DISTRIBUTED DATABASES (R22CSE3146)

III B.Tech I Semester (Information Technology)

UNIT – I

Introduction : Distributed Data Processing, Distributed Database System, Promises of DDBSs, Problem areas.
Distributed DBMS Architecture : Architectural Models for Distributed DBMS, DDMBS Architecture.
Distributed Database Design : Alternative Design Strategies, Distribution Design issues, Fragmentation, Allocation.

Introduction – Distributed Databases:

- A distributed database is a database that runs and stores data across multiple computers, as opposed to doing everything on a single machine.
- Typically, distributed databases operate on two or more interconnected servers on a computer network.
- Each location where a version of the database is running is often called an instance or a node.
- A distributed database is basically a database that is not limited to one system, it is spread over different sites, i.e, on multiple computers or over a network of computers.
- A distributed database system is located on various sites that don't share physical components.
- This may be required when a particular database needs to be accessed by various users globally. It needs to be managed such that for the users it looks like one single database.

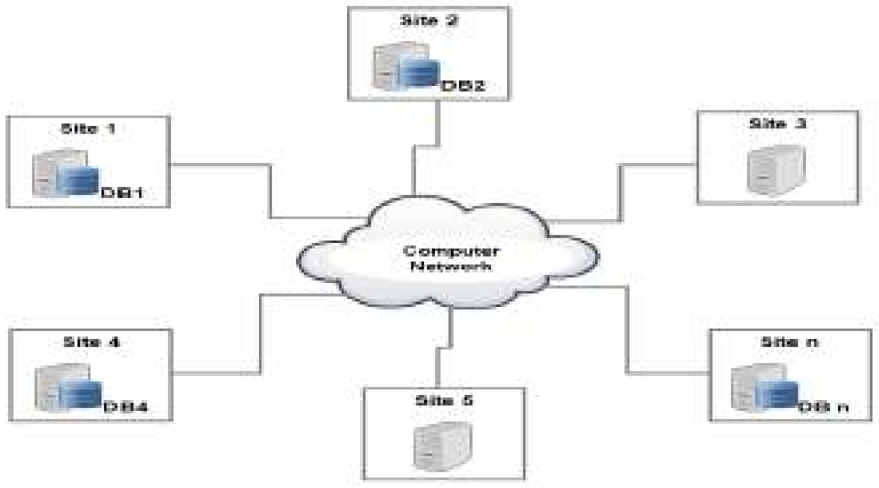


Fig: Distributed Database System

A distributed database system is a type of database management system that stores data across multiple computers or sites that are connected by a network. In a distributed database system, each site has its own database, and the databases are connected to each other to form a single, integrated system.

The main advantage of a distributed database system is that it can provide higher availability and reliability than a centralized database system. Because the data is stored across multiple sites, the system can continue to function even if one or more sites fail. In addition, a distributed database system can provide better performance by distributing the data and processing load across multiple sites.

Distributed Database Features

Some general features of distributed databases are:

- **Location independency -** Data is physically stored at multiple sites and managed by an independent DDBMS.
- **Distributed query processing -** Distributed databases answer queries in a distributed environment that manages data at multiple sites. High-level queries are transformed into a query execution plan for simpler management.
- **Distributed transaction management -** Provides a consistent distributed database through commit protocols, distributed concurrency control techniques, and distributed recovery methods in case of many transactions and failures.
- **Seamless integration -** Databases in a collection usually represent a single logical database, and they are interconnected.
- **Network linking -** All databases in a collection are linked by a network and communicate with each other.
- **Transaction processing -** Distributed databases incorporate transaction processing, which is a program including a collection of one or more database operations. Transaction processing is an atomic process that is either entirely executed or not at all.

Applications / Uses of Distributed Database

- It is used in Corporate Management Information System.
- It is used in multimedia applications.
- Used in Military's control system, Hotel chains etc.
- It is also used in manufacturing control system.

Architectures for distributed database systems

There are several different architectures for distributed database systems, including:

- **Client-server architecture:** In this architecture, clients connect to a central server, which manages the distributed database system. The server is responsible for coordinating transactions, managing data storage, and providing access control.

- **Peer-to-peer architecture:** In this architecture, each site in the distributed database system is connected to all other sites. Each site is responsible for managing its own data and coordinating transactions with other sites.

- **Federated architecture: I**n this architecture, each site in the distributed database system maintains its own independent database, but the databases are integrated through a middleware layer that provides a common interface for accessing and querying the data.

- Distributed database systems can be used in a variety of applications, including e-commerce, financial services, and telecommunications. However, designing and managing a distributed database system can be complex and requires careful consideration of factors such as data distribution, replication, and consistency.

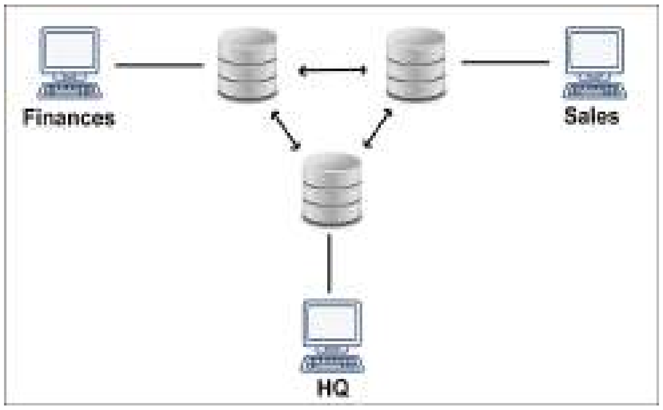
Distributed Database Types

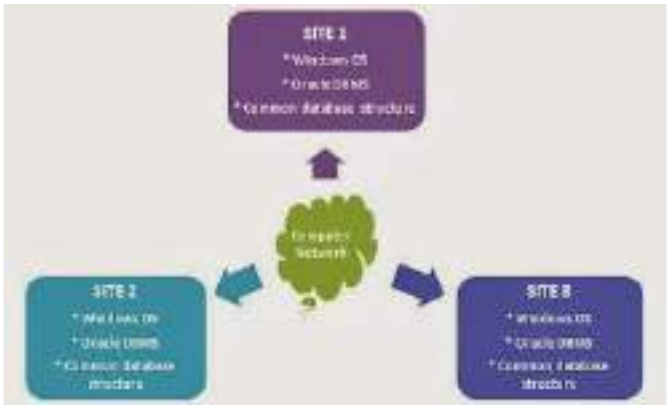
There are two types of distributed databases:

- **Homogenous**
- **Heterogeneous**

Homogeneous

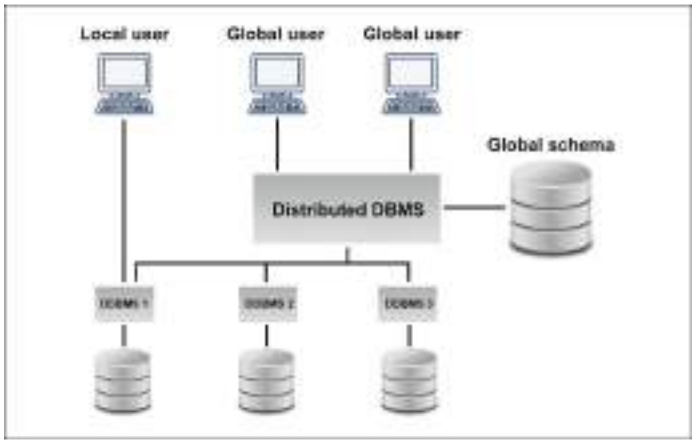
- A homogenous distributed database is a network of **identical databases** stored on multiple sites. The sites have the **same** operating system, DDBMS, and data structure, making them easily manageable.
- Homogenous databases allow users to access data from each of the databases seamlessly.
- The following diagram shows an example of a homogeneous database:

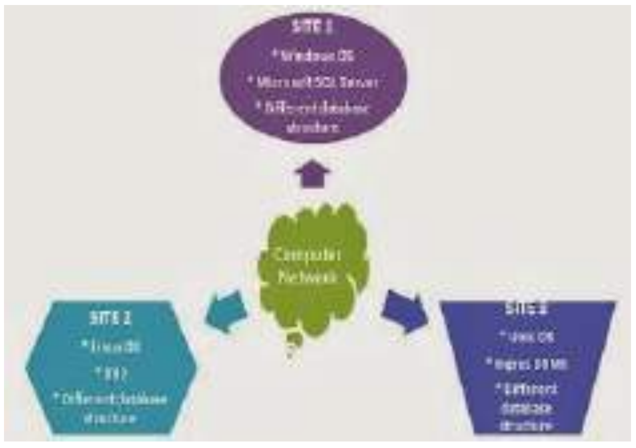




Heterogeneous

- A heterogeneous distributed database uses **different** schemas, operating systems, DDBMS, and different data models.
- In the case of a heterogeneous distributed database, a particular site can be completely unaware of other sites causing limited cooperation in processing user requests. The limitation is why translations are required to establish communication between sites.
- The following diagram shows an example of a heterogeneous database:





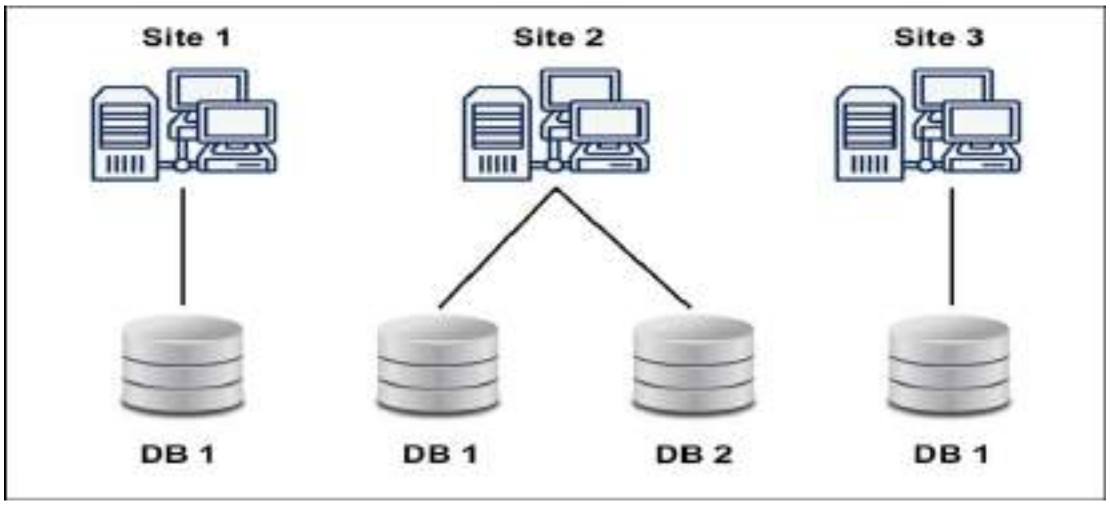
Distributed Database Storage

Distributed database storage is managed in two ways:

- **Replication**
- **Fragmentation**

Replication

- In database replication, the systems store **copies of data on different sites**. If an entire database is available on multiple sites, it is a fully redundant database.
- The advantage of database replication is that it **increases data availability o**n different sites and allows for parallel query requests to be processed.
- However, database replication means that data requires constant updates and synchronization with other sites to maintain an exact database copy. Any changes made on one site must be recorded on other sites, or else inconsistencies occur.
- Constant updates cause a lot of server overhead and complicate concurrency control, as a lot of concurrent queries must be checked in all available sites.



Fragmentation

- When it comes to fragmentation of distributed database storage, the relations are fragmented, which means they are **split into smaller parts**. Each of the fragments are stored on a different site, where it is required.
- The prerequisite for fragmentation is to make sure that the fragments can later be reconstructed into the original relation without losing data.
- The advantage of fragmentation is that there are **no data copies**, which prevents data inconsistency.

There are two types of fragmentation:

- **Horizontal fragmentation -** The relation schema is fragmented into groups of rows, and each group (tuple) is assigned to one fragment.

- **Vertical fragmentation -** The relation schema is fragmented into smaller schemas, and each fragment contains a common candidate key to guarantee a lossless join.

  Note: In some cases, a mix of fragmentation and replication is possible.

Distributed Database Advantages and Disadvantages

Below are some key advantages and disadvantages of distributed databases:

Advantages	**Disadvantages**
Modular development	Costly software
Reliability	Large overhead
Lower communication costs	Data integrity
Better response	Improper data distribution

The advantages and disadvantages are explained in detail in the following sections.

Advantages / Benefits of Distributed Databases:

- **Modular Development**. Modular development of a distributed database implies that a system **can be expanded to new locations or units** by adding new servers and data to the existing setup and connecting them to the distributed system without interruption. This type of expansion causes no interruptions in the functioning of distributed databases.

- **Reliability**. Distributed databases offer greater reliability in contrast to centralized databases. In case of a database failure in a centralized database, the system comes to a complete stop. In a distributed database, the system functions even when failures occur, only delivering reduced performance until the issue is resolved.

- **Lower Communication Cost**. Locally storing data reduces communication costs for data manipulation in distributed databases. Local data storage is not possible in centralized databases.

- **Better Response**. Efficient data distribution in a distributed database system provides a faster response when user requests are met locally. In centralized databases, user requests pass through the central machine, which processes all requests. The result is an increase in response time, especially with a lot of queries.

Disadvantages / Issues of Distributed Databases:

- **Costly Software**. Ensuring data transparency and coordination across multiple sites often requires using expensive software in a distributed database system.
- **Large Overhead**. Many operations on multiple sites requires numerous calculations and constant synchronization when database replication is used, causing a lot of processing overhead.
- **Data Integrity**. A possible issue when using database replication is data integrity, which is compromised by updating data at multiple sites.
- **Improper Data Distribution**. Responsiveness to user requests largely depends on proper data distribution. That means responsiveness can be reduced if data is not correctly distributed across multiple sites.

