

# RELATIONAL DATABASE MANAGEMENT SYSTEM

## UNIT I

### INTRODUCTION:

Database management system is software that is used to manage the database. The database is a collection of inter-related data which is used to retrieve, insert and delete the data efficiently. It is also used to organize the data in the form of a table, schema, views, and reports, etc.

**For example:** The college Database organizes the data about the admin, staff, students and faculty etc.

Using the database, you can easily retrieve, insert, and delete the information.

### Database Management System

- Database management system is software which is used to manage the database. For example: [MySQL](#), [Oracle](#), etc are a very popular commercial database which is used in different applications.
- DBMS provides an interface to perform various operations like database creation, storing data in it, updating data, creating a table in the database and a lot more.
- It provides protection and security to the database. In the case of multiple users, it also maintains data consistency.

### DBMS allows users the following tasks:

- **Data Definition:** It is used for creation, modification, and removal of definition that defines the organization of data in the database.
- **Data Updation:** It is used for the insertion, modification, and deletion of the actual data in the database.
- **Data Retrieval:** It is used to retrieve the data from the database which can be used by applications for various purposes.
- **User Administration:** It is used for registering and monitoring users, maintain data integrity, enforcing data security, dealing with concurrency control, monitoring performance and recovering information corrupted by unexpected failure.

### Characteristics of DBMS

- It uses a digital repository established on a server to store and manage the information.
- It can provide a clear and logical view of the process that manipulates data.
- DBMS contains automatic backup and recovery procedures.
- It contains ACID properties which maintain data in a healthy state in case of failure.
- It can reduce the complex relationship between data.
- It is used to support manipulation and processing of data.
- It is used to provide security of data.
- It can view the database from different viewpoints according to the requirements of the user.

## Advantages of DBMS

- **Controls database redundancy:** It can control data redundancy because it stores all the data in one single database file and that recorded data is placed in the database.
- **Data sharing:** In DBMS, the authorized users of an organization can share the data among multiple users.
- **Easily Maintenance:** It can be easily maintainable due to the centralized nature of the database system.
- **Reduce time:** It reduces development time and maintenance need.
- **Backup:** It provides backup and recovery subsystems which create automatic backup of data from [hardware](#) and [software](#) failures and restores the data if required.
- **multiple user interface:** It provides different types of user interfaces like graphical user interfaces, application program interfaces

## Disadvantages of DBMS

- **Cost of Hardware and Software:** It requires a high speed of data processor and large memory size to run DBMS software.
- **Size:** It occupies a large space of disks and large memory to run them efficiently.
- **Complexity:** Database system creates additional complexity and requirements.
- **Higher impact of failure:** Failure is highly impacted the database because in most of the organization, all the data stored in a single database and if the database is damaged due to electric failure or database corruption then the data may be lost forever.

## DATA MODELS:

- The data which is stored in the database at a particular moment of time is called an instance of the database.
- The overall design of a database is called schema.
- A database schema is the skeleton structure of the database. It represents the logical view of the entire database.
- A schema contains schema objects like table, foreign key, primary key, views, columns, data types, stored procedure, etc.
- A database schema can be represented by using the visual diagram. That diagram shows the database objects and relationship with each other.
- A database schema is designed by the database designers to help programmers whose software will interact with the database. The process of database creation is called data modelling.

A schema diagram can display only some aspects of a schema like the name of record type, data type, and constraints. Other aspects can't be specified through the schema diagram. For example, the given figure neither shows the data type of each data item nor the relationship among various files.

In the database, actual data changes quite frequently. For example, in the given figure, the database changes whenever we add a new grade or add a student. The data at a particular moment of time is called the instance of the database.

### STUDENT

Name	Student_number	Class	Major
------	----------------	-------	-------

### COURSE

Course_name	Course_number	Credit_hours	Department
-------------	---------------	--------------	------------

### PREREQUISITE

Course_number	Prerequisite_number
---------------	---------------------

### SECTION

Section_identifier	Course_number	Semester	Year	Instructor
--------------------	---------------	----------	------	------------

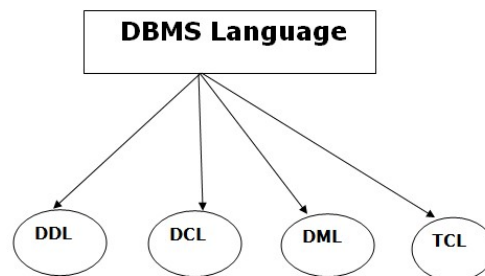
### GRADE\_REPORT

Student_number	Section_identifier	Grade
----------------	--------------------	-------

## DATA LANGUAGES:

- A DBMS has appropriate languages and interfaces to express database queries and updates.
- Database languages can be used to read, store and update the data in the database.

### Types of Database Languages



### 1. Data Definition Language (DDL)

- **DDL** stands for **Data Definition Language**. It is used to define database structure or pattern.
- It is used to create schema, tables, indexes, constraints, etc. in the database.
- Using the DDL statements, you can create the skeleton of the database.
- Data definition language is used to store the information of metadata like the number of tables and schemas, their names, indexes, columns in each table, constraints, etc.

Here are some tasks that come under DDL:

- **Create:** It is used to create objects in the database.
- **Alter:** It is used to alter the structure of the database.
- **Drop:** It is used to delete objects from the database.
- **Truncate:** It is used to remove all records from a table.

- **Rename:** It is used to rename an object.
- **Comment:** It is used to comment on the data dictionary.

These commands are used to update the database schema that's why they come under Data definition language.

## 2. Data Manipulation Language (DML)

**DML** stands for **Data Manipulation Language**. It is used for accessing and manipulating data in a database. It handles user requests.

Here are some tasks that come under DML:

**Select:** It is used to retrieve data from a database.

- **Insert:** It is used to insert data into a table.
- **Update:** It is used to update existing data within a table.
- **Delete:** It is used to delete all records from a table.
- **Merge:** It performs UPSERT operation, i.e., insert or update operations.
- **Call:** It is used to call a structured query language or a Java subprogram.
- **Explain Plan:** It has the parameter of explaining data.
- **Lock Table:** It controls concurrency.

## 3. Data Control Language (DCL)

- **DCL** stands for **Data Control Language**. It is used to retrieve the stored or saved data.
- The DCL execution is transactional. It also has rollback parameters.

(But in Oracle database, the execution of data control language does not have the feature of rolling back.)

Here are some tasks that come under DCL:

- **Grant:** It is used to give user access privileges to a database.
- **Revoke:** It is used to take back permissions from the user.

There are the following operations which have the authorization of Revoke:

CONNECT, INSERT, USAGE, EXECUTE, DELETE, UPDATE and SELECT.

## 4. Transaction Control Language (TCL)

TCL is used to run the changes made by the DML statement. TCL can be grouped into a logical transaction.

Here are some tasks that come under TCL:

- **Commit:** It is used to save the transaction on the database.
- **Rollback:** It is used to restore the database to original since the last Commit

## Transaction Management

Transactions are a set of operations used to perform a logical set of work. It is the bundle of all the instructions of a logical operation. A transaction usually means that the data in the database has changed. One of the major uses of DBMS is to protect the user's data from system failures. It is done by ensuring that all the

data is restored to a consistent state when the computer is restarted after a crash. The transaction is any one execution of the user program in a DBMS. One of the important properties of the transaction is that it contains a finite number of steps. Executing the same program multiple times will generate multiple transactions.

**Example:** Consider the following example of transaction operations to be performed to withdraw cash from an ATM vestibule.

#### Steps for ATM Transaction

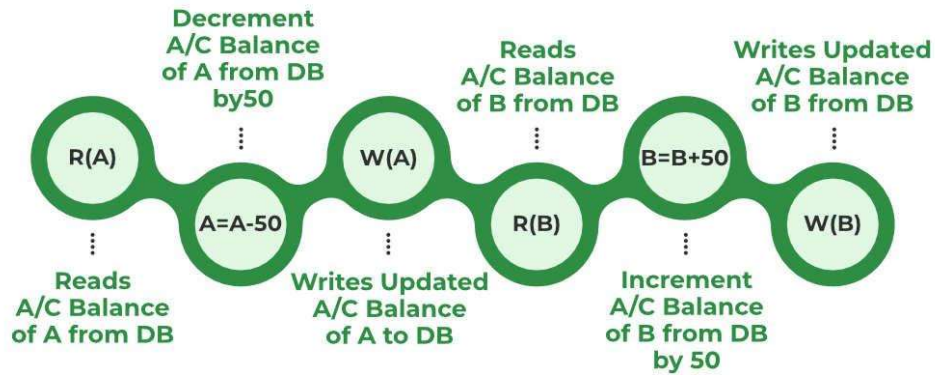
1. Transaction Start.
2. Insert your ATM card.
3. Select a language for your transaction.
4. Select the Savings Account option.
5. Enter the amount you want to withdraw.
6. Enter your secret pin.
7. Wait for some time for processing.
8. Collect your Cash.
9. Transaction Completed.

**A transaction can include the following basic database access operation.**

- **Read/Access data (R):** Accessing the database item from disk (where the database stored data) to memory variable.
- **Write/Change data (W):** Write the data item from the memory variable to the disk.
- **Commit:** Commit is a transaction control language that is used to permanently save the changes done in a transaction

**Example:** Transfer of 50₹ from Account A to Account B. Initially A= 500₹, B= 800₹. This data is brought to RAM from Hard Disk.

```
R(A) -- 500    // Accessed from RAM.
A = A-50       // Deducting 50₹ from A.
W(A)--450      // Updated in RAM.
R(B) -- 800    // Accessed from RAM.
B=B+50        // 50₹ is added to B's Account.
W(B) --850     // Updated in RAM.
commit        // The data in RAM is taken back to Hard Disk.
```



### *Stages of Transaction*

**Note:** The updated value of Account A = 450₹ and Account B = 850₹.

All instructions before committing come under a partially committed state and are stored in RAM. When the commit is read the data is fully accepted and is stored on Hard Disk.

If the transaction is failed anywhere before committing we have to go back and start from the beginning. We can't continue from the same state. This is known as Roll Back.

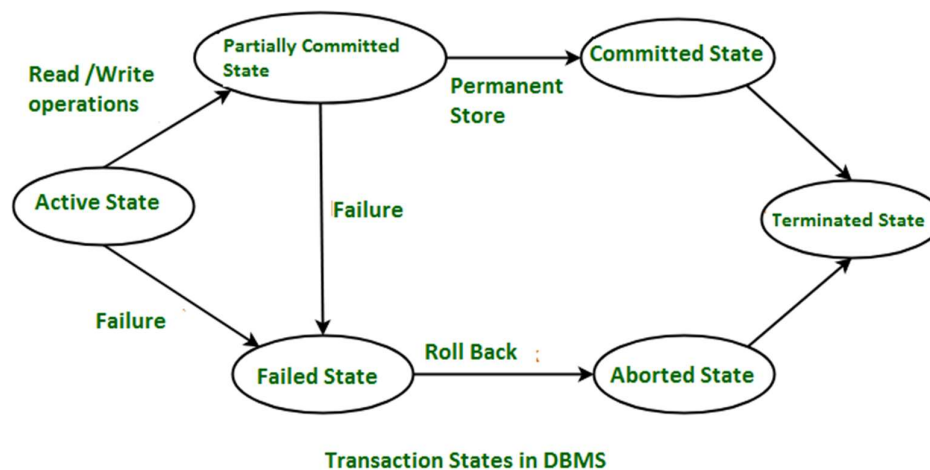
### **Desirable Properties of Transaction (ACID Properties)**

For a transaction to be performed in DBMS, it must possess several properties often called ACID properties.

- **A** – Atomicity
- **C** – Consistency
- **I** – Isolation
- **D** – Durability

### **Transaction States**

Transactions can be implemented using SQL queries and Servers. In the below-given diagram, you can see how transaction states work.



### *Transaction States*

The transaction has the four properties. These are used to maintain consistency in a database, before and after the transaction.

Property of Transaction:

1. Atomicity
2. Consistency
3. Isolation
4. Durability

#### **Atomicity:**

- It states that all operations of the transaction take place at once if not, the transactions are aborted.
- There is no midway, i.e., the transaction cannot occur partially. Each transaction is treated as one unit and either run to completion or is not executed at all.
- Atomicity involves the following two operations:
- **Abort:** If a transaction aborts, then all the changes made are not visible.
- **Commit:** If a transaction commits then all the changes made are visible.

#### **Consistency:**

- The integrity constraints are maintained so that the database is consistent before and after the transaction.
- The execution of a transaction will leave a database in either its prior stable state or a new stable state.
- The consistent property of database states that every transaction sees a consistent database instance.
- The transaction is used to transform the database from one consistent state to another consistent state.

#### **Isolation:**

- It shows that the data which is used at the time of execution of a transaction cannot be used by the second transaction until the first one is completed.
- In isolation, if the transaction T1 is being executed and using the data item X, then that data item can't be accessed by any other transaction T2 until the transaction T1 ends.
- The concurrency control subsystem of the DBMS enforces the isolation property.

#### **Durability:**

- The durability property is used to indicate the performance of the database's consistent state. It states that the transaction made the permanent changes.
- They cannot be lost by the erroneous operation of a faulty transaction or by the system failure. When a transaction is completed, then the database reaches a state known as the consistent state. That consistent state cannot be lost, even in the event of a system's failure.
- The recovery subsystem of the DBMS has the responsibility of Durability property.

#### **IMPLEMENTATION OF ATOMICITY AND DURABILITY:**

The recovery-management component of a database system can support atomicity and durability by a variety of schemes.

E.g. the shadow-database scheme:

### Shadow copy:

- In the shadow-copy scheme, a transaction that wants to update the database first creates a complete copy of the database.
- All updates are done on the new database copy, leaving the original copy, the shadow copy, untouched. If at any point the transaction has to be aborted, the system merely deletes the new copy. The old copy of the database has not been affected.
- This scheme is based on making copies of the database, called shadow copies, assumes that only one transaction is active at a time.
- The scheme also assumes that the database is simply a file on disk. A pointer called db pointer is maintained on disk; it points to the current copy of the database.

### TRANSACTION ISOLATION LEVELS IN DBMS:

Some other transaction may also have used value produced by the failed transaction. So we also have to rollback those transactions.

The SQL standard defines four isolation levels :

- 1. Read Uncommitted** – Read Uncommitted is the lowest isolation level. In this level, one transaction may read not yet committed changes made by other transaction, thereby allowing dirty reads. In this level, transactions are not isolated from each other.
- 2. Read Committed** – This isolation level guarantees that any data read is committed at the moment it is read. Thus it does not allow dirty read. The transaction holds a read or write lock on the current row, and thus prevent other transactions from reading, updating or deleting it.
- 3. Repeatable Read** – This is the most restrictive isolation level. The transaction holds read locks on all rows it references and writes locks on all rows it inserts, updates, deletes. Since other transaction cannot read, update or delete these rows, consequently it avoids non-repeatable read.
- 4. Serializable** – This is the Highest isolation level. A serializable execution is guaranteed to be serializable. Serializable execution is defined to be an execution of operations in which concurrently executing transactions appears to be serially executing.

### 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 transactions 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.

### Serializability:

It is an important aspect of Transactions. In simple meaning, you can say that serializability is a way to check whether two transactions working on a database are maintaining database consistency or not.

It is of two types:

1. Conflict Serializability
2. View Serializability

### Schedule

Schedule, as the name suggests is a process of lining the transactions and executing them one by one. When there are multiple transactions that are running in a concurrent manner and the order of operation is needed to be set so that the operations do not overlap each other, Scheduling is brought into play and the transactions are timed accordingly.

It is of two types:

1. Serial Schedule
2. Non-Serial Schedule

### Uses of Transaction Management

- The DBMS is used to schedule the access of data concurrently. It means that the user can access multiple data from the database without being interfered with by each other. Transactions are used to manage concurrency.
- It is also used to satisfy ACID properties.
- It is used to solve Read/Write Conflicts.
- It is used to implement Recoverability, Serializability, and Cascading.
- Transaction Management is also used for Concurrency Control Protocols and the Locking of data.

### Disadvantages of using a Transaction

- It may be difficult to change the information within the transaction database by end-users.
- We need to always roll back and start from the beginning rather than continue from the previous state

### Storage Management:

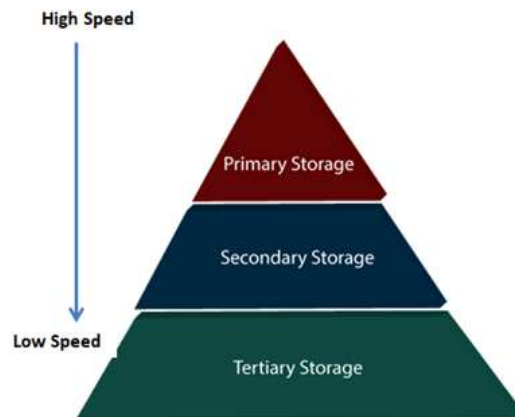
A database system provides an ultimate view of the stored data. However, data in the form of bits, bytes get stored in different storage devices.

In this section, we will take an overview of various types of storage devices that are used for accessing and storing data.

## Types of Data Storage

For storing the data, there are different types of storage options available. These storage types differ from one another as per the speed and accessibility. There are the following types of storage devices used for storing the data:

- Primary Storage
- Secondary Storage
- Tertiary Storage



### Primary Storage

It is the primary area that offers quick access to the stored data. We also know the primary storage as volatile storage. It is because this type of memory does not permanently store the data. As soon as the system leads to a power cut or a crash, the data also get lost. Main memory and cache are the types of primary storage.

**Main Memory:** It is the one that is responsible for operating the data that is available by the storage medium. The main memory handles each instruction of a computer machine. This type of memory can store gigabytes of data on a system but is small enough to carry the entire database. At last, the main memory loses the whole content if the system shuts down because of power failure or other reasons.

1. **Cache:** It is one of the costly storage media. On the other hand, it is the fastest one. A cache is a tiny storage media which is maintained by the computer hardware usually. While designing the algorithms and query processors for the data structures, the designers keep concern on the cache effects.

### Secondary Storage

Secondary storage is also called as Online storage. It is the storage area that allows the user to save and store data permanently. This type of memory does not lose the data due to any power failure or system crash. That's why we also call it non-volatile storage.

There are some commonly described secondary storage media which are available in almost every type of computer system:

- **Flash Memory:** A flash memory stores data in USB (Universal Serial Bus) keys which are further plugged into the USB slots of a computer system. These USB keys help transfer data to a computer system, but it varies in size limits. Unlike the main memory, it is possible to get back the stored data which may be lost due to a power cut or other reasons. This type of memory storage is most commonly used in the server systems for caching the frequently used data. This leads the systems towards high performance and is capable of storing large amounts of databases than the main memory.

- **Magnetic Disk Storage:** This type of storage media is also known as online storage media. A magnetic disk is used for storing the data for a long time. It is capable of storing an entire database. It is the responsibility of the computer system to make availability of the data from a disk to the main memory for further accessing. Also, if the system performs any operation over the data, the modified data should be written back to the disk. The tremendous capability of a magnetic disk is that it does not affect the data due to a system crash or failure, but a disk failure can easily ruin as well as destroy the stored data.

### Tertiary Storage

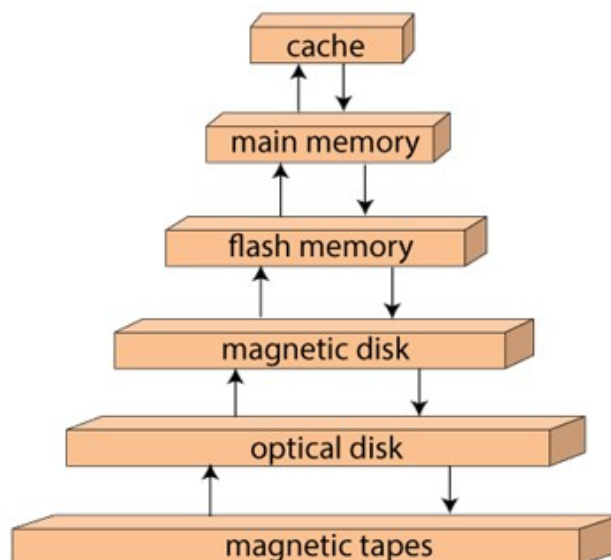
It is the storage type that is external from the computer system. It has the slowest speed. But it is capable of storing a large amount of data. It is also known as Offline storage. Tertiary storage is generally used for data backup. There are following tertiary storage devices available:

- **Optical Storage:** An optical storage can store megabytes or gigabytes of data. A Compact Disk (CD) can store 700 megabytes of data with a playtime of around 80 minutes. On the other hand, a Digital Video Disk or a DVD can store 4.7 or 8.5 gigabytes of data on each side of the disk.
- **Tape Storage:** It is the cheapest storage medium than disks. Generally, tapes are used for archiving or backing up the data. It provides slow access to data as it accesses data sequentially from the start. Thus, tape storage is also known as sequential-access storage. Disk storage is known as direct-access storage as we can directly access the data from any location on disk.

### Storage Hierarchy

Besides the above, various other storage devices reside in the computer system. These storage media are organized on the basis of data accessing speed, cost per unit of data to buy the medium, and by medium's reliability. Thus, we can create a hierarchy of storage media on the basis of its cost and speed.

Thus, on arranging the above-described storage media in a hierarchy according to its speed and cost, we conclude the below-described image:



Storage device hierarchy

In the image, the higher levels are expensive but fast. On moving down, the cost per bit is decreasing, and the access time is increasing. Also, the storage media from the main memory to up represents the volatile nature, and below the main memory, all are non-volatile devices.

### DATABASE ADMINISTRATOR:

A database administrator (DBA) is a person or group in charge of implementing DBMS in an organization. The DBA job requires a high degree of technical expertise. DBA consists of a team of people rather than just one person.

The primary role of Database administrator is as follows –

- Database design
- Performance issues
- Database accessibility
- Capacity issues
- Data replication
- Table Maintenance

#### Responsibilities of DBA

The responsibilities of DBA are as follows –

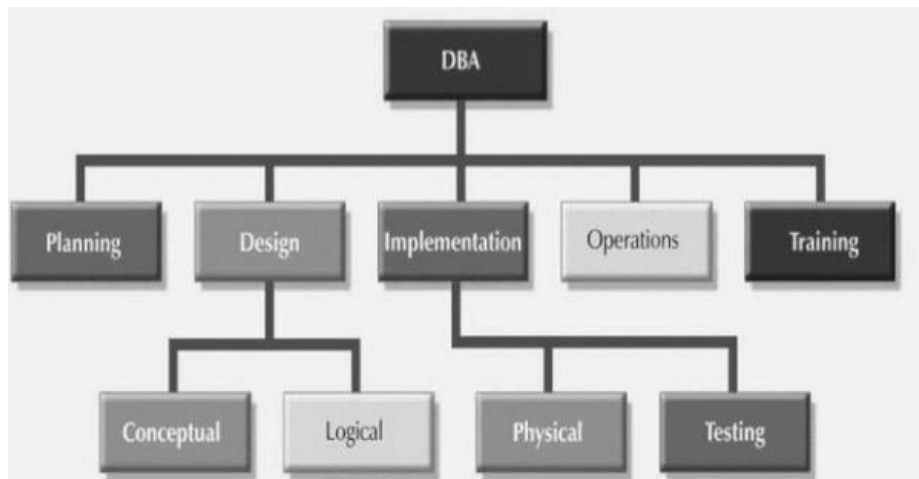
- Makes the decision concerning the content of the database.
- Plans the storage structure and access strategy.
- Provides the support to the users.
- Defines the security and integrity checks.
- Interpreter backup and recovery strategies.
- Monitoring the performance and responding to the changes in the requirements.

#### Skills required for DBA

The skills required to be a successful DBA are as follows –

- Database designing.
- Knowledge of Structured Query Language (SQL).
- Know about distributed architecture.
- Knowledge on different operating servers.
- Idea on Relational Database Management System (RDBMS).
- Ready to face challenges and solve the problems quickly.

The role of DBA is as shown below –



### Types of Database Administrator (DBA) :

- **Administrative DBA –**  
Their job is to maintain the server and keep it functional. They are concerned with data backups, security, troubleshooting, replication, migration, etc.
- **Data Warehouse DBA –**  
Assigned earlier roles, but held accountable for merging data from various sources into the data warehouse. They also design the warehouse, with cleaning and scrubs data prior to loading.
- **Cloud DBA –**  
Nowadays companies are preferring to save their workpiece on cloud storage. As it reduces the chance of data loss and provides an extra layer of data security and integrity.
- **Development DBA –**  
They build and develop queries, stores procedure, etc. that meets firm or organization needs. They are par at programming.
- **Application DBA –**  
They particularly manage all requirements of application components that interact with the database and accomplish activities such as application installation and coordination, application upgrades, database cloning, data load process management, etc.
- **Architect –**  
They are held responsible for designing schemas like building tables. They work to build a structure that meets organizational needs. The design is further used by developers and development DBAs to design and implement real applications.
- **OLAP DBA –**  
They design and build multi-dimensional cubes for determination support or OLAP systems.
- **Data Modeler –**  
In general, a data modeler is in charge of a portion of a data architect's duties. A data modeler is typically not regarded as a DBA, but this is not a hard and fast rule.
- **Task-Oriented DBA –**  
To concentrate on a specific DBA task, large businesses may hire highly specialised DBAs. They are quite uncommon outside of big corporations. Recovery and backup DBA, whose responsibility it is to guarantee that the databases of businesses can be recovered, is an example of a task-oriented DBA.

However, this specialism is not present in the majority of firms. These task-oriented DBAs will make sure that highly qualified professionals are working on crucial DBA tasks when it is possible.

- **Database Analyst –**

This position doesn't actually have a set definition. Junior DBAs may occasionally be referred to as database analysts. A database analyst occasionally performs functions that are comparable to those of a database architect. The term "Data Administrator" is also used to describe database analysts and data analysts. Additionally, some businesses occasionally refer to database administrators as data analysts.

