;
}
```

PL/SQL NULL Statement

Introduction to PL/SQL NULL statement

The PL/SQL NULL statement has the following format:

```
NULL;
```

The NULL statement is a NULL keyword followed by a semicolon (;). The NULL statement does nothing except that it passes control to the next statement.

The NULL statement is useful to:

- Improve code readability
- Provide a target for a [GOTO](#) statement
- Create placeholders for subprograms

Improving code readability

The following code sends an email to employees whose job titles are Sales Representative.

```
IF job_title = 'Sales Representative' THEN
    send_email;
END IF;
```

What should the program do for employees whose job title are not Sales Representative? You might assume that it should do nothing. Because this logic is not explicitly mentioned in the code, you may wonder if it misses something else.

To make it more clear, you can add a comment. For example

```
-- Send email to only Sales Representative,
-- for other employees, do nothing
IF job_title = 'Sales Representative' THEN
    send_email;
END IF;
```

Or you can add an [ELSE](#) clause that consists of a NULL statement to clearly state that no action is needed for other employees.

```
IF job_title = 'Sales Representative' THEN
    send_email;
ELSE
    NULL;
END IF;
```

Similarly, you can use a NULL statement in the ELSE clause of a simple [CASE](#) statement as shown in the following example:

```
DECLARE
    n_credit_status VARCHAR2( 50 );
BEGIN
    n_credit_status := 'BLOCK';

    CASE n_credit_status
    WHEN 'BLOCK' THEN
        request_for_aproval;
    WHEN 'WARNING' THEN
        send_email_to_accountant;
    ELSE
        NULL;
    END CASE;
END;
```

In this example, if the credit status is not blocked or warning, the program does nothing.

Providing a target for a GOTO statement

When using a [GOTO](#) statement, you need to specify a label followed by at least one executable statement.

The following example uses a GOTO statement to quickly move to the end of the program if no further processing is required:

```
DECLARE
    b_status BOOLEAN;
BEGIN
    IF b_status THEN
        GOTO end_of_program;
    END IF;
    -- further processing here
    -- ...
    <<end_of_program>>
    NULL;
END;
```

Note that an error will occur if you don't have the [NULL](#) statement after the end_of_program label.

PL/SQL LOOP

PL/SQL LOOP syntax

The PL/SQL LOOP statement has the following structure:

```
<<label>> LOOP
    statements;
END LOOP loop_label;
```

This structure is the most basic of all the loop constructs including `FOR LOOP` and [WHILE LOOP](#). This basic LOOP statement consists of a LOOP keyword, a body of executable code, and the END LOOP keywords.

The LOOP statement executes the statements in its body and returns control to the top of the loop. Typically, the body of the loop contains at least one EXIT or EXIT WHEN statement for terminating the loop. Otherwise, the loop becomes an infinite loop.

The LOOP statement can have an optional label that appears at the beginning and the end of the statement.

It is a good practice to use the LOOP statement when:

- You want to execute the loop body at least once.
- You are not sure the number of times you want the loop to execute.

EXIT statement

The EXIT statement allows you to unconditionally exit the current iteration of a loop.

```
LOOP
    EXIT;
END LOOP;
```

Typically, you use the EXIT statement with an [IF](#) statement to terminate a loop when a condition is true:

```
LOOP
```

```
    IF condition THEN
```

```
        EXIT;
```

```
    END IF;
```

```
END LOOP;
```

The following example illustrates how to use the LOOP statement to execute a sequence of code and EXIT statement to terminate the loop.

```
DECLARE
```

```
    l_counter NUMBER := 0;
```

```
BEGIN
```

```
    LOOP
```

```
        l_counter := l_counter + 1;
```

```
        IF l_counter > 3 THEN
```

```
            EXIT;
```

```
        END IF;
```

```
        dbms_output.put_line( 'Inside loop: ' || l_counter ) ;
```

```
    END LOOP;
```

```
    -- control resumes here after EXIT
```

```
    dbms_output.put_line( 'After loop: ' || l_counter );
```

```
END;
```

Here is the output:

```
Inside loop: 1
```

```
Inside loop: 2
```

```
Inside loop: 3
```

```
After loop: 4
```

The following explains the logic of the code:

- First, declare and initialize a [variable](#) l_counter to zero.
- Second, increase the l_counter by one inside the loop and exit the loop if the l_counter is greater than three. If

the l_counter is less than or equal three, show the l_counter value. Because the initial value of l_counter is zero, the code in the body of the loop executes three times before it is terminated.

- Third, display the value of the l_counter after the loop.

EXIT WHEN statement

The EXIT WHEN statement has the following syntax:

```
EXIT WHEN condition;
```

The EXIT WHEN statement exits the current iteration of a loop when the condition in the WHEN clause is TRUE. Essentially, the EXIT WHEN statement is a combination of an EXIT and an IF THEN statement.

Each time the control reaches the EXIT WHEN statement, the condition is evaluated. If the condition evaluates to TRUE, then the loop terminates. Otherwise, the EXIT WHEN clause does nothing. Inside the loop body, you must make the condition TRUE at some point to prevent an infinite loop.

The following example uses the EXIT WHEN statement to terminate a loop.

```
DECLARE
```

```
    l_counter NUMBER := 0;
```

```
BEGIN
```

```
    LOOP
```

```
        l_counter := l_counter + 1;
```

```
        EXIT WHEN l_counter > 3;
```

```
        dbms_output.put_line( 'Inside loop: ' || l_counter );
```

```
    END LOOP;
```

```
    -- control resumes here after EXIT
```

```
    dbms_output.put_line( 'After loop: ' || l_counter );
```

```
END;
```

Notice that this example is logically equivalent to the example that uses the EXIT statement above.

Nested loops

It is possible to nest a LOOP statement within another LOOP statement as shown in the following example:

```
DECLARE
    l_i NUMBER := 0;
    l_j NUMBER := 0;
BEGIN
    <<outer_loop>>
    LOOP
        l_i := l_i + 1;
        EXIT outer_loop WHEN l_i > 2;
        dbms_output.put_line('Outer counter ' || l_i);
        -- reset inner counter
        l_j := 0;
        <<inner_loop>> LOOP
            l_j := l_j + 1;
            EXIT inner_loop WHEN l_j > 3;
            dbms_output.put_line(' Inner counter ' || l_j);
        END LOOP inner_loop;
    END LOOP outer_loop;
END;
```

Here is the output: Outer counter 1

Inner counter 1

Inner counter 2

Inner counter 3

Outer counter 2

Inner counter 1

Inner counter 2

Inner counter 3

Introduction to PL/SQL GOTO statement

The GOTO statement allows you to transfer control to a labeled block or statement. The following illustrates the syntax of the GOTO statement:

```
GOTO label_name;
```

The label_name is the name of a label that identifies the target statement. In the program, you surround the label name with double enclosing angle brackets as shown below:

```
<<label_name>>;
```

When PL/SQL encounters a GOTO statement, it transfers control to the first executable statement after the label.

PL/SQL GOTO statement example

The following shows an example of using the GOTO statements.

```
BEGIN
```

```
  GOTO second_message;
```

```
  <<first_message>>
```

```
    DBMS_OUTPUT.PUT_LINE( 'Hello' );
```

```
    GOTO the_end;
```

```
  <<second_message>>
```

```
    DBMS_OUTPUT.PUT_LINE( 'PL/SQL GOTO Demo' );
```

```
    GOTO first_message;
```

```
  <<the_end>>
```

```
    DBMS_OUTPUT.PUT_LINE( 'and good bye...' );
```

```
END;
```

The output is:

```
    PL/SQL GOTO Demo
```

```
    Hello
```

```
    and good Bye...
```

The following explains the sequence of the `block` in detail:

- First, the GOTO `second_message` statement is encountered, therefore, the control is passed to the statement after the `second_message` label.
- Second, the GOTO `first_message` is encountered, so the control is transferred to the statement after the `first_message` label.
- Third, the GOTO `the_end` is reached, hence the control is passed to the statement after the `the_end` label.

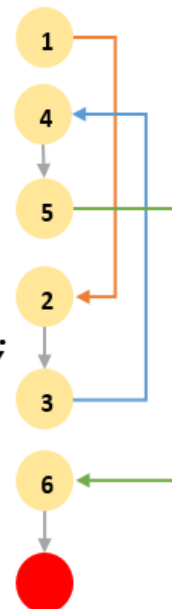
The picture below illustrates the sequence:

```
BEGIN
  GOTO second_message;

<first_message>
  DBMS_OUTPUT.PUT_LINE('Hello');
  GOTO the_end;

<second_message>;
  DBMS_OUTPUT.PUT_LINE('PL/SQL GOTO Demo');
  GOTO first_message;

<the_end>
  DBMS_OUTPUT.PUT_LINE('and good bye...');
END;
```



PL/SQL FOR LOOP

Introduction to PL/SQL FOR LOOP statement

PL/SQL FOR LOOP executes a sequence of statements a specified number of times. The PL/SQL FOR LOOP statement has the following structure:

```
FOR index IN lower_bound .. upper_bound
LOOP
    statements;
END LOOP;
```

The index is an implicit variable. It is local to the FOR LOOP statement. In other words, you cannot reference it outside the loop.

Inside the loop, you can reference index but you cannot change its value. After the FOR LOOP statement executes, the index becomes undefined.

Both lower_bound and upper_bound are numbers or expressions that evaluate to numbers. The lower_bound and upper_bound are evaluated once when the FOR LOOP statement starts. Their values are stored as temporary PLS_INTEGER values. The results of lower_bound and upper_bound are rounded to the nearest integer if necessary.

If you modify the values of lower_bound or upper_bound inside the loop, the change will have no effect because they are evaluated once only before the first loop iteration starts.

Typically, lower_bound is less than upper_bound. In this case, index is set to lower_bound, the statements execute, and control returns to the top of the loop, where index is compared to upper_bound. If index is less than upper_bound, index is

incremented by one, the statements execute, and control again returns to the top of the loop. When index is greater than upper_bound, the loop terminates, and control transfers to the statement after the FOR LOOP statement.

If lower_bound is equal to upper_bound, the statements execute only once. When lower_bound is greater than upper_bound, the statements do not execute at all.

PL/SQL FOR LOOP examples

Let's take some examples of using the FOR LOOP statement to understand how it works.

A) Simple PL/SQL FOR LOOP example

In this example, the loop index is l_counter, lower_bound is one, and upper_bound is five. The loop shows a list of integers from 1 to 5.

```
BEGIN
  FOR l_counter IN 1..5
  LOOP
    DBMS_OUTPUT.PUT_LINE( l_counter );
  END LOOP;
END;
```

Here is the result:

```
1
2
3
4
5
```

B) Simulating STEP clause in FOR LOOP statement

The loop index is increased by one after each loop iteration and you cannot change the increment e.g., two, three and four. However, you can use an additional variable to simulate the increment by two, three, four, etc., as shown in the example below:

```
DECLARE
    l_step PLS_INTEGER := 2;
BEGIN
    FOR l_counter IN 1..5
    LOOP
        dbms_output.put_line (l_counter*l_step);
    END LOOP;
END;
```

Result:

```
2
4
6
8
10
```

The result shows that, after each loop iteration, the output number is incremented by two instead of one.

C) Referencing variable with the same name as the loop index

Consider the following example:

```
DECLARE
    l_counter PLS_INTEGER := 10;
BEGIN
    FOR l_counter IN 1.. 5 loop
        DBMS_OUTPUT.PUT_LINE (l_counter);
    end loop;
```

```
-- after the loop
DBMS_OUTPUT.PUT_LINE (l_counter);
END;
```

Result:

```
1
2
3
4
5
10
```

In this example, we had a variable named `l_counter`, which is also the name of the index. The result shows that `l_counter` in the FOR loop hides the variable `l_counter` declared in the enclosing block.

To reference the variable `l_counter` inside the loop, you must qualify it using a block label as shown below:

<<outer>>

```
DECLARE
    l_counter PLS_INTEGER := 10;
BEGIN
    FOR l_counter IN 1.. 5 loop
        DBMS_OUTPUT.PUT_LINE ('Local counter:' || l_counter);
        outer.l_counter := l_counter;
    end loop;
    -- after the loop
    DBMS_OUTPUT.PUT_LINE ('Global counter' || l_counter);
END outer;
```

D) Referencing loop index outside the FOR LOOP

The following example causes an error because it references the loop index, which is undefined, outside the FOR LOOP statement.

```
BEGIN
  FOR l_index IN 1..3 loop
    DBMS_OUTPUT.PUT_LINE (l_index);
  END LOOP;
  -- referencing index after the loop
  DBMS_OUTPUT.PUT_LINE (l_index);
END;
```

Oracle issued the following error:

PLS-00201: identifier 'L_INDEX' must be declared

FOR LOOP with REVERSE keyword

The following shows the structure of the FOR LOOP statement with REVERSE keyword:

```
FOR index IN REVERSE lower_bound .. upper_bound
  LOOP
    statements;
  END LOOP;
```

With the REVERSE keyword, the index is set to upper_bound and decreased by one in each loop iteration until it reaches lower_bound.

See the following example:

```
BEGIN
  FOR l_counter IN REVERSE 1..3
  LOOP
    DBMS_OUTPUT.PUT_LINE( l_counter );
  END LOOP;
END;
```

Result:

3

2

1

Without the REVERSE keyword, the output will be:

1

2

3

What is PL/SQL block?

In PL/SQL, the code is not executed in single line format, but it is always executed by grouping the code into a single element called Blocks. In this tutorial, you are going to learn about these blocks.

Blocks contain both PL/SQL as well as SQL instruction. All these instruction will be executed as a whole rather than executing a single instruction at a time.

PL/SQL blocks have a pre-defined structure in which the code is to be grouped. Below are different sections of PL/SQL blocks.

1. Declaration section
2. Execution section
3. Exception-Handling section

Declaration Section

This is the first section of the PL/SQL blocks. This section is an optional part. This is the section in which the declaration of variables, cursors, exceptions, subprograms, pragma instructions and collections that are needed in the block will be declared. Below are few more characteristics of this part.

- This particular section is optional and can be skipped if no declarations are needed.
- This should be the first section in a PL/SQL block, if present.
- This section starts with the keyword 'DECLARE' for triggers and anonymous block. For other subprograms, this keyword will not be present. Instead, the part after the subprogram name definition marks the declaration section.
- This section should always be followed by execution section.

Execution Section

Execution part is the main and mandatory part which actually executes the code that is written inside it. Since the PL/SQL expects the executable statements from this block this cannot be an empty block, i.e., it should have at least one valid executable code line in it. Below are few more characteristics of this part.

- This can contain both PL/SQL code and SQL code.
- This can contain one or many blocks inside it as a nested block.
- This section starts with the keyword 'BEGIN'.
- This section should be followed either by 'END' or Exception-Handling section (if present)

Exception-Handling Section:

The exception is unavoidable in the program which occurs at run-time and to handle this Oracle has provided an Exception-handling section in blocks. This section can also contain PL/SQL statements. This is an optional section of the PL/SQL blocks.

- This is the section where the exception raised in the execution block is handled.
- This section is the last part of the PL/SQL block.
- Control from this section can never return to the execution block.
- This section starts with the keyword 'EXCEPTION'.
- This section should always be followed by the keyword 'END'.

The Keyword 'END' marks the end of PL/SQL block.

PL/SQL Block Syntax

Below is the syntax of the PL/SQL block structure.

```
DECLARE --optional
    <declarations>

BEGIN    --mandatory
    <executable statements. At least one executable statement is mandatory>

EXCEPTION --optional
    <exception handles>

END;    --mandatory
/
```

Note: A block should always be followed by '/' which sends the information to the compiler about the end of the block.

Types of PL/SQL block

PL/SQL blocks are of mainly two types.

1. Anonymous blocks
2. Named Blocks

Anonymous blocks:

Anonymous blocks are PL/SQL blocks which do not have any names assigned to them. They need to be created and used in the same session because they will not be stored in the server as database objects.

Since they need not store in the database, they need no compilation steps. They are written and executed directly, and compilation and execution happen in a single process.

- These blocks don't have any reference name specified for them.
- These blocks start with the keyword 'DECLARE' or 'BEGIN'.

- Since these blocks do not have any reference name, these cannot be stored for later purpose. They shall be created and executed in the same session.
- They can call the other named blocks, but call to anonymous block is not possible as it is not having any reference.
- It can have nested block in it which can be named or anonymous. It can also be nested in any blocks.
- These blocks can have all three sections of the block, in which execution section is mandatory, the other two sections are optional.

Named blocks:

Named blocks have a specific and unique name for them. They are stored as the database objects in the server. Since they are available as database objects, they can be referred to or used as long as it is present on the server. The compilation process for named blocks happens separately while creating them as a database objects.

Below are few more characteristics of Named blocks.

- These blocks can be called from other blocks.
- The block structure is same as an anonymous block, except it will never start with the keyword 'DECLARE'. Instead, it will start with the keyword 'CREATE' which instruct the compiler to create it as a database object.
- These blocks can be nested within other blocks. It can also contain nested blocks.
- Named blocks are basically of two types:
 1. Procedure
 2. Function

How to write a simple program using PL/SQL

- We need to execute "set serveroutput on" if we need to see the output of the code.
- Now we are ready to work with the SQL* Plus tool.

we are going to write a simple program for printing "Hello World" using "Anonymous block".

```
1. BEGIN
2.  dbms_output.put_line ('Hello World. . ');
3. END;
4. /
```

Output:

Hello World. . .

```
BEGIN
dbms_output.put_line ('Hello World..');
END;
/
```

Output:

Hello World...

Code Explanation:

- **Code line 2:** Prints the message "Hello World. . ."
- The below screenshot explains how to enter the code in SQL* Plus.

Note: A block should be always followed by '/' which sends the information to the compiler about the end of the block. Till the compiler encounters '/', it will not consider the block is completed, and it will not execute it.

```
Connected to:
Oracle Database 11g Enterprise Edition Release 11.2.0.3.0 - 64bit Production
With the Partitioning, OLAP, Data Mining and Real Application Testing options

SQL> set serveroutput on
SQL> begin
  2  dbms_output.put_line('Hello World. . .');
  3  end;
  4  /
Hello World. . .

PL/SQL procedure successfully completed.

SQL>
```

EXAMPLE :

```
SQL> SET SERVEROUTPUT ON
SQL> BEGIN
      DBMS_OUTPUT.PUT_LINE('HELLO');
      END;
      /
```

OUTPUT:HELLO

PL/SQL procedure successfully completed.

SQL>

Declaring and usage of variables in the program

Here we are going to print the "Hello World" using the variables.

```
1. DECLARE
2. text VARCHAR2(25);
3. BEGIN
4. text:= 'Hello World';
5. dbms_output.put_line (text);
6. END;
7. /
```

Output:

Hello World

```
DECLARE
text VARCHAR2(25);
BEGIN
text:= 'Hello World';
dbms_output.put_line (text);
END;
/
```

Output:

Hello World

Code Explanation:

- **Code line 2:** Declaring a variable "text" of a VARCHAR2 type with size 25
- **Code line 4:** Assigning the value "Hello World" to the variable "text".
- **Code line 5:** Printing the value of the variable "text".

```
SQL> DECLARE
      TEXT VARCHAR(15);
      BEGIN
      TEXT := 'HELLO PL/SQL';
      DBMS_OUTPUT.PUT_LINE(TEXT);
      END;
      /
HELLO PL/SQL
```

PL/SQL procedure successfully completed.

```
SQL> DECLARE
      TEXT VARCHAR(15);
      BEGIN
      TEXT := 'HELLO PL/SQL';
      -- DBMS_OUTPUT.PUT_LINE(TEXT); //SINGLE LINE COMMENT

      END;
      /
```

PL/SQL procedure successfully completed.

Comments in PL/SQL

Commenting code simply instructs the compiler to ignore that particular code from executing.

Comment can be used in the program to increase the readability of the program. In PL/SQL codes can be commented in two ways.

- Using '--' in the beginning of the line to comment that particular line.
- Using '/*.....*/' we can use multiple lines. The symbol '/*' marks the starting of the comment and the symbol '*/' marks the end of the comment. The code between these two symbols will be treated as comments by the compiler.

Example: In this example, we are going to print 'Hello World' and we are also going to see how the commented lines behave in the code



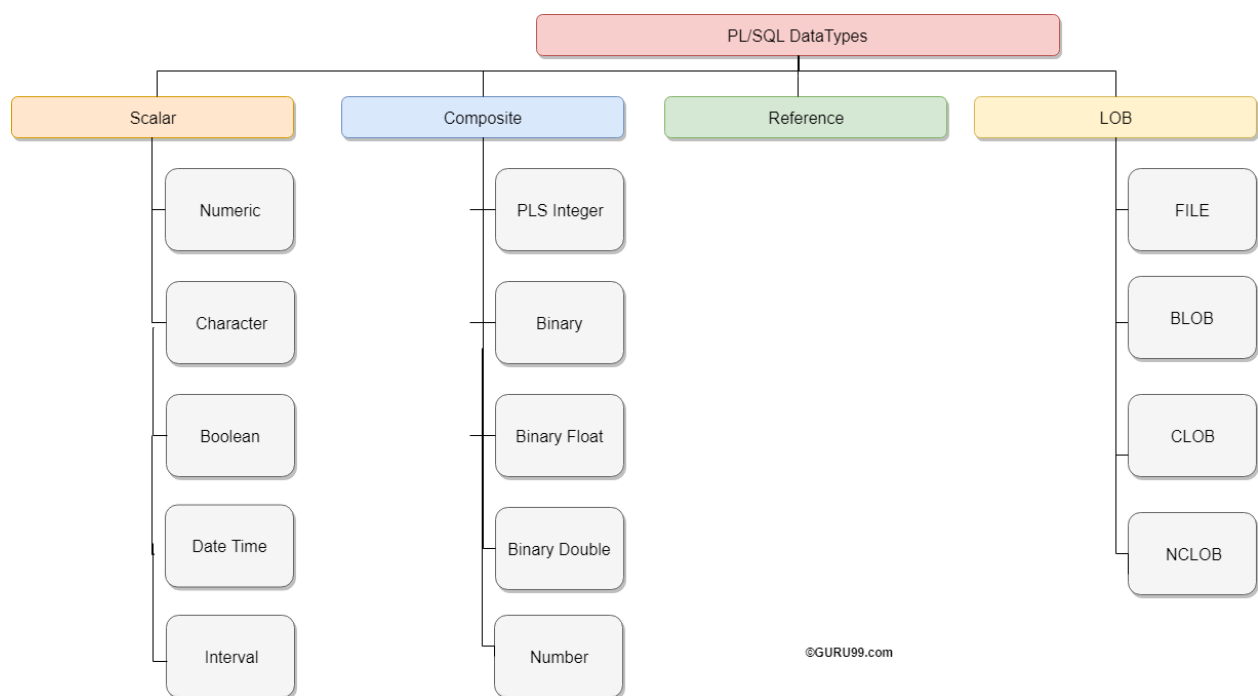
```
BEGIN
--single line comment
dbms_output.put_line(' Hello World ');
/*Multi line commenting begins
Multi line commenting ends */
END;
/
```

Output:

Hello World

Code Explanation:

- **Code line 2:** Single line comment and compiler ignored this line from execution.
- **Code line 3:** Printing the value "Hello World."
- **Code line 4:** Multiline commenting starts with '/*'
- **Code line 5:** Multiline commenting ends with '*/'



The literal values should always be enclosed in single quotes while assigning them to CHARACTER data type.

This character data type is further classified as follows:

- CHAR Data type (fixed string size)
- VARCHAR2 Data type (variable string size)
- VARCHAR Data type
- NCHAR (native fixed string size)
- NVARCHAR2 (native variable string size)
- LONG and LONG RAW

<https://www.guru99.com/pl-sql-identifiers.html>--Refer this link for more details

Declaration of Variables

Variables are mainly used to store data during the data manipulation or data processing. They need to be declared before using them inside the program. This declaration needs to be done in the declarative section of the PL/SQL blocks.

Declaration of variables is a process of assigning the name to the placeholder and associate the same with a valid datatype.

Syntax

```
<variable name> <datatype>;
```

The above syntax shows how to declare the variable in the declarative section.

Data storing in Variables

Once the variable is declared, they are ready to hold the data of defined type. The values of these variables can be assigned either in execution section or at the time of declaring itself. The value can be either a literal or another variable's value. Once a particular value has been assigned, it will be stored in the allocated memory space for that variable.

Syntax

```
<variable_name> <datatype> := <default_value>;
```

The above syntax shows how to declare the variable and assign value in the declarative section.

