LECTURE NOTES

ON

DATABASE MANAGEMENT SYSTEMS

BBA 4TH SEMESTER

UNIT -1

UNIT -1

The Worlds of Database Systems

INTRODUCTION TO BASIC CONCEPTS OF DATABASE SYSTEMS:

What is Data?

The raw facts are called as data. The word “raw” indicates that they have not been processed.

Ex: For example 89 is the data.

What is information?

The processed data is known as information.

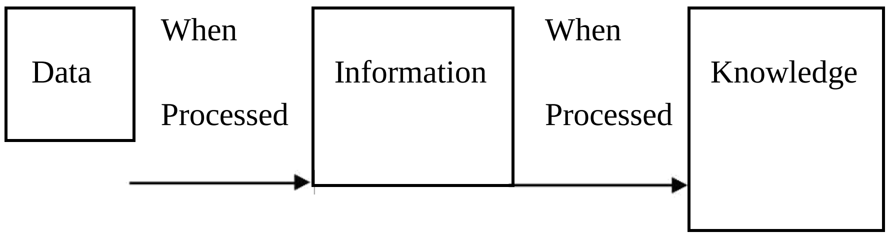
Ex: Marks: 89; then it becomes information.

What is Knowledge?

1. Knowledge refers to the practical use of information.
2. Knowledge necessarily involves a personal experience.

DATA/INFORMATION PROCESSING:

The process of converting the data (raw facts) into meaningful information is called as data/information processing.



Note: In business processing knowledge is more useful to make decisions for any organization.

DIFFERENCE BETWEEN DATA AND INFORMATION:

.

DATA	INFORMATION
1.Raw facts	1.Processed data
2. It is in unorganized form	2. It is in organized form
3. Data doesn't help in decision making process	3. Informationhelpsin decision making process

FILE ORIENTED APPROACH:

The earliest business computer systems were used to process business records and produce information. They were generally faster and more accurate than equivalent manual systems. These systems stored groups of records in separate files, and so they were called **file processing systems.**

1. File system is a collection of data. Any management with the file system, user has to write the procedures

2. File system gives the details of the data representation and Storage of data.

3. In File system storing and retrieving of data cannot be done efficiently.

4. Concurrent access to the data in the file system has many problems like a Reading the file while other deleting some information, updating some information

5. File system doesn't provide crash recovery mechanism.
Eg. While we are entering some data into the file if System crashes then content of the file is lost.

6. Protecting a file under file system is very difficult.

The typical file-oriented system is supported by a conventional operating system. Permanent records are stored in various files and a number of different application programs are written to extract records from and add records to the appropriate files.

DISADVANTAGES OF FILE-ORIENTED SYSTEM:

The following are the disadvantages of File-Oriented System:

Data Redundancy and Inconsistency:

Since files and application programs are created by different programmers over a long period of time, the files are likely to be having different formats and the programs may be written in several programming languages. Moreover, the same piece of information may be duplicated in several places. This redundancy leads to higher storage and access cost. In addition, it may lead to data inconsistency.

Difficulty in Accessing Data:

The conventional file processing environments do not allow needed data to be retrieved in a convenient and efficient manner. Better data retrieval system must be developed for general use.

Data Isolation:

Since data is scattered in various files, and files may be in different formats, it is difficult to write new application programs to retrieve the appropriate data.

Concurrent Access Anomalies:

In order to improve the overall performance of the system and obtain a faster response time, many systems allow multiple users to update the data simultaneously. In such an environment, interaction of concurrent updates may result in inconsistent data.

Security Problems:

Not every user of the database system should be able to access all the data. For example, in banking system, payroll personnel need only that part of the database that has information about various bank employees. They do not need access to information about customer accounts. It is difficult to enforce such security constraints.

Integrity Problems:

The data values stored in the database must satisfy certain types of consistency constraints. For example, the balance of a bank account may never fall below a prescribed amount. These constraints are enforced in the system by adding appropriate code in the various application programs. When new constraints are added, it is difficult to change the programs to enforce them. The problem is compounded when constraints involve several data items for different files.

Atomicity Problem:

A computer system like any other mechanical or electrical device is subject to failure. In many applications, it is crucial to ensure that once a failure has occurred and has been detected, the data are restored to the consistent state existed prior to the failure

Example:

Consider part of a savings-bank enterprise that keeps information about all customers and savings accounts. One way to keep the information on a computer is to store it in operating system files. To allow users to manipulate the information, the system has a number of application programs that manipulate the files, including:

A program to debit or credit an account

A program to add a new account

A program to find the balance of an account

A program to generate monthly statements

Programmers wrote these application programs to meet the needs of the bank. New application programs are added to the system as the need arises. For example, suppose that the savings bank decides to offer checking accounts.

As a result, the bank creates new permanent files that contain information about all the checking accounts maintained in the bank, and it may have to write new application programs to deal with situations that do not arise in savings accounts, such as overdrafts. Thus, as time goes by, the system acquires more files and more application programs. The system stores permanent records in various files, and it needs different

Application programs to extract records from, and add records to, the appropriate files. Before database management systems (DBMS) came along, organizations usually stored information in such systems. Organizational information in a file-processing system has a number of major disadvantages:

1. Data Redundancy and Inconsistency:

The address and telephone number of a particular customer may appear in a file that consists of savings-account records and in a file that consists of checking-account records. This redundancy leads to higher storage and access cost. In, it may lead to data inconsistency; that is, the various copies of the same data may no longer agree. For example, a changed customer address may be reflected in savings-account records but not elsewhere in the system.

2. **Difficulty in Accessing Data:**

Suppose that one of the bank officers needs to find out the names of all customers who live within a particular postal-code area. The officer asks the data-processing department to generate such a list. Because there is no application program to generate that. The bank officer has now two choices: either obtain the list of all customers and extract the needed information manually or ask a system programmer to write the necessary application program. Both alternatives are obviously unsatisfactory.

3. **Data Isolation:**

Because data are scattered in various files and files may be in different formats, writing new application programs to retrieve the appropriate data is difficult.

4. **Integrity Problems:**

The balance of a bank account may never fall below a prescribed amount (say, $25). Developers enforce these constraints in the system by adding appropriate code in the various application programs. However, when new constraints are added, it is difficult to change the programs to enforce them. The problem is compounded when constraints involve several data items from different files.

5. **Atomicity Problems:**

A computer system, like any other mechanical or electrical device, is subject to failure. In many applications, it is crucial that, if a failure occurs, the data be restored to the consistent state that existed prior to the failure. Consider a program to transfer $50 from account *A* to account *B*. If a system failure occurs during the execution of the program, it is possible that the $50 was removed from account *A* but was not credited to account *B*, resulting in an inconsistent database state. Clearly, it is essential to database consistency that either both the credit and debit occur, or that neither occur. That is, the funds transfer must be *atomic*—it must happen in its entirety or not at all. It is difficult to ensure atomicity in a conventional file-processing system.

6. **Concurrent-Access Anomalies:**

For the sake of overall performance of the system and faster response, many systems allow multiple users to update the data simultaneously. In such an environment, interaction of concurrent updates may result in inconsistent data. Consider bank account *A*, containing $500. If two customers withdraw funds (say $50 and $100 respectively) from account *A* at about the same time, the result of the concurrent executions may leave the account in an incorrect (or inconsistent) state. Suppose that the programs executing on behalf of each withdrawal read the old balance, reduce that value by the amount being withdrawn, and write the result back. If the two programs run concurrently, they may both read the value $500, and write back $450 and $400, respectively. Depending on which one writes the value last, the account may contain $450 or $400, rather than the correct value of $350. To guard against this possibility, the system must maintain some form of supervision. But supervision is difficult to provide because data may be accessed by many different application programs that have not been coordinated previously.

7. **Security Problems:**

Not every user of the database system should be able to access all the data. For example, in a banking system, payroll personnel need to see only that part of the database that has information about the various bank employees. They do not need access to information about customer accounts. But, since application programs are added to the system in an ad hoc manner, enforcing such security constraints is difficult. These difficulties, among others, prompted the development of database systems.

INTRODUCTION TO DATABASES:

History of Database Systems:

1950s and early 1960s:

Magnetic tapes were developed for data storage

Data processing tasks such as payroll were automated, with data stored on tapes.

Data could also be input from punched card decks, and output to printers.

Late 1960s and 1970s: The use of hard disks in the late 1960s changed the scenario for data processing greatly, since hard disks allowed direct access to data.

With disks, network and hierarchical databases could be created that allowed data structures such as lists and trees to be stored on disk. Programmers could construct and manipulate these data structures.

With disks, network and hierarchical databases could be created that allowed data structures such as lists and trees to be stored on disk. Programmers could construct and manipulate these data structures.

In the 1970's the EF CODD defined the **Relational Model.**

In the 1980's:

Initial commercial relational database systems, such as IBM DB2, Oracle, Ingress, and DEC Rdb, played a major role in advancing techniques for efficient processing of declarative queries.

In the early 1980s, relational databases had become competitive with network and hierarchical database systems even in the area of performance.

The 1980s also saw much research on parallel and distributed databases, as well as initial work on object-oriented databases.

Early 1990s:

The SQL language was designed primarily in the 1990's.

And this is used for the transaction processing applications.

Decision support and querying re-emerged as a major application area for databases.

Database vendors also began to add object-relational support to their databases.

Late 1990s:

The major event was the explosive growth of the World Wide Web.

Databases were deployed much more extensively than ever before. Database systems now had to support very high transaction processing rates, as well as very high reliability and 24 * 7 availability (availability 24 hours a day, 7 days a week, meaning no downtime for scheduled maintenance activities).

Database systems also had to support Web interfaces to data.

The Evolution of Database systems:

The Evolution of Database systems are as follows:

1. File Management System
2. Hierarchical database System
3. Network Database System
4. Relational Database System

File Management System:

The file management system also called as FMS in short is one in which all data is stored on a single large file. The main disadvantage in this system is searching a record or data takes a long time. This lead to the introduction of the concept, of indexing in this system. Then also the FMS system had lot of drawbacks to name a few like updating or modifications to the data cannot be handled easily, sorting the records took long time and so on. All these drawbacks led to the introduction of the Hierarchical Database System.

Hierarchical Database System:

The previous system FMS drawback of accessing records and sorting records which took a long time was removed in this by the introduction of parent-child relationship between records in database. The origin of the data is called the root from which several branches have data at different levels and the last level is called the leaf. The main drawback in this was if there is any modification or addition made to the structure then the whole structure needed alteration which made the task a tedious one. In order to avoid this next system took its origin which is called as the Network Database System.

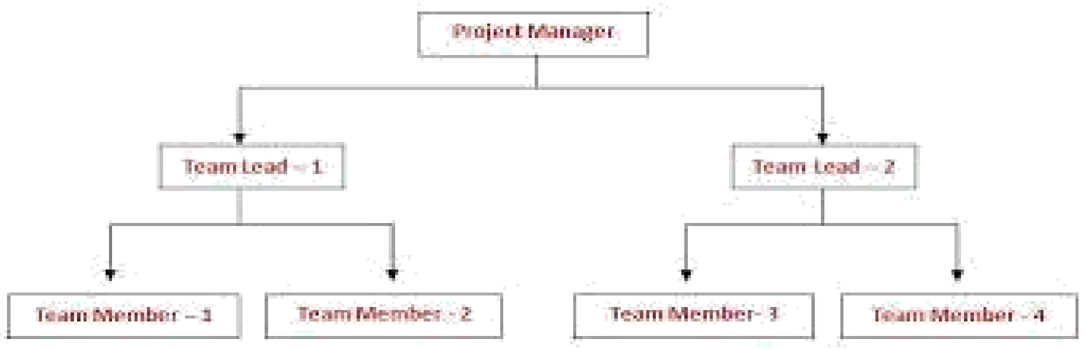
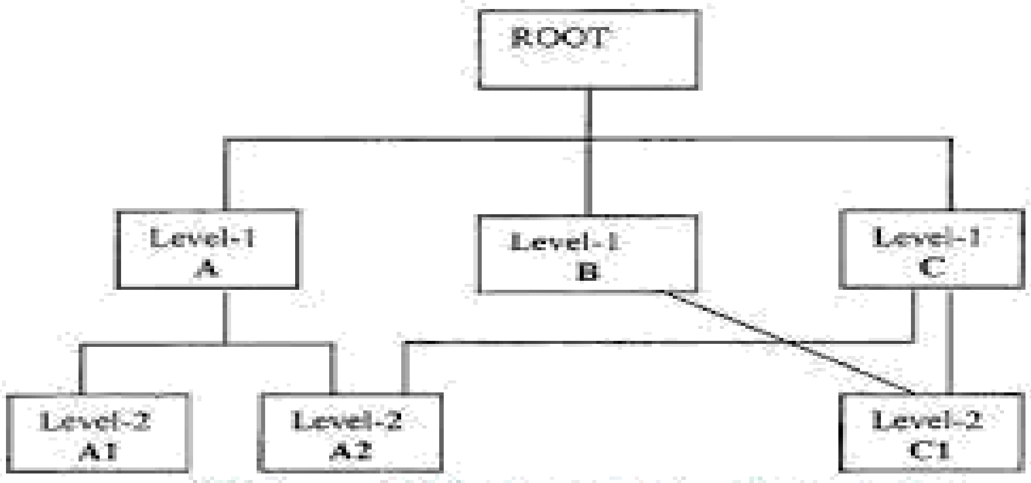


Fig: Hierarchical Database System

Network Database System:

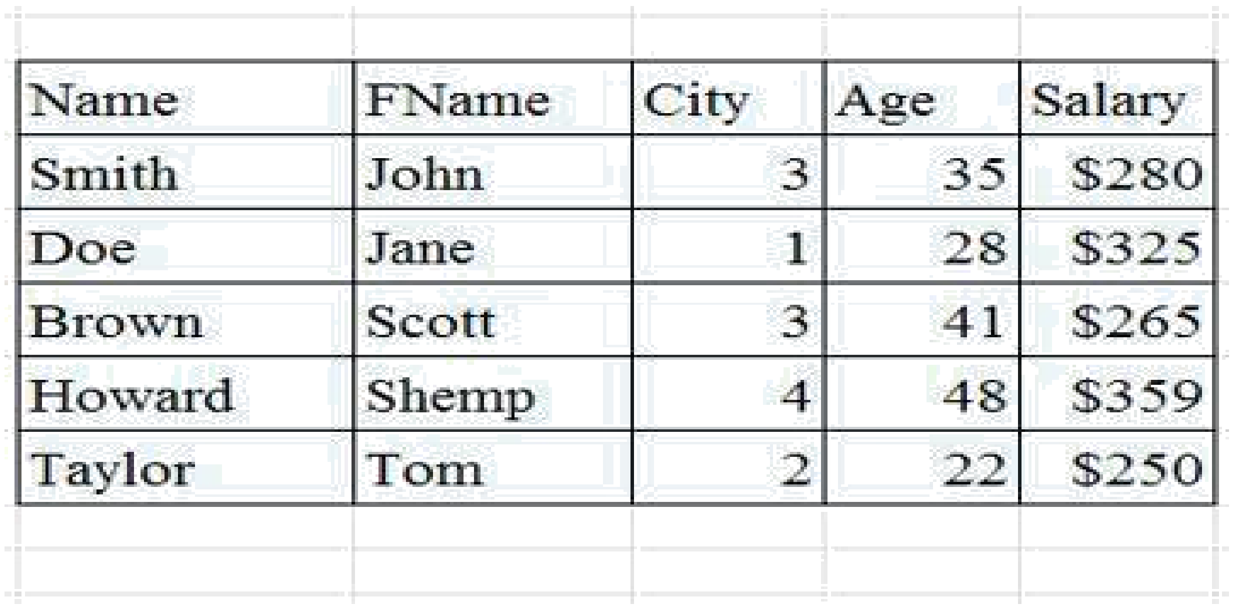
In this the main concept of many-many relationships got introduced. But this also followed the same technology of pointers to define relationships with a difference in this made in the introduction if grouping of data items as sets.



Network Database Model Diagram

Relational Database System:

In order to overcome all the drawbacks of the previous systems, the Relational Database System got introduced in which data get organized as tables and each record forms a row with many fields or attributes in it. Relationships between tables are also formed in this system.



Name	FName	City	Age	Salary
Smith	John	3	35	$280
Doe	Jane	1	28	$325
Brown	Scott	3	41	$265
Howard	Shemp	4	48	$359
Taylor	Tom	2	22	$250

DATABASE:

A database is a collection of related data.

(OR)

A database is a collection of information that is organized so that it can be easily accessed, managed and updated.

Examples / Applications of Database Systems:

The following are the various kinds of applications/organizations uses databases for their business processing activities in their day-to-day life. They are:

1.**Banking:** For customer information, accounts, and loans, and banking transactions.

2. **Airlines:** For reservations and schedule information. Airlines were among the first to use

databases in a geographically distributed manner—terminals situated around the world accessed the central database system through phone lines and other data networks.

3. **Universities:** For student information, course registrations, and grades.

4. **Credit Card Transactions:** For purchases on credit cards and generation of monthly statements.

5. **Telecommunication:** For keeping records of calls made, generating monthly bills, maintaining balances on prepaid calling cards, and storing information about the communication networks.

6. **Finance:** For storing information about holdings, sales, and purchases of financial instruments such as stocks and bonds.

7. **Sales:** For customer, product, and purchase information.

8. **Manufacturing:** For management of supply chain and for tracking production of items in factories, inventories of items in warehouses/stores, and orders for items.

9. **Human resources:** For information about employees, salaries, payroll taxes and benefits, and for generation of paychecks.

10. **Railway Reservation Systems:** For reservations and schedule information.

11. **Web:** For access the Back accounts and to get the balance amount.

12. **E –Commerce:** For Buying a book or music CD and browse for things like watches, mobiles from the Internet.

CHARACTERISTICS OF DATABASE:

The database approach has some very characteristic features which are discussed in detail below:

Structured and Described Data:

Fundamental feature of the database approach is that the database system does not only contain the data but also the complete definition and description of these data. These descriptions are basically details about the extent, the structure, the type and the format of all data and, additionally, the relationship between the data. This kind of stored data is called metadata ("data about data").

Separation of Data and Applications:

Application software does not need any knowledge about the physical data storage like encoding, format, storage place, etc. It only communicates with the management system of a database (DBMS) via a standardized interface with the help of a standardized language like SQL. The access to the data and the metadata is entirely done by the DBMS. In this way all the applications can be totally separated from the data.

Data Integrity:

Data integrity is a byword for the quality and the reliability of the data of a database system. In a broader sense data integrity includes also the protection of the database from unauthorized access (confidentiality) and unauthorized changes. Data reflect facts of the real world.

Transactions:

A transaction is a bundle of actions which are done within a database to bring it from one consistent state to a new consistent state. In between the data are inevitable inconsistent. *A* transaction is atomic what

means that it cannot be divided up any further. Within a transaction all or none of the actions need to be carried out. Doing only a part of the actions would lead to an inconsistent database state.

Example: One example of a transaction is the transfer of an amount of money from one bank account to another.

