# FILE PROCESSING SYSTEMS

- In earlier days,to keep the informations on a computer is to store it in operating system files.
- To allow user to manipulate the information system has a number of application programs that manipulates the files.
- The typical file processing system is supported by a conventional operating system.
- The system stores permanent record in various files,and it needs different application programs to extract records from and add records to appropriate files.
- File processing system is a way of storing,retrieving and manipulating data which is present in various files.
- All files are grouped based on their categories.The files names are very related to each other and arranged properly to easily access the files.
- In file processing system,if one needs to insert,delete,modify,store or update data,one must know the entire hierarchy of the files.

## ADVANTAGES OF FILE PROCESSING SYSTEMS

- Cost friendly:-

  There is a very minimal to no set up and usage fee for file processing system.

- Easy to use:-

  File systems require very basic learning and understanding,hence can be easily used.

- High scalability:-

  One can very easily switch from smaller to large files as per his needs.

## DISADVANTAGES OF FILE PROCESSING SYSTEMS

- Slow access time:-

  Direct access of files is very difficult and one needs to know the entire hierarchy of folders to get to a specific file.This involves a lot of time.

- Presence of redundant data:-

The same data can be present in two or more files which takes up more disc space.

- Inconsistent data:-

  Due to data redundancy,same data stored at different places might not match to each other.

- Lack of atomicity:-

  Operations performed in the database must be atomis i.e. either the operation takes place as a whole or does not take place at all.

- Problem in concurrent access:-

  When a number of users operates on a common data in database at the same time then anomalies arises,due to lack of concurrency control.

## DBMS APPROACH

## ADVANTAGES OF DBMS

- Data independence:
  Application programs are insulated from the details of data representation and storage.
- Efficient data access:

  A Dbms utilizes various sophisticated techniques to store and retrieve data efficiently.

- Data integrity and security:

  DBMS can enforce integrity constaints

  Eg:primary key,not null,check etc.

- Data Administration:

  It is responsible for organizing the data representation to minimize redundancy and for proper storage of data to make retrieval efficient.

- Concurrent access and crash recovery:

  Concurrent access to data are provided such that users think of the data being accessed by only one user at a time.Also Dbms protects users from the effects of system failure.

- Reduced application development time:

  Dbms supports important functions that are common to many applications accessing data in the dbms.

## DISADVANTAGES OF DBMS

- Complexity:

  Database designers,developers,database administrators and end users must understand the functionality to take full advantage of it.

- Size:

  The complexity and breadth of functionality makes the Dbms an extremely large piece of software,occupying many megabytes of disk space.Requires substantial amounts of memory to run efficiently.

- Higher impact of a failure:

  The centralization of resources increase the vulnerability of the system.since all users and applications rely on the availability of the Dbms the failure of any component can bring operations to a halt.

- Performance:

  A file based sytem is written for a specific application,such as invoicing.so, performance is generally very good.

- Cost of Dbms:

  The cost of Dbms varies significally,depending on the environment and functionality provided.Recurrent annual maintenance cost.

## DESCRIBING AND STORING THE DATA

- A **data model** is a collection of high-level data description constructs that hide many low-level storage details.
- A **semantic data model** is a more abstract, high-level data model that makes it easier for a user to come up with a good initial description of the data in an enterprise.

- A database design in terms of a semantic model serves as a useful starting point and is subsequently translated into a database design in terms of the data model the DBMS actually supports.
- A widely used semantic data model called the **entity-relationship (ER) model** allows us to pictorially denote entities and the relationships among them

## 1. The Relational Model:

- The central data description construct in this model is **relation**, which can be thought of as a set of **records**.
- A description of data in terms of a data model is called a **schema**.
- The schema for a relation specifies its name, the name of each field or attribute or column.
- Example: student information in a university database my be stored in a relation with the following schema (with 5 fields):
    - Students(*sid*: string, *name*: string, *login:* string, *age:* integer, *gpa*: real)
    - An example *instance* of the Students relation:

| *sid* | *name* | *login* | *age* | *gpa* |
|---|---|---|---|---|
| 53666 | Jones | jones@cs | 18 | 3.4 |
| 53588 | Smith | smith@ee | 18 | 3.2 |

    - Each row in the Students relation is a record that describes a student. Every row follows the schema of the Student relation and schema can therefore be regarded as a template for describing a student.
    - We can make the description of a collection of students more precise by specifying **integrity constraints,** which are conditions that the records in a relation must staisfy.
    - Other notable models: hierarchial model, network model, object-oriented model, and the object-relational model.

## 2. LEVELS OF ABSTRACTION IN A DBMS:

- A **data definition language** (DDL) is used to define the external and conceptual schemas.
- Information about conceptual, external, and physical schemas is stored in the **system catalogs**.

- Any given database has exactly one *conceptual schema* and one *physical schema* because it has just one set of stored relations, but it may have several *external schemas*, each tailored to a particular group of users.

## 1.Conceptual Schema

- The conceptual schema (sometimes called the **logical schema**) describes the stored data in terms of the data model of the DBMS.
- Relations contain information about *entities* and *relationships*

## 2.Physical Schema

- The physical schema specifies additional storage detail, summarizes how the relations described in conceptual schema are actually stored on secondary storage devices such as disks and tapes.
- Decide what file organizations to use to store the relations, then create **indexes** to speed up data retrieval operations.

## 3.External Schema

- External schemas allow data access to be customized and authorized at the level of individual user or groups of users.
- Each external schema consists of a collection of **views** and relations from the conceptual schema.
- A **view** is conceptually a relation, but the records in a view are not stored in the DBMS. The records are computed using a definition for the view, in terms of relations stored in the DBMS.
- The external schema design is guided by the end user requirements.

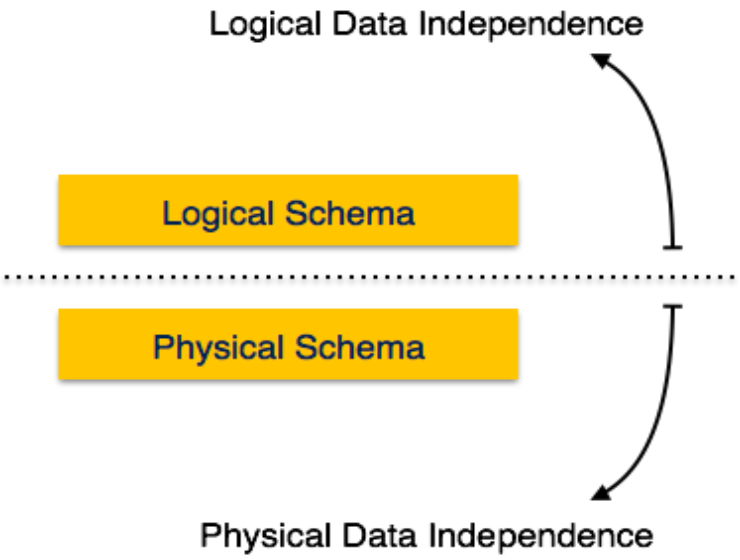
## 3. DATA INDEPENDENCE:

- Data independence is achieved through the use of the three levels of data abstraction; in particular, the conceptual schema and the external schema provide distinct benefits in this area.
- Logical data Independence:
    - Users can be shielded from changes in the logical structure of the data, or changes in the choice of relations to be stored.
    - Example: Student_public, Student_private => create levels using views in external schema
- Physical data independence:
    - The conceptual schema insulates users from changes in the physical storage of the data.

- The conceptual schema hides details such as how the data is actually laid out on disk, the file structure, and the choice of indexes.

# DATA INDEPENDENCE

One of the highest advantage of databases is data independence. The ability to modify schema definition in one level without affecting schema of that definition in the next higher level is called data independence. It also means we can change the structure of a database without affecting the data required by users and programs. This feature was not available in the file-oriented approach. There are two levels of data independence, they are Physical data independence and Logical data independence.




## PHYSICAL DATA INDEPENDENCE

The ability to modify the physical schema without causing application programs to be rewritten. Modifications at the physical level are occasionally necessary to improve performance. It means we change the physical storage/level without affecting the conceptual or external view of the data. The new changes are absorbed by mapping techniques.

## LOGICAL DATA INDEPENDENCE

The ability to modify the logical schema without causing application programs to be rewritten. Modifications at the logical level are necessary whenever the logical structure of the database is altered (for example, when money-market accounts are added to banking system). Logical Data independence means if we add some new columns or remove some columns from table then the user view and programs should not change. For example : consider two users A & B. Both are selecting the fields "EmployeeNumber" and "EmployeeName". If user B adds a new column (e.g. salary) to his table, it will not affect the external view for user A, though the internal schema of the database has been changed for both users A & B.

## NOTE

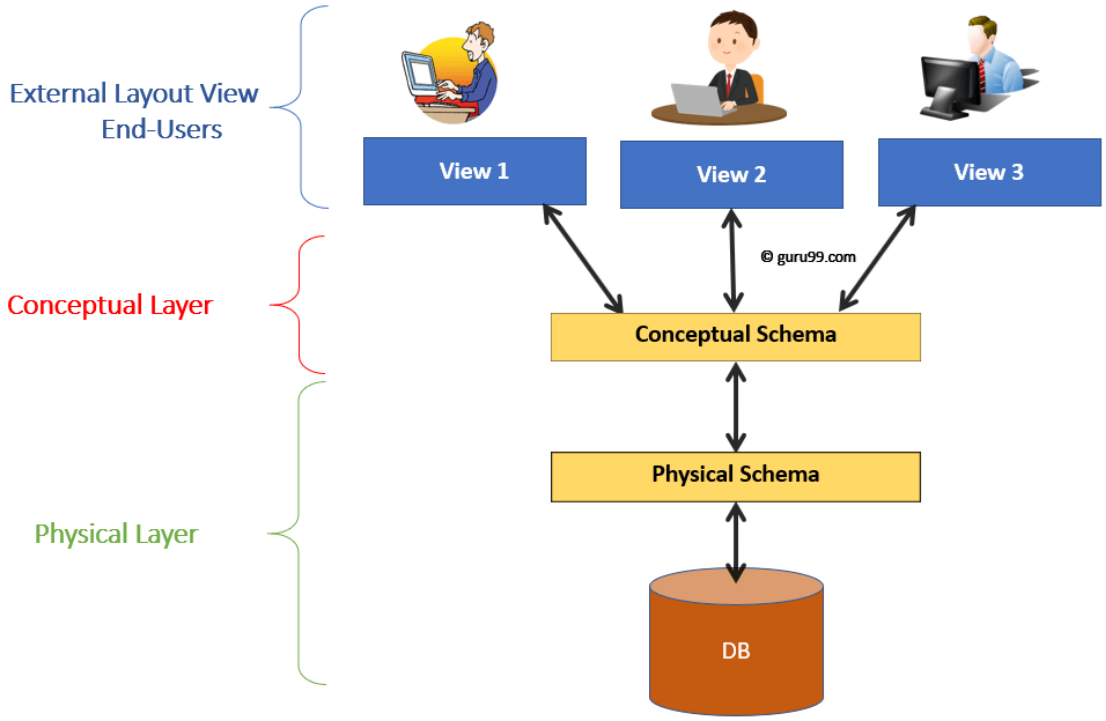
Physical data independence is present in most databases and file environment in which hardware storage of encoding, exact location of data on disk, merging of records, so on this are hidden from user. Logical data independence is more difficult to achieve than physical data independence, since application programs are heavily dependent on the logical structure of the data that they access.

# LEVEL OF ABSTARCTION IN DBMS

Database systems comprise of complex data structures. Thus, to make the system efficient for retrieval of data and reduce the complexity of the users, developers use the method of Data Abstraction.

There are mainly three levels of data abstraction :

- Internal/Physical Level : Actual physical storage structure and access paths.
- Conceptual/Logical Level : Structure and constraints for the entire database.
- External/View level : Describes various user views.



## INTERNAL SCHEMA/LEVEL

The internal schema defines the physical storage structure of the database. Essentially, the physical schema summarizes how the relations described in the conceptual schema are actually stored on secondary storage devices such as disks and tapes. It is a very low-level representation of the entire database. It contains multiple occurrences of multiple types of internal record. In the ANSI term, it is also called "stored record'.

**Facts about Physical schema :**

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

## CONCEPTUAL SCHEMA/ LEVEL

The conceptual schema describes the Database structure of the whole database for the community of users. This schema hides information about the physical storage structures and focuses on describing data types, entities, relationships, etc. This logical level comes between the user level and physical storage view. However, there is only single conceptual view of a single database.

**Facts about Conceptual schema :**

- Describes the stored data in terms of data model in DBMS.
- Defines all database entities, their attributes, and their relationships.
- Security and integrity information.
- In the conceptual level, the data available to a user must be contained in or derivable from the physical level.
- The process of arriving into good conceptual schema is called conceptual database design.

## EXTERNAL SCHEMA/ LEVEL

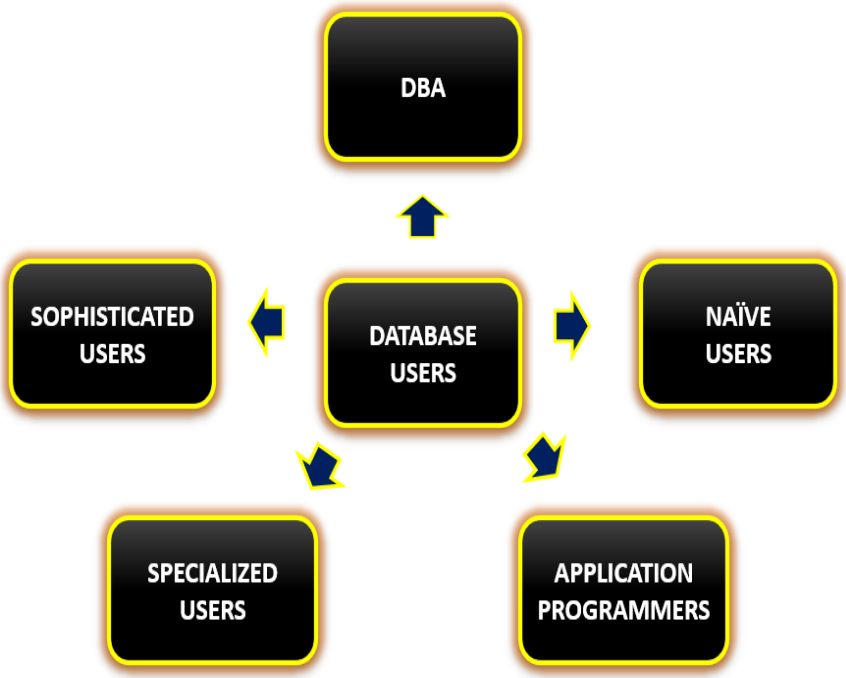
The external schema describes the part of the database which specific user is interested in. It hides the unrelated details of the database from the user. There may be "n" number of external views for each database. Each external view is defined using an external schema, which consists of definitions of various types of external record of that specific view. An external view is just the content of the database as it is seen by some specific particular user. For example, a user from the sales department will see only sales related data.

**Facts about External schema :**

- External schema which usually are also in terms of the data model of DBMS.
- Allows data access to customize at the level of individual user or group of users.
- External level is only related to the data which is viewed by specific end users.
- Any given database can have only one conceptual schema and physical schema, but can have several external schemas, each tailored to particular group of users.
- This level includes some external schemas.
- External schema level is nearest to the user.
- The external schema describes the segment of the database which is needed for a certain user group and hides the remaining details from the database from the specific user group.
- The external schema design is guided by end user requirements.

# DATABASE USERS

Database users are the ones who really use and take the benefits of the database. There will be different types of users depending on their needs and way of accessing the database. There are four different types of database-system users, differentiated by the way they expect to interact with the system. They are,



## NAIVE USERS

Naive users are unsophisticated users who interact with the system by invoking one of the application programs that have been written previously. They don't have any DBMS knowledge but they frequently use the data base applications in their daily life to get the desired results. For examples, Railway's ticket booking users are naïve users. Clerks in any bank is a naïve user because they don't have any DBMS knowledge but they still use the database and perform their given task.

## APPLICATION PROGRAMMERS

Application programmers are computer professionals who write application programs to access the stored data. These programs could be written in Programming languages such as Visual Basic, Developer, C, FORTRAN, COBOL etc. Application programmers can choose from many tools to develop user interfaces. Rapid application development ( RAD ) tools are tools that enable an application programmer to construct forms and reports with minimal programming effort.

## SOPHISTICATED USERS

Sophisticated users interact with the system without writing programs. They can be engineers, scientists, business analyst, who are familiar with the database. They can develop their own data base applications according to their requirement. They don't write the program code but they interact the data base by writing SQL queries directly through the query processor.

## SPECIALIZED USERS

Specialized users are sophisticated users who write specialized database applications that do not fit into the traditional data-processing framework. specialized users create applications such as computer-aided design, expert systems, and knowledge bases that can store more complicated data types than any simple application program.

## DATABASE ADMINISTRATOR (DBA)

One of the main reasons for using DBMS is to have central control of both the data and the programs that access those data. A person who has such central control over the system is called a database administrator ( DBA ). This may be one person or a group of people in an organization responsible for authorizing access to the database, monitoring its use and managing all of the resources to support the use of the entire database system. DBA is also responsible for providing security to the data base and he allows only the authorized users to access/modify the data base.

The functions of a DBA include :

**Schema definition :**

The DBA definition the logical Schema of the database. A Schema refers to the overall logical structure of the database. According to this schema, database will be developed to store required data for an organization. The DBA creates the original database schema by executing a set of data definition statements in the DDL.

**Storage structure and access - method definition :**

The DBA decides how the data is to be represented in the stored database.

**Granting of authorization for data access :**

The DBA determines which user needs access to which part of the database. According to this, various types of authorizations are granted to different users. By granting different types of authorization, the database administrator can regulate which parts of the database various users can access. The authorization information is kept in a special system structure that the database system consults whenever someone attempts to access the data in the system.

**Routine maintenance:**

A DBA often collaborates on the initial installation and configuration of a new Oracle, SQL Server etc database. The system administrator sets up hardware and deploys the operating system for the database server, then the DBA installs the database software and configures it for use. As updates and patches are required, the DBA handles this on-going maintenance. Examples of the database administrator's routine maintenance activities are:

- Periodically backing up the database, either onto tapes or onto remote servers, to prevent loss of data in case of disasters such as flooding.
- Ensuring that enough free disk space is available for normal operations, and upgrading disk space as required.
- Monitoring jobs running on the database and ensuring that performance is not degraded by very expensive tasks submitted by some users.

**Backup and Recovery :**

Database should not be lost or damaged. The DBA ensures this periodically backing up the database on magnetic tapes or remote servers. In case of failure, such as virus attack database is recovered from this backup.

# STRUCTURE OF DBMS

- The DBMS accepts SQL commands generated from a variety of user interfaces produces query evaluation plans, executes plans against the database and then

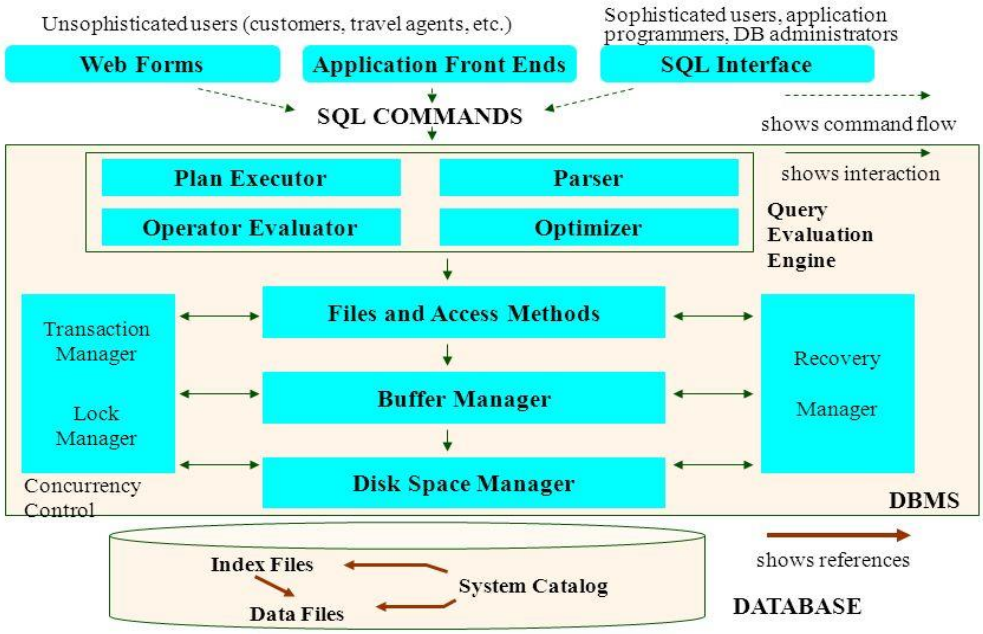



Figure 1.3 Architecture of a DBMS



returns the answer.

- SQL commands can be embedded in host language application programs.
  Eg : Java ,COBOL programs etc.
- When the user issues a query, the query is presented to query optimizer which uses information about how the data is stored to produce an efficient execution plan for evaluating the query.
- An execution plan is the blue print for evaluating a query usually represented as a tree of relational operators.
- The files and access methods layer code sits on top of the buffer manager, which brings pages in from disk to main memory ct." needed in response to read requests.
- The lowest layer of DBMS software deals with management of space on disc, where the data is stored.
- Higher layers allocate, deallocate, read and write pages through the layer called disc space manager.

- DBMS supports concurrency and crash recover by scheduling user requests and maintaining a log of all changes to the database.
- DBMS component associated with concurrency control and recovery includes transaction manager, lock manager and recovery manager
- Transaction manager which ensures that transaction request and release locks according to suitable locking protocol and schedules execution transactions.
- Lock Manager which keeps track of request for locks and grants locks on database objects when they become available.
- Recovery manager responsible for restoring the system to a consistent state after a crash.
- The disk space manager, buffer manager, and file and access method layers must interact with these components.

# Introduction to Database Design

**What is Database Design?**

**Database Design** is a collection of processes that facilitate the designing, development, implementation and maintenance of enterprise data management systems. Properly designed database are easy to maintain, improves data consistency and are cost effective in terms of disk storage space. The database designer decides how the data elements correlate and what data must be stored.

The main objectives of database design in DBMS are to produce logical and physical designs models of the proposed database system.

The logical model concentrates on the data requirements and the data to be stored independent of physical considerations. It does not concern itself with how the data will be stored or where it will be stored physically.

The physical data design model involves translating the logical DB design of the database onto physical media using hardware resources and software systems such as database management systems (DBMS).

**Major Steps in Database Design**

1. **Requirements Analysis:** Talk to the potential users! Understand what data is to be stored, and what operations and requirements are desired.
2. **Conceptual Database Design:** Develop a high-level description of the data and constraints (we will use the *ER data model*)
3. **Logical Database Design:** Convert the conceptual model to a schema in the chosen data model of the DBMS. For a relational database, this means converting the conceptual to a relational schema (logical schema).
4. **Schema Refinement:** Look for potential problems in the original choice of schema and try to redesign.
5. **Physical Database Design:** Direct the DBMS into choice of underlying data layout (e.g., indexes and clustering) in hopes of optimizing the performance.
6. **Applications and Security Design:** How will the underlying database interact with surrounding applications.