```
<Variable_name> <datatype>;  
<variable name> := <value>;
```

The above syntax shows how to assign the value to an already declared variable.

Example1: In this example, we are going to learn how to declare the variable and how to assign the value to them. We are going to print 'GURU99' in the following program by using the variables.


```
1. DECLARE
2. lv_name VARCHAR2(50);
3. lv_name_2 VARCHAR2(50) := 'GURU99';
4. BEGIN
5. lv_name := lv_name_2;
6. dbms_output.put_line(lv_name);
7. END;
```

Value assigned
in declaration
part

Output:
GURU99

```
DECLARE
lv_name VARCHAR2(50);
lv_name_2 VARCHAR2(50) := 'PL / SQL';
BEGIN
lv_name := lv_name_2;
dbms_output .put_line(lv_name);
END;
/
```

Code Explanation:

- **Code line 2:** Declaring the variable 'lv_name' of VARCHAR2 with size 50.
- **Code line 3:** Declaring the variable 'lv_name_2' of VARCHAR2 with size 50 and assigned the default value using literal ' PL / SQL'.
- **Code line 5:** Value for variable 'lv_name' has been assigned from the variable 'lv_name_2'.
- **Code line 6:** Printing the stored value of variable 'lv_name'.

When the above code is executed, you will get the following output.

Output: PL / SQL

A trigger is a named PL/SQL module that is stored in a database and can be invoked again. You can enable and disable a trigger, but you cannot explicitly call it.

When the Trigger is enabled, the database shall automatically call the Trigger whenever the event that triggers the Trigger occurs. While the Trigger is disabled, it shall not trigger.

You shall create the Trigger with the CREATE TRIGGER operator. You specify the triggering event in terms of the triggering operators and the object they act on. The Trigger is considered to be created or defined for an object that is either a table, representation, scheme or database.

You also specify the synchronization point that determines whether the Trigger starts before or after execution of the Trigger Operator and whether it starts for each line affected by the Trigger Operator. By default, the Trigger is created in the enabled state.

TRIGGER WITH BEFORE INSERT A ROW

Here we are taking simple table SUPERHEROS

```
SQL> CREATE TABLE SUPERHEROS( SP_NAME VARCHAR(10));
```

Table created.

Creating a triggers before inserting a value in to SUPERHEROS TABLE

```
SQL> CREATE OR REPLACE TRIGGER BI_SH
    BEFORE INSERT ON SUPERHEROS
    FOR EACH ROW
    DECLARE
    I_USER VARCHAR2(10);
    BEGIN
    SELECT USER INTO USER FROM DUAL;
    DBMS_OUTPUT.PUT_LINE('YOU JUST INSERTED A LINE MR. ' || I_USER);
    END;
/
```

Trigger created.

Now we are inserting a record into table using DML (INSERT) Command:

```
SQL> INSERT INTO SUPERHEROS VALUES ('RAM');
```

YOU JUST INSERTED A LINE MR. DSR - //The message we are created in TRIGGER

1 row created.

```
SQL> SELECT * FROM SUPERHEROS;
```

```
SP_NAME
-----
RAM
```

TRIGGER WITH BEFORE UPDATE A ROW

Creating a triggers before UPDATING a value in to SUPERHEROS TABLE

```
SQL> CREATE OR REPLACE TRIGGER BU_SH
```

```
BEFORE UPDATE ON SUPERHEROS
```

```
FOR EACH ROW
```

```
DECLARE
```

```
U_USER VARCHAR2(10);
```

```
BEGIN
```

```
SELECT USER INTO U_USER FROM DUAL;
```

```
DBMS_OUTPUT.PUT_LINE('YOU JUST UPDATED A LINE MR. ' || U_USER);
```

```
END;
```

```
/
```

Trigger created.

Now we are UPDATING record into table using DML (UPDATE) command:

```
SQL> UPDATE SUPERHEROS SET SP_NAME = 'KRISHNA' WHERE SP_NAME = 'RAM';
```

YOU JUST UPDATED A LINE MR. DSR

1 row updated.

```
SQL> SELECT * FROM SUPERHEROS;
```

```
SP_NAME  
-----  
KRISHNA
```

TRIGGER WITH BEFORE DELETE A ROW

Creating a trigger before DELETE a value from SUPERHEROS TABLE.

```
SQL>CREATE OR REPLACE TRIGGER BD_SH  
      BEFORE UPDATE ON SUPERHEROS  
      FOR EACH ROW  
      DECLARE  
      D_USER VARCHAR2(10);  
      BEGIN  
      SELECT USER INTO D_USER FROM DUAL;  
      DBMS_OUTPUT.PUT_LINE('YOU JUST DELETED A LINE MR. ' || D_USER);  
      END;  
      /
```

Trigger created.

Now we are DELETING A record from table using DML (DELETE) command:

```
SQL> DELETE FROM SUPERHEROS WHERE SP_NAME = 'KRISHNA';
```

1 row deleted.

```
SQL> SELECT * FROM SUPERHEROS;
```

no rows selected

Now we are creating a single trigger for all three DML commands

```
SQL>CREATE OR REPLACE TRIGGER TR_SH
    BEFORE INSERT OR DELETE OR UPDATE ON SUPERHEROS
    FOR EACH ROW
    DECLARE
    TR_USER VARCHAR2(10);
    BEGIN
        SELECT USER INTO TR_USER FROM DUAL;
        IF INSERTING THEN
            DBMS_OUTPUT.PUT_LINE ('ONE ROW INSERTED BY. ' || TR_USER);
        ELSIF DELETING THEN
            DBMS_OUTPUT.PUT_LINE ('ONE ROW DELETED BY. ' || TR_USER);
        ELSIF UPDATING THEN
            DBMS_OUTPUT.PUT_LINE ('ONE ROW UPDATED BY. ' || TR_USER);
        END IF;
    END;
/
```

Trigger created.

CHECK THE INSERT, UPDATE AND DELETE COMMANDS:

```
SQL> INSERT INTO SUPERHEROS VALUES ('RAM');
```

ONE ROW INSERTED BY. DSR

1 row created.

```
SQL> UPDATE SUPERHEROS SET SP_NAME = 'KRISHNA' WHERE SP_NAME = 'RAM';
```

ONE ROW UPDATED BY. DSR

1 row updated.

```
SQL> DELETE FROM SUPERHEROS WHERE SP_NAME = 'KRISHNA';
```

ONE ROW DELETED BY. DSR

1 row deleted.

PL/SQL IF THEN statement example

In the following example, the statements between THEN and END IF execute because the sales revenue is greater than 100,000.

EXAMPLE

```
DECLARE
    n_sales NUMBER := 2000000;
BEGIN
    IF n_sales > 100000 THEN
        DBMS_OUTPUT.PUT_LINE( 'Sales revenue is greater than 100K ' );
    END IF;
END;
/
```

PL/SQL IF THEN ELSE statement:

The IF THEN ELSE statement has the following structure:

```
DECLARE
    n_sales NUMBER := 210345;
    n_commission NUMBER( 10, 2 ) := 0;
BEGIN
    IF n_sales > 200000 THEN
        n_commission := n_sales * 0.1;
        DBMS_OUTPUT.PUT_LINE(n_commission);
    ELSE
        n_commission := n_sales * 0.05;
        DBMS_OUTPUT.PUT_LINE(n_commission);
    END IF;
END;
/
```

- **PL/SQL IF THEN ELSIF statement:**

The following illustrates the structure of the IF THEN ELSIF statement:

In this structure, the condition between IF and THEN, which is the first condition, is always evaluated. Each other condition between ELSEIF and THEN is evaluated only if the preceding condition is FALSE.

For example, the condition_2 is evaluated only if the condition_1 is false, the condition_3 is evaluated only if the condition_2 is false, and so on.

If a condition is true, other subsequent conditions are not evaluated.

If no condition is true, the else_statements between the ELSE and ENDIF execute.

In case you skip the the ELSE clause and no condition is TRUE, then the IF THEN ELSIF does nothing

EXAMPLE:

DECLARE

n_sales NUMBER := 123456;

n_commission NUMBER(10, 2) := 0;

BEGIN

IF n_sales > 200000 THEN

n_commission := n_sales * 0.1;

DBMS_OUTPUT.PUT_LINE(n_commission);

ELSIF n_sales <= 200000 AND n_sales > 100000 THEN

n_commission := n_sales * 0.05;

DBMS_OUTPUT.PUT_LINE(n_commission);

ELSIF n_sales <= 100000 AND n_sales > 50000 THEN

n_commission := n_sales * 0.03;

DBMS_OUTPUT.PUT_LINE(n_commission);

ELSE

```
        n_commission := n_sales * 0.02;
        DBMS_OUTPUT.PUT_LINE(n_commission);
    END IF;
END;
```

- **Nested IF statement:**

You can nest an IF statement within another IF statement as shown below:

```
IF condition_1 THEN
    IF condition_2 THEN
        nested_if_statements;
    END IF;
ELSE
    else_statements;
END IF;
```


PL/SQL CASE Statement

The CASE statement chooses one sequence of statements to execute out of many possible sequences.

The CASE statement has two types: simple CASE statement and searched CASE statement. Both types of the CASE statements support an optional ELSE clause.

Simple CASE statement

A simple CASE statement evaluates a single expression and compares the result with some values.

The simple CASE statement has the following structure:

The CASE statement chooses one sequence of statements to execute out of many possible sequences.

The CASE statement has two types: simple CASE statement and searched CASE statement. Both types of the CASE statements support an optional ELSE clause.

Simple CASE statement

A simple CASE statement evaluates a single expression and compares the result with some values.

The simple CASE statement has the following structure:

Let's examine the syntax of the simple CASE statement in detail:

1) selector

The selector is an expression which is evaluated once. The result of the selector is used to select one of the several alternatives e.g., selector_value_1 and selector_value_2.

2) WHEN selector_value THEN statements

The selector values i.e., selector_value_1, selector_value_2, etc., are evaluated sequentially. If the result of a selector value equals the result of the selector, then the associated sequence of statements executes and the CASE statement ends. In addition, the subsequent selector values are not evaluated.

3) ELSE else_statements

If no values in WHERE clauses match the result of the selector in the CASE clause, the sequence of statements in the ELSE clause executes.

Because the ELSE clause is optional, you can skip it. However, if you do so, PL/SQL will implicitly use the following:

```
ELSE
    RAISE CASE_NOT_FOUND;
```

In other words, PL/SQL raises a CASE_NOT_FOUND error if you don't specify an ELSE clause and the result of the CASE expression does not match any value in the WHEN clauses.

Note that this behavior of the CASE statement is different from the [IF THEN statement](#). When the IF THEN statement has no ELSE clause and the condition is not met, PL/SQL does nothing instead raising an error.

Simple CASE statement example

The following example compares single value (c_grade) with many possible values 'A', 'B', 'C', 'D', and 'F':

```
DECLARE
    c_grade CHAR( 1 );
    c_rank  VARCHAR2( 20 );
BEGIN
    c_grade := 'D';
    CASE c_grade
    WHEN 'A' THEN
        c_rank := 'Excellent' ;
    WHEN 'B' THEN
        c_rank := 'Very Good' ;
    WHEN 'C' THEN
        c_rank := 'Good' ;
    WHEN 'D' THEN
        c_rank := 'Fair' ;
```

```

WHEN 'F' THEN
    c_rank := 'Poor' ;
ELSE
    c_rank := 'No such grade' ;
END CASE;
DBMS_OUTPUT.PUT_LINE( c_rank );
END;

```

Searched CASE statement

The searched CASE statement evaluates multiple Boolean expressions and executes the sequence of statements associated with the first condition that evaluates to TRUE.

The searched CASE statement has the following structure:

```

CASE
WHEN condition_1 THEN statements_1
WHEN condition_2 THEN statements_2
...
WHEN condition_n THEN statements_n
[ ELSE
    else_statements ]
END CASE;]

```

The searched CASE statement follows the rules below:

- The conditions in the WHEN clauses in are evaluated in order, from top to bottom.
- The sequence of statements associated with the WHEN clause whose condition evaluates to TRUE is executed. If more than one condition evaluates to TRUE, only the first one executes.
- If no condition evaluates to TRUE, the else_statements in the ELSE clause executes. If you skip the ELSE clause and no expressions are TRUE, a CASE_NOT_FOUND [exception](#) is raised.

Searched CASE statement example

The following example illustrates how to use the searched CASE statement to calculate sales commission based on sales revenue.

```
DECLARE
    n_sales      NUMBER;
    n_commission NUMBER;
BEGIN
    n_sales := 200000;
    CASE
        WHEN n_sales > 200000 THEN
            n_commission := 0.2;
        WHEN n_sales >= 100000 AND n_sales < 200000 THEN
            n_commission := 0.15;
        WHEN n_sales >= 50000 AND n_sales < 100000 THEN
            n_commission := 0.1;
        WHEN n_sales > 30000 THEN
            n_commission := 0.05;
        ELSE
            n_commission := 0;
    END CASE;
    DBMS_OUTPUT.PUT_LINE( 'Commission is ' || n_commission * 100 ||
'% ' );
END;
```

In this example, the sales revenue was set to 150,000. The first expression evaluated to FALSE:

```
n_sales > 200000
```

But the second expression evaluates to TRUE and the sale commission was set to 15%:

```
n_commission := 0.15;
```

Natural JOIN

Definition of joins:

SQL Join is used to fetch data from two or more tables, which is joined to appear as single set of data. It is used for combining column from two or more tables by using values common to both tables.

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

The **syntax** for Natural Join is,

```
SELECT * FROM TABLE1 NATURAL JOIN TABLE2;
```

table1: first table.

table2: second table

CONSIDER THE FOLLOWING TWO TABLES:

```
SQL> CREATE TABLE STUDENT (S_ID INT NOT NULL, SNAME VARCHAR (20), AGE INT, ADDRESS VARCHAR (20));
```

```
SQL> INSERT INTO STUDENT VALUES (101, 'RAM', 19, 'VIJ');
```

1 row created.

```
SQL> INSERT INTO STUDENT VALUES (102, 'LAKSHMAN', 18, 'HYD');
```

1 row created.

```
SQL> INSERT INTO STUDENT VALUES (103, 'KRISH', 20, 'VIZAG');
```

1 row created.

```
SQL> INSERT INTO STUDENT VALUES (104, 'VENKAT',21, 'vij');
```

1 row created.

```
SQL> INSERT INTO STUDENT VALUES (105, 'SIVA', 22, 'VIZAG');
```

1 row created.

```
SQL> INSERT INTO STUDENT VALUES (106, 'SATYA', 22, 'TIRUPATI');
```

1 row created.

```
SQL> SELECT * FROM STUDENT;
```

S_ID	SNAME	AGE	ADDRESS
101	RAM	19	VIJ
102	LAKSHMAN	18	HYD
103	KRISH	20	VIZAG
104	VENKAT	21	vi j
105	SIVA	22	VIZAG
106	SATYA	22	TIRUPATI

```
SQL> CREATE TABLE COURSE (COURSE_ID INT, S_ID INT);
```

Table created.

```
SQL> INSERT INTO COURSE VALUES (1, 101);
```

1 row created.

```
SQL> INSERT INTO COURSE VALUES (2, 102);
```

1 row created.

```
SQL> INSERT INTO COURSE VALUES (2, 103);
```

1 row created.

```
SQL> INSERT INTO COURSE VALUES (3, 104);
```

1 row created.

```
SQL> INSERT INTO COURSE VALUES (1, 105);
```

1 row created.

```
SQL> INSERT INTO COURSE (COURSE_ID) VALUES (4);
```

1 row created.

```
SQL> INSERT INTO COURSE (COURSE_ID) VALUES (5);
```

1 row created.

```
SQL> INSERT INTO COURSE (COURSE_ID) VALUES (6);
```

1 row created.

SQL> SELECT * FROM COURSE;

COURSE_ID	S_ID
1	101
2	102
2	103
3	104
1	105
4	
5	
6	

SQL> SELECT * FROM STUDENT NATURAL JOIN COURSE;

S_ID	SNAME	AGE	ADDRESS	COURSE_ID
101	RAM	19	VIJ	1
102	LAKSHMAN	18	HYD	2
103	KRISH	20	VIZAG	2
104	VENKAT	21	vij	3
105	SIVA	22	VIZAG	1

SQL> SELECT * FROM COURSE NATURAL JOIN STUDENT;

S_ID	COURSE_ID	SNAME	AGE	ADDRESS
101	1	RAM	19	VIJ
102	2	LAKSHMAN	18	HYD
103	2	KRISH	20	VIZAG
104	3	VENKAT	21	vij
105	1	SIVA	22	VIZAG

LEFT JOIN / LEFT OUTER JOIN

LEFT JOIN: This join returns all the rows of the table on the left side of the join and matching rows for the table on the right side of join. The rows for which there is no matching row on right side, the result-set will contain *null*. LEFT JOIN is also known as **LEFT OUTER JOIN**.

Syntax:

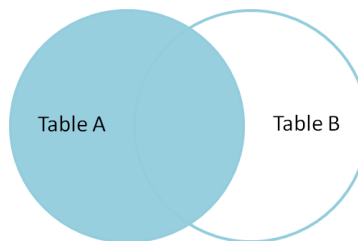
```
SELECT table1.column1, table.column2, table2.column1,...  
FROM table1  
LEFT JOIN table2  
ON table1.matching_column = table2.matching_column;
```

table1: first table.

table2: second table.

Mathching_column: Column common to both the tables.

Note: We can also use LEFT OUTER JOIN instead of LEFT JOIN, both are same.



LEFT JOIN / LEFT OUTER JOIN

CONSIDER THE FOLLOWING TWO TABLES:

```
SQL> CREATE TABLE STUDENT (S_ID INT NOT NULL, SNAME VARCHAR (20), AGE INT,  
ADDRESS VARCHAR (20));
```

```
SQL> INSERT INTO STUDENT VALUES(101, 'RAM', 19, 'VIJ');
```

1 row created.

```
SQL> INSERT INTO STUDENT VALUES(102, 'LAKSHMAN', 18, 'HYD');
```

1 row created.

```
SQL> INSERT INTO STUDENT VALUES(103, 'KRISH', 20, 'VIZAG');
```


1 row created.

```
SQL> INSERT INTO STUDENT VALUES(104, 'VENKAT',21, 'vij');
```

1 row created.

```
SQL> INSERT INTO STUDENT VALUES(105, 'SIVA', 22, 'VIZAG');
```

1 row created.

```
SQL> INSERT INTO STUDENT VALUES (106, 'SATYA', 22, 'TIRUPATI');
```

1 row created.

```
SQL> SELECT * FROM STUDENT;
```

S_ID	SNAME	AGE	ADDRESS
101	RAM	19	VIJ
102	LAKSHMAN	18	HYD
103	KRISH	20	VIZAG
104	VENKAT	21	vij
105	SIVA	22	VIZAG
106	SATYA	22	TIRUPATI

```
SQL> CREATE TABLE COURSE( COURSE_ID INT, S_ID INT);
```

Table created.

```
SQL> INSERT INTO COURSE VALUES(1, 101);
```

1 row created.

```
SQL> INSERT INTO COURSE VALUES(2, 102);
```

1 row created.

```
SQL> INSERT INTO COURSE VALUES(2, 103);
```

1 row created.

```
SQL> INSERT INTO COURSE VALUES(3, 104);
```

1 row created.

```
SQL> INSERT INTO COURSE VALUES(1, 105);
```

1 row created.

```
SQL> SELECT * FROM COURSE;
```

COURSE_ID	S_ID
1	101
2	102
2	103
3	104
1	105

LEFT/LEFT OUTER JOIN:

```
SQL> SELECT STUDENT.S_ID, STUDENT.SNAME, COURSE.COURSE_ID FROM STUDENT LEFT  
JOIN COURSE ON STUDENT.S_ID = COURSE.S_ID;
```

S_ID	SNAME	COURSE_ID
101	RAM	1
102	LAKSHMAN	2
103	KRISH	2
104	VENKAT	3
105	SIVA	1
106	SATYA	--

6 rows selected.

```
SQL> SELECT STUDENT.S_ID, COURSE.COURSE_ID, STUDENT.SNAME FROM STUDENT LEFT  
JOIN COURSE ON STUDENT.S_ID = COURSE.S_ID;
```

S_ID	COURSE_ID	SNAME
101	1	RAM
102	2	LAKSHMAN
103	2	KRISH
104	3	VENKAT
105	1	SIVA
106	--	SATYA

6 rows selected.

```
SQL> SELECT * FROM STUDENT LEFT JOIN COURSE ON STUDENT.S_ID= COURSE.S_ID;
```

S_ID	SNAME	AGE	ADDRESS	COURSE_ID	S_ID
101	RAM	19	VIJ	1	101
102	LAKSHMAN	18	HYD	2	102
103	KRISH	20	VIZAG	2	103
104	VENKAT	21	vi j	3	104
105	SIVA	22	VIZAG	1	105
106	SATYA	20	TIRUPATI		

6 rows selected.

INNER JOINS

A SQL Join statement is used to combine data or rows from two or more tables based on a common field between them.

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

Following are the types of JOIN that we can use in SQL:

- Inner Join
- Natural Join
- Left Join
- Right Join
- Full Join

CONSIDER THE FOLLOWING TWO TABLES:

```
SQL> CREATE TABLE STUDENT (S_ID INT NOT NULL, SNAME VARCHAR(20), AGE INT, ADDRESS VARCHAR(20));
```

```
SQL> INSERT INTO STUDENT VALUES (101, 'RAM', 19, 'VIJ');
```

1 row created.

```
SQL> INSERT INTO STUDENT VALUES (102, 'LAKSHMAN', 18, 'HYD');
```

1 row created.

```
SQL> INSERT INTO STUDENT VALUES(103, 'KRISH', 20, 'VIZAG');
```

1 row created.

```
SQL> INSERT INTO STUDENT VALUES (104, 'VENKAT',21, 'vij');
```

1 row created.

```
SQL> INSERT INTO STUDENT VALUES (105, 'SIVA', 22, 'VIZAG');
```

1 row created.

```
SQL> INSERT INTO STUDENT VALUES (106, 'SATYA', 22, 'TIRUPATI');
```

1 row created.

```
SQL> SELECT * FROM STUDENT;
```

S_ID	SNAME	AGE	ADDRESS
101	RAM	19	VIJ
102	LAKSHMAN	18	HYD
103	KRISH	20	VIZAG
104	VENKAT	21	vi j
105	SIVA	22	VIZAG
106	SATYA	22	TIRUPATI

```
SQL> CREATE TABLE COURSE (COURSE_ID INT, S_ID INT);
```

Table created.

```
SQL> INSERT INTO COURSE VALUES (1, 101);
```

1 row created.

```
SQL> INSERT INTO COURSE VALUES (2, 102);
```

1 row created.