Centralized Database Vs Distributed Database

Centralized DBMS	***Distributed DBMS***
In Centralized DBMS the database are stored in a only one site	In Distributed DBMS the database are stored in different site and help of network it can access it
If the data is stored at a single computer site, which can be used by multiple users	Database and DBMS software distributed over many sites, connected by a computer network
Database is maintained at one site	Database is maintained at a number of different sites
If centralized system fails, entire system is halted	If one system fails, system continues work with other site
It is a less reliable	It is a more reliable

Centralized database

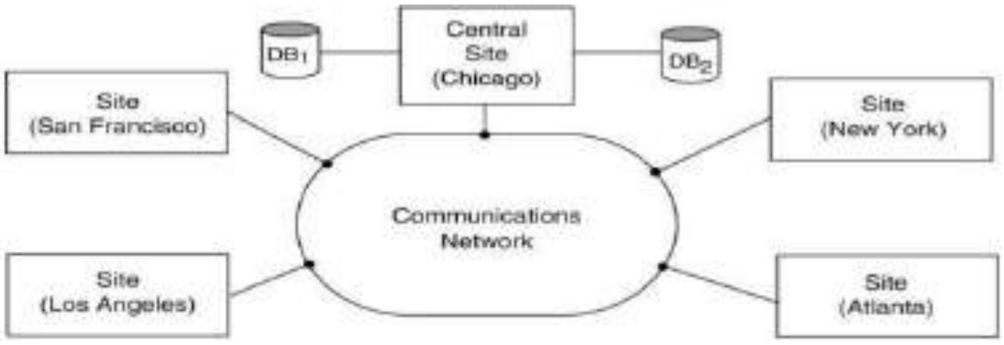


Fig : Centralized database

Distributed database

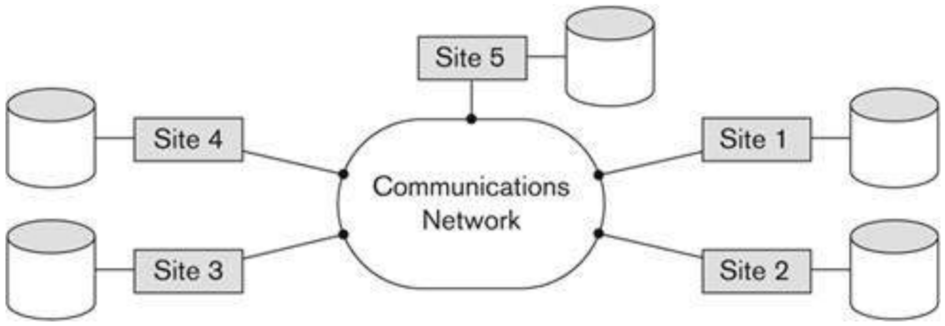


Fig : Distributed database

Types of Distributed Databases

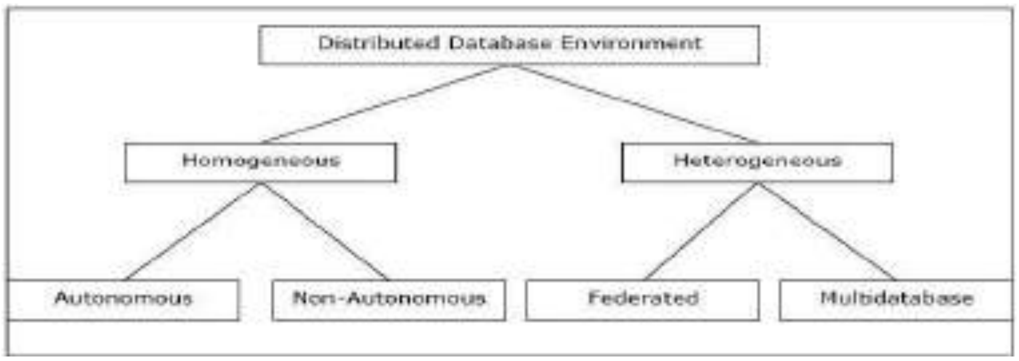


Fig : Types of Distributed Databases

Examples of distributed databases

Some common examples of distributed databases include:

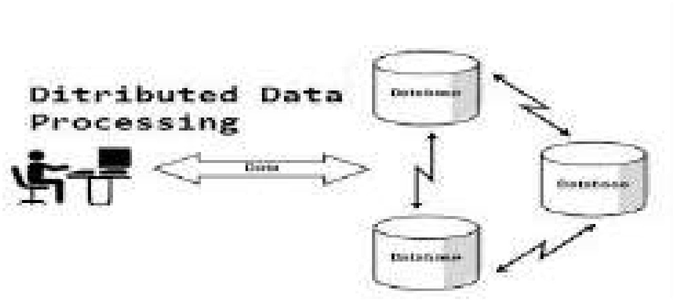
- Apache Ignite
- Apache Cassandra
- Apache HBase
- Couchbase Server
- Amazon SimpleDB
- Clusterpoint
- FoundationDB

Distributed data processing

Distributed data processing refers to the approach of handling and analyzing data across multiple interconnected devices or nodes. (or)

Distributed data processing having different database files located at different sites in a network is known as DDP (Distributed Data Processing).

In contrast to centralized data processing, where all data operations occur on a single, powerful system, distributed processing decentralizes these tasks across a network of computers.



Distributed Processing is a computing approach that involves dividing tasks across multiple machines or nodes in a network. Instead of relying on a single machine to process large amounts of data, the workload is distributed among multiple machines, enabling **parallel processing**. The distributed nature of processing allows for increased performance, scalability, and **fault tolerance**.

How Distributed Data Processing works?

In a distributed processing system, a central coordinator assigns tasks to different nodes in the network. Each node processes its assigned task independently and communicates the results back to the coordinator. The coordinator then combines the results to produce the final output.

Distributed processing can be achieved through various mechanisms, including message passing, shared memory, or a combination of both. Communication between nodes can occur through direct point-to-point connections or via a shared communication infrastructure such as a message queue or **distributed file system**.

In a Distributed data processing system, a massive amount of data flows through several different sources into the system. This process of data flow is known as **data ingestion.**

Once the data streams in, there are different layers in the system architecture that breakdown the entire processing into several different parts.

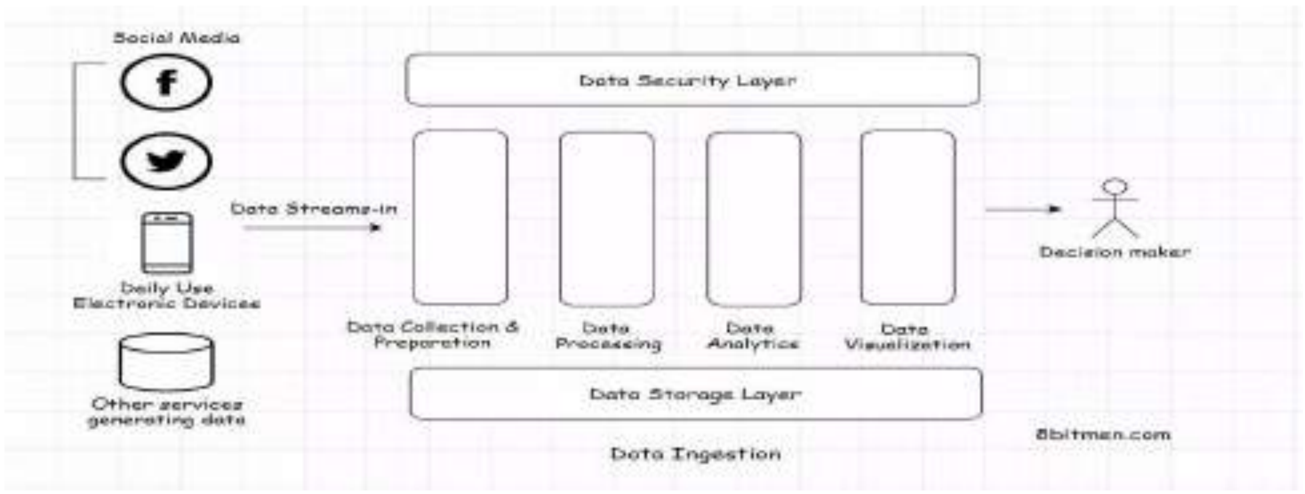


Fig : Data Ingestion

Data Collection and Preparation:

- This layer takes care of collecting data from different external sources and prepares it to be processed by the system.
- It may be Text, audio, video, image, tax returns forms, insurance forms, medical bills, etc.
- The task of the data preparation layer is to convert the data into a consistent standard format, also to classify it as per the business logic to be processed by the system. This is automated fashion without any sort of human intervention.

Data Security Layer

The role of this layer is to ensure that the data transit is secure by watching over it through out with applied security protocols, encryption like that.

Data Storage Layer

Here, Data storage layer is used to store the big amount of data.

Data Processing Layer

This is the layer that contains the business logic for data processing. Machine Learning, predictive, descriptive and decision modeling are primarily used to extract meaningful information.

Data Visualization Layer

All the information extracted is sent to the data visualization layer which typically contains browser based dashboards which display the information in the form of graphs, charts and infographics.

Why Distributed Processing is important

Distributed processing offers several benefits that make it important for data processing and analytics:

- **Improved Performance:** By distributing the workload across multiple machines, distributed processing can significantly reduce the processing time compared to a single machine. This is especially crucial when dealing with large datasets or complex computational tasks.
- **Scalability:** Distributed processing allows organizations to scale their computing resources by adding or removing nodes as needed. This flexibility enables businesses to handle increased workloads and accommodate future growth without a significant impact on performance.
- **Fault Tolerance:** In a distributed processing system, if one node fails or experiences issues, the workload can be automatically rerouted to other available nodes. This fault tolerance ensures that processing continues uninterrupted and reduces the risk of data loss.
- **Cost Efficiency:** With distributed processing, organizations can utilize commodity hardware instead of relying on expensive high-end servers. This reduces hardware costs and allows businesses to achieve higher computing power at a lower price point.

Advantages of distributed data processing (DDP)
Inexpensive:

Data is also distributed so adding and removing nodes (computers) can be easy. To achieve distributed networking, we can use Beowulf cluster technology. In Beowulf cluster, remote computers are assigned processing through network switches and routers.

Easy to replace remote computers:

Microsoft Windows server has a feature called failover clustering that helps to remove faulty computers. If any computer on the network fails or corrupted by some means, then that computer is automatically replaced by other computers.

Optimized processing:

Managing data on online server solves slow processing. On the personal computer, we can do extra tasks also. Doing extra tasks consumes processor power. But the online computer is dedicated to one type of processing and it is more likely to increase processing powers. Database server can only handle database queries and file server stores files. So data processing is optimized.

Easy to expand:

Suppose your company needs more data processing than expected then you can easily attach more computers to the distributed network.

Parallel processing:

Adding and removing computers from the network cannot disturb data flow. All data from different computers are processed in parallel. Parallel processing means data is updated at the same time from all nodes.

Better performance:

The overall performance of the company gets better and data is filtered and processed more rapidly in the distributed environment.

Backup of data:

Data can be backup from any computer connected to the network. So the user can backup data at a different time and work with that data locally and then upload the data to the server.

Local data synchronization:

All the computers on the network can have local storage of important data. Suppose there are different office branches interconnected to each other. All branch computers are interlinked with the main branch office. All office branch computers have a local copy of data. Office users edit and update data and then upload to the main server. So the data is synced and available to all computers. Working locally with data is easy and fast and when the user thinks that his work is complete then at the end of the day he can sync that data with the main server.

Data recovery:

If some data like the database is a loss in any computer then it can be recovered by another interconnected computer i.e. main database server.

Disadvantages of distributed data processing (DDP)
Complexity:

Computers attached in DDP are difficult to troubleshoot, design and administrate.

Planning data synchronization is difficult:

Doing the correct synchronization of data is difficult to develop. Sometimes data is updated in wrong order. So administrators have to keep the focus on it before making a distributed network.

Data security:

If the unauthorized computer is connected to a distributed network then it can affect other computer performance and data can be a loss also.

Examples of distributed data processing

- Hosting a website on the online server
- Online photo editing tools
- Airline ticketing system
- Processing user data by mobile companies
- Dropbox, Google drive, MSN drive, Google photos
- Report generation from satellite
- Weather forecast system

Promises of DDBSs

There are four fundamentals which may also be viewed as promises of DDBS technology:

- Transparent management of distributed and replicated data
- Reliable access to data through distributed transactions
- Improved performance
- Easiersystem expansion

1. Transparent Management of Distributed and Replicated Data

- A transparent system “hides” the implementation details from users.
- The advantage of a fully transparent DBMS is the high level of support that it provides for the development of complex applications.

Example:

- Consider an engineering firm that has offices in Boston, Waterloo, Paris and San Francisco.
- They run projects at each of these sites and would like to maintain a database of their employees, the projects and other related data.
- Assuming that the database is relational, we can storethis information in two relations:
  EMP(ENO, ENAME, TITLE)
  PROJ(PNO, PNAME, BUDGET).
- We also introduce a third relation to store salary information: SAL(TITLE, AMT) and a fourth relation ASG which indicates which employees have been assigned to which projects for what duration with what responsibility: ASG(ENO, PNO, RESP, DUR).
- If all of this data were stored in a centralized DBMS, and we wanted to find out the names and employees who worked on a project for more than 12 months, we would specify this using the following SQL query:

```
SELECT ENAME, AMT FROM EMP, ASG, SAL
WHERE  ASG.DUR > 12

AND   EMP.ENO = ASG.ENO

AND   SAL.TITLE = EMP.TITLE
```

- Based on the query it is going to search in different databases of Boston, Paris, etc..
- In order to quick processing of query we are going to partition each of the relations and store each partition at a different site. This is known as ***fragmentation.***
- Some times we also duplicate some of this data at other sites for performance and reliability reasons. The result is a distributed database which is fragmented and replicated.
- Fully transparent access means that the users can still pose the query as specified above, without paying any attention to the fragmentation, location, or replication of data, and let the system worry about resolving these issues.
- For a system to adequately deal with this type of query over a distributed, fragmented and replicated database, it needs to be able to deal with a number of different types of transparencies.