**Entity-Relationship Data Model (ER)**

- **entity**: An entity is a real-world object or concept which is distinguishable from other objects. It may be something tangible, such as a particular student or building. It may also be somewhat more conceptual, such as CS A-341, or an email address.
- **attributes**: These are used to describe a particular entity (e.g. name, SS#, height).
- **domain**: Each attribute comes from a specified domain (e.g., name may be a 20 character string; SS# is a nine-digit integer)
- **entity set**: a collection of similar entities (i.e., those which are distinguished using the same set of attributes. As an example, *I* may be an entity, whereas *Faculty* might be an entity set to which I belong. Note that entity sets need not be disjoint. I may also be a member of *Staff* or of *Softball Players*.

- **key**: a minimal set of attributes for an entity set, such that each entity in the set can be uniquely identified. In some cases, there may be a single attribute (such as SS#) which serves as a key, but in some models you might need multiple attributes as a key. There may be several possible candidate keys. We will generally designate one such key as the **primary key**.

## ER diagrams

It is often helpful to visualize an ER model via a diagram. There are many variant conventions for such diagrams; we will adapt the one used in the text.

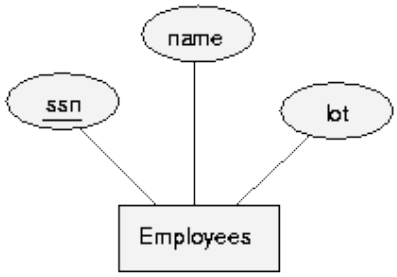
***Diagram conventions***

- An entity set is drawn as a rectangle.


- Attributes are drawn as ovals.


- Attributes which belong to the primary key are underlined.


***Diagram example***




## Relationships

A **relationship** is an association among two or more entities. *The relationship must be uniquely identified by the participating entities*.

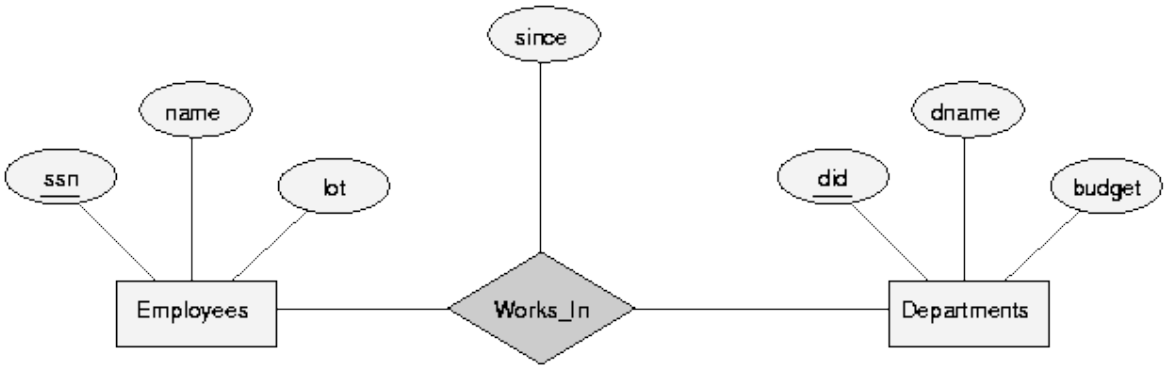
A relationship can also have **descriptive attributes**, to record additional information about the relationship (as opposed to about any one participating entity).

For example, I am an entity, as is the Department of Math/CS. A relationship exists in that I work in that department.

Similarly, a **relationship set** is a set of "similar" relationships (the similarity is based on the type of underlying entities involved in each such relationship) For example, if you have an entity set *Employees* and another entity set *Departments*, you might define a relationship set *Works_In* which associates members of those two entity sets.

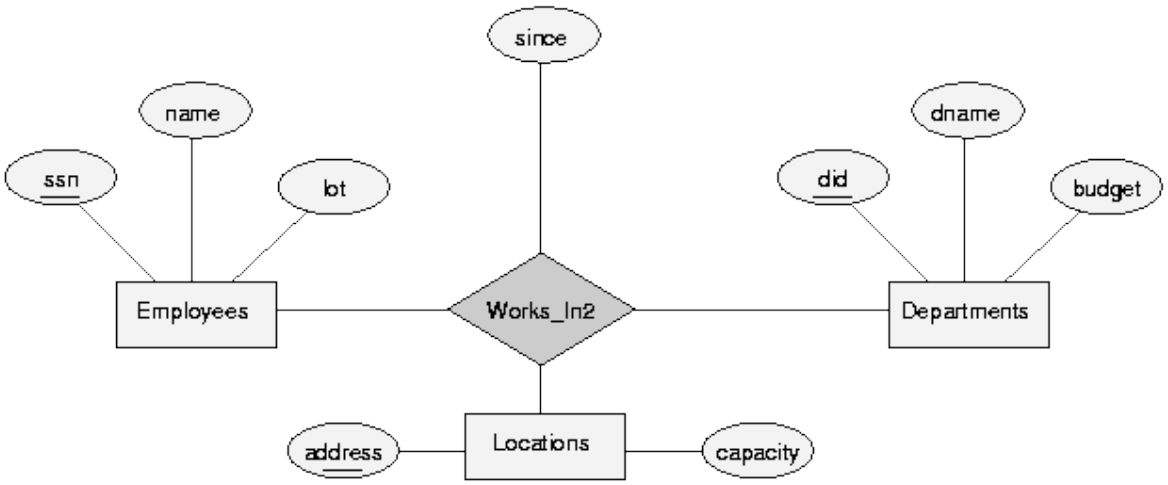
- In the ER diagrams, we will draw a relationship set is drawn as a shaded diamond.


***Diagram example***



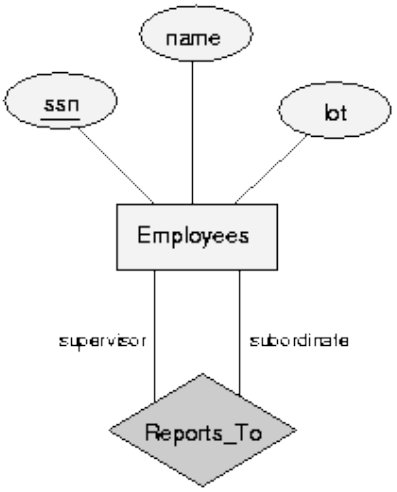

***Ternary Relationship Set***

A relationship set need not be an association of precisely two entities; it can involve three or more when applicable. Here is another example from the text, in which a store has multiple locations.



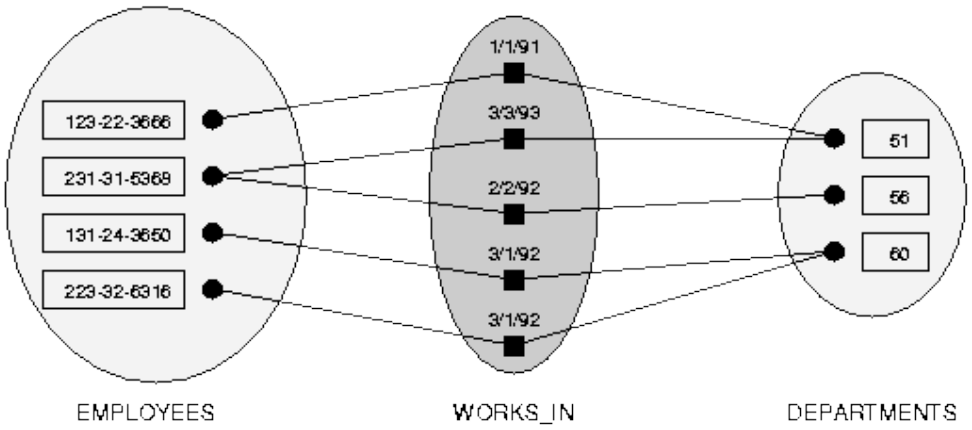

***Using several entities from same entity set***

A relationship might associate several entities from the same underlying entity set, such as in the following example, *Reports_To*. In this case, an additional **role indicator** (e.g., "supervisor") is used in the diagram to further distinguish the two similar entities.




**Specifying additional constraints**

If you took a 'snapshot' of the relationship set at some instant in time, we will call this an **instance**. It can be diagrammed separately.




**Relationship Types**

- **one-to-one:**

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

- **one-to-many:**

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.

- **many-to-one:**

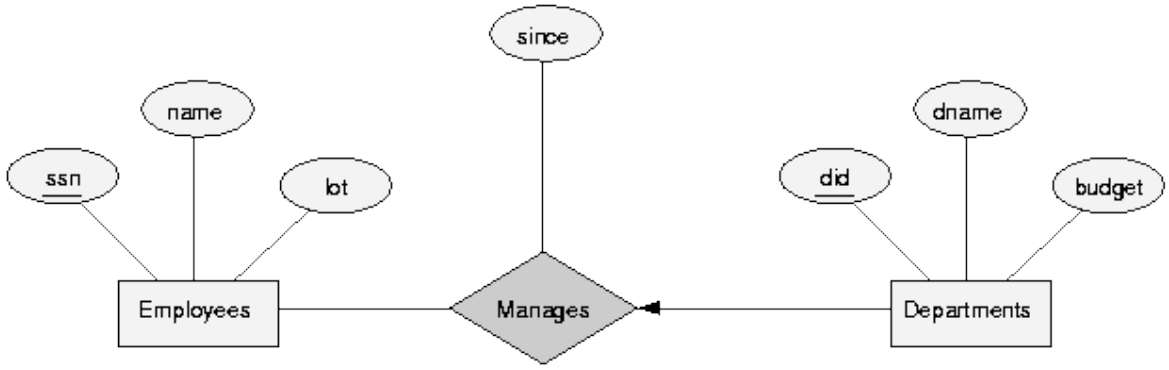
When more than one instance of the entity on the left, and only one instance of an entity on the right associates with the relationship then it is known as a many-to-one relationship.

- **many-to-many:**

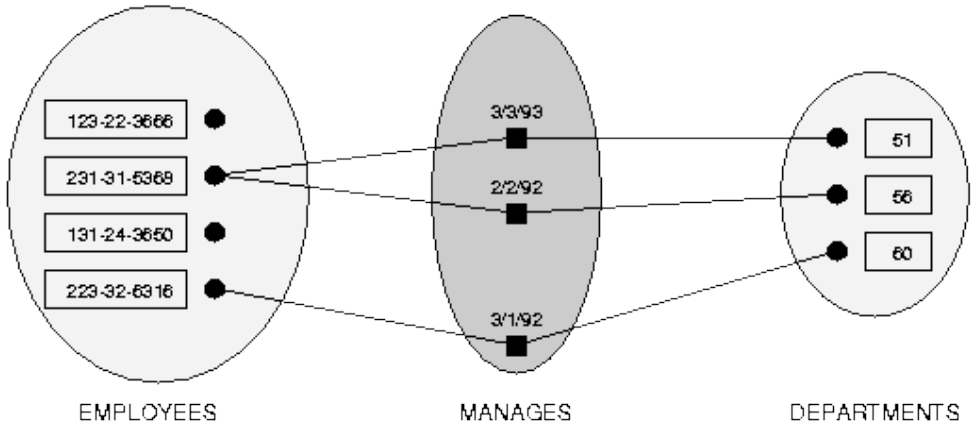
When more than one instance of the entity on the left, and more than one instance of an entity on the right associates with the relationship then it is known as a many-to-many relationship.

Sometimes, an additional constraint exists for a given relationship set, that any entity from one of the associated sets appears in at most one such relationship. For example, consider a relationship set "Manages" which associates departments with employees. If a department cannot have more than one manager, this is an example of a one-to-many relationship set (it may be that an individual manages multiple departments).

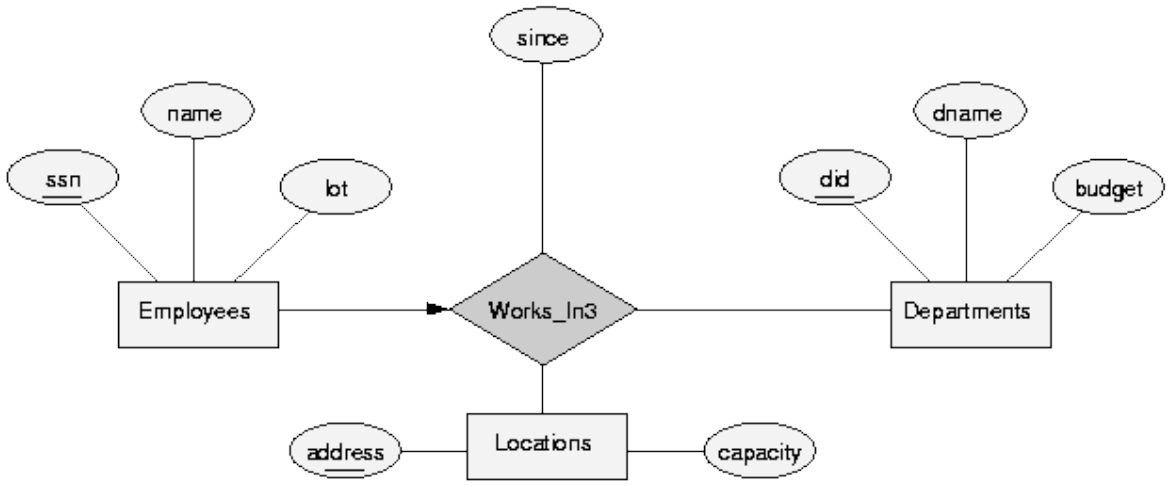
This type of constraint is called a **key constraint**. It is represented in the ER diagrams by drawing an arrow from an entity set E to a relationship set R when each entity in an instance of E appears in at most one relationship in (a corresponding instance of) R.



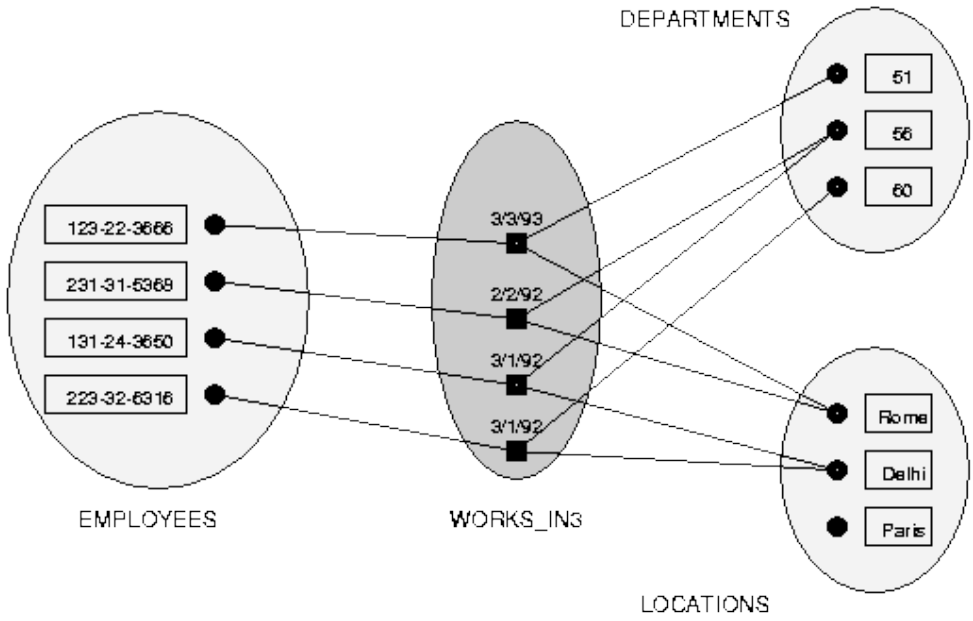

An instance of this relationship is given .




If both entity sets of a relationship set have key constraints, we would call this a "one-to-one" relationship set. In general, note that key constraints can apply to relationships between more than two entities, as in the following example.



An instance of this relationship:




***Weak Entities***

There are times you might wish to define an entity set even though its attributes do not formally contain a key (recall the definition for a key).

Usually, this is the case only because the information represented in such an entity set is only interesting when combined through an **identifying relationship set** with another entity set we call the **identifying owner**.

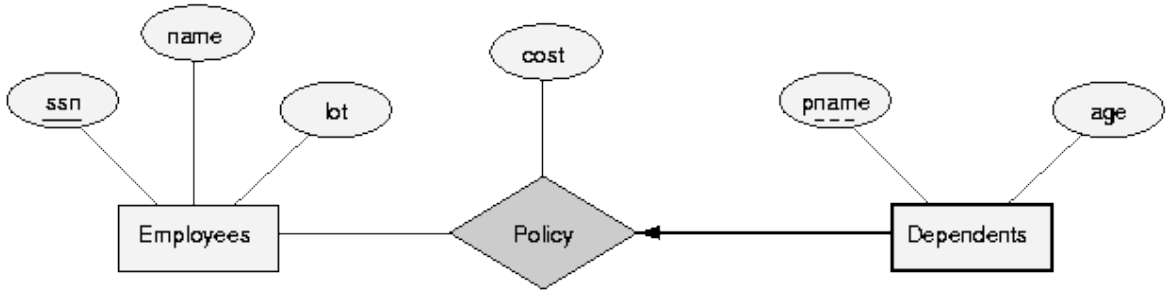
We will call such a set a **weak entity set**, and insist on the following:

- The weak entity set must exhibit a key constraint with respect to the identifying relationship set.
- The weak entity set must have total participation in the identifying relationship set.

Together, this assures us that we can uniquely identify each entity from the weak set by considering the primary key of its identifying owner together with a **partial key** from the weak entity.

In this ER diagrams, we will represent a weak entity set by outlining the entity and the identifying relationship set with dark lines. The required key constraint and total participation

are diagrammed with existing conventions. So underline the partial key with a dotted line.

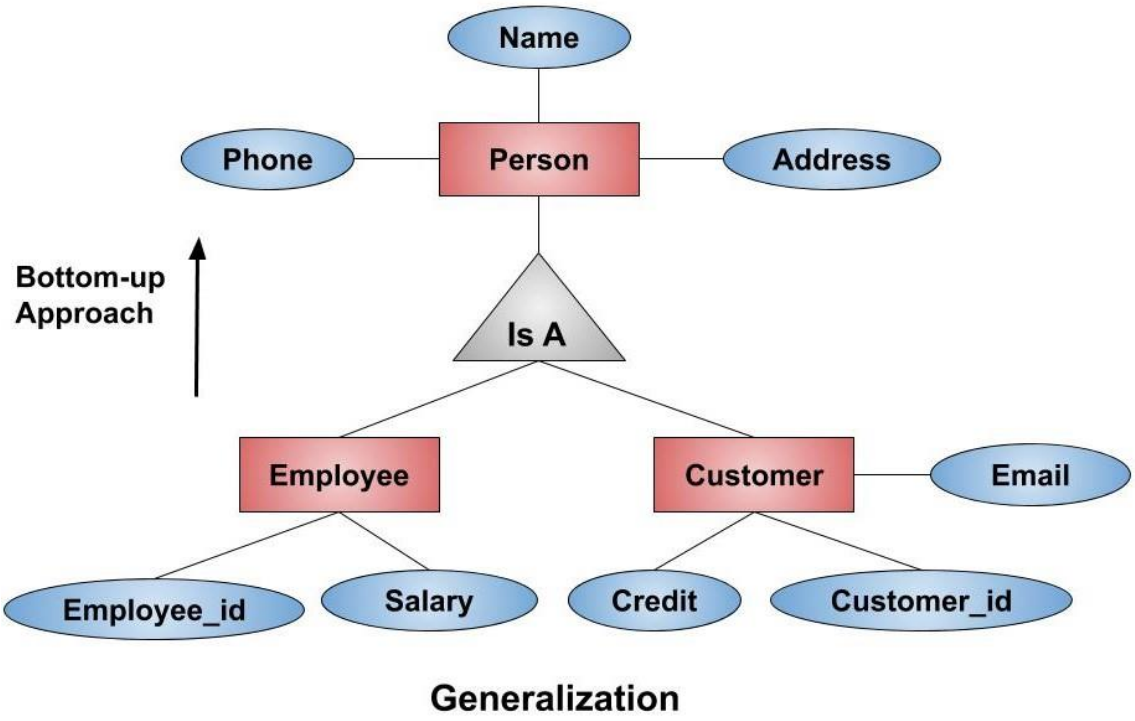



**Additional Features of ER Model-Class Hierarchies**

### Generalization

Generalization is a bottom-up approach in which multiple lower-level entities are combined to form a single higher-level entity. Generalization is usually used to find common attributes among entities to form a generalized entity. It can also be thought of as the opposite of specialization.

The following enhanced entity relationship diagram expresses entities in a hierarchical database to demonstrate generalization:

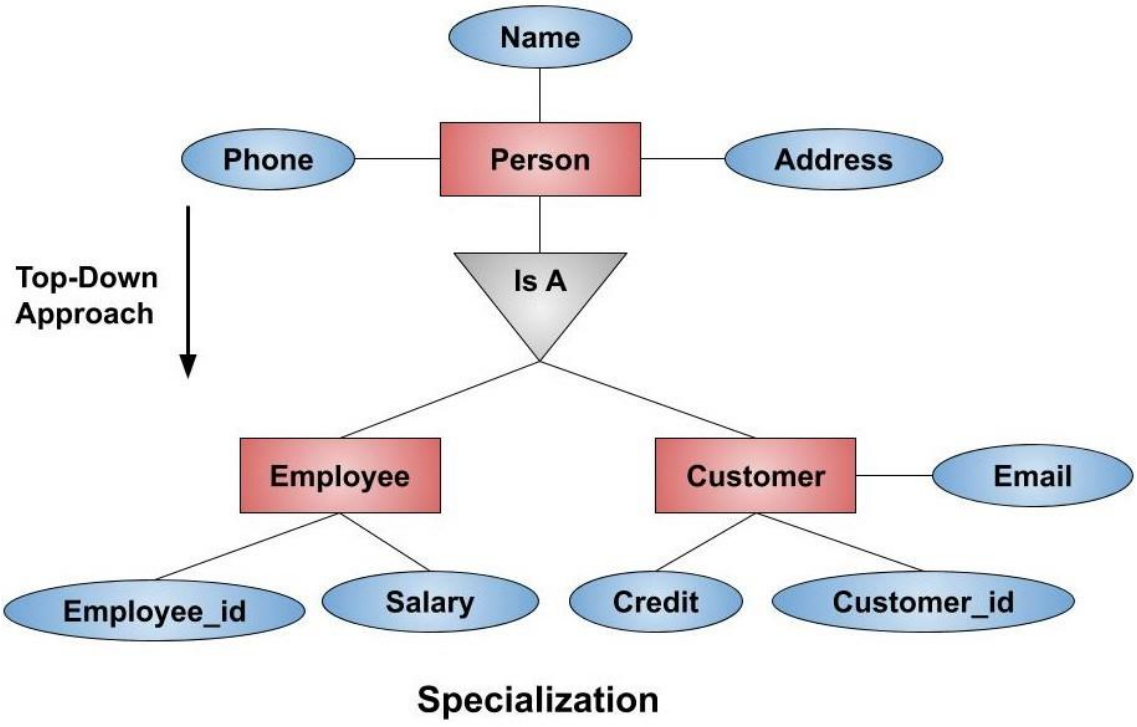



Generalization

**Specialization**

Specialization is a top-down approach in which a higher-level entity is divided into multiple *specialized* lower-level entities. In addition to sharing the attributes of the higher-level entity, these lower-level entities have *specific* attributes of their own. Specialization is usually used to find subsets of an entity that has a few different or additional attributes.