Data Persistence:

Data persistence means that in a DBMS all data is maintained as long as it is not deleted explicitly. The life span of data needs to be determined directly or indirectly be the user and must not be dependent on system features. Additionally data once stored in a database must not be lost. Changes of a database which are done by a transaction are persistent. When a transaction is finished even a system crash cannot put the data in danger

TYPES OF DATABASES:

Database can be classified according to the following factors. They are:

1.Number of Users

2.Database Location

3.Expected type

4.Extent of use

1. Based on number of Users:

According to the number of users the databases can be classified into following types. They are :

a). Single user b). Multiuser

Single user database:

Single user database supports only one user at a time.

Desktop or personal computer database is an example for single user database.

Multiuser database:

Multi user database supports multiple users at the same time.

Workgroup database and enterprise databases are examples for multiuser database.

Workgroup database:

If the multiuser database supports relatively small number of users (fewer than 50) within an organization is called as Workgroup database.

Enterprise database:

If the database is used by the entire organization and supports multiple users (more than 50) across many departments is called as Enterprise database.

2. Based on Location:

According to the location of database the databases can be classified into following types. They are:
a).CentralizedDatabase
b).Distributed Database

Centralized Database:

It is a database that is located, stored, and maintained in a single location. This location is most often a central computer or database system, for example a desktop or server CPU, or a mainframe computer. In most cases, a centralized database would be used by an organization (e.g. a business company) or an institution (e.g. a university.)

Distributed Database:

A distributed database is a database in which storage devices are not all attached to a common CPU. It may be stored in multiple computers located in the same physical location, or may be dispersed over a network of interconnected computers.

INTRODUCTION TO DATABASE-MANAGEMENT SYSTEM:

Database Management System:

- A **database-management system** (DBMS) is a collection of interrelated data and a set of programs to access those data.
- The DBMS is a general purpose software system that facilitates the process of defining constructing and manipulating databases for various applications.

Goals of DBMS:

The primary goal of a DBMS is to provide a way to store and retrieve database information that is both *convenient* and *efficient*

1.Manage large bodies of information

2. Provide convenient and efficient ways to store and access information

3. Secure information against system failure or tampering

4. Permit data to be shared among multiple users

Properties of DBMS:

1. A Database represents some aspect of the real world. Changes to the real world reflected in the database.

2. A Database is a logically coherent collection of data with some inherent meaning.

3. A Database is designed and populated with data for a specific purpose.

Need of DBMS:

1. Before the advent of DBMS, organizations typically stored information using a "File Processing Systems".

Example of such systems is File Handling in High Level Languages like C, Basic and COBOL etc., these systems have Major disadvantages to perform the Data Manipulation. So to overcome those drawbacks now we are using the DBMS.

2. Database systems are designed to manage large bodies of information.

3. In addition to that the database system must ensure the safety of the information stored, despite system crashes or attempts at unauthorized access. If data are to be shared among several users, the system must avoid possible anomalous results.

ADVANTAGES OF A DBMS OVER FILE SYSTEM:

Using a DBMS to manage data has many advantages:

Data Independence:

Application programs should be as independent as possible from details of data representation and storage. The DBMS can provide an abstract view of the data to insulate application code from such details.

Efficient Data Access:

A DBMS utilizes a variety of sophisticated techniques to store and retrieve data efficiently. This feature is especially important if the data is stored on external storage devices.

Data Integrity and Security:

If data is always accessed through the DBMS, the DBMS can enforce integrity constraints on the data. For example, before inserting salary information for an employee, the DBMS can check that the department budget is not exceeded. Also, the DBMS can enforce *access controls* that govern what data is visible to different classes of users.

Concurrent Access and Crash Recovery:

A database system allows several users to access the database concurrently. Answering different questions from different users with the same (base) data is a central aspect of an information system. Such concurrent use of data increases the economy of a system.

An example for concurrent use is the travel database of a bigger travel agency. The employees of different branches can access the database concurrently and book journeys for their clients. Each travel agent sees on his interface if there are still seats available for a specific journey or if it is already fully booked.

A DBMS also protects data from failures such as power failures and crashes etc. by the recovery schemes such as backup mechanisms and log files etc.

Data Administration:

When several users share the data, centralizing the administration of data can offer significant improvements. Experienced professionals, who understand the nature of the data being managed, and how different groups of users use it, can be responsible for organizing the data representation to minimize redundancy and fine-tuning the storage of the data to make retrieval efficient.

Reduced Application Development Time:

DBMS supports many important functions that are common to many applications accessing data stored in the DBMS. This, in conjunction with the high-level interface to the data, facilitates quick development of applications. Such applications are also likely to be more robust than applications developed from scratch because many important tasks are handled by the DBMS instead of being implemented by the application.

DISADVANTAGES OF DBMS:

Danger of a Overkill:

For small and simple applications for single users a database system is often not advisable.

Complexity:

A database system creates additional complexity and requirements. The supply and operation of a database management system with several users and databases is quite costly and demanding.

Qualified Personnel:

`The professional operation of a database system requires appropriately trained staff. Without a qualified database administrator nothing will work for long.

Costs:

Through the use of a database system new costs are generated for the system itself but also for additional hardware and the more complex handling of the system.

Lower Efficiency:

A database system is a multi-use software which is often less efficient than specialized software which is produced and optimized exactly for one problem.

DATABASE USERS & DATABASE ADMINISTRATORS:

People who work with a database can be categorized as database users or database administrators.

Database Users:

There are four different types of database-system users, differentiated by the way they expect to interact with the system.

Naive users:

Naive users are unsophisticated users who interact with the system by invoking one of the application programs that have been written previously.

For example, a bank teller who needs to transfer $50 from account *A* to account *B* invokes a program called *transfer*. This program asks the teller for the amount of money to be transferred, the account from which the money is to be transferred, and the account to which the money is to be transferred.

Application programmers:

Application programmers are computer professionals who write application programs. 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 without writing a program.

Sophisticated users:

Sophisticated users interact with the system without writing programs. Instead, they form their requests in a database query language. They submit each such query to a **query processor**, whose function is to break down DML statements into instructions that the storage manager understands. Analysts who submit queries to explore data in the database fall in this category.

Specialized users:

Specialized users are sophisticated users who write specialized database applications that do not fit into the traditional data-processing framework.

Database Administrator:

One of the main reasons for using DBMSs 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**).

Database Administrator Functions/Roles:

The functions of a DBA include:

Schema definition:

The DBA creates the original database schema by executing a set of data definition statements in the DDL, Storage structure and access-method definition.

Schema and physical-organization modification:

The DBA carries out changes to the schema and physical organization to reflect the changing needs of the organization, or to alter the physical organization to improve performance.

Granting of authorization for data access:

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:

Examples of the database administrator's routine maintenance activities are:

1. Periodically backing up the database, either onto tapes or onto remote servers, to prevent loss of data in case of disasters such as flooding.

2. Ensuring that enough free disk space is available for normal operations, and upgrading disk space as required.

3. Monitoring jobs running on the database and ensuring that performance is not degraded by very expensive tasks submitted by some users.

LEVELS OF ABSTRACTION IN A DBMS:

Hiding certain details of how the data are stored and maintained. A major purpose of database system is to provide users with an "Abstract View" of the data. In DBMS there are 3 levels of data abstraction. The goal of the abstraction in the DBMS is to separate the users request and the physical storage of data in the database.

Levels of Abstraction:

Physical Level:

- The lowest Level of Abstraction describes "How" the data are actually stored.
  The physical level describes complex low level data structures in detail.

Logical Level:

- This level of data Abstraction describes "What" data are to be stored in the database and what relationships exist among those data.

- Database Administrators use the logical level of abstraction.

View Level:

- It is the highest level of data Abstracts that describes only part of entire database.
- Different users require different types of data elements from each database.

  The system may provide many views for the some database.

THREE SCHEMA ARCHITECTURE:

Schema:

The overall design of the database is called the "Schema" or "Meta Data". A database schema corresponds to the programming language type definition. The value of a variable in programming language corresponds to an "Instance" of a database Schema.

Three Schema Architecture:

The goal of this architecture is to separate the user applications and the physical database. In this architecture, schemas can be defined at the following three levels:

1. The **internal level** has an **internal schema,** which describes the physical storage structure of the database. The internal schema uses a physical data model and describes the complete details of data storage and access paths for the database.

2. The **conceptual level** has a **conceptual schema,** which describes the structure of the whole database for a community of users. The conceptual schema hides the details of physical storage structures and concentrates on describing entities, data types, relationships, user operations, and constraints. A high-level data model or an implementation data model can be used at this level.

3. The **external** or **view level** includes a number of **external schemas** or **user views.** Each external schema describes the part of the database that a particular user group is interested in and hides the rest of the database from that user group. A high-level data model or an implementation data model can be used at this level.

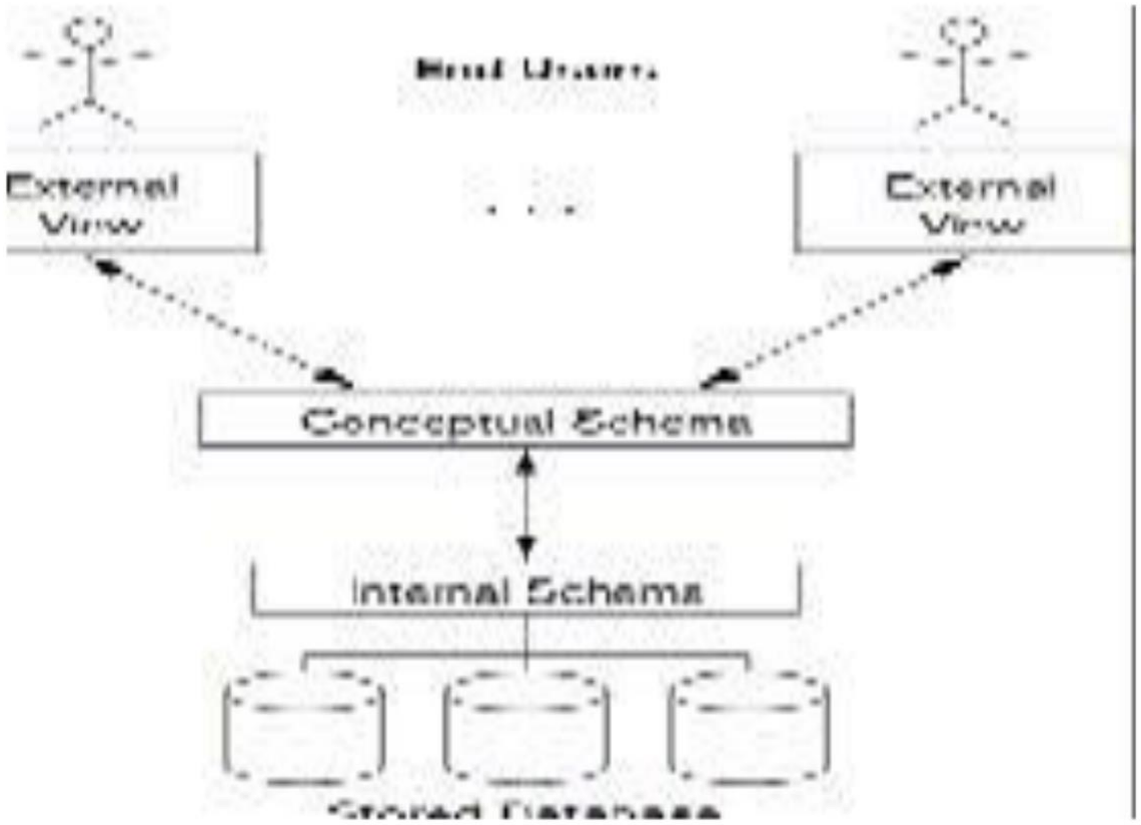


Fig: Three-Schema Architecture

DATA INDEPENDENCE:

- A very important advantage of using DBMS is that it offers Data Independence.
- The ability to modify a scheme definition in one level without affecting a scheme definition in a higher level is called **data independence**.
- There are two kinds:

1. Physical Data Independence
2. Logical Data Independence

Physical Data Independence:

- The ability to modify the physical schema without causing application programs to be rewritten

- Modifications at this level are usually to improve performance.

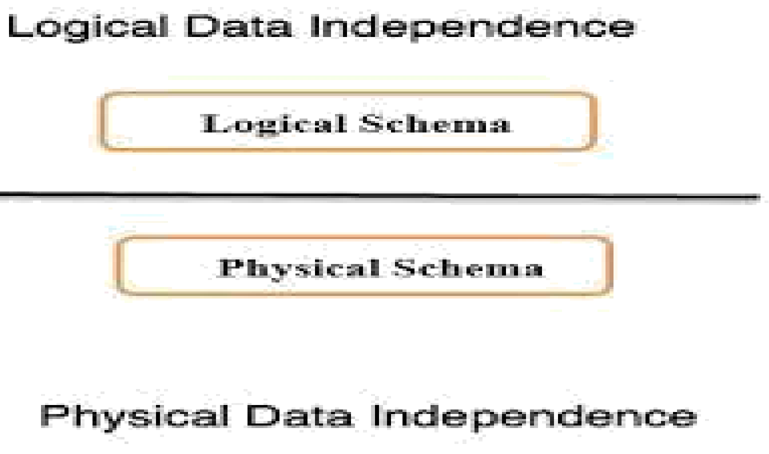


Fig: Data Independence

Logical Data Independence:

- The ability to modify the conceptual schema without causing application programs to be rewritten
- Usually done when logical structure of database is altered

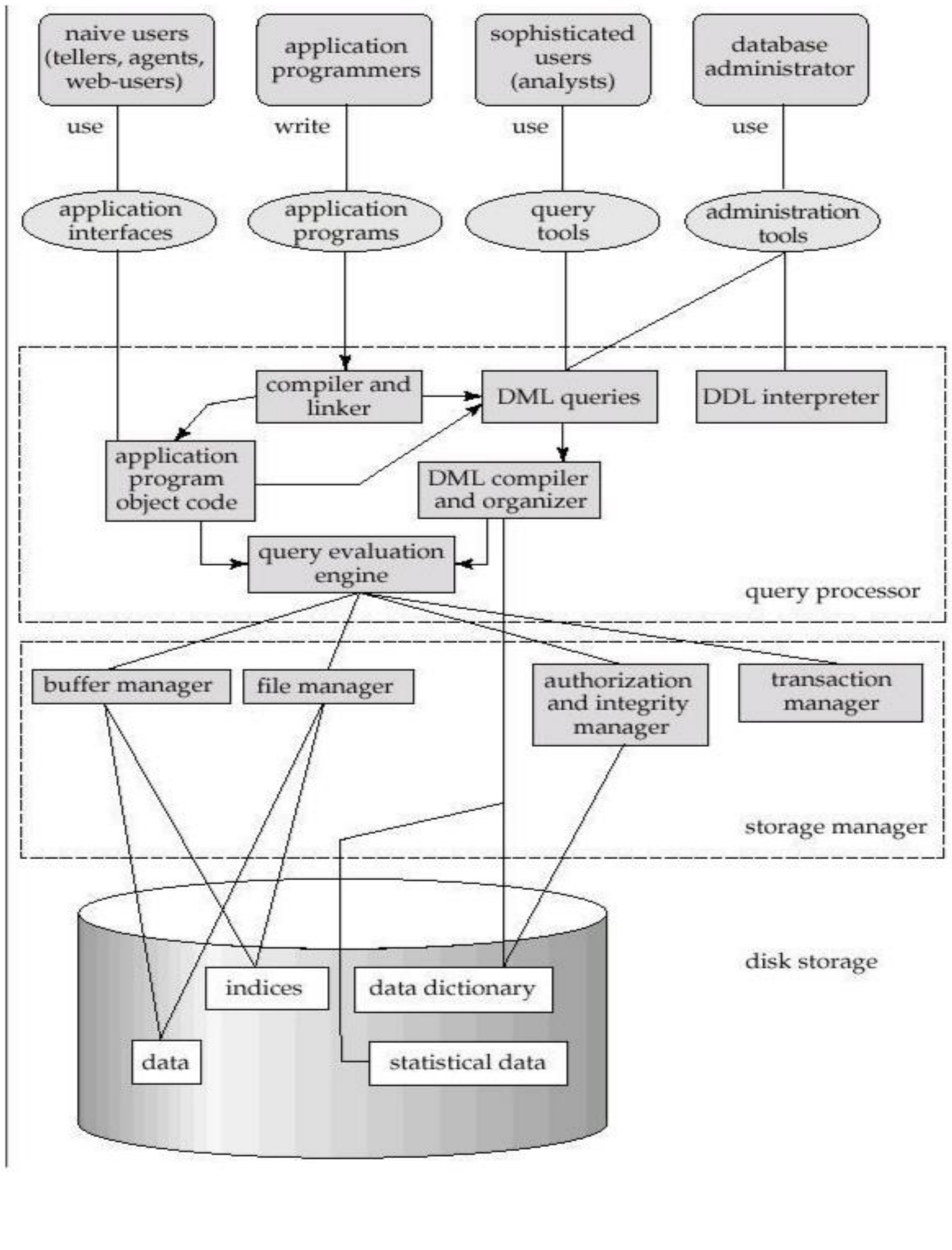
- Logical data independence is harder to achieve as the application programs are usually heavily dependent on the logical structure of the data.

DATABASE SYSTEM STRUCTURE:

A database system is partitioned into modules that deal with each of the responsibilities of the overall system. The functional components of a database system can be broadly divided into the storage manager and the query processor components.

The storage manager is important because databases typically require a large amount of storage space. Some Big organizations Database ranges from Giga bytes to Tera bytes. So the main memory of computers cannot store this much information, the information is stored on disks. Data are moved between disk storage and main memory as needed.

The query processor also very important because it helps the database system simplify and facilitate access to data. So quick processing of updates and queries is important. It is the job of the database system to translate updates and queries written in a nonprocedural language,


naive users (tellers, agents, web-users)
application programmers
sophisticated users (analysts)
database administrator
use
write
use
use
application interfaces
application programs
query tools
administration tools
compiler and linker
DML queries
DDL interpreter
application program object code
DML compiler and organizer
query evaluation engine
query processor
buffer manager
file manager
authorization and integrity manager
transaction manager
storage manager
disk storage
indices
data dictionary
data
statistical data

StorageManager:

A storage manager is a program module that provides the interface between the low level data stored in the database and the application programs and queries submitted to the system. The storage manager is responsible for the interaction with the file manager. The storage manager translates the various DML statements into low-level file-system commands. Thus, the storage manager is responsible for storing, retrieving, and updating data in the database.

Storage Manager Components:

Authorization and integrity manager which tests for the satisfaction of integrity constraints and checks the authority of users to access data.

Transaction manager which ensures that the database itself remains in a consistent state despite system failures, and that concurrent transaction executions proceed without conflicting.

File manager: which manages the allocation of space on disk storage and the data structures used to represent information stored on disk.

Buffer manager which is responsible for fetching data from disk storage into main memory. Storage manager implements several data structures as part of the physical system implementation. Data files are used to store the database itself. Data dictionary is used to stores metadata about the structure of the database, in particular the schema of the database.

Query Processor Components:

DDL interpreter: It interprets DDL statements and records the definitions in the data dictionary.

DML compiler: It translates DML statements in a query language into an evaluation plan consisting of low-level instructions that the query evaluation engine understands.