```
SQL> INSERT INTO COURSE VALUES (2, 103);
```

1 row created.

```
SQL> INSERT INTO COURSE VALUES (3, 104);
```

1 row created.

```
SQL> INSERT INTO COURSE VALUES (1, 105);
```

1 row created.

```
SQL> SELECT * FROM COURSE;
```

COURSE_ID	S_ID
1	101
2	102
2	103
3	104
1	105

INNER JOIN:

The simplest Join is INNER JOIN.

The INNER JOIN keyword selects all rows from both the tables as long as the condition satisfies.

This keyword will create the result-set by combining all rows from both the tables where the condition satisfies i.e value of the common field will be same.

Syntax:

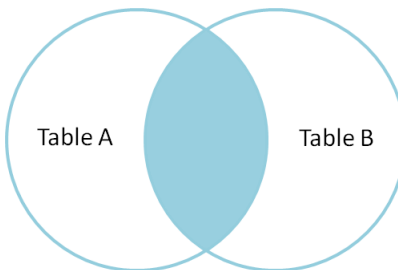
```
SELECT table1.column1, table1.column2, table2.column1,...  
FROM table1  
INNER JOIN table2  
ON table1.matching_column = table2. Matching_column;
```

Table1: First table

Table2: Second table

Matching_column: Column common to both the tables.

Note: We can also write JOIN instead of INNER JOIN. JOIN is same as INNER JOIN.



INNER JOIN

- This query will show the student ID and age of students enrolled in different courses.

```
SQL> SELECT COURSE.COURSE_ID, STUDENT.S_ID, STUDENT.AGE FROM STUDENT INNER  
JOIN COURSE ON STUDENT.S_ID = COURSE.S_ID;
```

COURSE_ID	S_ID	AGE
1	101	19
2	102	18
2	103	20
3	104	21
1	105	22

- WRITE A QUERY TO DISPLAY THE COURSE_ID FROM COURSE TABLE AND S_ID, SNAME AND AGE OF STUDENT FORM STUDENT TABLE USING INNER JOIN.

```
SQL> SELECT COURSE.COURSE_ID, STUDENT.S_ID, STUDENT.SNAME, STUDENT.AGE FROM
STUDENT INNER JOIN COURSE ON STUDENT.S_ID = COURSE.S_ID;
```

COURSE_ID	S_ID	SNAME	AGE
1	101	RAM	19
2	102	LAKSHMAN	18
2	103	KRISH	20
3	104	VENKAT	21
1	105	SIVA	22

```
SQL> SELECT STUDENT.S_ID, STUDENT.SNAME, STUDENT.AGE, COURSE.COURSE_ID FROM
STUDENT INNER JOIN COURSE ON STUDENT.S_ID = COURSE.S_ID;
```

S_ID	SNAME	AGE	COURSE_ID
101	RAM	19	1
102	LAKSHMAN	18	2
103	KRISH	20	2
104	VENKAT	21	3
105	SIVA	22	1

FULL JOIN

FULL JOIN: FULL JOIN creates the result-set by combining result of both LEFT JOIN and RIGHT JOIN. The result-set will contain all the rows from both the tables. The rows for which there is no matching, the result-set will contain *NULL* values.

Syntax:

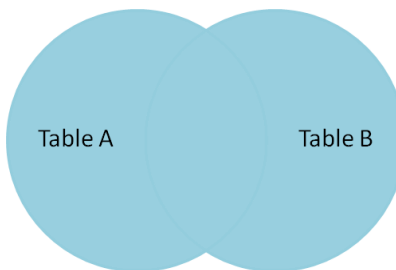
Syntax:

```
SELECT COLUMN-NAME-LIST  
FROM table1  
FULL JOIN table2  
ON table1.matching_column = table2.matching_column;
```

table1: first table.

table2: second table.

Mathching_column: Column common to both the tables.



FULL JOIN

CONSIDER THE FOLLOWING TWO TABLES:

```
SQL> CREATE TABLE STUDENT (S_ID INT NOT NULL, SNAME VARCHAR (20), AGE INT,  
ADDRESS VARCHAR (20));
```

```
SQL> INSERT INTO STUDENT VALUES (101, 'RAM', 19, 'VIJ');
```

1 row created.

```
SQL> INSERT INTO STUDENT VALUES (102, 'LAKSHMAN', 18, 'HYD');
```

1 row created.


```
SQL> INSERT INTO STUDENT VALUES (103, 'KRISH', 20, 'VIZAG');
```

1 row created.

```
SQL> INSERT INTO STUDENT VALUES (104, 'VENKAT', 21, 'vij');
```

1 row created.

```
SQL> INSERT INTO STUDENT VALUES (105, 'SIVA', 22, 'VIZAG');
```

1 row created.

```
SQL> INSERT INTO STUDENT VALUES (106, 'SATYA', 22, 'TIRUPATI');
```

1 row created.

```
SQL> SELECT * FROM STUDENT;
```

S_ID	SNAME	AGE	ADDRESS
101	RAM	19	VIJ
102	LAKSHMAN	18	HYD
103	KRISH	20	VIZAG
104	VENKAT	21	vij
105	SIVA	22	VIZAG
106	SATYA	22	TIRUPATI

```
SQL> CREATE TABLE COURSE (COURSE_ID INT, S_ID INT);
```

Table created.

```
SQL> INSERT INTO COURSE VALUES (1, 101);
```

1 row created.

```
SQL> INSERT INTO COURSE VALUES (2, 102);
```

1 row created.

```
SQL> INSERT INTO COURSE VALUES (2, 103);
```

1 row created.

```
SQL> INSERT INTO COURSE VALUES (3, 104);
```

1 row created.

```
SQL> INSERT INTO COURSE VALUES (1, 105);
```

1 row created.

```
SQL> INSERT INTO COURSE (COURSE_ID) VALUES (4);
```

1 row created.

```
SQL> INSERT INTO COURSE (COURSE_ID) VALUES(5);
```

1 row created.

```
SQL> INSERT INTO COURSE (COURSE_ID) VALUES (6);
```

1 row created.

```
SQL> SELECT * FROM COURSE;
```

COURSE_ID	S_ID
1	101
2	102
2	103
3	104
1	105
4	
5	
6	

```
SQL> SELECT STUDENT.SNAME, COURSE.COURSE_ID FROM STUDENT FULL JOIN COURSE ON  
(COURSE.S_ID = STUDENT.S_ID);
```

SNAME	COURSE_ID
RAM	1
LAKSHMAN	2
KRISH	2
VENKAT	3
SIVA	1
SATYA	
	6
	5
	4

9 rows selected.

```
SQL> SELECT STUDENT.S_ID, COURSE.COURSE_ID FROM STUDENT FULL JOIN COURSE ON
(STUDENT.S_ID = COURSE.S_ID);
```

S_ID	COURSE_ID
101	1
102	2
103	2
104	3
105	1
106	---
---	6
---	5
---	4

9 rows selected.

```
SQL> SELECT COURSE.COURSE_ID, STUDENT.S_ID FROM STUDENT FULL JOIN COURSE ON
STUDENT.S_ID = COURSE.S_ID;
```

COURSE_ID	S_ID
1	101
2	102
2	103
3	104
1	105
---	106
6	
5	
4	

```
SQL> SELECT * FROM STUDENT FULL JOIN COURSE ON STUDENT.S_ID = COURSE.S_ID;
```

S_ID	SNAME	AGE	ADDRESS	COURSE_ID	S_ID
101	RAM	19	VIJ	1	101
102	LAKSHMAN	18	HYD	2	102
103	KRISH	20	VIZAG	2	103
104	VENKAT	21	vi j	3	104
105	SIVA	22	VIZAG	1	105
106	SATYA	22	TIRUPATI	---	
				6	
				5	
				4	

Cross JOIN or Cartesian Product

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

Cross JOIN Syntax is,

```
SELECT COLUMN-NAME-LIST FROM TABLE-NAME1 CROSS JOIN TABLE-NAME2;
```

OR

```
SELECT * FROM TABLE-NAME1, TABLE-NAME2;
```

Table1: First table

Table2: Second table

- Consider the following relations/tables:

```
SQL> CREATE TABLE CROSS1( ID INT, NAME VARCHAR(20));
```

Table created.

```
SQL> INSERT INTO CROSS1 VALUES (1, 'RAM');
```

1 row created.

```
SQL> INSERT INTO CROSS1 VALUES (2, 'LAKSH');
```

1 row created.

```
SQL> INSERT INTO CROSS1 VALUES (3, 'KRISH');
```

1 row created.

```
SQL> SELECT * FROM CROSS1;
```

ID	NAME
1	RAM
2	LAKSH
3	KRISH

```
SQL> CREATE TABLE CROSS2 (ID INT, ADDRESS VARCHAR (20));
```

Table created.

```
SQL> INSERT INTO CROSS2 VALUES (1, 'VIZ');
```

1 row created.

```
SQL> INSERT INTO CROSS2 VALUES (2, 'VIZAG');
```

1 row created.

```
SQL> INSERT INTO CROSS2 VALUES (3, 'TIRUPATI');
```

1 row created.

```
SQL> SELECT * FROM CROSS2;
```

ID	ADDRESS
1	VIZ
2	VIZAG
3	TIRUPATI

```
SQL> SELECT * FROM CROSS1 CROSS JOIN CROSS2;
```

ID	NAME	ID	ADDRESS
1	RAM	1	VIZ
1	RAM	2	VIZAG
1	RAM	3	TIRUPATI
2	LAKSH	1	VIZ
2	LAKSH	2	VIZAG
2	LAKSH	3	TIRUPATI
3	KRISH	1	VIZ
3	KRISH	2	VIZAG
3	KRISH	3	TIRUPATI

9 rows selected.

```
SQL> SELECT CROSS1.ID, CROSS1.NAME, CROSS2.ADDRESS FROM CROSS1 CROSS JOIN CROSS2;
```

ID	NAME	ADDRESS
1	RAM	VIZ
1	RAM	VIZAG
1	RAM	TIRUPATI
2	LAKSH	VIZ
2	LAKSH	VIZAG
2	LAKSH	TIRUPATI
3	KRISH	VIZ
3	KRISH	VIZAG
3	KRISH	TIRUPATI

9 rows selected.