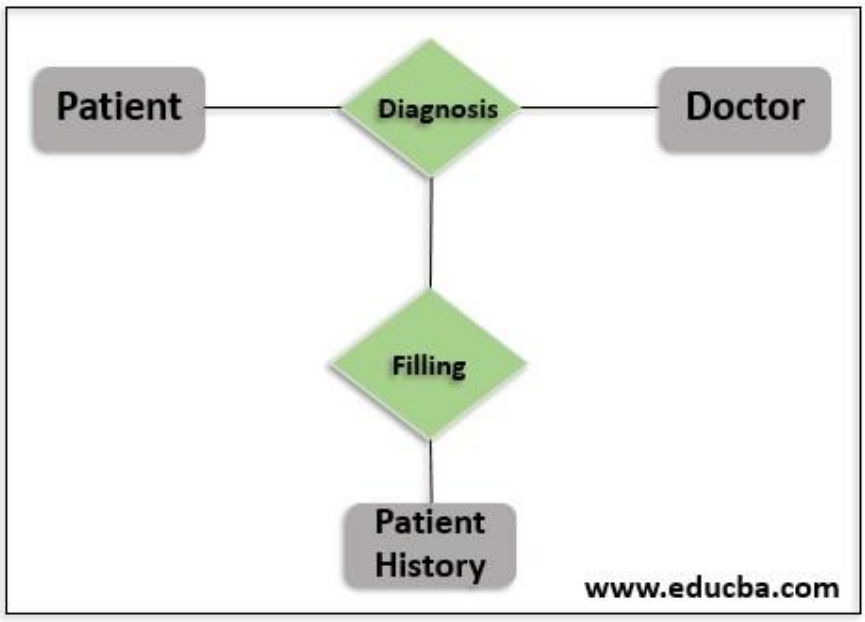
The following enhanced entity relationship diagram expresses the entities in a hierarchical database to demonstrate specialization:




Specialization

**Aggregation in DBMS**

Aggregation refers to the process by which entities are combined to form a single meaningful entity. The specific entities are combined because they do not make sense on their own. To establish a single entity, aggregation creates a relationship that combines these entities. The resulting entity makes sense because it enables the system to function well.


Patient
Diagnosis
Doctor
Filling
Patient History
www.educba.com

# 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**

DBMS Integrity Constraints

## 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.

## 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.

## 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.

## 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.

**What is a Primary Key**

A Primary Key is the minimal set of attributes of a table that has the task to uniquely identify the rows, or we can say the tuples of the given particular table.

Use of Primary Key

As defined above, a primary key is used to uniquely identify the rows of a table. Thus, a row that needs to be uniquely identified, the key constraint is set as the Primary key to that particular field. A primary key can never have a NULL value because the use of the primary key is to identify a value uniquely, but if no value will be there, how could it sustain. Thus, the field set with the primary key constraint cannot be NULL. Also, it all depends on the user that the user can add or delete the key if applied.

Creation :

CREATE TABLE STUDENT_DETAIL (

Roll_no int NOT NULL PRIMARY KEY,

Name varchar (200) NOT NULL,

Marks int NOT NULL

} ;

Removing Primary Key

It is also possible to delete the set primary key from an attribute using ALTER and DROP commands.

ALTER TABLE STUDENT_DETAIL DROP PRIMARY KEY ;

Adding Primary Key after creating the table

In order to set the primary key after creating a table, use the ALTER command and add the primary key constraint to do so. The syntax is shown below:

ALTER TABLE STUDENT_DETAIL

ADD CONSTRAINT PK_STUDENT_DETAIL PRIMARY KEY (Roll_no, Name);

**What is a Foreign Key**

A foreign key is the one that is used to link two tables together via the primary key. It means the columns of one table points to the primary key attribute of the other table. It further means that if any attribute is set as a primary key attribute will work in another table as a foreign key attribute. But one should know that a foreign key has nothing to do with the primary key.

Creating Foreign Key constraint

On CREATE TABLE

Below is the syntax that will make us learn the creation of a foreign key in a table:

```
CREATE TABLE Department (
   Dept_name varchar (120) NOT NULL,
   Stud_Id int,
    FOREIGN KEY (Stud_Id) REFERENCES Student (Stud_Id)
);
```

Following is the syntax for creating a foreign key constraint on ALTER TABLE:

ALTER TABLE Department

ADD FOREIGN KEY (Stud_Id) REFERENCES Student (Stud_Id);

Dropping Foreign Key

In order to delete a foreign key, there is a below-described syntax that can be used:

ALTER TABLE Department

DROP FOREIGN KEY FK_StudentDepartment;

Constraints are the rules enforced on the data columns of a table. These are used to limit the type of data that can go into a table. This ensures the accuracy and reliability of the data in the database.

Constraints could be either on a column level or a table level. The column level constraints are applied only to one column, whereas the table level constraints are applied to the whole table.

Following are some of the most commonly used constraints available in SQL. These constraints have already been discussed in SQL - RDBMS Concepts chapter, but it's worth to revise them at this point.

**NOT NULL Constraint** − Ensures that a column cannot have NULL value.

**DEFAULT Constraint** − Provides a default value for a column when none is specified.

**UNIQUE Constraint** − Ensures that all values in a column are different.

**PRIMARY Key** − Uniquely identifies each row/record in a database table.

**FOREIGN Key** − Uniquely identifies a row/record in any of the given database table.

**CHECK Constraint** − The CHECK constraint ensures that all the values in a column satisfies certain conditions.

**INDEX** − Used to create and retrieve data from the database very quickly.

Constraints can be specified when a table is created with the CREATE TABLE statement or you can use the ALTER TABLE statement to create constraints even after the table is created.

Dropping Constraints

Any constraint that you have defined can be dropped using the ALTER TABLE command with the DROP CONSTRAINT option.

For example, to drop the primary key constraint in the EMPLOYEES table, you can use the following command.

```
ALTER TABLE EMPLOYEES DROP CONSTRAINT EMPLOYEES_PK;
```

Some implementations may provide shortcuts for dropping certain constraints. For example, to drop the primary key constraint for a table in Oracle, you can use the following command.

```
ALTER TABLE EMPLOYEES DROP PRIMARY KEY;
```

Some implementations allow you to disable constraints. Instead of permanently dropping a constraint from the database, you may want to temporarily disable the constraint and then enable it later.

# CONVERTING ER-MODEL TO RELATIONAL MODEL

There is more than one approach to translating an ER diagram to a relational schema

**1.Entity Set to Table**

**Algorithm**

- Create a table for the entity set.
- Make each attribute of the entity set a field of the table, with an appropriate type.
- Declare the field or fields comprising the primary key

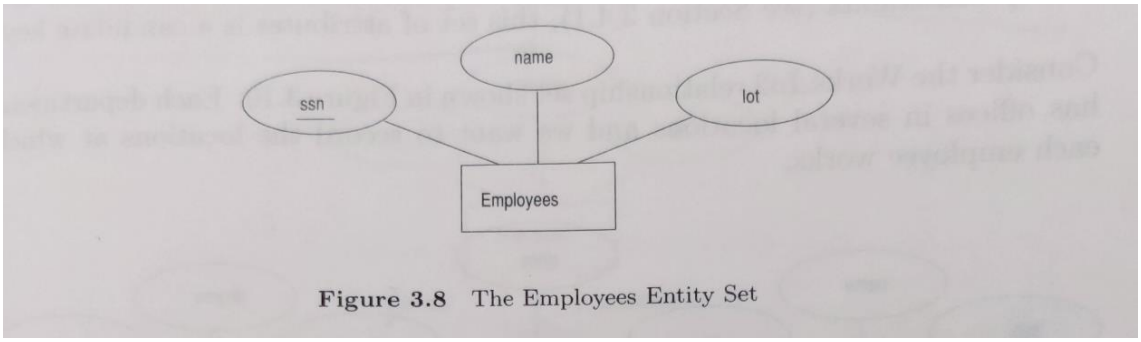



**Figure 3.8** The Employees Entity Set

entities, is shown in Figure 3.9 in a tabular format.

| *ssn* | *name* | *lot* |
|---|---|---|
| 123-22-3666 | Attishoo | 48 |
| 231-31-5368 | Smiley | 22 |
| 131-24-3650 | Smethurst | 35 |

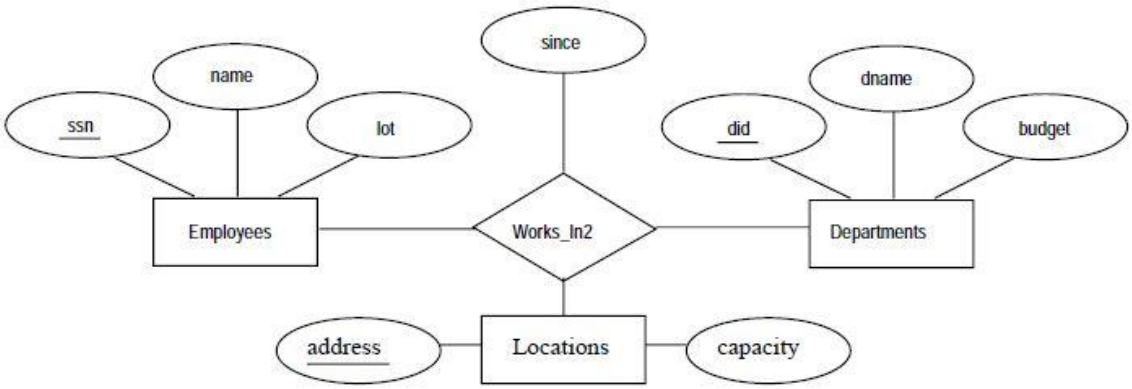
The following SQL statement captures the preceding information, including the domain constraints and key information:

```
CREATE TABLE Employees ( ssn CHAR(11),
                         name CHAR(30),
                         lot INTEGER,
                         PRIMARY KEY (ssn) )
```

**2.Relationship Sets (without Constraints) to Tables**

**Algorithm**

- Create a table for the relationship set.
- Add all primary keys of the participating entity sets as fields of the table.
- Add a field for each attribute of the relationship.
- Declare a primary key using all key fields from the entity sets.
- Declare foreign key constraints for all these fields from the entity sets.

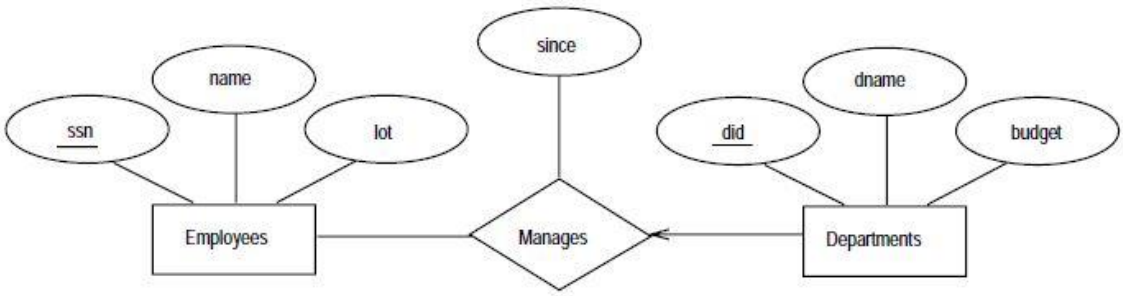



```
CREATE TABLE Works In2 ( ssn CHAR(11),
                         did INTEGER,
                         address CHAR(20),
                         since DATE,
                         PRIMARY KEY (ssn, did, address),
                         FOREIGN KEY (ssn) REFERENCES Employees,
                         FOREIGN KEY (address) REFERENCES Locations,
                         FOREIGN KEY (did) REFERENCES Departments )
```

**Translating Relationship Sets with Key Constraints**

**Algorithm**

- Create a table for the relationship set.
- Add all primary keys of the participating entity sets as fields of the table.
- Add a field for each attribute of the relationship.
- Declare a primary key using the key fields from the source entity set only.
- Declare foreign key constraints for all the fields from the source and target entity sets.




```
CREATE TABLE Manages ( ssn CHAR(11),

                       did INTEGER,
                       since DATE,
                       PRIMARY KEY (did),
                       FOREIGN KEY (ssn) REFERENCES Employees,
                       FOREIGN KEY (did) REFERENCES Departments )
```

**Mapping relationship sets (with key constraints, 2nd method)**

**Algorithm**

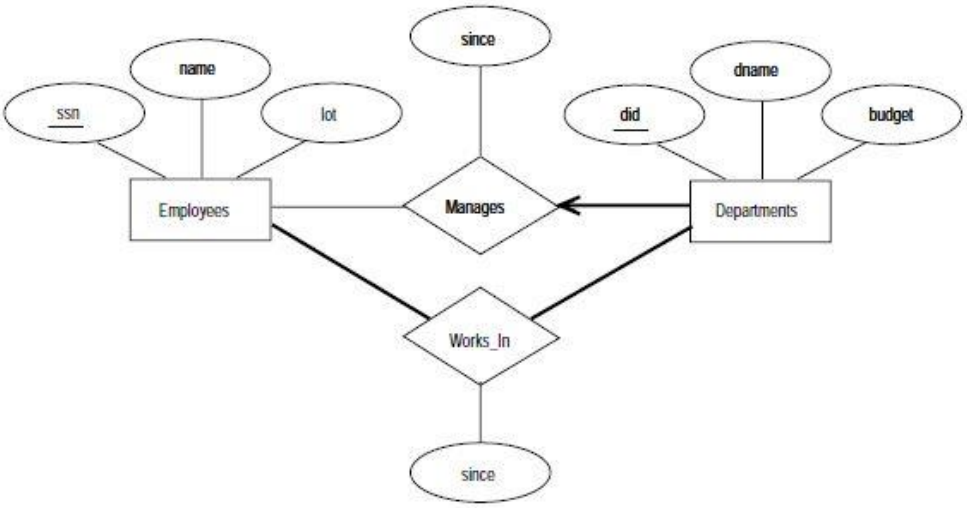
- Create a table for the source entity sets as usual.
- Add every primary key field of the target as a field in the source.
- Declare these fields as foreign keys.

***SOLUTION***

Create table department(ssn char(11),name varchar(20),lot int(5),did int(5),primary key(ssn),

foreign key (did) references Employees)

**Translating Relationship Sets with Participation Constraints**



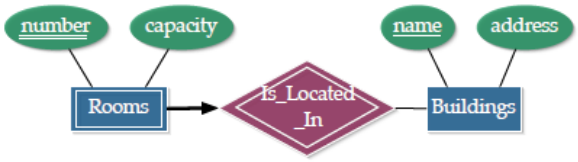

- Consider the ER diagram in Figure 3.13, which shows two relationship sets, Manages and Works In.
- Every Department is required to have a manager due to participation constraint.
- ***In a Relationship, Participation constraint specifies the existence of an entity when it is related to another entity in a relationship type. It is also called minimum cardinality constraint.***

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

**Translating Weak Entity Sets**

Algorithm

- Create a table for the weak entity set.
- Make each attribute of the weak entity set a field of the table.
- Add fields for the primary key attributes of the identifying owner and weak entity.
- Declare a foreign key constraint on these identifying owner fields.
- Instruct the system to automatically delete any tuples in the table for which there are no owners




```
create table Rooms (
     number          char(8),
     capacity        integer,
     building_name   char(20),
     primary key     (number,building_name),
     foreign key     (building_name) references Buildings
                     on delete cascade )
```

## a)Translating Relationship Sets with Key Constraints

- Algorithm

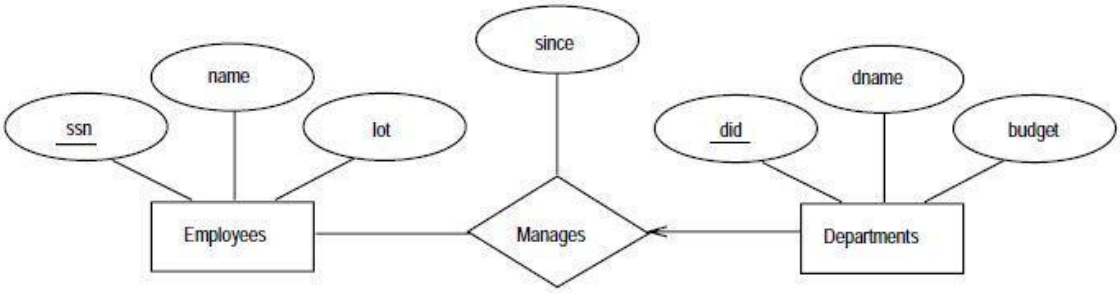
  -Create a table for the relationship set.

  -Add all primary keys of the participating entity sets as fields of the table.

  -Add a field for each attribute of the relationship.

  -Declare a primary key using the key fields from the source entity set only.

  -Declare foreign key constraints for all the fields from the source and target entity sets.




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

## a.1 Mapping relationship sets (with key constraints, 2nd method)

- Algorithm

  -Create a table for the source entity sets as usual.

  -Add every primary key field of the target as a field in the source.

  -Declare these fields as foreign keys.

*SOLUTION*

Create table department(ssn char(11),

name varchar(20),

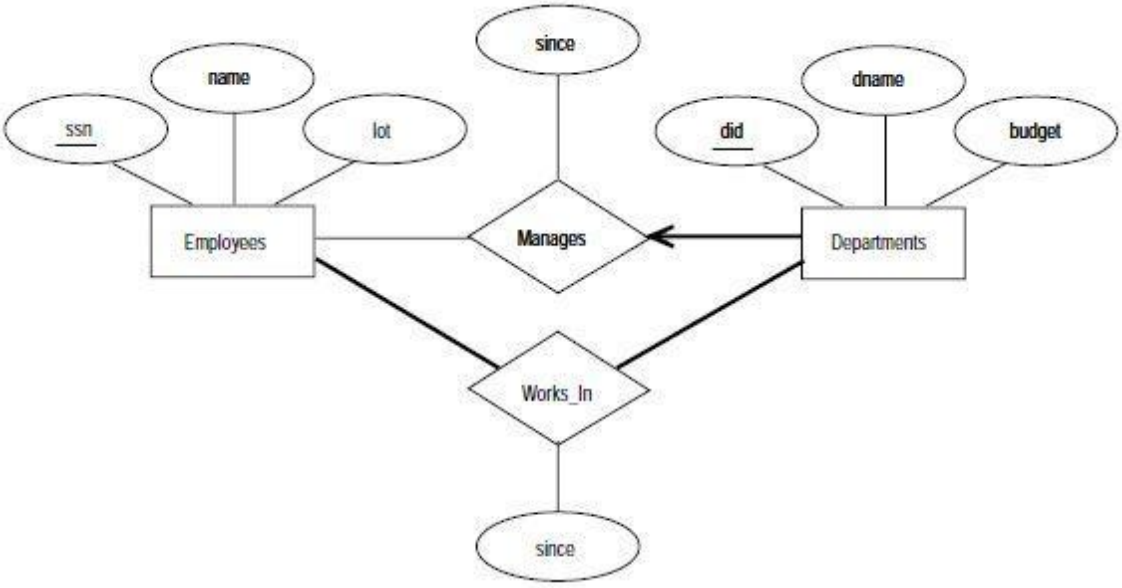
Iot int(5),

did int(5),

primary key(ssn),

foreign key (did) references Employees)

## b)Translating Relationship Sets with Participation Constraints




- Consider the ER diagram in Figure 3.13, which shows two relationship sets, Manages and Works In.
- Every Department is required to have a manager due to participation constraint.
- In a Relationship, Participation constraint specifies the existence of an entity when it is related to another entity in a relationship type. It is also called minimum cardinality constraint.

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

### c) Translating Weak Entity Sets

- Algorithm

-Create a table for the weak entity set.

-Make each attribute of the weak entity set a field of the table.

-Add fields for the primary key attributes of the identifying owner and weak entity.

-Declare a foreign key constraint on these identifying owner fields.

-Instruct the system to automatically delete any tuples in the table for which there are no owners


```
create table Rooms (
      number          char(8),
      capacity        integer,
      building_name   char(20),
      primary key     (number,building_name),
      foreign key     (building_name) references Buildings
                      on delete cascade )
```

The CASCADE option ensures that information about an Rooms and Buildings will be deleted if the corresponding Building tuple is deleted.

# SQL QUERIES

## SQL OVERVIEW

he SQL language has several aspects :

**1.The Data Manipulation Language (DML)** :- This subset of SQL allows users to write queries and to insert, delete and modify rows.

**2.The Data Definition Language(DDL) :-** This subset of SQL supports the creation, deletion and modification of definitions for tables and views. Integrity Constraints can be defined on table, either when the table is created or later.

**3.Trigger and Advanced Integrity Constraint :-** The new SQL includes supports for triggers, which are actions executed by the DBMS whenever changes to database meet conditions specified in the trigger. SQL allows the use of queries to specify complex integrity constraint specifications.

CONT…

Embedded and Dynamic SQL :- Embedded SQL feature allow SQL code to be called from host language such as C or COBOL. Dynamic SQL allows a query to be executed at run time.

Client-Server Execution and Database Access :- These commands control how a client application can connect to SQL server or access data from database over networks.

Transaction Management :- Various commands allow a user to explicitly control aspects of how transactions to be executed.

Security :- SQL provides mechanism to control users access to data objects such as tables or views.

Advanced Features :- The New SQL supports oops concepts, recursive queries , decision support queries etc.

# THE FORM OF BASIC SQL QUERY

## SQL Data Definition

The set of relations in a database must be specified to the system by means of a data-definition language (DDL).

The SQL DDL allows specification of not only a set of relations, but also information about each relation, including:

-***The schema for each relation.***

***-The types of values associated with each attribute.***

***-The integrity constraints.***

***-The set of indices to be maintained for each relation.***

-***The security and authorization information for each relation.***

***-The physical storage structure of each relation on disk.***

## Basic Types

- **char**($n$): A fixed-length character string with user-specified length $n$. The full form, **character**, can be used instead.
- **varchar**($n$): A variable-length character string with user-specified maximum length $n$. The full form, **character varying**, is equivalent.
- **int**: An integer (a finite subset of the integers that is machine dependent). The full form, **integer**, is equivalent.
- **smallint**: A small integer (a machine-dependent subset of the integer type).
- **numeric**($p$, $d$): A fixed-point number with user-specified precision. The number consists of $p$ digits (plus a sign), and $d$ of the $p$ digits are to the right of the decimal point. Thus, **numeric**(3,1) allows 44.5 to be stored exactly, but neither 444.5 or 0.32 can be stored exactly in a field of this type.
- **real, double precision**: Floating-point and double-precision floating-point numbers with machine-dependent precision.
- **float**($n$): A floating-point number, with precision of at least $n$ digits.

## Basic Schema Definition

We define an SQL relation by using the **create table** command. The following command creates a relation *department* in the database.

```
create table department
        (dept_name  varchar (20),
         building   varchar (15),
         budget     numeric (12,2),
         primary key (dept_name));
```

-The relation created above has three attributes, *dept name*, which is a character string of maximum length 20, *building*, which is a character string of maximum length 15, and *budget*, which is a number with 12 digits in total, 2 of which are after the decimal point.