Query evaluation engine: It executes low-level instructions generated by the DML compiler.

Application Architectures:

Most users of a database system today are not present at the site of the database system, but connect to it through a network. We can therefore differentiate between client machines, on which remote database users' work, and server machines, on which the database system runs. Database applications are usually partitioned into two or three parts. They are:

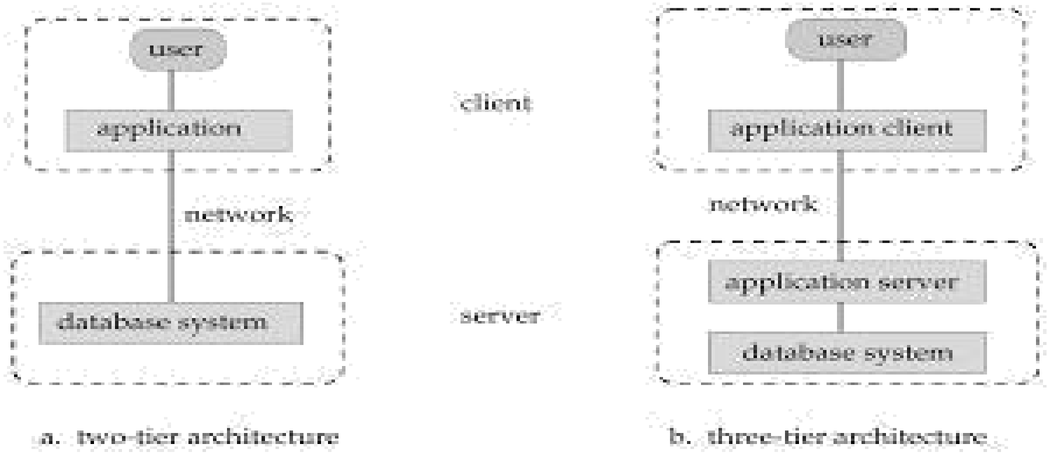
1. Two – Tier Architecture
2. Three – Tier Architecture.

Two-Tier Architecture:

The application is partitioned into a component that resides at the client machine, which invokes database system functionality at the server machine through query language statements. Application program interface standards like ODBC and JDBC are used for interaction between the client and the server.

Three-Tier Architecture:

The client machine acts as merely a front end and does not contain any direct database calls. Instead, the client end communicates with an application server, usually through forms interface. The application server in turn communicates with a database system to access data. The business logic of the application, which says what actions to carry out under what conditions, is embedded in the application server, instead of being distributed across multiple clients. Three-tier applications are more appropriate for large applications, and for applications that run on the World Wide Web.



DATABASE DESIGN:

The database design process can be divided into six steps. The ER Model is most relevant to the first three steps. Next three steps are beyond the ER Model.

1. Requirements Analysis:

The very first step in designing a database application is to understand what data is to be stored in the database, what applications must be built on top of it, and what operations are most frequent and subject to performance requirements. The database designers collect information of the organization and analyzer, the information to identify the user's requirements. The database designers must find out what the users want from the database.

2. Conceptual Database Design:

Once the information is gathered in the requirements analysis step a conceptual database design is developed and is used to develop a high level description of the data to be stored in the database, along with the constraints that are known to hold over this data. This step is often carried out using the ER model, or a similar high-level data model.

3. Logical Database Design:

In this step convert the conceptual database design into a database schema (Logical Database Design) in the data model of the chosen DBMS. We will only consider **relational DBMSs**, and therefore, the task in the

logical design step is to convert an ER schema into a relational database schema. The result is a conceptual schema, sometimes called the **logical schema**, in the relational data model.

Beyond the ER Design:

The first three steps are more relevant to the ER Model. Once the logical scheme is defined designer consider the physical level implementation and finally provide certain security measures. The remaining three steps of database design are briefly described below:

4. Schema Refinement:

The fourth step in database design is to analyze the collection of relations in our relational database schema to identify potential problems, and to refine it. In contrast to the requirements analysis and conceptual design steps, which are essentially subjective, schema refinement can be guided by some elegant and powerful theory.

5. **Physical Database Design:**

In this step we must consider typical expected workloads that our database must support and further refine the database design to ensure that it meets desired performance criteria. This step may simply involve building indexes on some tables and clustering some tables, or it may involve a substantial redesign of parts of the database schema obtained from the earlier design steps.

6. Security Design:

The last step of database design is to include security features. This is required to avoid unauthorized access to database practice after all the six steps. We required Tuning step in which all the steps are interleaved and repeated until the design is satisfactory.

DBMS FUNCTIONS:

- DBMS performs several important functions that guarantee the integrity and consistency of the data in the database.
- Those functions transparent to end users and can be accessed only through the use of DBMS. They include:
  - Data Dictionary Management
  - Data Storage Management
  - Data transformation and Presentation
  - Security Management
  - Multiple Access Control
  - Backup and Recovery Management
  - Data Integrity Management
  - Database Access Languages

  Databases Communication Interfaces

Data Dictionary Management:

- DBMS stores definitions of database elements and their relationship (Metadata) in the data dictionary.
- The DBMS uses the data dictionary to look up the required data component structures and relationships.
- Any change made in database structure is automatically recorded in the data dictionary.

Data Storage Management:

- Modern DBMS provides storage not only for data but also for related data entities.
- Data Storage Management is also important for database “performance tuning”.
- Performance tuning related to activities that make database more efficiently.

Data Transformation and Presentation:

- DBMS transforms entered data to confirm to required data structures.
- DBMS formats the physically retrieved data to make it confirms to user’s logical expectations.
- DBMS also presents the data in the user’s expected format.

Security Management:

- DBMS creates a security system that enforces the user security and data privacy.
- Security rules determines which users can access the database, which data items each user can access etc.
- DBA and authenticated user logged to DBMS through username and password or through Biometric authentication such as Finger print and face reorganization etc.

Multiuser Access Control:

- To provide data integrity and data consistency, DBMS uses sophisticated algorithms to ensure that multiple users can access the database concurrently without compromising the integrity of database.

Backup and Recovery Management:

- DBMS provides backup and recovery to ensure data safety and integrity.
- Recovery management deals with the recovery of database after failure such as bad sector in the disk or power failure. Such capability is critical to preserve database integrity.

Data Integrity Management:

- DBMS provides and enforces integrity rules, thus minimizing data redundancy and maximizing data consistency.
- Ensuring data integrity is especially important in transaction- oriented database systems.

Database Access Languages:

- DBMS provides data access through a query language.

- A query language is a non-procedural language i.e. it lets the user specify what must be done without specifying how it is to be done.

- SQL is the default query language for data access.

<u>Databases Communication Interfaces:</u>

- Current DBMS's are accepting end-user requests via different network environments.

- For example, DBMS might provide access to database via Internet through the use of web browsers such as Mozilla Firefox or Microsoft Internet Explorer.

What is Schema?

A database schema is the skeleton structure that represents the logical view of the entire database. (or)

The logical structure of the database is called as Database Schema. (or)

The overall design of the database is the database schema.

- It defines how the data is organized and how the relations among them are associated.

  It formulates all the constraints that are to be applied on the data.

EG:

STUDENT

SID	SNAME	PHNO

What is Instance?

The actual content of the database at a particular point in time. (Or)

The data stored in the database at any given time is an instance of the database

Student

Sid	Name	phno
1201	Venkat	9014901442
1202	teja	9014774422

In the above table 1201, 1202, Venkat etc are said to be instance of student table.

Difference between File system & DBMS:

File system	DBMS
File system is a collection of data. Any 1.management with the file system, user has to write the procedures	1. DBMS is a collection of data and user is not required to write the procedures for managing the database.
File system gives the details of 2. the data representation and Storage of data.	2. DBMS provides an abstract view of data that hides the details.
In File system storing and retrieving of data 3.cannot be done efficiently.	3. DBMS is efficient to use since there are wide varieties of sophisticated techniques to store and retrieve the data.
Concurrent access to the data in the file system 4.has many problems like : Reading the file while other deleting some information, updating some information	4. DBMS takes care of Concurrent access using some form of locking.
File system doesn't provide crash 5. recovery mechanism. Eg. While we are entering some data into the file if System crashes then content of the file is lost	5. DBMS has crash recovery mechanism, DBMS protects user from the effects of system failures.
6.Protecting a file under file system is very difficult.	6. DBMS has a good protection mechanism.

UNIT-2

....

UNIT-2

Relational Algebra & Calculus

....

Preliminaries

A query language is a language in which user requests to retrieve some information from the database. The query languages are considered as higher level languages than programming languages. Query languages are of two types,

Procedural Language
Non-Procedural Language

1. In procedural language, the user has to describe the specific procedure to retrieve the information from the database.

Example: The Relational Algebra is a procedural language.

2. In non-procedural language, the user retrieves the information from the database without describing the specific procedure to retrieve it.

Example: The Tuple Relational Calculus and the Domain Relational Calculus are non-procedural languages.

Relational Algebra

The relational algebra is a procedural query language. It consists of a set of operations that take one or two relations (tables) as input and produce a new relation, on the request of the user to retrieve the specific information, as the output.

The relational algebra contains the following operations,

1) Selection	2) Projection	3) Union	4) Rename
5) Set-Difference	6) Cartesian product	7) Intersection	8) Join
9) Divide	10) Assignment		

The Selection, Projection and Rename operations are called unary operations because they operate only on one relation. The other operations operate on pairs of relations and are therefore called binary operations

1) The Selection (□) operation:

The Selection is a relational algebra operation that uses a condition to select rows from a relation. A new relation (output) is created from another existing relation by selecting only rows requested by the user that satisfy a specified condition. The lower greek letter 'sigma ' is used to denote selection operation.

General Syntax: **Selection condition (relation_name)**

Example: Find the customer details who are living in Hyderabad city from customer relation.

□ city = 'Hyderabad' (customer)

The selection operation uses the column names in specifying the selection condition. Selection conditions are same as the conditions used in the 'if' statement of any programming languages, selection condition uses the relational operators < > <= >= != . It is possible to combine several conditions into a large condition using the logical connectives 'and' represented by ' ' and 'or' represented by ' '.

Example:

Find the customer details who are living in Hyderabad city and whose customer_id is greater than 1000 in Customer relation.

□ city = 'Hyderabad' □□ customer_id > 1000 (customer)

2) The Projection (□) operation:

The projection is a relational algebra operation that creates a new relation by deleting columns from an existing relation i.e., a new relation (output) is created from another existing relation by selecting only those columns requested by the user from projection and is denoted by letter pi (□ .

The Selection operation eliminates unwanted rows whereas the projection operation eliminates unwanted columns. The projection operation extracts specified columns from a table.

Example: Find the customer names (not all customer details) who are living in Hyderabad city from customer relation.

□ customer_name (□ city = 'Hyderabad' (customer))

In the above example, the selection operation is performed first. Next, the projection of the resulting relation on the customer_name column is carried out. Thus, instead of all customer details of customers living in Hyderabad city, we can display only the customer names of customers living in Hyderabad city.

The above example is also known as relational algebra expression because we are combining two or more relational algebra operations (ie., selection and projection) into one at the same time.

Example: Find the customer names (not all customer details) from customer relation.

π customer_name (customer)

The above stated query lists all customer names in the customer relation and this is not called as relational algebra expression because it is performing only one relational algebra operation.

3) The Set Operations: (Union, Intersection, Set-Difference, Cartesian product **)**

i) Union ‘ $\cup$ ’ Operation:

The union denoted by ‘$\cup$’ □ It is a relational algebra operation that creates a union or combination of two relations. The result of this operation, denoted by d $\cup$ b is a relation that includes all tuples that all either in d or in b or in both d and b, where duplicate tuples are eliminated.

Example: Find the customer_id of all customers in the bank who have either an account or a loan or both.

π customer_id (depositor) $\cup$ π customer_id (borrower)

To solve the above query, first find the customers with an account in the bank. That is **π customer_id (depositor)**. Then, we have to find all customers with a loan in the bank, **π customer_id (borrower)**. Now, to answer the above query, we need the union of these two sets, that is, all customer names that appear in either or both of the two relations by π customer_id (depositor) $\cup$ π customer_id (borrower)

If some customers A, B and C are both depositors as well as borrowers, then in the resulting relation, their customer ids will occur only once because duplicate values are eliminated.

Therefore, for a union operation d □ b to be valid, we require that two conditions to be satisfied,

i) The relations depositor and borrower must have same number of attributes / columns.

ii) The domains of ith attribute of depositor relation and the ith attribute of borrower relation must be the same, for all i.

- **The Intersection ‘ □ ’ Operation:**

The intersection operation denoted by ‘ □’ □ It is a relational algebra operation that finds tuples that are in both relations. The result of this operation, denoted by d □ b, is a relation that includes all tuples common in both depositor and borrower relations.

Example: Find the customer_id of all customers in the bank who have both an account and a loan.

□ customer_id (depositor) □□□ □ customer_id (borrower)

The resulting relation of this query, lists all common customer ids of customers who have both an account and a loan. Therefore, for an intersection operation d □ b to be valid, it requires that two conditions to be satisfied as was the case of union operation stated above.

iii) The Set-Difference ‘ □ ’ Operation:

The set-difference operation denoted by’ ’□It□is a relational algebra operation that finds tuples that are in one relation but are not in another.

Example:

□ customer_id (depositor) □□□ □ customer_id (borrower)

The resulting relation for this query, lists the customer ids of all customers who have an account but not a loan. Therefore a difference operation d □ b to be valid, it requires that two conditions to be satisfied as was case of union operation stated ablove.

iv) The Cross-product (or) Cartesian Product ‘ X ’ Operation:

The Cartesian-product operation denoted by a cross ‘X’□ It is a relational algebra operation which allows to combine information from who relations into one relation.

Assume that there are n1 tuple in borrower relation and n2 tuples in loan relation. Then, the result of this operation, denoted by r = borrower X loan, is a relation ‘r’ that includes all the tuples formed by each possible pair of tuples one from the borrower relation and one from the loan relation. Thus, ‘r’ is a large relation containing n1 * n2 tuples.

The drawback of the Cartesian-product is that same attribute name will repeat.

Example: Find the customer_id of all customers in the bank who have loan > 10,000.

□ customer_id (□ borrower.loan_no= loan.loan_no ((□ borrower.loan_no= (borrower X loan)))

That is, get customer_id from borrower relation and loan_amount from loan relation. First, find Cartesian product of borrower X loan, so that the new relation contains both customer_id, loan_amoount with each combination. Now, select the amount, by □ **bloan_ampunt** > 10000.

So, if any customer have taken the loan, then borrower.loan_no = loan.loan_no should be selected as their entries of loan_no matches in both relation.

4) The Renaming “ □ ” Operation:

The Rename operation is denoted by rho ’□. It is a relational algebra operation which is used to give the new names to the relation algebra expression. Thus, we can apply the rename operation to a relation ‘borrower’ to get the same relation under a new name. Given a relation ‘customer’, then the expression returns the same relation ‘customer’ under a new name ‘x’.

□ □ **x (customer)**

After performed this operation, Now there are two relations, one with customer name and second with

‘x’ name. The ‘rename’ operation is useful when we want to compare the values among same column attribute in a relation.

Example: Find the largest account balance in the bank.

Π account.balance (σ account.balance > d.balance (account X ρd (account)))

If we want to find the largest account balance in the bank, Then we have to compare the values among same column (balance) with each other in a same relation account, which is not possible.

So, we rename the relation with a new name'd'. Now, we have two relations of account, one with account name and second with 'd' name. Now we can compare the balance attribute values with each other in separate relations.

5) The Joins " ⋈ " Operation:

The join operation, denoted by join ' ⋈ '. It is a relational algebra operation, which is used to combine

(join) two relations like Cartesian-product but finally removes duplicate attributes and makes the operations (selection, projection, ..) very simple. In simple words, we can say that join connects relations on columns containing comparable information.

There are three types of joins,

i) Natural Join
ii) Outer Join
iii) Theta Join (or) Conditional Join

i) Natural Join:

The natural join is a binary operation that allows us to combine two different relations into one relation and makes the same column in two different relations into only one-column in the resulting relation. Suppose we have relations with following schemas, which contain data on full-time employees.

employee (emp_name, street, city) and

employee_works(emp_name, branch_name, salary)

The relations are,

emp_name	**street**	**city**
Coyote	Town	Hollywood
Rabbit	Tunnel	Carrotville
Smith	Revolver	Vally
Williams	Seaview	Seattle

employee relation

emp_name	**branch_name**	**salary**
Coyote	Mesa	15000
Rabbit	Mesa	12000
Gates	Redmond	25000
Williams	Redmond	23000

employee_works relation

If we want to generate a single relation with all the information (emp_name, street, city, branch_name and salary) about full-time employees. then, a possible approach would be to use the natural-join operation as follows,

employee $\bowtie$ **employee_works**

The result of this expression is the relation,

emp_name	**street**	**city**	**branch_name**	**salary**
Coyote	Town	Hollywood	Mesa	15000
Rabbit	Tunnel	Carrotville	Mesa	12000
Williams	Seaview	Seattle	Redmond	23000

result of Natural join

We have lost street and city information about Smith, since tuples describing smith is absent in employee_works. Similarly, we have lost branch_name and salary information about Gates, since the tuple describing Gates is absent from the employee relation. Now, we can easily perform select or reject query on new join relation.

Example: Find the employee names and city who have salary details.

Π **emp_name, salary, city (employee $\bowtie$ employee_works)**

The join operation selects all employees with salary details, from where we can easily project the employee names, cities and salaries. Natural Join operation results in some loss of information.

ii) Outer Join:

The drawback of natural join operation is some loss of information. To overcome the drawback of natural join, we use outer-join operation. The outer-join operation is of three types,

a) Left outer-join ($\bowtie$)
b) Right outer-join ($\bowtie$)
c) Full outer-join ($\bowtie$)

a) Left Outer-join:

The left outer-join takes all tuples in left relation that did not match with any tuples in right relation, adds the tuples with null values for all other columns from right relation and adds them to the result of natural join as follows,

The relations are,

emp_name	**street**	**city**
Coyote	Town	Hollywood
Rabbit	Tunnel	Carrotville
Smith	Revolver	Valley
Williams	Seaview	Seattle

employee relation

emp_name	**branch_name**	**salary**
Coyote	Mesa	15000
Rabbit	Mesa	12000
Gates	Redmond	25000
Williams	Redmond	23000

employee_works relation

The result of this expression is the relation,

emp_name	street	city	branch_name	salary
Coyote	Town	Hollywood	Mesa	15000
Rabbit	Tunnel	Carrotville	Mesa	12000
Smith	Revolver	Valley	null	null
Williams	Seaview	Seattle	Redmond	23000

result of Left Outer-join

b) Right Outer-join:

The right outer-join takes all tuples in right relation that did not match with any tuples in left relation, adds the tuples with null values for all other columns from left relation and adds them to the result of natural join as follows,

The relations are,

emp_name	street	city
Coyote	Town	Hollywood
Rabbit	Tunnel	Carrotville
Smith	Revolver	Valley
Williams	Seaview	Seattle

emp_name	branch_name	salary
Coyote	Mesa	15000
Rabbit	Mesa	12000
Gates	Redmond	25000
Williams	Redmond	23000

The result of this expression is the relation,

emp_name	street	city	branch_name	salary
Coyote	Town	Hollywood	Mesa	15000
Rabbit	Tunnel	Carrotville	Mesa	12000
Gates	null	null	Redmond	25000
Williams	Seaview	Seattle	Redmond	23000

result of Right Outer-join

c) Full Outer-join:

The full outer-join operation does both of those operations, by adding tuples from left relation that did not match any tuples from the reight relations, as well as adds tuples from the right relation that did not match any tuple from the left relation and adding them to the result of natural join as follows,

The relations are,

emp_name	street	city
Coyote	Town	Hollywood
Rabbit	Tunnel	Carrotville
Smith	Revolver	Valley
Williams	Seaview	Seattle

employee relation

emp_name	branch_name	salary
Coyote	Mesa	15000
Rabbit	Mesa	12000
Gates	Redmond	25000
Williams	Redmond	23000

employee_works relation

The result of this expression is the relation,

emp_name	street	city	branch_name	salary
Coyote	Town	Hollywood	Mesa	15000
Rabbit	Tunnel	Carrotville	Mesa	12000
Smith	Revolver	Valley	null	null
Gates	null	null	Redmond	25000
Williams	Seaview	Seattle	Redmond	23000

result of Full Outer-join

iii) Theta Join (or) Condition join:

The theta join operation, denoted by symbol “ ⋈□□ ”. It is an extension to the natural join operation that□combines two relations into one relation with a selection condition (□).