```
SQL> SELECT * FROM CROSS1, CROSS2;
```

ID NAME	ID ADDRESS
1 RAM	1 VIZ
1 RAM	2 VIZAG
1 RAM	3 TIRUPATI
2 LAKSH	1 VIZ
2 LAKSH	2 VIZAG
2 LAKSH	3 TIRUPATI
3 KRISH	1 VIZ
3 KRISH	2 VIZAG
3 KRISH	3 TIRUPATI

9 rows selected

Note: The number of rows in the output will always be the cross product of number of rows in each table. In our example table-1 has 3 rows and table-2 has 3 rows so the output has $3 \times 3 = 9$ rows.

What is ER Diagram?

ER Diagram stands for Entity Relationship Diagram, also known as ERD is a diagram that displays the relationship of entity sets stored in a database. In other words, ER diagrams help to explain the logical structure of databases. ER diagrams are created based on three basic concepts: entities, attributes and relationships.

ER Diagrams contain different symbols that use rectangles to represent entities, ovals to define attributes and diamond shapes to represent relationships.

What is ER Model?

ER Model stands for Entity Relationship Model is a high-level conceptual data model diagram. ER model helps to systematically analyze data requirements to produce a well-designed database. The ER Model represents real-world entities and the relationships between them. Creating an ER Model in DBMS is considered as a best practice before implementing your database.

ER Modeling helps you to analyze data requirements systematically to produce a well-designed database. So, it is considered a best practice to complete ER modeling before implementing your database.

OR

An Entity-relationship model (ER model) describes the structure of a database with the help of a diagram, which is known as Entity Relationship Diagram (ER Diagram). An ER model is a design or blueprint of a database that can later be implemented as a database. The main components of E-R model are: entity set and relationship set.

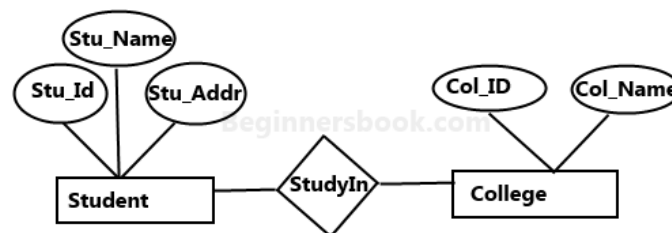
What is an Entity Relationship Diagram (ER Diagram)?

An ER diagram shows the relationship among entity sets. An entity set is a group of similar entities and these entities can have attributes.

In terms of DBMS, an entity is a table or attribute of a table in database, so by showing relationship among tables and their attributes, ER diagram shows the complete logical structure of a database. Lets have a look at a simple ER diagram to understand this concept.

A simple ER Diagram:

In the following diagram we have two entities **Student** and **College** and their relationship. The relationship between **Student** and **College** is many to one as a college can have many students however a student cannot study in multiple colleges at the same time. Student entity has attributes such as **Stu_Id**, **Stu_Name** & **Stu_Addr** and College entity has attributes such as **Col_ID** & **Col_Name**.



Sample E-R Diagram

Here are the geometric shapes and their meaning in an E-R Diagram. We will discuss these terms in detail in the next section(Components of a ER Diagram) of this guide so don't worry too much about these terms now, just go through them once.

Rectangle: Represents Entity sets.

Ellipses: Attributes

Diamonds: Relationship Set

Lines: They link attributes to Entity Sets and Entity sets to Relationship Set

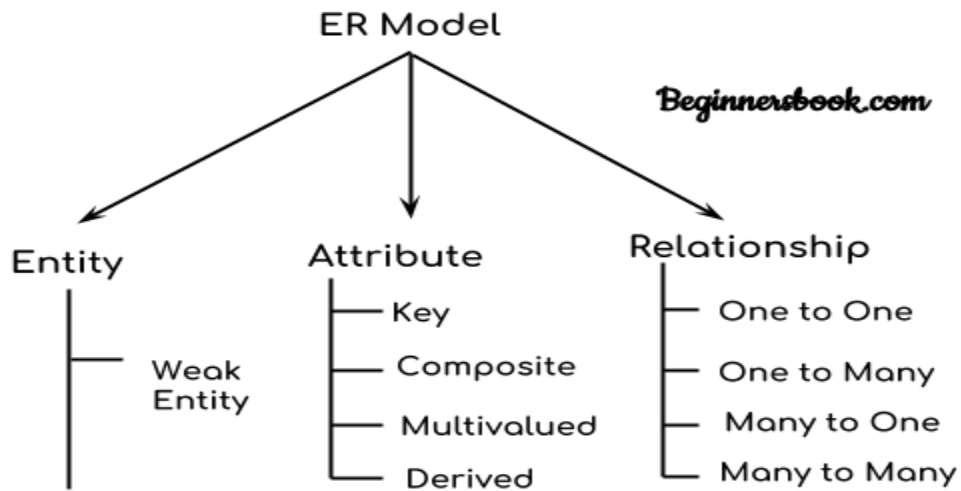
Double Ellipses: Multivalued Attributes

Dashed Ellipses: Derived Attributes

Double Rectangles: Weak Entity Sets

Double Lines: Total participation of an entity in a relationship set

Components of a ER Diagram



Components of ER Diagram

What is a Surrogate key?

SURROGATE KEYS is an **artificial** key which aims to **uniquely identify** each record is called a surrogate key. This kind of partial key in DBMS is unique because it is created when you don't have any natural primary key. They do not lend any meaning to the data in the table. Surrogate key is usually an integer. A surrogate key is a value generated right before the record is inserted into a table.

Surrogate Key has no actual meaning and is used to represent existence. It has an existence only for data analysis.

Fname	Lastname	Start Time	End Time
Anne	Smith	09:00	18:00
Jack	Francis	08:00	17:00
Anna	McLean	11:00	20:00
Shown	Willam	14:00	23:00

Above, given example, shown shift timings of the different employee. In this example, a surrogate key is needed to uniquely identify each employee.

Surrogate keys in sql are allowed when

- No property has the parameter of the primary key.
- In the table when the primary key is too big or complicated.

	UNIQUE	NOT UPDATED	NULL
SURROGATE KEY	YES	YES	YES
UNIQUE KEY	YES	NO	NO

Decomposition in DBMS removes redundancy, anomalies and inconsistencies from a database by dividing the table into multiple tables.

The following are the types -

- Lossless Decomposition
- Lossy Decomposition

Lossless Decomposition

Decomposition is lossless if it is feasible to reconstruct relation R from decomposed tables using Joins. This is the preferred choice. The information will not lose from the relation when decomposed. The join would result in the same original relation.

Let us see an example -

<EmpInfo>

Emp_ID	Emp_Name	Emp_Age	Emp_Location	Dept_ID	Dept_Name
E001	Jacob	29	Alabama	Dpt1	Operations
E002	Henry	32	Alabama	Dpt2	HR
E003	Tom	22	Texas	Dpt3	Finance

Decompose the above table into two tables:

<EmpDetails>

Emp_ID	Emp_Name	Emp_Age	Emp_Location
E001	Jacob	29	Alabama
E002	Henry	32	Alabama
E003	Tom	22	Texas

<DeptDetails>

Dept_ID	Emp_ID	Dept_Name
Dpt1	E001	Operations
Dpt2	E002	HR
Dpt3	E003	Finance

Now, Natural Join is applied on the above two tables -

The result will be -

Emp_ID	Emp_Name	Emp_Age	Emp_Location	Dept_ID	Dept_Name
E001	Jacob	29	Alabama	Dpt1	Operations
E002	Henry	32	Alabama	Dpt2	HR
E003	Tom	22	Texas	Dpt3	Finance

Therefore, the above relation had lossless decomposition i.e. no loss of information.

Lossy Decomposition

As the name suggests, when a relation is decomposed into two or more relational schemas, the loss of information is unavoidable when the original relation is retrieved.

Let us see an example -

<EmpInfo>

Emp_ID	Emp_Name	Emp_Age	Emp_Location	Dept_ID	Dept_Name
E001	Jacob	29	Alabama	Dpt1	Operations
E002	Henry	32	Alabama	Dpt2	HR
E003	Tom	22	Texas	Dpt3	Finance

Decompose the above table into two tables -

<EmpDetails>

Emp_ID	Emp_Name	Emp_Age	Emp_Location
E001	Jacob	29	Alabama
E002	Henry	32	Alabama
E003	Tom	22	Texas

<DeptDetails>

Dept_ID	Dept_Name
Dpt1	Operations
Dpt2	HR
Dpt3	Finance

Now, you won't be able to join the above tables, since **Emp_ID** isn't part of the **DeptDetails** relation. Therefore, the above relation has lossy decomposition.

Properties of Decomposition

The following two properties must be followed when decomposing a given relation-

1. Lossless decomposition--

Lossless decomposition ensures-

- No information is lost from the original relation during decomposition.
- When the sub relations are joined back, the same relation is obtained that was decomposed.

Every decomposition must always be lossless.

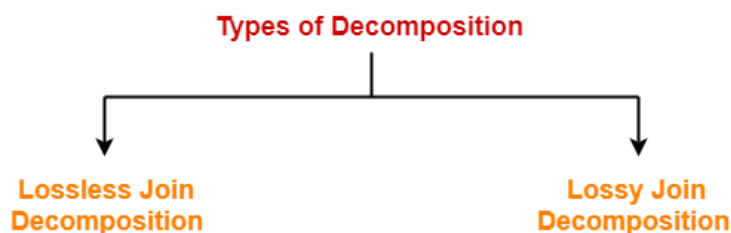
2. Dependency Preservation--

Dependency preservation ensures-

- None of the functional dependencies that hold on the original relation are lost.
- The sub relations still hold or satisfy the functional dependencies of the original relation.

Types of Decomposition-

Decomposition of a relation can be completed in the following two ways-



1. Lossless Join Decomposition-

- Consider there is a relation **R** which is decomposed into sub relations **R₁ , R₂ , , R_n**.
- This decomposition is called lossless join decomposition when the join of the sub relations results in the same relation **R** that was decomposed.
- For lossless join decomposition, we always have-

$$R_1 \bowtie R_2 \bowtie R_3 \dots \bowtie R_n = R$$

where \bowtie is a natural join operator

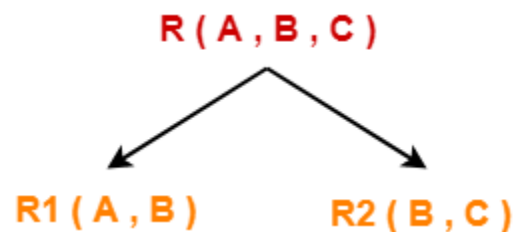
Example-

Consider the following relation **R(A , B , C)--**

A	B	C
1	2	1
2	5	3
3	3	3

R(A , B , C)

Consider this relation is decomposed into two sub relations **R₁(A , B)** and **R₂(B , C)-**



The two sub relations are-

A	B
1	2
2	5
3	3

$R_1(A, B)$

B	C
2	1
5	3
3	3

$R_2(B, C)$

Now, let us check whether this decomposition is lossless or not.

For lossless decomposition, we must have-

$$R_1 \bowtie R_2 = R$$

Now, if we perform the natural join (\bowtie) of the sub relations R_1 and R_2 , we get-

A	B	C
1	2	1
2	5	3
3	3	3

This relation is same as the original relation R .

Thus, we conclude that the above decomposition is lossless join decomposition.

NOTE-

Lossless join decomposition is also known as **non-additive join decomposition**.

- This is because the resultant relation after joining the sub relations is same as the decomposed relation.
- No extraneous tuples appear after joining of the sub-relations.

2. Lossy Join Decomposition-

- Consider there is a relation R which is decomposed into sub relations R_1, R_2, \dots, R_n .
- This decomposition is called lossy join decomposition when the join of the sub relations does not result in the same relation R that was decomposed.
- The natural join of the sub relations is always found to have some extraneous tuples.
- For lossy join decomposition, we always have-

$$R_1 \bowtie R_2 \bowtie R_3 \dots \bowtie R_n \supset R$$

where \bowtie is a natural join operator

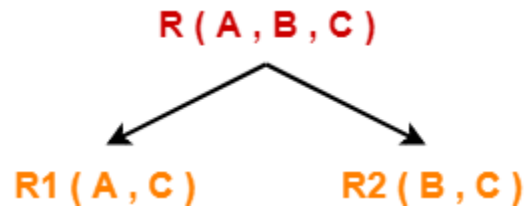
Example-

Consider the following relation R(A , B , C)-

A	B	C
1	2	1
2	5	3
3	3	3

R(A , B , C)

Consider this relation is decomposed into two sub relations as $R_1(A, C)$ and $R_2(B, C)$ -



The two sub relations are-

A	C
1	1
2	3
3	3

$R_1(A, B)$

B	C
2	1
5	3
3	3

$R_2(B, C)$

Now, let us check whether this decomposition is lossy or not.

For lossy decomposition, we must have-

$$R_1 \bowtie R_2 \supset R$$

Now, if we perform the natural join (\bowtie) of the sub relations R_1 and R_2 we get-

A	B	C
1	2	1
2	5	3
2	3	3
3	5	3
3	3	3

This relation is not same as the original relation R and contains some extraneous tuples.

Clearly, $R_1 \bowtie R_2 \supset R$.

Thus, we conclude that the above decomposition is lossy join decomposition.

NOTE-

- Lossy join decomposition is also known as **careless decomposition**.
- This is because extraneous tuples get introduced in the natural join of the sub-relations.
- Extraneous tuples make the identification of the original tuples difficult.

NORMALIZATION

Normalization is a process of organizing the data in database to avoid **data redundancy**, **insertion anomaly**, **update anomaly** & **deletion anomaly**. Let's discuss about anomalies first then we will discuss normal forms with examples.

Anomalies in DBMS

There are three types of anomalies that occur when the database is not normalized. These are – Insertion, update and deletion anomaly. Let's take an example to understand this.

Example: Suppose a manufacturing company stores the employee details in a table named employee that has four attributes: emp_id for storing employee's id, emp_name for storing employee's name, emp_address for storing employee's address and emp_dept for storing the department details in which the employee works. At some point of time the table looks like this:

emp_id	emp_name	emp_address	emp_dept
101	RAM	Delhi	D001
101	RAM	Delhi	D002
123	KRISH	Agra	D890
166	SIVA	Chennai	D900
166	SIVA	Chennai	D004

The above table is not normalized. We will see the problems that we face when a table is not normalized.

Update anomaly: In the above table we have two rows for employee **RAM** as he belongs to **two** departments of the company. If we want to update the address of **RAM** then we have to update the same in two rows or the data will become inconsistent. If somehow, the correct address gets updated in one department but not in other then as per the database, **RAM** would be having two different addresses, which is not correct and would lead to inconsistent data.

Insert anomaly: Suppose a new employee joins the company, who is under training and currently not assigned to any department then we would not be able to insert the data into the table if emp_dept field doesn't allow nulls.

Delete anomaly: Suppose, if at a point of time the company **closes** the department **D890** then deleting the rows that are having emp_dept as D890 would also delete the information of employee **KRISH** since he is assigned only to this department.

To overcome these anomalies we need to normalize the data. In the next section we will discuss about normalization.

Functional Dependency

The attributes of a table is said to be dependent on each other when an attribute of a table uniquely identifies another attribute of the same table.

The functional dependency is a relationship that exists between two attributes. It typically exists between the primary key and non-key attribute within a table.

$$X \rightarrow Y$$

The left side of FD is known as a **determinant**, the right side of the production is known as a **dependent**.

For example:

Assume we have an **employee** table with attributes: **Emp_Id**, **Emp_Name**, **Emp_Address**.

Here **Emp_Id** attribute can uniquely identify the **Emp_Name** attribute of employee table because if we know the **Emp_Id**, we can tell that employee name associated with it.

Functional dependency can be written as:

$$\mathbf{Emp_Id \rightarrow Emp_Name}$$

We can say that **Emp_Name** is functionally dependent on **Emp_Id**.

ID NAME -----	ADDRESS -----
111 RAM	DELHI
222 LAKSH	DELHI
333 VENKAT	MUMBAI
444 KRISH	HYD
555 SIVA	BANGLORE
666 SATYA	CHENNAI

Advantages of Functional Dependency

- Functional Dependency avoids data redundancy. Therefore same data do not repeat at multiple locations in that database
- It helps you to maintain the quality of data in the database

- It helps you to defined meanings and constraints of databases
- It helps you to identify bad designs
- It helps you to find the facts regarding the database design

What is Normalization?

Normalization is a method of organizing the data in the database which helps you to avoid data redundancy, insertion, update and deletion **anomaly** (irregularity). It is a process of analyzing the relation schemas based on their different functional dependencies and primary key.

Normalization is inherent to relational database theory. It may have the effect of duplicating the same data within the database which may result in the creation of additional tables.

- Normalization is the process of organizing the data in the database.
- Normalization is used to minimize the redundancy from a relation or set of relations. It is also used to eliminate the undesirable characteristics like Insertion, Update and Deletion Anomalies.
- Normalization divides the larger table into the smaller table and links them using relationship.
- The normal form is used to reduce redundancy from the database table.

Problems Without Normalization

If a table is not properly normalized and have data redundancy then it will not only eat up extra memory space but will also make it difficult to handle and update the database, without facing data loss. Insertion, Updation and Deletion Anomalies are very frequent if database is not normalized. To understand these anomalies let us take an example of a **Student** table.

rollno	name	branch	hod	office_tel
401	Akon	CSE	Mr. X	53337
402	Bkon	CSE	Mr. X	53337
403	Ckon	CSE	Mr. X	53337
404	Dkon	CSE	Mr. X	53337

In the table above, we have data of 4 Computer Sci. students. As we can see, data for the fields **branch**, **hod**(Head of Department) and **office_tel** is repeated for the students who are in the same branch in the college, this is **Data Redundancy**.

Insertion Anomaly

Suppose for a new admission, until and unless a student opts for a branch, data of the student cannot be inserted, or else we will have to set the branch information as **NULL**.

Also, if we have to insert data of 100 students of same branch, then the branch information will be repeated for all those 100 students.

These scenarios are nothing but **Insertion anomalies**.

Updation Anomaly

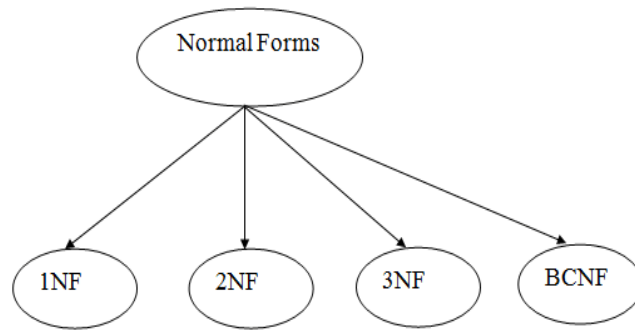
What if Mr. X leaves the college? or is no longer the HOD of computer science department? In that case all the student records will have to be updated, and if by mistake we miss any record, it will lead to data inconsistency. This is Updation anomaly.

Deletion Anomaly

In our **Student** table, two different informations are kept together, Student information and Branch information. Hence, at the end of the academic year, if student records are deleted, we will also lose the branch information. This is Deletion anomaly.

Types of Normal Forms

There are the four types of normal forms:



Normal Form	Description
<u>1NF</u>	A relation is in 1NF if it contains an atomic value.
<u>2NF</u>	A relation will be in 2NF if it is in 1NF and all non-key attributes are fully functional dependent on the primary key.
<u>3NF</u>	A relation will be in 3NF if it is in 2NF and no transitive dependency exists.
<u>4NF</u>	A relation will be in 4NF if it is in Boyce Codd normal form and has no multi-valued dependency.
<u>5NF</u>	A relation is in 5NF if it is in 4NF and not contains any join dependency and joining should be lossless.

Fourth Normal Form (4NF)

Rules for 4th Normal Form

For a table to satisfy the Fourth Normal Form, it should satisfy the following two conditions:

1. It should be in the **Boyce-Codd Normal Form**.
2. And, the table **should not have any Multi-valued Dependency**.

What is Multi-valued Dependency?

A table is said to have multi-valued dependency, if the following conditions are true,

1. For a dependency $A \twoheadrightarrow B$, if for a single value of A , multiple value of B exists, then the table may have multi-valued dependency.