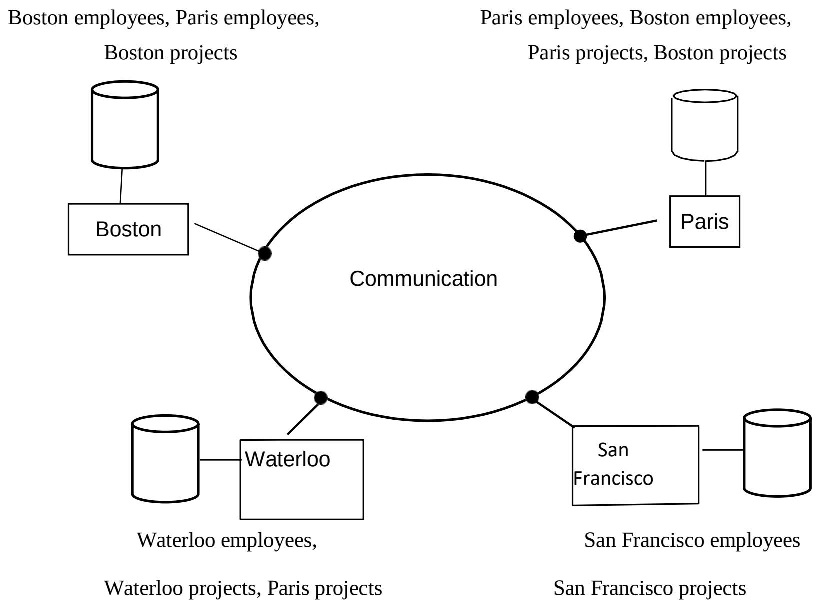

Boston employees, Paris employees,
Boston projects
Paris employees, Boston employees,
Paris projects, Boston projects
Boston
Paris
Communication
Waterloo
San Francisco
Waterloo employees,
Waterloo projects, Paris projects
San Francisco employees
San Francisco projects

Fig. A Distributed Application

Types of Transparencies

- Data Independency
- Network Transparency
- Replication Transparency
- Fragmentation Transparency

Data Independency

- Data independence is a fundamental form of transparency that we look for within a DBMS.
- It is also the only type that is important within the context of a centralized DBMS.
- It refers to the immunity of user applications to changes in the definition and organization of data.
- Data definition occurs at two levels.
- At one level the logical structure of the data are specified, and at the other level its physical structure.
- The former is commonly known as the *schema definition*, whereas the latter is referred to as the *physical data description*.
- Two types of data independence:
    1. **Logical data independence**

       Logical data independence refers to the immunity of user applications to changes in the logical structure (i.e., schema) of the database.

    2. **Physical data independence.**

       Physical data independence, on the other hand, deals with hiding the details of the storage structure from user applications. When a user application is written, it should not be concerned with the details of physical data organization. Therefore, the user application should not need to be modified when data organization changes occur due to performance considerations.

Network Transparency

- The user should be protected from the operational details of the network.
- This type of transparency is referred to as *network transparency* or *distribution transparency*.
- Sometimes two types of distribution transparency are identified:

    1. **Location transparency**
       Location transparency refers to the fact that the command used to perform a task is independent of both the location of the data and the system on which an operation is carried out.

    2. **Naming transparency.**
       Naming transparency means that a unique name is provided for each object in the database. In the absence of naming transparency, users are required to embed the location name (or an identifier) as part of the object name.

Replication Transparency

- Replication Transparency ensures that replication of databases are hidden from the users.
- It enables users to query upon a table as if only a single copy of the table exists.
- It is associated with concurrency transparency and failure transparency.
- Whenever the user updates a data item, the update is reflected in all the copies of the table. This operation should not be known to the user this is called concurrency transparency.
- In case of failure of a site, the user can still proceed with his queries using replicated copies without any knowledge of failure. This is failure transparency.

Fragmentation Transparency

- The final form of transparency that needs to be addressed within the context of a distributed database system is that of fragmentation transparency.
- it is commonly desirable to divide each database relation into smaller fragments and treat each fragment as a separate database object (i.e., another relation).
- This is commonly done for reasons of performance, availability, and reliability. Furthermore, fragmentation can reduce the negative effects of replication. Each replica is not the full relation but only a subset of it; thus less space is required and fewer data items need be managed.

- There are two general types of fragmentation alternatives.
  1. *Horizontal fragmentation*
  2. *Vertical fragmentation*

- *horizontal fragmentation*, a relation is partitioned into a set of sub-relations each of which have a subset of the tuples (rows) of the original relation.

- *vertical fragmentation* where each sub-relation is defined on a subset of the attributes (columns) of the original relation.

- When database objects are fragmented, we have to deal with the problem of handling user queries that are specified on entire relations but have to be executed on sub relations. In other words, the issue is one of finding a query processing strategy based on the fragments rather than the relations

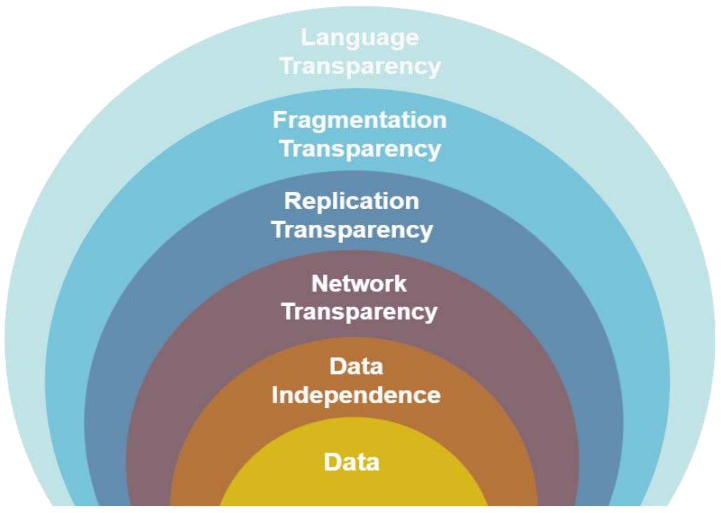


Fig : Levels of Transparency

2.Reliability Through Distributed Transactions

- Distributed DBMSs are intended to improve reliability since they have replicated components and, thereby eliminate single points of failure.
- The failure of a single site, or the failure of a communication link which makes one or more sites unreachable, is not sufficient to bring down the entire system.
- In the case of a distributed database, this means that some of the data may be unreachable, but with proper care, users may be permitted to access other parts of the distributed database.
- The "proper care" comes in the form of support for distributed transactions and application protocols.
- **Example:** Let us take an example of a transaction based on the engineering firm.
- Assume that there is an application that updates the salaries of all the employees by 10%.
- It is desirable to encapsulate the query (or the program code) that accomplishes this task within transaction boundaries. For example, if a system failure occurs half-way through the execution of this program, we would like the DBMS to be able to determine, upon recovery, where it left off and continue with its operation (or start all over again).
- This is the topic of failure atomicity. Alternatively, if some other user runs a query calculating the average salaries of the employees in this firm while the original update action is going on, the calculated result will be in error.
- Therefore, we would like the system to be able to synchronize the *concurrent* execution of these two programs. To encapsulate a query (or a program code) within transactional boundaries, it is sufficient to declare the begin of the transaction and its end:

```
Begin transaction SALARY UPDATE
begin
        EXEC SQL UPDATE   PAY  SET    SAL = SAL*1.1
end.
```

- Distributed transactions execute at a number of sites at which they access the local database.
- Here, we are providing a facility that there is no interruption in any transaction.

3. Improved Performance

- The improved performance of distributed DBMSs is typically made based on two points.
- First, a distributed DBMS fragments the conceptual database, enabling data to be stored in close proximity to its points of use (also called *data localization*).
- This has two potential advantages:

  1. Since each site handles only a portion of the database, contention for CPU and I/O services is not as severe as for centralized databases.

  2. Localization reduces remote access delays that are usually involved in wide area networks (for example, the minimum round-trip message propagation delay in satellite-based systems is about 1 second).
- Implementation of inherent parallelism of distributed system.
- parallelism of distributed systems may be exploited for inter-query and intra-query parallelism.
- **Inter-query parallelism** results from the ability to execute multiple queries at the same time.
- **Intra-query parallelism** is achieved by breaking up a single query into a number of subqueries each of which is executed at a different site, accessing a different part of the distributed database.

4. Easier System Expansion

- In a distributed environment, it is much easier to accommodate increasing database sizes.
- In general, expansion can usually be handled by adding processing and storage power to the network.
- This also depends on the overhead of distribution.
- One aspect of easier system expansion is economics.
- It normally costs much less to put together a system of "smaller" computers with the equivalent power of a single big machine

Problem Areas / Complications Introduced by Distribution

- The problems encountered in database systems take on additional complexity in a distributed environment, even though the basic underlying principles are the same.
- Furthermore, this additional complexity gives rise to new problems influenced mainly by three factors.

1.First, data may be replicated in a distributed environment. A distributed database can be designed so that the entire database, or portions of it, reside at different sites of a computer network.

- It is not essential that every site on the network contain the database; it is only essential that there be more than one site where the database resides.
- The possible duplication of data items is mainly due to reliability and efficiency considerations. Consequently, the distributed database system is responsible for
  (1) choosing one of the stored copies of the requested data for access in case of retrievals, and
  (2) making sure that the effect of an update is reflected on each and every copy of that data item.

2.Second, if some sites fail (e.g., by either hardware or software malfunction), or if some communication links fail (making some of the sites unreachable) while an update is being executed, the system must make sure that the effects will be reflected on the data residing at the failing or unreachable sites as soon as the system can recover from the failure.

3.The third point is that since each site cannot have instantaneous information on the actions currently being carried out at the other sites, the synchronization of transactions on multiple sites is considerably harder than for a centralized system.

- These difficulties point to a number of potential problems with distributed DBMSs.
- These are the inherent complexity of building distributed applications, increased cost of replicating resources, and, more importantly, managing distribution, the devolution of control to many centers and the difficulty of reaching agreements, and the exacerbated security concerns (the secure communication channel problem).
- These are well-known problems in distributed systems in general.

Distributed DBMS Architecture

- The architecture of a system defines its structure.
- This means that the components of the system are identified, the function of each component is specified, and the interrelationships and interactions among these components are defined.
- The specification of the architecture of a system requires identification of the various modules, with their interfaces and interrelationships, in terms of the data and control flow through the system.
- We develop three "reference" architectures for a distributed DBMS:
  - i. Client/Server Systems
  - ii. Peer-To-Peer Distributed DBMS
  - iii. Multidatabase Systems
- These are "idealized" views of a DBMS in that many of the commercially available systems may deviate from them.

- **Two types of Architectures:**
  1. ANSI/SPARC Architecture
  2. A Generic Centralized DBMS Architecture

1. ANSI/SPARC Architecture

- In late 1972, the Computer and Information Processing Committee (X3) of the American National Standards Institute (ANSI) established a Study Group on Database.
- The ANSI/SPARC Architecture Management Systems under the auspices of its Standards Planning and Requirements Committee (SPARC). The mission of the study group was to study the feasibility of setting up standards in this area, as well as determining which aspects should be standardized if it was feasible. The study group issued its interim report in 1975 [ANSI/SPARC, 1975], and its final report in 1977.
- The architectural framework proposed in these reports came to be known as the "ANSI/SPARC architecture," its full title being "ANSI/X3/SPARC DBMS Framework.".
- The study group proposed that the interfaces be standardized, and defined an architectural framework that contained 43 interfaces, 14 of which would deal with the physical storage subsystem of the computer and therefore not be considered essential parts of the DBMS architecture.
- A simplified version of the ANSI/SPARC architecture is depicted in Figure. There are three views of data: the external view, which is that of the end user, who might be a programmer; the internal view, that of the system or machine; and the conceptual view, that of the enterprise. For each of these views, an appropriate schema definition is required.
- At the lowest level of the architecture is the internal view, which deals with the physical definition and organization of data.
- The location of data on different storage devices and the access mechanisms used to reach and manipulate data are the issues dealt with at this level. At the other extreme is the external view, which is concerned with how users view the database.
- An individual user's view represents the portion of the database that will be accessed by that user as well as the relationships that the user would like to see among the data.

- A view can be shared among a number of users, with the collection of user views making up the external schema. In between these two ends is the conceptual schema, which is an abstract definition of the database.
- It is the “real world” view of the enterprise being modeled in the database. As such, it is supposed to represent the data and the relationships among data without considering the requirements of individual applications or the restrictions of the physical storage media. In reality, however, it is not possible to ignore these requirements completely, due to performance reasons.
- The transformation between these three levels is accomplished by mappings that specify how a definition at one level can be obtained from a definition at another level. This perspective is important, because it provides the basis for data independence that we discussed earlier.
- The separation of the external schemas from the conceptual schema enables logical data independence, while the separation of the conceptual schema from the internal schema allows physical data independence.

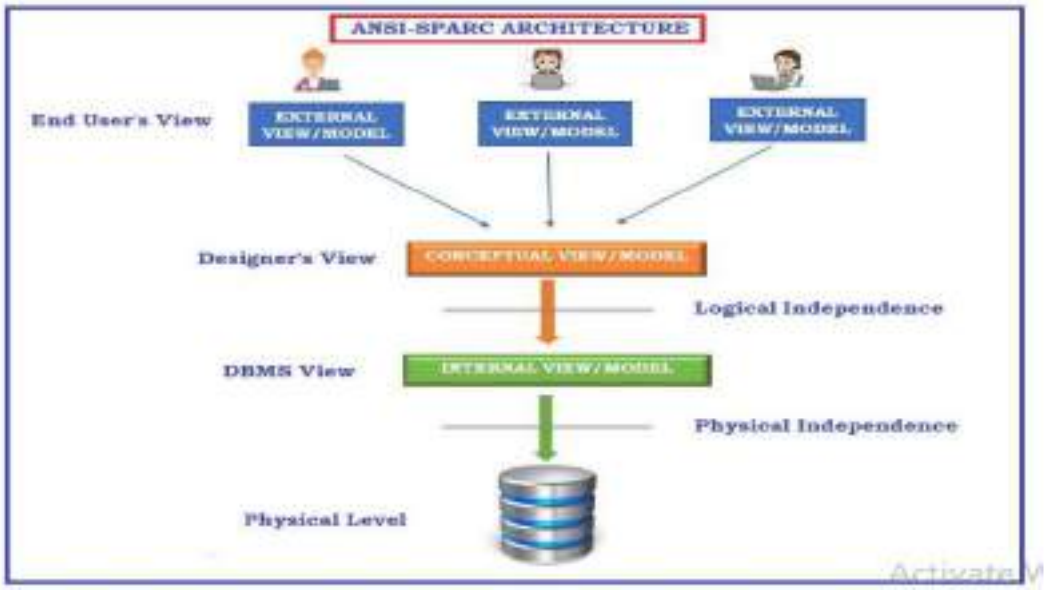


Fig : ANSI/ SPARC ARCHITECTURE

2.Generic Centralized DBMS Architecture

The operating system provides the interface between the DBMS and computer resources (processor, memory, disk drives, etc.).
The functions performed by a DBMS can be layered as in the following Figure, where the arrows indicate the direction of the data and the control flow. Taking a top-down approach, the layers are the interface, control, compilation, execution, data access, and consistency management.

- The First layer is interface layer. The interface layer manages the interface to the applications.
  There can be several interfaces used here.
- The control layer controls the query by adding semantic integrity predicates and authorization predicates. Semantic integrity constraints and authorizations are usually specified in a declarative language, The output of this layer is an enriched query in the high-level language accepted by the interface.