## **DATABASE ADMINISTRATORS:**

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

1. **Database Administrator (DBA) :** Database Administrator (DBA) is a person/team who defines the schema and also controls the 3 levels of database. The DBA will then create a new account id and password for the user if he/she need to access the database. DBA is also responsible for providing security to the database and he allows only the authorized users to access/modify the data base. DBA is responsible for the problems such as security breaches and poor system response time.
  - DBA also monitors the recovery and backup and provide technical support.
  - The DBA has a DBA account in the DBMS which called a system or superuser account.
  - DBA repairs damage caused due to hardware and/or software failures.
  - DBA is the one having privileges to perform DCL (Data Control Language) operations such as GRANT and REVOKE, to allow/restrict a particular user from accessing the database.
2. **Naive / Parametric End Users :** Parametric End Users are the unsophisticated who don't have any DBMS knowledge but they frequently use the database applications in their daily life to get the desired results. For examples, Railway's ticket booking users are naive users. Clerks in any bank is a naive user because they don't have any DBMS knowledge but they still use the database and perform their given task.
3. **System Analyst :**

System Analyst is a user who analyzes the requirements of parametric end users. They check whether all the requirements of end users are satisfied.
4. **Sophisticated Users :** Sophisticated users can be engineers, scientists, business analyst, who are familiar with the database. They can develop their own database applications according to their requirement. They don't write the program code but they interact the database by writing SQL queries directly through the query processor.
5. **Database Designers :** Data Base Designers are the users who design the structure of database which includes tables, indexes, views, triggers, stored procedures and constraints which are usually enforced before the database is created or populated with data. He/she controls what data must be stored and how the data items to be related. It is responsibility of Database Designers to understand the requirements of different user groups and then create a design which satisfies the need of all the user groups.
6. **Application Programmers :** Application Programmers also referred as System Analysts or simply Software Engineers, are the back-end programmers who writes the code for the application programs. They are the computer professionals. These programs could be written in Programming languages such as Visual Basic, Developer, C, FORTRAN, COBOL etc. Application programmers design, debug, test,

and maintain set of programs called “canned transactions” for the Naive (parametric) users in order to interact with database.

7. **Casual Users / Temporary Users** : Casual Users are the users who occasionally use/access the database but each time when they access the database they require the new information, for example, Middle or higher level manager.
8. **Specialized users** : Specialized users are sophisticated users who write specialized database application that does not fit into the traditional data-processing framework. Among these applications are computer aided-design systems, knowledge-base and expert systems etc.

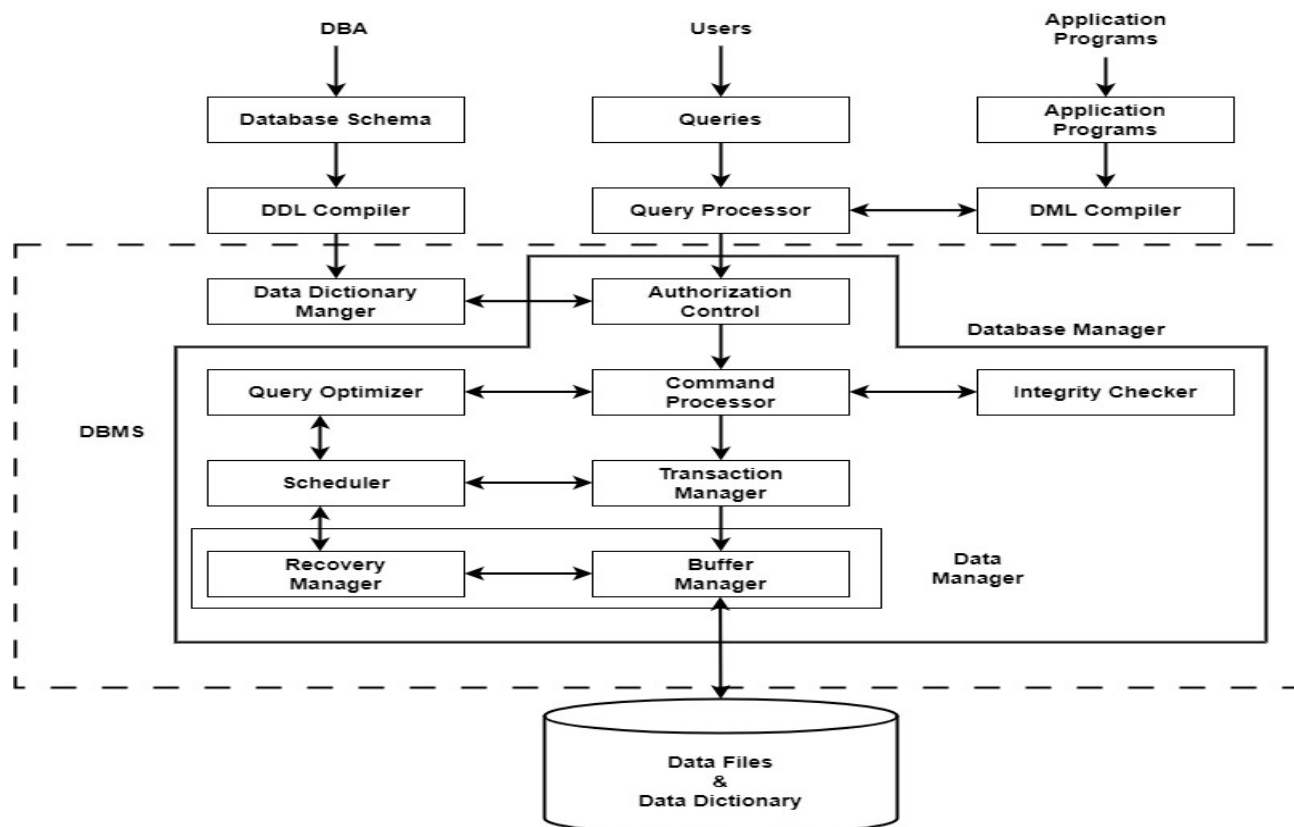
## Structure of DBMS

DBMS means Database Management System, which is a tool or software used to create the database or delete or manipulate the database. A software programme created to store, retrieve, query, and manage data is known as a Database Management System (DBMS). Data can be generated, read, updated, and destroyed by authorized entities thanks to user interfaces (UIs).

Because they give programmers, Database Managers, and end users a consolidated view of the data, Database Management Systems are crucial because they relieve applications and end users of the need to comprehend the physical location of the data. Application Programme Interfaces (APIs) manage internet requests and responses for particular sorts of data.

In marketing materials, the phrase "database as a service" (DBaaS) may be used to refer to both relational and non-relational DBMS components that are given via the internet. Users of DBMSs include application programmers, Database Administrators (DBAs), and end users.

Database Administrators are typically the only people who work directly with a DBMS. Today, end users read and write to databases using front-end interfaces made by programmers, while programmers use cloud APIs to connect with DBMSs.



## Three Parts that make up the Database System are:

- Query Processor
- Storage Manager
- Disk Storage

The explanations for these are provided below:

### 1. Query Processor

The query processing is handled by the query processor, as the name implies. It executes the user's query, to put it simply. In this way, the query processor aids the database system in making data access simple and easy. The query processor's primary duty is to successfully execute the query. The Query Processor transforms (or interprets) the user's application program-provided requests into instructions that a computer can understand.

#### Components of the Query Processor

- **DDL Interpreter:**

Data Definition Language is what DDL stands for. As implied by the name, the DDL Interpreter interprets DDL statements like those used in schema definitions (such as create, remove, etc.). This interpretation yields a set of tables that include the meta-data (data of data) that is kept in the data dictionary. Metadata may be stored in a data dictionary. In essence, it is a part of the disc storage that will be covered in a later section of this article.

- **DML Compiler:**

Compiler for DML Data Manipulation Language is what DML stands for. In keeping with its name, the DML Compiler converts DML statements like select, update, and delete into low-level instructions or simply machine-readable object code, to enable execution. The optimization of queries is another function of the DML compiler. Since a single question can typically be translated into a number of evaluation plans. As a result, some optimization is needed to select the evaluation plan with the lowest cost out of all the options. This process, known as query optimization, is exclusively carried out by the DML compiler. Simply put, query optimization determines the most effective technique to carry out a query.

- **Embedded DML Pre-compiler:**

Before the query evaluation, the embedded DML commands in the application program (such as SELECT, FROM, etc., in SQL) must be pre-compiled into standard procedural calls (program instructions that the host language can understand). Therefore, the DML statements which are embedded in an application program must be converted into routine calls by the Embedded DML Pre-compiler.

- **Query Optimizer:**

It starts by taking the evaluation plan for the question, runs it, and then returns the result. Simply said, the query evaluation engine evaluates the SQL commands used to access the database's contents before returning the result of the query. In a nutshell, it is in charge of analyzing the queries and running the object code that the DML Compiler produces. Apache Drill, Presto, and other Query Evaluation Engines are a few examples.

### 2. Storage Manager:

An application called Storage Manager acts as a conduit between the queries made and the data kept in the database. Another name for it is Database Control System. By applying the restrictions and running the DCL instructions, it keeps the database's consistency and integrity. It is in charge of retrieving, storing, updating, and removing data from the database.



## Components of Storage Manager

Following are the components of Storage Manager:

- **Integrity Manager:**

Whenever there is any change in the database, the Integrity manager will manage the integrity constraints.

- **Authorization Manager:**

Authorization manager verifies the user that he is valid and authenticated for the specific query or request.

- **File Manager:**

All the files and data structure of the database are managed by this component.

- **Transaction Manager:**

It is responsible for making the database consistent before and after the transactions. Concurrent processes are generally controlled by this component.

- **Buffer Manager:**

The transfer of data between primary and main memory and managing the cache memory is done by the buffer manager.

## 3. Disk Storage

A DBMS can use various kinds of Data Structures as a part of physical system implementation in the form of disk storage.

Components of Disk Storage

Following are the components of Disk Manager:

- **Data Dictionary:**

It contains the metadata (data of data), which means each object of the database has some information about its structure. So, it creates a repository which contains the details about the structure of the database object.

- **Data Files:**

This component stores the data in the files.

- **Indices:**

These indices are used to access and retrieve the data in a very fast and efficient way

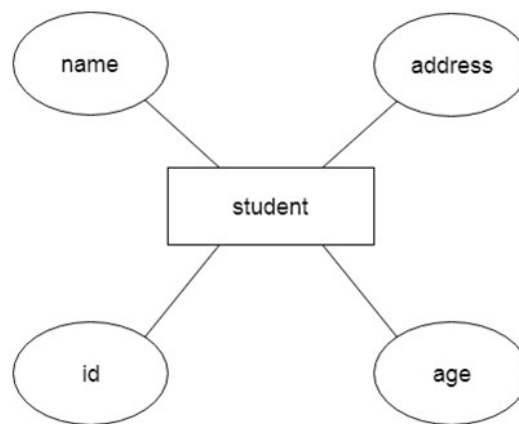
## UNIT II

### INTRODUCTION:

ER (Entity Relationship) Diagram in DBMS

- ER model stands for an Entity-Relationship model. It is a high-level data model. This model is used to define the data elements and relationship for a specified system.
- It develops a conceptual design for the database. It also develops a very simple and easy to design view of data.
- In ER modeling, the database structure is portrayed as a diagram called an entity-relationship diagram.

**For example,** Suppose we design a school database. In this database, the student will be an entity with attributes like address, name, id, age, etc. The address can be another entity with attributes like city, street name, pin code, etc and there will be a relationship between them.



### Uses of ER Diagrams in DBMS:











- ER diagrams are used to represent the E-R model in a database, which makes them easy to be converted into relations (tables).
- ER diagrams provide the purpose of real-world modeling of objects which makes them intently useful.
- ER diagrams require no technical knowledge and no hardware support.
- These diagrams are very easy to understand and easy to create even for a naive user.
- It gives a standard solution for visualizing the data logically.

### Symbols Used in ER Model

ER Model is used to model the logical view of the system from a data perspective which consists of these symbols:

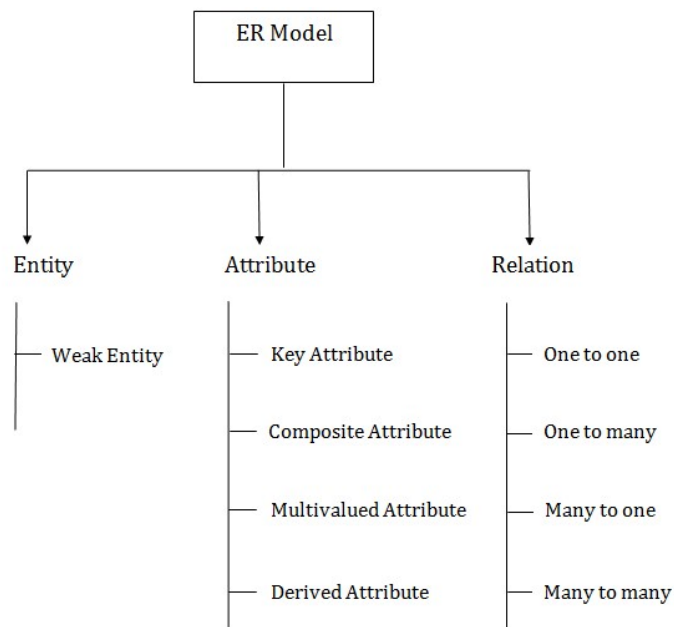
- **Rectangles:** Rectangles represent Entities in ER Model.
- **Ellipses:** Ellipses represent Attributes in ER Model.
- **Diamond:** Diamonds represent Relationships among Entities.
- **Lines:** Lines represent attributes to entities and entity sets with other relationship types.
- **Double Ellipse:** Double Ellipses represent [Multi-Valued Attributes](#).
- **Double Rectangle:** Double Rectangle represents a Weak Entity.

Table shows the notation used in E-R diagram

Symbol	Notation	Meaning
	Rectangle	Entity set
	Doubly outlined box or Doubly outlined rectangle	Weak Entity set
	Diamond	Relationship set
	Doubly outlined diamond	Identifying Relationship set
	Ellipse	Attribute
	Ellipse with underline	Primary Key Attribute
	Doubly lined Ellipse	Multi-valued Attribute
	Dashed Ellipse	Derived Attribute
	Line	Link between an entity set & attribute, and Link between an entity set & Relationship set
	Double Line	Total Participation

*Symbols used in ER Diagram*

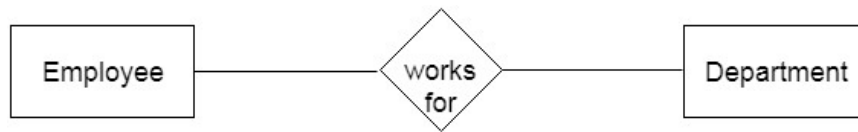
## Basic Concepts of ER Diagram



## 1. Entity:

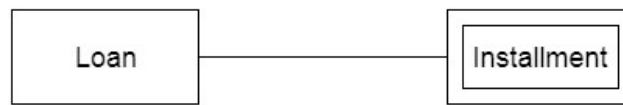
An entity may be any object, class, person or place. In the ER diagram, an entity can be represented as rectangles.

Consider an organization as an example- manager, product, employee, department etc. can be taken as an entity.



### a. Weak Entity

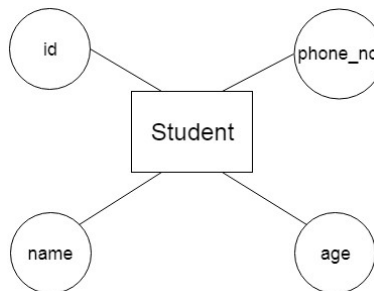
An entity that depends on another entity called a weak entity. The weak entity doesn't contain any key attribute of its own. The weak entity is represented by a double rectangle.



## 2. Attribute

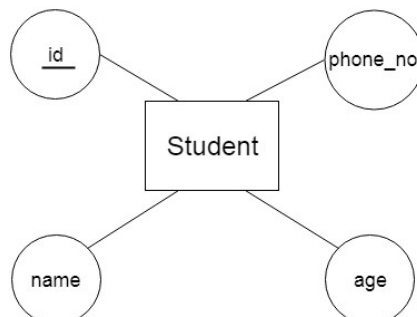
The attribute is used to describe the property of an entity. Eclipse is used to represent an attribute.

**For example,** id, age, contact number, name, etc. can be attributes of a student.



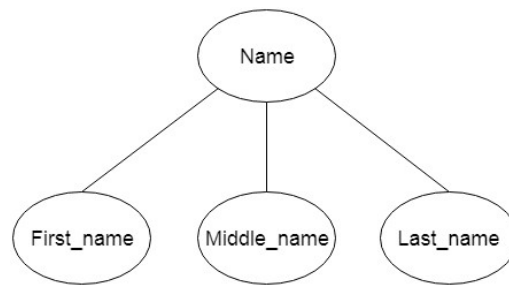
### a. Key Attribute

The key attribute is used to represent the main characteristics of an entity. It represents a primary key. The key attribute is represented by an ellipse with the text underlined.



### b. Composite Attribute

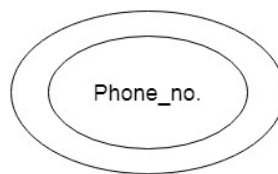
An attribute that composed of many other attributes is known as a composite attribute. The composite attribute is represented by an ellipse, and those ellipses are connected with an ellipse.



### c. Multivalued Attribute

An attribute can have more than one value. These attributes are known as a multivalued attribute. The double oval is used to represent multivalued attribute.

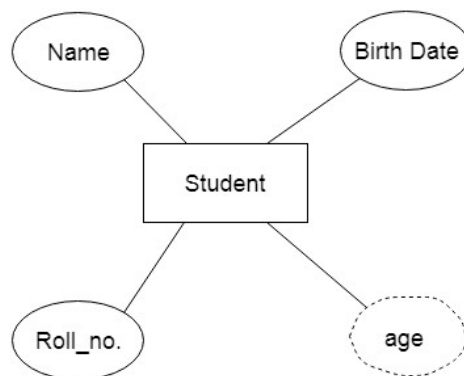
**For example,** a student can have more than one phone number.



### d. Derived Attribute

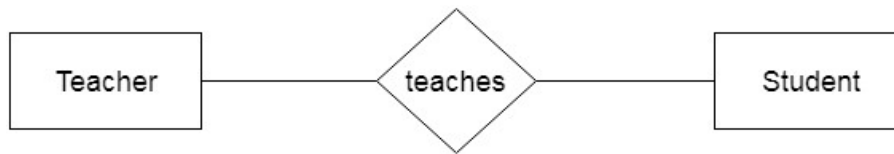
An attribute that can be derived from other attribute is known as a derived attribute. It can be represented by a dashed ellipse.

**For example,** A person's age changes over time and can be derived from another attribute like Date of birth.



## 3. Relationship

A relationship is used to describe the relation between entities. Diamond or rhombus is used to represent the relationship.



Types of relationship are as follows:

**a. One-to-One Relationship**

When only one instance of an entity is associated with the relationship, then it is known as one to one relationship.

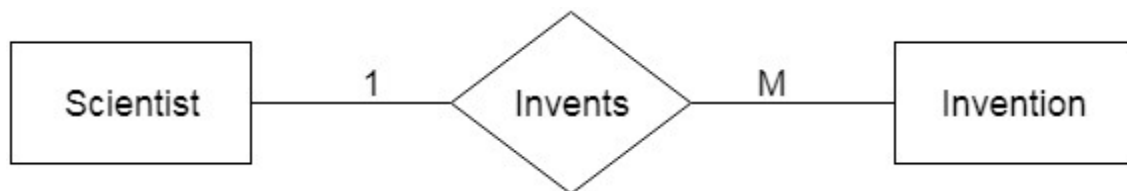
**For example,** A female can marry to one male, and a male can marry to one female.



**b. One-to-many relationship**

When only one instance of the entity on the left, and more than one instance of an entity on the right associates with the relationship then this is known as a one-to-many relationship.

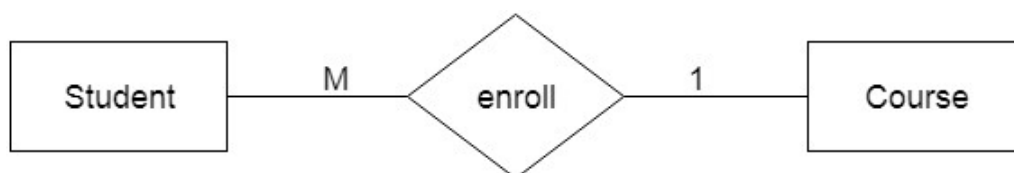
**For example,** Scientist can invent many inventions, but the invention is done by the only specific scientist.



**c. Many-to-one relationship**

When more than one instance of the entity on the left, and only one instance of an entity on the right associates with the relationship then it is known as a many-to-one relationship.

**For example,** Student enrolls for only one course, but a course can have many students.



**d. Many-to-many relationship**

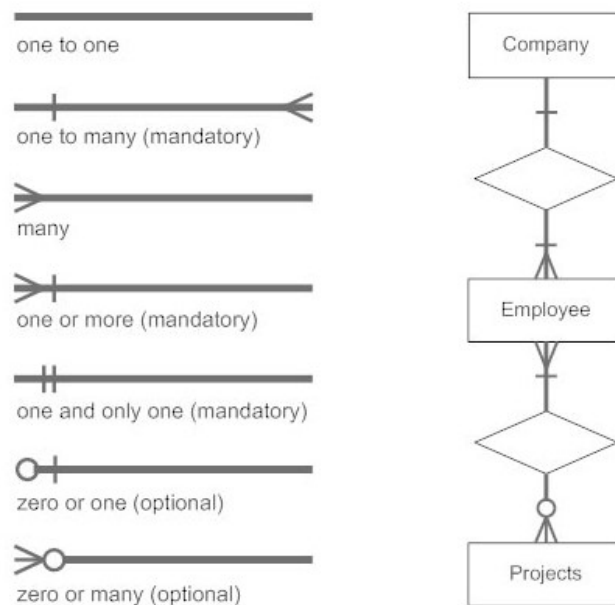
When more than one instance of the entity on the left, and more than one instance of an entity on the right associates with the relationship then it is known as a many-to-many relationship.

**For example,** Employee can assign by many projects and project can have many employees.



### Notation of ER diagram

Database can be represented using the notations. In ER diagram, many notations are used to express the cardinality. These notations are as follows:



**Fig: Notations of ER diagram**

### Design Issues

#### 1) Use of Entity Set vs Attributes

The use of an entity set or attribute depends on the structure of the real-world enterprise that is being modelled and the semantics associated with its attributes. It leads to a mistake when the user use the primary key of an entity set as an attribute of another entity set. Instead, he should use the relationship to do so. Also, the primary key attributes are implicit in the relationship set, but we designate it in the relationship sets.

#### 2) Use of Entity Set vs. Relationship Sets

It is difficult to examine if an object can be best expressed by an entity set or relationship set. To understand and determine the right use, the user need to designate a relationship set for describing an action that occurs in-between the entities. If there is a requirement of representing the object as a relationship set, then its better not to mix it with the entity set.

#### 3) Use of Binary vs n-ary Relationship Sets

Generally, the relationships described in the databases are binary relationships. However, non-binary relationships can be represented by several binary relationships. For example, we can create and represent a ternary relationship 'parent' that may relate to a child, his father, as well as his mother. Such relationship can also be represented by two binary relationships i.e, mother and father, that may relate to their child. Thus, it is possible to represent a non-binary relationship by a set of distinct binary relationships.

#### 4) Placing Relationship Attributes

The cardinality ratios can become an effective measure in the placement of the relationship attributes. So, it is better to associate the attributes of one-to-one or one-to-many relationship sets with any participating entity sets, instead of any relationship set. The decision of placing the specified attribute as a relationship or entity attribute should possess the characteristics of the real world enterprise that is being modelled.

**For example**, if there is an entity which can be determined by the combination of participating entity sets, instead of determining it as a separate entity. Such type of attribute must be associated with the many-to-many relationship sets.

Thus, it requires the overall knowledge of each part that is involved in designing and modelling an ER diagram. The basic requirement is to analyse the real-world enterprise and the connectivity of one entity or attribute with other.

#### **Mapping Cardinalities:**

##### **DBMS**

DBMS stands for Database Management System, which is a tool, or a software used to do various operations on a Database like the Creation of the Database, Deletion of the Database, or Updating the current Database. To simplify processing and data querying, the most popular types of Databases currently in use typically model their data as rows and columns in a set of tables. The data may then be handled, updated, regulated, and structured with ease. For writing and querying data, most Databases employ Structured Query Language (SQL).

##### **Cardinality**

Cardinality means how the entities are arranged to each other or what is the relationship structure between entities in a relationship set. In a Database Management System, Cardinality represents a number that denotes how many times an entity is participating with another entity in a relationship set. The Cardinality of DBMS is a very important attribute in representing the structure of a Database. In a table, the number of rows or tuples represents the Cardinality.

##### **Cardinality Ratio**

Cardinality ratio is also called **Cardinality Mapping**, which represents the mapping of one entity set to another entity set in a relationship set. We generally take the example of a binary relationship set where two entities are mapped to each other.

Cardinality is very important in the Database of various businesses. For example, if we want to track the purchase history of each customer then we can use the one-to-many cardinality to find the data of a specific customer. The Cardinality model can be used in Databases by Database Managers for a variety of purposes, but corporations often use it to evaluate customer or inventory data.