2. Also, a table should have at-least 3 columns for it to have a multi-valued dependency.
3. And, for a relation $R(A, B, C)$, if there is a multi-valued dependency between, A and B , then B and C should be independent of each other.

If all these conditions are true for any relation (table), it is said to have multi-valued dependency.

Example: STUDENT

STU_ID	COURSE	HOBBY
21	Computer	Dancing
21	Math	Singing
34	Chemistry	Dancing
74	Biology	Cricket
59	Physics	Hockey

The given STUDENT table is in 3NF, but the COURSE and HOBBY are two independent entity. Hence, there is no relationship between COURSE and HOBBY.

In the STUDENT relation, a student with STU_ID, **21** contains two courses, **Computer** and **Math** and two hobbies, **Dancing** and **Singing**. So there is a Multi-valued dependency on STU_ID, which leads to unnecessary repetition of data.

So to make the above table into 4NF, we can decompose it into two tables:

STUDENT_COURSE

STU_ID	COURSE
21	Computer
21	Math
34	Chemistry
74	Biology
59	Physics

STUDENT_HOBBY

STU_ID	HOBBY
21	Dancing
21	Singing
34	Dancing
74	Cricket
59	Hockey

Third Normal form (3NF)

A table design is said to be in 3NF if both the following conditions hold:

- Table must be in 2NF
- **Transitive functional dependency** of non-prime attribute on any super key should be removed.

An attribute that is not part of any **candidate key** is known as non-prime attribute.

In other words 3NF can be explained like this: A table is in 3NF if it is in 2NF and for each functional dependency $X \rightarrow Y$ at least one of the following conditions hold:

- X is a **super key** of table
- Y is a prime attribute of table

An attribute that is a part of one of the candidate keys is known as prime attribute.

Example: Suppose a company wants to store the complete address of each employee, they create a table named employee_details that looks like this:

emp_id	emp_name	emp_zip	emp_state	emp_city	emp_district
1001	John	282005	UP	Agra	Dayal Bagh
1002	Ajeet	222008	TN	Chennai	M-City
1006	Lora	282007	TN	Chennai	Urrapakkam
1101	Lilly	292008	UK	Pauri	Bhagwan
1201	Steve	222999	MP	Gwalior	Ratan

Super keys: {emp_id}, {emp_id, emp_name}, {emp_id, emp_name, emp_zip}.. so on

Candidate Keys: {emp_id}

Non-prime attributes: all attributes except emp_id are non-prime as they are not part of any candidate keys.

Here, emp_state, emp_city & emp_district dependent on emp_zip. And, emp_zip is dependent on emp_id that makes non-prime attributes (emp_state, emp_city & emp_district) transitively dependent on super key (emp_id). This violates the rule of 3NF.

To make this table complies with 3NF we have to break the table into two tables **to remove the transitive dependency**:

employee table:

emp_id	emp_name	emp_zip
1001	John	282005
1002	Ajeet	222008
1006	Lora	282007
1101	Lilly	292008
1201	Steve	222999

employee_zip table:

emp_zip	emp_state	emp_city	emp_district
282005	UP	Agra	Dayal Bagh
222008	TN	Chennai	M-City
282007	TN	Chennai	Urrapakkam
292008	UK	Pauri	Bhagwan
222999	MP	Gwalior	Ratan

Second Normal Form (2NF)

For a table to be in the Second Normal Form,

1. It should be in the **First Normal form**.
2. And, it should not have **Partial Dependency**
3. In the second normal form, all non-key attributes are fully functional dependent on the primary key

Example: Let's assume, a school can store the data of teachers and the subjects they teach. In a school, a teacher can teach more than one subject.

TEACHER table

TEACHER_ID	SUBJECT	TEACHER_AGE
25	Chemistry	30
25	Biology	30
47	English	35
83	Math	38
83	Computer	38

In the given table, non-prime attribute TEACHER_AGE is dependent on TEACHER_ID which is a proper subset of a candidate key. That's why it violates the rule for 2NF.

To convert the given table into 2NF, we decompose it into two tables:

TEACHER_DETAIL table:

TEACHER_ID	TEACHER_AGE
25	30
47	35
83	38

TEACHER_SUBJECT table:

TEACHER_ID	SUBJECT
25	Chemistry
25	Biology
47	English
83	Math
83	Computer

First Normal Form (1NF)

For a table to be in the First Normal Form, it should follow the following rules:

- A relation will be 1NF if it contains an **atomic value**.
- It states that an attribute of a table **cannot hold multiple values**. It **must hold only** single-valued attribute.
- First normal form **disallows** the **multi-valued attribute, composite attribute**, and their combinations.

Example: Relation EMPLOYEE is not in 1NF because of multi-valued attribute EMP_PHONE.

EMPLOYEE table:

EMP_ID	EMP_NAME	EMP_PHONE	EMP_STATE
14	John	7272826385, 9064738238	UP
20	Harry	8574783832	Bihar
12	Sam	7390372389, 8589830302	Punjab

The decomposition of the EMPLOYEE table into 1NF has been shown below:

EMP_ID	EMP_NAME	EMP_PHONE	EMP_STATE
14	John	7272826385	UP
14	John	9064738238	UP
20	Harry	8574783832	Bihar
12	Sam	7390372389	Punjab
12	Sam	8589830302	Punjab

Second Normal Form (2NF)

For a table to be in the Second Normal Form,

1. It should be in the First Normal form.
2. And, it should not have Partial Dependency.

What is dependency?

Let's take an example of a student table with columns `student_id`, `name`, `reg_no`(registration number), `branch` and `address`(student's home address).

student_id	name	reg_no	branch	address

In this table, `student_id` is the primary key and will be unique for every row, hence we can use `student_id` to fetch any row of data from this table

Even for a case, where student names are same, if we know the `student_id` we can easily fetch the correct record.

student_id	name	reg_no	branch	address
IT01	RAM	101	CSE	VIJAYAWADA
IT02	LAKSH	102	IT	GUNTUR

Hence we can say a **Primary Key** for a table is the column or a group of columns(composite key) which can uniquely identify each record in the table. I can ask from branch name of student with `student_id` IT101, and I can get it.

Similarly, if I ask for name of student with `student_id` `IT01` or `IT02`, I will get it. So all I need is `student_id` and every other column **depends** on it, or can be fetched using it.

❖ This is **Dependency** and we also call it **Functional Dependency**.

What is Partial Dependency?

For a simple table like `Student`, a single column like `student_id` can uniquely identify all the records in a table.

But this is not true all the time. So now let's extend our example to see if more than 1 column together can act as a primary key.

Let's create another table for **Subject**, which will have `subject_id` and `subject_name` fields and `subject_id` will be the primary key.

subject_id	subject_name
IT01	Java
IT02	C++
IT03	Php

Now we have a **Student** table with student information and another table **Subject** for storing subject information.

Let's create another table **Score**, to store the **marks** obtained by students in the respective subjects. We will also be saving **name of the teacher** who teaches that subject along with marks.

score_id	student_id	subject_id	marks	teacher
1	IT01	1	70	Java Teacher
2	IT02	2	75	C++ Teacher
3	IT02	1	80	Java Teacher

In the score table we are saving the **student_id** to know which student's marks are these and **subject_id** to know for which subject the marks are for.

Together, **student_id + subject_id** forms a **Candidate Key** for this table, which can be the **Primary key**.

if I ask you to get me marks of student with **student_id** IT01, can you get it from this table? No, because you don't know for which subject. And if I give you **subject_id**, you would not know for which student. Hence we need **student_id + subject_id** to uniquely identify any row.

But where is Partial Dependency?

Now if you look at the **Score** table, we have a column names **teacher** which is only dependent on the subject, for Java it's Java Teacher and for C++ it's C++ Teacher & so on.

Now as we just discussed that the primary key for this table is a composition of two columns which is **student_id** & **subject_id** but the teacher's name only depends on subject, hence the **subject_id**, and has nothing to do with **student_id**.

This is **Partial Dependency**, where an attribute in a table depends on only a part of the primary key and not on the whole key.

Boyce Codd normal form (BCNF)

- BCNF is the advance version of 3NF. It is stricter than 3NF.
- A table is in BCNF if every functional dependency $X \rightarrow Y$, X is the super key of the table.
- For BCNF, the table should be in 3NF, and for every FD, LHS is super key.

Example: Let's assume there is a company where employees work in more than one department.

EMPLOYEE table:

EMP_ID	EMP_COUNTRY	EMP_DEPT	DEPT_TYPE	EMP_DEPT_NO
264	India	Designing	D394	283
264	India	Testing	D394	300
364	UK	Stores	D283	232
364	UK	Developing	D283	549

In the above table Functional dependencies are as follows:

$EMP_ID \rightarrow EMP_COUNTRY$
 $EMP_DEPT \rightarrow \{DEPT_TYPE, EMP_DEPT_NO\}$

Candidate key: {EMP-ID, EMP-DEPT}

The table is not in BCNF because neither EMP_DEPT nor EMP_ID alone are keys.

To convert the given table into BCNF, we decompose it into three tables:

EMP_COUNTRY table:

EMP_ID	EMP_COUNTRY
264	India
364	UK

EMP_DEPT table:

EMP_DEPT	DEPT_TYPE	EMP_DEPT_NO
Designing	D394	283
Testing	D394	300
Stores	D283	232
Developing	D283	549

EMP_DEPT_MAPPING table:

EMP_ID	EMP_DEPT
D394	283
D394	300
D283	232
D283	549

Functional dependencies:

EMP_ID → EMP_COUNTRY
EMP_DEPT → {DEPT_TYPE, EMP_DEPT_NO}

Candidate keys:

For the first table: EMP_ID
For the second table: EMP_DEPT
For the third table: {EMP_ID, EMP_DEPT}

Now, this is in BCNF because left side part of both the functional dependencies is a key.

Boyce-Codd Normal Form (BCNF)

Boyce-Codd Normal Form or BCNF is an extension to the [third normal form](#), and is also known as 3.5 Normal Form.

We learned about the third normal form and we also learned how to remove **transitive dependency** from a table

Rules for BCNF

For a table to satisfy the Boyce-Codd Normal Form, it should satisfy the following two conditions:

1. It should be in the **Third Normal Form**.
2. And, for any dependency $A \rightarrow B$, **A** should be a **super key**.

The second point sounds a bit tricky, right? In simple words, it means, that for a dependency $A \rightarrow B$, A cannot be a **non-prime attribute**, if B is a **prime attribute**.

Example

Below we have a college enrolment table with columns **student_id**, **subject** and **professor**.

student_id	subject	professor
101	Java	P.Java
101	C++	P.Cpp
102	Java	P.Java2
103	C#	P.Chash
104	Java	P.Java

As you can see, we have also added some sample data to the table.
In the table above:

- One student can enrol for multiple subjects. For example, student with `student_id` 101, has opted for subjects - Java & C++
- For each subject, a professor is assigned to the student.
- And, there can be multiple professors teaching one subject like we have for Java.

What do you think should be the **Primary Key**?

Well, in the table above `student_id`, `subject` together form the primary key, because using `student_id` and `subject`, we can find all the columns of the table.

One more important point to note here is, one professor teaches only one subject, but one subject may have two different professors.

Hence, there is a dependency between `subject` and `professor` here, where `subject` depends on the professor name.

This table satisfies the **1st Normal form** because all the values are atomic, column names are unique and all the values stored in a particular column are of same domain.

This table also satisfies the **2nd Normal Form** as there is no **Partial Dependency**.

And, there is no **Transitive Dependency**, hence the table also satisfies the **3rd Normal Form**.

But this table is not in **Boyce-Codd Normal Form**.

Why this table is not in BCNF?

In the table above, `student_id`, `subject` form primary key, which means `subject` column is a **prime attribute**.

But, there is one more dependency, `professor` → `subject`.

And while `subject` is a prime attribute, `professor` is a **non-prime attribute**, which is not allowed by BCNF.

How to satisfy BCNF?

To make this relation(table) satisfy BCNF, we will decompose this table into two tables, `student` table and `professor` table.

Below we have the structure for both the tables.

Student Table

student_id	p_id
101	1
101	2
and so on...	

And, Professor Table

p_id	professor	subject
1	P.Java	Java
2	P.Cpp	C++
and so on...		

Operations in Transaction-

The main operations in a transaction are-

1. Read Operation
2. Write Operation

1. Read Operation-

- Read operation reads the data from the database and then stores it in the buffer in main memory.
- For example- **Read(A)** instruction will read the value of A from the database and will store it in the buffer in main memory(RAM).

2. Write Operation-

- Write operation writes the updated data value back to the database from the buffer.
- For example- **Write(A)** will write the updated value of A from the buffer to the database.

Transaction States-

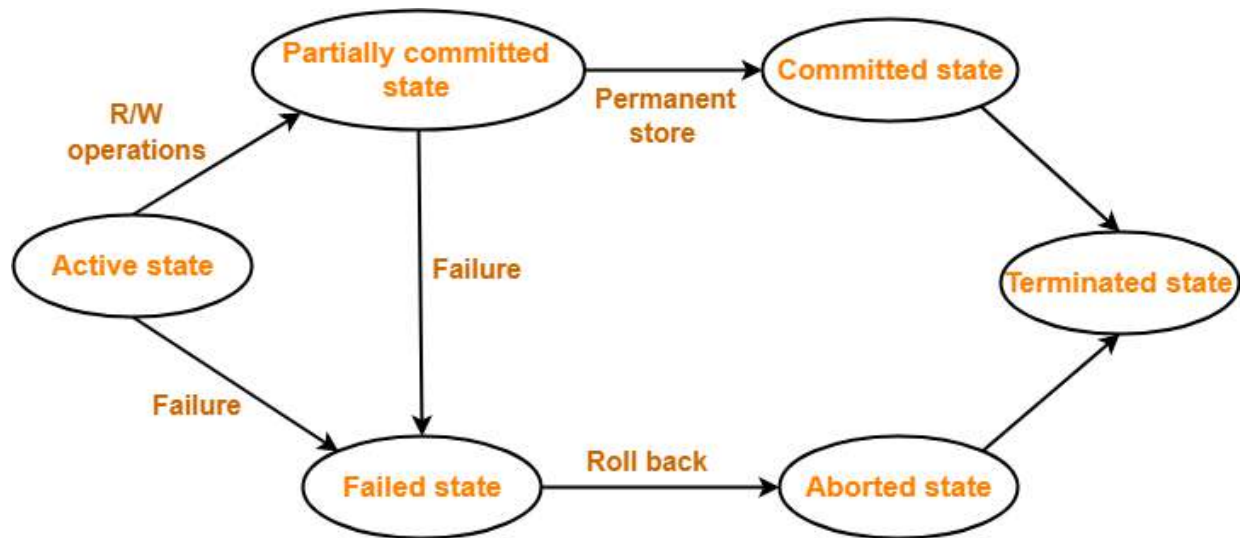
A transaction goes through many different states throughout its life cycle.

These states are called as **transaction states**.

Transaction states are as follows-

1. Active state
2. Partially committed state

3. Committed state
4. Failed state
5. Aborted state
6. Terminated state



Transaction States in DBMS

1. Active State-

- This is the first state in the life cycle of a transaction.
- A transaction is called in an **active state** as long as its instructions are getting executed.
- All the changes made by the transaction now are stored in the buffer in main memory.

2. Partially Committed State-

- After the last instruction of transaction has executed, it enters into a **partially committed state**.

- After entering this state, the transaction is considered to be partially committed.
- It is not considered fully committed because all the changes made by the transaction are still stored in the buffer in main memory.

3. Committed State-

- After all the changes made by the transaction have been successfully stored into the database(RDBMS), it enters into a **committed state**.
- Now, the transaction is considered to be fully committed.

NOTE-

- After a transaction has entered the committed state, it is not possible to roll back the transaction.
- In other words, it is not possible to undo the changes that have been made by the transaction.
- This is because the system is updated into a new consistent state.
- The only way to undo the changes is by carrying out another transaction called as **compensating transaction** that performs the reverse operations.

4. Failed State-

- When a transaction is getting executed in the active state or partially committed state and some failure occurs due to which it

becomes impossible to continue the execution, it enters into a **failed state**.

5. Aborted State-

- After the transaction has failed and entered into a failed state, all the changes made by it have to be undone.
- To undo the changes made by the transaction, it becomes necessary to roll back the transaction.
- After the transaction has rolled back completely, it enters into an **aborted state**.

6. Terminated State-

- This is the last state in the life cycle of a transaction.
- After entering the committed state or aborted state, the transaction finally enters into a **terminated state** where its life cycle finally comes to an end.

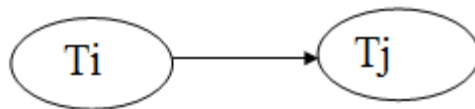
TESTING OF SERIALIZABILITY

Serialization Graph is used to test the Serializability of a schedule.

Assume a schedule S . For S , we construct a graph known as **precedence graph**. This graph has a pair $G = (V, E)$, where V consists a set of vertices, and E consists a set of edges. The set of vertices is used to contain all the transactions participating in the schedule. The set of edges is used to contain all edges $T_i \rightarrow T_j$ for which one of the three conditions holds:

1. Create a node $T_i \rightarrow T_j$ if T_i executes write (Q) before T_j executes read (Q).
2. Create a node $T_i \rightarrow T_j$ if T_i executes read (Q) before T_j executes write (Q).
3. Create a node $T_i \rightarrow T_j$ if T_i executes write (Q) before T_j executes write (Q).

Precedence graph for Schedule S



- If a precedence graph contains a single edge $T_i \rightarrow T_j$, then all the instructions of T_i are executed before the first instruction of T_j is executed.
- If a precedence graph for schedule S contains a cycle, then S is non-serializable. If the precedence graph has no cycle, then S is known as serializable.

For example:

	T1	T2	T3
Time ↓	Read(A)	Read(B)	
	A := f ₁ (A)		
		B := f ₂ (B) Write(B)	Read(C)
	Write(A)		C := f ₃ (C) Write(C)
		Read(A) A := f ₄ (A)	Read(B)
	Read(C) C := f ₅ (C) Write(C)	Write(A)	
		B := f ₆ (B) Write(B)	

Schedule S1

Explanation:

Read(A): In T1, no subsequent writes to A, so no new edges

Read(B): In T2, no subsequent writes to B, so no new edges

Read(C): In T3, no subsequent writes to C, so no new edges

Write(B): B is subsequently read by T3, so add edge T2 → T3

Write(C): C is subsequently read by T1, so add edge T3 → T1

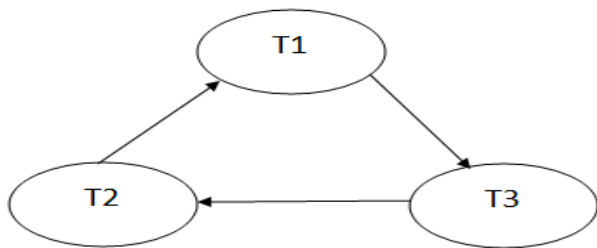
Write(A): A is subsequently read by T2, so add edge T1 → T2

Write(A): In T2, no subsequent reads to A, so no new edges

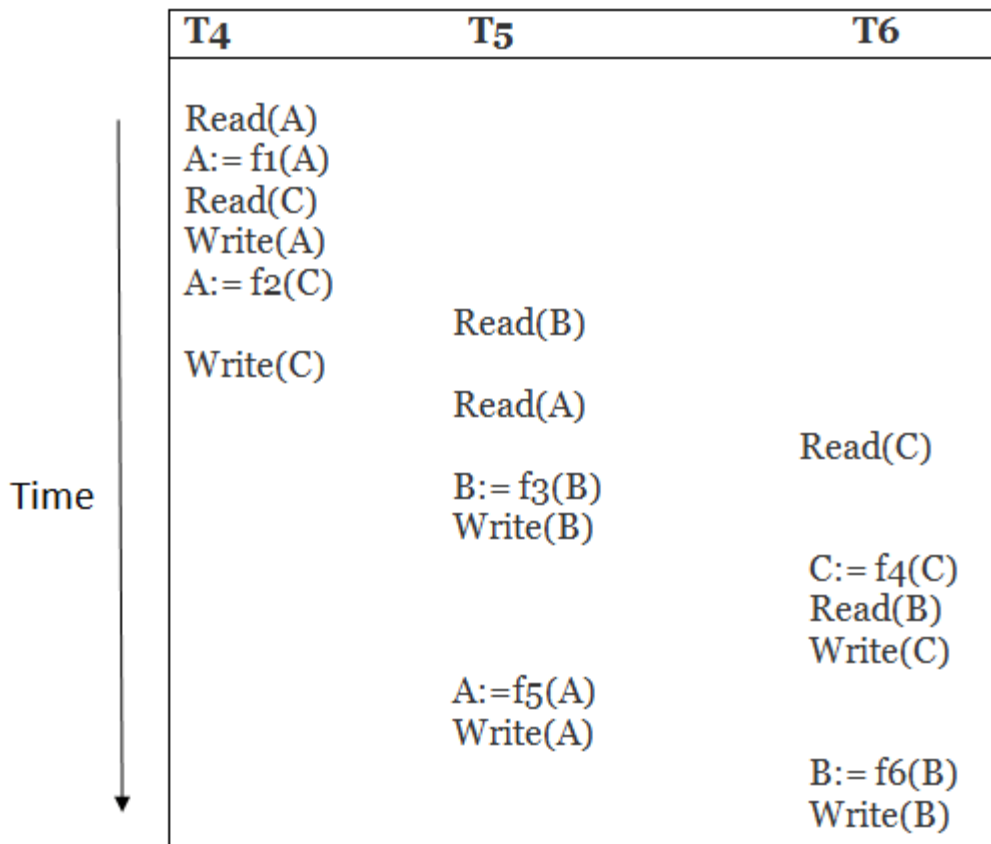
Write(C): In T1, no subsequent reads to C, so no new edges

Write(B): In T3, no subsequent reads to B, so no new edges

Precedence graph for schedule S1:



The precedence graph for schedule S1 contains a cycle that's why Schedule S1 is non-serializable.



Schedule S2

Explanation:

Read(A): In T4, no subsequent writes to A, so no new edges

Read(C): In T4, no subsequent writes to C, so no new edges

Write(A): A is subsequently read by T5, so add edge T4 → T5