- The query processing (or compilation) layer maps the query into an optimized sequence of lower-level operations. This layer is concerned with performance. Functional Layers of a Centralized DBMS decomposes the query into a tree of algebra operations and tries to find the “optimal” ordering of the operations. The result is stored in an access plan. The output of this layer is a query expressed in lower-level code (algebra operations).
- The execution layer directs the execution of the access plans, including transaction management (commit, restart) and synchronization of algebra operations. It interprets the relational operations by calling the data access layer through the retrieval and update requests.
- The data access layer manages the data structures that implement the files, indices, etc. It also manages the buffers by caching the most frequently accessed data. Careful use of this layer minimizes the access to disks to get or write data.
- Finally, the consistency layer manages concurrency control and logging for update requests. This layer allows transaction, system, and media recovery after failure.

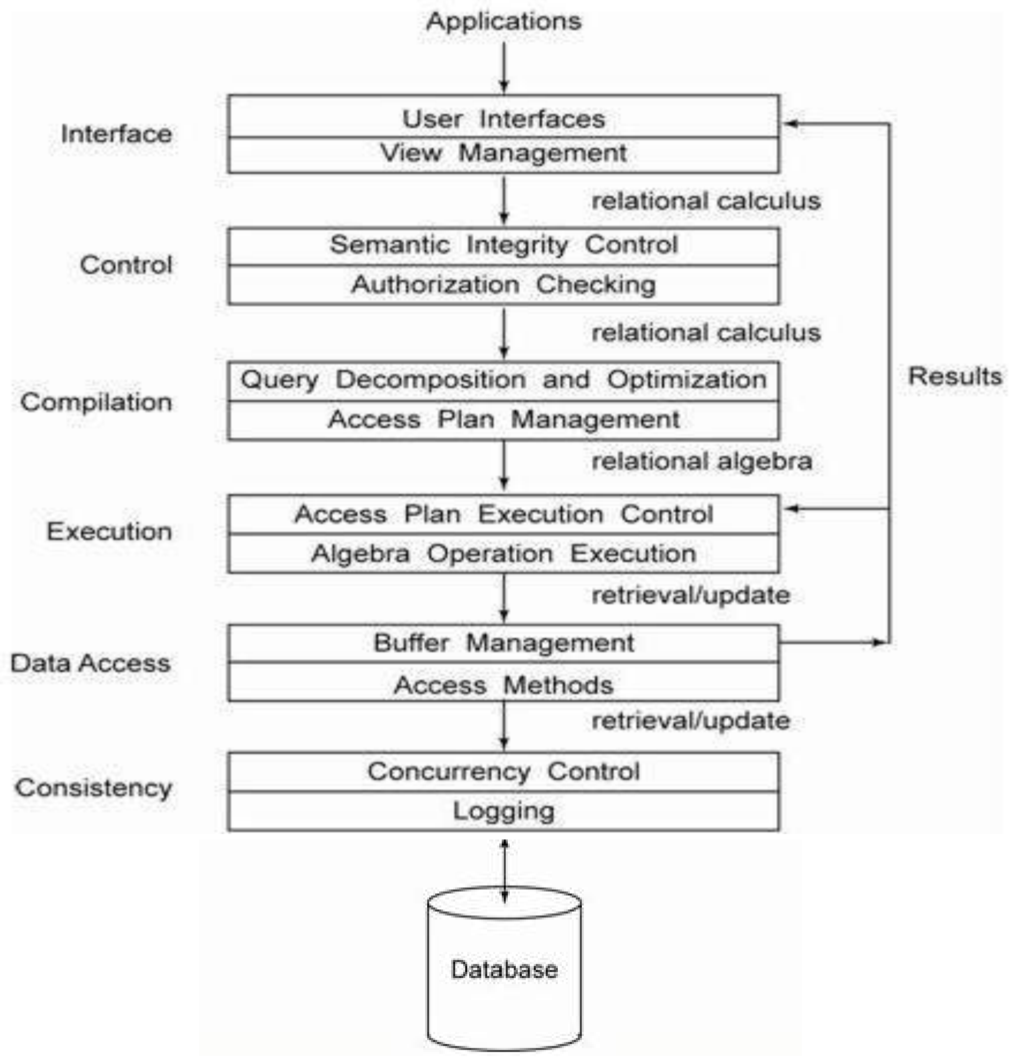


Fig : Centralized DBMS Architecture

Architectural Models for Distributed DBMSs

We now consider the possible ways in which a distributed DBMS may be architected. We use a classification (Figure) that organizes the systems as characterized with respect to
(1) The Autonomy of Local Systems
(2) Their Distribution
(3) Their Heterogeneity

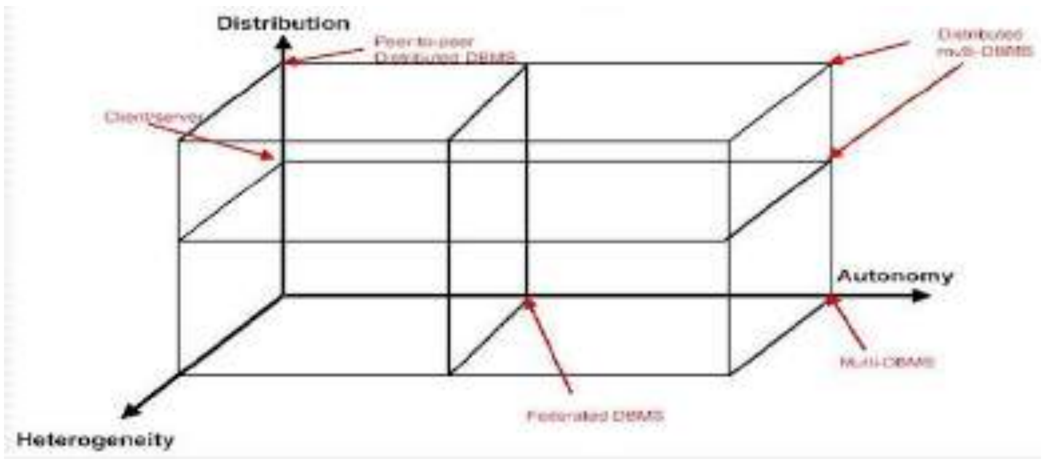


Fig : Architectural Models for DDBMS

Autonomy

- Autonomy, in this context, refers to the distribution of control, not of data. It indicates the degree to which individual DBMSs can operate independently.
- Autonomy is a function of a number of factors such as whether the component systems (i.e., individual DBMSs) exchange information, whether they can independently execute transactions, and whether one is allowed to modify them.

Requirements of an autonomous system have been specified as follows:
1. The local operations of the individual DBMSs are not affected by their participation in the distributed system.
2. The manner in which the individual DBMSs process queries and optimize them should not be affected by the execution of global queries that access multiple databases.
3. System consistency or operation should not be compromised when individual DBMSs join or leave the distributed system.

On the other hand, the dimensions of autonomy can be specified as follows:
1. **Design autonomy**: Individual DBMSs are free to use the data models and transaction management techniques that they prefer.
2. **Communication autonomy**: Each of the individual DBMSs is free to make its own decision as to what type of information it wants to provide to the other DBMSs or to the software that controls their global execution.
3. **Execution autonomy**: Each DBMS can execute the transactions that are submitted to it in any way that it wants to.

We will use a classification that covers the important aspects of these features. Other alternatives are

1. Tight Integration
2. Semiautonomous Systems
3. Total Isolation

Tight Integration

- A single-image of the entire database is available to any user who wants to share the information, which may reside in multiple databases. From the users' perspective, the data are logically integrated in one database.
- In these tightly-integrated systems, the data managers are implemented so that one of them is in control of the processing of each user request even if that request is serviced by more than one data manager.
- The data managers do not typically operate as independent DBMSs even though they usually have the functionality to do so.

Semiautonomous Systems

- It consist of DBMSs that can (and usually do) operate independently, but have decided to participate in a federation to make their local data sharable.
- Each of these DBMSs determine what parts of their own database they will make accessible to users of other DBMSs.
- They are not fully autonomous systems because they need to be modified to enable them to exchange information with one another.

Total Isolation

- The last alternative that we consider is total isolation, where the individual systems are stand-alone DBMSs that know neither of the existence of other DBMSs nor how to communicate with them.
- In such systems, the processing of user transactions that access multiple databases is especially difficult since there is no global control over the execution of individual DBMSs.
- It is important to note at this point that the three alternatives that we consider for autonomous systems are not the only possibilities.

Distribution

- Autonomy refers to the distribution (or decentralization) of control, the distribution dimension of the taxonomy deals with data. Of course, we are considering the physical distribution of data over multiple sites.
- The user sees the data as one logical pool. There are a number of ways DBMSs have been distributed.
- We abstract these alternatives into two classes:
    1. Client/server distribution
    2. Peer-to-peer distribution (or full distribution).

Together with the non-distributed option, the taxonomy identifies three alternative architectures.

The client/server distribution

- It concentrates data management duties at servers while the clients focus on providing the application environment including the user interface.
- The communication duties are shared between the client machines and servers. Client/server DBMSs represent a practical compromise to distributing functionality.
- There are a variety of ways of structuring them, each providing a different level of distribution. With respect to the framework, which we devote to client/server DBMS architectures.
- What is important at this point is that the sites on a network are distinguished as "clients" and "servers" and their functionality is different.

Peer-to-peer distribution (or full distribution).

- In peer-to-peer systems, there is no distinction of client machines versus servers. Each machine has full DBMS functionality and can communicate with other machines to execute queries and transactions.
- Most of the very early work on distributed database systems have assumed peer-to-peer architecture.

Heterogeneity

- Heterogeneity may occur in various forms in distributed systems, ranging from hardware heterogeneity and differences in networking protocols to variations in data managers.
- Representing data with different modeling tools creates heterogeneity because of the inherent expressive powers and limitations of individual data models.
- Heterogeneity in query languages not only involves the use of completely different data access paradigms in different data models (set-at-a-time access in relational systems versus record-at-a-time access in some object-oriented systems), but also covers differences in languages even when the individual systems use the same data model.
- Although SQL is now the standard relational query language, there are many different implementations and every vendor's language has a slightly different flavor (sometimes even different semantics, producing different results)

Architectural Alternatives

- The distribution of databases, their possible heterogeneity, and their autonomy are orthogonal issues. Consequently, following the above characterization, there are 18 different possible architectures.
- Not all of these architectural alternatives that form the design space are meaningful.
- In Figure we have identified three alternative architectures.
- That we discuss in more detail in the next three subsections:

1. (A0, D1, H0) that corresponds to client/server distributed DBMSs,
2. (A0, D2, H0) that is a peer-to-peer distributed DBMS and
3. (A2, D2, H1) which represents a (peer-to- peer) distributed, heterogeneous multi database system.

Note that we discuss the heterogeneity issues within the context of one system architecture, although the issue arises in other models as well.

Client/Server Systems

- Client/server DBMSs entered the computing scene at the beginning of 1990's and have made a significant impact on both the DBMS technology and the way we do computing.
- The general idea is very simple and elegant: distinguish the functionality that needs to be provided and divide these functions into two classes:
  1. Server functions
  2. Client functions.
- This provides a *two-level architecture* which makes it easier to manage the complexity of modern DBMSs and the complexity of distribution.
- Thus, we focus on what software should run on the client machines and what software should run on the server machine.
- The functionality allocation between clients and servers differ in different types of distributed DBMSs (e.g., relational versus object-oriented).
- In relational systems, the server does most of the data management work. This means that all of query processing and optimization, transaction management and storage management is done at the server.
- The client, in addition to the application and the user interface, has a *DBMS client* module that is responsible for managing the data that is cached to the client and (sometimes) managing the transaction locks that may have been cached as well.
- There is operating system and communication software that runs on both the client and the server, but we only focus on the DBMS related functionality. This architecture, depicted in Figure is quite common in relational systems where the communication between the clients and the server(s) is at the level of SQL statements.
- In other words, the client passes SQL queries to the server without trying to understand or optimize them. The server does most of the work and returns the result relation to the client.
- There are a number of different types of client/server architecture. The simplest is the case where there is only one server which is accessed by multiple clients.
- We call this *multiple client/single server*. From a data management perspective, this is not much different from centralized databases since the database is stored on only one machine (the server) that also hosts the software to manage it.
- However, there are some (important) differences from centralized systems in the way transactions are executed and caches are managed. We do not consider such issues at this point. A more sophisticated client/server architecture is one where there are multiple servers in the system (the so-called *multiple client/multiple server* approach).
- In this case, two alternative management strategies are possible:
  1. Either each client manages its own connection to the appropriate server or each client knows of only its "home server" which then communicates with other servers as required.

The former approach simplifies server code, but loads the client machines with additional responsibilities. This leads to what has been called **“heavy client”** systems.

- The latter approach, on the other hand, concentrates the data management functionality at the servers. Thus, the transparency of data access is provided at the server interface, leading to **“light clients.”**

- Thus the primary distinction between client/server systems and peer-to-peer ones is not in the level of transparency that is provided to the users and applications, but in the architectural paradigm that is used to realize this level of transparency.

- Client/server can be naturally extended to provide for a more efficient function distribution on different kinds of servers:

- *client servers* run the user interface (e.g., web servers)
- *application servers* run application programs,
- *database servers* run database management functions.

- This leads to the present trend in three-tier distributed system architecture, where sites are organized as specialized servers rather than as general-purpose computers.