There are four types of Cardinality Mapping in Database Management Systems:

1. One to one
2. Many to one
3. One to many
4. Many to many

##### **One to One**

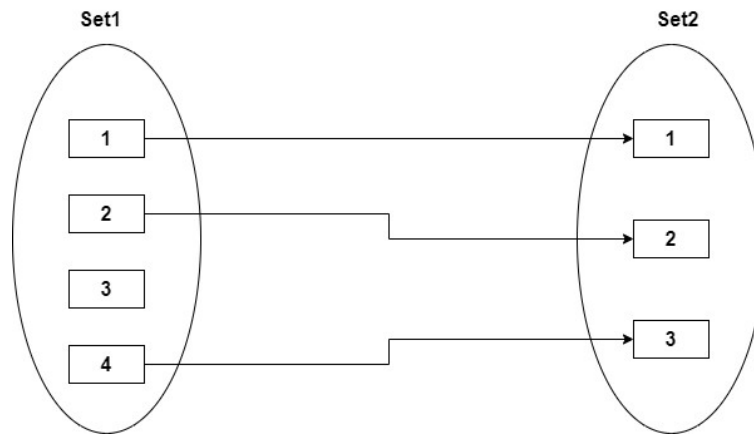
One to one cardinality is represented by a **1:1** symbol. In this, there is at most one relationship from one entity to another entity. There are a lot of examples of one-to-one cardinality in real life databases.



**For example**, one student can have only one student id, and one student id can belong to only one student. So, the relationship mapping between student and student id will be one to one cardinality mapping.

Another example is the relationship between the director of the school and the school because one school can have a maximum of one director, and one director can belong to only one school.

Note: it is not necessary that there would be a mapping for all entities in an entity set in one-to-one cardinality. Some entities cannot participate in the mapping.

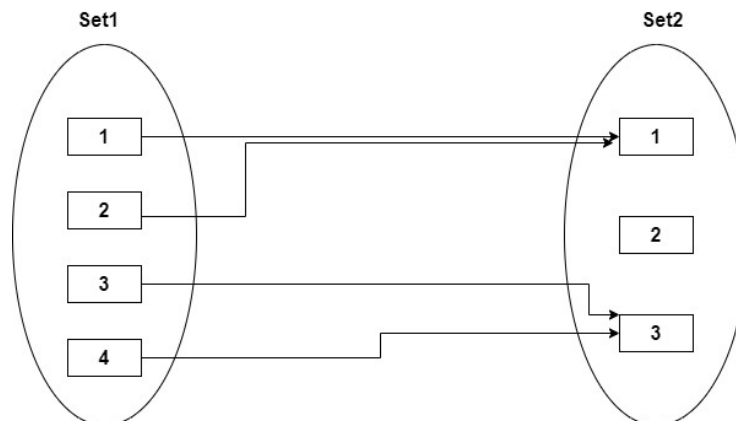


### Many to One Cardinality:

In many to one cardinality mapping, from set 1, there can be multiple sets that can make relationships with a single entity of set 2. Or we can also describe it as from set 2, and one entity can make a relationship with more than one entity of set 1.

One to one Cardinality is the subset of Many to one Cardinality. It can be represented by **M:1**.

**For example**, there are multiple patients in a hospital who are served by a single doctor, so the relationship between patients and doctors can be represented by Many to one Cardinality.

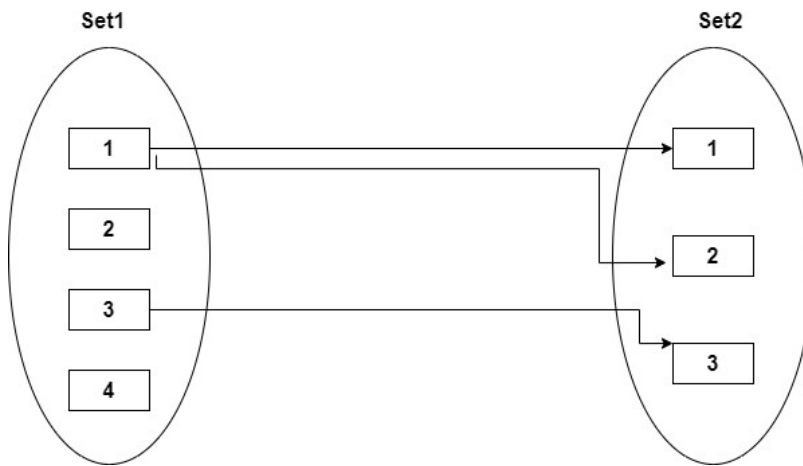


### One to Many Cardinalities:

In One-to-many cardinality mapping, from set 1, there can be a maximum single set that can make relationships with a single or more than one entity of set 2. Or we can also describe it as from set 2, more than one entity can make a relationship with only one entity of set 1.

One to one cardinality is the subset of One-to-many Cardinality. It can be represented by **1: M**.

**For Example**, in a hospital, there can be various compounds, so the relationship between the hospital and compounds can be mapped through One-to-many Cardinality.



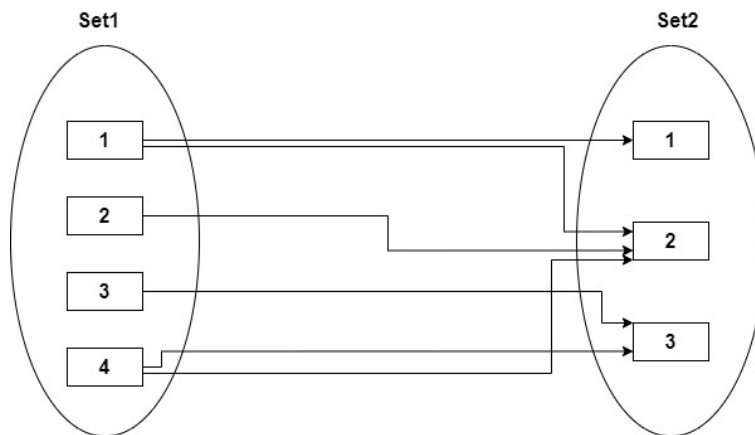
### Many to Many Cardinalities:

In many, many cardinalities mapping, there can be one or more than one entity that can associate with one or more than one entity of set 2. In the same way from the end of set 2, one or more than one entity can make a relation with one or more than one entity of set 1.

It is represented by **M: N** or **N: M**.

One to one cardinality, One to many cardinalities, and Many to one cardinality is the subset of the many to many cardinalities.

**For Example**, in a college, multiple students can work on a single project, and a single student can also work on multiple projects. So, the relationship between the project and the student can be represented by many to many cardinalities.



### Appropriate Mapping Cardinality

Evidently, the real-world context in which the relation set is modeled determines the Appropriate Mapping Cardinality for a specific relation set.

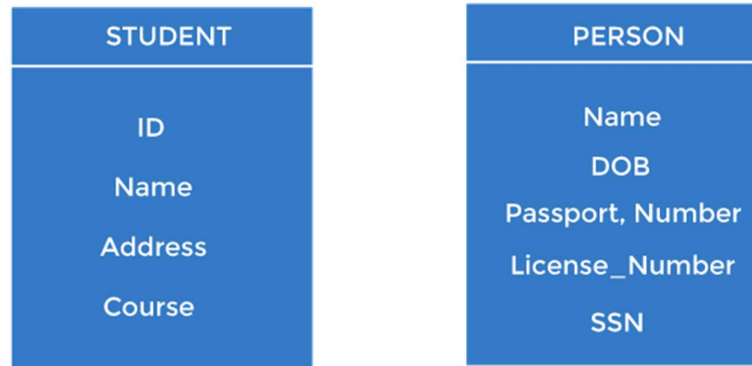
- We can combine relational tables with many involved tables if the Cardinality is one-to-many or many-to-one.
- One entity can be combined with a relation table if it has a one-to-one relationship and total participation, and two entities can be combined with their relation to form a single table if both of them have total participation.
- We cannot mix any two tables if the Cardinality is many-to-many.

### Keys

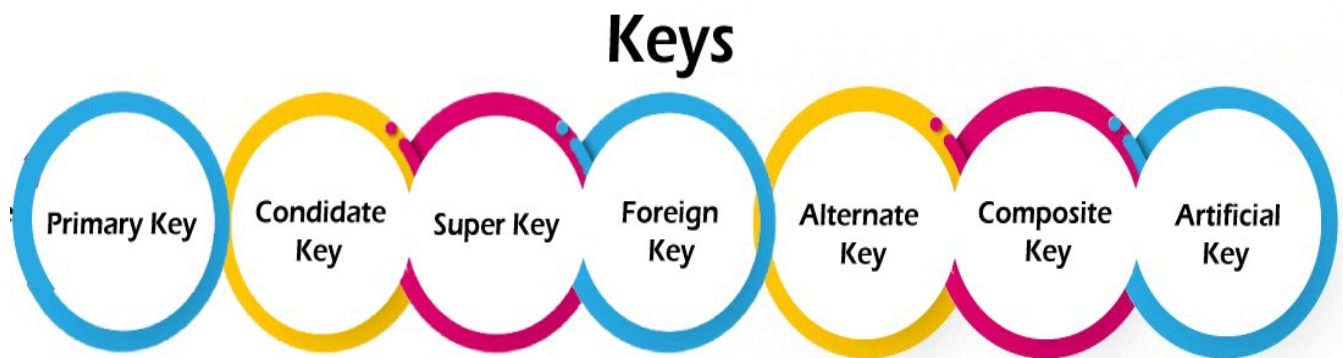
- Keys play an important role in the relational database.

- It is used to uniquely identify any record or row of data from the table. It is also used to establish and identify relationships between tables.

**For example,** ID is used as a key in the Student table because it is unique for each student. In the PERSON table, passport\_number, license\_number, SSN are keys since they are unique for each person.

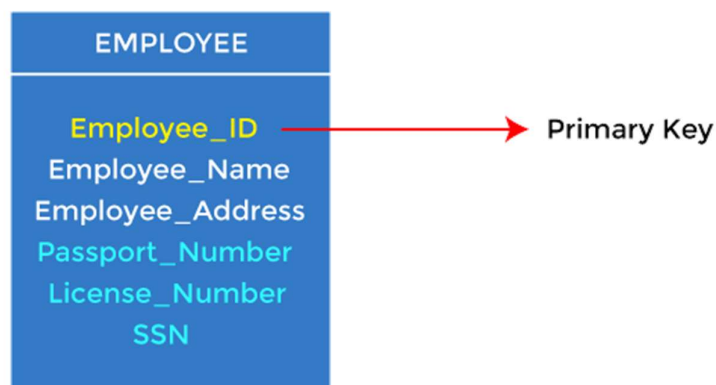


Types of keys:



### 1. Primary key

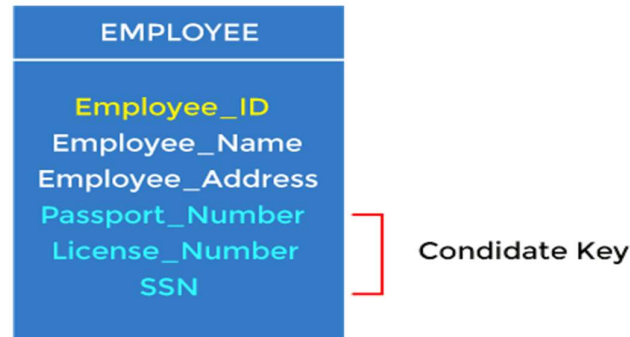
- It is the first key used to identify one and only one instance of an entity uniquely. An entity can contain multiple keys, as we saw in the PERSON table. The key which is most suitable from those lists becomes a primary key.
- In the EMPLOYEE table, ID can be the primary key since it is unique for each employee. In the EMPLOYEE table, we can even select License\_Number and Passport\_Number as primary keys since they are also unique.
- For each entity, the primary key selection is based on requirements and developers.



## 2. Candidate key

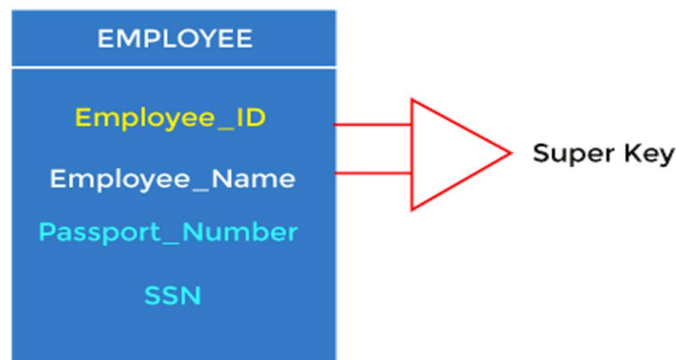
- A candidate key is an attribute or set of attributes that can uniquely identify a tuple.
- Except for the primary key, the remaining attributes are considered a candidate key. The candidate keys are as strong as the primary key.

**For example:** In the EMPLOYEE table, id is best suited for the primary key. The rest of the attributes, like SSN, Passport\_Number, License\_Number, etc., are considered a candidate key.



## 3. Super Key

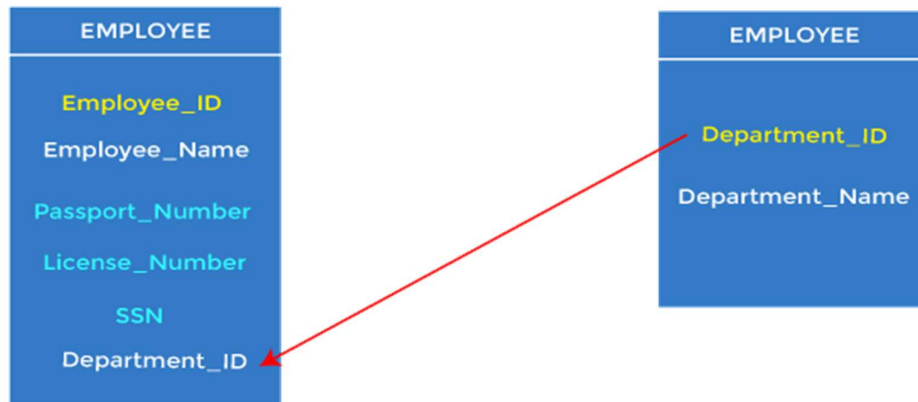
Super key is an attribute set that can uniquely identify a tuple. A super key is a superset of a candidate key.



**For example:** In the above EMPLOYEE table, for(EMPLOYEE\_ID, EMPLOYEE\_NAME), the name of two employees can be the same, but their EMPLOYEE\_ID can't be the same. Hence, this combination can also be a key.

## 4. Foreign key

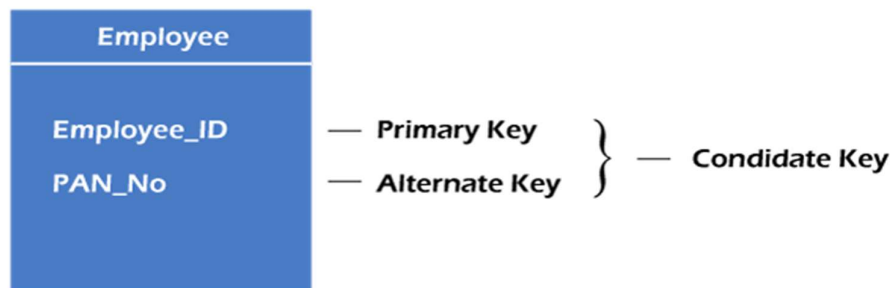
- Foreign keys are the column of the table used to point to the primary key of another table.
- Every employee works in a specific department in a company, and employee and department are two different entities. So we can't store the department's information in the employee table. That's why we link these two tables through the primary key of one table.
- We add the primary key of the DEPARTMENT table, Department\_Id, as a new attribute in the EMPLOYEE table.
- In the EMPLOYEE table, Department\_Id is the foreign key, and both the tables are related.



## 5. Alternate key

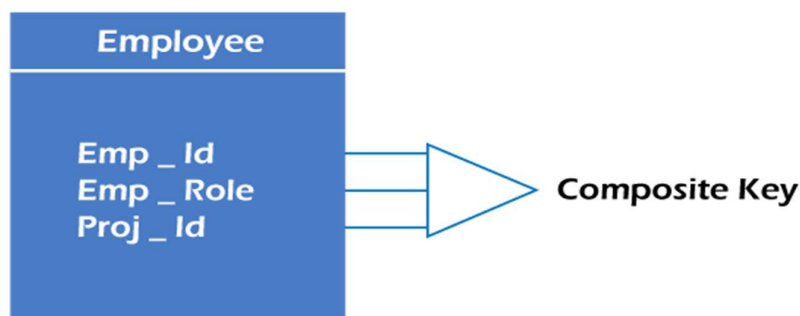
There may be one or more attributes or a combination of attributes that uniquely identify each tuple in a relation. These attributes or combinations of the attributes are called the candidate keys. One key is chosen as the primary key from these candidate keys, and the remaining candidate key, if it exists, is termed the alternate key. **In other words**, the total number of the alternate keys is the total number of candidate keys minus the primary key. The alternate key may or may not exist. If there is only one candidate key in a relation, it does not have an alternate key.

**For example**, employee relation has two attributes, Employee\_Id and PAN\_No, that act as candidate keys. In this relation, Employee\_Id is chosen as the primary key, so the other candidate key, PAN\_No, acts as the Alternate key.



## 6. Composite key

Whenever a primary key consists of more than one attribute, it is known as a composite key. This key is also known as Concatenated Key.



**For example**, in employee relations, we assume that an employee may be assigned multiple roles, and an employee may work on multiple projects simultaneously. So the primary key will be composed of all three attributes, namely Emp\_ID, Emp\_role, and Proj\_ID in combination. So these attributes act as a composite key since the primary key comprises more than one attribute.

## 7. Artificial key

The key created using arbitrarily assigned data are known as artificial keys. These keys are created when a primary key is large and complex and has no relationship with many other relations. The data values of the artificial keys are usually numbered in a serial order.

**For example**, the primary key, which is composed of Emp\_ID, Emp\_role, and Proj\_ID, is large in employee relations. So it would be better to add a new virtual attribute to identify each tuple in the relation uniquely.



### ER DIAGRAMS:

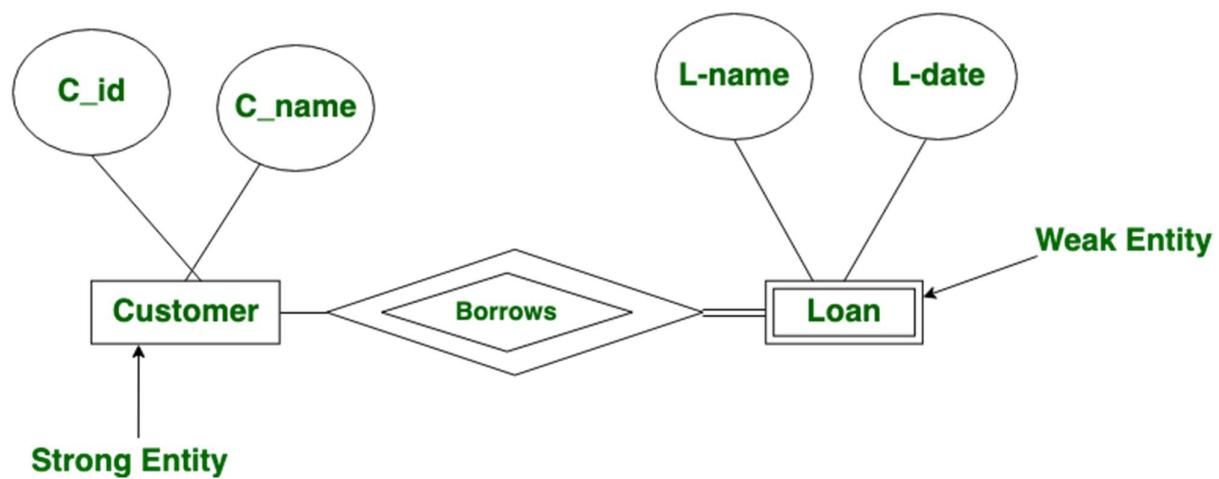
- The very first step is identifying all the Entities, and place them in a Rectangle, and labeling them accordingly.
- The next step is to identify the relationship between them and place them accordingly using the Diamond, and make sure that, Relationships are not connected to each other.
- Attach attributes to the entities properly.
- Remove redundant entities and relationships.
- Add proper colors to highlight the data present in the database.

### Weak Entity Set in ER diagrams

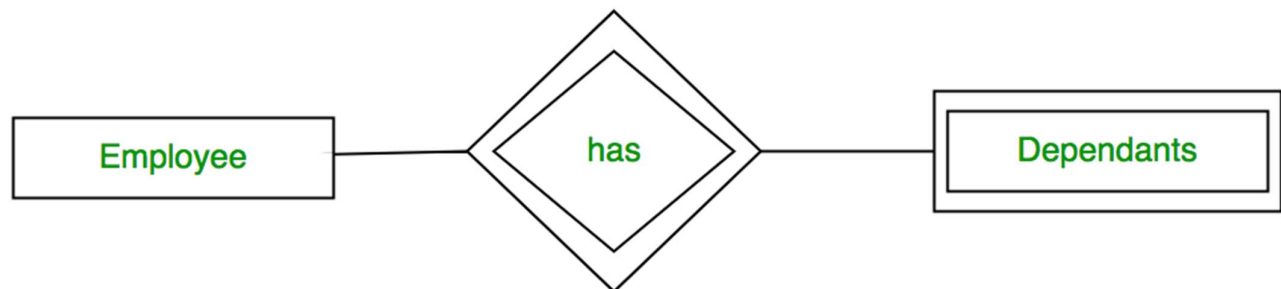
An entity type should have a key attribute which uniquely identifies each entity in the entity set, but there exists some entity type for which key attribute can't be defined. These are called Weak Entity type. The entity sets which do not have sufficient attributes to form a primary key are known as **weak entity sets** and the entity sets which have a primary key are known as strong entity sets.

As the weak entities do not have any primary key, they cannot be identified on their own, so they depend on some other entity (known as owner entity). The weak entities have total participation constraint (existence dependency) in its identifying relationship with owner identity. Weak entity types have partial keys. Partial Keys are set of attributes with the help of which the tuples of the weak entities can be distinguished and identified.

**Note** – Weak entity always has total participation but Strong entity may not have total participation. Weak entity is **depend on strong entity** to ensure the existence of weak entity. Like strong entity, weak entity does not have any primary key, It has partial discriminator key. Weak entity is represented by double rectangle. The relation between one strong and one weak entity is represented by double diamond.

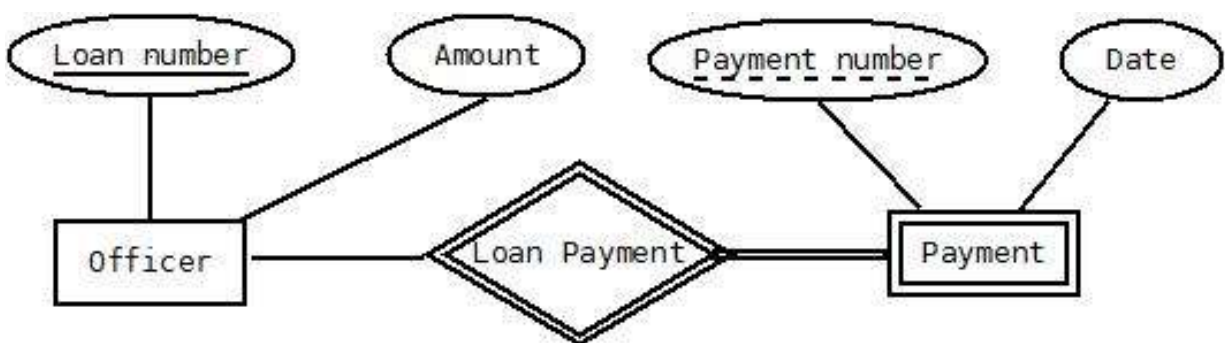


**Weak entities** are represented with **double rectangular** box in the ER Diagram and the identifying relationships are represented with double diamond. Partial Key attributes are represented with dotted lines.



#### Example-1:

In the below ER Diagram, 'Payment' is the weak entity. 'Loan Payment' is the identifying relationship and 'Payment Number' is the partial key. Primary Key of the Loan along with the partial key would be used to identify the records.



#### Example-2:

The existence of rooms is entirely dependent on the existence of a hotel. So room can be seen as the weak entity of the hotel.

#### Example-3:

The bank account of a particular bank has no existence if the bank doesn't exist anymore.

#### Example-4:

A company may store the information of dependents (Parents, Children, Spouse) of an Employee. But the dependents don't have existence without the employee. So Dependent will be weak entity type and Employee will be Identifying Entity type for Dependent.

## Other examples:

Strong entity | Weak entity

Order | Order Item

Employee | Dependent

Class | Section

Host | Logins

**Note** – Strong-Weak entity set always has parent-child relationship.

## Extended E-R Features

It is getting harder and harder to apply the conventional ER paradigm for database modeling as data complexity rises today. The existing ER model needs to be enhanced or improved in order for it to better handle the complicated application in order to reduce the modeling complexity.

The requirements and complexity of complicated databases are represented using enhanced entity-relationship diagrams, which are sophisticated database diagrams very similar to standard ER diagrams.

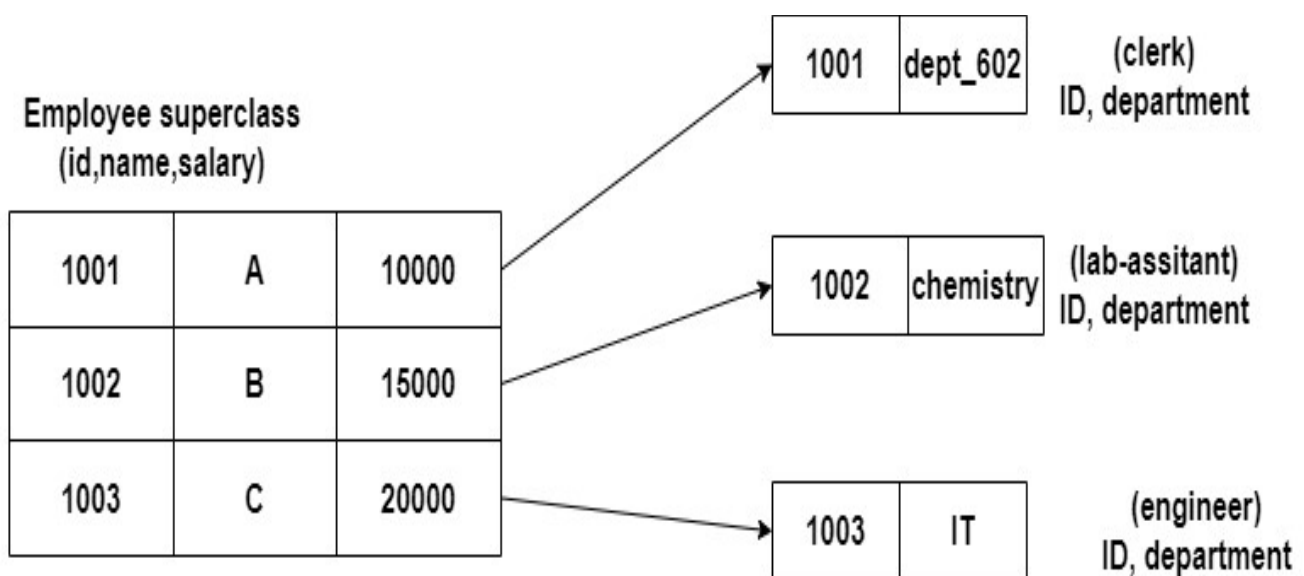
The SubClass and SuperClass, Specialization and Generalization, Union or Category, Aggregation, etc., are displayed using this diagrammatic style.