-The **create table** command also specifies that the *dept name* attribute is the primary key of the *department* relation.

## DML COMMANDS IN SQL

SELECT – retrieve data from the a database.

INSERT – insert data into a table.

UPDATE – updates existing data within a table.

DELETE – deletes all records from a table, the space for the records remain.

## TABLE CREATION

- **Problem :** Create a table Student with following fields : Student Rollno, Student Name,Student Age.

- ***create table tablename(column_name column_type);***

- **Example: create table student(Rollno int(5),Name varchar(10),Age int(5));**

- **Problem :** To view the created table Student
- ***describe tablename;***
- **Example: describe Student;**

## INTRODUCTION

- SQL constraints are used to specify rules for data in a table.
- Constraints can be specified when the table is created with the CREATE TABLE statement, or after the table is created with the ALTER TABLE statement.

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

## SQL CONSTRAINTS

- SQL constraints are used to specify rules for the data in a table.
- Constraints are used to limit the type of data that can go into a table. This ensures the accuracy and reliability of the data in the table. If there is any violation between the constraint and the data action, the action is aborted.

Constraints can be column level or table level. Column level constraints apply to a column, and table level constraints apply to the whole table.

THE FOLLOWING CONSTRAINTS ARE COMMONLY USED IN SQL:

- **NOT NULL** - Ensures that a column cannot have a NULL value
- **UNIQUE** - Ensures that all values in a column are different
- **PRIMARY KEY** - A combination of a NOT NULL and UNIQUE. Uniquely identifies each row in a table
- **FOREIGN KEY** - Uniquely identifies a row/record in another table
- **CHECK** - Ensures that all values in a column satisfies a specific condition
- **DEFAULT** - Sets a default value for a column when no value is specified
- **INDEX** - Used to create and retrieve data from the database very quickly

## 1.NOT NULL ON CREATE TABLE

- By default, a column can hold NULL values.
- The NOT NULL constraint enforces a column to NOT accept NULL values.
- This enforces a field to always contain a value, which means that you cannot insert a new record, or update a record without adding a value to this field.

Example

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

## 1.a.NOT NULL ON ALTER TABLE

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

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

## 2.UNIQUE CONSTRAINT

- The UNIQUE constraint ensures that all values in a column are different.
- Both the UNIQUE and PRIMARY KEY constraints provide a guarantee for uniqueness for a column or set of columns.

## 2.a SQL UNIQUE CONSTRAINT ON CREATE TABLE

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

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

## 2.b SQL UNIQUE CONTRAINT ON ALTER TABLE

- To create a UNIQUE constraint on the "ID" column when the table is already created, use the following SQL:

```sql
ALTER TABLE Persons
ADD UNIQUE (ID);
```

## 2.c TO DROP A UNIQUE CONSTRAINT, USE THE FOLLOWING SQL:

- To drop a UNIQUE constraint, use the following SQL:

```sql
ALTER TABLE Persons
DROP INDEX UC_Person;
```

## 3.SQL PRIMARY KEY CONSTRAINT

- The PRIMARY KEY constraint uniquely identifies each record in a table.
- Primary keys must contain UNIQUE values, and cannot contain NULL values.
- A table can have only ONE primary key; and in the table, this primary key can consist of single or multiple columns (fields).

### 3.a SQL PRIMARY KEY ON CREATE TABLE

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

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

## 3.b SQL PRIMARY KEY ON ALTER TABLE

To create a PRIMARY KEY constraint on the "ID" column when the table is already created, use the following SQL:

```
ALTER TABLE Persons
ADD PRIMARY KEY (ID);
```

```
ALTER TABLE Persons
DROP PRIMARY KEY;
```

# 4.FOREIGN KEY CONSTRAINTS

- A FOREIGN KEY is a key used to link two tables together.
- A FOREIGN KEY is a field (or collection of fields) in one table that refers to the PRIMARY KEY in another table.

| PersonID | LastName | FirstName | Age |
|---|---|---|---|
| 1 | Hansen | Ola | 30 |
| 2 | Svendson | Tove | 23 |
| 3 | Pettersen | Kari | 20 |

"Orders" table:

| OrderID | OrderNumber | PersonID |
|---|---|---|
| 1 | 77895 | 3 |
| 2 | 44678 | 3 |
| 3 | 22456 | 2 |
| 4 | 24562 | 1 |

*-The "PersonID" column in the "Persons" table is the PRIMARY KEY in the "Persons" table.*

*-The "PersonID" column in the "Orders" table is a FOREIGN KEY in the "Orders"*

*table.*

## 4.a SQL FOREIGN KEY ON CREATE TABLE

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

## 5.SQL CREATE INDEX STATEMENT

- The CREATE INDEX statement is used to create indexes in tables.
- Indexes are used to retrieve data from the database more quickly than otherwise. The users cannot see the indexes, they are just used to speed up searches/queries.

```
CREATE INDEX index_name
ON table_name (column1, column2, ...);
```

```
CREATE INDEX idx_pname
ON Persons (LastName, FirstName);
```

## BASIC FORM OF SQL QUERY IS AS FOLLOWS

***SELECT [DISTINCT] select-list FROM from-list WHERE qualification.***

- **From-list** :- List of table name.
- **Select-list** :- list of column names of tables named in from-list.
- **Qualification** :- in the WHERE clause is a Boolean combination(ie, an

expression using logical connectives AND,OR and NOT),can be conditions with comparison operators {<, <=, >, >= ,=,<>}.

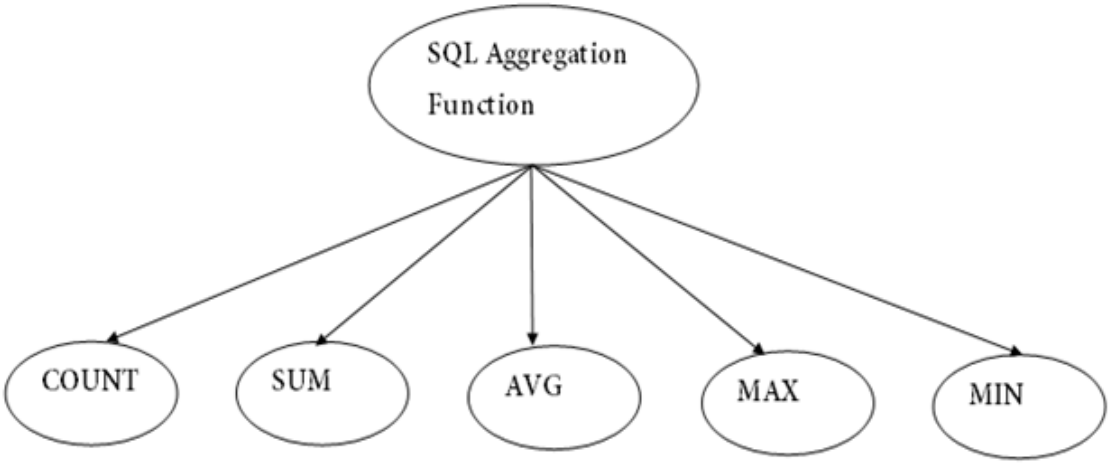
- **DISTINCT** :- Keyword is optional. It indicates that the table computed as an answer to the query should not contain duplicates (ie) two copies of same row.

# DBMS(Aggregate functions,null values,date and string functions )

## AGGEGRATE FUNCTIONS

Aggregate functions are functions that make a collection (a set or multiset) of values as input and return a single value.

SQL offers five built in aggregate functions.




## 1 COUNT FUNCTION

- COUNT function is used to Count the number of rows in a database table. It can work on both numeric and non-numeric data types.
- COUNT function uses the COUNT(*) that returns the count of all the rows in a specified table. COUNT(*) considers duplicate and Null.

COUNT SYNTAX

- We use aggregate function count frequently to count the number of tuples in a relation. The notation for this function in SQL is count (*)
- Thus to find the number of tuples in the course relation we write

```
Select count(*)
From course
```

Sql does not allow the use of distinctwith count (*) .It is legal to use distinct with max and min , even though the result does not change.

COUNT EXAMPLE

**select count(bcity) from branch;**

**OUTPUT**

```
+--------------+
| count(bcity) |
+--------------+
|            6 |
+--------------+
```

## 2. SUM FUNCTION

- The sum function is used to calculate the sum of all selected columns
- It works on numeric fields only

```
SUM()
Or
SUM([ALL|DISTINCT] expression)
```

SUM EXAMPLE

select sum(AMT) from borrow where BNAME='ANDHERI';

**OUTPUT:**

```
+-----------+
| sum(AMT)  |
+-----------+
|      3000 |
+-----------+
```

## 3. AVG FUNCTION

SQL **AVG** function is used to find out the average of a field in various records.

AVG EXAMPLE

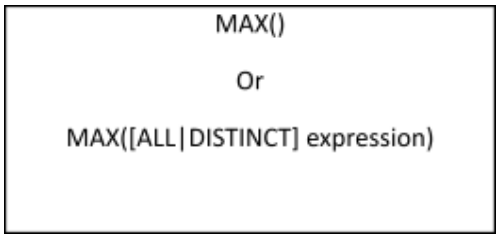
select avg(AMT) from deposit;

OUTPUT

```
+-------------+
| avg(AMT)    |
+-------------+
| 27857.1429 |
+-------------+
```

## 4. MAX FUNCTION

MAX function is used to find the maximum value of a certain column. This function determines the largest value of all selected values of a column.




MAX EXAMPLE

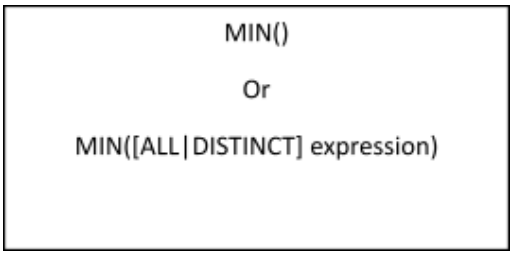
**select max(amt) from deposit where cname in (select cname from customer where lcity='delhi');**

**OUTPUT:**

```
+----------+
| max(amt) |
+----------+
|    50000 |
+----------+
```

## 5 . MIN FUNCTION

MIN function is used to find the number of a certain column. This function determines the smallest value  of all selected values of a column.




MIN EXAMPLE

**select min(AMT) from borrow where bname='ajne';**

**OUTPUT:**

```
+-----------+
| min(AMT)  |
+-----------+
|      5000 |
+-----------+
```

**NULL VALUES**

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

- Null values present special problems in relational operations including arithmetic operations, comparison operations and set operations.

  PROBLEM 1
- The result of an arithmetic expressions (involving , example +,-,* or/) is null if any of the input value is null.

- For example if a query has an expression r.A+5, and r.A null for a particular tuple , then the expression result must also be null for that tuple.

PROBLEM 2

- Comparisons involving nulls are more of a problem. For example consider the comparison “1<null”.

- It would be wrong to say this is true since we do not know what the
- But it would likewise be wrong to claim the expression is false ; if we did , “not(1<null)” would evaluate to true , which does not make sense.

- Since the predicate in a where clause can involve Boolean operations such as and, or, and not on the results of comparisons, the definition of the Boolean operations are extend to deal with the value unknown.

- and: The result is true and unknown is unknown , false and unknown is false while unknown and unknown is unknown,
- or: The result of true or unknown is is true , false or unknown is unknown, while unknown or unknown is unknown .
- not: The result of not unknown is unknown.

You can verify that if r.A is null, then “”r.A” as well as “not(1<r.A)” evaluate to unknown

- If the where clause predicate evaluates to either false unknown for a tuple, that tuple is not added to the result.
- SQL uses the special keyword null in a predicate to test for a null value.

NULL EXAMPLE

create table employe(emp_id int(5) NOT NULL, name varchar(20) NOT NULL ,salary int(5));

OUTPUT

```
+--------+-------------+------+-----+---------+-------+
| Field  | Type        | Null | Key | Default | Extra |
+--------+-------------+------+-----+---------+-------+
| emp_id | int(5)      | NO   |     | NULL    |       |
| name   | varchar(20) | NO   |     | NULL    |       |
| salary | int(5)      | YES  |     | NULL    |       |
+--------+-------------+------+-----+---------+-------+
```

## 3. DATE FUNCTIONS

### 1. Curdate

The CURDATE() function returns the current date.

**Note:** The date is returned as "YYYY-MM-DD" (string) or as YYYYMMDD (numeric).

**Note:** This function equals the CURRENT_DATE() function.

EXAMPLE

**select curdate();**

**OUTPUT**

```
+-------------+
| curdate()   |
+-------------+
| 2021-07-18 |
+-------------+
```

### 2.Curtime()

The CURTIME() function returns the current time.

**Note:** The time is returned as "HH-MM-SS" (string) or as HHMMSS.uuuuuu (numeric).

**Note:** This function equals the CURRENT_TIME() function.

EXAMPLE

**select curtime();**

**OUTPUT**

```
+-----------+
| curtime() |
+-----------+
| 20:02:09  |
+-----------+
```

2. **Year()**

To select the ccurrent year

EXAMPLE

**select year('2021/07/19') as year;**

**OUTPUT**

```
+------+
| year |
+------+
| 2021 |
+------+
```

3. **Month()**

The MONTH() function returns the month part for a specified date (a number from 1 to 12).

**EXAMPLE**

**select month('2021/07/19') as month;**

**OUTPUT**

```
+-------+
| month |
+-------+
|     7 |
+-------+
```

4. **monthname()**

The MONTHNAME() function returns the name of the month for a given date.

EXAMPLE

**select monthname('2021/07/19') as monthname;**

**OUTPUT**

```
+-----------+
| monthname |
+-----------+
| July      |
+-----------+
```

5. **Dayofyear()**

The DAYOFYEAR() function returns the day of the year for a given date (a number from 1 to 366).

EXAMPLE

**select dayofyear('2021/07/19') as dayofyear;**

**OUTPUT**

```
+-----------+
| dayofyear |
+-----------+
|       200 |
+-----------+
```

6. **dayofmonth()**

The DAYOFMONTH() function returns the day of the month for a given date (a number from 1 to 31).

**Note:** This function equals the DAY() function.

EXAMPLE

**select dayofmonth('2021/07/19') as dayofmonth;**

**OUTPUT**

```
+-------------+
| dayofmonth  |
+-------------+
|          19 |
+-------------+
```

7. **dayofweek()**

The DAYOFWEEK() function returns the weekday index for a given date (a number from 1 to 7).

**Note:** 1=Sunday, 2=Monday, 3=Tuesday, 4=Wednesday, 5=Thursday, 6=Friday, 7=Saturday.

EXAMPLE

**select dayofweek('2021/07/19') as dayofweek;**

**OUTPUT**

```
+------------+
| dayofweek  |
+------------+
|          2 |
+------------+
```

8. **hour()**

The HOUR() function returns the hour part for a given date (from 0 to 838).

EXAMPLE

**select hour('2021/05/03 04:02:00') as hour;**

**OUTPUT**

```
+------+
| hour |
+------+
|    4 |
+------+
```

9. **minute()**

The MINUTE() function returns the minute part of a time/datetime (from 0 to 59).

EXAMPLE

**select minute('2021/05/03 04:03:00') as minute;**

**OUTPUT**

```
+--------+
| minute |
+--------+
|      3 |
+--------+
```

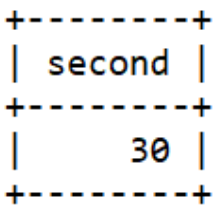
10. **second()**

The SECOND() function returns the seconds part of a time/datetime (from 0 to 59).

EXAMPLE

**select second('2021/05/03 04:05:30') as second;**

**OUTPUT**




### 11. To_days()

The TO_DAYS() function returns the number of days between a date and year 0 (date "0000-00-00").

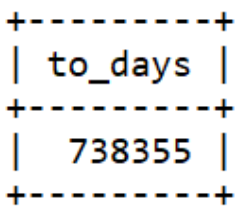
The TO_DAYS() function can be used only with dates within the Gregorian calendar.

**Note:** This function is the opposite of the FROM_DAYS() function.

**EXAMPLE**

select to_days('2021/07/19') as to_days;

OUTPUT




### 12. From_days()

The FROM_DAYS() function returns a date from a numeric datevalue.

The FROM_DAYS() function is to be used only with dates within the Gregorian calendar.

**Note:** This function is the opposite of the TO_DAYS() function.

**EXAMPLE**

**select from_days(738355) as from_days;**

**OUTPUT**

```
+------------+
| from_days  |
+------------+
| 2021-07-19 |
+------------+
```

### 13.Date_add()

The DATE_ADD() function adds a time/date interval to a date and then returns the date.

**EXAMPLE**

**select date_add('2021/07/19',interval 10 day) as date_add;**

**OUTPUT**

```
+------------+
| date_add   |
+------------+
| 2021-07-29 |
+------------+
```

### 14.date_sub()

The DATE_SUB() function subtracts a time/date interval from a date and then returns the date.

**EXAMPLE**

**select date_sub('2021/07/19',interval 10 day) as date_sub;**

**OUTPUT**

```
+------------+
| date_sub   |
+------------+
| 2021-07-09 |
+------------+
```

### 15.Extract()

The EXTRACT() function extracts a part from a given date.

EXAMPLE

**select extract(year from '2021-07-19') as extrct;**

**OUTPUT**

```
+--------+
| extrct |
+--------+
|   2021 |
+--------+
```

### 16.Period_diff()

The PERIOD_DIFF() function returns the difference between two periods. The result will be in months.

**Note:** *period1* and *period2* should be in the same format.

EXAMPLE

**select period_diff(202110,202103) as period_diff;**

**OUTPUT**

```
+-------------+
| period_diff |
+-------------+
|           7 |
+-------------+
```

## 4. STRING FUNCTIONS

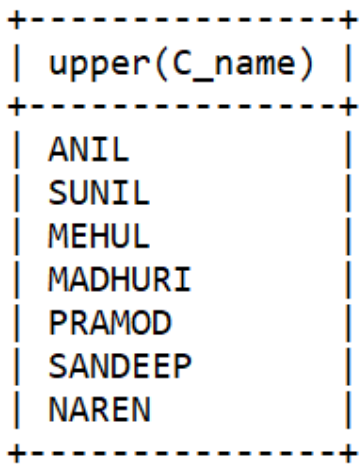
1. **Upper()**

   The UPPER() function converts a string to upper-case.

   EXAMPLE

**select upper(C_name)  from deposit;**

**OUTPUT**



| upper(C_name) |
|---|
| ANIL |
| SUNIL |
| MEHUL |
| MADHURI |
| PRAMOD |
| SANDEEP |
| NAREN |

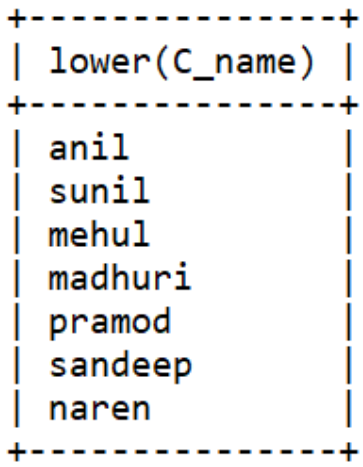
**2. Lower()**

The LOWER() function converts a string to lower-case.

EXAMPLE

**select lower(C_name) from deposit;**

**OUTPUT**



| lower(C_name) |
|---|
| anil |
| sunil |
| mehul |
| madhuri |
| pramod |
| sandeep |
| naren |

**3.Concat()**

The CONCAT() function adds two or more strings together.

EXAMPLE

**select concat('good','morning') as concat;**

**OUTPUT**

```
+-------------+
| concat      |
+-------------+
| goodmorning |
+-------------+
```

### 3. Concat_ws

The CONCAT_WS() function adds two or more strings together with a separator.

EXAMPLE

**select concat_ws(',','class','room') as concat_ws;**

**OUTPUT**

```
+------------+
| concat_ws  |
+------------+
| class,room |
+------------+
```

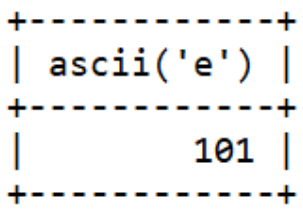
### 4. ASCI()

The ASCII() function returns the ASCII value for the specific character.

EXAMPLE

**select ascii('e');**

**OUTPUT**



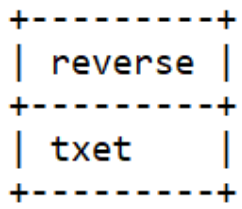

5. **Reverse()**

The REVERSE() function reverses a string and returns the result.

EXAMPLE

**select reverse('text') as reverse;**

**OUTPUT**



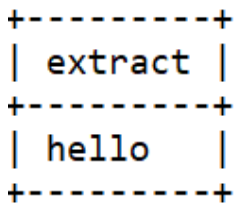

6. **Substring()**

The SUBSTRING() function extracts some characters from a string.

EXAMPLE

**select substring('hello world','1','5') as extract;**

**OUTPUT**



**Replace()**

The REPLACE() function replaces all occurrences of a substring within a string, with a new substring.

EXAMPLE

**select replace('bname','b','branch') as replce;**

**OUTPUT**

```
+------------+
| replce     |
+------------+
| branchname |
+------------+
```

# Views In SQL

## Definition and Introduction:

- A view is a table whose rows are not explicitly stored in the database but are computed as needed from a view definition.
- Views in SQL are kind of virtual tables.
- A view also has rows and columns as they are in a real table in the database.
- We can create a view by selecting fields from one or more tables present in the database.
- A View can either have all the rows of a table or specific rows based on certain condition.
- Any such relation that is not part of the logical model, but is made visible to a user as a virtual relation, is called a view.
- It is possible to support a large number of views on top of any given set of actual relations.
- We define a view in SQL by using the create view command. To define a view, we must give the view a name and must state the query that computes the view. The form of the create view command is:

**create view** *v* **as** <query expression>;

where <query expression> is any legal query expression. The view name is represented by v.

## Creating View:

We can create View using CREATE VIEW statement. A View can be created from a single table or multiple tables.

Syntax:

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

view_name: name for the view

table_name: Name of the table
condition: condition to select rows

**Creating view from a single table**:

Eg:
create a View named Details View from the table Student Details.
Query:

```
CREATE VIEW DetailsView AS
SELECT NAME,ADRESS
FROM StudentDetails
WHERE S_ID <5;
```

To see the data in the view:

```
SELECT * FROM DetailsView;
```

**Creating view from multiple tables:**

Eg:
Create a View named MarksView from two tables StudentDetails and StudentMarks. To create a View from multiple tables we can simply include multiple tables in the SELECT statement.

Query:

```
  CREATE VIEW MarksView AS
  SELECT StudentDetails.Name,StudentDetails.ADDRESS,StudentMark
FROM StudentDetails,StudentMarks WHERE
StudentDetails.Name=StudentMarks.Name;
```

**To display data of view MarksView:**

```
SELECT * FROM MarksView;
```

## Deleting Views:

SQL allows us to delete an existing View. We can delete or drop a View using the DROP statement.

**Syntax:**

```
DROP VIEW view_name;
```

View_name: Name of view which we want to delete.