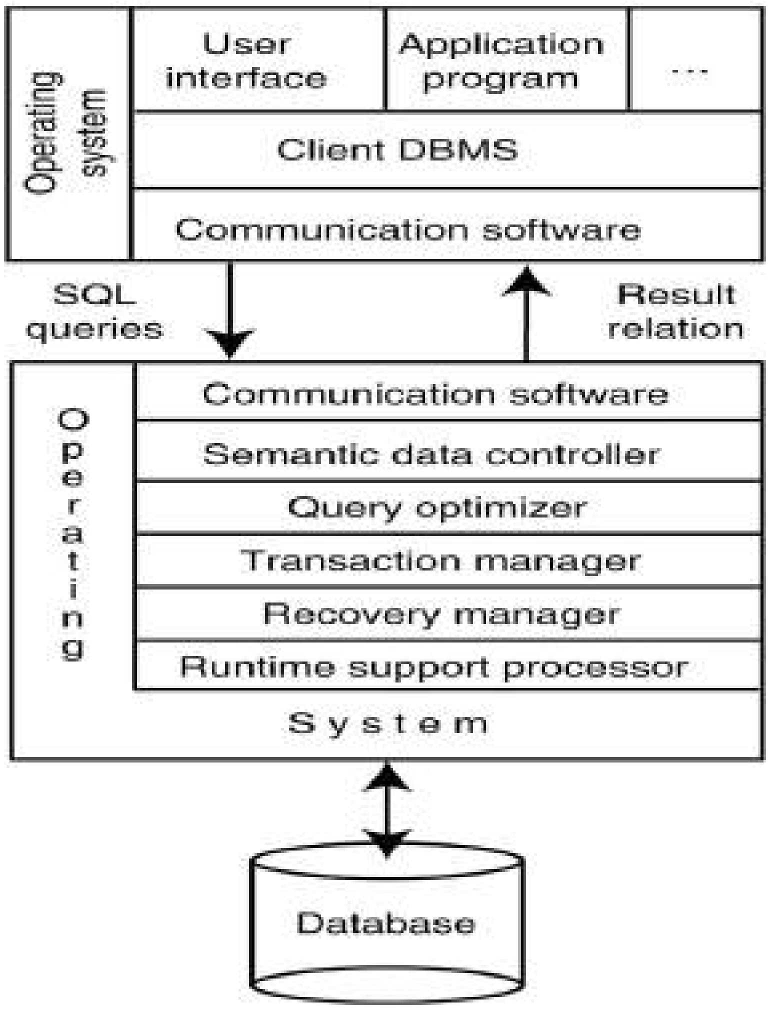


Fig: Client/ Server Reference Architecture

- The following Figure illustrates a simple view of the database server approach, with application servers connected to one database server via a communication network.

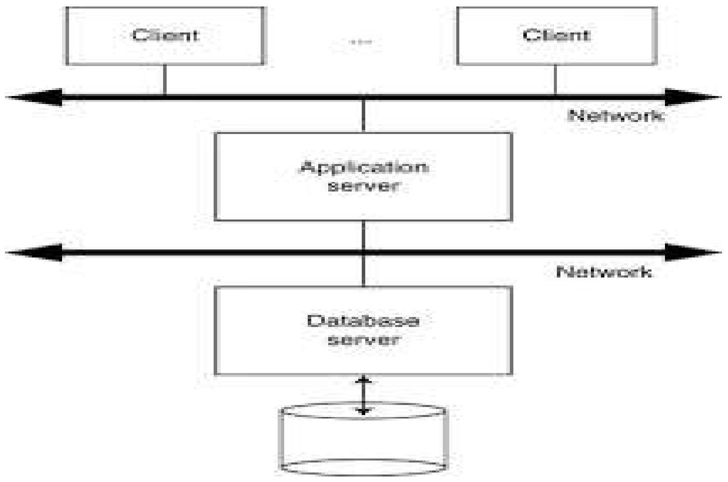


Fig : Database Server Approach

The database server approach, as an extension of the classical client/server architecture, has several potential advantages.

- First, the single focus on data management makes possible the development of specific techniques for increasing data reliability and availability, e.g. using parallelism.
- Second, the overall performance of database management can be significantly enhanced by the tight integration of the database system and a dedicated database operating system. Finally, a database server can also exploit recent hardware architectures, such as multiprocessors or clusters of PC servers to enhance both performance and data availability.
- The application server approach (indeed, a n-tier distributed approach) can be extended by the introduction of multiple database servers and multiple application servers can be done in classical client/server architectures.

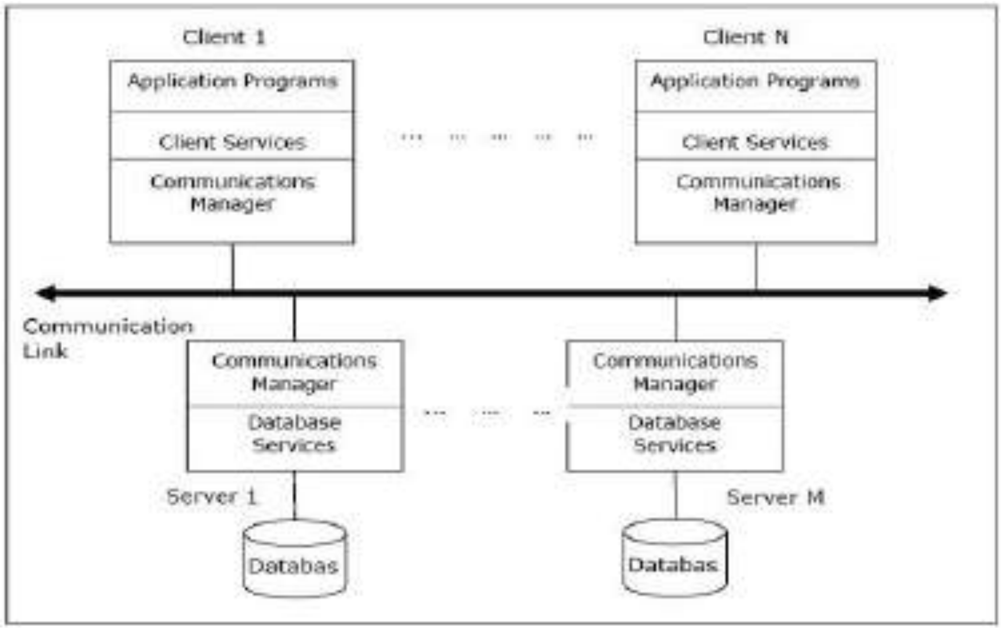


Fig : Distributed Database Servers

Peer-to-Peer Systems

- In these systems, each peer acts both as a client and a server for imparting database services. The peers share their resource with other peers and co-ordinate their activities.
- This architecture generally has four levels of schemas −
- Location and replication transparencies are supported by the definition of the local and global conceptual schemas and the mapping in between.

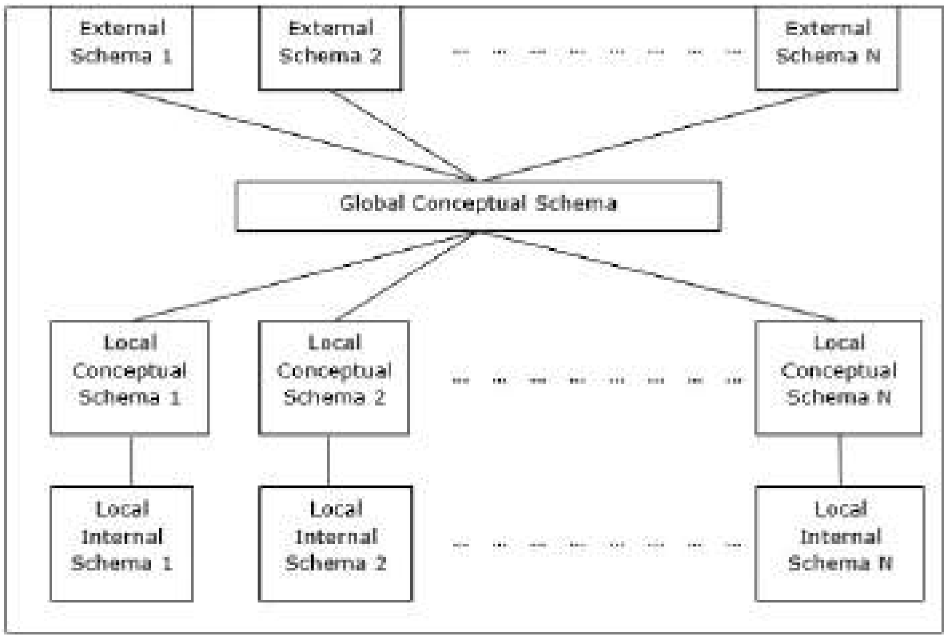


Fig : Distributed Database Reference Architecture

Local internal schema (LIS) : Individual internal schema definition at each site. Depicts physical data organization at each site.

Global conceptual schema (GCS) : The enterprise view of the data**.** Depicts the global logical view of data.

Local conceptual schema (LCS) : Logical organization of data at each site. Depicts logical data organization at each site.

External schemas (Ess) : Supports user applications and user access to the database. Depicts user view of data.

The detailed components of a distributed DBMS are showed below:

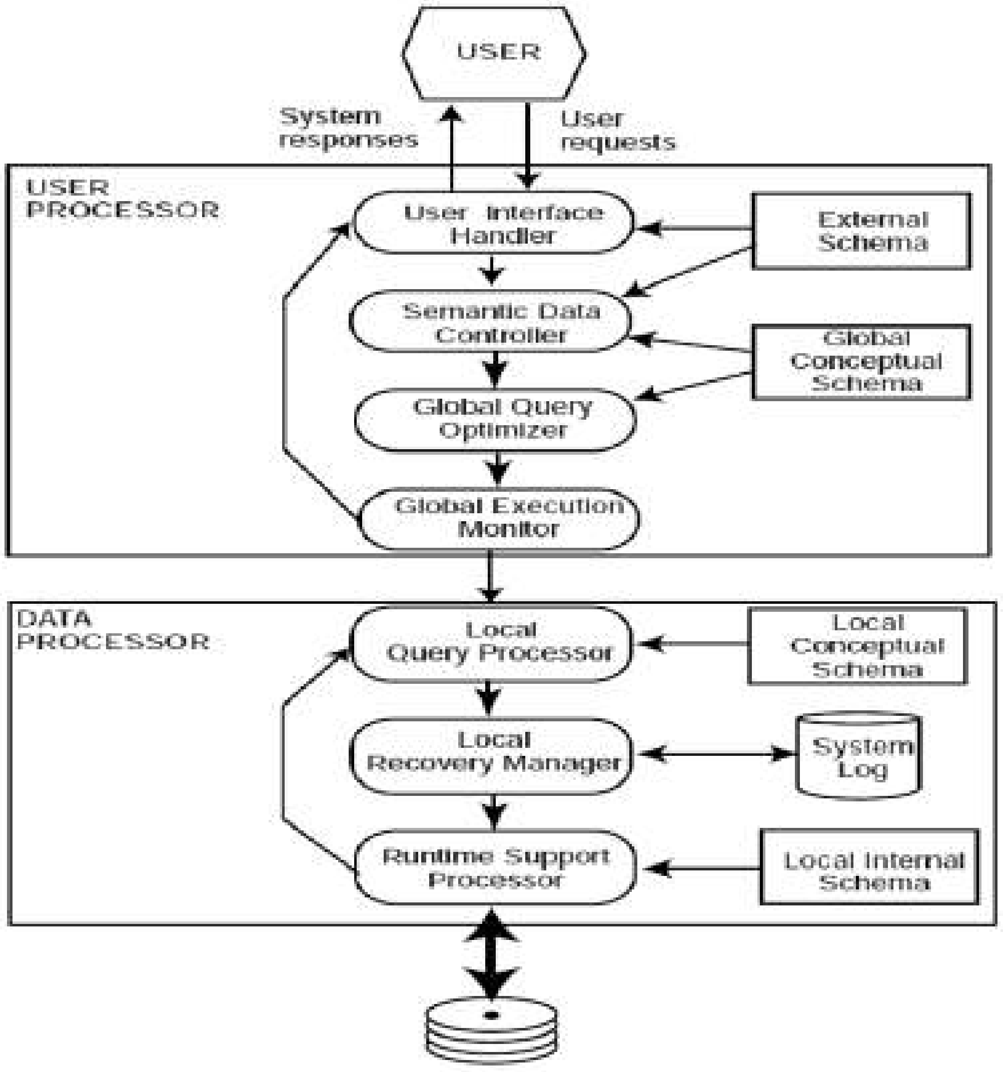


Fig : Components of a Distributed DBMS

One component handles the interaction with users, and another deals with the storage. The first major component, which we call the *user processor*, consists of four elements:

1. The *user interface handler* is responsible for interpreting user commands as they come in, and formatting the result data as it is sent to the user.
2. The *semantic data controller* uses the integrity constraints and authorizations that are defined as part of the global conceptual schema to check if the user query can be processed.
3. The *global query optimizer and decomposer* determines an execution strategy to minimize a cost function, and translates the global queries into local ones using the global and local conceptual schemas as well as the global directory. The global query optimizer is responsible, among other things, for generating the best strategy to execute distributed join operations.
4. The *distributed execution monitor* coordinates the distributed execution of the user request. The execution monitor is also called the *distributed transaction manager*. In executing queries in a distributed fashion, the execution monitors at various sites may, and usually do, communicate with one another.

The second major component of a distributed DBMS is the *data processor* and consists of three elements:

1. The *local query optimizer*, which actually acts as the *access path selector,* is responsible for choosing the best access path[5] to access any data item.
2. The *local recovery manager* is responsible for making sure that the local database remains consistent even when failures occur.
3. The *run-time support processor* physically accesses the database according to the physical commands in the schedule generated by the query optimizer.

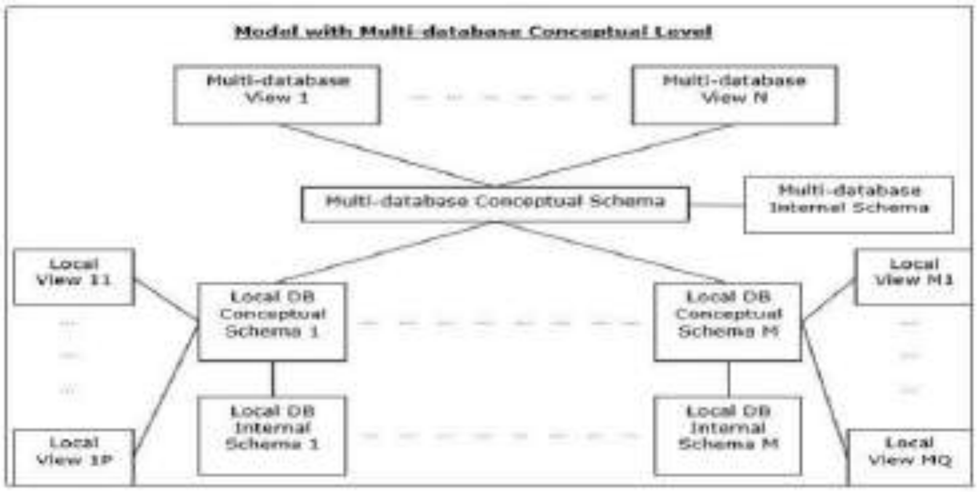
The run-time support processor is the interface to the operating system and contains the *database buffer* (or *cache*) *manager*, which is responsible for maintaining the main memory buffers and managing the data accesses.