### Generalization and Specialization

These are two normal kinds of relationships that were added to the normal ER model for enhancement. These are inspired by the object-oriented paradigm, where we divide the code into classes and objects, and in the same way, we have divided entities into subclass and superclasses. Specialized classes are called subclasses, and generalized classes are called superclasses or base classes. We can learn the concept of subclass by 'IS-A' analysis. For example, 'Laptop IS-A computer.' Or 'Clerk IS-A employee.'

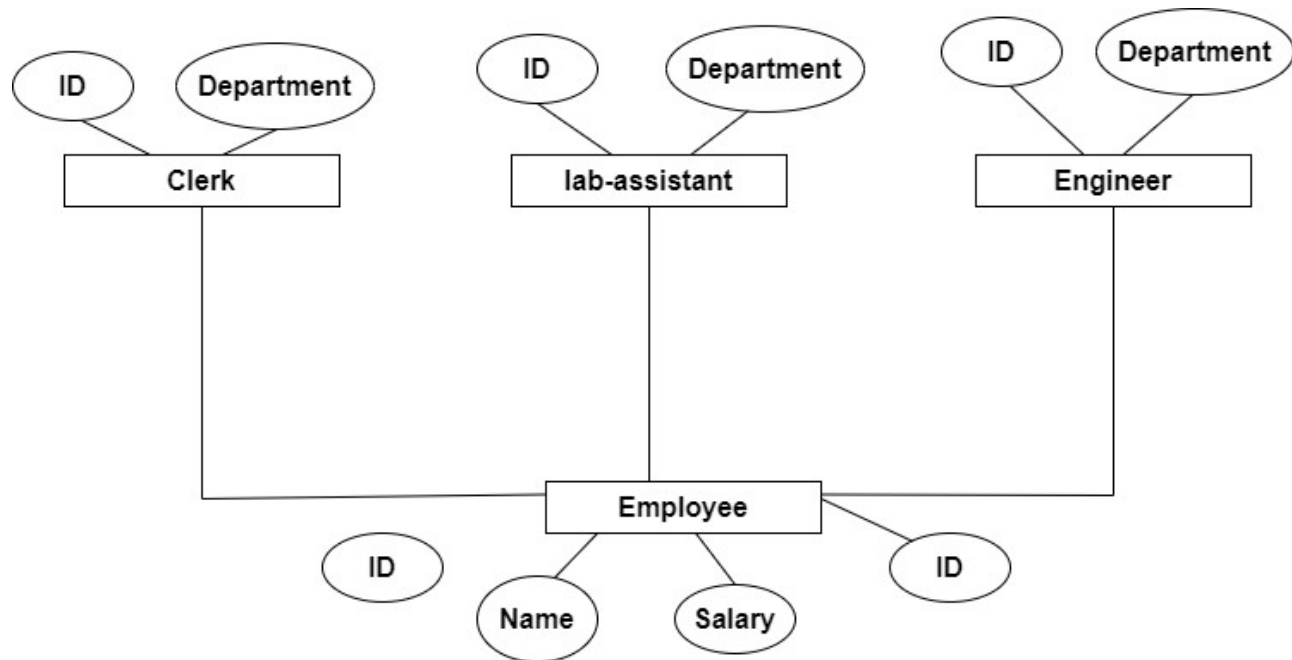
In this relationship, one entity is a subclass or superclass of another entity. For example, in a university, a faculty member or clerk is a specialized class of employees. So an employee is a generalized class, and all others are its subclass.

We can draw the ER diagram for these relationships. Let's suppose we have a superclass Employee and subclasses as a clerk, engineer, and lab assistant.





The Extended ER diagram of the above example will look like this:



In the above example, we have one superclass and three subclasses. Each subclass inherits all the attributes from its superclass so that a lab assistant will have all its attributes, like its name, salary, etc.

### Constraints

There are two types of constraints on subclasses which are described below:

- Total or Partial:

A total subclass relationship is one where the union of all the subclasses is equal to the superclass. It means if every superclass entity has some subclass entity, then it is called a total subclass relationship. Let's suppose if the union of all the subclasses (engineer, clerk, lab assistant) is equal to the total employee. Then the relationship is total. In the above example, it is a total relationship.

If all the entities of a superclass are not associated with a subclass, then it is called a partial subclass relationship.

- Overlapped or Disjoint:

If any entity from the superclass is associated with more than one subclass, then it is known as overlapped subclassing, and if it is associated with zero or only one subclass, then it is called disjoint subclassing.

Note: The above two constraints are independent of each other, and they follow the transitive property.

### Multiple Inheritance

When one subclass is associated with more than one superclass, then this phenomenon is known as multiple inheritance. In multiple inheritance, the attributes of the subclass will be the union of all the superclass attributes which are associated with it. For example, a teacher is a subclass that can be associated with the superclass of an employee and a superclass of faculty. In the same way, a monitor in the class can be a subclass of a student superclass as well as an alumni superclass.

### UNION

UNION is a different topic from subclassing. Let's suppose we have a vehicle superclass, and we have two subclasses, car and bike. These two subclasses will inherit the attributes from the vehicle superclass. Now we have a UNION of those vehicles which are RTO registered, so we have a UNION of cars and bikes, but they will inherit all the attributes from the vehicle superclass.

## Design of an ER Database Schema:

- The data which is stored in the database at a particular moment of time is called an instance of the database.
- The Overall design of a database is called schema.
- A database schema is the skeleton structure of the database. It represents the logical view of the entire database.
- A Schema contains schema objects like table, foreign key, primary key, views, columns, data types, stored procedure, etc.,
- A database schema can be represented by using the visual diagram. That diagram shows the database objects and relationship with each other.
- A database schema is designed schema is designed by the database designers to help programmers whose software will interact with the database. The process of database creation is called data modelling

The schema diagram can display only some aspects of a schema like the name of record type, data type and constraints. Other aspects can't be specified through the schema diagram. For example, the given figure neither show the data type of each data item nor the relationship among various file.

In the database, actual data changes quite frequently. For example, in the given figure, the database changes whenever we add a new grade or add a student. The data at a particular moment of time is called the instance of the database.

### STUDENT

Name	Student_number	Class	Major
------	----------------	-------	-------

### COURSE

Course_name	Course_number	Credit_hours	Department
-------------	---------------	--------------	------------

### PREREQUISITE

Course_number	Prerequisite_number
---------------	---------------------

### SECTION

Section_identifier	Course_number	Semester	Year	Instructor
--------------------	---------------	----------	------	------------

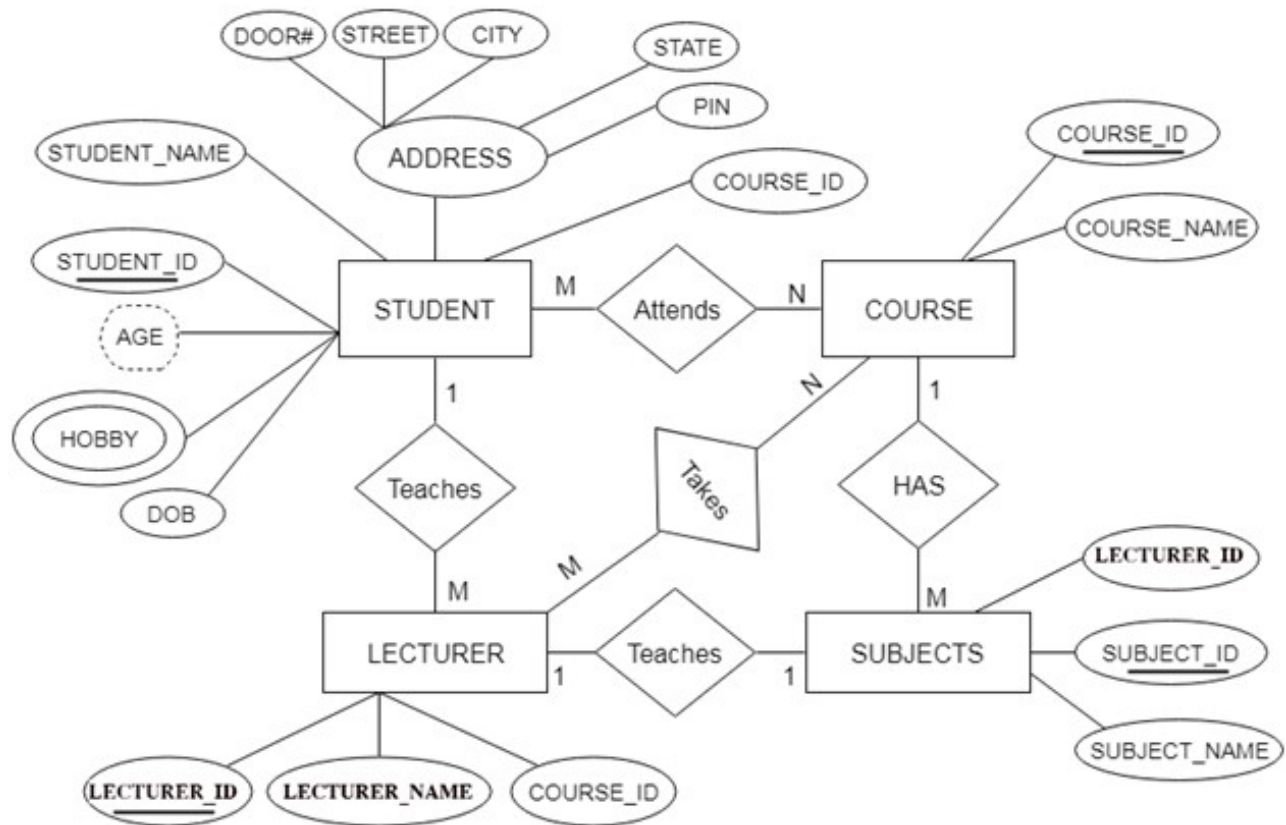
### GRADE\_REPORT

Student_number	Section_identifier	Grade
----------------	--------------------	-------

## 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.

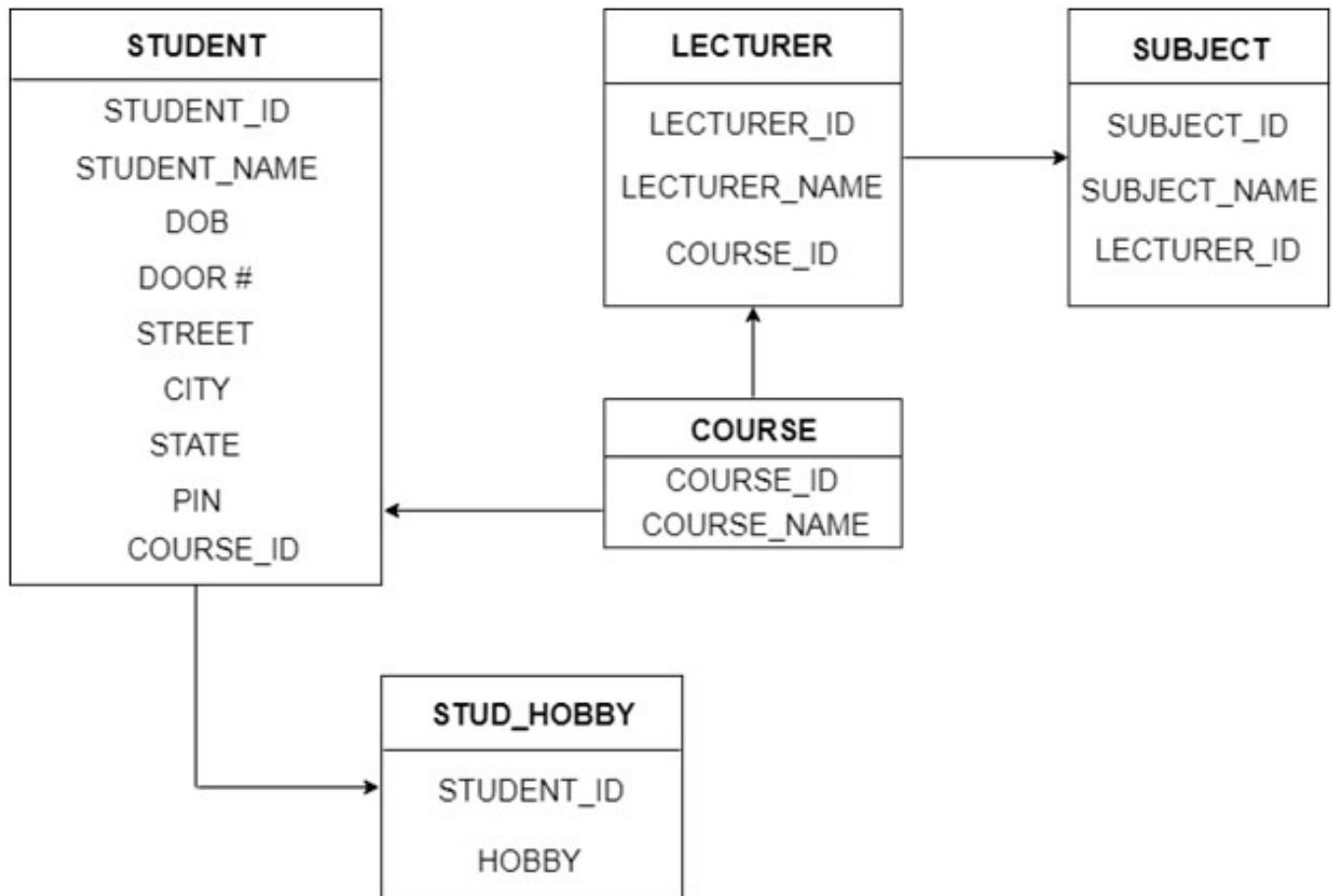
- Composite attribute represented by components.

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**

## UNIT III

### Relational Model in DBMS

Relational model can represent as a table with columns and rows. Each row is known as a tuple. Each table of the column has a name or attribute.

**Domain:** It contains a set of atomic values that an attribute can take.

**Attribute:** It contains the name of a column in a particular table. Each attribute  $A_i$  must have a domain,  $\text{dom}(A_i)$

**Relational instance:** In the relational database system, the relational instance is represented by a finite set of tuples. Relation instances do not have duplicate tuples.

**Relational schema:** A relational schema contains the name of the relation and name of all columns or attributes.

**Relational key:** In the relational key, each row has one or more attributes. It can identify the row in the relation uniquely.

#### Example: STUDENT Relation

NAME	ROLL_NO	PHONE_NO	ADDRESS	AGE
Ram	14795	7305758992	Noida	24
Shyam	12839	9026288936	Delhi	35
Laxman	33289	8583287182	Gurugram	20
Mahesh	27857	7086819134	Ghaziabad	27
Ganesh	17282	9028 9i3988	Delhi	40

- In the given table, NAME, ROLL\_NO, PHONE\_NO, ADDRESS, and AGE are the attributes.
- The instance of schema STUDENT has 5 tuples.
- $t_3 = \langle \text{Laxman}, 33289, 8583287182, \text{Gurugram}, 20 \rangle$

#### Properties of Relations

- Name of the relation is distinct from all other relations.
- Each relation cell contains exactly one atomic (single) value
- Each attribute contains a distinct name
- Attribute domain has no significance
- tuple has no duplicate value
- Order of tuple can have a different sequence

#### Structure of Relational Model:

A relational database consists of a collection of **tables**, each of which is assigned a unique name. For example, consider the instructor table of Figure 2.1, which stores

information about instructors. The table has four column headers: *ID*, *name*, *dept\_name*, and *salary*. Each row of this table records information about an *instructor*, consisting of the instructor's *ID*, *name*, *dept\_name*, and *salary*. Similarly, the *course* table of Figure 2.2 stores information about courses, consisting of a *course\_id*, *title*, *dept\_name*, and *credits*, for each course. Note that each instructor is identified by the value of the column *ID*, while each course is identified by the value of the column *course\_id*. Figure 2.3 shows a third table, *prereq*, which stores the prerequisite courses for each course. The table has two columns, *course\_id* and *prereq\_id*. Each row consists of a pair of course identifiers such that the second course is a prerequisite for the first course.

Thus, a row in the *prereq* table indicates that two courses are related in the sense that one course is a prerequisite for the other. As another example, we consider the table *instructor*, a row in the table can be thought of as representing the relationship between a specified *ID* and the corresponding values for *name*, *dept\_name*, and *salary* values.

<i>ID</i>	<i>name</i>	<i>dept_name</i>	<i>salary</i>
10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	60000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	62000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000

**Figure 2.1** The *instructor* relation.

In general, a row in a table represents a relationship among a set of values. Since a table is a collection of such relationships, there is a close correspondence between the concept of table and the mathematical concept of relation, from which the relational data model takes its name. In mathematical terminology, a ***tuple*** is simply a sequence (or list) of values. A relationship between  $n$  values is represented mathematically by an ***n-tuple*** of values, i.e., a tuple with  $n$  values, which corresponds to a row in a table.

<i>course_id</i>	<i>title</i>	<i>dept_name</i>	<i>credits</i>
BIO-101	Intro. to Biology	Biology	4
BIO-301	Genetics	Biology	4
BIO-399	Computational Biology	Biology	3
CS-101	Intro. to Computer Science	Comp. Sci.	4
CS-190	Game Design	Comp. Sci.	4
CS-315	Robotics	Comp. Sci.	3
CS-319	Image Processing	Comp. Sci.	3
CS-347	Database System Concepts	Comp. Sci.	3
EE-181	Intro. to Digital Systems	Elec. Eng.	3
FIN-201	Investment Banking	Finance	3
HIS-351	World History	History	3
MU-199	Music Video Production	Music	3
PHY-101	Physical Principles	Physics	4

**Figure 2.2** The *course* relation.

<i>course_id</i>	<i>prereq_id</i>
BIO-301	BIO-101
BIO-399	BIO-101
CS-190	CS-101
CS-315	CS-101
CS-319	CS-101
CS-347	CS-101
EE-181	PHY-101

**Figure 2.3** The *prereq* relation.

Thus, in the relational model the term **relation** is used to refer to a table, while the term **tuple** is used to refer to a row. Similarly, the term **attribute** refers to a column of a table.

Examining Figure 2.1, we can see that the relation *instructor* has four attributes: *ID*, *name*, *dept\_name*, and *salary*.

We use the term **relation instance** to refer to a specific instance of a relation, i.e., containing a specific set of rows. The instance of *instructor* shown in Figure 2.1 has 12 tuples, corresponding to 12 instructors.

The order in which tuples appear in a relation is irrelevant, since a relation is a set of tuples. Thus, whether the tuples of a relation are listed in sorted order, as in Figure 2.1, or are unsorted, as in Figure 2.4, does not matter; the relations in the two figures are the same, since both contain the same set of tuples. For ease of exposition, we will mostly show the relations sorted by their first attribute.

<i>ID</i>	<i>name</i>	<i>dept_name</i>	<i>salary</i>
22222	Einstein	Physics	95000
12121	Wu	Finance	90000
32343	El Said	History	60000
45565	Katz	Comp. Sci.	75000
98345	Kim	Elec. Eng.	80000
76766	Crick	Biology	72000
10101	Srinivasan	Comp. Sci.	65000
58583	Califieri	History	62000
83821	Brandt	Comp. Sci.	92000
15151	Mozart	Music	40000
33456	Gold	Physics	87000
76543	Singh	Finance	80000

**Figure 2.4** Unsorted display of the *instructor* relation.

For each attribute of a relation, there is a set of permitted values, called the **domain** of that attribute. Thus, the domain of the *salary* attribute of the *instructor* relation is the set of all possible salary values, while the domain of the *name* attribute is the set of all possible instructor names.

We require that, for all relations  $r$ , the domains of all attributes of  $r$  be atomic. A domain is **atomic** if elements of the domain are considered to be indivisible units. For example, suppose the table *instructor* had an attribute *phone\_number*, which can store a set of phone numbers corresponding to the instructor. Then the domain of *phone\_number* would not be atomic, since an element of the domain is a set of phone numbers, and it has subparts, namely the individual phone numbers in the set.

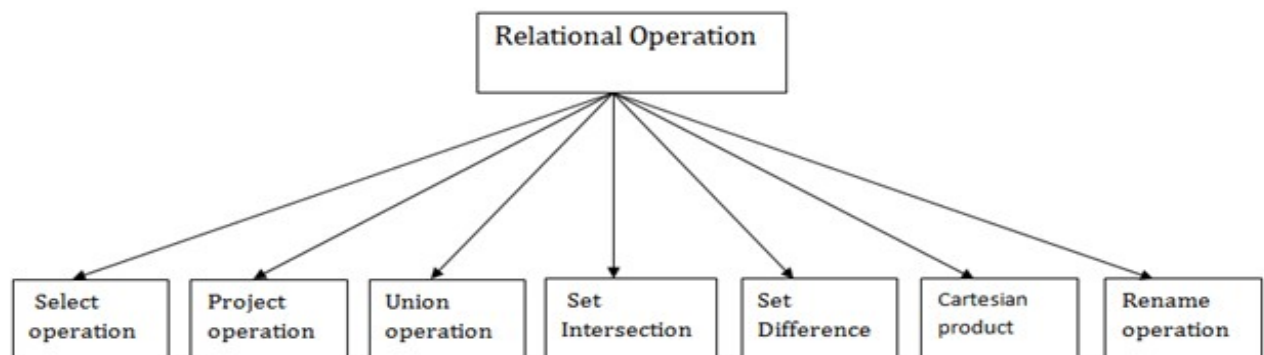
The important issue is not what the domain itself is, but rather how we use domain elements in our database. Suppose now that the phone number attribute stores a single *phone\_number*. Even then, if we split the value from the phone number attribute into a country code, an area code and a local number, we would be treating it as a nonatomic value. If we treat each phone number as a single indivisible unit, then the attribute *phone\_number* would have an atomic domain.

The **null** value is a special value that signifies that the value is unknown or does not exist. For example, suppose as before that we include the attribute *phone\_number* in the *instructor* relation. It may be that an instructor does not have a phone number at all, or that the telephone number is unlisted. We would then have to use the null value to signify that the value is unknown or does not exist. We shall see later that null values cause a number of difficulties when we access or update the database, and thus should be eliminated if at all possible.

### Relational Algebra:

Relational algebra is a procedural query language. It gives a step by step process to obtain the result of the query. It uses operators to perform queries.

Types of Relational operation



#### 1. Select Operation:

- The select operation selects tuples that satisfy a given predicate.
- It is denoted by sigma ( $\sigma$ ).

1. Notation:  $\sigma p(r)$

**Where:**

$\sigma$  is used for selection predicate

$r$  is used for relation



**p** is used as a propositional logic formula which may use connectors like: AND OR and NOT. These relational can use as relational operators like =, ≠, ≥, <, >, ≤.

**For example: LOAN Relation**

BRANCH_NAME	LOAN_NO	AMOUNT
Downtown	L-17	1000
Redwood	L-23	2000
Perryride	L-15	1500
Downtown	L-14	1500
Mianus	L-13	500
Roundhill	L-11	900
Perryride	L-16	1300

**Input:**

1.  $\sigma$  BRANCH\_NAME="perryride" (LOAN)

**Output:**

BRANCH_NAME	LOAN_NO	AMOUNT
Perryride	L-15	1500
Perryride	L-16	1300

2. Project Operation:

- This operation shows the list of those attributes that we wish to appear in the result. Rest of the attributes are eliminated from the table.
- It is denoted by  $\Pi$ .
- 1. Notation:  $\Pi A_1, A_2, A_n (r)$

**Where**

**A1, A2, A3** is used as an attribute name of relation **r**.

**Example: CUSTOMER RELATION**

NAME	STREET	CITY
Jones	Main	Harrison
Smith	North	Rye
Hays	Main	Harrison
Curry	North	Rye
Johnson	Alma	Brooklyn
Brooks	Senator	Brooklyn

**Input:**

1.  $\Pi$  NAME, CITY (CUSTOMER)

**Output:**

NAME	CITY
Jones	Harrison
Smith	Rye
Hays	Harrison
Curry	Rye
Johnson	Brooklyn
Brooks	Brooklyn

**3. Union Operation:**

- Suppose there are two tuples R and S. The union operation contains all the tuples that are either in R or S or both in R & S.
- It eliminates the duplicate tuples. It is denoted by  $\cup$ .

1. Notation:  $R \cup S$

A union operation must hold the following condition:

- R and S must have the attribute of the same number.
- Duplicate tuples are eliminated automatically.