Eg:

```
DROP VIEW MarksView;
```

**Deleting a row from a view**:

Deleting rows from a view is also simple as deleting rows from a table. We can use the DELETE statement of SQL to delete rows from a view. Also deleting a row from a view first delete the row from the actual table and the change is then reflected in the views.

**Syntax:**

```
DELETE FROM view_name
WHERE condition;
```

View_name: Name of view from where we want to delete rows.

Condition: condition to select rows.

**Inserting a row in a view**:

We can insert a row in a view in a same way as we do in a table. We can use the INSERT INTO statement of SQL to insert a row in a view.

**Syntax:**

```
INSERT INTO view_name(column1,column2,column3,…..)
VALUES(value1,value2,value3….);
```

## Materialized Views:

- Certain database systems allow view relations to be stored, but they make sure that, if the actual relations used in the view definition change, the view is kept up-to-date. Such views are called **materialized views**.
- For example, consider the view *departments total salary*. If the above view is materialized, its results would be stored in the database.
- However, if an *instructor* tuple is added to or deleted from the *instructor* relation, the result of the query defining the view would change, and as a result the materialized view's contents must be updated.
- The process of keeping the materialized view up-to-date is called **materialized view maintanenance.**

## Embedded and Dynamic SQL

There are two approaches to accessing SQL from a general-purpose programming language:

Embedded and dynamic SQL define how SQL statements can be embedded within general-purpose programming languages, such as C, C++, and Java**.**

### Dynamic SQL:

- A general-purpose program can connect to and communicate with a database server using a collection of functions (for procedural languages) or methods (for object-oriented languages).
- Dynamic SQL allows the program to construct an SQL query as a character string at runtime, submit the query, and then retrieve the result into program variables a tuple at a time.
- The *dynamic SQL* component of SQL allows programs to construct and submit SQL queries at runtime.

### Embedded SQL:

- Like dynamic SQL, embedded SQL provides a means by which a program can interact with a database server.

- However, under embedded SQL, the SQL statements are identified at compile time using a preprocessor.
- The preprocessor submits the SQL statements to the database system for precompilation and optimization;
- then it replaces the SQL statements in the application program with appropriate code and function calls before invoking the programming-language compiler.
- A major challenge in mixing SQL with a general-purpose language is the mismatch in the ways these languages manipulate data.
- In SQL, the primary type of data is the relation. SQL statements operate on relations and return relations as a result.
- Programming languages normally operate on a variable at a time, and those variables correspond roughly to the value of an attribute in a tuple in a relation.
- Thus, integrating these two types of languages into a single application requires providing a mechanism to return the result of a query in a manner that the program can handle.

**Aspects of Embedded SQL:**

- An SQL statements used in a program is preceded by the keywords

  EXEC SQL

Format is : EXEC SQL<SQL Statement><Terminator>

*->The terminator is assumed to be a semicolon(;).*

*->The keywords EXEC SQL and the terminator are used by the precompiler to identify the embedded SQL commands and replace it with appropriate function calls.*

## JDBC

- The JDBC standard defines an application program interface (API) that Java programs can use to connect to database servers.

- (The word JDBC was originally an abbreviation for Java Database Connectivity, but the full form is no longer used.).
- The Java program must import java.sql.*, which contains the interface definitions for the functionality provided by JDBC.

**Steps involved in Accessing database from java program:**

1. **Connecting to the Database :-**
   - The first step in accessing a database from a Java program is to open a connection to the database.
   - This step is required to select which database to use, for example, an instance of Oracle running on your machine, or a PostgreSQL database running on another machine.
   - Only after opening a connection can a Java program execute SQL statements.

**2. Shipping SQL statements to the database system**.

   - Once a database connection is open, the program can use it to send SQL statements to the database system for execution. This is done via an instance of the class Statement.

**3. Retrieving the result of a query.**

   - The example program executes a query by using stmt.executeQuery. It retrieves the set of tuples in the result into a ResultSet object rset and fetches them one tuple at a time.

## Triggers and Cursors

## Cursor:

## Definition:

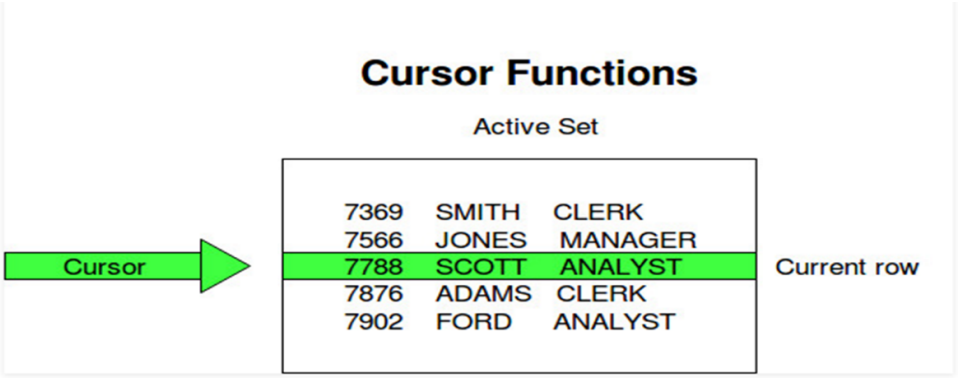
- Cursor is a Temporary Memory/Temporary work station.
- This area will be allocated by database server at the time of DML operations are performed by the user on database tables.

- This area contains all the relevant information relating to the statement and its execution.
- The **cursor** is a pointer to this context area and allows the PL/SQL program to control this area.

**Uses of Cursor:**

- For storing database tables.
- The major function of a cursor is to retrieve data, one row at a time, from a result set, unlike the SQL commands which operate on all the rows in the result set at one time. Cursors are used when the user needs to update records in a singleton fashion or in a row by row manner, in a database table.
- The Data that is stored in the Cursor is called the *Active Data Set*.

  Example:




**Types of Cursor**:

Implicit cursor
Explicit Cursor

## Implicit Cursor:

- default cursors od SQL Server database.
- these memory is allocated by database servers when we perform DML operations.
- 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

**Attributes:**

**1. % FOUND:**

Returns TRUE if an INSERT,UPDATE or DELETE statement affected one or more rows. Otherwise ,it returns FALSE.

**2. %NOFOUND:**

The logical opposite of %found. It returns TRUE if an INSERT,UPDATE, or DELETE statements affects no rows.Otherwise ,it return false.

**3.%ISOPEN**:

Always returns FALSE for implicit cursors,because oracle closes the SQL cursor automatically after executing its associated SQL statement.

**4. %ROWCOUNT:**

Returns the number of rows affected by an INSERT,UPDATE, or DELETE statement, or returned by a SELECT INTO statement.

## Explicit cursor:

These are cursors created by user to fetch data from table in row-by row manner.

**Cursor Actions**:

Declare Cursor: A cursor is declared by defining the SQL statement that returns a result set.
Open: A Cursor is opened and populated by executing the SQL statement defined by the cursor.
Fetch: When the cursor is opened, rows can be fetched from the cursor one by one or in a block to perform data manipulation.
Close: After data manipulation, close the cursor explicitly.
Deallocate: Finally, delete the cursor definition and release all the system resources associated with the cursor.

**There are four steps in using an Explicit Cursor.**

1. DECLARE the cursor in the Declaration section.

2. OPEN the cursor in the Execution Section.

3. FETCH the data from the cursor into PL/SQL variables or records in the Execution Section.

4. CLOSE the cursor in the Execution Section before you end the PL/SQL Block

**Syntax:**

DECLARE variables;

Records;

Create a cursor;

BEGIN;

OPEN cursor;

FETCH cursor;

Process the records;

CLOSE cursor;

END;

## TRIGGER:

A **trigger** is a statement that the system executes automatically as a side effect of a modification to the database. To design a trigger mechanism, we must meet

***Two Requirements:***

**1.**Specify when a trigger is to be executed. This is broken up into an *event* that causes the trigger to be checked and a *condition* that must be satisfied for trigger execution to proceed.

**2.** Specify the *actions* to be taken when the trigger executes.

Once we enter a trigger into the database, the database system takes on the responsibility of executing it whenever the specified event occurs and the corresponding condition is satisfied.

The SQL standard defines two types of triggers: row-level triggers and statement-level triggers.
A row-level trigger is activated for each row that is inserted, updated, or deleted. For example, if a table has 100 rows inserted, updated, or deleted, the trigger is automatically invoked 100 times for the 100 rows affected.
A statement-level trigger is executed once for each transaction regardless of how many rows are inserted, updated, or deleted.
MySQL supports only row-level triggers. It doesn't support statement-level triggers.

**Advantages of Trigger**:

Triggers can be used to implement certain integrity constraints that cannot be specified using the constraint mechanism of SQL.
Maintains integrity of data

**Disadvantages of trigger:**

Triggers can only provide extended validations,not all validations. For simple validations ,you can use the NOT NULL, UNIQUE, CHECK, and FOREIGN KEY constraints.
Triggers can be difficult to troubleshoot because they execute automatically in the database ,which may not invisible to the client applications.
Triggers may increase the overhead of the MySQL server.

**Creating Trigger in MySQL**:

The CREATE TRIGGER statement creates a new trigger.

Syntax:

```
CREATE TRIGGER  trigger_name;
{ BEFORE| AFTER}  {INSERT |UPDATE | DELETE}
ON table_name  FOR EACH ROW;
Trigger_body;
```

In this syntax:

Specify the name of the trigger you want to create.

Trigger name must be **Unique.**
Specify the trigger action time which can be either BEFORE or AFETR which indicates that the trigger is invoked before or after each row is modified.
Specify the operation that activates the trigger ,which can be INSERT, UPDATE,DELETE.
Specify the name of the table to which the trigger belongs after the ON keyword.
Finally , specify the statement to execute when the trigger activates.If you want to execute multiple staements , you see the BEGIN END compound statement.

# RELATIONAL DATABASE DESIGN

- Schema Refinement
- Decomposition
- Functional Dependency
- Normalization

# SCHEMA REFINEMENT

**Schema Refinement is**

- **a technique of organizing the data in the database.**
- **is a systematic approach of decomposing tables to eliminate data redundancy and undesirable characteristics like Insertion, Update and Deletion Anomalies.**

- **Redundancy - repetition of same data or duplicate copies of same data stored in different locations.**
- **Anomalies: the problems occurred after poorly planned and normalised databases where all the data is stored in one table**

# *Problems Caused by Redundancy*

**Redundant storage: Some information is stored repeatedly.**

**Update anomalies: If one copy of such repeated data is updated, an inconsistency is created unless all copies are similarly updated.**

**Insertion anomalies: It may not be possible to store some information unless some other information is stored aswell.**

**Deletion anomalies: It may not be possible to delete some information without losing some other information as well.**

**Update anomaly - to update the address of Rick then we have to update the same in two rows or the data will becomeinconsistent**

**Insert anomaly - a new employee joins the company, who is under training and currently not assigned to any department**

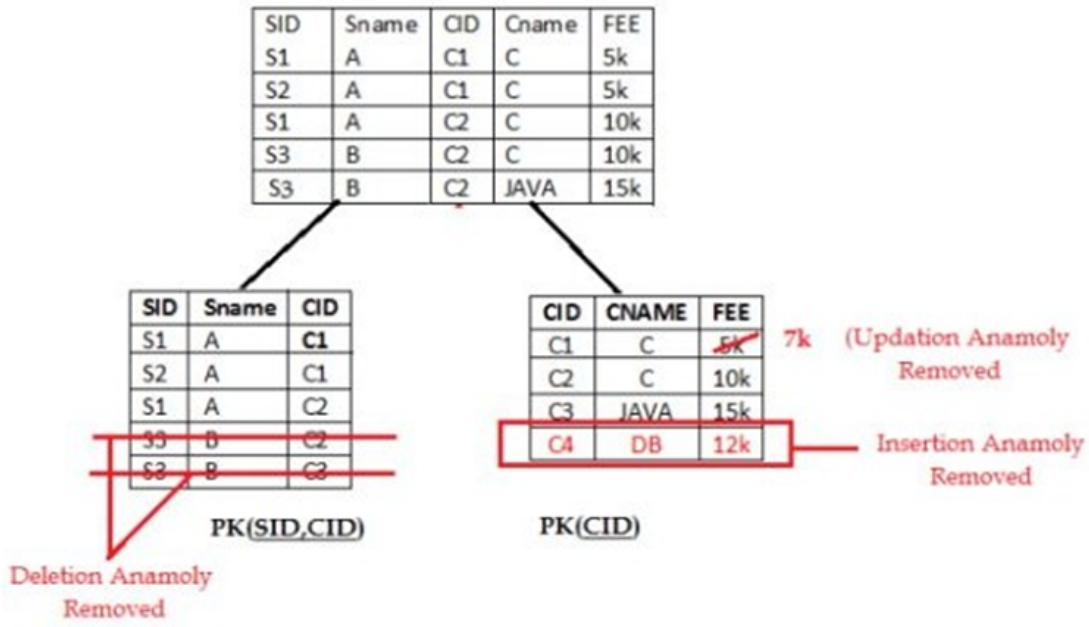
**Delete anomaly -company closes the department D890 then deleting the rows that are having emp_dept as D890 would also delete the information of employee Maggie**

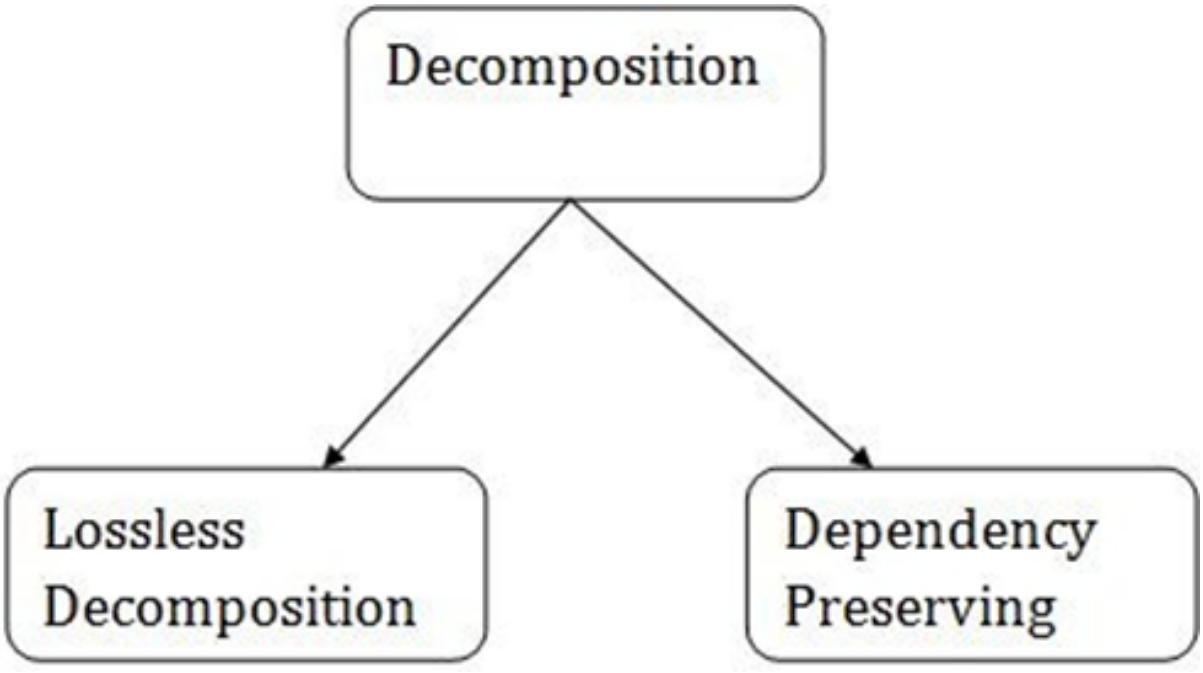
# DECOMPOSITION

**TO AVOID REDUNDANCY and problems due to redundancy, we use refinement technique called DECOMPOSITION.**

**Use of Decomposition:-**

- **Process of decomposing a larger relation into smaller relations.**
- **Each of smaller relations contain subset of attributes of original relation.**
- **Functional dependencies can be used to identify redundancy and to suggest refinements to the schema.**
  **Eg: To deal with the redundancy in Hourly_Emps by decomposing it into two relations**
- **If the relation has no proper decomposition, then it may lead to problems like loss of information.**
- **Decomposition is used to eliminate some of the problems of bad design like anomalies, inconsistencies, and redundancy.**


SID Sname CID Cname FEE
S1 A C1 C 5k
S2 A C1 C 5k
S1 A C2 C 10k
S3 B C2 C 10k
S3 B C2 JAVA 15k
SID Sname CID
S1 A C1
S2 A C1
S1 A C2
S3 B C2
S3 B C3
PK(SID,CID)
CID CNAME FEE
C1 C 5k
7k
(Updation Anamoly Removed
C2 C 10k
C3 JAVA 15k
C4 DB 12k
Insertion Anamoly Removed
PK(CID)
Deletion Anamoly Removed


Decomposition
Lossless Decomposition
Dependency Preserving

**a) Lossless Decomposition**

- **to recover any instance of the decomposed relation from corresponding instances of the smaller relations**
- **If the information is not lost from the relation that is decomposed, then the decomposition will be lossless.**
- **It guarantees that the join of relations will result in the same relation as it was decomposed.**
- **The relation is said to be lossless decomposition if natural joins of all the decomposition give the original relation.**

# example of loseless decomposition

| EMP_ID | EMP_NAME | EMP_AGE | EMP_CITY | DEPT_ID | DEPT_NAME |
|---|---|---|---|---|---|
| 22 | Denim | 28 | Mumbai | 827 | Sales |
| 33 | Alina | 25 | Delhi | 438 | Marketing |
| 46 | Stephan | 30 | Bangalore | 869 | Finance |
| 52 | Katherine | 36 | Mumbai | 575 | Production |
| 60 | Jack | 40 | Noida | 678 | Testing |

**b) Dependency Preserving**

**Enables us to enforce any constraint on the original relation by simply enforcing some contraints on each of the smaller relations.**

**We need not perform joins of the smaller relations to check whether a constraint on the original relation is violated.**

# FUNCTIONAL DEPENDENCIES

**Functional dependency is**

- **a relationship that exist when one attribute uniquely determines another attribute.**
- **a form of integrity constraint that can identify schema with redundant storage problems and to suggest refinement**

## *Representation*

**X → Y**

- **The left side of FD is known as a determinant**
- **The right side of the production is known as a dependent.**
- **Functional dependency is represented by an arrow sign (→) that is, X→Y, where X functionally determines Y.**
- **The left-hand side attributes determine the values of attributes on the right-hand side.**

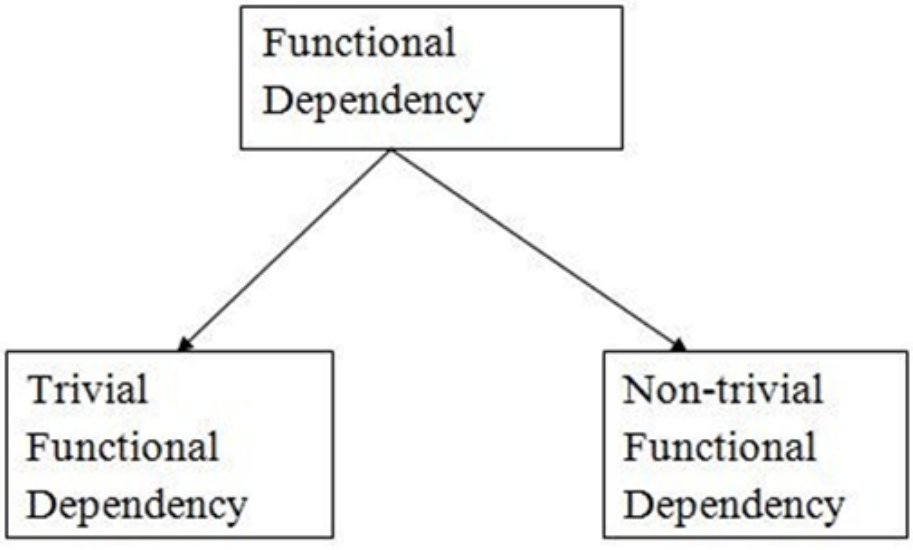
Example:

**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: Emp_Id → Emp_Name**
**We can say that Emp_Name is functionally dependent on Emp_Id.**

# Types of Functional dependency



**1. Trivial functional dependency**

**A → B has trivial functional dependency if B is a subset of A.**

**Other trivial like: A → A, B → B**

**Eg:Consider a table with two columns Employee_Id and Emplo yee_Name.**

**{Employee_id, Employee_Name} → Employee_Id is a trivial functional dependency as Employee_Id is a subset of {Employ ee_Id, Employee_Name}.**

**Also, Employee_Id → Employee_Id and**

**Employee_Name → Employee_Name are trivial dependenci es**

**2. Non-trivial functional dependency**

**A → B has a non-trivial functional dependency if B is not a subset of A.**

**When A intersection B is NULL, then A → B is called as complete non-trivial.**

**Example:**

**ID → Name,**

**Name → DOB**

# NORMALIZATION

- **First normal form(1NF)**
- **Second normal form(2NF)**
- **Third normal form(3NF)**
- **Boyce & Codd normal form (BCNF)**

# THE FIRST NORMAL FORM (1NF)

**A relation (table) is in 1NF if**

- **There are no duplicate rows or tuples in the relation.**
- **Each data value stored in the relation is single-valued**
- **Entries in a column (attribute) are of the same kind (type).**

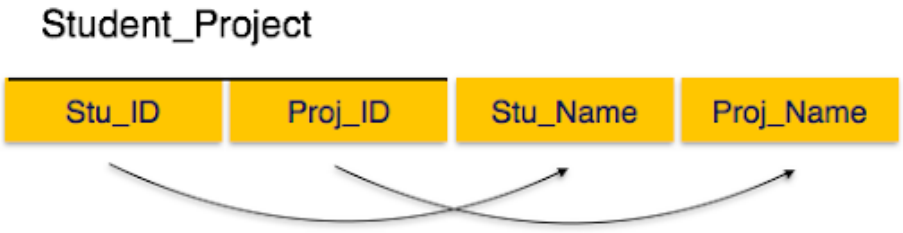
| Course | Content |
|---|---|
| Programming | Java, c++ |
| Web | HTML, PHP, ASP |

**This table is not in 1NF as the rule says "each attribute of a table must have atomic (single) values".**
**We re-arrange the relation (table) as below, to convert it to First Normal Form.**

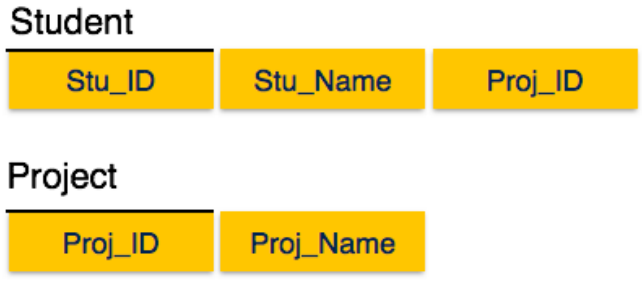
| Course | Content |
|---|---|
| Programming | Java |
| Programming | c++ |
| Web | HTML |
| Web | PHP |
| Web | ASP |

# THE SECOND NORMAL FORM (2NF)

**A relation is in 2NF if it is in 1NF and every non-key attribute is fully dependent on each candidate key of the relation.**