Multi - Database System Architecture

- This is an integrated database system formed by a collection of two or more autonomous database systems.
- Multi-DBMS can be expressed through six levels of schemas –
    1. Multi-database View Level − Depicts multiple user views comprising of subsets of the integrated distributed database.
    2. Multi-database Conceptual Level − Depicts integrated multi-database that comprises of global logical multi-database structure definitions.
    3. Multi-database Internal Level − Depicts the data distribution across different sites and multi-database to local data mapping.
    4. Local database View Level − Depicts public view of local data.
    5. Local database Conceptual Level − Depicts local data organization at each site.
    6. Local database Internal Level − Depicts physical data organization at each site.

- There are two design alternatives for multi-DBMS
    - Model with multi-database conceptual level
    - Model without multi-database conceptual level.

Models Using a Global Conceptual Schema

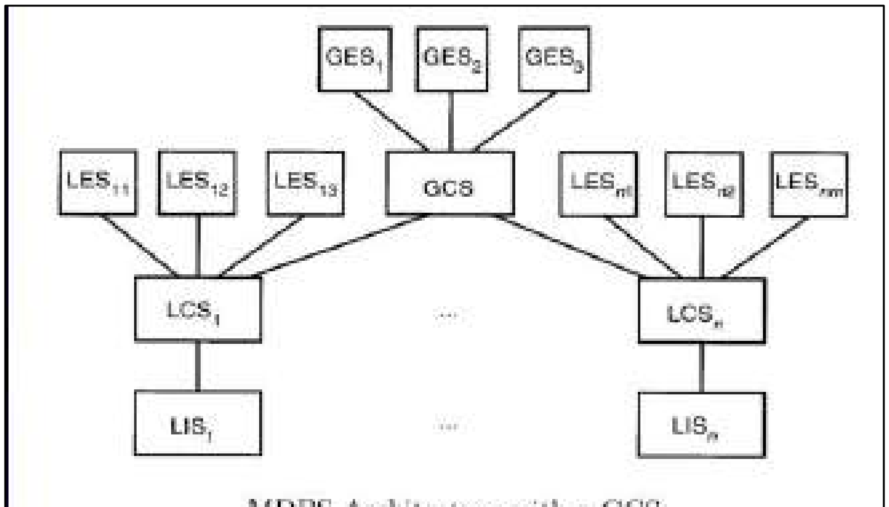

Model with Multi-database Conceptual Level
Multi-database View 1
Multi-database View N
Multi-database Conceptual Schema
Multi-database Internal Schema
Local View 11
Local DB Conceptual Schema 1
Local DB Conceptual Schema M
Local View M1
Local View 1P
Local DB Internal Schema 1
Local DB Internal Schema M
Local View MQ

Model with multi-database conceptual level

- GCS is defined by integrating either the external schemas of local autonomous databases or parts of their local conceptual schema
- Users of a local DBMS define their own views on the local database.
- If heterogeneity exists in the system, then two implementation alternatives exist: unilingual and multilingual
- Unilingual requires the users to utilize possibly different data models and languages
- Basic philosophy of multilingual architecture, is to permit each user to access the global database.

GCS in multi-DBMS

- Mapping is from local conceptual schema to a global schema
- Bottom-up design



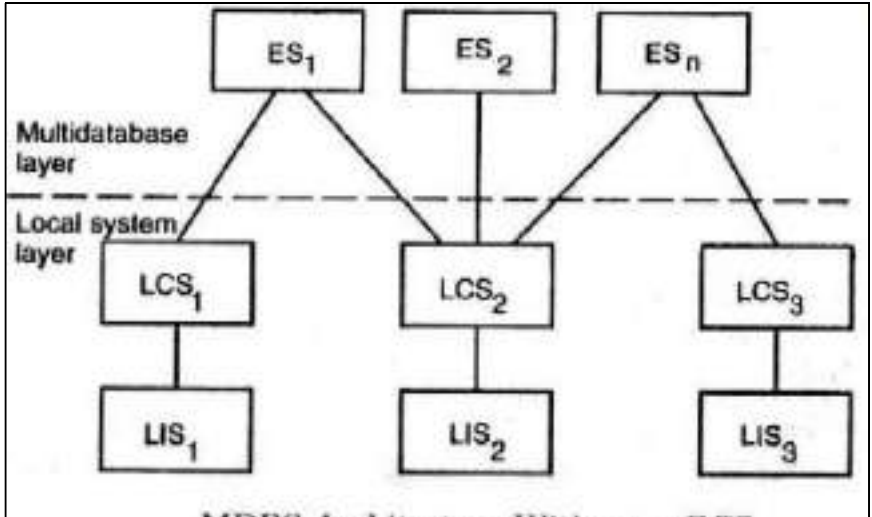
MDBS Architecture with a GCS

Model without multi-database conceptual level.

- Consists of two layers, local system layer and multi database layer.
- Local system layer , present to the multi-database layer the part of their local database they are willing share with users of other database.
- System views are constructed above this layer
- Responsibility of providing access to multiple database is delegated to the mapping between the external schemas and the local conceptual schemas.
- Full-fledged DBMs, exists each of which manages a different database.

GCS in Logically integrated distributed DBMS

- Mapping is from global schema to local conceptual schema
- Top-down procedure

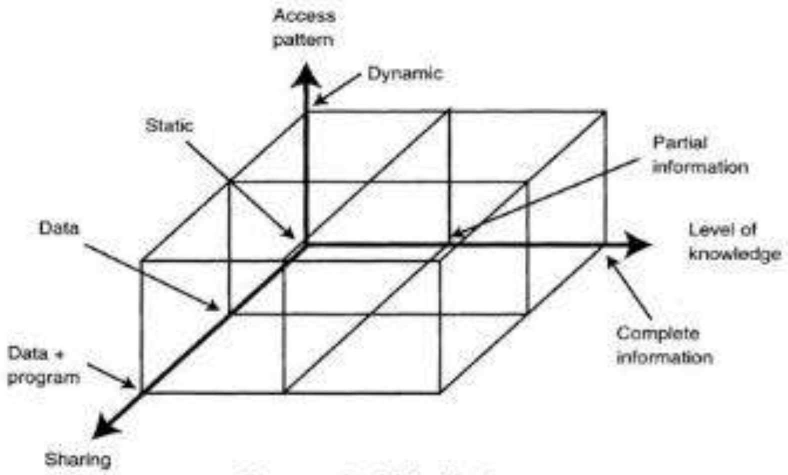


MDBS Architecture Without a GCS

Distributed Database Design

Alternative Design Strategies

- The design of a distributed computer system involves making decisions on the placement of *data* and *programs* across the sites of a computer network, as well as possibly designing the network itself.
- In the case of distributed DBMSs, the distribution of applications involves two things: the distribution of the distributed DBMS software and the distribution of the application programs that run on it.
- The organization of distributed systems can be investigated along three orthogonal dimensions
    1. Level of sharing
    2. Behavior of access patterns
    3. Level of knowledge on access pattern behavior



Framework of Distribution

- Level of sharing
    - ➢ No sharing, each application and data execute at one site
    - ➢ Data sharing, all the programs are replicated at other sites but not the data.

        - Data-plus-program sharing, both data and program can be shared
- Behavior of access patterns
    - Static
      Does not change over time Very easy to manage
    - Dynamic
      Most of the real life applications are dynamic
- Level of knowledge on access pattern behavior.
    - No information
    - Complete information
    - Access patterns can be reasonably predicted
    - No deviations from predictions
    - Partial information
    - Deviations from predictions

- Two major strategies that have been identified for designing distributed databases are
    1. The *Top-Down Approach*
    2. The *Bottom-up approach*

- **The *Top-Down Approach***

  Top- down approach is more suitable for tightly integrated, homogeneous distributed DBMSs.

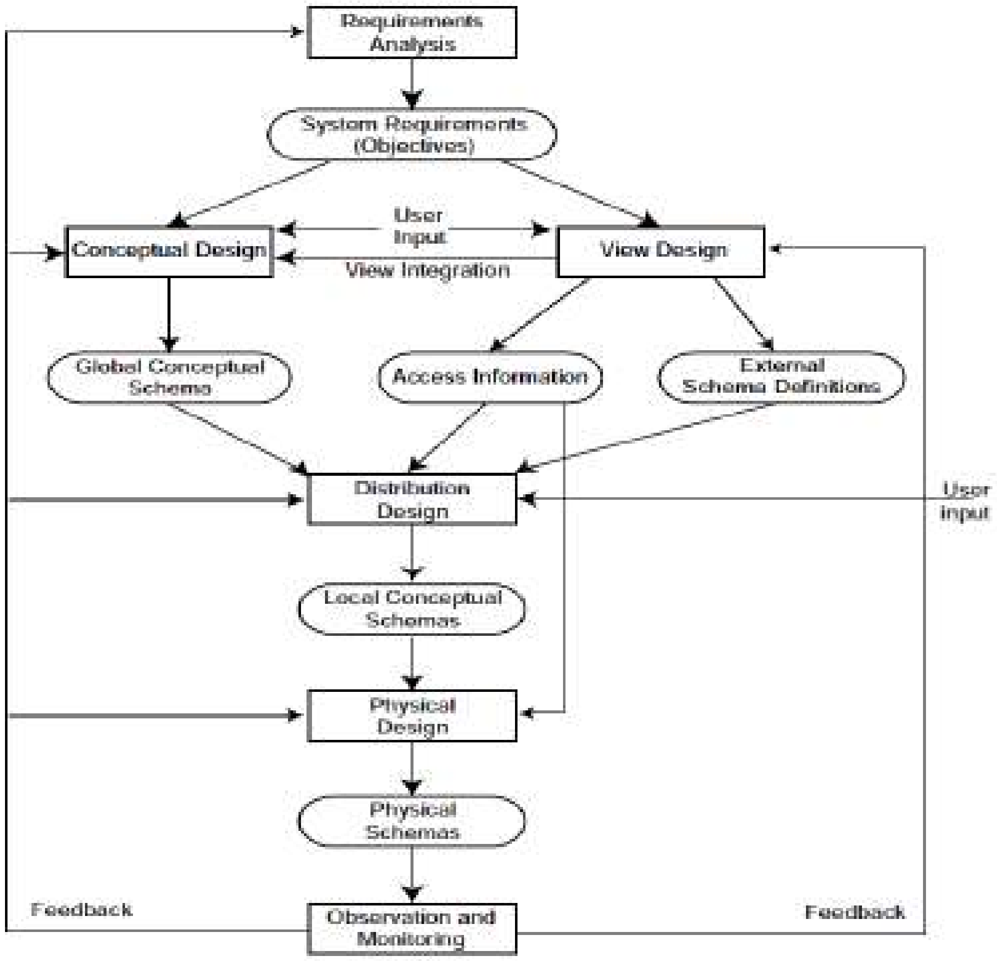


Fig : Top-Down Design Process

- The activity begins with a requirements analysis that defines the environment of the system and "elicits both the data and processing needs of all potential database users".
- The requirements study also specifies where the final system is expected to stand with respect to the objectives of a distributed DBMS.
- These objectives are defined with respect to performance, reliability and availability, economics, and expandability (flexibility).
- The requirements document is input to two parallel activities: view design and conceptual design.
- The ***view design*** activity deals with defining the interfaces for end users.
- The ***conceptual design*** is the process by which the enterprise is examined to determine entity types and relationships among these entities.
- *Entity analysis* is concerned with determining the entities, their attributes, and the relationships among them. *Functional analysis* is concerned with determining the fundamental functions with which the modeled enterprise is involved.
- The results of these two steps need to be cross-referenced to get a better understanding of which functions deal with which entities.
- There is a relationship between the conceptual design and the view design. In one sense, the conceptual design can be interpreted as being an integration of user views.
- This *view integration* activity is very important, the conceptual model should support not only the existing applications, but also future applications. View integration should be used to ensure that entity and relationship requirements for all the views are covered in the conceptual schema.
- In conceptual design and view design activities the user needs to specify the data entities and must determine the applications that will run on the database as well as statistical information about these applications.
- Statistical information includes the specification of the frequency of user applications, the volume of various information, and the like.
- **The Global Conceptual Schema** (GCS) and access pattern information collected as a result of view design are inputs to the *distribution design* step.
- The objective at this stage is to design the local conceptual schemas (LCSs) by distributing the entities over the sites of the distributed system.
- Rather than distributing relations, it is quite common to divide them into sub relations, called *fragments*, which are then distributed.
- Thus, the distribution design activity consists of two steps: *fragmentation* and *allocation*. The reason for separating the distribution design into two steps is to better deal with the complexity of the problem.
- The last step in the design process is the physical design, which maps the local conceptual schemas to the physical storage devices available at the corresponding sites.
- It is well known that design and development activity of any kind is an ongoing process requiring constant monitoring and periodic adjustment and tuning. We have therefore included observation and monitoring as a major activity in this process.
- Note that one does not monitor only the behavior of the database implementation but also the suitability of user views.
- The result is some form of feedback, which may result in backing up to one of the earlier steps in the design.

- **The *Bottom-up approach***
  The Top-down design is a suitable approach when a database system is being designed from scratch.
- Commonly, a number of databases already exist, and the design task involves integrating them into one database.
- The bottom-up approach is suited for this type of environment.
- The starting point is individual local schemas into the global conceptual schemas.
- This type of environment exists primarily in the context of heterogeneous databases.

Distribution Design Issues

- The relations in a database schema are usually decomposed into smaller fragments.
- The following set of interrelated questions covers the entire issue.

1. Why fragment at all?
2. How should we fragment?
3. How much should we fragment?
4. Is there any way to test the correctness of decomposition?
5. How should we allocate?
6. What is the necessary information for fragmentation and allocation?