Example:

**DEPOSITOR RELATION**

CUSTOMER_NAME	ACCOUNT_NO
Johnson	A-101
Smith	A-121
Mayes	A-321
Turner	A-176
Johnson	A-273
Jones	A-472
Lindsay	A-284

## BORROW RELATION

CUSTOMER_NAME	LOAN_NO
Jones	L-17
Smith	L-23
Hayes	L-15
Jackson	L-14
Curry	L-93
Smith	L-11
Williams	L-17

### Input:

1.  $\prod \text{CUSTOMER\_NAME (BORROW)} \cup \prod \text{CUSTOMER\_NAME (DEPOSITOR)}$

### Output:

CUSTOMER_NAME
Johnson
Smith
Hayes
Turner
Jones
Lindsay
Jackson
Curry
Williams
Mayes

### 4. Set Intersection:

- Suppose there are two tuples R and S. The set intersection operation contains all tuples that are in both R & S.
  - It is denoted by intersection  $\cap$ .
1. Notation:  $R \cap S$

**Example:** Using the above DEPOSITOR table and BORROW table

### Input:

1.  $\prod \text{CUSTOMER\_NAME (BORROW)} \cap \prod \text{CUSTOMER\_NAME (DEPOSITOR)}$

**Output:**

CUSTOMER_NAME
Smith
Jones

**5. Set Difference:**

- Suppose there are two tuples R and S. The set intersection operation contains all tuples that are in R but not in S.
- It is denoted by intersection minus (-).
- 1. Notation:  $R - S$

**Example:** Using the above DEPOSITOR table and BORROW table

**Input:**

1.  $\Pi \text{ CUSTOMER\_NAME (BORROW) - } \Pi \text{ CUSTOMER\_NAME (DEPOSITOR)}$

**Output:**

CUSTOMER_NAME
Jackson
Hayes
Willians
Curry

**6. Cartesian product**

- The Cartesian product is used to combine each row in one table with each row in the other table. It is also known as a cross product.
- It is denoted by X.
- 1. Notation:  $E \times D$

Example:

**EMPLOYEE**

EMP_ID	EMP_NAME	EMP_DEPT
1	Smith	A
2	Harry	C
3	John	B

## DEPARTMENT

DEPT_NO	DEPT_NAME
A	Marketing
B	Sales
C	Legal

**Input:**

1. EMPLOYEE X DEPARTMENT

**Output:**

EMP_ID	EMP_NAME	EMP_DEPT	DEPT_NO	DEPT_NAME
1	Smith	A	A	Marketing
1	Smith	A	B	Sales
1	Smith	A	C	Legal
2	Harry	C	A	Marketing
2	Harry	C	B	Sales
2	Harry	C	C	Legal
3	John	B	A	Marketing
3	John	B	B	Sales
3	John	B	C	Legal

## 7. Rename Operation:

The rename operation is used to rename the output relation. It is denoted by **rho** ( $\rho$ ).

**Example:** We can use the rename operator to rename STUDENT relation to STUDENT1.

1.  $\rho(\text{STUDENT1}, \text{STUDENT})$

## Tuple Relational Calculus (TRC) in DBMS

**Tuple Relational Calculus (TRC)** is a non-procedural query language used in relational database management systems (RDBMS) to retrieve data from tables. TRC is based on the concept of tuples, which are ordered sets of attribute values that represent a single row or record in a database table.

TRC is a declarative language, meaning that it specifies what data is required from the database, rather than how to retrieve it. TRC queries are expressed as logical formulas that describe the desired tuples.

**Syntax:** The basic syntax of TRC is as follows:

$$\{ t \mid P(t) \}$$

where  $t$  is a **tuple variable** and

$P(t)$  is a **logical formula** that describes the conditions that the tuples in the result must satisfy.

The **curly braces**  $\{ \}$  are used to indicate that the expression is a set of tuples.

For example, let's say we have a table called "Employees" with the following attributes:

Employee ID
Name
Salary
Department ID

To retrieve the names of all employees who earn more than \$50,000 per year, we can use the following TRC query:

$$\{ t \mid \text{Employees}(t) \wedge t.\text{Salary} > 50000 \}$$

In this query, the “Employees(t)” expression specifies that the tuple variable t represents a row in the “Employees” table. The “ $\wedge$ ” symbol is the logical AND operator, which is used to combine the condition “t.Salary > 50000” with the table selection. The result of this query will be a set of tuples, where each tuple contains the Name attribute of an employee who earns more than \$50,000 per year.

TRC can also be used to perform more complex queries, such as joins and nested queries, by using additional logical operators and expressions. While TRC is a powerful query language, it can be more difficult to write and understand than other SQL-based query languages, such as Structured Query Language (SQL). However, it is useful in certain applications, such as in the formal verification of database schemas and in academic research.

Tuple Relational Calculus is a non-procedural query language, unlike relational algebra. Tuple Calculus provides only the description of the query but it does not provide the methods to solve it. Thus, it explains what to do but not how to do it.

### Tuple Relational Query

In Tuple Calculus, a query is expressed as

$$\{t \mid P(t)\}$$

where t = resulting tuples,

P(t) = known as Predicate and these are the conditions that are used to fetch t. Thus, it generates a set of all tuples t, such that Predicate P(t) is true for t.

P(t) may have various conditions logically combined with OR ( $\vee$ ), AND ( $\wedge$ ), NOT ( $\neg$ ).

It also uses quantifiers:

$\exists t \in r (Q(t))$  = ”there exists” a tuple in t in relation r such that predicate Q(t) is true.

$\forall t \in r (Q(t))$  = Q(t) is true “for all” tuples in relation r.

### Tuple Relational Calculus Examples

#### Table Customer

Customer name	Street	City
Saurabh	A7	Patiala
Mehak	B6	Jalandhar
Sumiti	D9	Ludhiana
Ria	A5	Patiala

#### Table Branch

Branch name	Branch City
ABC	Patiala

Branch name	Branch City
DEF	Ludhiana
GHI	Jalandhar

Table Account

Account number	Branch name	Balance
1111	ABC	50000
1112	DEF	10000
1113	GHI	9000
1114	ABC	7000

Table Loan

Loan number	Branch name	Amount
L33	ABC	10000
L35	DEF	15000
L49	GHI	9000
L98	DEF	65000

Table Borrower

Customer name	Loan number
Saurabh	L33
Mehak	L49

Customer name	Loan number
Ria	L98

**Table Depositor**

Customer name	Account number
Saurabh	1111
Mehak	1113
Suniti	1114

**Example 1:** Find the loan number, branch, and amount of loans greater than or equal to 10000 amount.

$\{t \mid t \in \text{loan} \wedge t[\text{amount}] \geq 10000\}$

Resulting relation:

Loan number	Branch name	Amount
L33	ABC	10000
L35	DEF	15000
L98	DEF	65000

In the above query,  $t[\text{amount}]$  is known as a tuple variable.

**Example 2:** Find the loan number for each loan of an amount greater or equal to 10000.

$\{t \mid \exists s \in \text{loan}(t[\text{loan number}] = s[\text{loan number}]$

$\wedge s[\text{amount}] \geq 10000)\}$

Resulting relation:

Loan number
L33
L35



Loan number
L98

**Example 3:** Find the names of all customers who have a loan and an account at the bank.

$$\{t \mid \exists s \in \text{borrower}(t[\text{customer-name}] = s[\text{customer-name}]) \\ \wedge \exists u \in \text{depositor}(t[\text{customer-name}] = u[\text{customer-name}])\}$$

Resulting relation:

Customer name
Saurabh
Mehak

**Example 4:** Find the names of all customers having a loan at the “ABC” branch.

$$\{t \mid \exists s \in \text{borrower}(t[\text{customer-name}] = s[\text{customer-name}] \\ \wedge \exists u \in \text{loan}(u[\text{branch-name}] = \text{“ABC”} \wedge u[\text{loan-number}] = s[\text{loan-number}]))\}$$

Resulting relation:

Customer name
Saurabh

## Domain Relational Calculus in DBMS

**Domain Relational Calculus** is a non-procedural query language equivalent in power to Tuple Relational Calculus. Domain Relational Calculus provides only the description of the query but it does not provide the methods to solve it. In Domain Relational Calculus, a query is expressed as,

$$\{ \langle x_1, x_2, x_3, \dots, x_n \rangle \mid P(x_1, x_2, x_3, \dots, x_n) \}$$

where,  $\langle x_1, x_2, x_3, \dots, x_n \rangle$  represents resulting domains variables and

$P(x_1, x_2, x_3, \dots, x_n)$  represents the condition or formula equivalent to the Predicate calculus.

### Predicate Calculus Formula:

1. Set of all comparison operators
2. Set of connectives like and, or, not
3. Set of quantifiers

**Example:****Table-1: Customer**

Customer name	Street	City
Debomit	Kadamtala	Alipurduar
Sayantana	Udaypur	Balurghat
Soumya	Nutanchati	Bankura
Ritu	Juhu	Mumbai

**Table-2: Loan**

Loan number	Branch name	Amount
L01	Main	200
L03	Main	150
L10	Sub	90
L08	Main	60

**Table-3: Borrower**

Customer name	Loan number
Ritu	L01
Debomit	L08
Soumya	L03

**Query-1:** Find the loan number, branch, amount of loans of greater than or equal to 100 amount.

$$\{ \langle l, b, a \rangle \mid \langle l, b, a \rangle \in \text{loan} \wedge (a \geq 100) \}$$

Resulting relation:

Loan number	Branch name	Amount
L01	Main	200
L03	Main	150

**Query-2:** Find the loan number for each loan of an amount greater or equal to 150.

$$\{ \langle l \rangle \mid \exists b, a (\langle l, b, a \rangle \in \text{loan} \wedge (a \geq 150)) \}$$

Resulting relation:

Loan number
L01
L03

**Query-3:** Find the names of all customers having a loan at the “Main” branch and find the loan amount .

$$\{ \langle c, a \rangle \mid \exists l (\langle c, l \rangle \in \text{borrower} \wedge \exists b (\langle l, b, a \rangle \in \text{loan} \wedge (b = \text{“Main”}))) \}$$

Resulting relation:

Customer Name	Amount
Ritu	200
Debomit	60
Soumya	150

**Note:**

The domain variables those will be in resulting relation must appear before | within  $<$  and  $>$  and all the domain variables must appear in which order they are in original relation or table.

**Extended Relational Algebra Operations:**

Extended operators are those operators which can be derived from basic operators. There are mainly three types of extended operators in Relational Algebra:

- **Join**
- **Intersection**
- **Divide**

The relations used to understand extended operators are STUDENT, STUDENT\_SPORTS, ALL\_SPORTS and EMPLOYEE which are shown in Table 1, Table 2, Table 3 and Table 4 respectively. **STUDENT**

ROLL_NO	NAME	ADDRESS	PHONE	AGE
1	RAM	DELHI	9455123451	18
2	RAMESH	GURGAON	9652431543	18
3	SUJIT	ROHTAK	9156253131	20
4	SURESH	DELHI	9156768971	18

**Table 1**

**STUDENT\_SPORTS**

ROLL_NO	SPORTS
1	Badminton
2	Cricket
2	Badminton
4	Badminton

**Table 2**

**ALL\_SPORTS**

SPORTS
--------

Badminton
Cricket

**Table 3**

**EMPLOYEE**

EMP_NO	NAME	ADDRESS	PHONE	AGE
1	RAM	DELHI	9455123451	18
5	NARESH	HISAR	9782918192	22
6	SWETA	RANCHI	9852617621	21
4	SURESH	DELHI	9156768971	18

**Table 4**

**Intersection ( $\cap$ ):** Intersection on two relations R1 and R2 can only be computed if R1 and R2 are **union compatible** (These two relation should have same number of attributes and corresponding attributes in two relations have same domain). Intersection operator when applied on two relations as  $R1 \cap R2$  will give a relation with tuples which are in R1 as well as R2. Syntax:

$$\text{Relation1} \cap \text{Relation2}$$

Example: Find a person who is student as well as employee- **STUDENT  $\cap$  EMPLOYEE**

In terms of basic operators (union and minus) :

$$\text{STUDENT} \cap \text{EMPLOYEE} = \text{STUDENT} + \text{EMPLOYEE} - (\text{STUDENT} \cup \text{EMPLOYEE})$$

**RESULT:**

ROLL_NO	NAME	ADDRESS	PHONE	AGE
1	RAM	DELHI	9455123451	18
4	SURESH	DELHI	9156768971	18

**Conditional Join( $\bowtie_c$ ):** Conditional Join is used when you want to join two or more relation based on some conditions. Example: Select students whose ROLL\_NO is greater than EMP\_NO of employees

$$\text{STUDENT} \bowtie_c \text{STUDENT.ROLL\_NO} > \text{EMPLOYEE.EMP\_NO} \text{EMPLOYEE}$$

In terms of basic operators (cross product and selection) :

$\sigma$  (STUDENT.ROLL\_NO>EMPLOYEE.EMP\_NO)(STUDENT $\times$ EMPLOYEE)

**RESULT:**

ROLL_NO	NAME	ADDRESS	PHONE	AGE	EMP_NO	NAME	ADDRESS	PHONE	AGE
2	RAMESH	GURGAON	9652431543	18	1	RAM	DELHI	9455123451	18
3	SUJIT	ROHTAK	9156253131	20	1	RAM	DELHI	9455123451	18
4	SURESH	DELHI	9156768971	18	1	RAM	DELHI	9455123451	18

**Equijoin( $\bowtie$ ):** Equijoin is a **special case of conditional join** where only equality condition holds between a pair of attributes. As values of two attributes will be equal in result of equijoin, only one attribute will be appeared in result. Example: Select students whose ROLL\_NO is equal to EMP\_NO of employees.

**STUDENT $\bowtie$ STUDENT.ROLL\_NO=EMPLOYEE.EMP\_NOEMPLOYEE**

In terms of basic operators (cross product, selection and projection) :

$\Pi$  (STUDENT.ROLL\_NO, STUDENT.NAME, STUDENT.ADDRESS, STUDENT.PHONE, STUDENT.AGE EMPLOYEE.NAME, EMPLOYEE.ADDRESS, EMPLOYEE.PHONE, EMPLOYEE>AGE)( $\sigma$  (STUDENT.ROLL\_NO=EMPLOYEE.EMP\_NO) (STUDENT $\times$ EMPLOYEE))

**RESULT:**

ROLL_NO	NAME	ADDRESS	PHONE	AGE	NAME	ADDRESS	PHONE	AGE
1	RAM	DELHI	9455123451	18	RAM	DELHI	9455123451	18
4	SURESH	DELHI	9156768971	18	SURESH	DELHI	9156768971	18

**Natural Join( $\bowtie$ ):** It is a special case of equijoin in which equality condition hold on all attributes which have same name in relations R and S (relations on which join operation is applied). While applying natural join on two relations, there is no need to write equality condition explicitly. Natural Join will also return the similar attributes only once as their value will be same in resulting relation. Example: Select students whose ROLL\_NO is equal to ROLL\_NO of STUDENT\_SPORTS as:

**STUDENT $\bowtie$ STUDENT\_SPORTS**

In terms of basic operators (cross product, selection and projection) :

$\Pi$  (STUDENT.ROLL\_NO, STUDENT.NAME, STUDENT.ADDRESS, STUDENT.PHONE, STUDENT.AGE STUDENT\_SPORTS.SPORTS)( $\sigma$  (STUDENT.ROLL\_NO=STUDENT\_SPORTS.ROLL\_NO) (STUDENT $\times$ STUDENT\_SPORTS))

**RESULT:**

ROLL_NO	NAME	ADDRESS	PHONE	AGE	SPORTS
1	RAM	DELHI	9455123451	18	Badminton
2	RAMESH	GURGAON	9652431543	18	Cricket
2	RAMESH	GURGAON	9652431543	18	Badminton
4	SURESH	DELHI	9156768971	18	Badminton

Natural Join is by default inner join because the tuples which does not satisfy the conditions of join does not appear in result set. e.g.; The tuple having ROLL\_NO 3 in STUDENT does not match with any tuple in STUDENT\_SPORTS, so it has not been a part of result set.

**Left Outer Join( $\bowtie$ ):** When applying join on two relations R and S, some tuples of R or S does not appear in result set which does not satisfy the join conditions. But Left Outer Joins gives all tuples of R in the result set. The tuples of R which do not satisfy join condition will have values as NULL for attributes of S.  
Example: Select students whose ROLL\_NO is greater than EMP\_NO of employees and details of other students as well

**STUDENT $\bowtie$ STUDENT.ROLL\_NO>EMPLOYEE.EMP\_NOEMPLOYEE**

**RESULT**

ROLL_NO	NAME	ADDRESS	PHONE	AGE	EMP_NO	NAME	ADDRESS	PHONE	AGE
2	RAMESH	GURGAON	9652431543	18	1	RAM	DELHI	9455123451	18
3	SUJIT	ROHTAK	9156253131	20	1	RAM	DELHI	9455123451	18
4	SURESH	DELHI	9156768971	18	1	RAM	DELHI	9455123451	18
1	RAM	DELHI	9455123451	18	NULL	NULL	NULL	NULL	NULL

**Right Outer Join( $\bowtie$ ):** When applying join on two relations R and S, some tuples of R or S does not appear in result set which does not satisfy the join conditions. But Right Outer Joins gives all tuples of S in the result set. The tuples of S which do not satisfy join condition will have values as NULL for attributes of R.  
Example: Select students whose ROLL\_NO is greater than EMP\_NO of employees and details of other Employees as well

**STUDENT $\bowtie$ STUDENT.ROLL\_NO>EMPLOYEE.EMP\_NOEMPLOYEE**

**RESULT:**

ROLL_NO	NAME	ADDRESS	PHONE	AGE	EMP_NO	NAME	ADDRESS	PHONE	AGE
2	RAMESH	GURGAON	9652431543	18	1	RAM	DELHI	9455123451	18
3	SUJIT	ROHTAK	9156253131	20	1	RAM	DELHI	9455123451	18
4	SURESH	DELHI	9156768971	18	1	RAM	DELHI	9455123451	18
NULL	NULL	NULL	NULL	NULL	5	NARESH	HISAR	9782918192	22
NULL	NULL	NULL	NULL	NULL	6	SWETA	RANCHI	9852617621	21
NULL	NULL	NULL	NULL	NULL	4	SURESH	DELHI	9156768971	18

**Full Outer Join( $\bowtie$ ):** When applying join on two relations R and S, some tuples of R or S does not appear in result set which does not satisfy the join conditions. But Full Outer Joins gives all tuples of S and all tuples of R in the result set. The tuples of S which do not satisfy join condition will have values as NULL for attributes of R and vice versa. Example: Select students whose ROLL\_NO is greater than EMP\_NO of employees and details of other Employees as well and other Students as well

**STUDENT $\bowtie$ STUDENT.ROLL\_NO>EMPLOYEE.EMP\_NOEMPLOYEE**

**RESULT:**

ROLL_NO	NAME	ADDRESS	PHONE	AGE	EMP_NO	NAME	ADDRESS	PHONE	AGE
2	RAMESH	GURGAON	9652431543	18	1	RAM	DELHI	9455123451	18
3	SUJIT	ROHTAK	9156253131	20	1	RAM	DELHI	9455123451	18
4	SURESH	DELHI	9156768971	18	1	RAM	DELHI	9455123451	18
NULL	NULL	NULL	NULL	NULL	5	NARESH	HISAR	9782918192	22
NULL	NULL	NULL	NULL	NULL	6	SWETA	RANCHI	9852617621	21
NULL	NULL	NULL	NULL	NULL	4	SURESH	DELHI	9156768971	18
1	RAM	DELHI	9455123451	18	NULL	NULL	NULL	NULL	NULL

**Division Operator ( $\div$ ):** Division operator  $A \div B$  or  $A/B$  can be applied if and only if:

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

**A**

x	y
a	1
b	2
a	2
d	4

$\div$

**B**

y
1
2

The resultant of A/B is

**$A \div B$**

x
a

Division can be expressed in terms of **Cross Product** , **Set Difference** and **Projection**.

In the above example , for A/B , compute all x values that are not disqualified by some y in B.

x value is disqualified if attaching y value from B, we obtain xy tuple that is not in A.

**Disqualified x values:**  $\pi_x((\pi_x(A) \times B) - A)$

So  $A/B = \pi_x(A) - \text{all disqualified tuples}$

$$A/B = \pi_x(A) - \pi_x((\pi_x(A) \times B) - A)$$



In the above example , disqualified tuples are

b	2
d	4

So, the resultant is

x
a

### Advantages:

**Expressive Power:** Extended operators allow for more complex queries and transformations that cannot be easily expressed using basic relational algebra operations.

**Data Reduction:** Aggregation operators, such as SUM, AVG, COUNT, and MAX, can reduce the amount of data that needs to be processed and displayed.

**Data Transformation:** Extended operators can be used to transform data into different formats, such as pivoting rows into columns or vice versa.

**More Efficient:** Extended operators can be more efficient than expressing the same query in terms of basic relational algebra operations, since they can take advantage of specialized algorithms and optimizations.

### Disadvantages:

**Complexity:** Extended operators can be more difficult to understand and use than basic relational algebra operations. They require a deeper understanding of the underlying data and the operators themselves.

**Performance:** Some extended operators, such as outer joins, can be expensive in terms of performance, especially when dealing with large data sets.

**Non-standardized:** There is no universal set of extended operators, and different relational database management systems may implement them differently or not at all.

**Data Integrity:** Some extended operators, such as aggregate functions, can introduce potential problems with data integrity if not used properly. For example, using AVG on a column that contains null values can result in unexpected or incorrect results.

### Modification of the Database

The modification of a database has three commands, namely:

- DELETE
- INSERT
- UPDATE

Let us take the following table and understand each command with a few examples

## R5: FACULTY

---

FNo FName	DNo Qual	Salary
-----------	----------	--------

---

22 Alice	21 Ph.D.	35000
24 Ben	22 MTech	30000
25 Max	22 MTech	42000
27 Becca	23 MTech	28000
30 Bella	23 MTech	32000
33 Priya	24 Ph.D.	33000
35 Riya	24 Ph.D.	32000
37 Sia	25 MTech	26000
39 Tom	25 MTech	24000
40 Bella	25 MTech	32000

---

### Delete Command

This command helps us to remove rows from the table.

**Syntax :** DELETE      from r      where P

Example-1: Remove all the employees from Dept. no 24.

DELETE From FACULTY-1

Where DNo = 24 ;

Output : There will be 8 tuples left in Faculty-1.

Example-2 : Remove all the employees from ECE Dept.

DELETE From FACULTY-1

Where DNo = (Select DNo

From DEPT

Where DName = 'ECE' );

Output : There will be 8 tuples left in Faculty-1.

Example-3 : Remove all the employees whose salary is less than 30000.

Delete From FACULTY-1

Where Salary < 30000 ;

Output : There will be 7 tuples left in Faculty-1.

Example-4 : Remove all the employees whose salary is less than the average salary of all the employees.

Delete From FACULTY-1

Where Salary < (Select avg(Salary)

From FACULTY-1) ;

FNo	FName	DNo	Qual	Salary
22	Alice	21	Ph.D.	35000
25	Max	22	MTech	42000
30	Bella	23	MTech	32000
33	Priya	24	Ph.D.	33000
35	Riya	24	Ph.D.	32000
40	Bella	25	MTech	34000

Note : The average salary is : 31,600

### **Insert Command**

This command helps us to insert rows into the table.

**Syntax :** INSERT into relation-name values (.....)

Example-1 : Add a tuple to FACULTY-1.

Insert into FACULTY-1 values (532, 'XX', 28, 'MTech', 25000) ;

Example-2 : Add a tuple to STUD relation

Insert into STUD values ( 130, 'def', 24) ;

Example-3 :select tuples from FACULTY-1 and create a relation FACULTY-2 with tuples belonging to DNo 25.

Let's assume table structure for FACULTY-2 is already there.

```
Insert into FACULTY-2
(Select FNo, FName, DNo, Qual, Salary
From FACULTY-1
Where DNo = 25) ;

R5 : FACULTY-2
```

---

FNo	FName	DNo	Qual	Salary
37	Sia	25	MTech	26000
39	Tom	25	MTech	24000
40	Bella	25	MTech	32000

---

Note : One can create a new relation by considering data from one or more relations using the insert command.

### Update Command

This command helps us to modify columns in table rows.

```
Syntax :  update    <relation name>
          set        <assignment>
          where      <condition>
```

Example-1 :Increase the salary of the employees who are drawing more than 35000 by 5 %.

Update FACULTY-1

```
Set Salary = Salary * 1.05
```

```
Where Salary >35000 ;
```

Example-2 :Increase the salary of the employees who are drawing less than average of all the employees, by 10 %.

Update FACULTY-1

Set Salary = Salary \* 1.1

Where Salary <

(select avg(salary)

From FACULTY-1 );

Example-3 :Increase the salary of the employees who are drawing less than 30000, by 8 % and for others by 6 %.

Update FACULTY-1

Set Salary = Salary \* 1.06

Where Salary >30000 ;

Update FACULTY-1

Set Salary = Salary \* 1.08

Where Salary <30000 ;

## Views

Views are like virtual tables, which also contain rows and columns like that of a normal table in a database. A view can be created using more than one table in a database. A view can either have a specific row with a specification or may also contain all the rows.

Syntax : Create view V as <query expression> 'View' is a logical relation.

**Example-1 :** Create a view 'student' containing SNo, SName.

Create view student as

(Select SNo, SName

From STUD) ;

Output :

student (VIEW)

SNo	SName
121	xyz
123	pqr
126	mnp
128	abc
130	jkl

**Example-2 :** Create a view containing SNo, SName, CNo, CName.

```
Create view stud_course  
  
(Select SNo, SName, CNo, CName  
  
From STUD, COURSE  
  
Where STUD.CNo = COURSE.CNo ;
```

Output : stud\_course (VIEW)

SNo	SName	CNo	CName
121	xyz	61	DBMS
126	mnp	65	CO
128	abc	67	OS

**Example-3:** Find SName and the CName the student is studying using stud\_course view

```
Select SName, CName From stud_course ;
```

Output :

SName	CName
xyz	DBMS
mnp	CO
abc	OS

**Example-4 :** Create a view containing SName and DName in which he is studying.

Consider following relations :

R1 : STUD			R2 : DEPT	
SNo	SName	DNo	DNo	DName
121	xyz	21	21	CSE
123	pqr	21	22	IT
126	mnp	22	23	ECE
128	abc	22	24	ME
130	jkl	23	25	EEE

Create a view stud\_dept as

(Select SName, DName

From STUD, DEPT

Where STUD.DNo = DEPT.DNo ;

Output : stud\_dept (VIEW)

SName	DName
xyz	CSE
pqr	CSE
mnp	IT
abc	IT
jkl	ECE

**Example-5 :** Find the list of students studying in ECE department

using stud\_dept view.