The theta join operation is expressed as employee ⋈ salary < 19000 employee_works and the resulting is as follows,

employee ⋈ salary > 20000 employee_works

There are two tuples selected because their salary greater than 20000 (salary > 20000). The result of theta join as follows,

The relations are,

emp_name	street	city
Coyote	Town	Hollywood
Rabbit	Tunnel	Carrotville
Smith	Revolver	Valley
Williams	Seaview	Seattle

emp_name	branch_name	salary
Coyote	Mesa	15000
Rabbit	Mesa	12000
Gates	Redmond	25000
Williams	Redmond	23000

The result of this expression is the relation,

emp_name	street	city	branch_name	salary
Gates	null	null	Redmond	25000
Williams	Seaview	Seattle	Redmond	23000

result of Theta Join (or) Condition Join

6) The Division “ ÷ ” Operation:

The division operation, denoted by “ ÷ ”, is a relational algebra operation that creates a new relation by selecting the rows in one relation that does not match rows in another relation.

Let, Relation A is (x1, x2, …., xn, y1, y2, …, ym) and
Relation B is (y1, y2, …, ym),

Where, y1, y2, …, ym tuples are common to the both relations A and B with same domain compulsory.

Then, A ÷ B = new relation with x1, x2, …., xn tuples. Relation A and B represents the dividend and divisor respectively. A tuple ‘t’ is in a ÷ b, if and only if two conditions are to be satisfied,

- t is in ∏A-B (r)

- for every tuple tb in B, there is a tuple ta in A satisfying the following two things,

  1. ta[B] = tb[B]

  2. ta[A-B] = t

Relational Calculus

Relational calculus is an alternative to relational algebra. In contrast to the algebra, which is procedural, the relational calculus is non-procedural or declarative.

It allows user to describe the set of answers without showing procedure about how they should be computed. Relational calculus has a big influence on the design of commercial query languages such as SQL and QBE (Query-by Example).

Relational calculus are of two types,

- Tuple Relational Calculus (TRC)
- Domain Relational Calculus (DRC)

Variables in TRC takes tuples (rows) as values and TRC had strong influence on SQL.

Variables in DRC takes fields (attributes) as values and DRC had strong influence on QBE.

i) Tuple Relational Calculus (TRC):

The tuple relational calculus, is a non-procedural query language because it gives the desired information without showing procedure about how they should be computed.

A query in Tuple Relational Calculus (TRC) is expressed as **{ T | p(T) }**

Where, T - tuple variable,
P(T) - 'p' is a condition or formula that is true for 't'.

In addition to that we use,

T[A] - to denote the value of tuple t on attribute A and

T □ r - to denote that tuple t is in relation r.

<u>Examples:</u>

1) Find all loan details in loan relation.

{ t | t □ loan }

This query gives all loan details such as loan_no, loan_date, loan_amt for all loan table in a bank.
2) Find all loan details for loan amount over 100000 in loan relation.

{ t | t □ loan □ t [loan_amt] > 100000 }
This query gives all loan details such as loan_no, loan_date, loan_amt for all loan over 100000 in a loan table in a bank.

ii) Domain Relational Calculus (DRC):

A Duple Relational Calculus (DRC) is a variable that comes in the range of the values of domain (data types) of some columns (attributes).

A Domain Relational Calculus query has the form,

{ < x1, x2, …., xn > | p(< x1, x2, …., xn >) }

Where, each xi is either a domain variable or a constant and p(< x1, x2, …., xn >) denotes a DRC formula.

A DRC formula is defined in a manner that is very similar to the definition of a TRC formula. The main difference is that the variables are domain variables.

Examples:

1) Find all loan details in loan relation.

{ < N, D, A > | < N, D, A > □ loan }

This query gives all loan details such as loan_no, loan_date, loan_amt for all loan table in a bank. Each column is represented with an initials such as N- loan_no, D – loan_date, A – loan_amt. The condition < N, D, A > □ loan ensures that the domain variables N, D, A are restricted to the column domain.

2.3.1 Expressive power of Algebra and Calculus

The tuple relational calculus restricts to safe expressions and is equal in expressive power to relational algebra. Thus, for every relational algebra expression, there is an equivalent expression in the tuple relational calculus and for tuple relational calculus expression, there is an equivalent relational algebra expression.

A safe TRC formula Q is a formula such that,

□ For any given I, the set of answers for Q contains only values that are in dom(Q, I).

□ For each sub expression of the form R(p(R)) in Q, if a tuple r makes the formula true, then r contains only constraints in dom(Q, I).

3) For each sub expression of the form□ R(p(R)) in Q, if a tuple r contains a constant that is not in dom(Q, I), then r must make the formula true.

The expressive power of relational algebra is often used as a metric how powerful a relational database query language is. If a query language can express all the queries that we can express in relational algebra, it is said to be relationally complete. A practical query language is expected to be relationally complete. In addition, commercial query languages typically support features that allow us to express some queries that cannot be expressed in relational algebra.

When the domain relational calculus is restricted to safe expression, it is equivalent in expressive power to the tuple relational calculus restricted to safe expressions. All three of the following are equivalent,

- The relational algebra
- The tuple relational calculus restricted to safe expression
- The domain relational calculus restricted to safe expression

UNIT - 3

UNIT-3

THE DATABASE LANGUAGE SQL

Introduction to SQL:

What is SQL?

1. SQL is Structured Query Language, which is a computer language for storing, manipulating and retrieving data stored in relational database.

2. SQL is the standard language for Relation Database System. All relational database management systems like MySQL, MS Access, and Oracle, Sybase, Informix, postgres and SQL Server use SQL as standard database language.

Why SQL?

3. Allows users to access data in relational database management systems.

4. Allows users to describe the data.

5. Allows users to define the data in database and manipulate that data.

6. Allows embedding within other languages using SQL modules, libraries & pre-compilers.

7. Allows users to create and drop databases and tables.

8. Allows users to create view, stored procedure, functions in a database.

9. Allows users to set permissions on tables, procedures and views

History:

10. **1970 --** Dr. E. F. "Ted" of IBM is known as the father of relational databases. He described a relational model for databases.

11. **1974 --** Structured Query Language appeared.

12. **1978 --** IBM worked to develop Codd's ideas and released a product named System/R.

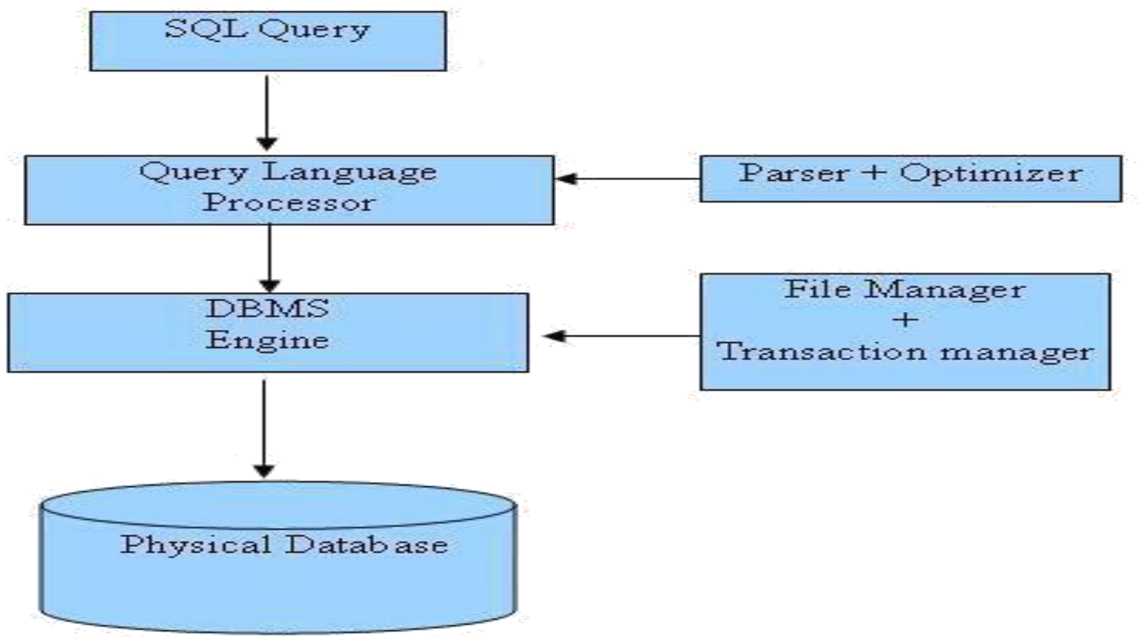
13. **1986 --** IBM developed the first prototype of relational database and standardized by ANSI. The first relational database was released by Relational Software and its later becoming Oracle.

SQL Process:

14. When you are executing an SQL command for any RDBMS, the system determines the best way to carry out your request and SQL engine figures out how to interpret the task.

15. There are various components included in the process. These components are Query Dispatcher, Optimization Engines, Classic Query Engine and SQL Query Engine, etc. Classic query engine handles all non-SQL queries, but SQL query engine won't handle logical files.

SQL Process:



SQL Commands:

The standard SQL commands to interact with relational databases are CREATE, SELECT, INSERT, UPDATE, DELETE and DROP. These commands can be classified into groups based on their nature. They are:

DDL Commands

DML Commands

DCL Commands

DRL/DQL Commands

TCL Commands

Data Definition Language (DDL) Commands:

Command	Description
CREATE	Creates a new table, a view of a table, or other object in database
ALTER	Modifies an existing database object, such as a table.
DROP	Deletes an entire table, a view of a table or other object in the database.
TRUNCATE	Truncates the table values without delete table structure

Data Manipulation Language (DML) Commands:

Command	Description
INSERT	Creates a record
UPDATE	Modifies records
DELETE	Deletes records

Data Control Language (DCL) Commands:

Command	Description
GRANT	Gives a privilege to user
REVOKE	Takes back privileges granted from user

Data Query Language (DQL) Commands:

Command	Description
SELECT	Retrieves certain records from one or more tables

Transaction Control Language (TCL) Commands:

Command	Description
commit	Save work done
Save point	Identify a point in a transaction to which we can later roll back.
Roll backs	Restore database to original since the last Commit

What is Query?

- A query is a question.
- A query is formulated for a relation/table to retrieve some useful information from the table.
- Different query languages are used to frame queries.

Form of Basic SQL Query

- The basic form of an SQL query is as follows:
  SELECT [DISTINCT] select-list (List of Attributes) **FROM** from-list (Table (s) Name (s))

  WHERE qualification (Condition)

- This SELECT command is used to retrieve the data from the database.

- For retrieving the data every query must have SELECT clause, which specifies what columns to be selected.

- And FROM clause, which specifies the table's names. The WHERE clause, specifies the selection condition.

- **SELECT**: The SELECT list is list of column names of tables named in the FROM list. Column names can be prefixed by a range variable.

- **FROM:** The FROM list in the FROM clause is a list of table names. A Table name can be followed by a range variable. A range variable is particularly useful when the same table name appears more than once in the from-list.

- **WHERE:** The qualification in the WHERE clause is a Boolean combination (i.e., an expression using the logical connectives AND, OR, and NOT) of conditions of the form expression op expression, where op is one of the comparison operators {<, <=, =, <>, >=,>}.

- **An expression is a column name, a constant, or an (arithmetic or string) expression.**

- **DISTINCT**: The DISTINCT keyword is used to display the unique tuple or eliminated the duplicate tuple.

- This DISTINCT keyword is Optional.

<u>DDL Commands:</u>

- The following are the DDL commands. They are:

Create

Alter

Truncate

Drop

CREATE:

- The SQL CREATE TABLE statement is used to create a new table.

- Creating a basic table involves naming the table and defining its columns and each column's data type.

Syntax:

- Basic syntax of CREATE TABLE statement is as follows:

CREATE TABLE table name (column1 datatype (size), column2 datatype (size), column3 datatype (size) ... columnN datatype (size), PRIMARY KEY (one or more columns)); Example:
SQL> create table customers (id number (10) not null, name varchar2 (20) not null, age number (5) not null, address char (25), salary decimal (8, 2), primary key (id));

ALTER:

- SQL **ALTER TABLE** command is used to add, delete or modify columns in an existing table

Syntax:

- The basic syntax of **ALTER TABLE** to add a new column in an existing table is as follows:

ALTER TABLE table_name ADD column_name datatype;

EX: ALTER TABLE CUSTOMERS ADD phno number (12);

ii) The basic syntax of ALTER TABLE to **DROP COLUMN** in an existing table is as follows:

ALTER TABLE table_name DROP COLUMN column_name; EX: ALTER TABLE CUSTOMERS DROP column phno;

- The basic syntax of ALTER TABLE to change the **DATA TYPE** of a column in a table is as follows:

ALTER TABLE table_name MODIFY COLUMN column_name datatype;

Ex: ALTER TABLE customer MODIFY COLUMN phno number(12);

- The basic syntax of ALTER TABLE to add a **NOT NULL** constraint to a column in a table is as follows:

  ALTER TABLE table_name MODIFY column_name datatype NOT NULL;

  Ex:

  ALTER TABLE customers MODIFY phno number (12); NOT NULL;

- The basic syntax of ALTER TABLE to **ADD PRIMARY KEY** constraint to a table is as follows:

  ALTER TABLE table_name ADD PRIMARY KEY (column1, column2...);

  Ex:

  ALTER TABLE customer ADD PRIMARY KEY (id,phno);

TRUNCATE:

- SQL TRUNCATE TABLE command is used to delete complete data from an existing table.

Syntax:
The basic syntax of **TRUNCATE TABLE** is as follows:

TRUNCATE TABLE table name;

EX:

TRUNCATE TABLE student;

SELECT * FROM student;

Empty set (0.00 sec).

DROP:

SQL DROP TABLE statement is used to remove a table definition and all data, indexes, triggers, constraints, and permission specifications for that table.

Syntax:
Basic syntax of DROP TABLE statement is as follows:

DROP TABLE table_name;

EX: DROP TABLE student;

DML Commands:

The following are the DML commands. They are:

- Insert
- Update
- Delete

INSERT:

SQL INSERT INTO Statement is used to add new rows of data to a table in the database. There are two basic syntaxes of INSERT INTO statement as follows:

Syntax1:

INSERT INTO TABLE_NAME (column1, column2, column3,...columnN)]
VALUES (value1, value2, value3,...valueN);

- Here, column1, column2...columnN are the names of the columns in the table into which you want to insert data.

EX:

```
insert into customers (id,name,age,address,salary) values (1, 'ramesh', 32, 'ahmedabad', 2000);
insert into customers (id,name,age,address,salary) values (2, 'khilan', 25, 'delhi', 1500.00 );

2 rows inserted.
```

<u>Syntax2:</u>

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

Ex:

```
insert into customers values (1, 'ramesh', 32, 'ahmedabad', 2000.00 );
```

<u>UPDATE:</u>

- SQL UPDATE Query is used to modify the existing records in a table.
- We can use WHERE clause with UPDATE query to update selected rows, otherwise all the rows would be affected.

<u>Syntax:</u>

- The basic syntax of UPDATE query with WHERE clause is as follows:

UPDATE table_name SET column1 = value1, column2 = value2...., columnN = valueN WHERE [condition];

EX:

- UPDATE CUSTOMERS SET ADDRESS = 'Pune' WHERE ID = 6;
- UPDATE CUSTOMERS SET ADDRESS = 'Pune', SALARY = 1000.00;

<u>DELETE:</u>

SQL **DELETE** Query is used to delete the existing records from a table.

You can use WHERE clause with DELETE query to delete selected rows, otherwise all the records would be deleted.

Syntax:

The basic syntax of DELETE query with WHERE clause is as follows:

DELETE FROM table_name WHERE [condition];

Ex: DELETE FROM CUSTOMERS WHERE ID = 6;

If you want to DELETE all the records from CUSTOMERS table, you do not need to use WHERE clause and DELETE query would be as follows:

DELETE FROM CUSTOMERS;

DRL/DQL Command:

The select command is comes under DRL/DQL.

SELECT:

SELECT Statement is used to fetch the data from a database table which returns data in the form of result table. These result tables are called result-sets.

Syntax1:

The Following Syntax is used to retrieve specific attributes from the table is as follows:

SELECT column1, column2, columnN FROM table_name;

Here, column1, column2...are the fields of a table whose values you want to fetch.

The Following Syntax is used to retrieve all the attributes from the table is as follows:

SELECT * FROM table_name;

Ex: Select * from student;

Distinct:

- SQL **DISTINCT** keyword is used in conjunction with SELECT statement to eliminate all the duplicate records and fetching only unique records.

- There may be a situation when you have multiple duplicate records in a table. While fetching such records, it makes more sense to fetch only unique records instead of fetching duplicate records.

Syntax:

- The basic syntax of DISTINCT keyword to eliminate duplicate records is as follows:

SELECT DISTINCT column1, column2,.....columnN FROM table_name WHERE [condition];

Ex: SELECT DISTINCT SALARY FROM CUSTOMERS ORDER BY SALARY;

Queries involving more than one relation (or) Full Relation Operations :

- The following set operations are used to write a query to combine more than one relation. They are:

Union

Intersect

Except

UNION:

SQL UNION clause/operator is used to combine the results of two or more SELECT statements without returning any duplicate rows.

To use UNION, each SELECT must have the same number of columns selected, the same number of column expressions, the same data type, and have them in the same order, but they do not have to be the same length.

Syntax:

The basic syntax of **UNION** is as follows:

SELECT column1 [, column2] FROM table1 [, table2] [WHERE condition]
UNION

SELECT column1 [, column2] FROM table1 [, table2] [WHERE condition]

EX:

SELECT ID, NAME, AMOUNT, DATE FROM CUSTOMERS LEFT JOIN ORDERS
ON CUSTOMERS.ID = ORDERS.CUSTOMER_ID

UNION

SELECT ID, NAME, AMOUNT, DATE FROM CUSTOMERS RIGHT JOIN ORDERS
ON CUSTOMERS.ID = ORDERS.CUSTOMER_ID;

<u>UNION ALL Clause:</u>

The UNION ALL operator is used to combine the results of two SELECT statements including duplicate rows.