1.Reasons for Fragmentation

The decomposition of a relation into fragments, each being treated as a unit, permits a number of transactions to execute concurrently.
In addition, the fragmentation of relations typically results in the parallel execution of a single query by dividing it into a set of subqueries that operate on fragments.
Thus fragmentation typically increases the level of concurrency and therefore the system throughput.
This form of concurrency, which we refer to as *intraquery concurrency*

2.Fragmentation Alternatives

Relation instances are essentially tables, so the issue is one of finding alternative ways of dividing a table into smaller ones.
There are clearly two alternatives for this: dividing it *horizontally* or dividing it *vertically*.
Example :

PROJ

PNO	PNAME	BUDGET	LOC
P1	Instrumentation	150000	Montreal
P2	Database Develop.	135000	New York
P3	CAD/CAM	250000	New York
P4	Maintenance	310000	Paris

PROJ1

PNO	PNAME	BUDGET	LOC
P1	Instrumentation	150000	Motreal
P2	Database Develop.	135000	New York

PROJ2

PNO	PNAME	BUDGET	LOC
P3	CAD/CAM	255000	New York
P4	Maintenance	310000	Paris

Fig : Horizontal Partitioning

PROJ1

PNO	BUDGET
P1	150000
P2	135000
P3	250000
P4	310000

PROJ2

PNO	PNAME	LOC
P1	Instrumentation	Montreal
P2	Database Develop.	New York
P3	CAD/CAM	New York
P4	Maintenance	Paris

Fig : Vertical Partitioning

3.*Degree of Fragmentation*

- The extent to which the database should be fragmented is an important decision that affects the performance of query execution
- The degree of fragmentation goes from one extreme, that is, not to fragment at all, to the other extreme, to fragment to the level of individual tuples (in the case of horizontal fragmentation) or to the level of individual attributes (in the case of vertical fragmentation).

4. Correctness Rules of Fragmentation

We will enforce the following three rules during fragmentation, which, together, ensure that the database does not undergo semantic change during fragmentation.

1. ***Completeness.*** If a relation instance R is decomposed into fragments $F_R = R_1, R_2, \ldots, R_n$, each data item that can be found in R can also be found in one or more of R_i's. This property, which is identical to the *lossless de-composition* property of normalization.

2. ***Reconstruction***. If a relation R is decomposed into fragments $F_R = \{R_1, R_2, \ldots, R_n\}$, it should be possible to define a relational operator Q such that

   $$R = \mathrm{Q} R_i,\ \forall R_i \in F_R$$

3. ***Disjointness***. If a relation R is horizontally decomposed into fragments $F_R = \{R_1, R_2, \ldots, R_n\}$ and data item d_i is in R_j, it is not in any other fragment R_k ($k \ \ j$). This criterion ensures that the horizontal fragments are disjoint.

5.Allocation Alternatives

Assuming that the database is fragmented properly, one has to decide on the allocation of the fragments to various sites on the network.
When data are allocated, it may either be replicated or maintained as a single copy.
The reasons for replication are reliability and efficiency of read-only queries.
Hence the decision regarding replication is a trade-off that depends on the ratio of the read-only queries to the update queries.

6.Information Requirements

The information needed for distribution design can be divided into four categories: database information, application information, communication network information, and computer system information

Fragmentation

- Fragmentation is the task of dividing a table into a set of smaller tables. The subsets of the table are called fragments. Fragmentation can be of three types: horizontal, vertical, and hybrid (combination of horizontal and vertical).
- Fragmentation should be done in a way so that the original table can be reconstructed from the fragments. This is needed so that the original table can be reconstructed from the fragments whenever required. This requirement is called "reconstructiveness."

  Advantages

  1. Permits a number of transactions to executed concurrently
  2. Results in parallel execution of a single query
  3. Increases level of concurrency, also referred to as, intra query concurrency
  4. Increased System throughput.
  5. Since data is stored close to the site of usage, efficiency of the database system is increased.
  6. Local query optimization techniques are sufficient for most queries since data is locally available.
  7. Since irrelevant data is not available at the sites, security and privacy of the database system can be maintained.

  Disadvantages

  1. Applications whose views are defined on more than one fragment may suffer performance degradation, if applications have conflicting requirements.

2. Simple tasks like checking for dependencies, would result in chasing after data in a number of sites
3. When data from different fragments are required, the access speeds may be very high.
4. In case of recursive fragmentations, the job of reconstruction will need expensive techniques.
5. Lack of back-up copies of data in different sites may render the database ineffective in case of failure of a site.

Horizontal Fragmentation

- Horizontal fragmentation partitions a relation along its tuples. Thus each fragment has a subset of the tuples of the relation.
- There are two versions of horizontal partitioning: primary and derived.
- *Primary horizontal fragmentation* of a relation is performed using predicates that are defined on that relation.
- *Derived horizontal fragmentation* is the partitioning of a relation that results from predicates being defined on another relation.
- Each horizontal fragment must have all columns of the original base table.

Primary horizontal fragmentation

- Primary horizontal fragmentation is defined by a selection operation on the owner relation of a database schema.
- Given relation R_i, its horizontal fragments are given by $R_i = \sigma F_i(R)$, $1 <= i <= w$
  F_i selection formula used to obtain fragment R_i
- The example mentioned in slide 20, can be represented by using the above formula as $Emp1 = \sigma Sal <= 20K\ (Emp)$

  $Emp2 = \sigma Sal > 20K\ (Emp)$
- For example, in the student schema, if the details of all students of Computer Science Course needs to be maintained at the School of Computer Science, then the designer will horizontally fragment the database as follows −

```
CREATE COMP_STD AS SELECT * FROM STUDENT  WHERE COURSE = "Computer Science";
```

Derived Horizontal Fragmentation

- Defined on a member relation of a link according to a selection operation specified on its owner.
- Link between the owner and the member relations is defined as equi-join
- An equi-join can be implemented by means of semijoins.

- Given a link L where owner (L) = S and member (L) = R, the derived horizontal fragments of R are defined as

  $R_i = R \, \alpha \, S_i, \quad 1 <= I <= w$

  Where, $S_i = \sigma \, F_i \, (S)$

  w is the max number of fragments that will be defined on

  F_i is the formula using which the primary horizontal fragment S_i is defined

Vertical Fragmentation

- In vertical fragmentation, the fields or columns of a table are grouped into fragments.

- In order to maintain reconstructiveness, each fragment should contain the primary key field(s) of the table. Vertical fragmentation can be used to enforce privacy of data.

Grouping

- Starts by assigning each attribute to one fragment
- At each step, joins some of the fragments until some criteria is satisfied.
- Results in overlapping fragments

Splitting

- Starts with a relation and decides on beneficial partitioning based on the access behavior of applications to the attributes
- Fits more naturally within the top-down design
- Generates non-overlapping fragments
- For example, let us consider that a University database keeps records of all registered students in a Student table having the following schema.

STUDENT

Regd_No	Name	Course	Address	Semester	Fees	Marks

Now, the fees details are maintained in the accounts section. In this case, the designer will fragment

```
CREATE TABLE STD_FEES AS  SELECT  Regd_No,  Fees FROM  STUDENT;
```

Hybrid Fragmentation

- In hybrid fragmentation, a combination of horizontal and vertical fragmentation techniques are used.

- This is the most flexible fragmentation technique since it generates fragments with minimal extraneous information. However, reconstruction of the original table is often an expensive task.

- A vertical fragmentation may be followed by a horizontal one, or vice versa, producing a tree-structured partitioning.
- Since the two types of partitioning strategies are applied one after the other, this alternative is called *hybrid* fragmentation.
- It has also been named *mixed* fragmentation or *nested* fragmentation.

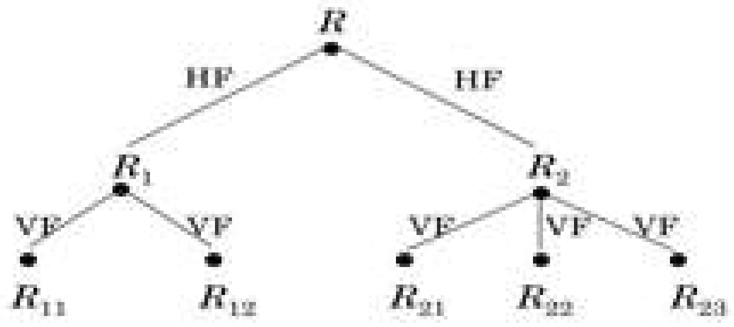


Fig : Hybrid Fragmentation

- For example, to reconstruct the original global relation in case of hybrid fragmentation, one starts at the leaves of the partitioning tree and moves upward by performing joins and unions.
- The fragmentation is complete if the intermediate and leaf fragments are complete. Similarly, disjointness is guaranteed if intermediate and leaf fragments are disjoint.

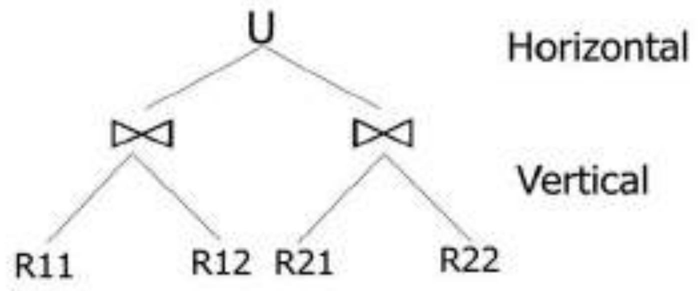


Fig : Reconstruction of Hybrid Fragmentation

Allocation

- Allocation is the process of requesting access to a dataset. If we allocate a dataset that exists, then the system allows you to open the dataset and vice versa.
- The allocation of the resources across the nodes or placing individual files of a computer network is a big task.

Allocation Problem

Assume that there are a set of fragments $F = \{F_1, F_2, \ldots, F_n\}$ and a distributed system consisting of sites $S = \{S_1, S_2, \ldots, S_m\}$ on which a set of applications $Q = \{q_1, q_2, \ldots, q_q\}$ is running.

The allocation problem involves finding the “optimal” distribution of F to S.

The optimality can be defined with respect to two measures:

1. ***Minimal cost*.**
   - The cost function consists of the cost of storing each F_i at a site S_j, the cost of querying F_i at site S_j, the cost of updating F_i at all sites where it is stored, and the cost of data communication.
   - The allocation problem, then, attempts to find an allocation scheme that minimizes a combined cost function.
2. ***Performance*.**
   - The allocation strategy is designed to maintain a performance metric.
   - Two well-known ones are to minimize the response time and to maximize the system throughput at each site.

To separate the traditional problem of file allocation from the fragment allocation in distributed database design, we refer to the former as the ***file allocation problem*** **(FAP)** and to the latter as the ***database allocation problem*** **(DAP).**

Information Requirements

It is at the allocation stage that we need the quantitative data about the database, the applications that run on it, the communication network, the processing capabilities, and storage limitations of each site on the network. We will discuss each of these in detail.

Database Information

To perform horizontal fragmentation, we defined the selectivity of minterms. We now need to extend that definition to fragments, and define the selectivity of a fragment F_j with respect to query q_i. This is the number of tuples of F_j that need to be accessed in order to process q_i. This value will be denoted as $sel_i(F_j)$.

Another piece of necessary information on the database fragments is their size.
The size of a fragment F_j is given by $size(F_j) = card(F_j) * length(F_j)$

where $length(F_j)$ is the length (in bytes) of a tuple of fragment F_j.

Application Information

- Most of the application-related information is already compiled during the fragmentation activity, but a few more are required by the allocation model.
- The two important measures are the number of read accesses that a query q_i makes to a fragment F_j during its execution (denoted as RR_{ij}), and its counterpart for the update accesses (UR_{ij}). These may, for example, count the number of block accesses required by the query.
- We also need to define two matrices UM and RM, with elements u_{ij} and r_{ij}, respectively,
- which are specified as follows:

$$u_{ij} = \begin{cases} 1 \text{ if query } q_i \text{ updates fragment } F_j \\ 0 \text{ otherwise} \end{cases}$$

$$r_{ij} = \begin{cases} 1 \text{ if query } q_i \text{ retrieves from fragment } F_j \\ 0 \text{ otherwise} \end{cases}$$

A vector O of values $o(i)$ is also defined, where $o(i)$ specifies the originating site of query q_i. Finally, to define the response-time constraint, the maximum allowable response time of each application should be specified.

Site Information

- For each computer site, we need to know its storage and processing capacity. Obvi- ously, these values can be computed by means of elaborate functions or by simple estimates.
- The unit cost of storing data at site S_k will be denoted as USC_k. There is also a need to specify a cost measure LPC_k as the cost of processing one unit of work at site S_k.
- The work unit should be identical to that of the *RR* and *UR* measures.

Network Information

- In our model we assume the existence of a simple network where the cost of communication is defined in terms of one frame of data.
- Thus g_{ij} denotes the communication cost per frame between sites S_i and S_j. To enable the calculation of the number of messages, we use $fsize$ as the size (in bytes) of one frame.
- There is no question that there are more elaborate network models which take into consideration the channel capacities, distances between sites, protocol overhead, and so on.

Allocation Model

- We discuss an allocation model that attempts to minimize the total cost of processing and storage while trying to meet certain response time restrictions.
- The model we use has the following form:

  min(Total Cost)

subject to

- Response-time constraint
- Storage constraint
- Processing constraint.

DISTRIBUTED DATABASES (R22CSE3146)

III B.Tech I Semester (Information Technology)

UNIT – II

Query processing and decomposition: Query processing objectives, characterization of query processors, layers of query processing, query decomposition, localization of distributed data.
Distributed query Optimization: Query optimization, centralized query optimization, distributed query optimization algorithms.

Query : A "Query " refers to the action of retrieving data from the database. Query is expressed by using high level language. (or)

A query is a request for data or information from a database table or combination of tables. In the context of queries in a database, it can be either a select query or an action query. A select query is a data retrieval query, while an action query asks for additional operations on the data, such as insertion, updating or deletion. Most formal queries are written in SQL (Structured Query Language).

A query can either be a request for data results from your database or for action on the data, or for both. A query can give you an answer to a simple question, perform calculations, combine data from different tables, add, change, or delete data from a database.

Query Processor : A Query Processor is a module in the DBMS that performs the tasks to process, to optimize and to generate execution strategy for a high level query.

Query Processing : The process of extracting data from a database is called query processing. It requires several steps to retrieve the data from the database during query processing. The actions involved actions are:
1. Parsing and translation
2. Optimization
3. Evaluation

Distributed Query Processing

- A query is a request for data that involves transmitting data between computers in a network. The process of answering queries on data stored at multiple sites in a computer network is called distributed query processing.
- Distributed query processing is the procedure of answering queries (which means mainly read operations on large data sets) in a distributed environment where data is managed at multiple sites in a computer network.
- Query processing involves the transformation of a high-level query (e.g., formulated in SQL) into a query execution plan (consisting of lower-level query operators in some variation of relational algebra) as well as the execution of this plan.

- The goal of the transformation is to produce a plan which is equivalent to the original query (returning the same result) and efficient, i.e., to minimize resource consumption like total costs or response time.