Select SName

From stud\_dept

Where DName = 'ECE' ;.

Output : SName

jkl

There are three stages of Views:

- **Materialized Views :** If a view is stored in the memory, then it is called a 'materialized view'.
- **Materialized View Maintenance :** The process of keeping materialized view up-to-date is called 'materialized view maintenance'.
- **Update of a View :** Like a relation, one can update a view also.

**Example-6 :** Insert a tuple into the view 'stud\_dept'.

Insert into stud\_dept values ('ghi', 'CIVIL')

**Example-7 :** After inserting the above tuple, display the student name studying in the CIVIL department using the view 'stud\_dept'.

Select SName  
From stud\_dept  
Where DName = 'CIVIL' ;

Output : SName

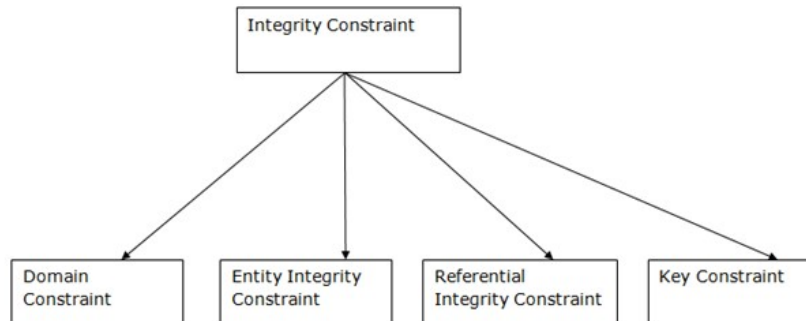
ghi

## UNIT IV

### INTEGRITY CONSTRAINTS:

- Integrity constraints are a set of rules. It is used to maintain the quality of information.
- Integrity constraints ensure that the data insertion, updating, and other processes have to be performed in such a way that data integrity is not affected.
- Thus, integrity constraint is used to guard against accidental damage to the database.

#### Types of Integrity Constraint



#### 1. Domain constraints

- Domain constraints can be defined as the definition of a valid set of values for an attribute.
- The data type of domain includes string, character, integer, time, date, currency, etc. The value of the attribute must be available in the corresponding domain.

#### Example:

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

Not allowed. Because AGE is an integer attribute

#### 2. Entity integrity constraints

- The entity integrity constraint states that primary key value can't be null.
- This is because the primary key value is used to identify individual rows in relation and if the primary key has a null value, then we can't identify those rows.
- A table can contain a null value other than the primary key field.



Example:

### EMPLOYEE

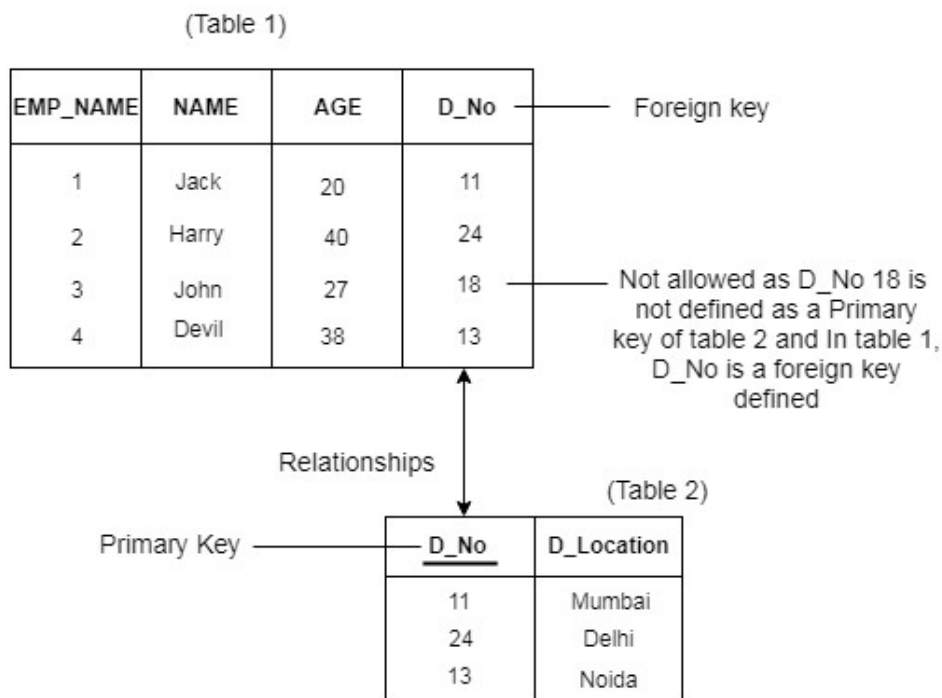
EMP_ID	EMP_NAME	SALARY
123	Jack	30000
142	Harry	60000
164	John	20000
	Jackson	27000

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

### 3. Referential Integrity Constraints

- A referential integrity constraint is specified between two tables.
- In the Referential integrity constraints, if a foreign key in Table 1 refers to the Primary Key of Table 2, then every value of the Foreign Key in Table 1 must be null or be available in Table 2.

Example:



### 4. Key constraints

- Keys are the entity set that is used to identify an entity within its entity set uniquely.
- An entity set can have multiple keys, but out of which one key will be the primary key. A primary key can contain a unique and null value in the relational table.

### Example:

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

Not allowed. Because all row must be unique

### Query By Example (QBE)

In normal queries we fire on the database they should be correct and in a well-defined structure which means they should follow a proper syntax if the syntax or query is wrong definitely we will get an error and due to that our application or calculation definitely going to stop. So to overcome this problem QBE was introduced. QBE stands for **Query By Example** and it was developed in 1970 by Moshe Zloof at IBM.

It is a graphical query language where we get a user interface and then we fill some required fields to get our proper result.

In **SQL** we will get an error if the query is not correct but in the case of QBE if the query is wrong either we get a wrong answer or the query will not be going to execute but we will never get any error.

#### Note:-

In QBE we don't write complete queries like SQL or other database languages it comes with some blank so we need to just fill that blanks and we will get our required result.

#### Example

Consider the example where a table 'SAC' is present in the database with Name, Phone\_Number, and Branch fields. And we want to get the name of the SAC-Representative name who belongs to the MCA Branch. If we write this query in SQL we have to write it like

```
SELECT NAME
```

```
FROM SAC
```

```
WHERE BRANCH = 'MCA'
```

And definitely, we will get our correct result. But in the case of QBE, it may be done as like there is a field present and we just need to fill it with "MCA" and then click on the SEARCH button we will get our required result.

#### Points about QBE:

- Supported by most of the database programs.
- It is a Graphical Query Language.
- Created in parallel to SQL development

## QUEL:

- QUEL stands for Query Language.
- It is a data definition and data manipulation for INGRES.
- INGRES stands for Interactive Graphics and Retrieval System.
- INGRES is a relational database management system developed by Michael Stonebraker.
- QUEL does not support relational algebraic operations such as intersection, minus or union.
- It is based on tuple calculus and does not support nested sub queries.

### Data Definition in QUEL

Following are the data definition statements used in QUEL,

1. CREATE
2. RANGE
3. INDEX
4. DESTROY
5. MODIFY

Statements	Description	Syntax
CREATE	It is used to create tables or relations.	CREATE <table-name> <list-of-column-name>
RANGE	It allows to declare a range variable and restricts to assume the values that are rows from the specified table. Row variables are called Range variables in QUEL.	RANGE OF <variable-name> IS <relation-name>
INDEX	It is used to specify the name of the secondary index to be built and the columns in the table on which the index is to be created.	INDEX ON <table-name> IS <index-name> (column-name [, column-name, ...])
DESTROY	It is used to eliminate a table, index or view.	DESTROY name [, name, ...]
MODIFY	It is used to modify the structure of a table. The storage structure supported by INGRES are B-tree, hash, ISAM and heap. The storage structure will be modified from the current one to the one specified in the statement.	MODIFY <table-name> TO <storage-structure> ON <column-name>

### Usage

QUEL statements are always defined by *tuple variables*, which can be used to limit queries or return result sets. Consider this example, taken from one of the first original Ingres papers:<sup>[2]</sup>

*Example 1.1.* Compute salary divided by age-18 for employee Jones.

range **of** E **is** EMPLOYEE

retrieve **into** W

(COMP = E.Salary / (E.Age - 18))

**where** E.Name = "Jones"

Here E is a tuple variable which ranges over the EMPLOYEE relation, and all tuples in that relation are found which satisfy the qualification E.Name = "Jones." The result of the query is a new relation W, which has a single domain COMP that has been calculated for each qualifying tuple.

An equivalent SQL statement is:

```
create table w as
select (e.salary / (e.age - 18)) as comp
from employee as e
where e.name = 'Jones'
```

Here is a sample of a simple session that creates a table, inserts a row into it, and then retrieves and modifies the data inside it and finally deletes the row that was added (assuming that name is a unique field).

QUEL	SQL
<b>create</b> student(name = c10, age = i4, sex = c1, <b>state</b> = c2)	<b>create table</b> student(name char(10), age int, sex char(1), <b>state</b> char(2));
range <b>of</b> s <b>is</b> student	
append <b>to</b> s (name = "philip", age = 17, sex = "m", <b>state</b> = "FL")	<b>insert into</b> student (name, age, sex, <b>state</b> ) <b>values</b> ('philip', 17, 'm', 'FL');
retrieve (s. <b>all</b> ) <b>where</b> s. <b>state</b> = "FL"	<b>select * from</b> student <b>where</b> <b>state</b> = 'FL';
<b>replace</b> s (age=s.age+1)	<b>update</b> student <b>set</b> age=age+1;
retrieve (s. <b>all</b> )	<b>select * from</b> student;
<b>delete</b> s <b>where</b> s.name="philip"	<b>delete from</b> student <b>where</b> name='philip';

Another feature of QUEL was a built-in system for moving records en-masse into and out of the system. Consider this command:

```
copy student(name=c0, comma=d1, age=c0, comma=d1, sex=c0, comma=d1, address=c0, nl=d1)
into "/student.txt"
```

which creates a comma-delimited file of all the records in the student table. The d1 indicates a delimiter, as opposed to a data type. Changing the into to a from reverses the process. Similar commands are available in many SQL systems, but usually as external tools, as opposed to being internal to the SQL language. This makes them unavailable to stored procedures.

QUEL has an extremely powerful aggregation capability. Aggregates can be nested, and different aggregates can have independent by-lists and/or restriction clauses. For example:

```
retrieve (a=count(y.i by y.d where y.str = "ii*" or y.str = "foo"), b=max(count(y.i by y.d)))
```

This example illustrates one of the arguably less desirable quirks of QUEL, namely that all string comparisons are potentially pattern matches. `y.str = "ii"` matches all `y.str` values starting with `ii`. In contrast, SQL uses `=` only for exact matches, while `like` is used when pattern matching is required.

## **DATALOG:**

Datalog is a programming language used in deductive database work. It is part of another language called Prolog and incorporates basic logic principles for data integration, database queries, etc. Datalog is used by many open-source systems and other database systems.

Database programmers like Datalog for its simplicity. As a simple declarative logic-based language, Datalog relies on a conventional clause format. In a declarative language, the user enters the items that he/she wants to find and then the system takes over, finding values that comply with the user's request.

Like other types of query systems, a Datalog query involves setting up a command-based premise: for example, many simpler Datalog queries consist of an object and a set of modifiers or constraints in parentheses. The simple syntax allows administrators to quickly learn how to get the results they need from the database. However, as with other systems, Datalog users have to deal with the emergence of raw or unstructured data sets in a database technology. In other words, whereas databases of the past tended to have strict "table" data formats, today's databases may have much more abstracted information that has to be queried and handled in a different way

In Datalog input, whitespace characters are ignored except when they separate adjacent tokens or when they occur in strings. Comments are also considered to be whitespace. The character `%` introduces a comment, which extends to the next line break. Comments do not occur inside strings.

A variable is a sequence of Unicode "Uppercase" and "Lowercase" letters, digits, and the underscore character. A variable must begin with a Unicode "Uppercase" letter.

An identifier is a sequence of printing characters that does not contain any of the following characters: `(, ` , ' , ) , = , : , . , ~ , ? , " , % ,` and space. An identifier must not begin with a Latin capital letter. Note that the characters that start punctuation are forbidden in identifiers, but the hyphen character is allowed.

A string is a sequence of characters enclosed in double quotes. Characters other than double quote, newline, and backslash may be directly included in a string. The remaining characters may be specified using escape characters, `\`, `\\`, and `\\` respectively.

A literal is a predicate symbol followed by an optional parenthesized list of comma separated terms, or it is an [external query](#) as described below. A predicate symbol is either an identifier or a string. A term is either a variable or a constant. A constant is an identifier, string, integer, or boolean, where booleans are written the same as the identifiers `true` and `false`, and integers are written the same as identifiers `0` or those with a nonempty sequence of digits, no leading zero, and optionally prefixed with `-`. As a special case, two terms separated by `= (!=)` is a literal for the equality (inequality) predicate. The following are literals:

`parent(john, douglas)`

`zero-arity-literal`

`"=(3,3)`

`""(-0-0-0,&&&,&&&,"\\00")`

42

A clause is a head literal followed by an optional body. A body is a comma separated list of literals. A clause without a body is called a *fact*, and a rule when it has one. The punctuation `:-` separates the head of a rule from its body. A clause is safe if every variable in its head occurs in some literal in its body. The following are safe clauses:

parent(john, douglas)

ancestor(A, B) :-

parent(A, B)

ancestor(A, B) :-

parent(A, C),

ancestor(C, B)

A program is a sequence of zero or more statements. A statement is an assertion, a retraction, a query, or a requirement. An assertion is a clause followed by a ., and it adds the clause to the database if it is safe. A retraction is a clause followed by ~, and it removes the clause from the database. A query is a literal followed by a ?. A requirement is a (, then an identifier, then ), then ., and it imports functions that can be called as external queries.

A *external query* is a variable, then :-, then an identifier, then a parenthesized list of comma separated terms. Beware that an external query can break Datalog's termination guarantee.

The following BNF describes the syntax of Datalog.

```
⟨program⟩      ::= ⟨statement⟩*
⟨statement⟩    ::= ⟨assertion⟩
                |  ⟨retraction⟩
                |  ⟨query⟩
                |  ⟨requirement⟩
⟨assertion⟩    ::= ⟨clause⟩ .
⟨retraction⟩   ::= ⟨clause⟩ ~
⟨query⟩        ::= ⟨literal⟩ ?
⟨requirement⟩  ::= ( ⟨IDENTIFIER⟩ ) .
⟨clause⟩       ::= ⟨literal⟩ :- ⟨body⟩
                |  ⟨literal⟩
⟨body⟩         ::= ⟨literal⟩ , ⟨body⟩
                |  ⟨literal⟩
⟨literal⟩      ::= ⟨predicate-sym⟩ ( )
                |  ⟨predicate-sym⟩ ( ⟨terms⟩ )
                |  ⟨predicate-sym⟩
                |  ⟨term⟩ = ⟨term⟩
                |  ⟨term⟩ != ⟨term⟩
                |  ⟨VARIABLE⟩ :- ⟨external-sym⟩ ( ⟨terms⟩ )
⟨predicate-sym⟩ ::= ⟨IDENTIFIER⟩
                |  ⟨STRING⟩
⟨terms⟩        ::= ⟨term⟩
```

		$\langle term \rangle$ , $\langle terms \rangle$
$\langle term \rangle$	::=	$\langle VARIABLE \rangle$
		$\langle constant \rangle$
$\langle constant \rangle$	::=	$\langle IDENTIFIER \rangle$
		$\langle STRING \rangle$
		$\langle INTEGER \rangle$
		true   false

The effect of running a Datalog program is to modify the database as directed by its statements, and then to return the literals designated by the query. The modified database is provided as theory.

The following is a program:

```
#lang datalog
edge(a, b). edge(b, c). edge(c, d). edge(d, a).
path(X, Y) :- edge(X, Y).
path(X, Y) :- edge(X, Z), path(Z, Y).
path(X, Y)?
```

Here is a program that uses  $\pm$  and  $-$  from [racket/base](#) as external queries:

```
#lang datalog
(racket/base).

fib(0, 0).
fib(1, 1).

fib(N, F) :- N != 1,
             N != 0,
             N1 :- -(N, 1),
             N2 :- -(N, 2),
             fib(N1, F1),
             fib(N2, F2),
             F :- +(F1, F2).

fib(30, F)?
```

The Datalog REPL accepts new statements that are executed as if they were in the original program text.

### **Domain constraints:**

Domain Constraints are user-defined columns that help the user to enter the value according to the data type. And if it encounters a wrong input it gives the message to the user that the column is not fulfilled properly. Or

in other words, it is an attribute that specifies all the possible values that the attribute can hold like integer, character, date, time, string, etc. It defines the domain or the set of values for an attribute and ensures that the value taken by the attribute must be an atomic value(Can't be divided) from its domain.

Domain Constraint = data type(integer / character/date / time / string / etc.) +

Constraints(NOT NULL / UNIQUE / PRIMARY KEY /  
FOREIGN KEY / CHECK / DEFAULT)

### **Type of domain constraints:**

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

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

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

Create table employee

```
(employee_id varchar(30),  
employee_name varchar(30) not null,  
salary NUMBER);
```

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

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

Create table employee

```
(employee_id varchar(30) not null check(employee_id > 0),  
employee_name varchar(30),  
salary NUMBER);
```

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

**Note:** In DBMS a table is a combination of rows and columns in which we have some unique attribute names associated with it. And basically, a domain is a unique set of values present in a table. Let's take an example, suppose we have a table student which consists of 3 attributes as NAME, ROLL NO, and MARKS. Now ROLL NO attributes can have only numbers associated with them and they won't contain any alphabet. So we can say that it contains the domain of integer only and it can be only a positive number greater than 0.

### **Example 1:**

Creating a table "student" with the "ROLL" field having a value greater than 0.

**Domain:**



```
create domain roll_no int
constraint roll_test
check(value > 0);
```

Table:

```
create table student (
ROLL roll_no PRIMARY KEY,
S_NAME varchar(30),
MARKS NUMBER
);
```

The above example will only accept the roll no. which is greater than 0.

### Example 2:

Creating a table “Employee” with the “AGE” field having a value greater than 18.

Domain:

```
create domain e_age int
constraint age_test
check(value > 18);
```

Table:

```
create table Employee (
AGE e_age,
E_NAME varchar(30),
E_ID NUMBER PRIMARY KEY
);
```

The above example will only accept the Employee with an age greater than 18.

### REFERENTIAL INTEGRITY:

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

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

**The syntax of the Master Table or Referenced table is:**

1. 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:*

1. *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.*

*Foreign Key Constraint OR Referential Integrity constraint.*

*There are two referential integrity constraint:*

**Insert Constraint:** *Value cannot be inserted in CHILD Table if the value is not lying in MASTER Table*

**Delete Constraint:** *Value cannot be deleted from MASTER Table if the value is lying in CHILD Table*

*Suppose you wanted to insert Roll = 05 with other values of columns in SUBJECT Table, then you will immediately see an error "**Foreign key Constraint Violated**" i.e. on running an insertion command as:*

**Insert into SUBJECT values(5, 786, OS); will not be entertained by SQL due to Insertion Constraint** ( As you cannot insert value in a child table if the value is not lying in the master table, since Roll = 5 is not present in the master table, hence it will not be allowed to enter Roll = 5 in child table )

*Similarly, if you want to delete Roll = 4 from STUDENT Table, then you will immediately see an error "**Foreign key Constraint Violated**" i.e. on running a deletion command as:*

**Delete from STUDENT where Roll = 4; will not be entertained by SQL due to Deletion Constraint.** ( As you cannot delete the value from the master table if the value is lying in the child table, since Roll = 5 is present in the child table, hence it will not be allowed to delete Roll = 5 from the master table, lets if somehow we managed to delete Roll = 5, then Roll = 5 will be available in child table which will ultimately violate insertion constraint. )

**ON DELETE CASCADE.**

As per deletion constraint: Value cannot be deleted from the MASTER Table if the value is lying in CHILD Table. The next question comes can we delete the value from the master table if the value is lying in the child table without violating the deletion constraint? i.e. The moment we delete the value from the master table the value corresponding to it should also get deleted from the child table.

The answer to the above question is YES, we can delete the value from the master table if the value is lying in the child table without violating the deletion constraint, we have to do slight modification while creating the child table, i.e. by adding **on delete cascade**.

### TABLE SYNTAX

1. CREATE TABLE Subject (Roll **int** references Student on delete cascade, SubCode **int**, SubName varchar(10) );

In the above syntax, just after references keyword( used for creating foreign key), we have added on delete cascade, by adding such now, we can delete the value from the master table if the value is lying in the child table without violating deletion constraint. Now if you wanted to delete Roll = 5 from the master table even though Roll = 5 is lying in the child table, it is possible because the moment you give the command to delete Roll = 5 from the master table, the row having Roll = 5 from child table will also get deleted.

### STUDENT TABLE

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

### SUBJECT TABLE

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

The above two tables STUDENT and SUBJECT having four values each are shown, now suppose you are looking to delete Roll = 4 from STUDENT(Master) Table by writing a SQL command: **delete from STUDENT where Roll = 4;**

The moment SQL execute the above command the row having Roll = 4 from SUBJECT( Child ) Table will also get deleted, The resultant **STUDENT and SUBJECT** table will look like:

### STUDENT TABLE

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

### SUBJECT TABLE

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

From the above two tables *STUDENT* and *SUBJECT*, you can see that in both the table Roll = 4 gets deleted at one go without violating deletion constraint.

Sometimes a very important question is asked in interviews that: Can Foreign Key have NULL values?

The answer to the above question is YES, it may have NULL values, whereas the Primary key cannot be NULL at any cost. To understand the above question practically let's understand below the concept of delete null.

### ON DELETE NULL.

As per deletion constraint: Value cannot be deleted from the MASTER Table if the value is lying in CHILD Table. The next question comes can we delete the value from the master table if the value is lying in the child table without violating the deletion constraint? i.e. The moment we delete the value from the master table the value corresponding to it should also get deleted from the child table or can be replaced with the NULL value.

The answer to the above question is YES, we can delete the value from the master table if the value is lying in child table without violating deletion constraint by inserting NULL in the foreign key, we have to do slight modification while creating child table, i.e. by adding **on delete null**.

### TABLE SYNTAX:

1. CREATE TABLE Subject (Roll **int** references Student on delete **null**, SubCode **int**, SubName varchar(10) );

In the above syntax, just after references keyword( used for creating foreign key), we have added on delete null, by adding such now, we can delete the value from the master table if the value is lying in the child table without violating deletion constraint. Now if you wanted to delete Roll = 4 from the master table even though Roll =4 is lying in the child table, it is possible because the moment you give the command to delete Roll = 4 from the master table, the row having Roll = 4 from child table will get replaced by a NULL value.

**STUDENT TABLE**

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

**SUBJECT TABLE**

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

The above two tables *STUDENT* and *SUBJECT* having four values each are shown, now suppose you are looking to delete Roll = 4 from *STUDENT*(Master ) Table by writing a SQL command: ***delete from STUDENT where Roll = 4;***

The moment SQL execute the above command the row having Roll = 4 from *SUBJECT*( Child ) Table will get replaced by a NULL value, The resultant ***STUDENT and SUBJECT*** table will look like:

**STUDENT TABLE**

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

**SUBJECT TABLE**

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

From the above two tables *STUDENT* and *SUBJECT*, you can see that in table *STUDENT* Roll = 4 get deleted while the value of Roll = 4 in the *SUBJECT* table is replaced by NULL. This proves that the Foreign key can

have null values. If in the case in SUBJECT Table, column Roll is Primary Key along with Foreign Key then in that case we could not make a foreign key to have NULL values.

## Assertions

Assertions are different from check constraints in the way that check constraints are rules that relate to one single row only. Assertions, on the other hand, can involve any number of other tables, or any number of other rows in the same table. Assertions also check a condition, which must return a Boolean value. We can take an illustrative example.

Let us imagine that we have the following table, which contains employees in a company — we then also store an attribute containing their salary.

ID	Name	Age	DepNo	Salary
0	Hannah	18	10	1000\$
1	Gavin	45	2	25.000\$
2	Bobby	70	21	700\$

We then want to make an assertion that there is no employee in our database, which is paid more than 30.000\$ or less than 500\$. It would then look like:

Create Assertion SalaryAssertion check

(Not exists

(select ID

From People p

Where p.salary > 30.000 OR p.salary < 500

);

Then it makes sure that we never have someone who receives a salary outside these bounds.

## Triggers

We can now consider the concept of triggers. Triggers are sometimes called **event-condition-action rules**. This is since triggers are only active in certain scenarios. We can describe the process of a trigger:

- Triggers are only awakened when certain events occur. This is usually the event of ‘delete’, ‘insert’, or ‘update’.
- When the trigger is awakened, the trigger tests a condition. If the condition does not hold, then nothing else associated with the trigger happens in response to the given event. On the other hand, if the trigger is satisfied then a pre-defined action is performed by the trigger.

We can take an example of a trigger. Let us imagine that we have the following database:

ID	Name	Age	DepNo	Salary
0	Hannah	18	10	1000\$
1	Gavin	45	2	25.000\$
2	Bobby	70	21	700\$

We already have a check constraint, where we check the salary. It could, however, also be written as a trigger. We could write a trigger that is awakened every time we want to insert something into our database, which then checks a condition upon the salary:

```

CREATE FUNCTION SalaryCheck()
RETURNS TRIGGER
AS $$ BEGIN
    IF (new.salary < 500 OR new.salary > 30.000) THEN
        RAISE EXCEPTION 'SalaryCheck: Salary must be within range'
        USING ERRORCODE = '3000';
    END IF;
    RETURN NEW;
END; $$ LANGUAGE plpgsql;

CREATE TRIGGER SalaryCheck
BEFORE INSERT OR UPDATE
ON People
FOR EACH ROW EXECUTE PROCEDURE Salarycheck();

```

We can now see that our trigger is awakened every time we want to insert something into our database and it is executed before it is inserted — this is since we need to check the condition before we are willing to insert. It is also possible to create triggers that are executed after, or even before and after.