The same rules that apply to UNION apply to the UNION ALL operator.

<u>Syntax:</u>

- The basic syntax of UNION ALL is as follows:

SELECT column1 [, column2] FROM table1 [, table2] [WHERE condition]
UNION ALL

SELECT column1 [, column2] FROM table1 [, table2] [WHERE condition]
EX:

SELECT ID, NAME, AMOUNT, DATE FROM CUSTOMERS LEFT JOIN ORDERS
ON CUSTOMERS.ID = ORDERS.CUSTOMER_ID

UNION ALL

SELECT ID, NAME, AMOUNT, DATE FROM CUSTOMERS RIGHT JOIN ORDERS
ON CUSTOMERS.ID = ORDERS.CUSTOMER_ID;

INTERSECT:

- The SQL **INTERSECT** clause/operator is used to combine two SELECT statements, but returns rows only from the first SELECT statement that are identical to a row in the second SELECT statement.

- This means INTERSECT returns only common rows returned by the two SELECT statements.

- Just as with the UNION operator, the same rules apply when using the INTERSECT operator. MySQL does not support INTERSECT operator

Syntax:

The basic syntax of **INTERSECT** is as follows:

SELECT column1 [, column2] FROM table1 [, table2] [WHERE condition]
INTERSECT

SELECT column1 [, column2] FROM table1 [, table2] [WHERE condition];
Ex:

```
SELECT ID, NAME, AMOUNT, DATE FROM CUSTOMERS LEFT JOIN ORDERS
ON CUSTOMERS.ID = ORDERS.CUSTOMER_ID

INTERSECT

SELECT ID, NAME, AMOUNT, DATE FROM CUSTOMERS RIGHT JOIN ORDERS
ON CUSTOMERS.ID = ORDERS.CUSTOMER_ID;
```

EXCEPT:

- The SQL EXCEPT clause/operator is used to combine two SELECT statements and returns rows from the first SELECT statement that are not returned by the second SELECT statement.

- This means EXCEPT returns only rows, which are not available in second SELECT statement.

- Just as with the UNION operator, the same rules apply when using the EXCEPT operator.

- MySQL does not support EXCEPT operator.

Syntax:

The basic syntax of EXCEPT is as follows:

SELECT column1 [, column2] FROM table1 [, table2] [WHERE condition]
EXCEPT

SELECT column1 [, column2] FROM table1 [, table2] [WHERE condition];
EX:

```
SELECT ID, NAME, AMOUNT, DATE FROM CUSTOMERS LEFT JOIN ORDERS
ON CUSTOMERS.ID = ORDERS.CUSTOMER_ID

EXCEPT

SELECT ID, NAME, AMOUNT, DATE FROM CUSTOMERS RIGHT JOIN ORDERS
ON CUSTOMERS.ID = ORDERS.CUSTOMER_ID;
```

SQL Operators

What is an Operator in SQL?

An operator is a reserved word or a character used primarily in an SQL statement's WHERE clause to perform operation(s), such as comparisons and arithmetic operations.

Operators are used to specify conditions in an SQL statement and to serve as conjunctions for multiple conditions in a statement.

1. Arithmetic operators
2. Comparison operators
3. Logical operators
4. Operators used to negate conditions

SQL Arithmetic Operators:

Operator	Description	Example
+	Addition - Adds values on either side of the operator	a + b will give 30
-	Subtraction - Subtracts right hand operand from left hand operand	a - b will give -10
*	Multiplication - Multiplies values on either side of the operator	a * b will give 200
/	Division - Divides left hand operand by right hand operand	b / a will give 2
%	Modulus - Divides left hand operand by right hand operand and returns remainder	b % a will give 0

SQL Comparison Operators:

Operator	Description	Example
		[redacted]
=	Checks if the values of two operands are equal or not, if yes then condition becomes true.	(a = b) is not true.
!=	Checks if the values of two operands are equal or not, if values are not equal then condition becomes true.	(a != b) is true.
<>	Checks if the values of two operands are equal or not, if values are not equal then condition becomes true.	(a <> b) is true.
>	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	(a > b) is not true.
<	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	(a < b) is true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes	(a >= b) is not true

	true.	
<=	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.	(a <= b) is true.
!<	Checks if the value of left operand is not less than the value of right operand, if yes then condition becomes true.	(a !< b) is false.
!>	Checks if the value of left operand is not greater than the value of right operand, if yes then condition becomes true.	a !> b) is true.

- The following are example illustrate the relational operators usage on tables:
  <u>Ex:</u>

  - SELECT * FROM CUSTOMERS WHERE SALARY > 5000;
  - SELECT * FROM CUSTOMERS WHERE SALARY = 2000;
  - SELECT * FROM CUSTOMERS WHERE SALARY != 2000;
  - SELECT * FROM CUSTOMERS WHERE SALARY >= 6500;

<u>SQL Logical Operators:</u>

Operator	Description
AND	The AND operator allows the existence of multiple conditions in an SQL statement's WHERE clause
OR	The OR operator is used to combine multiple conditions in an SQL statement's WHERE clause.
NOT	The NOT operator reverses the meaning of the logical operator with which it is used. Eg: NOT EXISTS, NOT BETWEEN, NOT IN, etc. **This is a negatation operator**

- SQL **AND** and **OR** operators are used to combine multiple conditions to narrow data in an SQL statement. These two operators are called conjunctive operators.

- These operators provide a means to make multiple comparisons with different operators in the same SQL statement.

AND Operator:

- The **AND** operator allows the existence of multiple conditions in an SQL statement's WHERE clause.

Syntax:

- The basic syntax of AND operator with WHERE clause is as follows:

SELECT column1, column2, columnN FROM table_name WHERE [condition1] AND [condition2]...AND [conditionN];

Ex:

SELECT * FROM CUSTOMERS WHERE AGE >= 25 AND SALARY >= 6500;

OR Operator:

- The OR operator is used to combine multiple conditions in an SQL statement's WHERE clause.

Syntax:

- The basic syntax of OR operator with WHERE clause is as follows:

~~**SELECT column1, column2, columnN FROM table_name WHERE [condition1]**~~
OR [condition2]...OR [conditionN];

Ex:

SELECT * FROM CUSTOMERS WHERE AGE >= 25 OR SALARY >= 6500;

NOT Operator:

- The NOT operator reverses the meaning of the logical operator with which it is used. Eg: NOT EXISTS, NOT BETWEEN, NOT IN, etc. **This is a negate operator.**

Syntax:

SELECT column1, column2, … column FROM table_name WHERENOT [condition];

EX:

SELECT * FROM CUSTOMERS WHERE AGE IS NOT NULL;

Special Operators in SQL:

Operator	**Description**
BETWEEN	The BETWEEN operator is used to search for values that are within a set of values, given the minimum value and the maximum value.
EXISTS	The EXISTS operator is used to search for the presence of a row in a specified table that meets certain criteria.
IN	The IN operator is used to compare a value to a list of literal values that have been specified.
LIKE	The LIKE operator is used to compare a value to similar values using wildcard operators.
IS NULL	The NULL operator is used to compare a value with a NULL value.
UNIQUE	The UNIQUE operator searches every row of a specified table for uniqueness (no duplicates).

LIKE Operator:

- SQL **LIKE** clause is used to compare a value to similar values using wildcard operators. There are two wildcards used in conjunction with the LIKE operator:

  1. The percent sign (%)

  2. The underscore (_)

- The percent sign represents zero, one, or multiple characters.

- The underscore represents a single number or character.

- The symbols can be used in combinations.

<u>Syntax:</u>

- The basic syntax of % and _ is as follows:

SELECT FROM table_name
WHERE column LIKE 'XXXX%'

or

SELECT FROM table_name
WHERE column LIKE '%XXXX%'

or

SELECT FROM table_name
WHERE column LIKE 'XXXX_'
or

SELECT FROM table_name WHERE column LIKE '_XXXX'

or

SELECT FROM table_name WHERE column LIKE '_XXXX_

Ex:

Statement	**Description**
WHERE SALARY LIKE 's%'	**Finds any values that start with s**
WHERE SALARY LIKE '%sad%'	**Finds any values that have sad in any position**
WHERE SALARY LIKE '_00%'	**Finds any values that have 00 in the second and third positions**
WHERE SALARY LIKE '2_%_%'	**Finds any values that start with 2 and are at least 3 characters in length**
WHERE SALARY LIKE '%r'	**Finds any values that end with r**
WHERE SALARY LIKE '_2%3'	**Finds any values that have a 2 in the second position and end with a 3**
WHERE SALARY LIKE '2___3'	**Finds any values in a five-digit number that start with 2 and end with**

BETWEEN Operator

The BETWEEN operator is used to select values within a range**.**

- **Syntax:**

  SELECT *column_name(s)*
  FROM *table_name*
  WHERE *column_name* BETWEEN *value1* AND *value2;*

  EX: SELECT * FROM Products WHERE Price BETWEEN 10 AND 20;

  NOT BETWEEN Operator:

  SELECT * FROM Products WHERE Price NOT BETWEEN 10 AND 20;

  IN Operator:

- The IN operator allows you to specify multiple values in a WHERE clause.

  Syntax

  SELECT *column_name(s)*
  FROM *table_name*

  WHERE *column_name* IN (*value1,value2,...*);

  Ex: SELECT * FROM Customers WHERE salary IN (5000, 10000);
  SQL Joins:

- SQL Joins clause is used to combine records from two or more tables in a database.
- A JOIN is a means for combining fields from two tables by using values common to each.
- Consider the following two tables, CUSTOMERS and ORDERS tables are as follows:

CUSTOMERS TABLE

| ID | NAME | AGE | ADDRESS |
SALARY | | 1 | Ramesh | 32 |
Ahmedabad | 2000.00 |

| 2 | Khilan | 25 | Delhi |
1500.00 | | 3 | kaushik | 23 |
Kota | 2000.00 |

| 4 | Chaitali | 25 | Mumbai |
6500.00 | | 5 | Hardik | 27 |
Bhopal | 8500.00 |

| 6 | Komal | 22 | MP | 4500.00 |

| 7 | Muffy | 24 | Indore | 10000.00 |

ORDERS TABLE

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

Ex:

SELECT ID, NAME, AGE, AMOUNT FROM CUSTOMERS, ORDERS WHERE CUSTOMERS.ID = ORDERS.CUSTOMER_ID;

This would produce the following
result: | ID | NAME | AGE |
AMOUNT |

| 3 | kaushik | 23 |
3000 | | 3 | kaushik
|23|1500||2|
Khilan | 25 | 1560

NOTE:

- Join is performed in the WHERE clause. Several operators can be used to join tables, such as =, <, >, <>, <=, >=, !=, BETWEEN, LIKE, and NOT; they can all be used to join tables. However, the most common operator is the equal symbol.

SQL Join Types:

- There are different types of joins available in SQL: They are:
  - INNER JOIN
  - OUTER JOIN
  - SELF JOIN
  - CARTESIAN JOIN

INNER JOIN:

- The most frequently used and important of the joins is the **INNER JOIN**. They are also referred to as an EQUIJOIN.

- The INNER JOIN creates a new result table by combining column values of two tables (table1 and table2) based upon the join-predicate.

- The query compares each row of table1 with each row of table2 to find all pairs of rows which satisfy the join-predicate.

- When the join-predicate is satisfied, column values for each matched pair of rows of A and B are combined into a result row.

Syntax:

- The basic syntax of INNER JOIN is as follows:

SELECT table1.column1, table2.column2... FROM table1 INNER JOIN table2 ON table1.common_filed = table2.common_field;

Ex: SELECT ID, NAME, AMOUNT, DATE FROM
CUSTOMERS INNER JOIN

______ORDERS CUSTOMERS.ID = ORDERS.CUSTOMER_ID;

OUTER JOIN:

- The Outer join can be classified into 3 types. They are:

Left Outer Join\
Right Outer Join
Full Outer Join

Left Outer Join:

- The SQL LEFT JOIN returns all rows from the left table, even if there are no matches in the right table.
- This means that a left join returns all the values from the left table, plus matched values from the right table or NULL in case of no matching join predicate.

Syntax:

- The basic syntax of **LEFT JOIN** is as follows:

SELECT table1.column1, table2.column2... FROM table1 LEFT JOIN table2 ON table1.common_filed = table2.common_field;

EX: SELECT ID, NAME, AMOUNT, DATE FROM CUSTOMERS

LEFT JOIN ORDERS ON CUSTOMERS.ID = ORDERS.CUSTOMER_ID**;**

RIGHT JOIN:

- The SQL **RIGHT JOIN** returns all rows from the right table, even if there are no matches in the left table.
- This means that a right join returns all the values from the right table, plus matched values from the left table or NULL in case of no matching join predicate.

Syntax:

- The basic syntax of **RIGHT JOIN** is as follows:

SELECT table1.column1, table2.column2... FROM table1 RIGHT JOIN table2 ON table1.common_filed = table2.common_field;

Ex: SELECT ID, NAME, AMOUNT, DATE FROM CUSTOMERS
RIGHT JOIN ORDERS ON CUSTOMERS.ID = ORDERS.CUSTOMER_ID;

FULL JOIN:

- The SQL **FULL JOIN** combines the results of both left and right outer joins.
- The joined table will contain all records from both tables, and fill in NULLs for missing matches on either side.

__Syntax:__

- The basic syntax of **FULL JOIN** is as follows:

SELECT table1.column1, table2.column2... FROM table1 FULL JOIN table2 ON table1.common_filed = table2.common_field;

__Ex:__ SELECT ID, NAME, AMOUNT, DATE FROM CUSTOMERS FULL JOIN ORDERS ON CUSTOMERS.ID = ORDERS.CUSTOMER_ID;

__**SELF JOIN:**__

- The SQL **SELF JOIN** is used to join a table to it as if the table were two tables, temporarily renaming at least one table in the SQL statement.

__Syntax:__

- The basic syntax of **SELF JOIN** is as follows:

SELECT a.column_name, b.column_name...FROM table1 a, table1 b WHERE a.common_filed = b.common_field;

__Ex:__

SELECT a.ID, b.NAME, a.SALARY FROM CUSTOMERS a, CUSTOMERS b WHERE a.SALARY < b.SALARY;

__**CARTESIAN JOIN:**__

- The CARTESIAN JOIN or CROSS JOIN returns the cartesian product of the sets of records from the two or more joined tables.

- Thus, it equates to an inner join where the join-condition always evaluates to True or where the join-condition is absent from the statement.

__Syntax:__

- The basic syntax of **CROSS JOIN** is as follows:

SELECT table1.column1, table2.column2... FROM table1, table2 [, table3]; __Ex:__ SELECT ID, NAME, AMOUNT, DATE FROM CUSTOMERS, ORDERS;

__**VIEWS IN SQL:**__

- A view is nothing more than a SQL statement that is stored in the database with an associated name.

- A view is actually a composition of a table in the form of a predefined SQL query.
- A view can contain all rows of a table or select rows from a table.

- A view can be created from one or many tables which depends on the written SQL query to create a view.

- Views, which are kind of virtual tables, allow users to do the following:
    - Structure data in a way that users or classes of users find natural or intuitive.

    - Restrict access to the data such that a user can see and (sometimes) modify exactly what they need and no more.

    - Summarize data from various tables which can be used to generate reports.

  Advantages of views:
    - Views provide data security

    - Different users can view same data from different perspective in different ways at the same time.

    - Views cal also be used to include extra/additional information

Creating Views:

- Database views are created using the CREATE VIEW statement. Views can be created from a single table, multiple tables, or another view.

- To create a view, a user must have the appropriate system privilege according to the specific implementation.

- The basic CREATE VIEW syntax is as follows:

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

  Ex: CREATE VIEW CUSTOMERS_VIEW AS SELECT name, age FROM CUSTOMERS**;**

  You can query CUSTOMERS_VIEW in similar way as you query an actual table. Following is the example:

  SELECT * FROM CUSTOMERS_VIEW;

Updating a View:

A view can be updated under certain conditions: TUTORIALS POINT Simply Easy Learning

- The SELECT clause may not contain the keyword DISTINCT.
- The SELECT clause may not contain summary functions.
- The SELECT clause may not contain set functions.
- The SELECT clause may not contain set operators.
- The SELECT clause may not contain an ORDER BY clause.
- The FROM clause may not contain multiple tables.
- The WHERE clause may not contain sub queries.
- The query may not contain GROUP BY or HAVING.

NOTE:

So if a view satisfies all the above mentioned rules then you can update a view. Following is an example to update the age of Ramesh:

Ex: UPDATE CUSTOMERS_VIEW SET AGE = 35 WHERE name='Ramesh';

Deleting Rows into a View:

- Rows of data can be deleted from a view. The same rules that apply to the UPDATE and INSERT commands apply to the DELETE command.
- Following is an example to delete a record having AGE= 22.

delete from customers_view where age = 22;

Dropping Views:

- Obviously, where you have a view, you need a way to drop the view if it is no longer needed.
- The syntax is very simple as given below:

DROP VIEW view_name;

- Following is an example to drop CUSTOMERS_VIEW from CUSTOMERS table:

DROP VIEW CUSTOMERS_VIEW;

GROUP BY:

- SQL **GROUP BY** clause is used in collaboration with the SELECT statement to arrange identical data into groups.

- The GROUP BY clause follows the WHERE clause in a SELECT statement and precedes the ORDER BY clause.

Syntax:

- The GROUP BY clause must follow the conditions in the WHERE clause and must precede the ORDER BY clause if one is used.

SELECT column1,
column2 FROM
table_name WHERE
[conditions]

GROUP BY column1,
column2 ORDER BY
column1, column2; **Ex:**

select name, sum(salary) from customers group by name**;**

ORDER BY:

- SQL **ORDER BY** clause is used to sort the data in ascending or descending order, based on one or more columns.

- Some database sorts query results in ascending order by default.

Syntax:

- The basic syntax of ORDER BY clause is as follows:

SELECT
column-list
FROM
table_name

[WHERE condition]

[ORDER BY column1, column2, .. columnN] [ASC | DESC]; **Ex:**

1. select * from customers order by name, salary;
2. select * from customers order by name desc;

HAVING Clause:

- The HAVING clause enables you to specify conditions that filter which group results appear in the final results.

- The WHERE clause places conditions on the selected columns, whereas the HAVING clause places conditions on groups created by the GROUP BY clause.