**We see here in Student_Project relation that the prime key attributes are Stu_ID and Proj_ID. According to the rule, non-key attributes, i.e. Stu_Name and Proj_Name must be dependent upon both and not on any of the prime key attribute individually. But we find that Stu_Name can be identified by Stu_ID and Proj_Name can be identified by Proj_ID independently. This is called partial dependency, which is not allowed in Second Normal Form.**



# THIRD NORMAL FORM (3NF)

**A table design is said to be in 3NF if both the following conditions hold:**

- **Table must be in 2NF**
- **Transitive functional dependency of non-prime attribute on any super key should be removed.**

**Transitive dependence**

- **Let A, B and C be three attributes of a relation R such that A ->B and B ->C. From these FDs, we may derive A -> C.**

**This dependence A -> C is transitive.**


**We find that in the above Student_detail relation, Stu_ID is the key and only prime key attribute.**
**We find that City can be identified by Stu_ID as well as Zip itself. Neither Zip is a superkey nor is City a prime attribute. Additionally, Stu_ID → Zip → City,so there exists transitive dependency.**

# BOYCE-CODD NORMAL FORM

**Boyce-Codd Normal Form (BCNF) is an extension of Third Normal Form on strict terms. BCNF states that –**
**For any non-trivial functional dependency, X → A, X must be a super-key.**
**In the above image, Stu_ID is the super-key in the relation Student_Detail and Zip is the super-key in the relation ZipCodes.**

**So,Stu_ID → Stu_Name, Zip and Zip → City**

**Which confirms that both the relations are in BCNF.**

# TRANSACTIONS

A transaction is a unit of program execution that accesses and possibly updates various data items. A transaction must see a consistent database. During transaction execution the database may be inconsistent. When the transaction is committed, the database must be consistent.
Two main issues to deal with: Failures of various kinds, such as hardware failures and system crashes Concurrent execution of multiple transactions

**Example:**
Suppose an employee of bank transfers Rs.800 from X's account to Y's account. This small transaction contains several low-level tasks:

| X's Account | Y's Account |
|---|---|
| Open_ Account(X) | Open_ Account(Y) |
| Old_ Balance = X. balance | Old_ Balance = Y. balance |
| New_ Balance = Old_ Balance -800 | New_ Balance =Old_ Balance +800 |
| X. balance = New_ Balance | Y. balance = New_ Balance |
| Close_ Account(X) | Close_ Account(Y) |

**Transactions access data using two operations:**

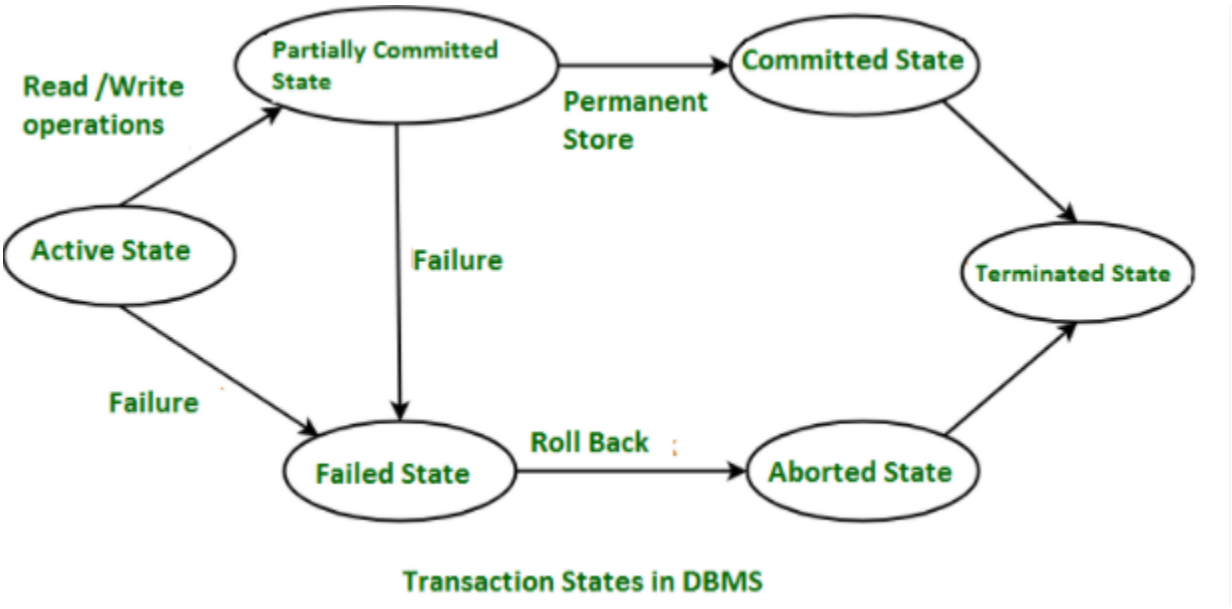
- **Read(X),** which transfers the data item X from the database to a variable, also called X, in a buffer in main memory belonging to the transaction that executed the read operation.
- **Write(X),** which transfers the value in the variable X in the main memory buffer of the transaction that executed the write to the data item X in the database.

**Example:** let *Ti* be a transaction that transfers $50 from account A to account B. This transaction can be defined as:

$T_i$: read($A$);
$A := A - 50$;
write($A$);
read($B$);
$B := B + 50$;
write($B$).

## Transaction states in DBMS:

States through which a transaction goes during its lifetime. These are the states which tell about the current state of the transaction and also tell how we will further do processing we will do on the transactions. These states govern the rules which decide the fate of the transaction whether it will commit or abort.




Transaction States in DBMS

These are different types of Transaction States:

1) Active state:

   When the instructions of the transaction are running then the transaction is in active state. If all the 'read and write' operations are performed without any error then it goes to the "partially committed state"; if any instruction fails, it goes to the "failed state".

2) Partially committed:

   After completion of all the read and write operation the changes are made in main memory or local buffer. If the changes are made permanent on the Database, then the state will change to "committed state" and in case of failure it will go to the "failed state".

3) Failed state:

   When any instruction of the transaction fails, it goes to the "failed state" or if failure occurs in making a permanent change of data on Data Base.

4) Aborted state:

   After having any type of failure the transaction goes from “failed state” to “aborted state” and since in previous states, the changes are only made to local buffer or main memory and hence these changes are deleted or rolled-back.

5) Committed state:

   It is the state when the changes are made permanent on the Data Base and the transaction is complete and therefore terminated in the “terminated state”.

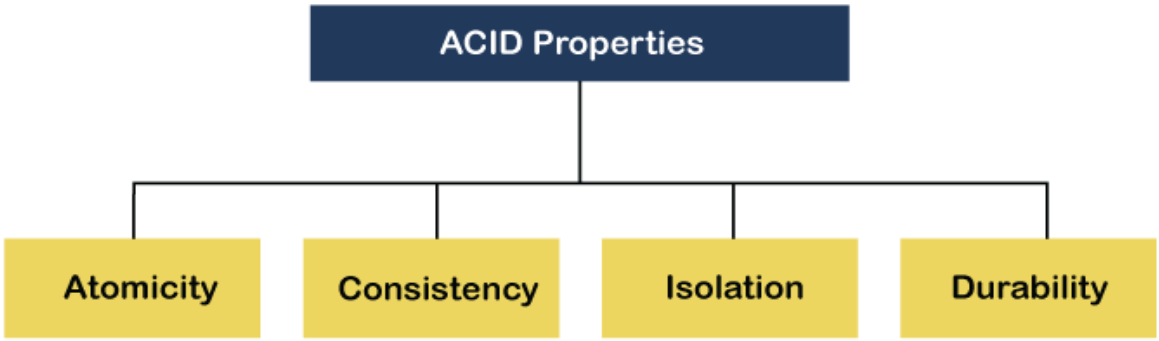
6) Terminated state:

   If there isn’t any roll-back or the transaction comes from the “committed state”, then the system is consistent and ready for new transaction and the old transaction is terminated.

# ACID PROPERTIES IN DBMS

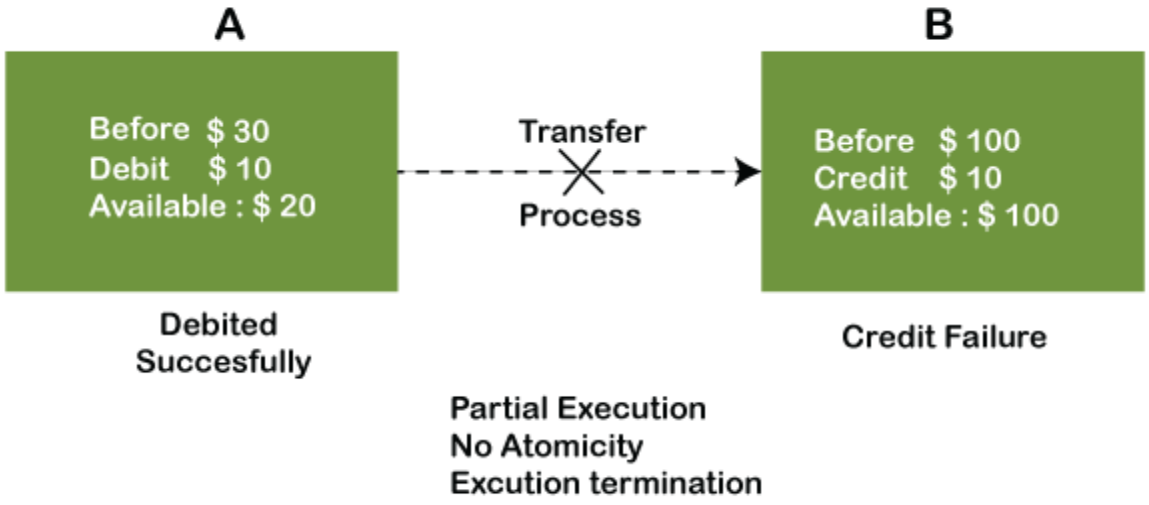
DBMS is the management of data that should remain integrated when any changes are done in it. It is because if the integrity of the data is affected, whole data will get disturbed and corrupted. Therefore, to maintain the integrity of the data, there are four properties described in the database management system, which are known as the **ACID** properties. The ACID properties are meant for the transaction that goes through a different group of tasks, and there we come to see the role of the ACID properties.

The expansion of the term ACID defines for:



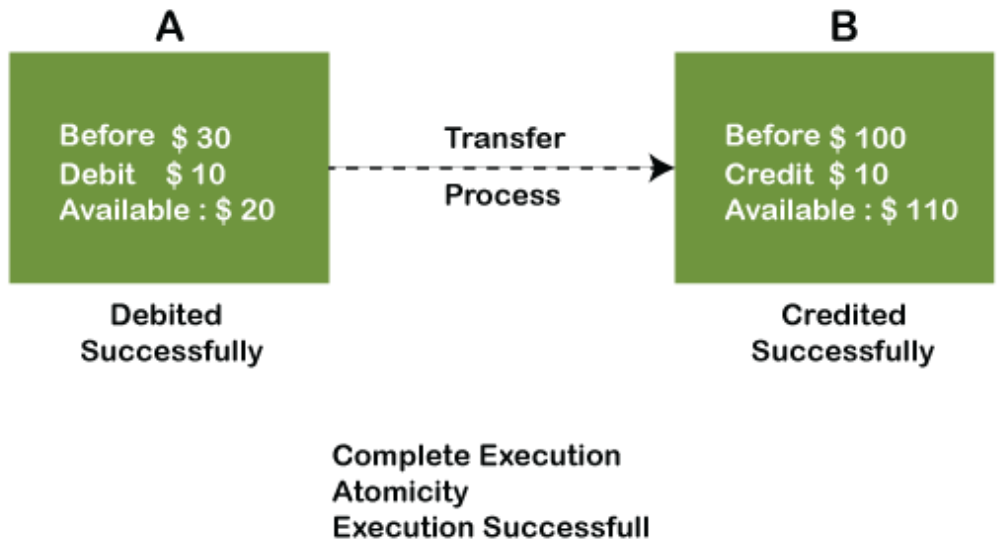
**1)Atomicity:** The term atomicity defines that the data remains atomic. It means if any operation is performed on the data, either it should be performed or executed completely or should not be executed at all. It further means that the operation should not break in between or execute partially. In the case of executing operations on the transaction, the operation should be completely executed and not partially.

**Example:**
If Remo has account A having $30 in his account from which he wishes to send $10 to Stereo's account, which is B. In account B, a sum of $ 100 is already present. When $10 will be transferred to account B, the sum will become $110. Now, there will be two operations that will take place. One is the amount of $10 that Remo wants to transfer will be debited from his account A, and the same amount will get credited to account B, i.e., into Stereo's account. Now, what happens - the first operation of debit executes successfully, but the credit operation, however, fails. Thus, in Remo's account A, the value becomes $20, and to that of Stereo's account, it remains $100 as it was previously present.




In the above diagram, it can be seen that after crediting $10, the amount is still $100 in account B. So, it is not an atomic transaction.

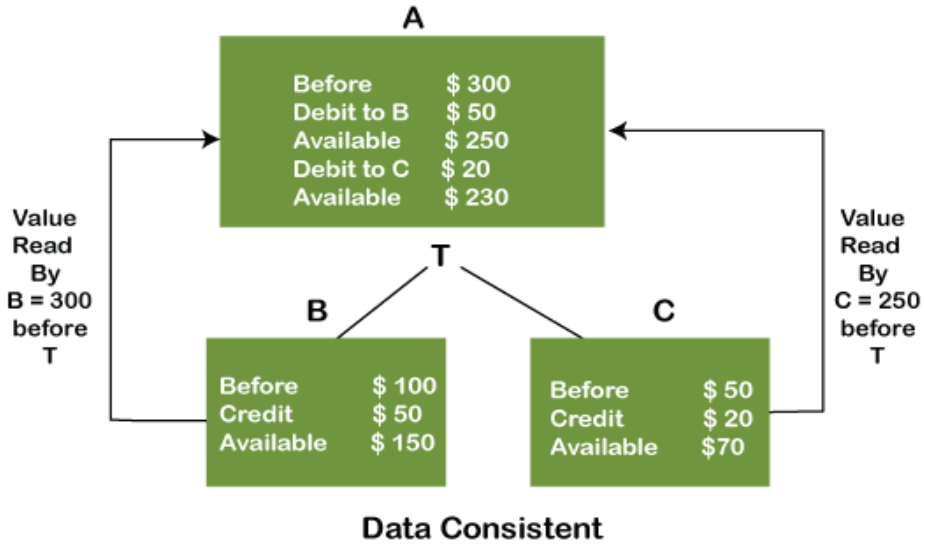
The below image shows that both debit and credit operations are done successfully. Thus the transaction is atomic.




Thus, when the amount loses atomicity, then in the bank systems, this becomes a huge issue, and so the atomicity is the main focus in the bank systems.

**2)Consistency:** The word **consistency** means that the value should remain preserved always. In DBMS, the integrity of the data should be maintained, which means if a change in the database is made, it should remain preserved always. In the case of transactions, the integrity of the data is very essential so that the database remains consistent before and after the transaction. The data should always be correct.

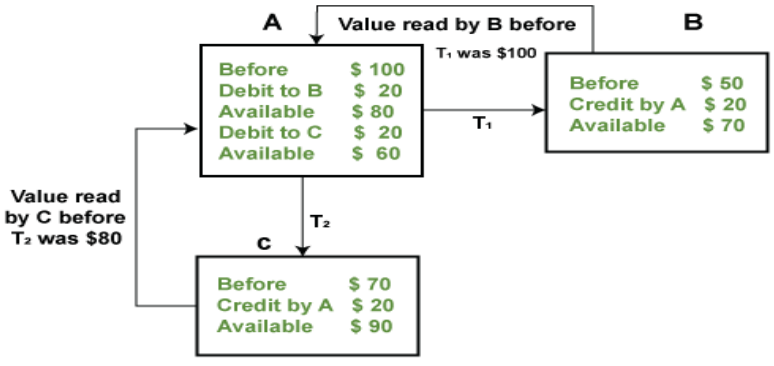
**Example:**



In the above figure, there are three accounts, A, B, and C, where A is making a transaction T one by one to both B & C. There are two operations that take place, i.e., Debit and Credit. Account A firstly debits $50 to account B, and the amount in account A is read $300 by B before the transaction. After the successful transaction T, the available amount in B becomes $150. Now, A debits $20 to account C, and that time, the value read by C is $250 (that is correct as a debit of $50 has been successfully done to B). The debit and credit operation from account A to C has been done successfully. We can see that the transaction is done successfully, and the value is also read correctly. Thus, the data is consistent. In case the value read by B and C is $300, which means that data is inconsistent because when the debit operation executes, it will not be consistent.

**4) Isolation:** The term 'isolation' means separation. In DBMS, Isolation is the property of a database where no data should affect the other one and may occur concurrently. In short, the operation on one database should begin when the operation on the first database gets complete. It means if two operations are being performed on two different databases, they may not affect the value of one another. In the case of transactions, when two or more transactions occur simultaneously, the consistency should remain maintained. Any changes that occur in any particular transaction will not be seen by other transactions until the change is not committed in the memory.

**Example:** If two operations are concurrently running on two different accounts, then the value of both accounts should not get affected. The value should remain persistent. As you can see in the below diagram, account A is making T1 and T2 transactions to account B and C, but both are executing independently without affecting each other. It is known as Isolation.




Isolation - Independent execution of $T_1$ & $T_2$ by A

**4) Durability:** Durability ensures the permanency of something. In DBMS, the term durability ensures that the data after the successful execution of the operation becomes permanent in the database. The durability of the data should be so perfect that even if the system fails or leads to a crash, the database still survives. However, if gets lost, it becomes the responsibility of the recovery manager for ensuring the durability of the database. For committing the values, the COMMIT command must be used every time we make changes.

Therefore, the ACID property of DBMS plays a vital role in maintaining the consistency and availability of data in the database.

# CONCURRENT EXECUTION OF TRANSATION, CONCURRENCY CONTROL

## CONCURRENT EXECUTION OF TRANSATION

Transaction-processing systems usually allow multiple transactions to run concurrently. Allowing multiple transactions to update data concurrently causes several complications with consistency of the data.

Ensuring consistency in spite of concurrent execution of transactions requires extra work; it is far easier to insist that transactions run serially—that is, one at a time, each starting only after the previous one has completed.

However, there are two good reasons for allowing concurrency:

Improved throughput and resource utilization:

- A transaction consists of many steps. Some involve I/O activity; others involve CPU activity. The CPU and the disks in a computer system can operate in parallel. Therefore, I/O activity can be done in parallel with processing at the CPU.
- The parallelism of the CPU and the I/O system can therefore be exploited to run multiple transactions in parallel.
- While a read or write on behalf of one transaction is in progress on one disk, another transaction can be running in the CPU, while another disk may be executing a read or write on behalf of a third transaction.
- All of this increases the throughput of the system—that is, the number of transactions executed in a given amount of time.
- Correspondingly, the processor and disk utilization also increase; in other words, the processor and disk spend less time idle, or not performing any useful work.

Reduced waiting time:

- There may be a mix of transactions running on a system, some short and some long.
- If transactions run serially, a short transaction may have to wait for a preceding long transaction to complete, which can lead to unpredictable delays in running a transaction.

- If the transactions are operating on different parts of the database, it is better to let them run concurrently, sharing the CPU cycles and disk accesses among them.
- Concurrent execution reduces the unpredictable delays in running transactions.
- Moreover, it also reduces the average response time: the average time for a transaction to be completed after it has been submitted.

The idea behind using concurrent execution in a database is essentially the same

as the idea behind using multi programming in an operating system.

The database system must control the interaction among the concurrent transactions to prevent them from destroying the consistency of the database. It is achieved using **concurrency-control schemes**.

## DBMS Schedule

A schedule is a process of combining the multiple transactions into one and executing the operations of these transactions in a predefined order. A schedule can have multiple transactions in it, each transaction comprising of several tasks or operations or instructions.

A schedule can also be defined as “a sequence of operations of multiple transactions that appears for execution”. Or we can say that when several transactions are executed concurrently in the database, then the order of execution of these transactions is known as a schedule. A specific sequence of operations of a set of instructions is called a schedule.

**Schedule Example**

**Schedule 1:**

| **Time** | **Transaction T1** | **Transaction T2** |
|---|---|---|
| t1 | Read(A) | |
| t2 | A=A+50 | |
| t3 | Write(A) | |

| | | |
|---|---|---|
| t4 | | Read(A) |
| t5 | | A+A+100 |
| t6 | | Write(A) |
| t7 | Read(B) | |
| t8 | B=B+100 | |
| t9 | Write(B) | |
| t10 | | Read(B) |
| t11 | | B=B+50 |
| t12 | | Write(B) |

**Types of Schedule**

Schedule is of two types:

- Serial schedule
- Non-serial schedule

**Serial Schedule**

The serial schedule is a type of schedule in which the transactions are executed one after other without interleaving. It is a schedule in which multiple transactions are organized in such a way that one transaction is executed first. When the operations of first transaction complete, then next transaction is executed.

In a serial schedule, when one transaction executes its operation, then no other transaction is allowed to execute its operations.

**For example:**
Suppose a schedule S with two transactions T1 and T2. If all the instructions of T1 are executed before T2 or all the instructions of T2 are executed before T1, then S is said to be a serial schedule.
Below tables shows the example of a serial schedule, in which the operations of first transaction are executed first before starting another transactionT2:

| **Time** | **Transaction T1** | **Transaction T2** |
|---|---|---|
| t1 | Read(A) | |
| t2 | A=A+50 | |
| t3 | Write(A) | |
| t4 | Read(B) | |
| t5 | B=B+100 | |
| t6 | Write(B) | |
| t7 | | Read(A) |
| t8 | | A+A+100 |
| t9 | | Write(A) |
| t10 | | Read(B) |

| | | |
|---|---|---|
| t11 | | B=B+50 |
| t12 | | Write(B) |

**Non-Serial Schedule**
The non-serial schedule is a type of schedule where the operations of multiple transactions are interleaved. Unlike the serial schedule, this schedule proceeds without waiting for the execution of the previous transaction to complete. This schedule may give rise to the problem of concurrency. In a non-serial schedule multiple transactions are executed concurrently.

| **Time** | **Transaction T1** | **Transaction T2** |
|---|---|---|
| t1 | Read(A) | |
| t2 | | Read(A) |
| t3 | A=A+50 | |
| t4 | | A+A+100 |
| t5 | Write(A) | |
| t6 | | Write(A) |

In this schedule S, there aretwo transaction T1 and T2. If T1 and T2 transactions are executed concurrently, and the operations of T1 and T2 are interleaved. So, this schedule is an example of a non-serial schedule.

## DBMS Serializability

When multiple transactions are running concurrently then there is a possibility that the database may be left in an inconsistent state. Serializability is a concept that helps us to check which schedules are serializable. A serializable schedule is the one that always leaves the database in consistent state

## What is a serializable schedule?

A serializable schedule always leaves the database in consistent state. A serial schedule is always a serializable schedule because in serial schedule, a transaction only starts when the other transaction finished execution. However a non-serial schedule needs to be checked for Serializability.

A non-serial schedule of n number of transactions is said to be serializable schedule, if it is equivalent to the serial schedule of those n transactions. A serial schedule doesn't allow concurrency, only one transaction executes at a time and the other starts when the already running transaction finished.