### **Functional Dependency**

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.

$$1. X \rightarrow Y$$

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.

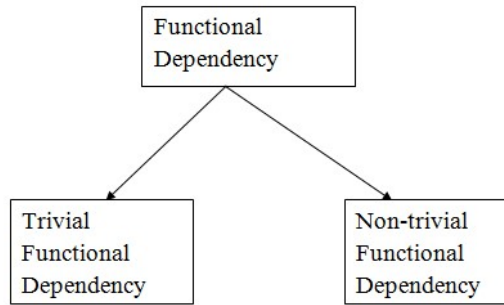
Functional dependency can be written as:

$$1. \text{Emp\_Id} \rightarrow \text{Emp\_Name}$$

We can say that Emp\_Name is functionally dependent on Emp\_Id.

Types of Functional dependency





### 1. Trivial functional dependency

- $A \rightarrow B$  has trivial functional dependency if  $B$  is a subset of  $A$ .
- The following dependencies are also trivial like:  $A \rightarrow A$ ,  $B \rightarrow B$

#### Example:

1. Consider a table with two columns `Employee_Id` and `Employee_Name`.
2.  $\{\text{Employee\_id}, \text{Employee\_Name}\} \rightarrow \text{Employee\_Id}$  is a trivial functional dependency as
3. `Employee_Id` is a subset of  $\{\text{Employee\_Id}, \text{Employee\_Name}\}$ .
4. Also,  $\text{Employee\_Id} \rightarrow \text{Employee\_Id}$  and  $\text{Employee\_Name} \rightarrow \text{Employee\_Name}$  are trivial dependencies too.

### 2. Non-trivial functional dependency

- $A \rightarrow B$  has a non-trivial functional dependency if  $B$  is not a subset of  $A$ .
- When  $A \cap B$  is NULL, then  $A \rightarrow B$  is called as complete non-trivial.

#### Example:

1.  $\text{ID} \rightarrow \text{Name}$ ,
2.  $\text{Name} \rightarrow \text{DOB}$



## UNIT V

### PL/SQL

#### **INTRODUCTION:**

The PL/SQL programming language was developed by Oracle Corporation in the late 1980s as procedural extension language for SQL and the Oracle relational database. Following are certain notable facts about PL/SQL –

- PL/SQL is a completely portable, high-performance transaction-processing language.
- PL/SQL provides a built-in, interpreted and OS independent programming environment.
- PL/SQL can also directly be called from the command-line SQL\*Plus interface.
- Direct call can also be made from external programming language calls to database.
- PL/SQL's general syntax is based on that of ADA and Pascal programming language.
- Apart from Oracle, PL/SQL is available in TimesTen in-memory database and IBM DB2.

#### Features of PL/SQL

PL/SQL has the following features –

- PL/SQL is tightly integrated with SQL.
- It offers extensive error checking.
- It offers numerous data types.
- It offers a variety of programming structures.
- It supports structured programming through functions and procedures.
- It supports object-oriented programming.
- It supports the development of web applications and server pages.

#### Advantages of PL/SQL

PL/SQL has the following advantages –

- SQL is the standard database language and PL/SQL is strongly integrated with SQL. PL/SQL supports both static and dynamic SQL. Static SQL supports DML operations and transaction control from PL/SQL block. In Dynamic SQL, SQL allows embedding DDL statements in PL/SQL blocks.
- PL/SQL allows sending an entire block of statements to the database at one time. This reduces network traffic and provides high performance for the applications.
- PL/SQL gives high productivity to programmers as it can query, transform, and update data in a database.
- PL/SQL saves time on design and debugging by strong features, such as exception handling, encapsulation, data hiding, and object-oriented data types.
- Applications written in PL/SQL are fully portable.
- PL/SQL provides high security level.
- PL/SQL provides access to predefined SQL packages.
- PL/SQL provides support for Object-Oriented Programming.
- PL/SQL provides support for developing Web Applications and Server Pages.

the Basic Syntax of PL/SQL which is a **block-structured** language; this means that the PL/SQL programs are divided and written in logical blocks of code. Each block consists of three sub-parts –

S.No	Sections & Description
1	<b>Declarations</b> This section starts with the keyword <b>DECLARE</b> . It is an optional section and defines all variables, cursors, subprograms, and other elements to be used in the program.
2	<b>Executable Commands</b> This section is enclosed between the keywords <b>BEGIN</b> and <b>END</b> and it is a mandatory section. It consists of the executable PL/SQL statements of the program. It should have at least one executable line of code, which may be just a <b>NULL command</b> to indicate that nothing should be executed.
3	<b>Exception Handling</b> This section starts with the keyword <b>EXCEPTION</b> . This optional section contains <b>exception(s)</b> that handle errors in the program.

Every PL/SQL statement ends with a semicolon (;). PL/SQL blocks can be nested within other PL/SQL blocks using **BEGIN** and **END**. Following is the basic structure of a PL/SQL block –

```
DECLARE
  <declarations section>
BEGIN
  <executable command(s)>
EXCEPTION
  <exception handling>
END;
```

#### The 'Hello World' Example

```
DECLARE
  message varchar2(20):= 'Hello, World!';
BEGIN
  dbms_output.put_line(message);
END;
/
```

The **end;** line signals the end of the PL/SQL block. To run the code from the SQL command line, you may need to type / at the beginning of the first blank line after the last line of the code. When the above code is executed at the SQL prompt, it produces the following result –

Hello World

PL/SQL procedure successfully completed.

#### The PL/SQL Identifiers

PL/SQL identifiers are constants, variables, exceptions, procedures, cursors, and reserved words. The identifiers consist of a letter optionally followed by more letters, numerals, dollar signs, underscores, and number signs and should not exceed 30 characters.

By default, **identifiers are not case-sensitive**. So you can use **integer** or **INTEGER** to represent a numeric value. You cannot use a reserved keyword as an identifier.

## The PL/SQL Delimiters

A delimiter is a symbol with a special meaning. Following is the list of delimiters in PL/SQL –

Delimiter	Description
+, -, *, /	Addition, subtraction/negation, multiplication, division
%	Attribute indicator
'	Character string delimiter
.	Component selector
(,)	Expression or list delimiter
:	Host variable indicator
,	Item separator
"	Quoted identifier delimiter
=	Relational operator
@	Remote access indicator
;	Statement terminator
:=	Assignment operator
=>	Association operator
	Concatenation operator
**	Exponentiation operator
<<, >>	Label delimiter (begin and end)
/*, */	Multi-line comment delimiter (begin and end)
--	Single-line comment indicator
..	Range operator

<, >, <=, >=	Relational operators
<>, !=, ~=, ^=	Different versions of NOT EQUAL

### The PL/SQL Comments

Program comments are explanatory statements that can be included in the PL/SQL code that you write and helps anyone reading its source code. All programming languages allow some form of comments.

The PL/SQL supports single-line and multi-line comments. All characters available inside any comment are ignored by the PL/SQL compiler. The PL/SQL single-line comments start with the delimiter -- (double hyphen) and multi-line comments are enclosed by /\* and \*/.

```
DECLARE
  -- variable declaration
  message varchar2(20):= 'Hello, World!';
BEGIN
  /*
   * PL/SQL executable statement(s)
   */
  dbms_output.put_line(message);
END;
/
```

When the above code is executed at the SQL prompt, it produces the following result –

Hello World

PL/SQL procedure successfully completed.

### PL/SQL Program Units

A PL/SQL unit is any one of the following –

- PL/SQL block
- Function
- Package
- Package body
- Procedure
- Trigger
- Type
- Type body

S.No	Category & Description
1	<b>Scalar</b> Single values with no internal components, such as a <b>NUMBER</b> , <b>DATE</b> , or <b>BOOLEAN</b> .
2	<b>Large Object (LOB)</b> Pointers to large objects that are stored separately from other data items, such as text, graphic images, video clips, and sound waveforms.

3	<b>Composite</b> Data items that have internal components that can be accessed individually. For example, collections and records.
4	<b>Reference</b> Pointers to other data items.

### PL/SQL Scalar Data Types and Subtypes

PL/SQL Scalar Data Types and Subtypes come under the following categories –

S.No	Date Type & Description
1	<b>Numeric</b> Numeric values on which arithmetic operations are performed.
2	<b>Character</b> Alphanumeric values that represent single characters or strings of characters.
3	<b>Boolean</b> Logical values on which logical operations are performed.
4	<b>Datetime</b> Dates and times.

PL/SQL provides subtypes of data types. For example, the data type NUMBER has a subtype called INTEGER. You can use the subtypes in your PL/SQL program to make the data types compatible with data types in other programs while embedding the PL/SQL code in another program, such as a Java program.

### PL/SQL Numeric Data Types and Subtypes

Following table lists out the PL/SQL pre-defined numeric data types and their sub-types –

S.No	Data Type & Description
1	<b>PLS_INTEGER</b> Signed integer in range -2,147,483,648 through 2,147,483,647, represented in 32 bits
2	<b>BINARY_INTEGER</b> Signed integer in range -2,147,483,648 through 2,147,483,647, represented in 32 bits
3	<b>BINARY_FLOAT</b> Single-precision IEEE 754-format floating-point number

4	<b>BINARY_DOUBLE</b> Double-precision IEEE 754-format floating-point number
5	<b>NUMBER(prec, scale)</b> Fixed-point or floating-point number with absolute value in range 1E-130 to (but not including) 1.0E126. A NUMBER variable can also represent 0
6	<b>DEC(prec, scale)</b> ANSI specific fixed-point type with maximum precision of 38 decimal digits
7	<b>DECIMAL(prec, scale)</b> IBM specific fixed-point type with maximum precision of 38 decimal digits
8	<b>NUMERIC(pre, scale)</b> Floating type with maximum precision of 38 decimal digits
9	<b>DOUBLE PRECISION</b> ANSI specific floating-point type with maximum precision of 126 binary digits (approximately 38 decimal digits)
10	<b>FLOAT</b> ANSI and IBM specific floating-point type with maximum precision of 126 binary digits (approximately 38 decimal digits)
11	<b>INT</b> ANSI specific integer type with maximum precision of 38 decimal digits
12	<b>INTEGER</b> ANSI and IBM specific integer type with maximum precision of 38 decimal digits
13	<b>SMALLINT</b> ANSI and IBM specific integer type with maximum precision of 38 decimal digits
14	<b>REAL</b> Floating-point type with maximum precision of 63 binary digits (approximately 18 decimal digits)

Following is a valid declaration –

```
DECLARE
  num1 INTEGER;
  num2 REAL;
  num3 DOUBLE PRECISION;
BEGIN
  null;
END;
/
```

When the above code is compiled and executed, it produces the following result –

PL/SQL procedure successfully completed

### PL/SQL Character Data Types and Subtypes

Following is the detail of PL/SQL pre-defined character data types and their sub-types –

S.No	Data Type & Description
1	<b>CHAR</b> Fixed-length character string with maximum size of 32,767 bytes
2	<b>VARCHAR2</b> Variable-length character string with maximum size of 32,767 bytes
3	<b>RAW</b> Variable-length binary or byte string with maximum size of 32,767 bytes, not interpreted by PL/SQL
4	<b>NCHAR</b> Fixed-length national character string with maximum size of 32,767 bytes
5	<b>NVARCHAR2</b> Variable-length national character string with maximum size of 32,767 bytes
6	<b>LONG</b> Variable-length character string with maximum size of 32,760 bytes
7	<b>LONG RAW</b> Variable-length binary or byte string with maximum size of 32,760 bytes, not interpreted by PL/SQL
8	<b>ROWID</b> Physical row identifier, the address of a row in an ordinary table
9	<b>UROWID</b>

	Universal row identifier (physical, logical, or foreign row identifier)
--	---

## PL/SQL Boolean Data Types

The **BOOLEAN** data type stores logical values that are used in logical operations. The logical values are the Boolean values **TRUE** and **FALSE** and the value **NULL**.

However, SQL has no data type equivalent to **BOOLEAN**. Therefore, Boolean values cannot be used in –

- SQL statements
- Built-in SQL functions (such as **TO\_CHAR**)
- PL/SQL functions invoked from SQL statements

## PL/SQL Datetime and Interval Types

The **DATE** datatype is used to store fixed-length datetimes, which include the time of day in seconds since midnight. Valid dates range from January 1, 4712 BC to December 31, 9999 AD.

The default date format is set by the Oracle initialization parameter **NLS\_DATE\_FORMAT**. For example, the default might be 'DD-MON-YY', which includes a two-digit number for the day of the month, an abbreviation of the month name, and the last two digits of the year. For example, 01-OCT-12.

Each **DATE** includes the century, year, month, day, hour, minute, and second. The following table shows the valid values for each field –

Field Name	Valid Datetime Values	Valid Interval Values
YEAR	-4712 to 9999 (excluding year 0)	Any nonzero integer
MONTH	01 to 12	0 to 11
DAY	01 to 31 (limited by the values of MONTH and YEAR, according to the rules of the calendar for the locale)	Any nonzero integer
HOUR	00 to 23	0 to 23
MINUTE	00 to 59	0 to 59
SECOND	00 to 59.9(n), where 9(n) is the precision of time fractional seconds	0 to 59.9(n), where 9(n) is the precision of interval fractional seconds
TIMEZONE_HOUR	-12 to 14 (range accommodates daylight savings time changes)	Not applicable
TIMEZONE_MINUTE	00 to 59	Not applicable



TIMEZONE_REGION	Found in the dynamic performance view V\$TIMEZONE_NAMES	Not applicable
TIMEZONE_ABBR	Found in the dynamic performance view V\$TIMEZONE_NAMES	Not applicable

### PL/SQL Large Object (LOB) Data Types

Large Object (LOB) data types refer to large data items such as text, graphic images, video clips, and sound waveforms. LOB data types allow efficient, random, piecewise access to this data. Following are the predefined PL/SQL LOB data types –

Data Type	Description	Size
BFILE	Used to store large binary objects in operating system files outside the database.	System-dependent. Cannot exceed 4 gigabytes (GB).
BLOB	Used to store large binary objects in the database.	8 to 128 terabytes (TB)
CLOB	Used to store large blocks of character data in the database.	8 to 128 TB
NCLOB	Used to store large blocks of NCHAR data in the database.	8 to 128 TB

### PL/SQL User-Defined Subtypes

A subtype is a subset of another data type, which is called its base type. A subtype has the same valid operations as its base type, but only a subset of its valid values.

PL/SQL predefines several subtypes in package **STANDARD**. For example, PL/SQL predefines the subtypes **CHARACTER** and **INTEGER** as follows –

```
SUBTYPE CHARACTER IS CHAR;
SUBTYPE INTEGER IS NUMBER(38,0);
```

You can define and use your own subtypes. The following program illustrates defining and using a user-defined subtype –

```
DECLARE
  SUBTYPE name IS char(20);
  SUBTYPE message IS varchar2(100);
  salutation name;
  greetings message;
BEGIN
  salutation := 'Reader ';
  greetings := 'Welcome to the World of PL/SQL';
  dbms_output.put_line('Hello ' || salutation || greetings);
END;
```

When the above code is executed at the SQL prompt, it produces the following result –

Hello Reader Welcome to the World of PL/SQL

PL/SQL procedure successfully completed.

## NULLs in PL/SQL

PL/SQL NULL values represent **missing** or **unknown data** and they are not an integer, a character, or any other specific data type. Note that **NULL** is not the same as an empty data string or the null character value '\0'. A null can be assigned but it cannot be equated with anything, including itself.

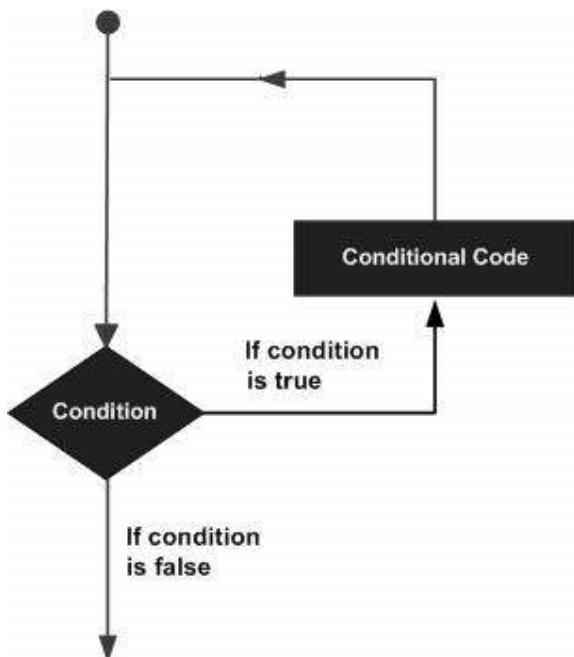
Following table shows all the arithmetic operators supported by PL/SQL. Let us assume **variable A** holds 10 and **variable B** holds 5, then –

Operator	Description	Example
+	Adds two operands	A + B will give 15
-	Subtracts second operand from the first	A - B will give 5
*	Multiplies both operands	A * B will give 50
/	Divides numerator by de-numerator	A / B will give 2
**	Exponentiation operator, raises one operand to the power of other	A ** B will give 100000

## LOOPS AND CONTROL STATEMENTS

Programming languages provide various control structures that allow for more complicated execution paths.

A loop statement allows us to execute a statement or group of statements multiple times and following is the general form of a loop statement in most of the programming languages –



PL/SQL provides the following types of loop to handle the looping requirements. Click the following links to check their detail.

S.No	Loop Type & Description
1	<a href="#">PL/SQL Basic LOOP</a> In this loop structure, sequence of statements is enclosed between the LOOP and the END LOOP statements. At each iteration, the sequence of statements is executed and then control resumes at the top of the loop.
2	<a href="#">PL/SQL WHILE LOOP</a> Repeats a statement or group of statements while a given condition is true. It tests the condition before executing the loop body.
3	<a href="#">PL/SQL FOR LOOP</a> Execute a sequence of statements multiple times and abbreviates the code that manages the loop variable.
4	<a href="#">Nested loops in PL/SQL</a> You can use one or more loop inside any another basic loop, while, or for loop.

#### Labeling a PL/SQL Loop

PL/SQL loops can be labeled. The label should be enclosed by double angle brackets (<< and >>) and appear at the beginning of the LOOP statement. The label name can also appear at the end of the LOOP statement. You may use the label in the EXIT statement to exit from the loop.

The following program illustrates the concept –

```
DECLARE
```

```
    i number(1);
```

```
    j number(1);
```

```
BEGIN
```

```
    << outer_loop >>
```

```
    FOR i IN 1..3 LOOP
```

```
        << inner_loop >>
```

```
        FOR j IN 1..3 LOOP
```

```
            dbms_output.put_line('i is: ' || i || ' and j is: ' || j);
```

```
        END loop inner_loop;
```

```
    END loop outer_loop;
```

```
END;
```

```
/
```

When the above code is executed at the SQL prompt, it produces the following result –

```
i is: 1 and j is: 1
```

```
i is: 1 and j is: 2
```

i is: 1 and j is: 3  
i is: 2 and j is: 1  
i is: 2 and j is: 2  
i is: 2 and j is: 3  
i is: 3 and j is: 1  
i is: 3 and j is: 2  
i is: 3 and j is: 3

PL/SQL procedure successfully completed.

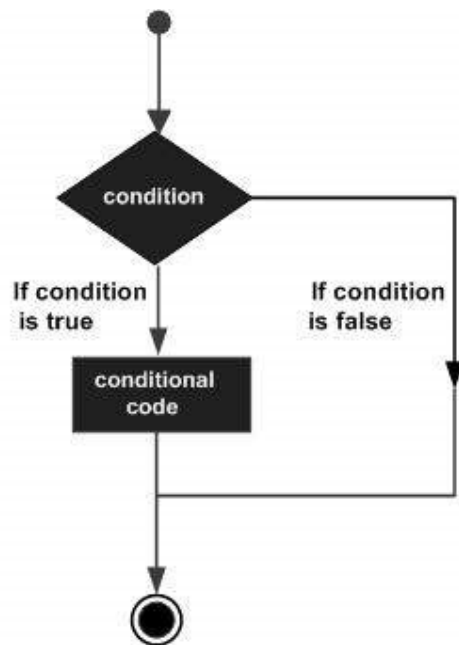
### The Loop Control Statements

Loop control statements change execution from its normal sequence. When execution leaves a scope, all automatic objects that were created in that scope are destroyed.

PL/SQL supports the following control statements. Labeling loops also help in taking the control outside a loop. Click the following links to check their details.

S.No	Control Statement & Description
1	<a href="#">EXIT statement</a> The Exit statement completes the loop and control passes to the statement immediately after the END LOOP.
2	<a href="#">CONTINUE statement</a> Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating.
3	<a href="#">GOTO statement</a> Transfers control to the labeled statement. Though it is not advised to use the GOTO statement in your program.

Following is the general form of a typical conditional (i.e., decision making) structure found in most of the programming languages –



PL/SQL programming language provides following types of decision-making statements. Click the following links to check their detail.

S.No	Statement & Description
1	<a href="#">IF - THEN statement</a> The <b>IF statement</b> associates a condition with a sequence of statements enclosed by the keywords <b>THEN</b> and <b>END IF</b> . If the condition is true, the statements get executed and if the condition is false or NULL then the IF statement does nothing.
2	<a href="#">IF-THEN-ELSE statement</a> <b>IF statement</b> adds the keyword <b>ELSE</b> followed by an alternative sequence of statement. If the condition is false or NULL, then only the alternative sequence of statements get executed. It ensures that either of the sequence of statements is executed.
3	<a href="#">IF-THEN-ELSIF statement</a> It allows you to choose between several alternatives.
4	<a href="#">Case statement</a> Like the IF statement, the <b>CASE statement</b> selects one sequence of statements to execute.  However, to select the sequence, the CASE statement uses a selector rather than multiple Boolean expressions. A selector is an expression whose value is used to select one of several alternatives.
5	<a href="#">Searched CASE statement</a>

	The searched CASE statement <b>has no selector</b> , and it's WHEN clauses contain search conditions that yield Boolean values.
6	<a href="#">nested IF-THEN-ELSE</a> You can use one <b>IF-THEN</b> or <b>IF-THEN-ELSIF</b> statement inside another <b>IF-THEN</b> or <b>IF-THEN-ELSIF</b> statement(s).

## **CURSOR MANAGEMENT:**

A **cursor** is a pointer to this context area. PL/SQL controls the context area through a cursor. A cursor holds the rows (one or more) returned by a SQL statement. The set of rows the cursor holds is referred to as the **active set**.

You can name a cursor so that it could be referred to in a program to fetch and process the rows returned by the SQL statement, one at a time. There are two types of cursors –

- Implicit cursors
- Explicit cursors

### **Implicit Cursors**

Implicit cursors are automatically created by Oracle whenever an SQL statement is executed, when there is no explicit cursor for the statement. Programmers cannot control the implicit cursors and the information in it.

Whenever a DML statement (INSERT, UPDATE and DELETE) is issued, an implicit cursor is associated with this statement. For INSERT operations, the cursor holds the data that needs to be inserted. For UPDATE and DELETE operations, the cursor identifies the rows that would be affected.

In PL/SQL, you can refer to the most recent implicit cursor as the **SQL cursor**, which always has attributes such as **%FOUND**, **%ISOPEN**, **%NOTFOUND**, and **%ROWCOUNT**. The SQL cursor has additional attributes, **%BULK\_ROWCOUNT** and **%BULK\_EXCEPTIONS**, designed for use with the **FORALL** statement. The following table provides the description of the most used attributes –

S.No	Attribute & Description
1	<b>%FOUND</b> Returns TRUE if an INSERT, UPDATE, or DELETE statement affected one or more rows or a SELECT INTO statement returned one or more rows. Otherwise, it returns FALSE.
2	<b>%NOTFOUND</b> The logical opposite of %FOUND. It returns TRUE if an INSERT, UPDATE, or DELETE statement affected no rows, or a SELECT INTO statement returned no rows. Otherwise, it returns FALSE.
3	<b>%ISOPEN</b> Always returns FALSE for implicit cursors, because Oracle closes the SQL cursor automatically after executing its associated SQL statement.

4	<b>%ROWCOUNT</b> Returns the number of rows affected by an INSERT, UPDATE, or DELETE statement, or returned by a SELECT INTO statement.
---	--

Any SQL cursor attribute will be accessed as **sql%attribute\_name** as shown below in the example.

Example

We will be using the CUSTOMERS table we had created and used in the previous chapters.

Select \* from customers;

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

The following program will update the table and increase the salary of each customer by 500 and use the **SQL%ROWCOUNT** attribute to determine the number of rows affected –

```
DECLARE
    total_rows number(2);
BEGIN
    UPDATE customers
    SET salary = salary + 500;
    IF sql%notfound THEN
        dbms_output.put_line('no customers selected');
    ELSIF sql%found THEN
        total_rows := sql%rowcount;
        dbms_output.put_line( total_rows || ' customers selected ');
    END IF;
END;
/
```

When the above code is executed at the SQL prompt, it produces the following result –

6 customers selected

PL/SQL procedure successfully completed.

If you check the records in customers table, you will find that the rows have been updated –

Select \* from customers;

```
+---+-----+---+-----+-----+
| ID | NAME   | AGE | ADDRESS | SALARY |
+---+-----+---+-----+-----+
| 1 | Ramesh | 32 | Ahmedabad | 2500.00 |
| 2 | Khilan | 25 | Delhi    | 2000.00 |
| 3 | kaushik | 23 | Kota     | 2500.00 |
| 4 | Chaitali | 25 | Mumbai   | 7000.00 |
| 5 | Hardik | 27 | Bhopal   | 9000.00 |
| 6 | Komal  | 22 | MP       | 5000.00 |
+---+-----+---+-----+-----+
```

### Explicit Cursors

Explicit cursors are programmer-defined cursors for gaining more control over the **context area**. An explicit cursor should be defined in the declaration section of the PL/SQL Block. It is created on a SELECT Statement which returns more than one row.