Syntax:

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

GROUP BY column1,
column2 HAVING
[ conditions ] ORDER BY
column1, column2;
```

Ex:

select id, name, age, address, salary from customers group by age having count(age) >= 2;

Aggregate Functions:

- SQL aggregate functions return a single value, calculated from values in a column.

- Useful aggregate functions:

  - **AVG()** - Returns the average value
  - **COUNT()** - Returns the number of rows
  - **MAX()** - Returns the largest value
  - **MIN()** - Returns the smallest value
  - **SUM()** - Returns the sum

AVG () Function

The AVG () function returns the average value of a numeric column.

AVG () Syntax

SELECT AVG (column_name) FROM table_name; Ex:

SELECT AVG (Price) FROM Products;

COUNT () Function

COUNT aggregate function is used to count the number of rows in a database table.

COUNT () Syntax:

SELECT COUNT (column_name) FROM table_name; Ex:

SELECT COUNT (Price) FROM Products**;**

MAX () Function

The SQL MAX aggregate function allows us to select the highest (maximum) value for a certain column.

MAX () Syntax:

SELECT MAX (column_name) FROM table_name; EX:

SELECT MAX (SALARY) FROM EMP;

SQL MIN Function:

SQL MIN function is used to find out the record with minimum value among a record set.

MIN () Syntax:

SELECT MIN (column_name) FROM table_name; EX:

SELECT MIN (SALARY) FROM EMP;

SQL SUM Function SQL:

SUM function is used to find out the sum of a field in various records.

SUM () Syntax:

SELECT COUNT (column_name) FROM
table_name; EX:

SELECT COUNT (EID) FROM EMP;

PRIMARY Key:

- A primary key is a field in a table which uniquely identifies each row/record in a database table.

Properties Primary key:

•A primary keys must contain:

1) Unique values

2) NOT NULL values.

- A table can have only one primary key, which may consist of single or multiple fields.

- If a table has a primary key defined on any field(s), then you cannot have two records having the same value of that field(s).

FOREIGN Key:

- A foreign key is a key used to link two tables together.
- This is sometimes called a referencing key.

- Foreign Key is a column or a combination of columns whose values match a Primary Key in a different table.

- The relationship between 2 tables matches the Primary Key in one of the tables with a Foreign Key in the second table.

Sub-Queries/Nested Queries in SQL: Introduction to Nested Queries :

- One of the most powerful features of SQL is nested queries.

- A nested query is a query that has another query embedded within it; the embedded query is called a sub query.

- When writing a query, we sometimes need to express a condition that refers to a table that must itself be computed.

- A subquery typically appears within the WHERE clause of a query. Subqueries can sometimes appear in the FROM clause or the HAVING clause.

- Subqueries can be used with the SELECT, INSERT, UPDATE, and DELETE statements along with the operators like =, <, >, >=, <=, IN, BETWEEN etc.

- There are a few rules that subqueries must follow:

1. Subqueries must be enclosed within parentheses.

2. A subquery can have only one column in the SELECT clause, unless multiple columns are in the main query for the subquery to compare its selected columns.

3. A subquery cannot be immediately enclosed in a set function.

Subqueries with the SELECT Statement:

Subqueries are most frequently used with the SELECT statement. The basic syntax is as follows:

```
SELECT column_name
[, column_name ] FROM table1
[, table2 ]

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

Ex: select *from customers where id in (select id from customers where salary >4500);

Subqueries with the INSERT Statement:

- Sub queries also can be used with INSERT statements.
- The INSERT statement uses the data returned from the subquery to insert into another table.

- The selected data in the subquery can be modified with any of the character, date or number functions.

Syntax

```
INSERT INTO table_name [ (column1
        [,   column2   ])   ]   SELECT
        [ *|column1 [, column2 ]

        FROM table1 [, table2]
        [ WHERE VALUE OPERATOR ]
```

Ex:

insert into customers_bkp select * from customers where id in (select id from customers) ;

Subqueries with the UPDATE Statement:

- The subquery can be used in conjunction with the UPDATE statement.

- Either single or multiple columns in a table can be updated when using a subquery with the UPDATE statement.

Syntax:

UPDATE table SET column_name = new_value [WHERE OPERATOR [VALUE] (SELECTCOLUMN_NAME FROM TABLE_NAME) [WHERE)];

EX:

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

Transactions:

A transaction is a unit of program execution that accesses and possibly updates various data items.

(or)

A transaction is an execution of a user program and is seen by the DBMS as a series or list of actions i.e., the actions that can be executed by a transaction includes the reading and writing of database.

Transaction Operations:

Access to the database is accomplished in a transaction by the following two operations,
read(X) : Performs the reading operation of data item X from the database. **write(X)** :
Performs the writing operation of data item X to the database.

Example:

Let T1 be a transaction that transfers $50 from account A to account B. This transaction can be illustrated as follows,

```
T1      : read(A);
          A:=A–50;
          write(A);
          read(B);
          B:=B+50;
          write(B);
```

Transaction Concept:

The concept of transaction is the foundation for concurrent execution of transaction in a DBMS and recovery from system failure in a DBMS.

A user writes data access/updates programs in terms of the high-level query language supported by the DBMS.

To understand how the DBMS handles such requests, with respect to concurrency control and recovery, it is convenient to regard an execution of a user program or transaction, as a series of reads and writes of database objects.

To read a database object, it is first brought in to main memory from disk and then its value is copied into a program. This is done by read operation.

To write a database object, in-memory, copy of the object is first modified and then written to disk. This is done by the write operation.

Properties of Transaction (ACID):

There are four important properties of transaction that a DBMS must ensure to maintain data in concurrent access of database and recovery from system failure in DBMS.

The four properties of transactions are,

UNIT - 4

UNIT – 4
Representing Data Elements & Index Structures

Data on External Storage:

Disks: Can retrieve random page at fixed cost

But reading several consecutive pages is much cheaper than reading them in random order

Tapes: Can only read pages in sequence

Cheaper than disks; used for archival storage.

File organization and Indexing:

File organization: Method of arranging a file of records on external storage.

Record id (rid) is sufficient to physically locate record

Indexes are data structures that allow us to find the record ids of records with given values in index search key fields

Architecture: Buffer manager stages pages from external storage to main memory buffer pool. File and index layers make calls to the buffer manager.

Primary and secondary Indexes:

Primary vs. secondary: If search key contains primary key, then called primary index.

Unique index: Search key contains a candidate key.

Clustered and unclustered:

Clustered vs. unclustered: If order of data records is the same as, or `close to', order of data entries, then called clustered index.

Alternative 1 implies clustered; in practice, clustered also implies Alternative 1(since sorted files are rare).

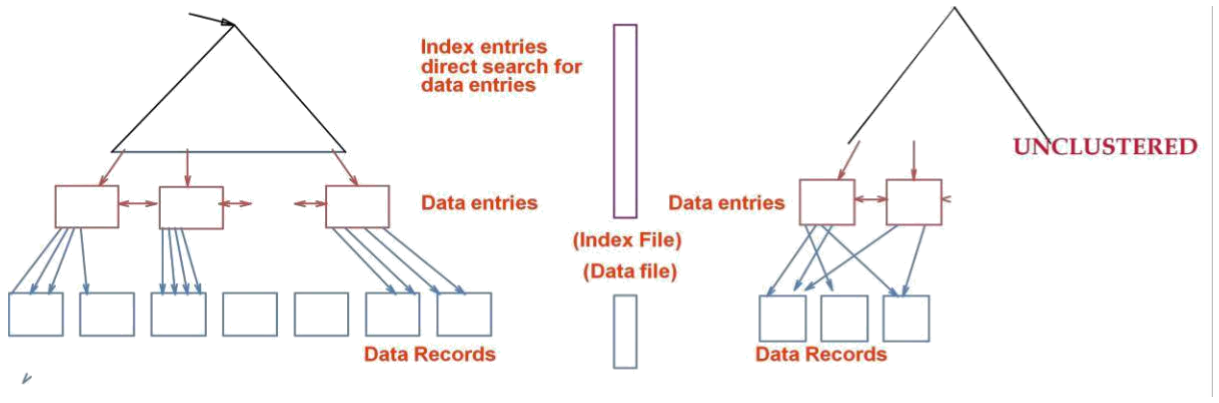
A file can be clustered on at most one search key.

Cost of retrieving data records through index varies *greatly* based on whether index is clustered or not!

Clustered vs. Unclustered Index

Suppose that Alternative (2) is used for data entries, and that the data records are stored in a Heap file.

To build clustered index, first sort the Heap file (with some free space on each page for future inserts).



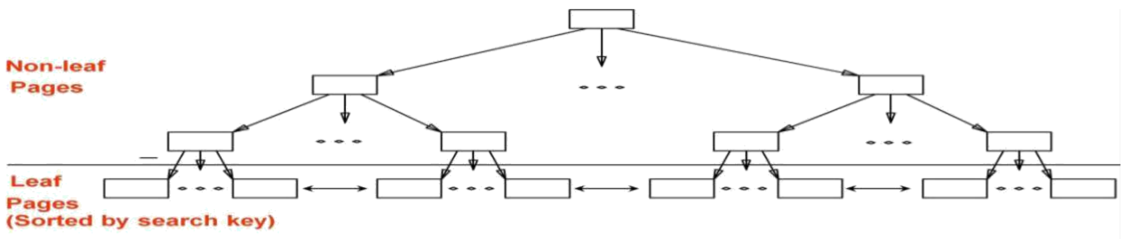
Overflow pages may be needed for inserts. (Thus, order of data recs is `close to', but not identical to, the sort order.)

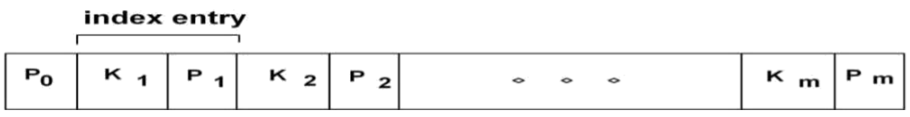
Index Data Structures:

An *index* on a file speeds up selections on the *search key fields* for the index.

Any subset of the fields of a relation can be the search key for an index on the relation.

Search key is not the same as *key* (minimal set of fields that uniquely identify a record in a relation).





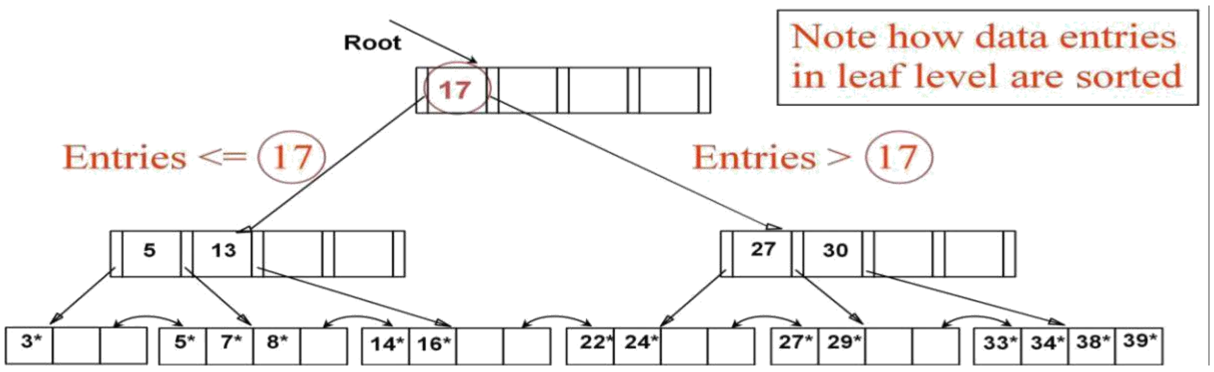
An index contains a collection of *data entries*, and supports efficient retrieval of all data entries **k*** with a given key value **k**.

Given data entry k*, we can find record with key k in at most one disk I/O.

(Details soon …)

B+ Tree Indexes

Example B+ Tree



1. Find 28*? 29*? All > 15* and < 30*

2. Insert/delete: Find data entry in leaf, then change it. Need to adjust parent sometimes.

And change sometimes bubbles up the tree

Hash-Based Indexing:

Hash-Based Indexes

Good for equality selections.

Index is a collection of *buckets*.

Bucket = *primary* page plus zero or more *overflow* pages. Buckets contain data entries.

Hashing function **h**: **h**(*r*) = bucket in which (data entry for) record *r* belongs. **h** looks atthe *search key* fields of *r*.

No need for "index entries" in this scheme.

Data record with key value **k,** or

✓ <**k**, rid of data record with search key value **k**>, or

✓ <**k**, list of rids of data records with search key **k**>

Choice of alternative for data entries is orthogonal to the indexing technique used to

locate data entries with a given key value **k**.

Tree Based Indexing:

– Examples of indexing techniques: B+ trees, hash-based structures

– Typically, index contains auxiliary information that directs searches to the desired data entries

– If this is used, index structure is a file organization for data records (instead of a Heap file or sorted file).

– At most one index on a given collection of data records can use Alternative 1. (Otherwise, data records are duplicated, leading to redundant storage and potential inconsistency.)

– If data records are very large, # of pages containing data entries is high.

Implies size of auxiliary information in the index is also large, typically.

Cost Model for Our Analysis

We ignore CPU costs, for simplicity:

– **B:** The number of data pages

– **R:** Number of records per page

– **D:** (Average) time to read or write disk page

– Measuring number of page I/O's ignores gains of pre-fetching a sequence of pages; thus, even I/O cost is only approximated.

– Average-case analysis; based on several simplistic assumptions.

Choice of Indexes

1. What indexes should we create?

– Which relations should have indexes? What field(s) should be the search key?

Should we build several indexes?

1. For each index, what kind of an index should it be?

Clustered? Hash/tree?

1. One approach: Consider the most important queries in turn. Consider the best plan using the current indexes, and see if a better plan is possible with an additional index.

– Obviously, this implies that we must understand how a DBMS evaluates queries and creates query evaluation plans!

– For now, we discuss simple 1-table queries.

Before creating an index, must also consider the impact on updates in the workload!

– Trade-off: Indexes can make queries go faster, updates slower. Require disk space, too.

Index Selection Guidelines

Attributes in WHERE clause are candidates for index keys.

– Exact match condition suggests hash index.

– Range query suggests tree index.

Clustering is especially useful for range queries; can also help on equality queries if there are many duplicates.

Multi-attribute search keys should be considered when a WHERE clause contains several conditions.

– Order of attributes is important for range queries.

– Such indexes can sometimes enable index-only strategies for important queries.

For index-only strategies, clustering is not important!

<u>B+ Tree:</u>

B+ Tree: Most Widely Used Index. Insert/delete at log $_F$ N cost; keep tree *height-balanced*. (F = fanout, N = # leaf pages) Minimum 50% occupancy (except for root). Each node contains **d** <= <u>*m*</u> <= 2**d** entries. The parameter **d** is called the *order* of the tree. Supports equality and range-searches efficiently.

Example B+ Tree

1. Search begins at root, and key comparisons direct it to a leaf (as in ISAM).

2. Search for 5*, 15*, all data entries >= 24* ...

B+ Trees in Practice

Typical order: 100. Typical fill-factor: 67%.

– average fanout = 133

Typical capacities:

–Height 4:133^4 = 312,900,700 records

– Height 3: 133^3 = 2,352,637 records

Can often hold top levels in buffer pool:

– Level 1 = 1 page = 8 Kbytes

– Level 2 = 133 pages = 1 Mbyte

– Level 3 = 17,689 pages = 133 MBytes

– If *L* has enough space, *done*!

– Else, must *split* *L (into L and a new node L2)*

• Redistribute entries evenly, **copy up** middle key.

• Insert index entry pointing to *L2* into parent of *L*.

This can happen recursively

– To split index node, redistribute entries evenly, but **push up** middle key. (Contrast with leaf splits.) Splits "grow" tree; root split increases height.

– Tree growth: gets *wider* or *one level taller at top.*

Inserting 8* into Example B+ Tree

Observe how minimum occupancy is guaranteed in both leaf and index pg splits.

Note difference between *copy-up* and *push-up*; be sure you understand the reasons for this.

Example B+ Tree After Inserting 8*

1. Deleting a Data Entry from a B+ Tree

2. Start at root, find leaf *L* where entry belongs.

3. Remove the entry.

– If L is at least half-full, *done!*

– If L has only **d-1** entries,

Try to re-distribute, borrowing from *sibling (adjacent node with same parent as L)*.

If re-distribution fails, *merge L* and sibling.

If merge occurred, must delete entry (pointing to *L* or sibling) from parent of *L*. Merge could propagate to root, decreasing height.

Example Tree After (Inserting 8*, Then) Deleting 19* and 20* ...

Deleting 19* is easy.

Deleting 20* is done with re-distribution. Notice how middle key is *copied up*.... And Then Deleting 24*

Must merge.

Observe `*toss*' of index entry (on right), and `*pull down*' of index entry (below).

Hash Based Indexing:

Bucket: Hash file stores data in bucket format. Bucket is considered a unit of storage. Bucket typically stores one complete disk block, which in turn can store one or more records.

Hash Function: A hash function h, is a mapping function that maps all set of search-keys K to the address where actual records are placed. It is a function from search keyto bucket addresses.

UNIT -5

UNIT-5
Coping with System Failures & Concurrency Control

Coping with System Failures

Issues and Models for Resilient Operation

A computer system, like any other mechanical or electrical device is subject to failure. There are many causes of such failure, such as disk crash, power failure, software error, etc. In each of these cases, information may be lost. Therefore, the database system maintains an integral part known as recovery manager. It is responsible for the restore of the database to a consistent state that existed prior to the occurrence of the failures.

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 and media failures.

When a DBMNS 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.

System Failure classifications:

1) Transaction Failure:

There are two types of errors that may cause a transaction failure.

i) **Logical Error:** The transaction can do longer continue with its normal execution with some internal conditions such as bad input, data not found, overflow or resource limits exceeded.

ii) **System Error:** The system has entered an undesirable state (deadlock) as a result of which a transaction cannot continue with its normal execution. This transaction can be reexecuted at a later time.

2) System Crash:

There is a hardware failure or an error in the database software or the operating system that causes the loss of the content of temporary storage and brings transaction processing to a halt. The content of permanent storage remains same and is not corrupted.

3) Disk failure:

A disk block loses its content as a result of either a head crash or failure during a data transfer operation. Copies of the data on other disks or backups on tapes are used to recover from the failure.

Causes of failures:

Some failures might cause the database to go down, some others might be trivial. On the other hand, if a data file has been lost, recovery requires additional steps. Some common causes of failures include:

*1) **System Crashes:***

It can be happen due to hardware or software errors resulting in loss of main memory.

*2) **User error:***

It can be happen due to a user inadvertently deleting a row or dropping a table.

*3) **Carelessness:***

It can be happen due to the destruction of data or facilities by operators/users because of lack of concentration.

*4) **Sabotage:***

It can be happen due to the intentional corruption or destruction of data, hardware or software facilities.

*5) **Statement failure:***

It can be happen due to the inability by the database to execute an SQL statement.

*6) **Application software errors:***

It can be happen due to the logical errors in the program to access the database, which causes one or more transactions to fail.

*7) **Network failure:***

It can be happen due to a network failure / communication software failure / aborted asynchronous connections.

*8) **Media failure:***

It can be happen due to the disk controller failure / disk head crash / disk to be lost. It is ht most dangerous failure.

*9) **Natural physical disasters**:*

It can be happen due to the natural disasters like fires, floods, earthquakes, power failure, etc.

Undo Logging

Logging is a way to assure that transactions are atomic. They appear to the database either to have executed in their entirety or not to have executed at all. A ***log*** is a sequence of ***log records***, each telling something about what some transaction has done. The actions of several transactions can "interleave," so that a step of one transaction may be executed and its effect logged, then the same happens for a step of another transaction, then for a second step of the first transaction or a step of a third transaction, and so on. This interleaving of transactions complicates logging; it is not sufficient simply to log the entire story of a transaction after that transaction completes.