Read(B): In T5, no subsequent writes to B, so no new edges

Write(C): C is subsequently read by T6, so add edge T4 → T6

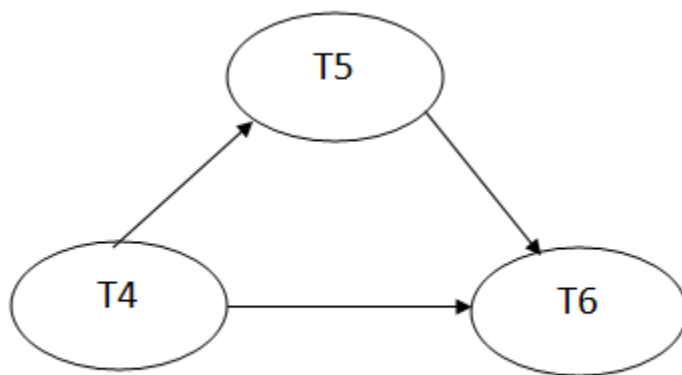
Write(B): A is subsequently read by T6, so add edge T5 → T6

Write(C): In T6, no subsequent reads to C, so no new edges

Write(A): In T5, no subsequent reads to A, so no new edges

Write(B): In T6, no subsequent reads to B, so no new edges

Precedence graph for schedule S2:

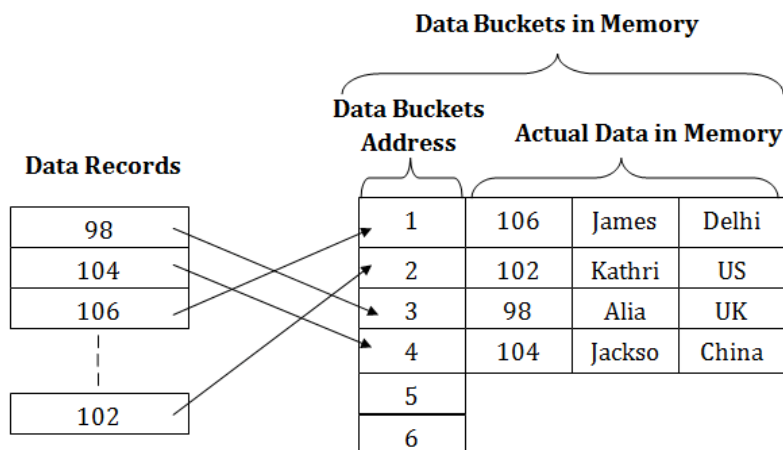
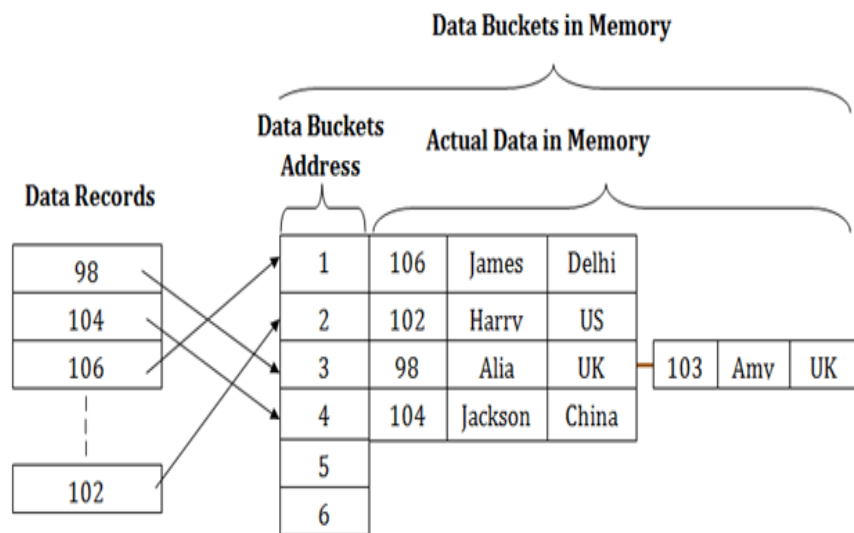


The precedence graph for schedule S2 contains no cycle that's why ScheduleS2 is serializable.

STATIC HASHING

In static hashing, the resultant data bucket address will always be the same. That means if we generate an address for EMP_ID =103 using the hash function mod (5) then it will always result in same bucket address 3. Here, there will be no change in the bucket address.

Hence in this static hashing, the number of data buckets in memory remains constant throughout. In this example, we will have five data buckets in the memory used to store the data.



Operations of Static Hashing

- **Searching a record**

When a record needs to be searched, then the same hash function retrieves the address of the bucket where the data is stored.

- **Insert a Record**

When a new record is inserted into the table, then we will generate an address for a new record based on the hash key and record is stored in that location.

- **Delete a Record**

To delete a record, we will first fetch the record which is supposed to be deleted. Then we will delete the records for that address in memory.

- **Update a Record**

To update a record, we will first search it using a hash function, and then the data record is updated.

If we want to insert some new record into the file but the address of a data bucket generated by the hash function is not empty, or data already exists in that address. This situation in the static hashing is known as **bucket overflow**. This is a critical situation in this method.

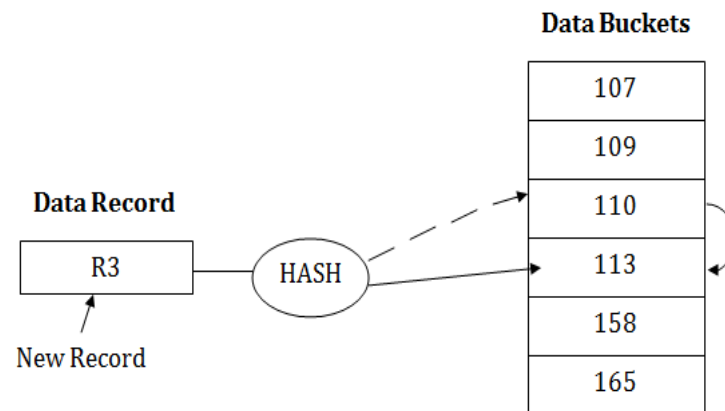
To overcome this situation, there are various methods. Some commonly used methods are as follows:

1. Open Hashing

When a hash function generates an address at which data is already stored, then the next bucket will be allocated to it. This mechanism is called as **Linear Probing**.

For example: suppose R3 is a new address which needs to be inserted, the hash function generates address as 112 for R3. But the generated

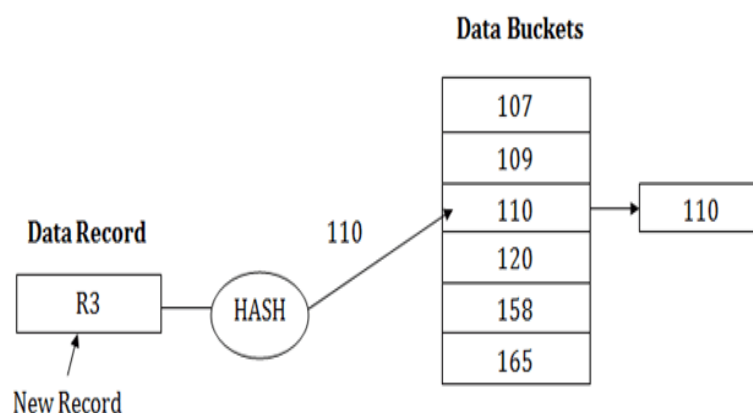
address is already full. So the system searches next available data bucket, 113 and assigns R3 to it.



2. Close Hashing

When buckets are full, then a new data bucket is allocated for the same hash result and is linked after the previous one. This mechanism is known as **Overflow chaining**.

For example: Suppose R3 is a new address which needs to be inserted into the table, the hash function generates address as 110 for it. But this bucket is full to store the new data. In this case, a new bucket is inserted at the end of 110 buckets and is linked to it.



RECOVERABILITY OF SCHEDULE

Sometimes a transaction may not execute completely due to a software issue, system crash or hardware failure. In that case, the failed transaction has to be rollback. But some other transaction may also have used value produced by the failed transaction. So we also have to rollback those transactions.

T1	T1's buffer space	T2	T2's buffer space	Database
				A = 6500
Read(A);	A = 6500			A = 6500
A = A - 500;	A = 6000			A = 6500
Write(A);	A = 6000			A = 6000
		Read(A);	A = 6000	A = 6000
		A = A + 1000;	A = 7000	A = 6000
		Write(A);	A = 7000	A = 7000
		Commit;		
Failure Point				
Commit;				

The above table 1 shows a schedule which has two transactions. T1 reads and writes the value of A and that value is read and written by T2. T2 commits but later on, T1 fails. Due to the failure, we have to rollback T1. T2 should also be rollback because it reads the value written by T1, but T2 can't be rollback because it already committed. So this type of schedule is known as irrecoverable schedule.

Irrecoverable schedule: The schedule will be irrecoverable if Tj reads the updated value of Ti and Tj committed before Ti commit.

T1	T1's buffer space	T2	T2's buffer space	Database
				A = 6500
Read(A);	A = 6500			A = 6500
A = A - 500;	A = 6000			A = 6500
Write(A);	A = 6000			A = 6000
		Read(A);	A = 6000	A = 6000
		A = A + 1000;	A = 7000	A = 6000
		Write(A);	A = 7000	A = 7000
Failure Point				
Commit;				
		Commit;		

The above table 2 shows a schedule with two transactions. Transaction T1 reads and writes A, and that value is read and written by transaction T2. But later on, T1 fails. Due to this, we have to rollback T1. T2 should be rollback because T2 has read the value written by T1. As it has not committed before T1 commits so we can rollback transaction T2 as well. So it is recoverable with cascade rollback.

Recoverable with cascading rollback: The schedule will be recoverable with cascading rollback if Tj reads the updated value of Ti. Commit of Tj is delayed till commit of Ti.

T1	T1's buffer space	T2	T2's buffer space	Database
				A = 6500
Read(A);	A = 6500			A = 6500
A = A - 500;	A = 6000			A = 6500
Write(A);	A = 6000			A = 6000
Commit;		Read(A);	A = 6000	A = 6000
		A = A + 1000;	A = 7000	A = 6000
		Write(A);	A = 7000	A = 7000
		Commit;		

The above Table 3 shows a schedule with two transactions. **Transaction T1 reads and write A and commits,** and **that value is read and written by T2.** So this is a cascade less recoverable schedule.

INDEXING

- Indexing is used to optimize the performance of a database by minimizing the number of disk accesses required when a query is processed.
- The index is a type of data structure. It is used to locate and access the data in a database table quickly.

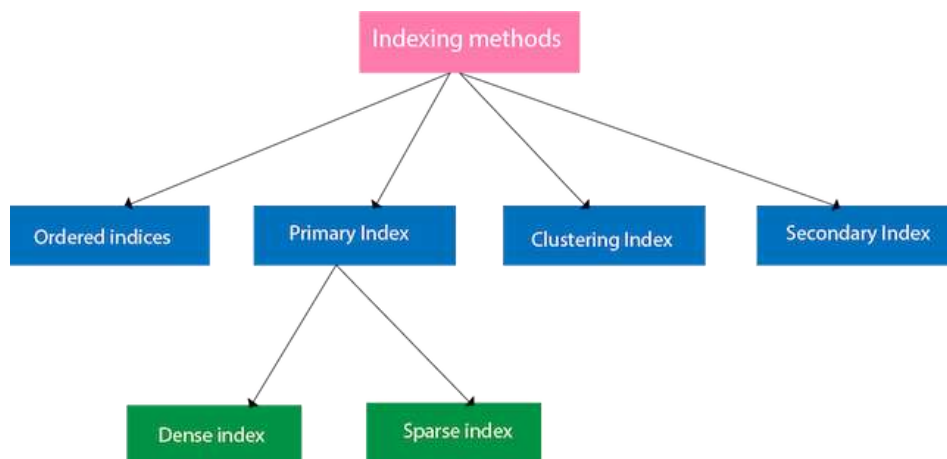
Index structure:

Indexes can be created using some database columns.

SEARCH KEY	DATA REFERENCE
---------------	----------------

- The first column of the database is the search key that contains a copy of the primary key or candidate key of the table. The values of the primary key are stored in sorted order so that the corresponding data can be accessed easily.
- The second column of the database is the data reference. It contains a set of pointers holding the address of the disk block where the value of the particular key can be found.

Indexing Methods



Ordered indices

The indices are usually sorted to make searching faster. The indices which are sorted are known as ordered indices.

Example: Suppose we have an employee table with thousands of record and each of which is 10 bytes long. If their IDs start with 1, 2, 3....and so on and we have to search employee with ID-543.

- In the case of a database with no index, we have to search the disk block from starting till it reaches 543. The DBMS will read the record after reading $543 \times 10 = 5430$ bytes.
- In the case of an index, we will search using indexes and the DBMS will read the record after reading $542 \times 2 = 1084$ bytes which are very less compared to the previous case.

Primary Index

- If the index is created on the basis of the primary key of the table, then it is known as primary indexing. These primary keys are unique to each record and contain 1:1 relation between the records.
- As primary keys are stored in sorted order, the performance of the searching operation is quite efficient.
- The primary index can be classified into two types: Dense index and Sparse index.

Dense index

- The dense index contains an index record for every search key value in the data file. It makes searching faster.
- In this, the number of records in the index table is same as the number of records in the main table.
- It needs more space to store index record itself. The index records have the search key and a pointer to the actual record on the disk.

UP	•	→	UP	Agra	1,604,300
USA	•	→	USA	Chicago	2,789,378
Nepal	•	→	Nepal	Kathmandu	1,456,634
UK	•	→	UK	Cambridge	1,360,364

Sparse index

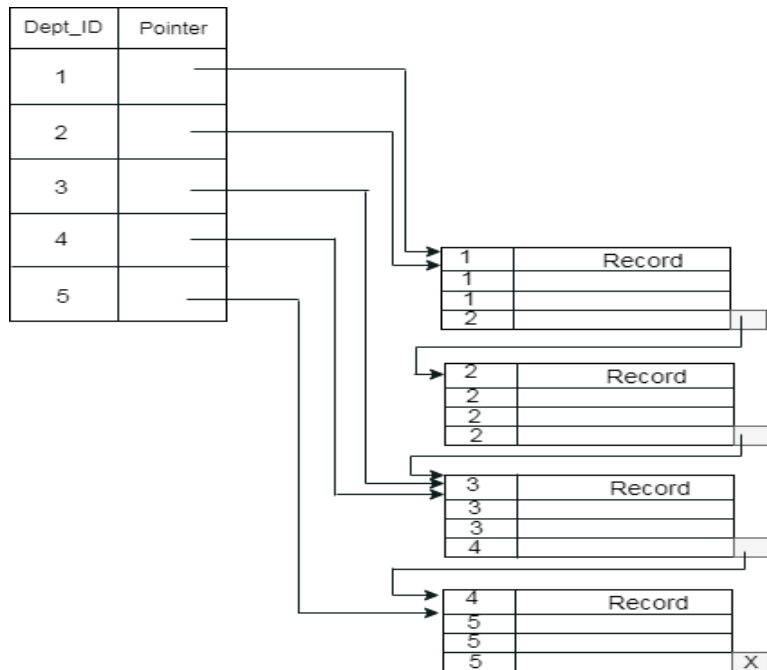
- In the data file, index record appears only for a few items. Each item points to a block.
- In this, instead of pointing to each record in the main table, the index points to the records in the main table in a gap.

UP	•	→	UP	Agra	1,604,300
Nepal	•	→	USA	Chicago	2,789,378
UK	•	→	Nepal	Kathmandu	1,456,634
		→	UK	Cambridge	1,360,364

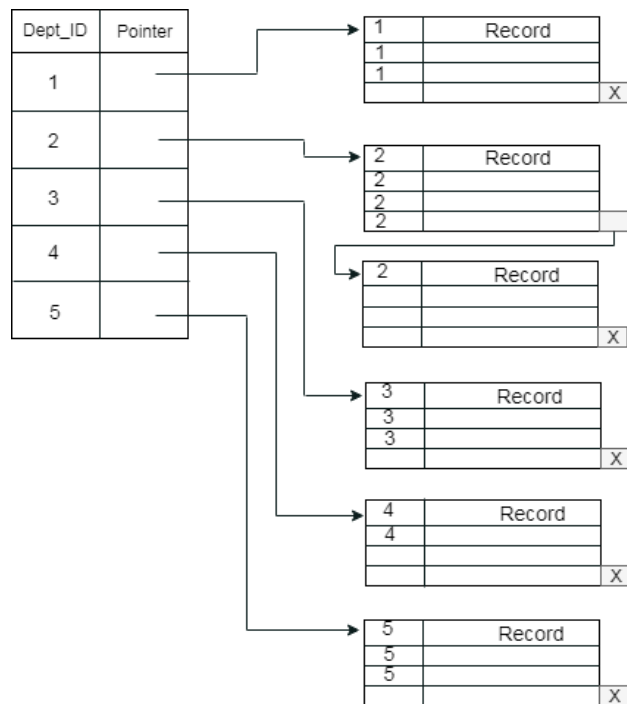
Clustering Index

- A clustered index can be defined as an ordered data file. Sometimes the index is created on non-primary key columns which may not be unique for each record.
- In this case, to identify the record faster, we will group two or more columns to get the unique value and create index out of them. This method is called a clustering index.
- The records which have similar characteristics are grouped, and indexes are created for these group.

Example: suppose a company contains several employees in each department. Suppose we use a clustering index, where all employees which belong to the same Dept_ID are considered within a single cluster, and index pointers point to the cluster as a whole. Here Dept_Id is a non-unique key.



The previous schema is little confusing because one disk block is shared by records which belong to the different cluster. If we use separate disk block for separate clusters, then it is called better technique.



Secondary Index

In the sparse indexing, as the size of the table grows, the size of mapping also grows.

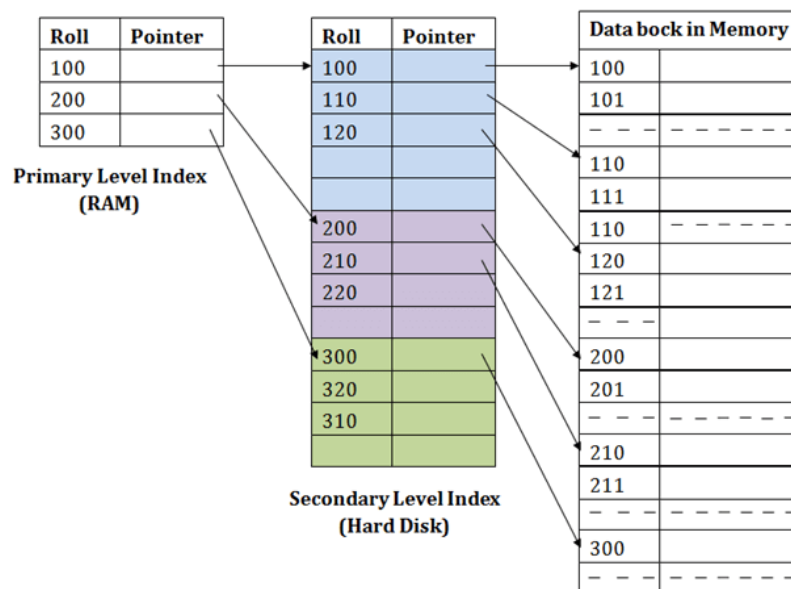
These mappings are usually kept in the primary memory so that address fetch should be faster. Then the secondary memory searches the actual data based on the address got from mapping.

If the mapping size grows then fetching the address itself becomes slower. In this case, the sparse index will not be efficient. To overcome this problem, secondary indexing is introduced.

In secondary indexing, to reduce the size of mapping, another level of indexing is introduced.

In this method, the huge range for the columns is selected initially so that the mapping size of the first level becomes small.

Then each range is further divided into smaller ranges. The mapping of the first level is stored in the primary memory, so that address fetch is faster. The mapping of the second level and actual data are stored in the secondary memory (hard disk).



For example:

- If you want to find the record of roll 111 in the diagram, then it will search the highest entry which is smaller than or equal to 111 in the first level index. It will get 100 at this level.
- Then in the second index level, again it does $\max(111) \leq 111$ and gets 110. Now using the address 110, it goes to the data block and starts searching each record till it gets 111.
- This is how a search is performed in this method. Inserting, updating or deleting is also done in the same manner.

Advantages of Indexing

Important pros/ advantage of Indexing are:

- It helps you to reduce the total number of I/O operations needed to retrieve that data, so you don't need to access a row in the database from an index structure.
- Offers Faster search and retrieval of data to users.
- Indexing also helps you to reduce tablespace as you don't need to link to a row in a table, as there is no need to store the ROWID in the Index. Thus you will be able to reduce the tablespace.
- You can't sort data in the leaf nodes as the value of the primary key classifies it.

Disadvantages of Indexing

Important drawbacks/cons of Indexing are:

- To perform the indexing database management system, you need a primary key on the table with a unique value.
- You can't perform any other indexes in Database on the Indexed data.
- You are not allowed to partition an index-organized table.
- SQL Indexing Decrease performance in INSERT, DELETE, and UPDATE query.

HASHING

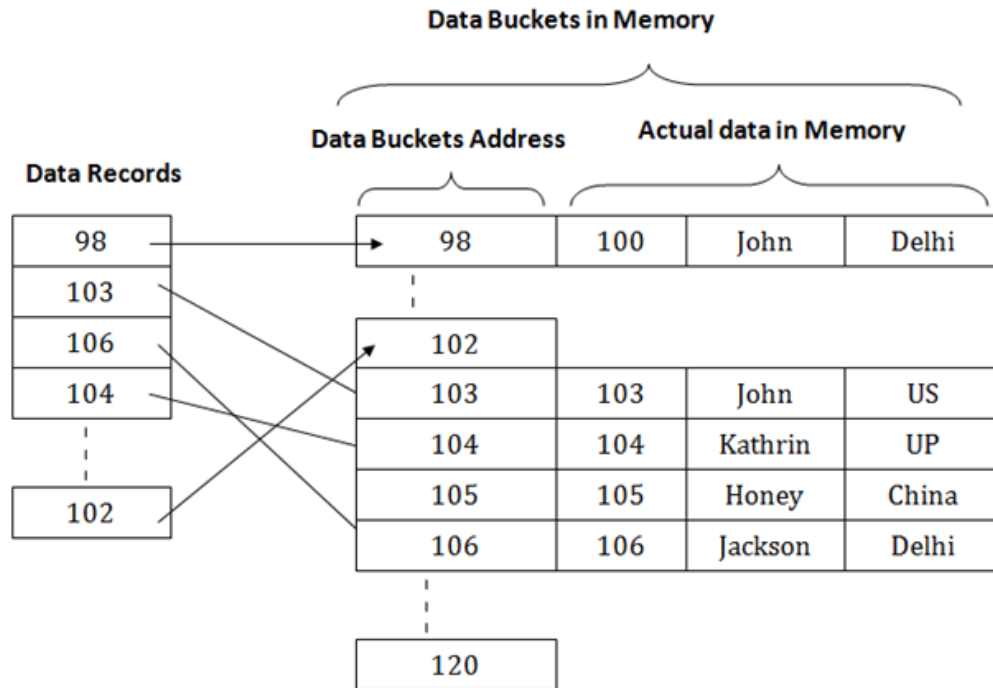
Hashing involves less key comparison and searching can be performed in constant time.

The goal of a hashed search is to find the largest data in only one test.

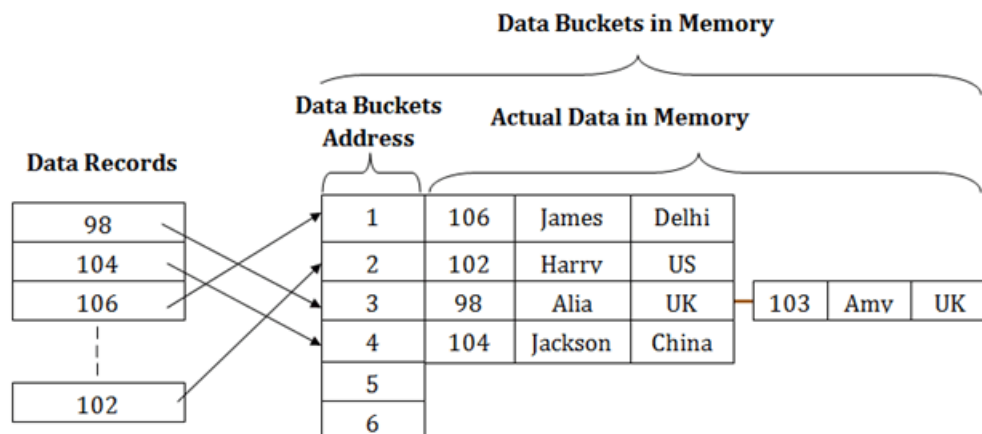
In a huge database structure, it is very inefficient to search all the index values and reach the desired data. Hashing technique is used to calculate the direct location of a data record on the disk without using index structure.

In this technique, data is stored at the data blocks whose address is generated by using the hashing function. The memory location where these records are stored is known as data bucket or data blocks.

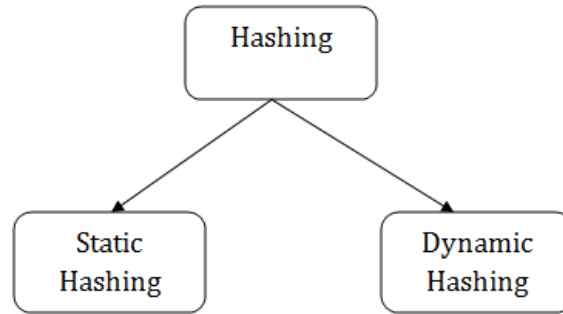
In this, a hash function can choose any of the column value to generate the address. Most of the time, the hash function uses the primary key to generate the address of the data block. A hash function is a simple mathematical function to any complex mathematical function. We can even consider the primary key itself as the address of the data block. That means each row whose address will be the same as a primary key stored in the data block.



The above diagram shows data block addresses same as primary key value. This hash function can also be a simple mathematical function like exponential, mod, cos, sin, etc. Suppose we have mod (5) hash function to determine the address of the data block. In this case, it applies mod (5) hash function on the primary keys and generates 3, 3, 1, 4 and 2 respectively, and records are stored in those data block addresses.



Types of Hashing:



File Organization in DBMS

A database consists of a huge amount of data. The data is grouped within a table in RDBMS, and each table has related records. A user can see that the data is stored in form of tables, but in actual this huge amount of data is stored in physical memory in form of files.

File - A file is named collection of related information that is recorded on secondary storage such as magnetic disks, magnetic tapes and optical disks.

What is File Organization?

File Organization refers to the logical relationships among various records that constitute the file, particularly with respect to the means of identification and access to any specific record. In simple terms, **Storing the files in certain order is called file Organization**. File Structure refers to the format of the label and data blocks and of any logical control record.

Types of File Organizations -

Various methods have been introduced to Organize files. These particular methods have advantages and disadvantages on the basis of access or selection. Thus it is all upon the programmer to decide the best suited file Organization method according to his requirements.

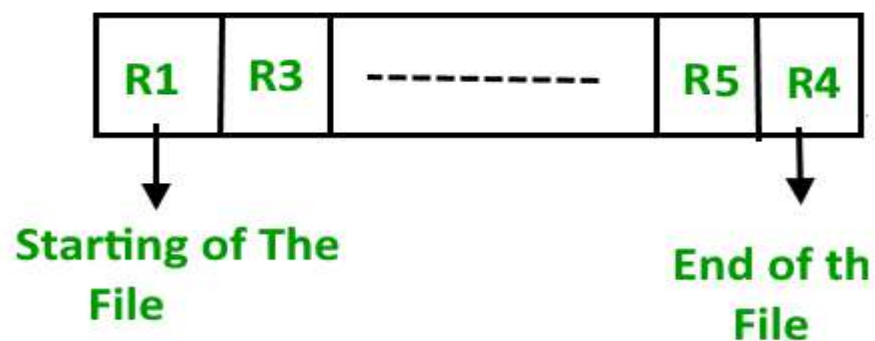
Some types of File Organizations are:

- Sequential File Organization
- Heap File Organization
- Hash File Organization
- B+ Tree File Organization
- Clustered File Organization

Sequential File Organization -

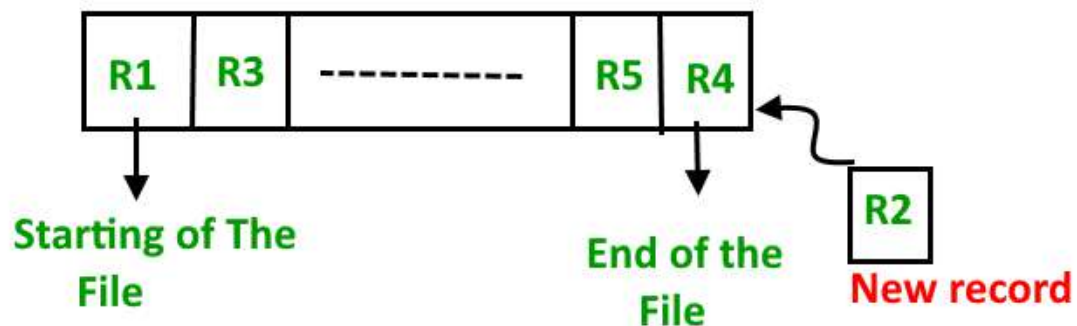
The easiest method for file Organization is Sequential method. In this method the file are stored one after another in a sequential manner. There are two ways to implement this method:

1. **Pile File Method** - This method is quite simple, in which we store the records in a sequence i.e one after other in the order in which they are inserted into the tables.



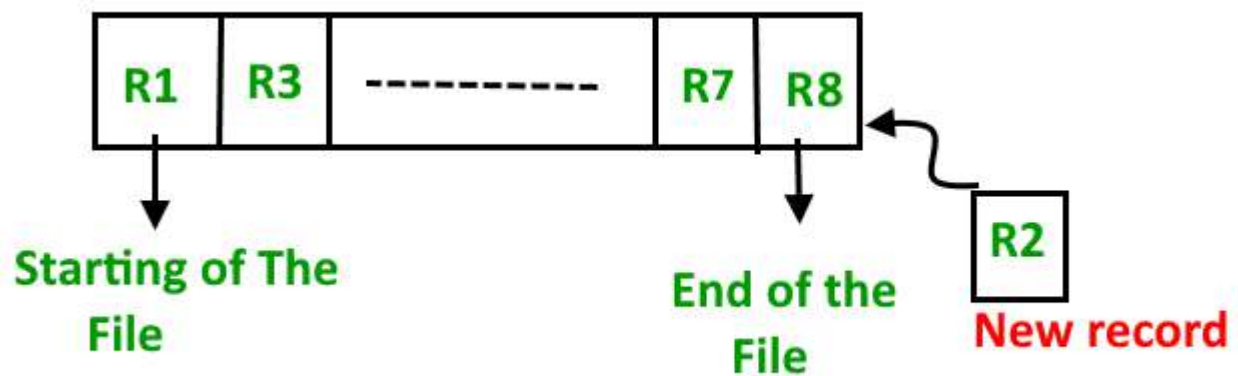
Insertion of new record -

Let the R1, R3 and so on upto R5 and R4 be four records in the sequence. Here, records are nothing but a row in any table. Suppose a new record R2 has to be inserted in the sequence, and then it is simply placed at the end of the file.



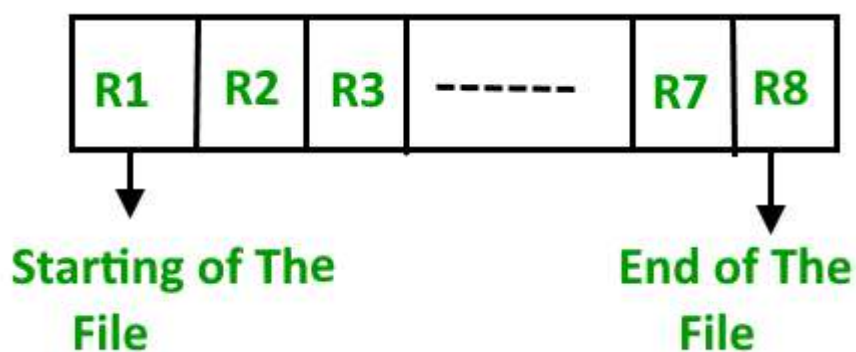
2. Sorted File Method -

In this method, As the name itself suggest whenever a new record has to be inserted, it is always inserted in a sorted (ascending or descending) manner. Sorting of records may be based on any primary key or any other key.



Insertion of new record -

Let us assume that there is a pre-existing sorted sequence of four records R1, R3, and so on up to R7 and R8. Suppose a new record R2 has to be inserted in the sequence, then it will be inserted at the end of the file and then it will sort the sequence.



Pros and Cons of Sequential File Organization -

Pros -

- Fast and efficient method for huge amount of data.
- Simple design.

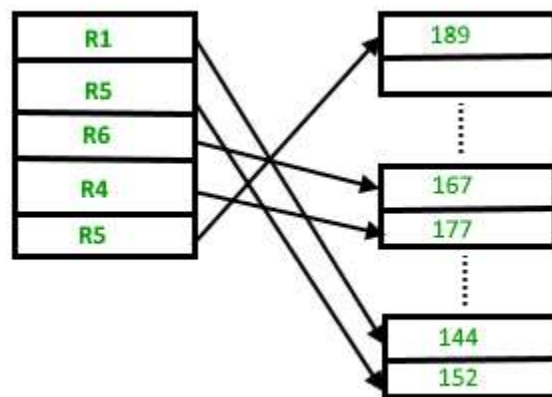
- Files can be easily stored in magnetic tapes i.e cheaper storage mechanism.

Cons -

- Time wastage as we cannot jump on a particular record that is required, but we have to move in a sequential manner which takes our time.
- Sorted file method is inefficient as it takes time and space for sorting records.

Heap File Organization -

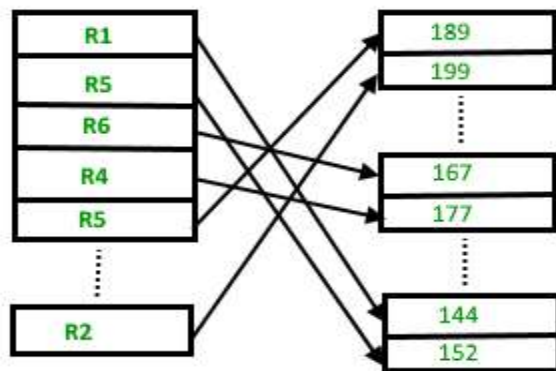
Heap File Organization works with data blocks. In this method records are inserted at the end of the file, into the data blocks. **No Sorting or Ordering is required in this method.** If a data block is full, the new record is stored in some other block, here **the other data block need not be the very next data block**, but it can be any block in the memory. It is the responsibility of DBMS to store and manage the new records.



Insertion of new record -

Suppose we have four records in the heap R1, R5, R6, R4 and R3 and suppose a new record R2 has to be inserted in the heap then, since

the last data block i.e. data block 3 is full it will be inserted in any of the data blocks selected by the DBMS, let's say data block 1.



If we want to search, delete or update data in heap file Organization then we will traverse the data from the beginning of the file till we get the requested record. Thus if the database is very huge, searching, deleting or updating the record will take a lot of time.