## Types of Serializability

There are two types of Serializability

1.Conflict Serializability
2. View Serializability

EXAMPLE:SERIALIZABLE SCHEDULE

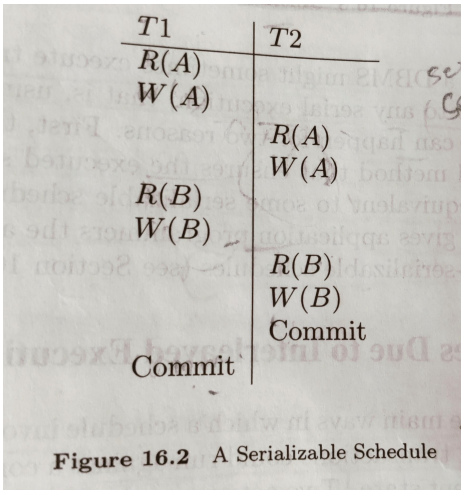
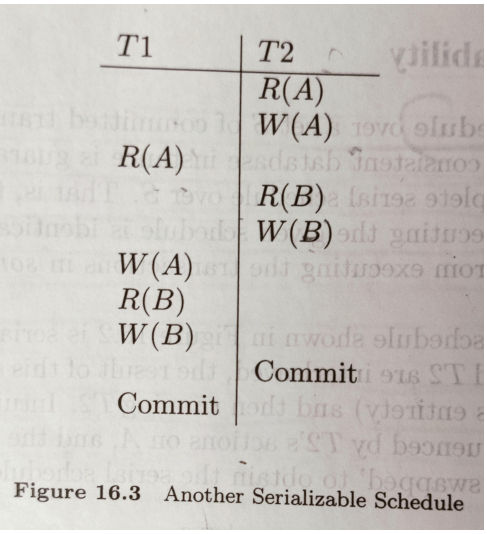


Figure 16.2 A Serializable Schedule



| T1 | T2 |
|---|---|
| | R(A) |
| | W(A) |
| R(A) | |
| | R(B) |
| | W(B) |
| W(A) | |
| R(B) | |
| W(B) | |
| | Commit |
| Commit | |

Figure 16.3 Another Serializable Schedule

## Serial Schedules Vs Serializable Schedules-

| Serial Schedules | Serializable Schedules |
|---|---|
| No concurrency is allowed.<br>Thus, all the transactions necessarily execute serially one after the other. | Concurrency is allowed.<br>Thus, multiple transactions can execute concurrently. |
| Serial schedules lead to less resource utilization and CPU throughput. | Serializable schedules improve both resource utilization and CPU throughput. |
| Serial Schedules are less efficient as compared to serializable schedules.<br>(due to above reason) | Serializable Schedules are always better than serial schedules.<br>(due to above reason) |

**Concurrency problems**

Several problems can occur when concurrent transactions are run in an uncontrolled manner, such type of problems is known as concurrency problems. There are following different types of problems or conflicts which occur due to concurrent execution of transaction:

**Lost update problem (Write – Write conflict)**

This type of problem occurs when two transactions in database access the same data item and have their operations in an interleaved manner that makes the value of some database item incorrect.

If there are two transactions T1 and T2 accessing the same data item value and then update it, then the second record overwrites the first record.
**Example:** Let's take the value of A is 100

| Time | Transaction T1 | Transaction T2 |
|---|---|---|
| t1 | Read(A) | |
| t2 | A=A-50 | |
| t3 | | Read(A) |
| t4 | | A=A+50 |
| t5 | Write(A) | |
| t6 | | Write(A) |

**Here,**

- At t1 time, T1 transaction reads the value of A i.e., 100.
- At t2 time, T1 transaction deducts the value of A by 50.
- At t3 time, T2 transactions read the value of A i.e., 100.
- At t4 time, T2 transaction adds the value of A by 150.
- At t5 time, T1 transaction writes the value of A data item on the basis of value seen at time t2 i.e., 50.
- At t6 time, T2 transaction writes the value of A based on value seen at time t4 i.e., 150.
- So at time T6, the update of Transaction T1 is lost because Transaction T2 overwrites the value of A without looking at its current value.
- Such type of problem is known as the Lost Update Problem.

**Dirty read problem (W-R conflict)**

This type of problem occurs when one transaction T1 updates a data item of the database, and then that transaction fails due to some reason, but its updates are accessed by some other transaction.
**Example:** Let's take the value of A is 100

| Time | Transaction T1 | Transaction T2 |
|---|---|---|
| t1 | Read(A) | |
| t2 | A=A+20 | |
| t3 | Write(A) | |
| t4 | | Read(A) |
| t5 | | A=A+30 |
| t6 | | Write(A) |
| t7 | Write(B) | |

**Here,**

- At t1 time, T1 transaction reads the value of A i.e., 100.
- At t2 time, T1 transaction adds the value of A by 20.
- At t3 time, T1transaction writes the value of A (120) in the database.
- At t4 time, T2 transactions read the value of A data item i.e., 120.
- At t5 time, T2 transaction adds the value of A data item by 30.
- At t6 time, T2transaction writes the value of A (150) in the database.
- At t7 time, a T1 transaction fails due to power failure then it is rollback according to atomicity property of transaction (either all or none).
- So, transaction T2 at t4 time contains a value which has not been committed in the database. The value read by the transaction T2 is known as a dirty read.

**Unrepeatable read (R-W Conflict)**

It is also known as an inconsistent retrieval problem. If a transaction $T_1$ reads a value of data item twice and the data item is changed by another transaction $T_2$ in between the two read operation. Hence $T_1$ access two different values for its two read operation of the same data item.
**Example:** Let's take the value of A is 100

| Time | Transaction T1 | Transaction T2 |
|---|---|---|
| t1 | Read(A) | |
| t2 | | Read(A) |
| t3 | | A=A+30 |
| t4 | | Write(A) |
| t5 | Read(A) | |

**Here,**

- At t1 time, T1 transaction reads the value of A i.e., 100.
- At t2 time, T2transaction reads the value of A i.e., 100.
- At t3 time, T2 transaction adds the value of A data item by 30.
- At t4 time, T2 transaction writes the value of A (130) in the database.
- Transaction T2 updates the value of A. Thus, when another read statement is performed by transaction T1, it accesses the new value of A, which was updated by T2. Such type of conflict is known as R-W conflict.

## CONCURRENCY CONTROL

Concurrency Control is the working concept that is required for controlling and managing the concurrent execution of database operations and thus avoiding the inconsistencies in the database. Thus, for maintaining the concurrency of the database, we have the concurrency control protocols.

Concurrency Control Protocols

The concurrency control protocols ensure the *atomicity, consistency, isolation, durability* and *serializability* of the concurrent execution of the database transactions. Therefore, these protocols are categorized as:

- Lock Based Concurrency Control Protocol
- Time Stamp Concurrency Control Protocol
- Validation Based Concurrency Control Protocol

- **Atomicity** − This property states that a transaction must be treated as an atomic unit, that is, either all of its operations are executed or none. There must be no state in a database where a transaction is left partially completed. States should be defined either before the execution of the transaction or after the execution/abortion/failure of the transaction.
- **Consistency** − The database must remain in a consistent state after any transaction. No transaction should have any adverse effect on the data residing in the database. If the database was in a consistent state before the execution of a transaction, it must remain consistent after the execution of the transaction as well.
- **Durability** − The database should be durable enough to hold all its latest updates even if the system fails or restarts. If a transaction updates a chunk of data in a database and commits, then the database will hold the modified data. If a transaction commits but the system fails before the data could be written on to the disk, then that data will be updated once the system springs back into action.
- **Isolation** − In a database system where more than one transaction are being executed simultaneously and in parallel, the property of isolation states that all the transactions will be carried out and executed as if it is the only transaction in the system. No transaction will affect the existence of any other transaction.

Lock-Based Protocol

In this type of protocol, any transaction cannot read or write data until it acquires an appropriate lock on it. There are two types of lock:

## 1. Shared lock:

- It is also known as a Read-only lock. In a shared lock, the data item can only read by the transaction.
- It can be shared between the transactions because when the transaction holds a lock, then it can't update the data on the data item.

## 2. Exclusive lock:

- In the exclusive lock, the data item can be both reads as well as written by the transaction.
- This lock is exclusive, and in this lock, multiple transactions do not modify the same data simultaneously.

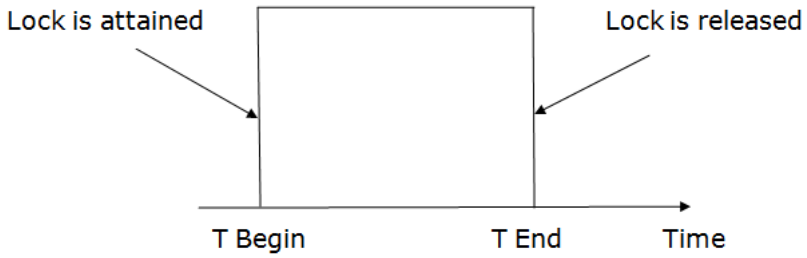
There are four types of lock protocols available:

### 1. Simplistic lock protocol

It is the simplest way of locking the data while transaction. Simplistic lock-based protocols allow all the transactions to get the lock on the data before insert or delete or update on it. It will unlock the data item after completing the transaction.

### 2. Pre-claiming Lock Protocol

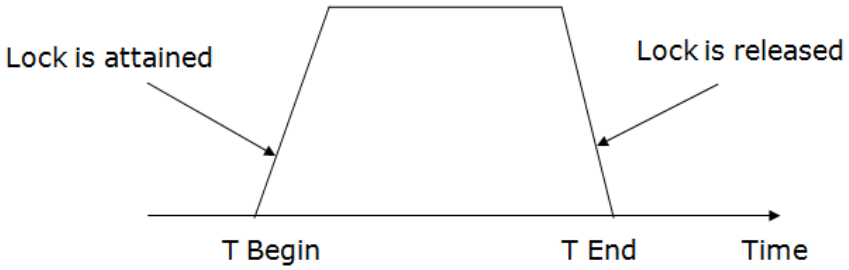
- Pre-claiming Lock Protocols evaluate the transaction to list all the data items on which they need locks.
- Before initiating an execution of the transaction, it requests DBMS for all the lock on all those data items.
- If all the locks are granted then this protocol allows the transaction to begin. When the transaction is completed then it releases all the lock.
- If all the locks are not granted then this protocol allows the transaction to rolls back and waits until all the locks are granted.




### 3. Two-phase locking (2PL)

- The two-phase locking protocol divides the execution phase of the transaction into three parts.

- In the first part, when the execution of the transaction starts, it seeks permission for the lock it requires.
- In the second part, the transaction acquires all the locks. The third phase is started as soon as the transaction releases its first lock.
- In the third phase, the transaction cannot demand any new locks. It only releases the acquired locks.




There are two phases of 2PL:

**Growing phase:** In the growing phase, a new lock on the data item may be acquired by the transaction, but none can be released.

**Shrinking phase:** In the shrinking phase, existing lock held by the transaction may be released, but no new locks can be acquired.

In the below example, if lock conversion is allowed then the following phase can happen:

1. Upgrading of lock (from S(a) to X (a)) is allowed in growing phase.
2. Downgrading of lock (from X(a) to S(a)) must be done in shrinking phase.

**Example:**

| | T1 | T2 |
|---|---|---|
| 0 | LOCK-S(A) | |
| 1 | | LOCK-S(A) |
| 2 | LOCK-X(B) | |
| 3 | —— | —— |
| 4 | UNLOCK(A) | |
| 5 | | LOCK-X(C) |
| 6 | UNLOCK(B) | |
| 7 | | UNLOCK(A) |
| 8 | | UNLOCK(C) |
| 9 | —— | —— |

The following way shows how unlocking and locking work with 2-PL.

**Transaction T1:**

- **Growing phase:** from step 1-3
- **Shrinking phase:** from step 5-7
- **Lock point:** at 3

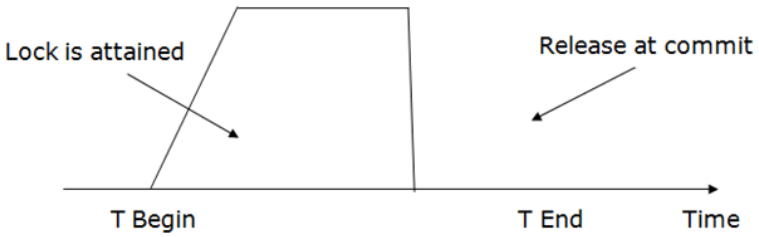
**Transaction T2:**

- **Growing phase:** from step 2-6
- **Shrinking phase:** from step 8-9
- **Lock point:** at 6

4. Strict Two-phase locking (Strict-2PL)

- The first phase of Strict-2PL is similar to 2PL. In the first phase, after acquiring all the locks, the transaction continues to execute normally.
- The only difference between 2PL and strict 2PL is that Strict-2PL does not release a lock after using it.
- Strict-2PL waits until the whole transaction to commit, and then it releases all the locks at a time.

- Strict-2PL protocol does not have shrinking phase of lock release.




It does not have cascading abort as 2PL does.

# DEADLOCK

In a multi-process system, deadlock is an unwanted situation that arises in a shared resource environment, where a process indefinitely waits for a resource that is held by another process.

For example, assume a set of transactions $\{T_0, T_1, T_2, ...,T_n\}$. $T_0$ needs a resource X to complete its task. Resource X is held by $T_1$, and $T_1$ is waiting for a resource Y, which is held by $T_2$. $T_2$ is waiting for resource Z, which is held by $T_0$. Thus, all the processes wait for each other to release resources. In this situation, none of the processes can finish their task. This situation is known as a deadlock.

Deadlocks are not healthy for a system. In case a system is stuck in a deadlock, the transactions involved in the deadlock are either rolled back or restarted.

A simple way to identify deadlocks is to use a timeout mechanism. If a transaction has been waiting too long for a lock, we can assume (pessimistically) that it is in a deadlock cycle and abort it.

There are two method of dealing with deadlock.

***1. Deadlock Prevention***

***2. Deadlock Detection & Recovery***

Deadlock Prevention protocol to ensure that the system will never enter a deadlock state. Alternately we can allow the system enter a deadlock state and then try to recover them by using a deadlock detection and deadlock recovery scheme. Prevention is commonly used if the probability that the system would enter a deadlock state is relatively high; otherwise detection and recovery are more efficient.

Deadlock Prevention

There are two approaches for deadlock prevention.

- One approach ensure that no cyclic waits can occur by ordering the request for locks.
- Second approach is the rollback of transactions

- Another approach for preventing deadlock is to impose an ordering of all data items and to require that transaction lock data item in sequence.

To prevent any deadlock situation in the system, the DBMS aggressively inspects all the operations, where transactions are about to execute. The DBMS inspects the operations and analyzes if they can create a deadlock situation. If it finds that a deadlock situation might occur, then that transaction is never allowed to be executed.

## Preemption-Prevention to Deadlock

The second approach for prevention of deadlock is **preemption and transaction rollback.**

**In preemption,** when a transaction T2 request a lock that T1 holds, the lock granted to T1 is preempted by rolling back of T1 and granting the lock to T2.

To control the preemption, we can assign a unique timestamp, based on a counter or system clock, to each transaction when it begins.

The system uses these timestamp only to decide whether a transaction should wait or rollback.

## Deadlock Prevention Technique Using Timestamp

There are deadlock prevention schemes that use timestamp ordering mechanism of transactions in order to predetermine a deadlock situation.

### Wait-Die Scheme

In this scheme, if a transaction requests to lock a resource (data item), which is already held with a conflicting lock by another transaction, then one of the two possibilities may occur −

- If $TS(T_i) < TS(T_j)$ − that is $T_i$, which is requesting a conflicting lock, is older than $T_j$ − then $T_i$ is allowed to wait until the data-item is available.
- If $TS(T_i) > TS(t_j)$ − that is $T_i$ is younger than $T_j$ − then $T_i$ dies. $T_i$ is restarted later with a random delay but with the same timestamp.

This scheme allows the older transaction to wait but kills the younger one.

**Wound-Wait Scheme**

In this scheme, if a transaction requests to lock a resource (data item), which is already held with conflicting lock by some another transaction, one of the two possibilities may occur −

- If $TS(T_i) < TS(T_j)$, then $T_i$ forces $T_j$ to be rolled back − that is $T_i$ wounds $T_j$. $T_j$ is restarted later with a random delay but with the same timestamp.
- If $TS(T_i) > TS(T_j)$, then $T_i$ is forced to wait until the resource is available.

This scheme, allows the younger transaction to wait; but when an older transaction requests an item held by a younger one, the older transaction forces the younger one to abort and release the item.

In both the cases, the transaction that enters the system at a later stage is aborted.

*The major problem with both of the scheme is unnecessary rollbacks may occur.*

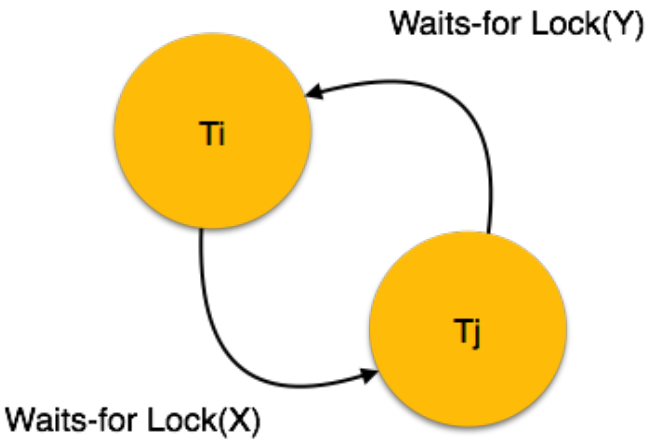
## Deadlock Avoidance

Aborting a transaction is not always a practical approach. Instead, deadlock avoidance mechanisms can be used to detect any deadlock situation in advance. Methods like "wait-for graph" are available but they are suitable for only those systems where transactions are lightweight having fewer instances of resource. In a bulky system, deadlock prevention techniques may work well.

**Wait-for Graph**

This is a simple method available to track if any deadlock situation may arise. For each transaction entering into the system, a node is created. When a transaction $T_i$ requests for a lock on an item, say X, which is held by some other transaction $T_j$, a directed edge is created from $T_i$ to $T_j$. If $T_j$ releases item X, the edge between them is dropped and $T_i$ locks the data item.

The system maintains this wait-for graph for every transaction waiting for some data items held by others. The system keeps checking if there's any cycle in the graph.




Here, we can use any of the two following approaches −

- First, do not allow any request for an item, which is already locked by another transaction. This is not always feasible and may cause starvation, where a transaction indefinitely waits for a data item and can never acquire it.
- The second option is to roll back one of the transactions. It is not always feasible to roll back the younger transaction, as it may be important than the older one. With the help of some relative algorithm, a transaction is chosen, which is to be aborted. This transaction is known as the **victim** and the process is known as **victim selection**.

## Lock Timeouts

Another simple approach to deadlock prevention is Lock Timeouts.

In this approach, a transaction that has requested a lock waits for at most a specified amount of time.

If the lock has not been granted within that time, the transaction is said to be timeout, and it rolls itself back and restarts.

Timeout scheme is easy to implement, and works well if the transactions are short.

However in general it is hard to decide how long a transaction has to wait before time out.

Timeout scheme has limited applicability.

## Deadlock Detection & Recovery

## Deadlock Detection:-

Deadlock can be described precisely in terms of a directed graph called Wait-for Graph.

This graph consist of pair G= (V, E), where V is the set of Vertices and E is the set of edges. The set of vertices consists of all the transactions in the system.

Each elements in the set E of edges is an ordered pair Ti  Tj.

If Ti  Tj is in E, then there is a directed connection from Ti to Tj implying that transaction Ti is waiting for Tj to release a data item that it needs.

**Working:-**

When the transaction Ti request for a data item currently held by transaction Tj, then the edge Ti  Tj is inserted in a wait – for graph. This edge is removed only when transaction Tj is no longer holding the data item needed by transaction Ti.

Deadlock occurs if and only if a wait-for graph contains a cycle.

To detect deadlock, the system needs to maintain a wait-for graph, and periodically to invoke an algorithm that searches for a cycle in the graph.

## INTRODUCTION TO CRASH RECOVERY

The recovery manager of a DBMS is responsible for ensuring transaction atomicity and durability. It ensures atomicity by undoing the actions of transactions that do not commit, and durability by making sure that all actions of committed transactions survive system crashes, (e.g., a core dump caused by a bus error) and media failures (e.g., a disk is corrupted).

Then a DBMS is restarted after crashes. The recovery manager is given control and must bring the database to a consistent state. The recovery manager is also responsible for undoing the actions of an aborted transaction. To see what it takes to implement a recovery manager, it is necessary to understand what happens during normal execution.

The transaction manager of a DBMS controls the execution of transactions. Before reading and writing objects during normal execution, locks must be acquired (and

released at some later time) according to a chosen locking protocol. For simplicity of exposition, we make the following assumption:

**Atomic Writes:** Writing a page to disk is an atomic action.

This implies that the system does not crash while a write is in progress and is unrealistic. In practice, disk writes do not have this property, and steps must be taken during restart after a crash to verify that the most recent write to a given page was completed successfully, and to deal with the consequences if not.

**Stealing Frames and Forcing Pages**

With respect to writing objects, two additional questions arise:

1. Can the changes made to an object 0 in the buffer pool by a transaction T be written to disk before T commits? Such writes are executed when another transaction wants to bring in a page and the buffer manager chooses to replace the frame containing 0; of course, this page must have been unpinned by T. If such writes are allowed, we say that a steal approach is used. (Informally, the second transaction '**steals**' a frame from T.)
2. When a transaction commits, must we ensure that all the changes it has made to objects in the buffer pool are immediately forced to disk? If so. We say that a **force** approach is used.

From the standpoint of implementing a recovery manager, it is simplest to use a buffer manager with a no-steal force approach. If a no-steal approach is used, we do not have to undo the changes of an aborted transaction (because these changes have not been written to disk) l and if a force approach is used, we do not have to redo the changes of a committed transaction if there is a subsequent crash (because all these changes are guaranteed to have been written to disk at commit time).

However, these policies have important drawbacks. The no-steal approach assumes that all pages modified by ongoing transactions can be accommodated in the buffer pool, and in the presence of large transactions (typically run in batch mode, e.g., payroll processing), this assumption is unrealistic. The force approach results in excessive page I/O costs. If a highly used page is updated in succession by 20 transactions, it would be written to disk 20 times. With a no-force approach, on the other hand, the in-memory copy of the page would be successively modified and written to disk just once, reflecting the effects of all 20 updates, when the page is

eventually replaced in the buffer pool (in accordance with the buffer manager's page replacement policy).

For these reasons, most systems use a steal, no-force approach. Thus, if a frame is dirty and chosen for replacement, the page it contains is written to disk even if the modifying transaction is still active *(steal);* in addition, pages in the buffer pool that are modified by a transaction are not forced to disk when the transaction commits *(no-force)*.

## Recovery-Related Steps during Normal Execution

The recovery manager of a DBMS maintains some information during normal execution of transactions to enable it to perform its task in the event of a failure. In particular, a log of all modifications to the database is saved on stable storage, which is guaranteed to survive crashes and media failures. Stable storage is implemented by maintaining multiple copies of information (perhaps in different locations) on nonvolatile storage devices such as disks or tapes.