The syntax for creating an explicit cursor is –

```
CURSOR cursor_name IS select_statement;
```

Working with an explicit cursor includes the following steps –

- Declaring the cursor for initializing the memory
- Opening the cursor for allocating the memory
- Fetching the cursor for retrieving the data
- Closing the cursor to release the allocated memory

### Declaring the Cursor

Declaring the cursor defines the cursor with a name and the associated SELECT statement. For example –

```
CURSOR c_customers IS
```

```
    SELECT id, name, address FROM customers;
```

### Opening the Cursor

Opening the cursor allocates the memory for the cursor and makes it ready for fetching the rows returned by the SQL statement into it. For example, we will open the above defined cursor as follows –

```
OPEN c_customers;
```



### Fetching the Cursor

Fetching the cursor involves accessing one row at a time. For example, we will fetch rows from the above-opened cursor as follows –

```
FETCH c_customers INTO c_id, c_name, c_addr;
```

### Closing the Cursor

Closing the cursor means releasing the allocated memory. For example, we will close the above-opened cursor as follows –

```
CLOSE c_customers;
```

### Example

Following is a complete example to illustrate the concepts of explicit cursors &minua;

```
DECLARE
    c_id customers.id%type;
    c_name customers.name%type;
    c_addr customers.address%type;
    CURSOR c_customers is
        SELECT id, name, address FROM customers;
BEGIN
    OPEN c_customers;
    LOOP
        FETCH c_customers into c_id, c_name, c_addr;
        EXIT WHEN c_customers%notfound;
        dbms_output.put_line(c_id || ' ' || c_name || ' ' || c_addr);
    END LOOP;
    CLOSE c_customers;
END;
/
```

When the above code is executed at the SQL prompt, it produces the following result –

```
1 Ramesh Ahmedabad
2 Khilan Delhi
3 kaushik Kota
4 Chaitali Mumbai
5 Hardik Bhopal
6 Komal MP
```

PL/SQL procedure successfully completed.

## **EXCEPTION HANDLING:**

An exception is an error condition during a program execution. PL/SQL supports programmers to catch such conditions using **EXCEPTION** block in the program and an appropriate action is taken against the error condition. There are two types of exceptions –

- System-defined exceptions
- User-defined exceptions

### Syntax for Exception Handling

The general syntax for exception handling is as follows. Here you can list down as many exceptions as you can handle. The default exception will be handled using ***WHEN others THEN*** –

DECLARE

<declarations section>

BEGIN

<executable command(s)>

EXCEPTION

<exception handling goes here >

WHEN exception1 THEN

exception1-handling-statements

WHEN exception2 THEN

exception2-handling-statements

WHEN exception3 THEN

exception3-handling-statements

.....

WHEN others THEN

exception3-handling-statements

END;

### Example

Let us write a code to illustrate the concept. We will be using the CUSTOMERS table we had created and used in the previous chapters –

DECLARE

c\_id customers.id%type := 8;

c\_name customerS.Name%type;

c\_addr customers.address%type;

BEGIN

SELECT name, address INTO c\_name, c\_addr

FROM customers

WHERE id = c\_id;

```
DBMS_OUTPUT.PUT_LINE ('Name: ' || c_name);
DBMS_OUTPUT.PUT_LINE ('Address: ' || c_addr);
```

EXCEPTION

```
WHEN no_data_found THEN
    dbms_output.put_line('No such customer!');
WHEN others THEN
    dbms_output.put_line('Error!');
```

END;

/

When the above code is executed at the SQL prompt, it produces the following result –

No such customer!

PL/SQL procedure successfully completed.

The above program displays the name and address of a customer whose ID is given. Since there is no customer with ID value 8 in our database, the program raises the run-time exception **NO\_DATA\_FOUND**, which is captured in the **EXCEPTION block**.

Raising Exceptions

Exceptions are raised by the database server automatically whenever there is any internal database error, but exceptions can be raised explicitly by the programmer by using the command **RAISE**. Following is the simple syntax for raising an exception –

DECLARE

```
exception_name EXCEPTION;
```

BEGIN

```
IF condition THEN
```

```
    RAISE exception_name;
```

```
END IF;
```

EXCEPTION

```
WHEN exception_name THEN
```

```
    statement;
```

END;

You can use the above syntax in raising the Oracle standard exception or any user-defined exception. In the next section, we will give you an example on raising a user-defined exception. You can raise the Oracle standard exceptions in a similar way.

User-defined Exceptions

PL/SQL allows you to define your own exceptions according to the need of your program. A user-defined exception must be declared and then raised explicitly, using either a RAISE statement or the procedure **DBMS\_STANDARD.RAISE\_APPLICATION\_ERROR**.

The syntax for declaring an exception is –

DECLARE

my-exception EXCEPTION;

### Example

The following example illustrates the concept. This program asks for a customer ID, when the user enters an invalid ID, the exception **invalid\_id** is raised.

DECLARE

c\_id customers.id%type := &cc\_id;

c\_name customerS.Name%type;

c\_addr customers.address%type;

-- user defined exception

ex\_invalid\_id EXCEPTION;

BEGIN

IF c\_id <= 0 THEN

RAISE ex\_invalid\_id;

ELSE

SELECT name, address INTO c\_name, c\_addr

FROM customers

WHERE id = c\_id;

DBMS\_OUTPUT.PUT\_LINE ('Name: ' || c\_name);

DBMS\_OUTPUT.PUT\_LINE ('Address: ' || c\_addr);

END IF;

EXCEPTION

WHEN ex\_invalid\_id THEN

dbms\_output.put\_line('ID must be greater than zero!');

WHEN no\_data\_found THEN

dbms\_output.put\_line('No such customer!');

WHEN others THEN

dbms\_output.put\_line('Error!');

END;

/

When the above code is executed at the SQL prompt, it produces the following result –

Enter value for cc\_id: -6 (let's enter a value -6)

old 2: c\_id customers.id%type := &cc\_id;

new 2: c\_id customers.id%type := -6;

ID must be greater than zero!

PL/SQL procedure successfully completed.

### Pre-defined Exceptions

PL/SQL provides many pre-defined exceptions, which are executed when any database rule is violated by a program. For example, the predefined exception NO\_DATA\_FOUND is raised when a SELECT INTO statement returns no rows. The following table lists few of the important pre-defined exceptions –

Exception	Oracle Error	SQLCODE	Description
ACCESS_INTO_NULL	06530	-6530	It is raised when a null object is automatically assigned a value.
CASE_NOT_FOUND	06592	-6592	It is raised when none of the choices in the WHEN clause of a CASE statement is selected, and there is no ELSE clause.
COLLECTION_IS_NULL	06531	-6531	It is raised when a program attempts to apply collection methods other than EXISTS to an uninitialized nested table or varray, or the program attempts to assign values to the elements of an uninitialized nested table or varray.
DUP_VAL_ON_INDEX	00001	-1	It is raised when duplicate values are attempted to be stored in a column with unique index.
INVALID_CURSOR	01001	-1001	It is raised when attempts are made to make a cursor operation that is not allowed, such as closing an unopened cursor.
INVALID_NUMBER	01722	-1722	It is raised when the conversion of a character string into a number fails because the string does not represent a valid number.
LOGIN_DENIED	01017	-1017	It is raised when a program attempts to log on to the database with an invalid username or password.
NO_DATA_FOUND	01403	+100	It is raised when a SELECT INTO statement returns no rows.

NOT_LOGGED_ON	01012	-1012	It is raised when a database call is issued without being connected to the database.
PROGRAM_ERROR	06501	-6501	It is raised when PL/SQL has an internal problem.
ROWTYPE_MISMATCH	06504	-6504	It is raised when a cursor fetches value in a variable having incompatible data type.
SELF_IS_NULL	30625	-30625	It is raised when a member method is invoked, but the instance of the object type was not initialized.
STORAGE_ERROR	06500	-6500	It is raised when PL/SQL ran out of memory or memory was corrupted.
TOO_MANY_ROWS	01422	-1422	It is raised when a SELECT INTO statement returns more than one row.
VALUE_ERROR	06502	-6502	It is raised when an arithmetic, conversion, truncation, or sizeconstraint error occurs.
ZERO_DIVIDE	01476	1476	It is raised when an attempt is made to divide a number by zero.

### **TRIGGERS:**

Triggers are stored programs, which are automatically executed or fired when some events occur. Triggers are, in fact, 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 can be defined on the table, view, schema, or database with which the event is associated.

### **Benefits of Triggers**

Triggers can be written for the following purposes –

- Generating some derived column values automatically
- Enforcing referential integrity
- Event logging and storing information on table access
- Auditing
- Synchronous replication of tables
- Imposing security authorizations
- Preventing invalid transactions

## Creating Triggers

The syntax for creating a trigger is –

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;

Where,

- CREATE [OR REPLACE] TRIGGER trigger\_name – Creates or replaces an existing trigger with the *trigger\_name*.
- {BEFORE | AFTER | INSTEAD OF} – This specifies when the trigger will 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 will 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, such as INSERT, UPDATE, and DELETE.
- [FOR EACH ROW] – This specifies a row-level trigger, i.e., the trigger will 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.

## Example

To start with, we will be using the CUSTOMERS table we had created and used in the previous chapters –

Select \* from customers;

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	32	Ahmedabad	2000.00
2	Khilan	25	Delhi	1500.00
3	kaushik	23	Kota	2000.00
4	Chaitali	25	Mumbai	6500.00
5	Hardik	27	Bhopal	8500.00
6	Komal	22	MP	4500.00

The following program creates a **row-level** trigger for the customers table that would fire for INSERT or UPDATE or DELETE operations performed on the CUSTOMERS table. This trigger will display the salary difference between the old values and new values –

```
CREATE OR REPLACE TRIGGER display_salary_changes
BEFORE DELETE OR INSERT OR UPDATE ON customers
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;
/
```

When the above code is executed at the SQL prompt, it produces the following result –

Trigger created.

The following points need to be considered here –

- OLD and NEW references are not available for table-level triggers, rather you can use them for record-level triggers.
- If you want to query the table in the same trigger, then you should use the AFTER keyword, because triggers can query the table or change it again only after the initial changes are applied and the table is back in a consistent state.
- The above trigger has been written in such a way that it will fire before any DELETE or INSERT or UPDATE operation on the table, but you can write your trigger on a single or multiple operations, for



example BEFORE DELETE, which will fire whenever a record will be deleted using the DELETE operation on the table.

### Triggering a Trigger

Let us perform some DML operations on the CUSTOMERS table. Here is one INSERT statement, which will create a new record in the table –

```
INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (7, 'Kriti', 22, 'HP', 7500.00 );
```

When a record is created in the CUSTOMERS table, the above create trigger, **display\_salary\_changes** will be fired and it will display the following result –

Old salary:

New salary: 7500

Salary difference:

Because this is a new record, old salary is not available and the above result comes as null. Let us now perform one more DML operation on the CUSTOMERS table. The UPDATE statement will update an existing record in the table –

### UPDATE customers

```
SET salary = salary + 500
```

```
WHERE id = 2;
```

When a record is updated in the CUSTOMERS table, the above create trigger, **display\_salary\_changes** will be fired and it will display the following result –

Old salary: 1500

New salary: 2000

Salary difference: 500

### **FUNCTIONS:**

A function is same as a procedure except that it returns a value. Therefore, all the discussions of the previous chapter are true for functions too.

### Creating a Function

A standalone function is created using the **CREATE FUNCTION** statement. The simplified syntax for the **CREATE OR REPLACE PROCEDURE** statement is as follows –

```
CREATE [OR REPLACE] FUNCTION function_name
```

```
[(parameter_name [IN | OUT | IN OUT] type [, ...])]
```

```
RETURN return_datatype
```

```
{IS | AS}
```

```
BEGIN
```

```
< function_body >
```

```
END [function_name];
```

Where,

- *function-name* specifies the name of the function.
- [OR REPLACE] option allows the modification of an existing function.
- The optional parameter list contains name, mode and types of the parameters. IN represents the value that will be passed from outside and OUT represents the parameter that will be used to return a value outside of the procedure.
- The function must contain a **return** statement.
- The *RETURN* clause specifies the data type you are going to return from the function.
- *function-body* contains the executable part.
- The AS keyword is used instead of the IS keyword for creating a standalone function.

### Example

The following example illustrates how to create and call a standalone function. This function returns the total number of CUSTOMERS in the customers table.

We will use the CUSTOMERS table, which we had created in the [PL/SQL Variables](#) chapter –

Select \* from customers;

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

```
CREATE OR REPLACE FUNCTION totalCustomers
```

```
RETURN number IS
```

```
total number(2) := 0;
```

```
BEGIN
```

```
SELECT count(*) into total
```

```
FROM customers;
```

```
RETURN total;
```

```
END;
```

```
/
```

When the above code is executed using the SQL prompt, it will produce the following result –

Function created.

### Calling a Function

While creating a function, you give a definition of what the function has to do. To use a function, you will have to call that function to perform the defined task. When a program calls a function, the program control is transferred to the called function.

A called function performs the defined task and when its return statement is executed or when the **last end statement** is reached, it returns the program control back to the main program.

To call a function, you simply need to pass the required parameters along with the function name and if the function returns a value, then you can store the returned value. Following program calls the function **totalCustomers** from an anonymous block –

```
DECLARE
    c number(2);
BEGIN
    c := totalCustomers();
    dbms_output.put_line('Total no. of Customers: ' || c);
END;
/
```

When the above code is executed at the SQL prompt, it produces the following result –

Total no. of Customers: 6

PL/SQL procedure successfully completed.

### Example

The following example demonstrates Declaring, Defining, and Invoking a Simple PL/SQL Function that computes and returns the maximum of two values.

```
DECLARE
    a number;
    b number;
    c number;
FUNCTION findMax(x IN number, y IN number)
RETURN number
IS
    z number;
BEGIN
    IF x > y THEN
        z:= x;
```

```

ELSE
    Z:= y;
END IF;
RETURN z;
END;
BEGIN
    a:= 23;
    b:= 45;
    c := findMax(a, b);
    dbms_output.put_line(' Maximum of (23,45): ' || c);
END;
/

```

When the above code is executed at the SQL prompt, it produces the following result –

Maximum of (23,45): 45

PL/SQL procedure successfully completed.

#### PL/SQL Recursive Functions

We have seen that a program or subprogram may call another subprogram. When a subprogram calls itself, it is referred to as a recursive call and the process is known as **recursion**.

To illustrate the concept, let us calculate the factorial of a number. Factorial of a number n is defined as –

$$\begin{aligned}
 n! &= n*(n-1)! \\
 &= n*(n-1)*(n-2)! \\
 &\dots \\
 &= n*(n-1)*(n-2)*(n-3)\dots 1
 \end{aligned}$$

The following program calculates the factorial of a given number by calling itself recursively –

```

DECLARE
    num number;
    factorial number;

FUNCTION fact(x number)
RETURN number
IS
    f number;
BEGIN
    IF x=0 THEN

```

```

    f := 1;

ELSE

    f := x * fact(x-1);

END IF;

RETURN f;

END;

BEGIN

    num:= 6;

    factorial := fact(num);

    dbms_output.put_line(' Factorial ' || num || ' is ' || factorial);

END;

/

```

When the above code is executed at the SQL prompt, it produces the following result –

Factorial 6 is 720

### **PROCEDURES:**

A **subprogram** is a program unit/module that performs a particular task. These subprograms are combined to form larger programs. This is basically called the 'Modular design'. A subprogram can be invoked by another subprogram or program which is called the **calling program**.

A subprogram can be created –

- At the schema level
- Inside a package
- Inside a PL/SQL block

At the schema level, subprogram is a **standalone subprogram**. It is created with the CREATE PROCEDURE or the CREATE FUNCTION statement. It is stored in the database and can be deleted with the DROP PROCEDURE or DROP FUNCTION statement.

A subprogram created inside a package is a **packaged subprogram**. It is stored in the database and can be deleted only when the package is deleted with the DROP PACKAGE statement. We will discuss packages in the chapter '**PL/SQL - Packages**'.

PL/SQL subprograms are named PL/SQL blocks that can be invoked with a set of parameters. PL/SQL provides two kinds of subprograms –

- **Functions** – These subprograms return a single value; mainly used to compute and return a value.
- **Procedures** – These subprograms do not return a value directly; mainly used to perform an action.

This chapter is going to cover important aspects of a **PL/SQL procedure**. We will discuss **PL/SQL function** in the next chapter.

### **Parts of a PL/SQL Subprogram**

Each PL/SQL subprogram has a name, and may also have a parameter list. Like anonymous PL/SQL blocks, the named blocks will also have the following three parts –

S.No	Parts & Description
1	<b>Declarative Part</b> It is an optional part. However, the declarative part for a subprogram does not start with the DECLARE keyword. It contains declarations of types, cursors, constants, variables, exceptions, and nested subprograms. These items are local to the subprogram and cease to exist when the subprogram completes execution.
2	<b>Executable Part</b> This is a mandatory part and contains statements that perform the designated action.
3	<b>Exception-handling</b> This is again an optional part. It contains the code that handles run-time errors.

#### Creating a Procedure

A procedure is created with the **CREATE OR REPLACE PROCEDURE** statement. The simplified syntax for the CREATE OR REPLACE PROCEDURE statement is as follows –

```
CREATE [OR REPLACE] PROCEDURE procedure_name
```

```
[(parameter_name [IN | OUT | IN OUT] type [, ...])]
```

```
{IS | AS}
```

```
BEGIN
```

```
< procedure_body >
```

```
END procedure_name;
```

Where,

- *procedure-name* specifies the name of the procedure.
- [OR REPLACE] option allows the modification of an existing procedure.
- The optional parameter list contains name, mode and types of the parameters. **IN** represents the value that will be passed from outside and **OUT** represents the parameter that will be used to return a value outside of the procedure.
- *procedure-body* contains the executable part.
- The AS keyword is used instead of the IS keyword for creating a standalone procedure.

#### Example

The following example creates a simple procedure that displays the string 'Hello World!' on the screen when executed.

```
CREATE OR REPLACE PROCEDURE greetings
```

```
AS
```

```
BEGIN
```

```
    dbms_output.put_line('Hello World!');
```

```
END;
```

```
/
```

When the above code is executed using the SQL prompt, it will produce the following result –

Procedure created.

Executing a Standalone Procedure

A standalone procedure can be called in two ways –

- Using the **EXECUTE** keyword
- Calling the name of the procedure from a PL/SQL block

The above procedure named '**greetings**' can be called with the EXECUTE keyword as –

```
EXECUTE greetings;
```

The above call will display –

Hello World

PL/SQL procedure successfully completed.

The procedure can also be called from another PL/SQL block –

```
BEGIN
```

```
    greetings;
```

```
END;
```

```
/
```

The above call will display –

Hello World

PL/SQL procedure successfully completed.

Deleting a Standalone Procedure

A standalone procedure is deleted with the **DROP PROCEDURE** statement. Syntax for deleting a procedure is –

```
DROP PROCEDURE procedure-name;
```

You can drop the greetings procedure by using the following statement –

```
DROP PROCEDURE greetings;
```

## Parameter Modes in PL/SQL Subprograms

The following table lists out the parameter modes in PL/SQL subprograms –

S.No	Parameter Mode & Description
1	<b>IN</b> An IN parameter lets you pass a value to the subprogram. <b>It is a read-only parameter.</b> Inside the subprogram, an IN parameter acts like a constant. It cannot be assigned a value. You can pass a constant, literal, initialized variable, or expression as an IN parameter. You can also initialize it to a default value; however, in that case, it is omitted from the subprogram call. <b>It is the default mode of parameter passing. Parameters are passed by reference.</b>
2	<b>OUT</b> An OUT parameter returns a value to the calling program. Inside the subprogram, an OUT parameter acts like a variable. You can change its value and reference the value after assigning it. <b>The actual parameter must be variable and it is passed by value.</b>
3	<b>IN OUT</b> An <b>IN OUT</b> parameter passes an initial value to a subprogram and returns an updated value to the caller. It can be assigned a value and the value can be read. The actual parameter corresponding to an IN OUT formal parameter must be a variable, not a constant or an expression. Formal parameter must be assigned a value. <b>Actual parameter is passed by value.</b>

### IN & OUT Mode Example 1

This program finds the minimum of two values. Here, the procedure takes two numbers using the IN mode and returns their minimum using the OUT parameters.

DECLARE

a number;

b number;

c number;

PROCEDURE findMin(x IN number, y IN number, z OUT number) IS

BEGIN

IF x < y THEN

z:= x;

ELSE

z:= y;

END IF;

END;



```

BEGIN
    a:= 23;
    b:= 45;
    findMin(a, b, c);
    dbms_output.put_line(' Minimum of (23, 45) : ' || c);
END;
/

```

When the above code is executed at the SQL prompt, it produces the following result –

Minimum of (23, 45) : 23

PL/SQL procedure successfully completed.

### IN & OUT Mode Example 2

This procedure computes the square of value of a passed value. This example shows how we can use the same parameter to accept a value and then return another result.

```

DECLARE
    a number;
PROCEDURE squareNum(x IN OUT number) IS
BEGIN
    x := x * x;
END;
BEGIN
    a:= 23;
    squareNum(a);
    dbms_output.put_line(' Square of (23): ' || a);
END;
/

```

When the above code is executed at the SQL prompt, it produces the following result –

Square of (23): 529

PL/SQL procedure successfully completed.

### Methods for Passing Parameters

Actual parameters can be passed in three ways –

- Positional notation
- Named notation
- Mixed notation

### Positional Notation

In positional notation, you can call the procedure as –

```
findMin(a, b, c, d);
```

In positional notation, the first actual parameter is substituted for the first formal parameter; the second actual parameter is substituted for the second formal parameter, and so on. So, **a** is substituted for **x**, **b** is substituted for **y**, **c** is substituted for **z** and **d** is substituted for **m**.

### Named Notation

In named notation, the actual parameter is associated with the formal parameter using the **arrow symbol (=>)**. The procedure call will be like the following –

```
findMin(x => a, y => b, z => c, m => d);
```

### Mixed Notation

In mixed notation, you can mix both notations in procedure call; however, the positional notation should precede the named notation.

The following call is legal –

```
findMin(a, b, c, m => d);
```

However, this is not legal:

```
findMin(x => a, b, c, d);
```

## RELATIONSHIP BETWEEN SQL AND PL/SQL:

**Introduction SQL: Structured Query Language (SQL)** is a standard Database language that is used to create, maintain and retrieve the relational database. The advantages of SQL are:

- [SQL](#) could be a high-level language that has a larger degree of abstraction than procedural languages.
- It enables the systems personnel end-users to deal with several database management systems where it is available.
- Portability. Such porting could be required when the underlying [DBMS](#) needs to be upgraded or changed.
- SQL specifies what's needed and not however it ought to be done.

**Introduction to PL/SQL:** [PL/SQL](#) is a block-structured language that enables developers to combine the power of SQL with procedural statements. All the statements of a block are passed to the oracle engine all at once which increases processing speed and decreases the traffic. PL/SQL stands for “Procedural Language extensions to SQL.” PL/SQL is a database-oriented programming language that extends SQL with procedural capabilities. It was developed by Oracle Corporation in the early 90s to boost the capabilities of SQL. PL/SQL adds selective (i.e. if...then...else...) and iterative constructs (ie. loops) to SQL. PL/SQL is most helpful put in writing triggers and keeping procedures. Stored procedures square measure units of procedural code keep during a compiled type inside the info. The advantages of PL/SQL are as below:

- **Block structures:** It consists of blocks of code, which can be nested within each other. Each block forms a unit of a task or a logical module. PL/SQL blocks are often kept within the info and reused.
- **Procedural language capability:** It consists of procedural language constructs like conditional statements (if-else statements) and loops like (FOR loops).
- **Better performance:** PL/SQL engine processes multiple SQL statements at the same time as one block, thereby reducing network traffic.

- **Error handling:** PL/SQL handles errors or exceptions effectively throughout the execution of a PL/SQL program. Once an associate degree exception is caught, specific actions can be taken depending upon the type of the exception or it can be displayed to the user with a message.

#### Comparisons of SQL and PLSQL:

Sr. No.	Basis of Comparison	SQL	PL/SQL
1.	Definition	It is a database <a href="#">Structured Query Language</a> .	It is a database programming language using SQL.
2.	Variables	Variables are not available in SQL.	Variables, constraints, and data types features are available in PL/SQL.
3.	Control structures	No Supported Control Structures like for loop, if, and other.	Control Structures are available like, for loop, while loop, if, and other.
4.	Nature of Orientation	It is a Data-oriented language.	It is an application-oriented language.
5.	Operations	Query performs the single operation in SQL.	PL/SQL block performs Group of Operation as a single block resulting in reduced network traffic.
6.	Declarative/ Procedural Language	SQL is a declarative language.	PL/SQL is a procedural language.
7.	Embed	SQL can be embedded in PL/SQL.	PL/SQL can't be embedded in SQL.
8.	Interaction with Server	It directly interacts with the database server.	It does not interact directly with the database server.
9.	Exception Handling	SQL does not provide error and exception handling.	PL/SQL provides error and exception handling.

<b>Sr. No.</b>	<b>Basis of Comparison</b>	<b>SQL</b>	<b>PL/SQL</b>
<b>10.</b>	<b>Writes</b>	It is used to write queries using DDL (Data Definition Language) and DML (Data Manipulation Language) statements.	The code blocks, functions, procedures triggers, and packages can be written using PL/SQL.
<b>11.</b>	<b>Processing Speed</b>	SQL does not offer a high processing speed for voluminous data.	PL/SQL offers a high processing speed for voluminous data.
<b>12.</b>	<b>Application</b>	You can fetch, alter, add, delete, or manipulate data in a database using SQL.	You can use PL/SQL to develop applications that show information from SQL in a logical manner.