Pros and Cons of Heap File Organization -

Pros -

- Fetching and retrieving records is faster than sequential record but only in case of small databases.
- When there is a huge number of data needs to be loaded into the database at a time, then this method of file Organization is best suited.

Cons -

- Problem of unused memory blocks.
- Inefficient for larger databases.

Prerequisite - [Hashing Data Structure](#)

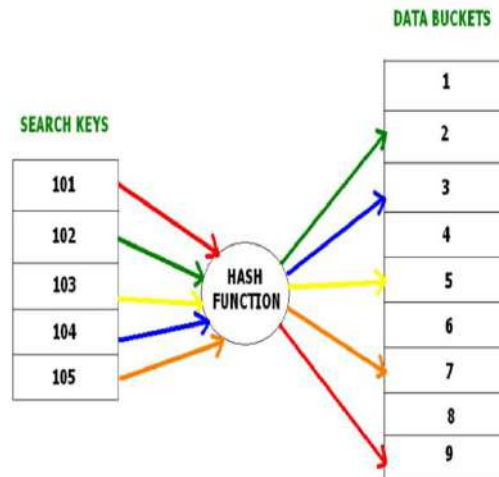
In database management system, When we want to retrieve a particular data, It becomes very inefficient to search all the index values and reach the desired data. In this situation, Hashing technique comes into picture.

Hashing is an efficient technique to directly search the location of desired data on the disk without using index structure. Data is stored at the data blocks whose address is generated by using hash function. The memory location where these records are stored is called as data block or data bucket.

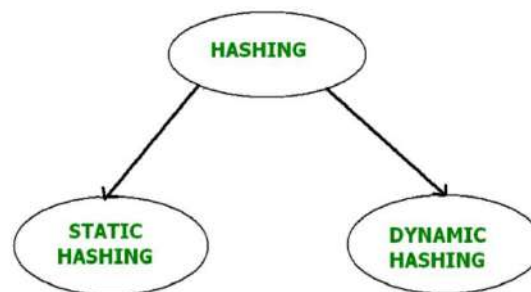
Hash File Organization:

- **Data bucket** - Data buckets are the memory locations where the records are stored. These buckets are also considered as *Unit Of Storage*.
- **Hash Function** - Hash function is a mapping function that maps all the set of search keys to actual record address. Generally, hash function uses primary key to generate the hash index - address of the data block. Hash function can be simple mathematical function to any complex mathematical function.
- **Hash Index**-The prefix of an entire hash value is taken as a hash index. Every hash index has a depth value to signify how many bits are used for computing a hash function. These bits can address 2^n buckets. When all these bits are consumed? Then the depth value is increased linearly and twice the buckets are allocated.

Below given diagram clearly depicts how hash function work:



Hashing is further divided into two sub categories:



Static Hashing -

In static hashing, when a search-key value is provided, the hash function always computes the same address. For example, if we want to generate address for STUDENT_ID = 76 using mod (5) hash function, it always result in the same bucket address 4. There will not be any changes to the bucket address here. Hence number of data buckets in the memory for this static hashing remains constant throughout.

Operations -

- **Insertion** - When a new record is inserted into the table, The hash function h generate a bucket address for the new record based on its hash key K .
Bucket address = $h(K)$
- **Searching** - When a record needs to be searched, The same hash function is used to retrieve the bucket address for the record. For

Example, if we want to retrieve whole record for ID 76, and if the hash function is mod (5) on that ID, the bucket address generated would be 4. Then we will directly go to address 4 and retrieve the whole record for ID 104. Here ID acts as a hash key.

- **Deletion** - If we want to delete a record, Using the hash function we will first fetch the record which is supposed to be deleted. Then we will remove the records for that address in memory.
- **Updation** - The data record that needs to be updated is first searched using hash function, and then the data record is updated.

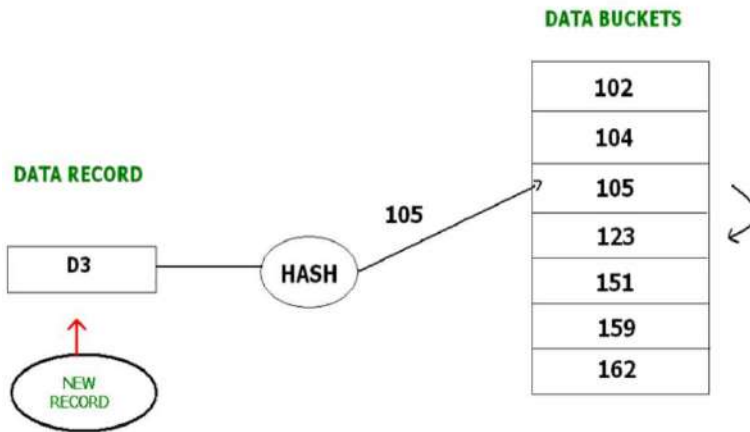
Now, If we want to insert some new records into the file But the data bucket address generated by the hash function is not empty or the data already exists in that address. This becomes a critical situation to handle. This situation in the static hashing is called **bucket overflow**.

How will we insert data in this case? There are several methods provided to overcome this situation. Some commonly used methods are discussed below:

1. Open Hashing -

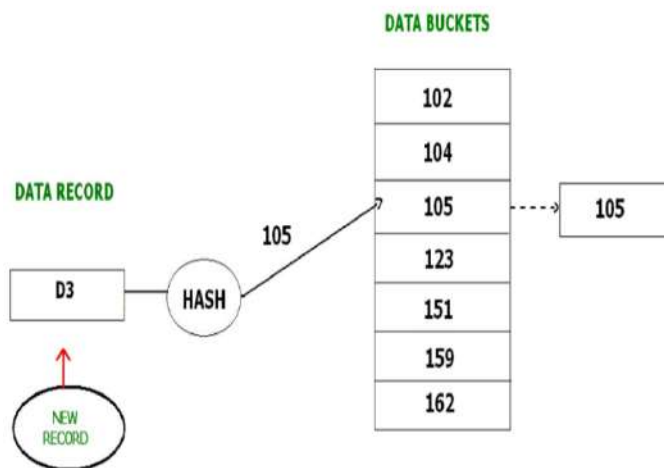
In Open hashing method, next available data block is used to enter the new record, instead of overwriting older one. This method is also called linear probing.

For example, D3 is a new record which needs to be inserted, the hash function generates address as 105. But it is already full. So the system searches next available data bucket, 123 and assigns D3 to it.



2. Closed hashing -

In Closed hashing method, a new data bucket is allocated with same address and is linked it after the full data bucket. This method is also known as overflow chaining. For example, we have to insert a new record D3 into the tables. The static hash function generates the data bucket address as 105. But this bucket is full to store the new data. In this case is a new data bucket is added at the end of 105 data bucket and is linked to it. Then new record D3 is inserted into the new bucket.



- **Quadratic probing :**

Quadratic probing is very much similar to open hashing or linear probing. Here, The only difference between old and new bucket is linear. Quadratic function is used to determine the new bucket address.

- **Double Hashing:**

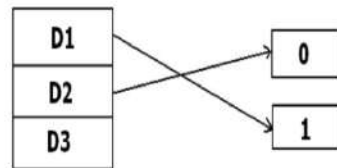
Double Hashing is another method similar to linear probing. Here the difference is fixed as in linear probing, but this fixed difference is calculated by using another hash function. That's why the name is double hashing.

Dynamic Hashing –

The drawback of static hashing is that that it does not expand or shrink dynamically as the size of the database grows or shrinks. In Dynamic hashing, data buckets grows or shrinks (added or removed dynamically) as the records increases or decreases. Dynamic hashing is also known as extended hashing.

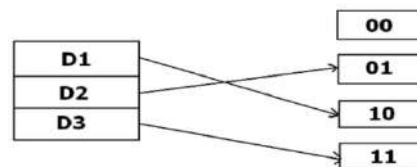
In dynamic hashing, the hash function is made to produce a large number of values. For Example, there are three data records D1, D2 and D3 . The hash function generates three addresses 1001, 0101 and 1010 respectively. This method of storing considers only part of this address – especially only first one bit to store the data. So it tries to load three of them at address 0 and 1.

$h(D1) \rightarrow 1001$
 $h(D2) \rightarrow 0101$
 $h(D3) \rightarrow 1010$



But the problem is that No bucket address is remaining for D3. The bucket has to grow dynamically to accommodate D3. So it changes the address have 2 bits rather than 1 bit, and then it updates the existing data to have 2 bit address. Then it tries to accommodate D3.

$h(D1) \rightarrow 1001$
 $h(D2) \rightarrow 0101$
 $h(D3) \rightarrow 1010$



FAILURE CLASSIFICATION

To find that where the problem has occurred, we generalize a failure into the following categories:

1. Transaction failure
2. System crash
3. Disk failure

1. Transaction failure

The transaction failure occurs when it fails to execute or when it reaches a point from where it can't go any further. If a few transaction or process is hurt, then this is called as transaction failure.

Reasons for a transaction failure could be -

1. **Logical errors:** If a transaction cannot complete due to some code error or an internal error condition, then the logical error occurs.
2. **Syntax error:** It occurs where the DBMS itself terminates an active transaction because the database system is not able to execute it. **For example,** the system aborts an active transaction, in case of deadlock or resource unavailability.

2. System Crash

- System failure can occur due to power failure or other hardware or software failure. **Example:** Operating system error.

Fail-stop assumption: In the system crash, non-volatile storage is assumed not to be corrupted.

3. Disk Failure

- It occurs where hard-disk drives or storage drives used to fail frequently. It was a common problem in the early days of technology evolution.

- Disk failure occurs due to the formation of bad sectors, disk head crash, and unreachability to the disk or any other failure, which destroy all or part of disk storage.

DYNAMIC HASHING

- The Dynamic Hashing method is used to overcome the problems of static hashing like **bucket overflow**.
- In this method, **data buckets grow or shrink as the records increases or decreases**. This method is also known as **extendable hashing method**.
- This method makes hashing dynamic, i.e., it allows insertion or deletion without resulting in poor performance.

How to search a key

- First, calculate the hash address of the key.
- Check how many bits are used in the directory, and these bits are called as **i**.
- Take the least significant i bits of the hash address. This gives an index of the directory.
- Now using the index, go to the directory and find bucket address where the record might be.

How to insert a new record

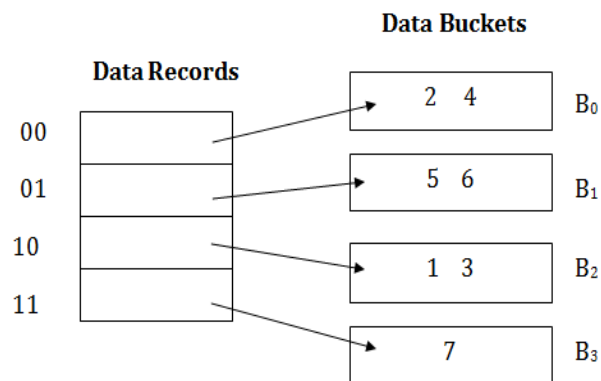
- Firstly, you have to follow the same procedure for retrieval, ending up in some bucket.
- If there is still space in that bucket, then place the record in it.
- If the bucket is full, then we will split the bucket and redistribute the records.

For example:

Consider the following grouping of keys into buckets, depending on the prefix of their hash address:

Key	Hash Address
1	11010
2	00000
3	11110
4	00000
5	01001
6	10101
7	10111

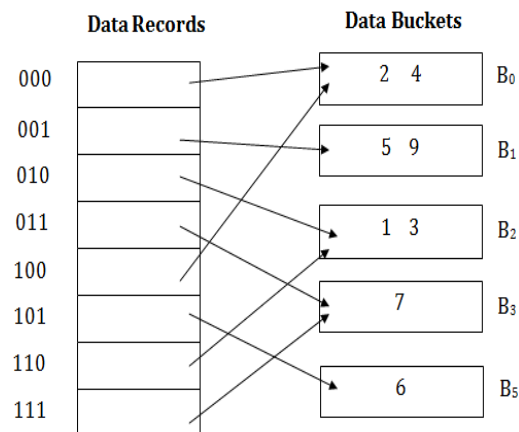
The last two bits of 2 and 4 are 00. So it will go into bucket B0.
The last two bits of 5 and 6 are 01, so it will go into bucket B1.
The last two bits of 1 and 3 are 10, so it will go into bucket B2.
The last two bits of 7 are 11, so it will go into B3.



Insert key 9 with hash address 10001 into the above structure:

- Since key 9 has hash address 10001, it must go into the first bucket. But bucket B1 is full, so it will get split.
- The splitting will separate 5, 9 from 6 since last three bits of 5, 9 are 001, so it will go into bucket B1, and the last three bits of 6 are 101, so it will go into bucket B5.
- Keys 2 and 4 are still in B0. The record in B0 pointed by the 000 and 100 entry because last two bits of both the entry are 00.
- Keys 1 and 3 are still in B2. The record in B2 pointed by the 010 and 110 entry because last two bits of both the entry are 10.

- Key 7 are still in B3. The record in B3 pointed by the 111 and 011 entry because last two bits of both the entry are 11.



Advantages of dynamic hashing

- In this method, the performance does not decrease as the data grows in the system. It simply increases the size of memory to accommodate the data.
- In this method, memory is well utilized as it grows and shrinks with the data. There will not be any unused memory lying.
- This method is good for the dynamic database where data grows and shrinks frequently.

Disadvantages of dynamic hashing

- In this method, if the data size increases then the bucket size is also increased. These addresses of data will be maintained in the bucket address table. This is because the data address will keep changing as buckets grow and shrink. If there is a huge increase in data, maintaining the bucket address table becomes tedious.
- In this case, the bucket overflow situation will also occur. But it might take little time to reach this situation than static hashing.

DBMS SERIALIZABILITY

When multiple transactions are running concurrently then there is a possibility that the database may be left in an inconsistent state. Serializability is a concept that helps us to check which schedules are serializable. A serializable schedule is the one that always leaves the database in consistent state.

A serial schedule doesn't allow concurrency, only one transaction executes at a time and the other starts when the already running transaction finished.

Types of Serializability

There are two types of Serializability.

1. Conflict Serializability
2. View Serializability

1. Conflict Serializable Schedule

- A schedule is called conflict serializability if after swapping of non-conflicting operations, it can transform into a serial schedule.
- The schedule will be a conflict serializable if it is conflict equivalent to a serial schedule.

Conflicting Operations

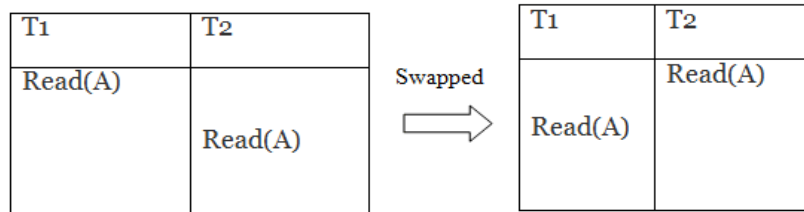
The two operations become conflicting if all conditions satisfy:

1. Both belong to separate transactions.
2. They have the same data item.
3. They contain at least one write operation.

Example:

Swapping is possible only if S1 and S2 are logically equal.

1. T1: Read(A) T2: Read(A)

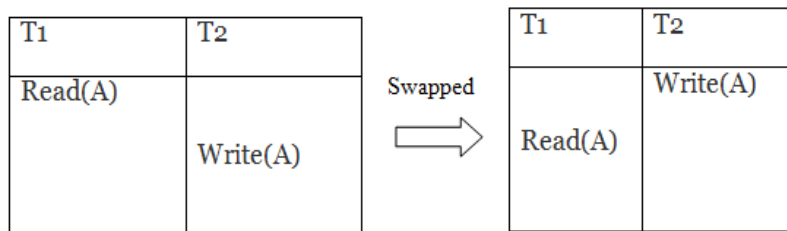


Schedule S1

Schedule S2

Here, $S1 = S2$. That means it is non-conflict.

2. T1: Read(A) T2: Write(A)



Schedule S1

Schedule S2

Here, $S1 \neq S2$. That means it is conflict.

Conflict Equivalent

In the conflict equivalent, one can be transformed to another by swapping non-conflicting operations. In the given example, S2 is conflict equivalent to S1 (S1 can be converted to S2 by swapping non-conflicting operations).

Two schedules are said to be conflict equivalent if and only if:

1. They contain the same set of the transaction.
2. If each pair of conflict operations are ordered in the same way.

Example:

Non-serial schedule

T1	T2
Read(A) Write(A)	
	Read(A) Write(A)
Read(B) Write(B)	
	Read(B) Write(B)

Schedule S1

Serial Schedule

T1	T2
Read(A) Write(A) Read(B) Write(B)	
	Read(A) Write(A) Read(B) Write(B)

Schedule S2

Schedule S2 is a serial schedule because, in this, all operations of T1 are performed before starting any operation of T2. Schedule S1 can be transformed into a serial schedule by swapping non-conflicting operations of S1.

After swapping of non-conflict operations, the schedule S1 becomes:

T1	T2
Read(A) Write(A) Read(B) Write(B)	
	Read(A) Write(A) Read(B) Write(B)

Since, S1 is conflict serializable.

2. View Serializability

- A schedule will view serializable if it is view equivalent to a serial schedule.
- If a schedule is conflict serializable, then it will be view serializable.
- The view serializable which does not conflict serializable contains blind writes.

View Equivalent

Two schedules S1 and S2 are said to be view equivalent if they satisfy the following conditions:

1. Initial Read

An initial read of both schedules must be the same. Suppose two schedule S1 and S2. In schedule S1, if a transaction T1 is reading the data item A, then in S2, transaction T1 should also read A.

T1	T2
Read(A)	Write(A)

Schedule S1

T1	T2
Read(A)	Write(A)

Schedule S2

Above two schedules are view equivalent because Initial read operation in S1 is done by T1 and in S2 it is also done by T1.

2. Updated Read

In schedule S1, if Ti is reading A which is updated by Tj then in S2 also, Ti should read A which is updated by Tj.

T1	T2	T3
Write(A)	Write(A)	Read(A)

Schedule S1

T1	T2	T3
Write(A)	Write(A)	<u>Read(A)</u>

Schedule S2

Above two schedules are not view equal because, in S1, T3 is reading A updated by T2 and in S2, T3 is reading A updated by T1.

3. Final Write

A final write must be the same between both the schedules. In schedule S1, if a transaction T1 updates A at last then in S2, final writes operations should also be done by T1.

T1	T2	T3
Write(A)	Read(A)	Write(A)

Schedule S1

T1	T2	T3
Write(A)	Read(A)	Write(A)

Schedule S2

Above two schedules is view equal because Final write operation in S1 is done by T3 and in S2, the final write operation is also done by T3.

Example:

T1	T2	T3
Read(A) Write(A)	Write(A)	Write(A)

Schedule S

With 3 transactions, the total number of possible schedule
= $3! = 6$

S1 = <T1 T2 T3> S2 = <T1 T3 T2> S3 = <T2 T3 T1> S4 = <T2 T1 T3>
S5 = <T3 T1 T2> S6 = <T3 T2 T1>

Taking first schedule S1:

T1	T2	T3
Read(A) Write(A)	Write(A)	Write(A)

Schedule S1

Step 1: final updation on data items

In both schedules S and S1, there is no read except the initial read that's why we don't need to check that condition.

Step 2: Initial Read

The initial read operation in S is done by T1 and in S1, it is also done by T1.

Step 3: Final Write

The final write operation in S is done by T3 and in S1, it is also done by T3. So, S and S1 are view Equivalent.

The first schedule S1 satisfies all three conditions, so we don't need to check another schedule.

Hence, view equivalent serial schedule is:

T1 → T2 → T3

Concurrency Control

Concurrency = Done At the Same Time

- In the concurrency control, the multiple transactions can be executed simultaneously.
- It may affect the transaction result. It is highly important to maintain the order of execution of those transactions.

Problems of concurrency control:

Several problems can occur when concurrent (*done at the same time.*) transactions are executed in an uncontrolled manner. Following are the three problems in concurrency control.

1. Lost updates
2. Dirty read (uncommitted data)
3. Unrepeatable read

1. Lost update problem

- When two transactions that access the same database items contain their operations in a way that makes the value of some database item incorrect, then the lost update problem occurs.
- If two transactions T1 and T2 read a record and then update it, then the effect of updating of the first record will be overwritten by the second update.

Example:

Transaction-X	Time	Transaction-Y
—	t1	—
Read A	t2	—
—	t3	Read A
Update A	t4	—
—	t5	Update A
—	t6	—

Here,

- At time t2, transaction-X reads A's value.
- At time t3, Transaction-Y reads A's value.
- At time t4, Transactions-X writes A's value on the basis of the value seen at time t2.

- At time t5, Transactions-Y writes A's value on the basis of the value seen at time t3.
- So at time T5, the update of Transaction-X is lost because Transaction y overwrites it without looking at its current value.
- Such type of problem is known as Lost Update Problem as update made by one transaction is lost here.

2. Dirty Read

- The dirty read occurs in the case when one transaction updates an item of the database, and then the transaction fails for some reason. The updated database item is accessed by another transaction before it is changed back to the original value.