It is important to ensure that the log entries describing a change to the database are written to stable storage *before* the change is made; otherwise, the system might crash just after the change, leaving us without a record of the change. (Write-Ahead Log, or WAL, property.)

The log enables the recovery manager to undo the actions of aborted and incomplete transactions and redo the actions of committed transactions. For example, a transaction that committed before the crash may have made updates to a copy (of a database object) in the buffer pool, and this change may not have been written to disk before the crash, because of a no-force approach. Such changes must be identified using the log and written to disk. Further, changes of transactions that did not commit prior to the crash might have been written to disk because of a steal approach. Such changes must be identified using the log and then undone.

The amount of work involved during recovery is proportional to the changes made by committed transactions that have not been written to disk at the time of the crash. To reduce the time to recover from a crash, the DBMS periodically forces buffer pages to disk during normal execution using a background process (while making sure that any log entries that describe changes these pages are written to

disk first, i.e., following the WAL protocol). A process called *checkpointing,* which saves information about active transactions and dirty buffer pool pages, also helps reduce the time taken to recover from a crash.

## Overview of ARIES

ARIES is a recovery algorithm that is designed to work with a steal, no-force approach. When the recovery manager is invoked after a crash, restart proceeds in three phases. In the Analysis phase, it identifies dirty pages in the buffer pool (i.e., changes that have not been written to disk) and active transactions at the time of the crash. In the Redo phase, it repeats all actions, starting from an appropriate point in the log, and restores the database state to what it was at the time of the crash. Finally, in the Undo phase, it undoes the actions of transactions that did not commit, so that the database reflects only the actions of committed transactions.

## Atomicity: Implementing Rollback

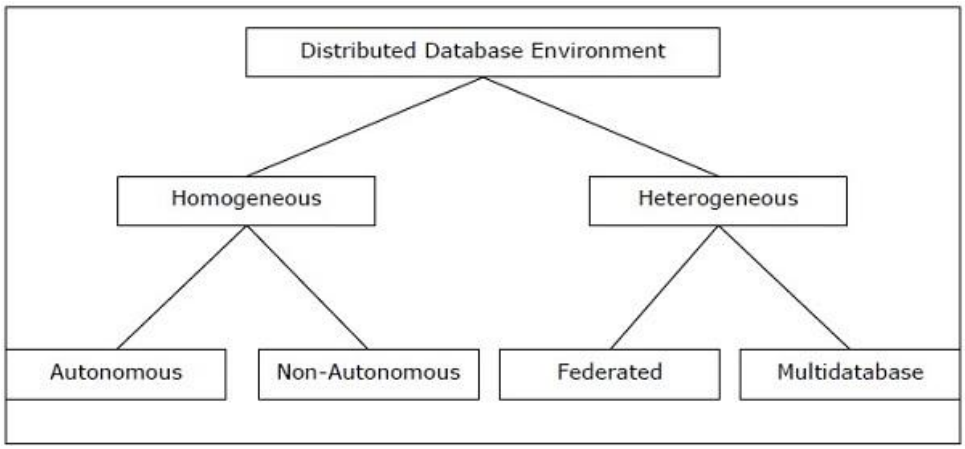
It is important to recognize that the recovery subsystem is also responsible for executing the ROLLBACK command, which aborts a single transaction. Indeed, the logic (and code) involved in undoing a single transaction is identical to that used during the Undo phase in recovering from a system crash. All log records for a given transaction are organized in a linked list and can be efficiently accessed in reverse order to facilitate transaction rollback.

# DISTRIBUTED DATABASE

- A distributed database (ddb) is **an integrated collection of databases that is physically distributed across sites in A computer network**. To form A distributed database system (ddbs), the files must be structured, logically interrelated, and physically distributed across multiple sites.

## TYPES OF DISTRIBUTED DATABASE

Distributed databases can be broadly classified into homogeneous and heterogeneous distributed database environments, each with further subdivisions.




## HOMOGENEOUS AND HETEROGENEOUS DISTRIBUTED SYSTEM

## HOMOGENEOUS

IN A HOMOGENEOUS DISTRIBUTED DATABASE, ALL THE SITES USE IDENTICAL DBMS AND OPERATING SYSTEMS. ITS PROPERTIES ARE −

- THE SITES USE VERY SIMILAR SOFTWARE.
- THE SITES USE IDENTICAL DBMS OR DBMS FROM THE SAME

VENDOR.

- EACH SITE IS AWARE OF ALL OTHER SITES AND COOPERATES WITH OTHER SITES TO PROCESS USER REQUESTS.
- THE DATABASE IS ACCESSED THROUGH A SINGLE INTERFACE AS IF IT IS A SINGLE DATABASE.

## TYPES OF HOMOGENEOUS DISTRIBUTED DATABASE

THERE ARE TWO TYPES OF HOMOGENEOUS DISTRIBUTED DATABASE

- **Autonomous** − each database is independent that functions on its own. They are integrated by a controlling application and use message passing to share data updates.
- **Non-autonomous** − data is distributed across the homogeneous nodes and a central or master dbms co-ordinates data updates across the sites.

# HETROGENEOUS

## HETEROGENEOUS DISTRIBUTED DATABASES

In a heterogeneous distributed database, different sites have different operating systems, DBMS products and data models. Its properties are −

- Different sites use dissimilar schemas and software.
- The system may be composed of a variety of dbmss like relational, network, hierarchical or object oriented.
- Query processing is complex due to dissimilar schemas.
- Transaction processing is complex due to dissimilar software.
- A site may not be aware of other sites and so there is limited co-operation in processing user

## TYPES OF HETEROGENEOUS DISTRIBUTED DATABASES

- **Federated** − the heterogeneous database systems are independent in nature and integrated together so that they function as a single database system.
- **Un-federated** − the database systems employ a central coordinating module through which the databases are accessed.

# DISTRIBUTED DBMS ARCHITECTURES

Three alternative approach are used to separate functionally across different database related process

Alternative distributed dbms architecture are called

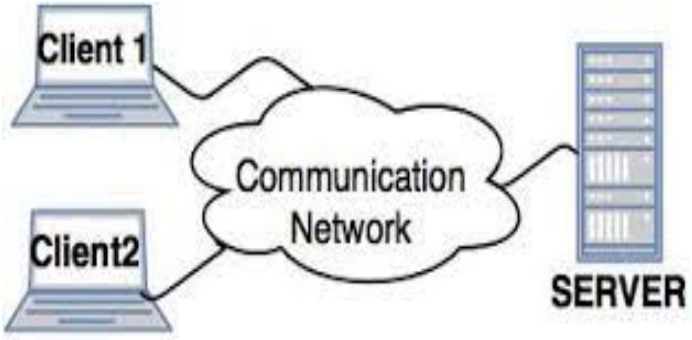
- Client-server-architecture
- Collaborating server
- Middleware

## CLIENT-SERVER SYSTEMS

- A client-server system has one or more client processes and one or more server processes, and a client process can send a query to any one server process.
- Client are responsible for user-interface issues, server manages data and execute transactions.
- Client process could run on a personal computer and send queries to server running on a mainframe.
- While writing client server applications, it is important to remember the boundary between the client and server and keep the communication between

them as set-oriented as possible.

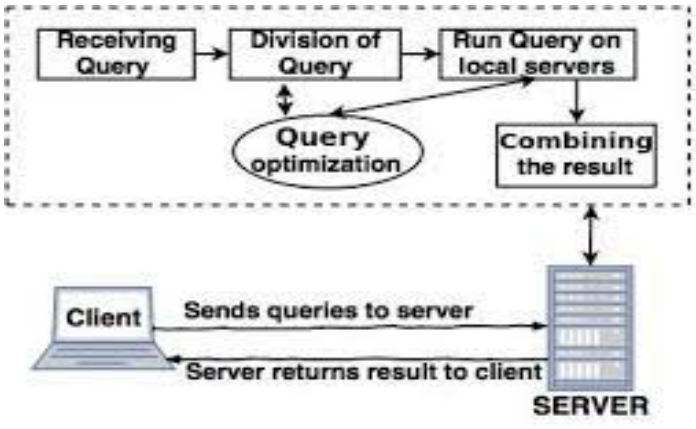
**EXAMPLE**




Client-server architecture

## COLLABORATING SERVER SYSTEMS

- Collaborating server architecture is **designed to run a single query on multiple servers**. Servers break single query into multiple small queries and the result is sent to the client. Collaborating server architecture has a collection of database servers.




Collabrorating server architecture

## MIDDLE WARE ARCHITECTURE

- Middleware sits "**in the middle" between application software** that may be working on different operating systems. ... It is similar to the middle layer of a three-tier single system architecture, except that it is stretched across multiple systems or applications.

## STORING DATA IN DISTRIBUTED DBMS

- In distributed DBMS, relations are stored across several sites.
- Accessing a relation stored at a remote site incurs message-passing cost and to reduce this overhead, a single relation may be partitioned or fragmented across several sites.

## FRAGMENTATION

- Fragmentation consists of breaking a relation into smaller relations or fragments and storing fragments possibly at different sites.
- In horizontal fragmentation, each fragment consist of a subset of rows of the original relation.
- In vertical fragmentation, each fragments consist of subset of columns of original relation.

## REPLICATION

- Replication means that we store several copies of a relation or relation fragments.An entire relation can be replicated at one or more sites. Similarly,one or more fragments can be replicated at other sites.
- The motivation of replication is twofold :

- -increased availability of data :- if a site that contains a replica goes down, we can find the same data at other sites.
- -Faster query evaluation :- queries can execute faster by using a local copy of a relation instead of going to a remote site.

# INTRODUCTION TO NOSQL

A noSQL orginally reffering to non SQL or non relational is a database that provide a machanism for storage and retrieval of data.This data is modeled in means other than the tabular relations used in relational databases.Its a wonderfull irony that the term "No SQL" first made its appearance in the late 90's as the name of an open-source relational database.The name comes from the fact that the database doesn't use SQL as a query language.Instead ,the database is manipulated through shell scripts.

# WHY NOSQL

The concept of NoSQL databases became popular with internet gaints like google,facebook,Amazon etc. who deal with huge volumes of data. The system response timebecomes slow when you use RDBMS for massive volume of data.To resolve this problem we could "scale up" our systems by upgrading our existing hardware and this process is very expensive.The alternative for this issue is to distribute database load on multiple hosts whenever the load increases.This method is known as "scaling out".Apart from this NoSQL database is a non-relational,so it scales out better than relational databases as they are designed with web applications in mind.

# FEATURES OF NOSQL

**Non Relational Database**

- NoSQL databases never follow the relational model.
- Never provide tables with flat fixed-column records
- work with self-contained aggregates or BLOBs
- Doesn't require object-relational mapping and data normalization.
- No complex features like query languages,query planners,referential integrity joins,ACID

**Schema-free**

- NoSQL databases are either schema-free or have relaxed

schemas.

- Do not require any sort of defnition of the schema of the data.
- Offers hetrogeneous structures of data in the same domain.

## WHERE TO USE NOSQL?

- Big Data
- Content Management and Delivery
- Mobile and Social Infrastructure
- User data Management
- Data Hub

## TYPES OF NOSQL DATABASES

NoSQL databases are mainly categorized into four types:

- Key-value pair Based
- coulmn-oriented Graph
- Graph-based
- Document-oriented

Every category has its unique attributes and limitations.None of the above-specified databases is better to solve all the problems.User should select the database based on their product needs.

## KEY VALUE PAIR BASED

- Data is stored in key/value pairs. It is designed in such a way to handle lots of data and heavy load.
- Key-value pair storage databases store data as a hash table where each key is unique, and the value can be a JSON, BLOB(Binary Large Objects), string, etc.
- For example, a key-value pair may contain a key like “Website" associated with a value like “Facebook".

| Key | Value |
|---|---|
| Name | Joe Bloggs |
| Age | 42 |
| Occupation | Stunt Double |
| Height | 175cm |
| Weight | 77kg |

- It is one of the most basic NoSQL database example.
- This kind of NoSQL database is used as a collection, dictionaries, associative arrays, etc. Key value stores help the developer to store schema-less data.
- They work best for shopping cart contents.
- Redis, Dynamo, Riak are some NoSQL examples of key-value store DataBases.

## COLOUMN BASED

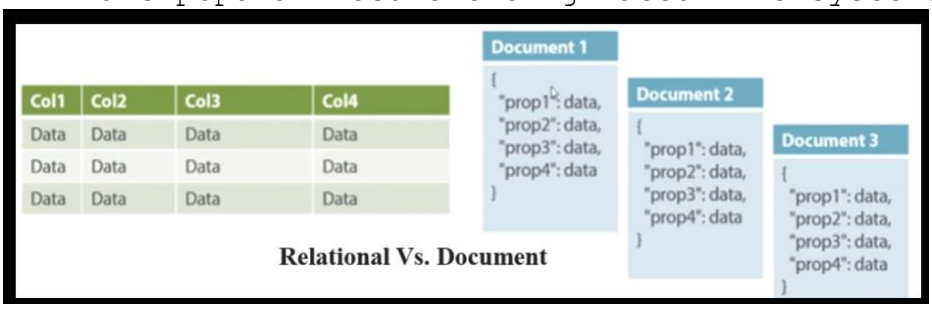
- Column-oriented databases work on columns and are based on BigTable paper by Google. Every column is treated separately.
- Values of single column databases are stored contiguously.

| ColumnFamily | | | |
|---|---|---|---|
| Row Key | Column Name | | |
| | Key | Key | Key |
| | Value | Value | Value |
| | Column Name | | |
| | Key | Key | Key |
| | Value | Value | Value |

- They deliver high performance on aggregation queries like SUM, COUNT, AVG, MIN etc. as the data is readily available in a column.
- Column-based NoSQL databases are widely used to manage data warehouses, business intelligence.
- HBase, Cassandra, HBase, Hypertable are NoSQL query examples of column based database.

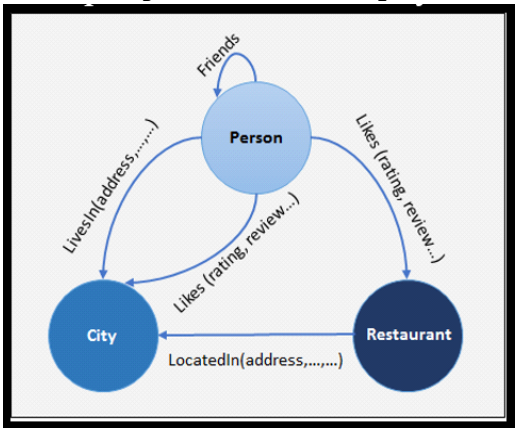
## DOCUMENT-ORIENTED

- Document-Oriented NoSQL DB stores and retrieves data as a key value pair but the value part is stored as a document. The document is stored in JSON or XML formats. The value is understood by the DB and can be queried.
- Amazon SimpleDB, CouchDB, MongoDB, Riak, Lotus Notes, MongoDB, are popular Document originated DBMS systems.




## GRAPH-BASED

- A graph type database stores entities as well the relations amongst those entities. The entity is stored as a node with the relationship as edges.
- An edge gives a relationship between nodes. Every node and edge has a unique identifier.




- Compared to a relational database where tables are loosely connected, a Graph database is a multi-relational in nature. Traversing relationship is fast as they are already captured into the DB, and there is no need to calculate them.
- Graph base database mostly used for social networks, logistics, spatial data.
- Neo4J, Infinite Graph, OrientDB, FlockDB are some popular graph-based databases.

## Four Types of NoSQL Databases

- Types of NoSQL Databases:

1. ***Key-value Pair Based***
2. ***Column-oriented Graph***
3. ***Graphs based***
4. ***Document-oriented***

- Every category has its unique attributes and limitations. None of the above-specified database is better to solve all the problems.
- Users should select the database based on their product needs.

### 1. Key Value Pair Based

- Data is stored in key/value pairs. It is designed in such a way to handle lots of data and heavy load.
- Key-value pair storage databases store data as a hash table where each key is unique, and the value can be a JSON, BLOB(Binary Large Objects), string, etc.
- For example, a key-value pair may contain a key like “Website" associated with a value like “Facebook".

| Key | Value |
|---|---|
| Name | Joe Bloggs |
| Age | 42 |
| Occupation | Stunt Double |
| Height | 175cm |
| Weight | 77kg |

- It is one of the most basic NoSQL database example.
- This kind of NoSQL database is used as a collection, dictionaries, associative arrays, etc. Key value stores help the developer to store schema-less data.
- They work best for shopping cart contents.
- Redis, Dynamo, Riak are some NoSQL examples of key-value store DataBases.

## 2. Column-based

- Column-oriented databases work on columns and are based on BigTable paper by Google. Every column is treated separately.
- Values of single column databases are stored contiguously.

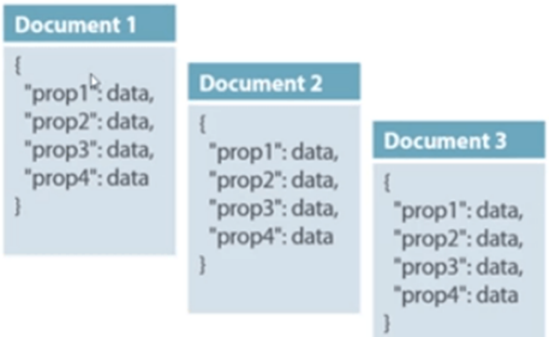
| ColumnFamily | | | |
|---|---|---|---|
| Row Key | Column Name | | |
| | Key | Key | Key |
| | Value | Value | Value |
| | Column Name | | |
| | Key | Key | Key |
| | Value | Value | Value |

- They deliver high performance on aggregation queries like SUM, COUNT, AVG, MIN etc. as the data is readily available in a column.
- Column-based NoSQL databases are widely used to manage data warehouses, business intelligence.
- HBase, Cassandra, HBase, Hypertable are NoSQL query examples of column based database.

## 3. Document-Oriented

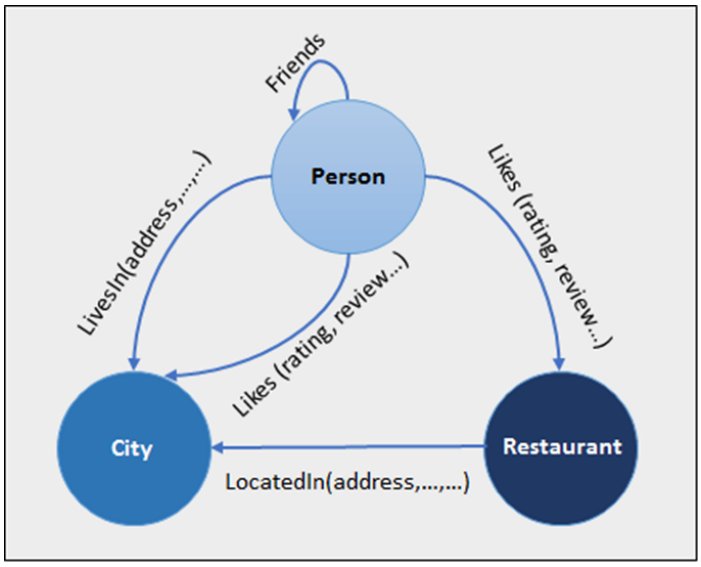
- Document-Oriented NoSQL DB stores and retrieves data as a key value pair but the value part is stored as a document. The document is stored in JSON or XML formats. The value is understood by the DB and can be queried.
- Amazon SimpleDB, CouchDB, MongoDB, Riak, Lotus Notes, MongoDB, are popular Document originated DBMS systems.

| Col1 | Col2 | Col3 | Col4 |
|---|---|---|---|
| Data | Data | Data | Data |
| Data | Data | Data | Data |
| Data | Data | Data | Data |



## 4. Graph-Based

- A graph type database stores entities as well the relations amongst those entities. The entity is stored as a node with the relationship as edges.
- An edge gives a relationship between nodes. Every node and edge has a unique identifier.




- Compared to a relational database where tables are loosely connected, a Graph database is a multi-relational in nature. Traversing relationship is fast as they are already captured into the DB, and there is no need to calculate them.
- Graph base database mostly used for social networks, logistics, spatial data.
- Neo4J, Infinite Graph, OrientDB, FlockDB are some popular graph-based databases.

# AGGREGATE DATA MODELS

- A data model is the model through which we perceive and manipulate our data

- For people using a database, the data model describes how we interact with the data in the database.

- This is distinct from a storage model, which describes how the database stores and manipulates the data internally.

- The dominant data model of the last couple of decades is the relational data model, which is best visualized as a set of tables, rather like a page of a spreadsheet. Each table has rows, with each row representing some entity of interest. We describe this entity through columns, each having a single value.

One of the most obvious shifts with NoSQL is a move away from the relational model. Each NoSQL solution has a different model that it uses, which we put into four categories widely used in the NoSQL ecosystem: key-value, document, column-family, and graph. Of these, the first three share a common characteristic of their data models which we will call aggregate orientation.

## AGGREGATES

- Aggregate is a term that comes from Domain-Driven Design .

- In Domain-Driven Design, an aggregate is a collection of related objects that we wish to treat as a unit.

- In particular, it is a unit for data manipulation and management of consistency.

- Typically, we like to update aggregates with atomic operations and communicate with our data storage in terms of aggregates.

- This definition matches really well with how key-value, document, and column-family databases work. Dealing in aggregates makes it much

easier for these databases to handle operating on a cluster, since the aggregate makes a natural unit for replication and sharding.

- Aggregates are also often easier for application programmers to work with, since they often manipulate data through aggregate structures.

## CONSEQUENCES OF AGGREGATE

1. **No Distributable Storage**

- Relational mapping can captures data elements and their relationship well.

- It does not need any notion of aggregate entity, because it uses foreign key relationship.

- But we cannot distinguish for a relationship that represent aggregations from those that don't.

- As result we cannot take advantage of that knowledge to store and distribute our data.

2. **Marking Aggregate Tools**

- Various data modeling techniques have provided ways of marking aggregate or composite structures.

- The problem, however, is that modelers rarely provide any semantics for what makes an aggregate relationship different from any other; where there are semantics, they vary.

- When working with aggregate-oriented databases, we have a clearer semantics to consider by focusing on the unit of interaction with the data storage.

3. **Aggregate Ignorant**

- Relational database are aggregate ignorant, since they don't have concepts of aggregates.
- Also graph database are aggregate-ignorant.
- This is not always bad.In domains where it is difficult to draw aggregate boundaries aggregate-ignorant databases are useful.

## AGGREGATES AND TRANSACTIONS

ACID Transactions

Relational database allow us to manipulate any combination of rows from any table in a single transaction.

- ACID transactions:
  – Atomic,
  – Consistent,
  – Isolated, and
  – Durable have the main point in Atomicity

## ATOMICITY & RDBMS

- Many rows spanning many tables are updated into an Atomic operation
- It may succeeded or failed entirely
- Concurrently operations are isolated and we cannot see partial updates
- However relational database still fail.

## ATOMICITY & NoSQL

- NoSQL don’t support Atomicity that spans multiple aggregates.

- This means that if we need to update multiple aggregates we have to manage that in the application code.

- Thus the Atomicity is one of the consideration for deciding how to divide up our data into aggregates