If there is a system crash, the log is consulted to reconstruct what transactions were doing when the crash occurred. The log also may be used, in conjunction with an archive, if there is a media failure of a disk that does not store the log. Generally, to repair the effect of the crash, some transactions will have their work done again, and the new values they wrote into the database are written again. Other transactions will have their work undone, and the database restored so that it appears that they never executed.

The first style of logging, which is called **undo logging**, makes only repairs of the second type. If it is not absolutely certain that the effects of a transaction have been completed and stored on disk, then any database changes that the transaction may have made to the database are undone, and the database state is restored to what existed prior to the transaction.

Log Records

The log is a file opened for appending only. As transactions execute, the log manager has the job of recording in the log each important event. One block of the log at a time is filled with log records, each representing one of these events. Log blocks are initially created in main memory and are allocated by the buffer manager like any other blocks that the DBMS needs. The log blocks are written to nonvolatile storage on disk as soon as is feasible.
There are several forms of log record that are used with each of the types of logging. These are:

1. <START T> : This record indicates that transaction T has begun.

2. <COMMIT T>: Transaction T has completed successfully and will make no more changes to database elements. Any changes to the database made by T should appear on disk. If we insist that the changes already be on disk, this requirement must be enforced by the log manager.

3. <ABORT T> : Transaction T could not complete successfully. If transaction T aborts, no changes it made can have been copied to disk, and it is the job of the transaction manager to make sure that such changes never appear on disk, or that their effect on disk is cancelled if they do.

The Undo-Logging Rules

There are two rules that transactions must obey in order that an undo log allows us to recover from a system failure. These rules affect what the buffer manager can do and also require that certain actions be taken whenever a transaction commits.

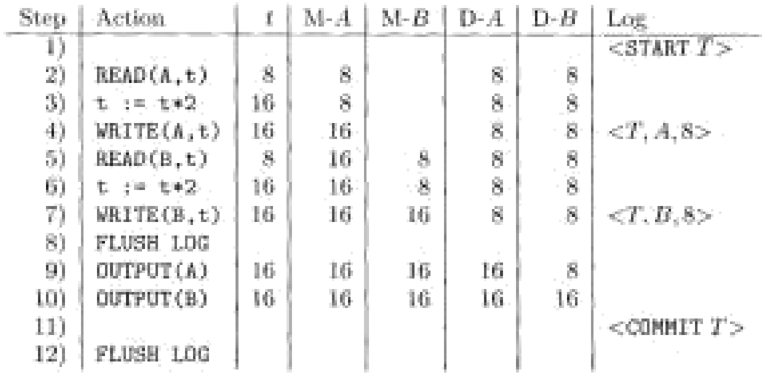
U_1 : If transaction *T* modifies database element *X*, then the log record of the form *<T, X,v>* must be written to disk *before* the new value of *X* is written to disk.

U_2 : If a transaction commits, then its COMMIT log record must be written to disk only *after* all database elements changed by the transaction have been written to disk, but as soon thereafter as possible.

To summarize rules U1 and U2, material associated with one transaction must be written to disk in the following order:

a) The log records indicating changed database elements.
b) The changed database elements themselves.
c) The COMMIT log record.

In order to force log records to disk, the log manager needs a flush-log command that tells the buffer manager to copy to disk any log blocks that have not previously been copied to disk or that have been changed since they were last copied. Example:



Step	Action	t	M-A	M-B	D-A	D-B	Log
1)							<START T>
2)	READ(A,t)	8	8		8	8	
3)	t := t*2	16	8		8	8	
4)	WRITE(A,t)	16	16		8	8	<T, A, 8>
5)	READ(B,t)	8	16	8	8	8	
6)	t := t*2	16	16	8	8	8	
7)	WRITE(B,t)	16	16	16	8	8	<T, B, 8>
8)	FLUSH LOG						
9)	OUTPUT(A)	16	16	16	16	8	
10)	OUTPUT(B)	16	16	16	16	16	
11)							<COMMIT T>
12)	FLUSH LOG						

Actions and their log entries

The transaction of undo logging to show the log entries and flushlog actions that have to take place along with the actions of the transaction *T*.

Recovery Using Undo Logging

It is the job of the **recovery manager** to use the log to restore the database state to some consistent state. The first task of the recovery manager is to divide the transactions into committed and uncommitted transactions.

If there is a log record <COMMIT T>, then by undo rule f/2 all changes made by transaction T were previously written to disk. Thus, T by itself could not have left the database in an inconsistent state when the system failure occurred.

If there is a log <START T> record on the log but no <COMMIT T> record. Then there could have been some changes to the database made by T that got written to disk before the crash, while other changes by T either were not made, even in the main-memory buffers, or were made in the buffers but not copied to disk.

After making all the changes, the recovery manager must write a log record < ABORT T> for each incomplete transaction T that was not previously aborted, and then flush the log. Now, normal operation of the database may resume, and new transactions may begin executing.

Redo Logging

While undo logging provides a natural and simple strategy for maintaining a log and recovering from a system failure, it is not the only possible approach.

The requirement for immediate backup of database elements to disk can be avoided if we use a logging mechanism called *redo logging.*

The principal *differences* between *redo* and *undo logging* are:

1. While undo logging cancels the effect of incomplete transactions and ignores committed ones during recovery, redo logging ignores incomplete transactions and repeats the changes made by committed transactions

2. While undo logging requires us to write changed database elements to disk before the COMMIT log record reaches disk, redo logging requires that the COMMIT record appear on disk before any changed values reach disk.

3. While the old values of changed database elements are exactly what we need to recover when the undo rules U1 and U2 are followed, to recover using redo logging, we need the new values.

The Redo-Logging Rule

Redo logging represents changes to database elements by a log record that gives the new value, rather than the old value, which undo logging uses. These records look the same as for undo logging: *<T, X, v>*. The difference is that the meaning of this record is "transaction *T* wrote new value *v* for database element *X."*

There is no indication of the old value of *X* in this record. Every time a transaction *T* modifies a database element *X*, a record of the form <*T*, *X*, *v*> must be written to the log. The order in which data and log entries reach disk can be described by a single "redo rule," called the *write-ahead logging rule.*

R1 : Before modifying any database element *X* on disk, it is necessary that all log records pertaining to this modification of *X*, including both the update record <*T*, *X*, *v*> and the <COMMIT *T*> record, must appear on disk.

The order of redo logging associated with one transaction gets written to disk is:

1. The log records indicating changed database elements.
2. The COMMIT log record.
3. The changed database elements themselves.

Step	Action	t	M-A	M-B	D-A	D-B	Log
1)							< START T>
2)	READ(A,t)	8	8		8	8	
3)	t : = t * 2	16	8		8	8	
4)	WRITE(A,t)	16	16		8	8	<T,A, 16>
5)	READ(B,t)	8	16	8	8	8	
6)	t : = t * 2	16	16	8	8	8	
7)	WRITE(B,t)	16	16	16	8	8	<T,B.IQ>
8)							<COMMIT T>
9)	FLUSH LOG						
10)	OUTPUT (A)	16	16	16	16	8	
11)	OUTPUT(B)	16	16	16	16	16	

Actions and their log entries using redo logging

Recovery with Redo Logging:

An important consequence of the redo lule *RI* is that unless the log has a <COMMIT T> record, we know that no changes to the database made by transaction T have been written to disk.

To recover, using a redo log after a system crash, we do the following:

1. Identify the committed transactions.
2. Scan the log forward from the beginning. For each log record <T,X,v> encountered:

(a) If T is not a committed transaction, do nothing.

(b) If T is committed, write value v for database element X.

3. For each incomplete transaction T, write an <ABORT T> record to the log and flush the log.

The steps to be taken to perform a nonquiescent checkpoint of a redo log are as follows:

1. Write a log record <START CKPT (T},..., T_k)>, where T_1,...,T_k are all the active (uncommitted) transactions, and flush the log.

2. Write to disk all database elements that were written to buffers but not yet to disk by transactions that had already committed when the START CKPT record was written to the log.

3. Write an <END CKPT> record to the log and flush the log.

```
<STAR T1>
<T1,A,5>
<START T2>
<COMMIT T1>
<T2,5,10>
<START CKPT (T2)>
<T2,c7,15>
<START T3>
<T3,D,20>
<END CKPT>
<COMMIT T2>
<COMMIT T3>
```

Recovery with a Check pointed Redo Log:

As for an undo log, the insertion of records to mark the start and end of a checkpoint helps us limit our examination of the log when a recovery is necessary. Also as with undo logging, there are two cases, depending on whether the last checkpoint record is START or END.

i) Suppose first that the last checkpoint record on the log before a crash is <END CKPT>. <START CKPT (T_1,..., *Tk)*> or that started after that log record appeared in the log. In searching the log, we do not look further back than the earliest of the < START T,> records. Linking backwards all the log records for a given transaction helps us to find the necessary records, as it did for undo logging.

ii) The last checkpoint record on the log is a <START CKPT *(T_1,... ,Tk)*> record. We must search back to the previous <END CKPT> record, find its matching <START CKPT *(S_i,..., Srn)*> record, and redo all those committed transactions that either started after that START CKPT or are among the *S_1's*.

Undo/Redo Logging

We have two different approaches to logging, differentiated by whether the log holds old values or new values when a database element is updated. Each has certain drawbacks:

i) Undo logging requires that data be written to disk immediately after a transaction finishes.

ii) Redo logging requires us to keep all modified blocks in buffers until the transaction commits and the log records have been flushed.

iii) Both undo and redo logs may put contradictory requirements on how buffers are handled during a checkpoint, unless the database elements are complete blocks or sets of blocks.

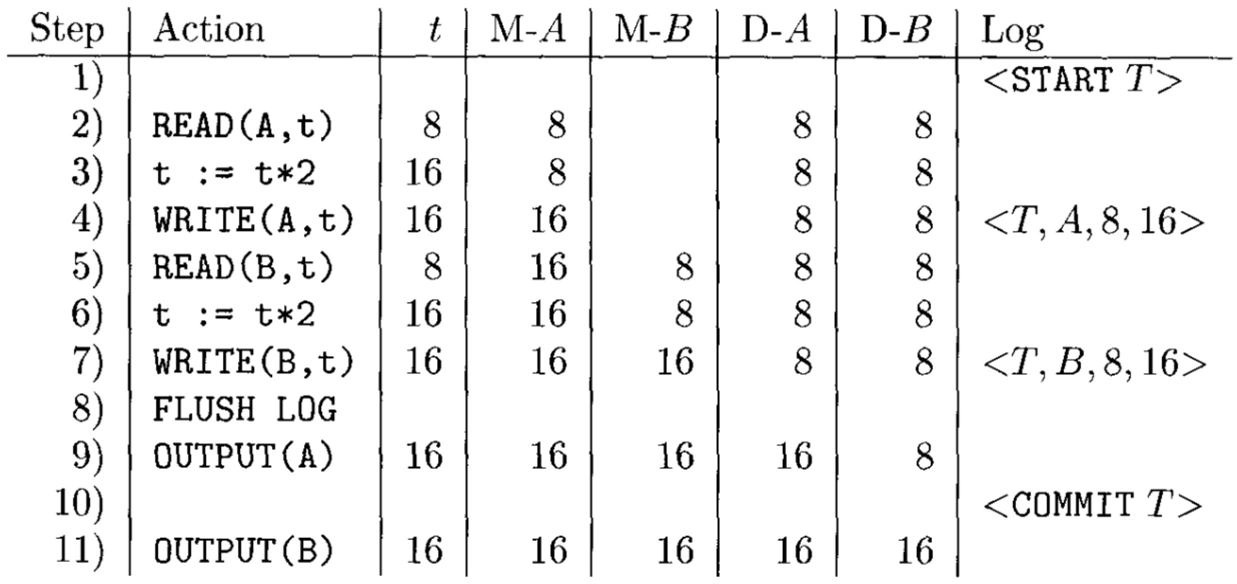
To overcome these drawbacks we have a kind of logging called ***undo/redo logging***, that provides increased flexibility to order actions, at the expense of maintaining more information on the log.

The Undo/Redo Rules:

An undo/redo log has the same sorts of log records as the other kinds of log, with one exception. The update log record that we write when a database element changes value has four components. Record <T, X, v,w> means that transaction T changed the value of database element X; its former value was v, and its new value is w. The constraints that an undo/redo logging system must follow are summarized by the following rule:

UR_1 : Before modifying any database element X on disk because of changes made by some transaction T, it is necessary that the update record <T,X,v,w> appear on disk.

Rule UR_1 for undo/redo logging thus enforces only the constraints enforced by both undo logging and redo logging. In particular, the <COMMIT T> log record can precede or follow any of the changes to the database elements on disk.



Step	Action	t	M-A	M-B	D-A	D-B	Log
1)							<START T>
2)	READ(A,t)	8	8		8	8	
3)	t := t*2	16	8		8	8	
4)	WRITE(A,t)	16	16		8	8	$<T, A, 8, 16>$
5)	READ(B,t)	8	16	8	8	8	
6)	t := t*2	16	16	8	8	8	
7)	WRITE(B,t)	16	16	16	8	8	$<T, B, 8, 16>$
8)	FLUSH LOG						
9)	OUTPUT(A)	16	16	16	16	8	
10)							<COMMIT T>
11)	OUTPUT(B)	16	16	16	16	16	

A possible sequence of actions and their log entries using undo/redo logging.

Recovery with Undo/Redo Logging:

When we need to recover using an undo/redo log, we have the information in the update records either to undo a transaction T, by restoring the old values of the database elements that T changed, or to redo T by repeating the changes it has made. The undo/redo recovery policy is:

1. Redo all the committed transactions in the order earliest-first, and
2. Undo all the incomplete transactions in the order latest-first.

It is necessary for us to do both. Because of the flexibility allowed by undo/redo logging regarding the relative order in which COMMIT log records and the database changes themselves are copied to disk, we could have either a committed transaction with some or all of its changes not on disk, or an uncommitted transaction with some or all of its changes on disk.

Check pointing an Undo/Redo Log:

A nonquiescent checkpoint is somewhat simpler for undo/redo logging than for the other logging methods. We have only to do the following:

1. Write a <START CKPT *(T_1,...,Tk)*> record to the log, where *T_1...,Tk* are all the active transactions, and flush the log.

2. Write to disk all the buffers that are *dirty;* i.e., they contain one or more changed database elements. Unlike redo logging, we flush all buffers, not just those written by committed transactions.

3. Write an <END CKPT> record to the log, and flush the log.

```
<START T1>
<T1, 4, 4, 5>
< START T2>
<COMMIT Ti > <T2,
B, 9, 10> <START
CKPT (T2)> <T2, C,
14, 15>
< START T3>
<T3, D, 19, 20>
<END CKPT>
<COMMIT T2>
<COMMIT T3>
```

An undo/redo log

Suppose the crash occurs just before the <COMMIT T_3> record is written to disk. Then we identify T2 as committed but T_3 as incomplete. We redo T2 by setting *C* to 15 on disk; it is not necessary to set *B* to 10 since we know that change reached disk before the <END CKPT>.

However, unlike the situation with a redo log, we also undo T_3*;* that is, we set *D* to 19 on disk. If T3 had been active at the start of the checkpoint, we would have had to look prior to the START-CKPT record to find if there were more actions by T_3 that may have reached disk and need to be undone.

Protecting Against Media Failures

The log can protect us against system failures, where nothing is lost from disk, but temporary data in main memory is lost. The serious failures involve the loss of one or more disks. We can reconstruct the database from the log if:

a) The log were on a disk other than the disk(s) that hold the data,

b) The log were never thrown away after a checkpoint, and

c) The log were of the redo or the undo/redo type, so new values are stored on the log. The

log will usually grow faster than the database. So it is not practical to keep the log forever.

The Archive:

To protect against media failures, we are thus led to a solution involving *archiving* — maintaining a copy of the database separate from the database itself. If it were possible to shut down the database for a while, we could make a backup copy on some storage medium such as tape or optical disk, and store them remote from the database in some secure location.

The backup would preserve the database state as it existed at this time, and if there were a media failure, the database could be restored to the state that existed then.

Since writing an archive is a lengthy process if the database is large, one generally tries to avoid copying the entire database at each archiving step. Thus, we distinguish between two levels of archiving:

1. A *full dump*, in which the entire database is copied.

2. An *incremental dump*, in which only those database elements changed, the previous full of incremental dump is copied.

It is also possible to have several levels of dump, with a full dump thought of as a "level 0" dump, and a "level *i*" dump copying everything changed since the last dump at level *i* or less.

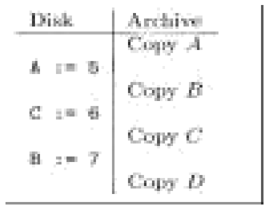
We can restore the database from a full dump and its subsequent incremental dumps, in a process much like the way a redo or undo/redo log can be used to repair damage due to a system failure. We copy the full dump back to the database, and then in an earliest-first order, make the changes recorded by the later incremental dumps. Since incremental dumps will tend to involve only a small fraction of the data changed since the last dump, they take less space and can be done faster than full dumps.

Nonquiescent Archiving:

A nonquiescent checkpoint attempts to make a copy on the disk of the (approximate) database state that existed when the checkpoint started.

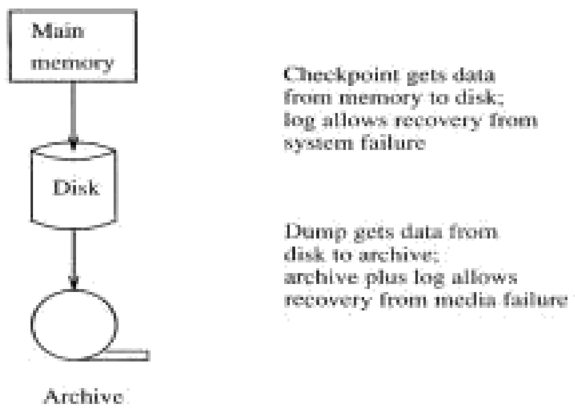
A nonquiescent dump tries to make a copy of the database that existed when the dump began, but database activity may change many database elements on disk during the minutes or hours that the dump takes. If it is necessary to restore the database from the archive, the log entries made during the dump can be used to sort things out and get the database to a consistent state.

A nonquiescent dump copies the database elements in some fixed order, possibly while those elements are being changed by executing transactions. As a result, the value of a database element that is copied to the archive may or may not be the value that existed when the dump began. As long as the log for the duration of the dump is preserved, the discrepancies can be corrected from the log.



Disk	Archive
	Copy A
A := 5	
	Copy B
C := 6	
	Copy C
B := 7	
	Copy D

Events during a nonquiescent dump


Main memory
Disk
Archive
Checkpoint gets data from memory to disk; log allows recovery from system failure
Dump gets data from disk to archive; archive plus log allows recovery from media failure

The analogy between checkpoints and dumps

The process of making an archive can be broken into the following steps. We assume that the logging method is either redo or undo/redo; an undo log is not suitable for use with archiving.