- A transaction T1 updates a record which is read by T2. If T1 aborts then T2 now has values which have never formed part of the stable database.

Example:

Transaction-X	Time	Transaction-Y
—	t1	—
—	t2	Update A
Read A	t3	—
—	t4	Rollback
—	t5	—

- At time t2, transaction-Y writes A's value.
- At time t3, Transaction-X reads A's value.
- At time t4, Transactions-Y rollbacks. So, it changes A's value back to that of prior to t1.
- So, Transaction-X now contains a value which has never become part of the stable database.
- Such type of problem is known as Dirty Read Problem, as one transaction reads a dirty value which has not been committed.

3. Inconsistent Retrievals Problem

- Inconsistent Retrievals Problem is also known as unrepeatable read (not able to be done or made again). When a transaction calculates some summary function over a set of data while the

other transactions are updating the data, then the Inconsistent Retrievals Problem occurs.

- A transaction T1 reads a record and then does some other processing during which the transaction T2 updates the record. Now when the transaction T1 reads the record, then the new value will be inconsistent with the previous value.

Example:

Suppose two transactions operate on three accounts.

Account-1	Account-2	Account-3
Balance = 200	Balance = 250	Balance = 150

Transaction-X	Time	Transaction-Y
—	t1	—
Read Balance of Acc-1 sum <-- 200 Read Balance of Acc-2	t2	—
Sum <-- Sum + 250 = 450	t3	—
—	t4	Read Balance of Acc-3
—	t5	Update Balance of Acc-3 150 --> 150 - 50 --> 100
—	t6	Read Balance of Acc-1
—	t7	Update Balance of Acc-1 200 --> 200 + 50 --> 250
Read Balance of Acc-3	t8	COMMIT
Sum <-- Sum + 250 = 550	t9	—

- Transaction-X is doing the sum of all balance while transaction-Y is transferring an amount 50 from Account-1 to Account-3.
- Here, transaction-X produces the result of 550 which is incorrect. If we write this produced result in the database, the

database will become an inconsistent state because the actual sum is 600.

- Here, transaction-X has seen an inconsistent state of the database.

Concurrency Control Protocol

Concurrency control protocols ensure atomicity, isolation, and serializability of concurrent transactions. The concurrency control protocol can be divided into three categories:

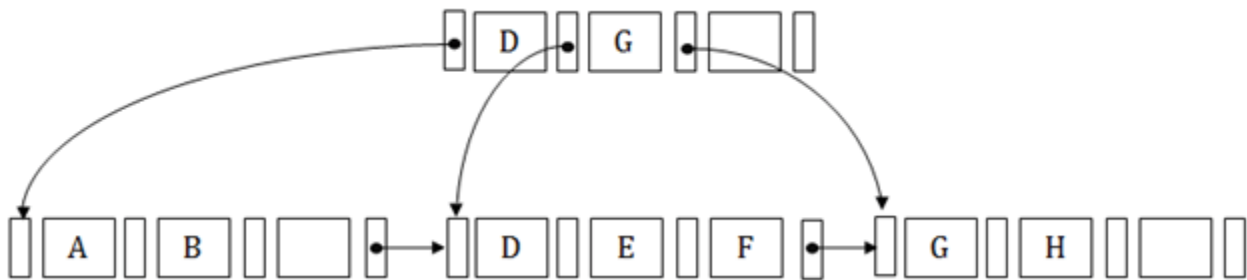
1. Lock based protocol
2. Time-stamp protocol
3. Validation based protocol

B+ Tree

- The B+ tree is a balanced binary search tree. It follows a multi-level index format.
- In the B+ tree, leaf nodes denote actual data pointers. B+ tree ensures that all leaf nodes remain at the same height.
- In the B+ tree, the leaf nodes are linked using a link list. Therefore, a B+ tree can support random access as well as sequential access.

Structure of B+ Tree

- In the B+ tree, every leaf node is at equal distance from the root node. The B+ tree is of the order n where n is fixed for every B+ tree.
- It contains an internal node and leaf node.



Internal node

- An internal node of the B+ tree can contain at least $n/2$ record pointers except the root node.
- At most, an internal node of the tree contains n pointers.

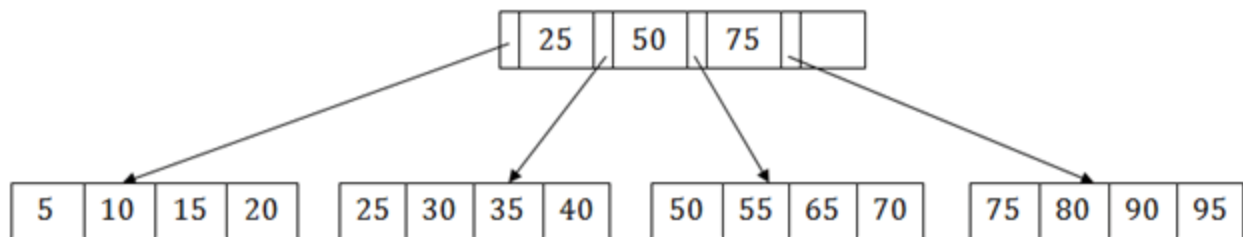
Leaf node

- The leaf node of the B+ tree can contain at least $n/2$ record pointers and $n/2$ key values.
- At most, a leaf node contains n record pointer and n key values.
- Every leaf node of the B+ tree contains one block pointer P to point to next leaf node.

Searching a record in B+ Tree

Suppose we have to search 55 in the below B+ tree structure. First, we will fetch for the intermediary node which will direct to the leaf node that can contain a record for 55.

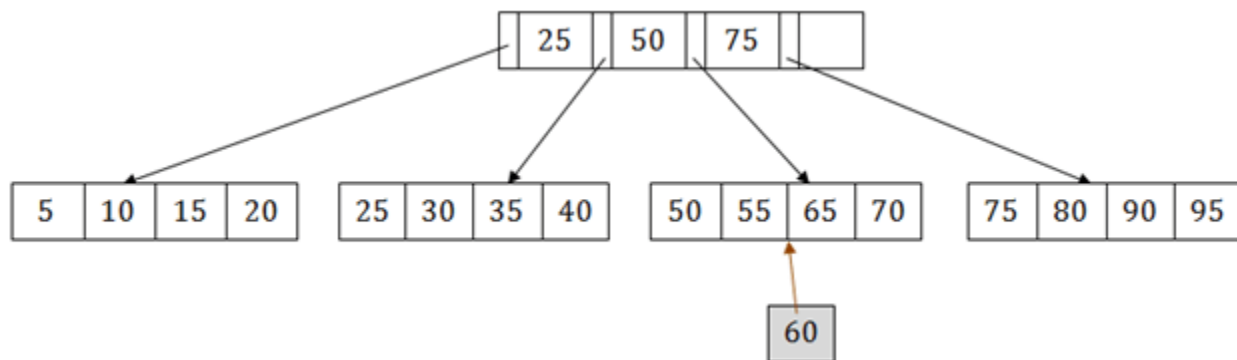
So, in the intermediary node, we will find a branch between 50 and 75 nodes. Then at the end, we will be redirected to the third leaf node. Here DBMS will perform a sequential search to find 55.



B+ Tree Insertion

Suppose we want to insert a record 60 in the below structure. It will go to the 3rd leaf node after 55. It is a balanced tree, and a leaf node of this tree is already full, so we cannot insert 60 there.

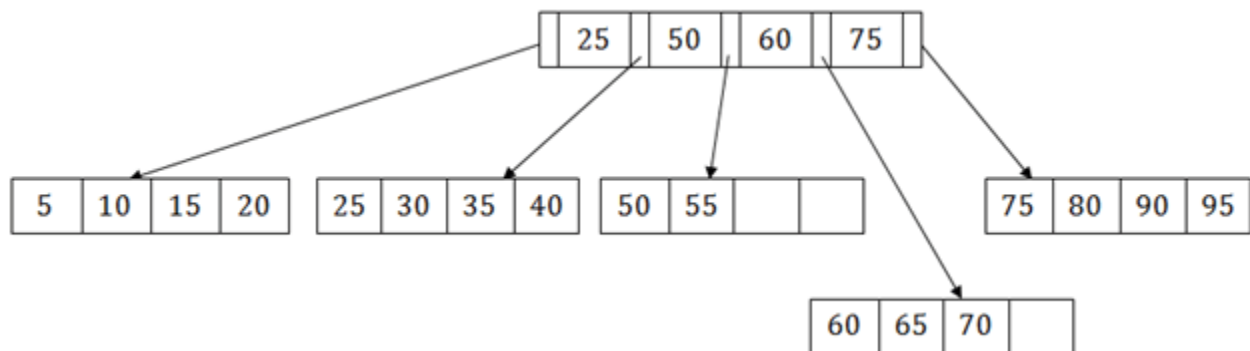
In this case, we have to split the leaf node, so that it can be inserted into tree without affecting the fill factor, balance and order.



The 3rd leaf node has the values (50, 55, 60, 65, 70) and its current root node is 50. We will split the leaf node of the tree in the middle

so that its balance is not altered. So we can group (50, 55) and (60, 65, 70) into 2 leaf nodes.

If these two has to be leaf nodes, the intermediate node cannot branch from 50. It should have 60 added to it, and then we can have pointers to a new leaf node.

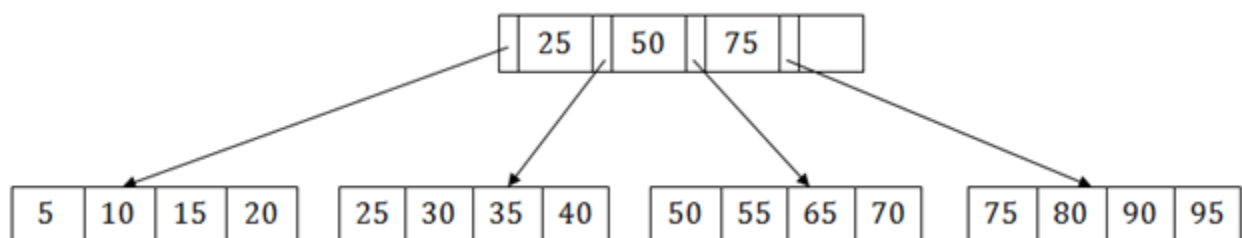


This is how we can insert an entry when there is overflow. In a normal scenario, it is very easy to find the node where it fits and then place it in that leaf node.

B+ Tree Deletion

Suppose we want to delete 60 from the above example. In this case, we have to remove 60 from the intermediate node as well as from the 4th leaf node too. If we remove it from the intermediate node, then the tree will not satisfy the rule of the B+ tree. So we need to modify it to have a balanced tree.

After deleting node 60 from above B+ tree and re-arranging the nodes, it will show as follows:



A **B+ tree** is an N-ary tree with a variable but often large number of children per node. A B+ tree consists of a root, internal nodes and leaves. The root may be either a leaf or a node with two or more children.

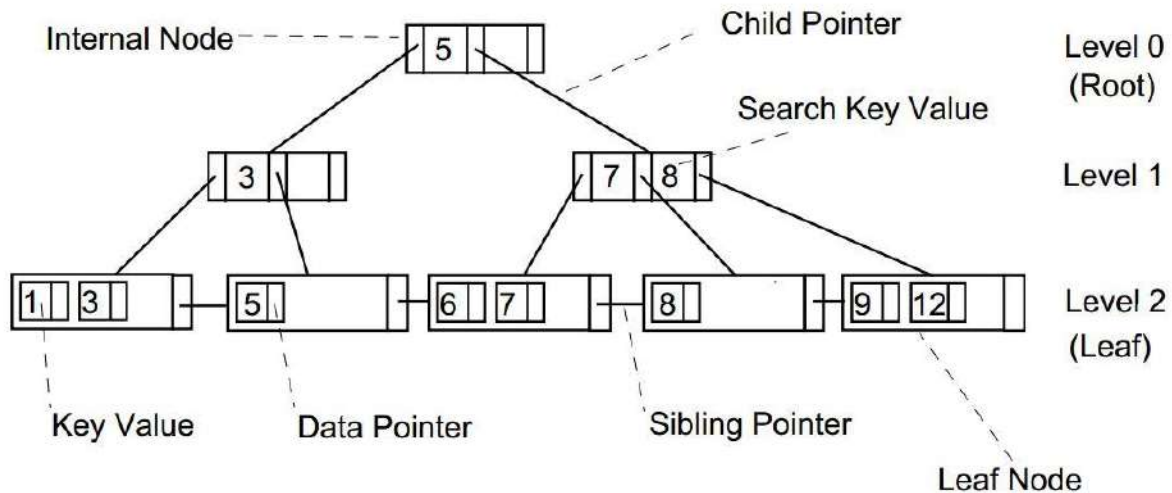
A B+ tree can be viewed as a B-tree in which each node contains only keys (not key-value pairs), and to which an additional level is added at the bottom with linked leaves.

The B+-Tree consists of two types of nodes:

- internal nodes
- leaf nodes

Properties:

- Internal nodes point to other nodes in the tree.
- Leaf nodes point to data in the database using data pointers. Leaf nodes also contain an additional pointer, called the sibling pointer, which is used to improve the efficiency of certain types of search.
- All the nodes in a B+-Tree must be at least half full except the root node which may contain a minimum of two entries. The algorithms that allow data to be inserted into and deleted from a B+-Tree guarantee that each node in the tree will be at least half full.
- Searching for a value in the B+-Tree always starts at the root node and moves downwards until it reaches a leaf node.
- Both internal and leaf nodes contain key values that are used to guide the search for entries in the index.
- The B+ Tree is called a balanced tree because every path from the root node to a leaf node is the same length. A balanced tree means that all searches for individual values require the same number of nodes to be read from the disc.



The figure depicts the basic structure of B+ Tree.

Algorithm

- **Basic operations associated with B+ Tree:**

- ❖ **Searching a node in a B+ Tree**

- Perform a binary search on the records in the current node.
- If a record with the search key is found, then return that record.
- If the current node is a leaf node and the key is not found, then report an unsuccessful search.
- Otherwise, follow the proper branch and repeat the process.

- ❖ **Insertion of node in a B+ Tree:**

- Allocate new leaf and move half the buckets elements to the new bucket.
- Insert the new leaf's smallest key and address into the parent.
- If the parent is full, split it too.
- Add the middle key to the parent node.

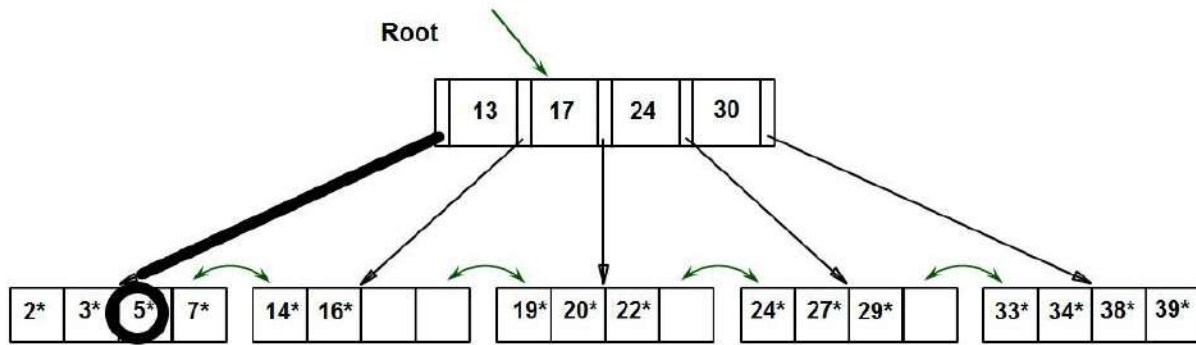
- Repeat until a parent is found that need not split.
- If the root splits, create a new root which has one key and two pointers. (That is, the value that gets pushed to the new root gets removed from the original node)

❖ **Deletion of a node in a B+ Tree:**

- Descend to the leaf where the key exists.
- Remove the required key and associated reference from the node.
- If the node still has enough keys and references to satisfy the invariants, stop.
- If the node has too few keys to satisfy the invariants, but its next oldest or next youngest sibling at the same level has more than necessary, distribute the keys between this node and the neighbor. Repair the keys in the level above to represent that these nodes now have a different “split point” between them; this involves simply changing a key in the levels above, without deletion or insertion.
- If the node has too few keys to satisfy the invariant, and the next oldest or next youngest sibling is at the minimum for the invariant, then merge the node with its sibling; if the node is a non-leaf, we will need to incorporate the “split key” from the parent into our merging.
- In either case, we will need to repeat the removal algorithm on the parent node to remove the “split key” that previously separated these merged nodes – unless the parent is the root and we are removing the final key from the root, in which case the merged node becomes the new root (and the tree has become one level shorter than before).

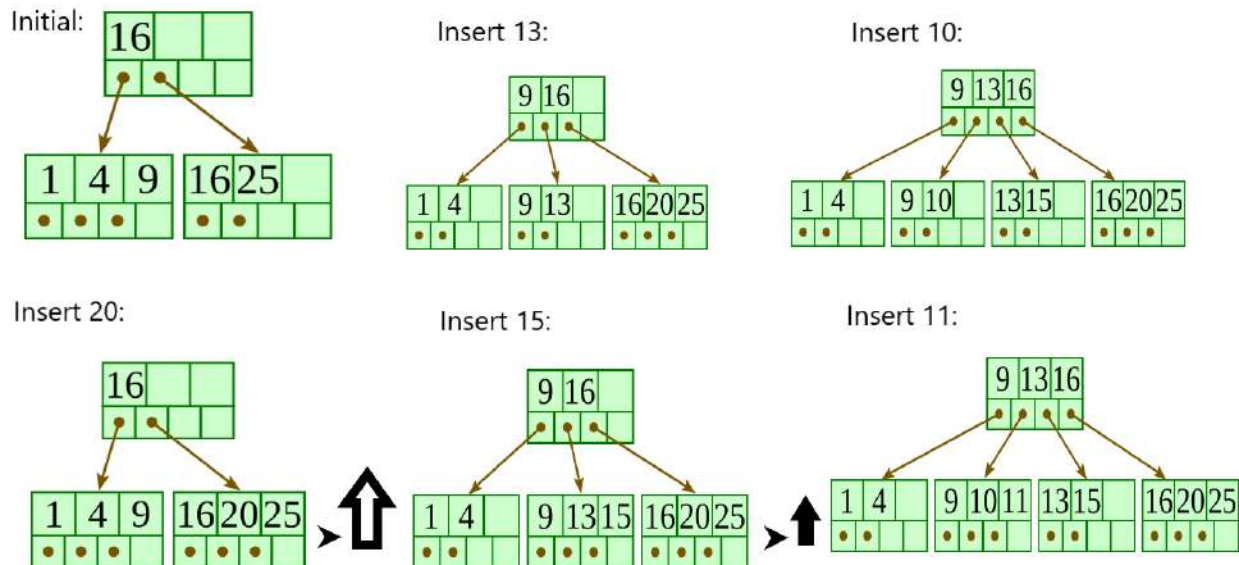
❖ Search begins at root, and key comparisons direct it to a leaf

❖ Search for 5* ➤ Traverse along the left pointer to the element with the value 13 in the root node.



The figure depicts the searching of an element in a B-Tree.

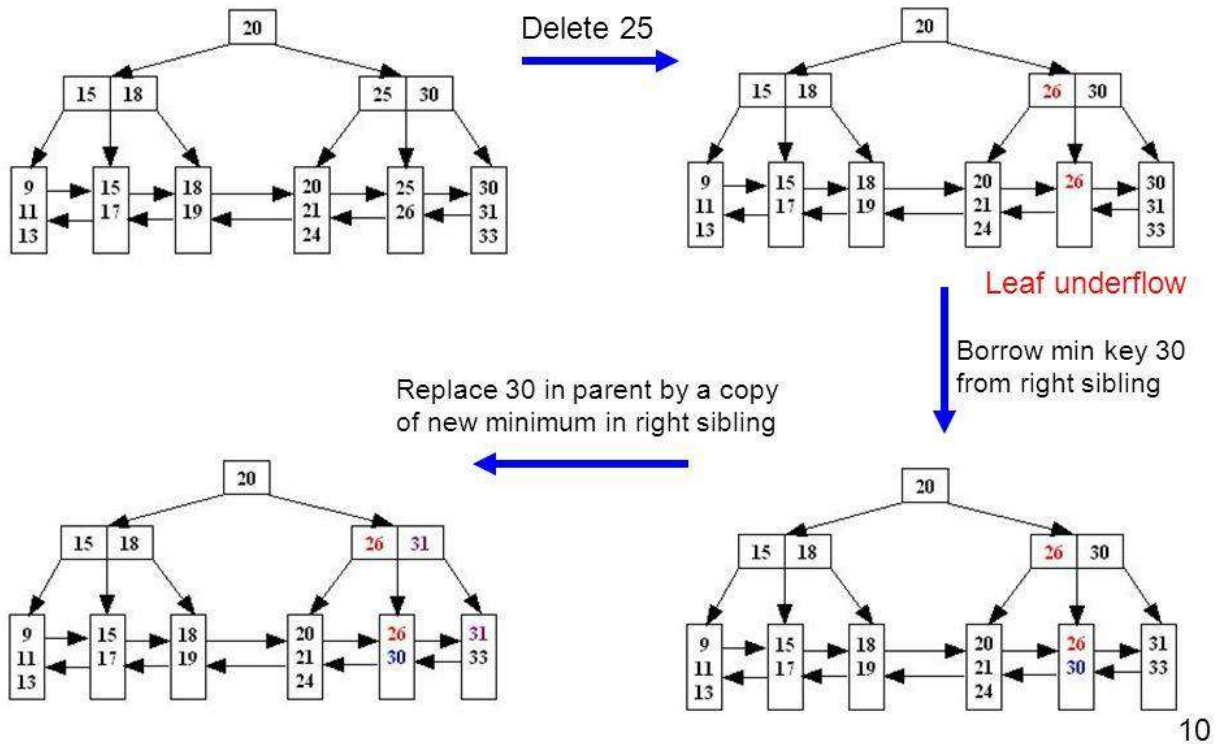
INSERT



The figure illustrates the insertion of an element in a B+ Tree. B+ tree grows at the root and not at the leaves.

Illustration when node has too few keys while sibling has extra keys

Example: Delete 25 from the following B+ tree of order $M = 3$ and $L = 3$



10

The figure illustrates the deletion of an element from a b+ tree.

B - Tree Data structure

In search trees like binary search tree, AVL Tree, Red-Black tree, etc., every node contains only one value (key) and a maximum of two children. But there is a special type of search tree called B-Tree in which a node contains more than one value (key) and more than two children.

B-Tree can be defined as follows...

B-Tree is a self-balanced search tree in which every node contains multiple keys and has more than two children.

Here, the number of keys in a node and number of children for a node depends on the order of B-Tree. Every B-Tree has an order.

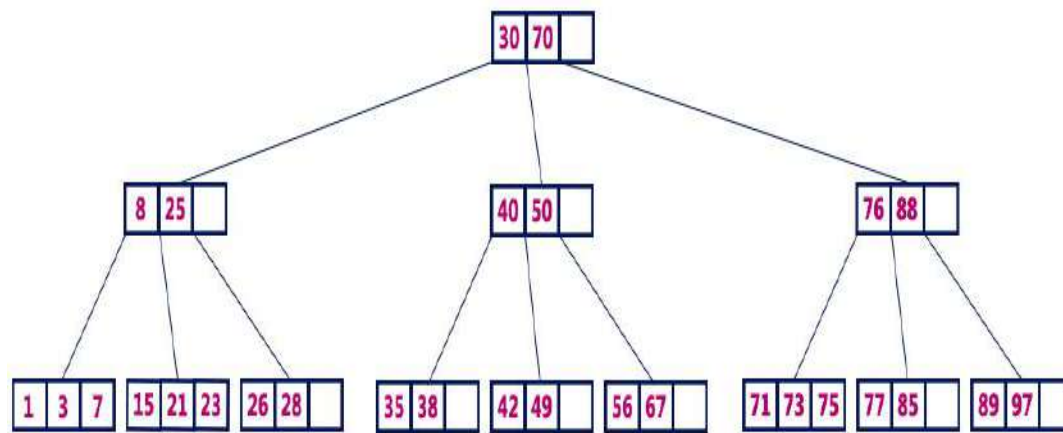
B-Tree of Order m has the following properties...

- **Property #1** - All leaf nodes must be at same level.
- **Property #2** - All nodes except root must have at least $\lceil m/2 \rceil - 1$ keys and maximum of $m-1$ keys.
- **Property #3** - All non leaf nodes except root (i.e. all internal nodes) must have at least $m/2$ children.
- **Property #4** - If the root node is a non leaf node, then it must have at least 2 children.
- **Property #5** - A non leaf node with $n-1$ keys must have n number of children.
- **Property #6** - All the key values in a node must be in Ascending Order.

For example, B-Tree of Order 4 contains a maximum of 3 key values in a node and maximum of 4 children for a node.

Example

B-Tree of Order 4



Operations on a B-Tree

The following operations are performed on a B-Tree...

1. Search
2. Insertion
3. Deletion

Search Operation in B-Tree

The search operation in B-Tree is similar to the search operation in Binary Search Tree. In a Binary search tree, the search process starts from the root node and we make a 2-way decision every time (we go to either left subtree or right subtree). In B-Tree also search process starts from the root node but here we make an n-way decision every time. Where 'n' is the total number of children the node has. In a B-Tree, the search operation is performed with $O(\log n)$ time complexity. The search operation is performed as follows...

- **Step 1** - Read the search element from the user.
- **Step 2** - Compare the search element with first key value of root node in the tree.
- **Step 3** - If both are matched, then display "Given node is found!!!" and terminate the function

- **Step 4** - If both are **not matched**, then check whether search element is smaller or larger than that key value.
- **Step 5** - If search element is smaller, then continue the search process in left subtree.
- **Step 6** - If search element is larger, then compare the search element with next key value in the same node and repeat steps 3, 4, 5 and 6 until we find the exact match or until the search element is compared with last key value in the leaf node.
- **Step 7** - If the last key value in the leaf node is also not matched then display "Element is not found" and terminate the function.

Insertion Operation in B-Tree

In a B-Tree, a new element must be added only at the leaf node. That means, the new key Value is always attached to the leaf node only. The insertion operation is performed as follows...

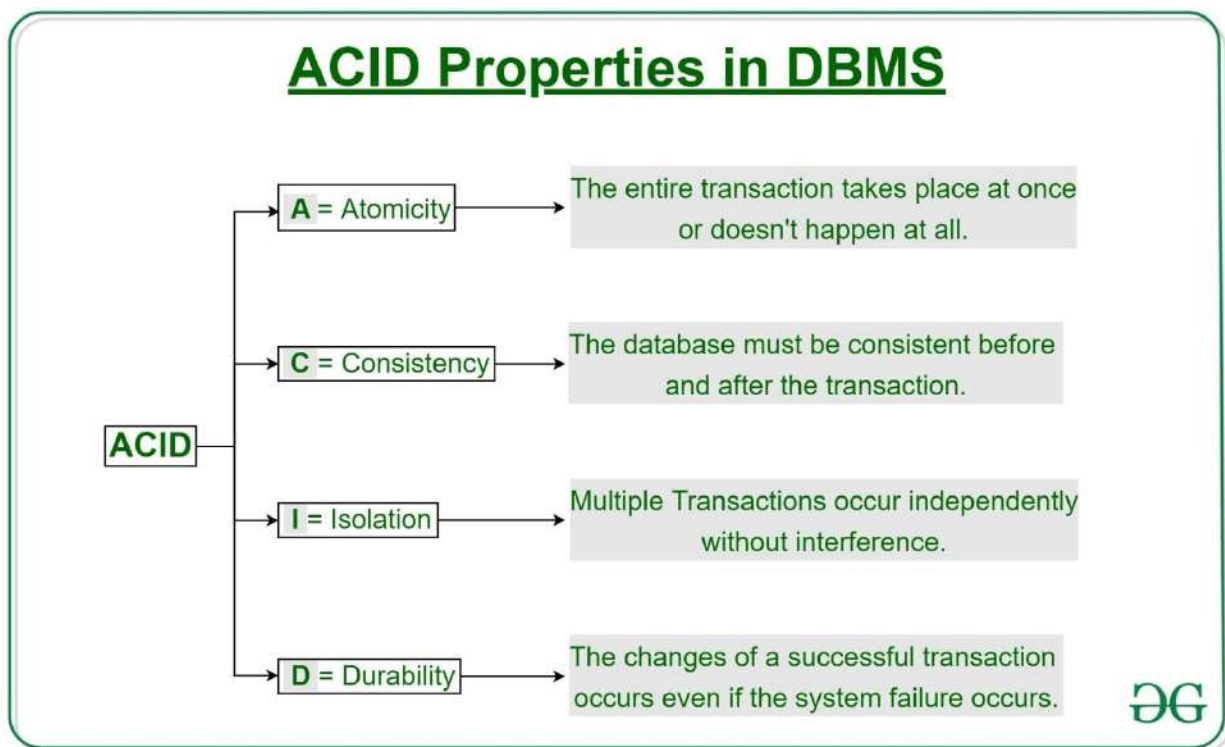
- **Step 1** - Check whether tree is Empty.
- **Step 2** - If tree is Empty, then create a new node with new key value and insert it into the tree as a root node.
- **Step 3** - If tree is Not Empty, then find the suitable leaf node to which the new key value is added using Binary Search Tree logic.
- **Step 4** - If that leaf node has empty position, add the new key value to that leaf node in ascending order of key value within the node.
- **Step 5** - If that leaf node is already full, split that leaf node by sending middle value to its parent node. Repeat the same until the sending value is fixed into a node.
- **Step 6** - If the splitting is performed at root node then the middle value becomes new root node for the tree and the height of the tree is increased by one.

Example

Construct a **B-Tree of Order 3** by inserting numbers from 1 to 10.

A **transaction** is a single logical unit of work which accesses and possibly modifies the contents of a database. Transactions access data using read and write operations.

In order to maintain consistency in a database, before and after the transaction, certain properties are followed. These are called **ACID** properties.



Atomicity

By this, we mean that either the entire transaction takes place at once or doesn't happen at all. There is no midway i.e. transactions do not occur partially. Each transaction is considered as one unit and either runs to completion or is not executed at all. It involves the following two operations.

–**Abort**: (Bring to a **premature end** because of a problem or fault): If a transaction aborts, changes made to database are not visible.

–**Commit:** If a transaction commits, changes made are visible. Atomicity is also known as the ‘All or nothing rule’. Consider the following transaction T consisting of T1 and T2: Transfer of 100 from account X to account Y.

Before: X : 500	Y: 200
Transaction T	
T1	T2
Read (X) X: = X – 100 Write (X)	Read (Y) Y: = Y + 100 Write (Y)
After: X : 400	Y : 300

If the transaction fails after completion of T1 but before completion of T2.(say, after **write(X)** but before **write(Y)**), then amount has been deducted from X but not added to Y. This results in an inconsistent database state. Therefore, the transaction must be executed in entirety in order to ensure correctness of database state.

Consistency

This means that integrity constraints must be maintained so that the database is consistent before and after the transaction. It refers to the correctness of a database. Referring to the example above, The total amount before and after the transaction must be maintained.

Total **before** T occurs = **500 + 200 = 700.**

Total **after** T occurs = **400 + 300 = 700.**

Therefore, database is **consistent**. Inconsistency occurs in case T1 completes but T2 fails. As a result T is incomplete.

Isolation

This property ensures that multiple transactions can occur concurrently

without leading to the inconsistency of database state. Transactions occur independently without interference. Changes occurring in a particular transaction will not be visible to any other transaction until that particular change in that transaction is written to memory or has been committed. This property ensures that the execution of transactions concurrently will result in a state that is equivalent to a state achieved these were executed serially in some order.

Let $X = 500$, $Y = 500$.

Consider two transactions T and T'.

T	T'
Read (X)	Read (X)
$X := X * 100$	Read (Y)
Write (X)	$Z := X + Y$
Read (Y)	Write (Z)
$Y := Y - 50$	
Write	

Suppose T has been executed till Read (Y) and then T' starts. As a result, interleaving of operations takes place due to which T' reads correct value of X but incorrect value of Y and sum computed by

T': ($X + Y = 50, 000 + 500 = 50, 500$)

is thus not consistent with the sum at end of transaction:

T: ($X + Y = 50, 000 + 450 = 50, 450$).

This results in database inconsistency, due to a loss of 50 units. Hence, transactions must take place in isolation and changes should be visible only after they have been made to the main memory.

Durability:

This property ensures that once the transaction has completed execution, the updates and modifications to the database are stored in and written to disk and they persist even if a system failure occurs. These updates now become permanent and are stored in non-volatile memory. The effects of the transaction, thus, are never lost.

The **ACID** properties, in totality, provide a mechanism to ensure correctness and consistency of a database in a way such that each transaction is a group of operations that acts a single unit, produces consistent results, acts in isolation from other operations and updates that it makes are durably stored.