1. Write a log record < START DUMP>.
2. Perform a checkpoint appropriate for whichever logging method is being used.
3. Perform a full or incremental dump of the data disk(s), as desired; making sure that the copy of the data has reached the secure, remote site.

4. Make sure that enough of the log has been copied to the secure, remote site that at least the prefix of the log up to and including the checkpoint in item (2) will survive a

media failure of the database.

5. Write a log record <END DUMP>.

At the completion of the dump, it is safe to throw away log prior to the beginning of the checkpoint previous to the one performed in item (2) above.

```
<START DUMP>
<START CKPT (T1, T2)>
<T1, A, l, 5>
<T2, C, 3, 6>
<COMMIT T2>
<T1, B, 2, 7>
<END CKPT>
Dump completes
<END DUMP>
```

Log taken during a dump

Note that we did not show T_1 committing. It would be unusual that a transaction remained active during the entire time a full dump was in progress, but that possibility doesn't affect the correctness of the recovery method.

Recovery Using an Archive and Log:

Suppose that a media failure occurs, and we must reconstruct the database from the most recent archive and whatever prefix of the log has reached the remote site and has not been lost in the crash. We perform the following steps:

1. Restore the database from the archive.
   (a) Find the most recent full dump and reconstruct the database from it (i.e., copy the archive into the database).
   (b) If there are later incremental dumps, modify the database according to each, earliest first.
2. Modify the database using the surviving log. Use the method of recovery appropriate to the log method being used.

PART – A [2 mark questions]

1) Transaction Management: The two principal tasks of the transaction manager are assuring recoverability of database actions through logging, and assuring correct, concurrent behavior of transactions through the scheduler (not discussed in this chapter).

2) ***Database Elements:*** The database is divided into elements, which are typically disk blocks, but could be tuples, extents of a class, or many other units. Database elements are the units for both logging and scheduling.

3) Logging: A record of every important action of a transaction — beginning, changing a database element, committing, or aborting — is stored on a log. The log must be backed up on disk at a time that is related to when the corresponding database changes migrate to disk, but that time depends on the particular logging method used.

4) ***Recovery:*** When a system crash occurs, the log is used to repair the database, restoring it to a consistent state.

5) ***Logging Methods:*** The three principal methods for logging are undo, redo, and undo/redo, named for the way(s) that they are allowed to fix the database during recovery.

6) Undo Logging : This method logs only the old value, each time a database element is changed. With undo logging, a new value of a database element can only be written to disk after the log record for the change has reached disk, but before the commit record for the transaction performing the change reaches disk. Recovery is done by restoring the old value for every uncommitted transaction.

7) Redo Logging : Here, only the new value of database elements is logged. With this form of logging, values of a database element can only be written to disk after both the log record of its change and the commit record for its transaction have reached disk. Recovery involves rewriting the new value for every committed transaction.

8) Undo/Redo Logging: In this method, both old and new values are logged. Undo/redo logging is more flexible than the other methods, since it requires only that the log record of a change appear on the disk before the change itself does. There is no requirement about when the commit record appears. Recovery is effected by redoing committed transactions and undoing the uncommitted transactions.

9) Check pointing: Since all methods require, in principle, looking at the entire log from the dawn of history when a recovery is necessary, the DBMS must occasionally checkpoint the log, to assure that no log records prior to the checkpoint will be needed during a recovery. Thus, old log records can eventually be thrown away and its disk space reused.

10) Nonquiescent Check pointing : To avoid shutting down the system while a checkpoint is made, techniques associated with each logging method allow the checkpoint to be made while the system is in operation and database changes are occurring. The only cost is that some log records prior to the nonquiescent checkpoint may need to be examined during recovery.

11) Archiving: While logging protects against system failures involving only the loss of main memory, archiving is necessary to protect against failures where the contents of disk are lost. Archives are copies of the database stored in a safe place.

12) Recovery from Media Failures: When a disk is lost, it may be restored by starting with a full backup of the database, modifying it according to any later incremental backups, and finally recovering to a consistent database state by using an archived copy of the log.

13) Incremental Backups : Instead of copying the entire database to an archive periodically, a single complete backup can be followed by several incremental backups, where only the changed data is copied to the archive.

14) Nonqmescent Archiving : Techniques for making a backup of the data while the database is in operation exist. They involve making log records of the beginning and end of the archiving, as well as performing a checkpoint for the log during the archiving.

Serializability

Serializability is a widely accepted standard that ensures the consistency of a schedule. A schedule is consistent if and only if it is serializable. A schedule is said to be serializable if the interleaved transactions produces the result, which is equivalent to the result produced by executing individual transactions separately.

Example:

Transaction T_1	**Transaction T_2**		**Transaction T_1**	**Transaction T_2**
read(X)			read(X)	
write(X)			write(X)	
read(Y)				read(X)
write(Y)				write(X)
	read(X)		read(Y)	
	write(X)		write(Y)	
	read(Y)			read(Y)
	write(Y)			write(Y)

Serial Schedule ***Two interleaved transaction Schedule***

The above two schedules produce the same result, these schedules are said to be serializable. The transaction may be interleaved in any order and DBMS doesn't provide any guarantee about the order in which they are executed.

There two different types of Serializability. They are,

i) Conflict Serializability
ii) View Serializability

i) Conflict Serializability:

Consider a schedule S_1, consisting of two successive instructions I_A and I_B belonging to transactions T_A and T_B refer to different data items then it is very easy to swap these instructions.

The result of swapping these instructions doesn't have any impact on the remaining instructions in the schedule. If I_a and I_B refers to same data item then the following four cases must be considered,

Case 1	:	I_A = read(x), I_B = read(x),
Case 2	:	I_A = read(x), I_B = write(x),
Case 3	:	I_A = write(x), I_B = read(x),
Case 4	:	I_A = write(x), I_B = write(x),

Case 1 : Here, both I_A and I_B are read instructions. In this case, the execution order of the instructions is not considered since the same data item x is read by both the transactions T_A and T_B.

Case 2 : Here, I_A and I_B are read and write instructions respectively. If the execution order of instructions is $I_A \rightarrow I_B$, then transaction T_A cannot read the value written by transaction T_B in instruction I_B. but order is $I_B \rightarrow I_A$, then transaction T_A can read the value written by transaction T_B. Therefore in this case, the execution order of the instructions is important.

Case 3 : Here, I_A and I_B are write and read instructions respectively. If the execution order of instructions is $I_A \rightarrow I_B$, then transaction T_B can read the value written by transaction T_A, but order is $I_B \rightarrow I_A$, then transaction T_B cannot read the value written by transaction T_B. Therefore in this case, the execution order of the instructions is important.

Case 1 : Here, both I_A and I_B are write instructions. In this case, the execution order of the instructions doesn't matter. If a read operation is performed before the write operation, then the data item which was already stored in the database is read.

ii) View Serializability:

Two schedules S_1 and S_1' consisting of some set of transactions are said to be view equivalent, if the following conditions are satisfied,

1) If a transaction T_A in schedule S_1 performs the read operation on the initial value of data item x, then the same transaction in schedule S_1' must also perform the read operation on the initial value of x.

2) If a transaction T_A in schedule S_1 reads the value x, which was written by transaction T_B, then T_A in schedule S_1**'** must also perform the read the value x written by transaction T_B.

3) If a transaction T_A in schedule S_1 performs the final write operation on data item x, then the same transaction in schedule S_1**'** must also perform the final write operation on x.

Example:

Transaction T_1	Transaction T_2
read(x)	
x := x -10	
write(x)	
	read(x)
	x := x *10
	write(x)
read(y)	
y := y -10	
write(y)	
	read(y)
	y := y / 10
	write(y)

View Serializability Schedule S_1

The view equivalence leads to another notion called view serializability. A schedule say S is said to be view Serializable, if it is view equivalent with the serial schedule.

Every conflict Serializable schedule is view Serializable but every view Serializable is not conflict Serializable.

Concurrency Control:

In a multiprogramming environment where multiple transactions can be executed simultaneously, it is highly important to control the concurrency of transactions.

We have concurrency control protocols to ensure atomicity, isolation, and serializability of concurrent transactions.

Why DBMS needs a concurrency control?

In general, concurrency control is an essential part of TM. It is a mechanism for correctness when two or more database transactions that access the same data or data set are executed concurrently with time overlap. According to Wikipedia.org, if multiple transactions are executed serially or sequentially, data is consistent in a database. However, if concurrent transactions with interleaving operations are executed, some unexpected data and inconsistent result may occur. Data interference is usually caused by a write operation among transactions on the same set of data in DBMS. For example, the lost update problem may occur when a second transaction writes a second value of data content on top of the first value written by a first concurrent transaction. Other problems such as the dirty read problem, the incorrect summary problem

Concurrency Control Techniques:

The following techniques are the various concurrency control techniques. They are:

1. concurrency control by Locks
2. Concurrency Control by Timestamps
3. Concurrency Control by Validation

1. Concurrency control by Locks

A lock is nothing but a mechanism that tells the DBMS whether a particular data item is being used by any transaction for read/write purpose.

There are two types of operations, i.e. read and write, whose basic nature are different, the locks for read and write operation may behave differently.

The simple rule for locking can be derived from here. If a transaction is reading the content of a sharable data item, then any number of other processes can be allowed to read the content of the same data item. But if any transaction is writing into a sharable data item, then no other transaction will be allowed to read or write that same data item.

Depending upon the rules we have found, we can classify the locks into two types.

Shared Lock: A transaction may acquire shared lock on a data item in order to read its content. The lock is shared in the sense that any other transaction can acquire the shared lock on that same data item for reading purpose.

Exclusive Lock: A transaction may acquire exclusive lock on a data item in order to both read/write into it. The lock is excusive in the sense that no other transaction can acquire any kind of lock (either shared or exclusive) on that same data item.

The relationship between Shared and Exclusive Lock can be represented by the following table which is known as **Lock Matrix**.

	Shared	Exclusive
Shared	TRUE	FALSE
Exclusive	FALSE	FALSE

Two Phase Locking Protocol

The use of locks has helped us to create neat and clean concurrent schedule. The Two Phase Locking Protocol defines the rules of how to acquire the locks on a data item and how to release the locks.

The Two Phase Locking Protocol assumes that a transaction can only be in one of two phases.

Growing Phase:

- In this phase the transaction can only acquire locks, but cannot release any lock
- The transaction enters the growing phase as soon as it acquires the first lock it wants.
- It cannot release any lock at this phase even if it has finished working with a locked data item.
- Ultimately the transaction reaches a point where all the lock it may need has been acquired. This point is called **Lock Point**.

Shrinking Phase:

- After Lock Point has been reached, the transaction enters the shrinking phase. In this phase the transaction can only release locks, but cannot acquire any new lock.
- The transaction enters the shrinking phase as soon as it releases the first lock after crossing the Lock Point.

Two Phase Locking Protocol:

- There are two different versions of the Two Phase Locking Protocol. They are:

1. Strict Two Phase Locking Protocol
2. Rigorous Two Phase Locking Protocol

- In this protocol, a transaction may release all the shared locks after the Lock Point has been reached, but it cannot release any of the exclusive locks until the transaction commits. This protocol helps in creating cascade less schedule.

A **Cascading Schedule** is a typical problem faced while creating concurrent schedule. Consider the following schedule once again.

T1	T2
Lock-X (A)	
Read A;	
A=A-100;	
Write A;	
Unlock (A)	
	Lock-S (A)
	Read A;
	Temp = A * 0.1;
	Unlock (A)
	Lock-X(C)
	Read C;
	C = C + Temp;
	Write C;
	Unlock (C)
Lock-X (B)	
Read B;	
B=B+100;	
Write B;	
Unlock (B)	

The schedule is theoretically correct, but a very strange kind of problem may arise here.

T1 releases the exclusive lock on A, and immediately after that the Context Switch is made.

T2 acquires a shared lock on A to read its value, perform a calculation, update the content of account C and then issue COMMIT. However, T1 is not finished yet. What if the remaining portion of T1 encounters a problem (power failure, disc failure etc) and cannot be committed?

In that case T1 should be rolled back and the old BFIM value of A should be restored. In such a case T2, which has read the updated (but not committed) value of A and calculated the value of C based on this value, must also have to be rolled back.

We have to rollback T2 for no fault of T2 itself, but because we proceeded with T2 depending on a value which has not yet been committed. This phenomenon of rolling back a child transaction if the parent transaction is rolled back is called Cascading Rollback, which causes a tremendous loss of processing power and execution time.

Using Strict Two Phase Locking Protocol, Cascading Rollback can be prevented.

In Strict Two Phase Locking Protocol a transaction cannot release any of its acquired exclusive locks until the transaction commits.

In such a case, T1 would not release the exclusive lock on A until it finally commits, which makes it impossible for T2 to acquire the shared lock on A at a time when A's value has not been committed. This makes it impossible for a schedule to be cascading.

Rigorous Two Phase Locking Protocol

In Rigorous Two Phase Locking Protocol, a transaction is not allowed to release any lock (either shared or exclusive) until it commits. This means that until the transaction commits, other transaction might acquire a shared lock on a data item on which the uncommitted transaction has a shared lock; but cannot acquire any lock on a data item on which the uncommitted transaction has an exclusive lock.

2. Concurrency Control by Timestamps

Timestamp ordering technique is a method that determines the serializability order of different transactions in a schedule. This can be determined by having prior knowledge about the order in which the transactions are executed.

Timestamp denoted by $T_S(T_A)$ is an identifier that specifies the start time of transaction and is generated by DBMS. It uniquely identifies the transaction in a schedule. The timestamp of older transaction (T_A) is less than the timestamp of a newly entered transaction (T_B) i.e., $T_S(T_A) < T_S(T_B)$.

In timestamp-based concurrency control method, transactions are executed based on priorities that are assigned based on their age. If an instruction I_A of transaction T_A conflicts with an instruction I_B of transaction T_B then it can be said that I_A is executed before I_B if and only if $T_S(T_A) < T_S(T_B)$ which implies that older transactions have higher priority in case of conflicts.

Ways of generating Timestamps:
Timestamps can be generated by using,

i) System Clock : When a transaction enters the system, then it is assigned a timestamp which is equal to the time in the system clock.
ii) Logical Counter: When a transaction enters the system, then it is assigned a timestamp which is equal to the counter value that is incremented each time for a newly entered transaction.
Every individual data item x consists of the following two timestamp values,

i) WTS(x) (W-Timestamp(x)) : It represents the highest timestamp value of the transaction that successfully executed the *write* instruction on x.

ii) RTS(x) (R-Timestamp(x)) : It represents the highest timestamp value of the transaction that successfully executed the *read* instruction on x.

Timestamp Ordering Protocol
This protocol guarantees that the execution of read and write operations that are conflicting is done in timestamp order.

Working of Timestamp Ordering Protocol:

The Time stamp ordering protocol ensures that any conflicting read and write operations are executed in time stamp order. This protocol operates as follows:

1) **If TA executes read(x) instruction**, then the following two cases must be considered,

i) $T_S(T_A) < WTS(x)$
ii) $T_S(T_A)$ □ WTS(x)

Case 1 : If a transaction T_A wants to read the initial value of some data item x that had been overwritten by some younger transaction then, the transaction T_A cannot perform the read operation and therefore the transaction must be rejected. Then the transaction T_A must be rolled back and restarted with a new timestamp.

Case 2 : If a transaction T_A wants to read the initial value of some data item x that had not been updated then the transaction can execute the read operation. Once the value has been read, changes occur in the read timestamp value (RTS(x)) which is set to the largest value of RTS(x) and T_S

2) **If TA executes write(x) instruction**, then the following two cases must be considered,

i) $T_S(T_A) < RTS(x)$

ii) $T_S(T_A) < WTS(x)$

iii) $T_S(T_A) > WTS(x)$

Case 1 : If a transaction T_A wants to write the value of some data item x on which the read operation has been performed by some younger transaction, then the transaction cannot execute the write operation. This is because the value of data item x that is being generated by T_A was required previously and therefore, the system assumes that the value will never be generated. The write operation is thereby rejected and the transaction T_A must be rolled back and should be restarted with new timestamp value.

Case 2 : If a transaction T_A wants to write a new value to some data item x, that was overwritten by some younger transaction, then the transaction cannot execute the write operation as it may lead to inconsistency of data item. Therefore, the write operation is rejected and the transaction should be rolled back with a new timestamp value.

Case 3 : If a transaction T_A wants to write a new value on some data item x that was not updated by a younger transaction, then the transaction can executed the write operation. Once the value has been written, changes occur on WTS(x) value which is set to the value of $T_S(T_A)$.

Example:

T_1	**T_2**
read(y)	
	read(y)
	y:= y + 100
	write(y)
read(x)	
	read(x)
show(x+y)	
	x:= x – 100
	write(x)
	show(x+y)

The above schedule can be executed under the timestamp protocol when $T_S(T_1) < T_S(T_2)$.

3. Concurrency Control by Validation

- Validation techniques are also called as Optimistic techniques.

- If read only transactions are executed without employing any of the concurrency control mechanisms, then the result generated is in inconsistent state.

- However if concurrency control schemes are used then the execution of transactions may be delayed and overhead may be resulted. To avoid such issue, optimistic concurrency control mechanism is used that reduces the execution overhead.

- But the problem in reducing the overhead is that, prior knowledge regarding the conflicting transactions will not be known. Therefore, a mechanism called "monitoring" the system is required to gain such knowledge.

Let us consider that every transaction T_A is executed in two or three-phases during its life-time. The phases involved in optimistic concurrency control are,

1) Read Phase
2) Validation Phase and
3) Write Phase

1) Read phase: In this phase, the copies of the data items (their values) are stored in local variables and the modifications are made to these local variables and the actual values are not modified in this phase.

2) Validation Phase: This phase follows the read phase where the assurance of the serializability is checked upon each update. If the conflicts occur between the transaction, then it is aborted and restarted else it is committed.

3) Write Phase : The successful completion of the validation phase leads to the write phase in which all the changes are made to the original copy of data items. This phase is applicable only to the read-write transaction. Each transaction is assigned three timestamps as follows,

i) When execution is initiated I(T)
ii) At the start of the validation phase V(T)
iii) At the end of the validation phase E(T)

Qualifying conditions for successful validation:

Consider two transactions, transaction T_A, transaction T_B and let the timestamp of transaction T_A is less than the timestamp of transaction T_B i.e., $T_S(T_A) < T_S(T_B)$ then,

1) Before the start of transaction T_B, transaction T_A must complete its execution. i.e., $E(T_A) < I(T_B)$

2) The values written by transaction T_A must not be necessarily matched with the values read by transaction T_B. T_A must execute the write phase before T_B initiate the execution of validation phase, i.e., $I(T_B) < E(T_A) < V(T_B)$

3) If transaction T_A starts its execution before transaction T_B completes, then the write phase of transaction T_B must be finished before transaction T_A starts the validation phase.

Advantages:

i) The efficiency of optimistic techniques lie in the scarcity of the conflicts.

ii) It doesn’t cause the significant delays.

iii) Cascading rollbacks never occurs.

Disadvantages:

i) Wastage in processing time during the rollback of aborting transactions which are very long.

ii) Hence, when one process is in its critical section (a portion of its code), no other process is allowed to enter. This is the principal of mutual exclusion.