- Query Processing is a translation of high-level queries into low-level expression.
- It is a step wise process that can be used at the physical level of the file system, query optimization and actual execution of the query to get the result.
- It requires the basic concepts of relational algebra and file structure.
- It refers to the range of activities that are involved in extracting data from the database.
- It includes translation of queries in high-level database languages into expressions that can be implemented at the physical level of the file system.
- In query processing, we will actually understand how these queries are processed and how they ae optimized.

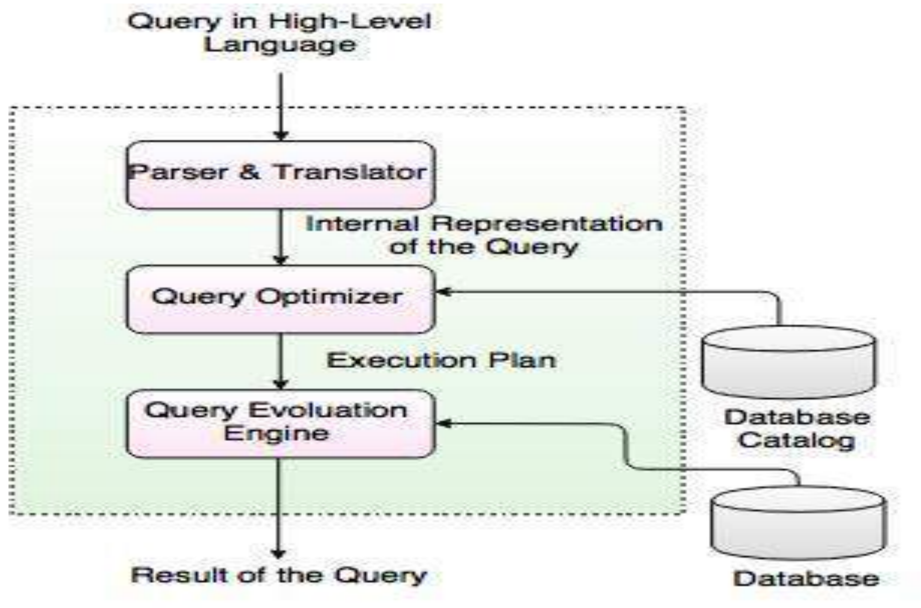


Fig. Query Processing

In the above diagram,

- The first step is to transform the query into a standard form.
- A query is translated into SQL and into a relational algebraic expression. During this process, Parser checks the syntax and verifies the relations and the attributes which are used in the query.
- The second step is Query Optimizer. In this, it transforms the query into equivalent expressions that are more efficient to execute.
- The third step is Query evaluation. It executes the above query execution plan and returns the result.

Decomposition

- Decomposition is the process of breaking up a single relation into two or more sub-relations.
- This process can help remove redundancy, inconsistencies, and anomalies from a database.
- Decomposition is also dependency preserving and lossless.
- In distributed query processing, decomposition is one of the functions performed by the first three layers of the process.
- These layers map the input query into an optimized distributed query execution plan.
- In query decomposition, a distributed calculus query is mapped into an algebraic query on global relations.

Query Processing Objectives

- The objective of query processing in a distributed context is to transform a high-level query on a distributed database, which is seen as a single database by the users, into an efficient execution strategy expressed in a low-level language on local databases.
- We assume that the high-level language is relational calculus, while the low-level language is an extension of relational algebra with communication operators.
- An important aspect of query processing is query optimization. Because many execution strategies are correct transformations of the same high-level query, the one that optimizes (minimizes) resource consumption should be retained.
- A good measure of resource consumption is the ***total cost***. *The* Total cost is the sum of all times incurred in processing the operators of the query at various sites and in inter site communication.
- Another good measure is the ***response time*** of the query, which is the time elapsed for executing the query. Since operators can be executed in parallel at different sites, the response time of a query may be significantly less than its total cost.
- In a distributed database system, the total cost to be minimized includes : 1. CPU, 2. I/O and 3. Communication costs.

1. **The CPU cost** is incurred when performing operators on data in main memory.
2. **The I/O cost** is the time necessary for disk accesses. This cost can be minimized by reducing the number of disk accesses through fast access methods to the data and efficient use of main memory (buffer management).
3. **The communication cost** is the time needed for exchanging data between sites participated in the execution of the query. This cost is incurred in processing the messages (formatting/de-formatting), and in transmitting the data on the communication network.

Characterization of Query Processors

- It is quite difficult to evaluate and compare query processors in the context of both centralized systems and distributed systems because they may differ in many aspects.
- Here are some important characteristics of query processors that can be used as a basis for comparison.
  1. Languages
  2. Types of Optimization
  3. Optimization Timing
  4. Statistics

5. Decision Sites
6. Exploitation of the Network Topology
7. Exploitation of Replicated Fragments
8. Use of Semijoins

- The first four characteristics hold for both centralized and distributed query processors.
- The next four characteristics are particular to distributed query processors in tightly-integrated distributed DBMSs.

1. Languages

Query processing must perform efficient mapping from the input language to the output language.

2. Types of Optimization

- Query optimization aims at choosing the “best” point in the solution space of all possible execution strategies.
- An immediate method for query optimization is to search the solution space, exhaustively predict the cost of each strategy, and select the strategy with minimum cost.
- The problem is that the solution space can be large; that is, there may be many equivalent strategies, even with a small number of relations.
- Therefore, an “exhaustive” search approach is often used whereby (almost) all possible execution strategies are considered.

3. Optimization Timing

- A query may be optimized at different times relative to the actual time of query execution.
- Optimization can be done *statically* before executing the query or *dynamically* as the query is executed.
- Static query optimization is done at query compilation time.
- Dynamic query optimization proceeds at query execution time.
- The main advantage over static query optimization is that the actual sizes of intermediate relations are available to the query processor, thereby minimizing the probability of a bad choice.

4. Statistics

- The effectiveness of query optimization relies on *statistics* on the database.
- Dynamic query optimization requires statistics in order to choose which operators shouldbe done first.
- Static query optimization is even more demanding since the size of intermediate relations must also be estimated based on statistical information.

5.Decision Sites

When static optimization is used, either a single site or several sites may participatein the selection of the strategy to be applied for answering the query.

- Most systems use the centralized decision approach, in which a single site generates the strategy.
- However, the decision process could be distributed among various sites participating in the elaboration of the best strategy.
- The centralized approach is simpler but requires knowledge of the entire distributed database, while the distributed approach requires only local information.
- Hybrid approaches where one site makes the major decisions and other sites can make local decisions are also frequent.

6.Exploitation of the Network Topology

- The network topology is generally exploited by the distributed query processor.
- With wide area networks (WAN), the cost function to be minimized can be restricted to the data communication cost, which is considered to be the dominant factor.
- With local area networks (LAN), communication costs are comparable to I/O costs. Therefore, it is reasonable for the distributed query processor to increase parallel execution at the expense of communication cost.

7.Exploitation of Replicated Fragments

- A distributed relation is usually divided into relation fragments.
- Distributed queries expressed on global relations are mapped into queries on physical fragments of relations by translating relations into fragments.
- We call this process *localization* because its main function is to localize the data involved in the query

8.Use of Semijoins

- The semijoin operator has the important property of reducing the size of the operand relation.
- A semijoin is particularly useful for improving the processing of distributed join operators as it reduces the size of data exchanged between sites.
- The early distributed DBMSs, such as SDD-1 which were designed for slow wide area networks, make extensive use of semijoins. Some later systems, such as R*, assume faster networks and do not employ semijoins.

Layers of Query Processing

- There are Four main layers are involved in distributed query processing.

    1. Query Decomposition
    2. Data Localization
    3. Global Query Optimization
    4. Distributed Query Execution

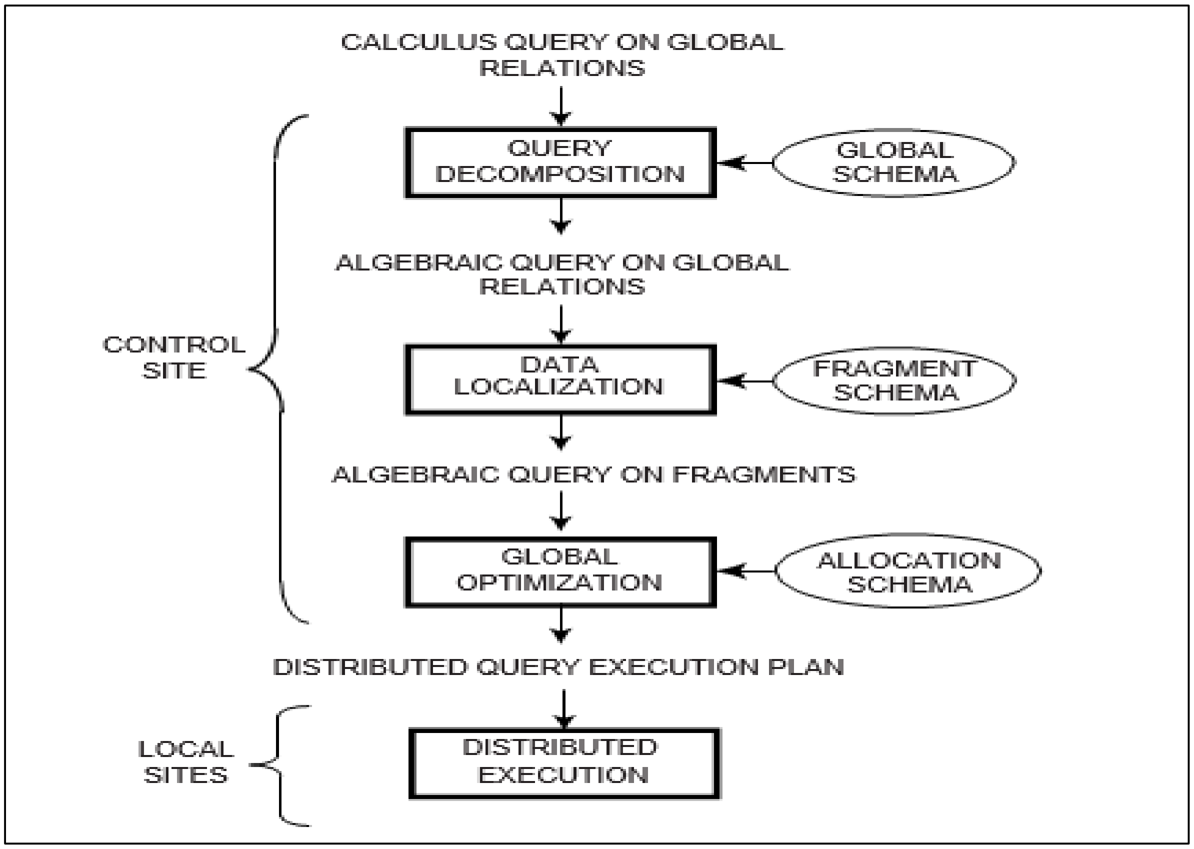


Fig : Generic Layering Scheme for Distributed Query Processing

- The input is a query on global data expressed in relational calculus.
- This query is posed on global (distributed) relations, meaning that data distribution is hidden.
- Four main layers are involved in distributed query processing.
- The first three layers map the input query into an optimized distributed query execution plan.
- They perform the functions of *query decomposition*, *data localization*, and *global query optimization*.
- Query decomposition and data localization correspond to query rewriting.
- The fourth layer performs *distributed query execution* by executing the plan and returns the answer to the query.
- It is done by the local sites and the control site.

1. Query Decomposition

- The first layer decomposes the calculus query into an algebraic query on global relations.
- The information needed for this transformation is found in the global conceptual schema describing the global relations.
- Query decomposition can be viewed as four successive steps.

   i. Normalization
   ii. Analyzation
   iii. *Simplified* ***(***eliminate redundant)
   iv. *Restructured / rewriting*

Query decomposition is the first phase of query processing that transforms a relational calculus query into a relational algebra query. Both input and output queries refer to global relations, without knowledge of the distribution of data. Therefore, query decomposition is the same for centralized and distributed systems.

i. Normalization

- It is the goal of normalization to transform the query to a normalized form to facilitate further processing.
- There are two possible normal forms for the predicate,
- The *conjunctive normal form* is a conjunction (∧ predicate)
- The *disjunctive normal form* is a disjunctions (∨ predicates)
- one giving precedence to the AND (∧) and the other to the OR (∨).

$$(p_{11} \vee p_{12} \vee \cdots \vee p_{1n}) \wedge \cdots \wedge (p_{m1} \vee p_{m2} \vee \cdots \vee p_{mn})$$

where p_{ij} is a simple predicate. A qualification in *disjunctive normal form*, on the other hand, is as follows:

$$(p_{11} \wedge p_{12} \wedge \cdots \wedge p_{1n}) \vee \cdots \vee (p_{m1} \wedge p_{m2} \wedge \cdots \wedge p_{mn})$$

The transformation of the quantifier-free predicate is straightforward using the well-known equivalence rules for logical operations (∧, ∨, and ¬):

1. $p_1 \wedge p_2 \Leftrightarrow p_2 \wedge p_1$
2. $p_1 \vee p_2 \Leftrightarrow p_2 \vee p_1$
3. $\neg(p_1 \vee p_2) \Leftrightarrow \neg p_1 \wedge \neg p_2$

Example: Let us consider the following query on the engineering database that we have been referring to: “Find the names of employees who have been working on project P1 for 12 or 24 months”

The query expressed in SQL is

```
SELECT ENAME FROM       EMP, ASG
WHERE  EMP.ENO = ASG.ENO
AND     ASG.PNO = "P1"
AND     DUR = 12 OR DUR = 24
```

The qualification in conjunctive normal form is

$$\text{EMP.ENO} = \text{ASG.ENO} \wedge \text{ASG.PNO} = \text{“P1”} \wedge (\text{DUR} = 12 \vee \text{DUR} = 24)$$

while the qualification in disjunctive normal form is

$$(\text{EMP.ENO} = \text{ASG.ENO} \wedge \text{ASG.PNO} = \text{“P1”} \wedge \text{DUR} = 12) \vee$$
$$(\text{EMP.ENO} = \text{ASG.ENO} \wedge \text{ASG.PNO} = \text{“P1”} \wedge \text{DUR} = 24)$$

In the latter form, treating the two conjunctions independently may lead to redundant work if common subexpressions are not eliminated.

ii. Analysis

- Query analysis enables rejection of normalized queries for which further processing is either impossible or unnecessary. The main reasons for rejection are that the query is *type incorrect* or *semantically incorrect*.
- When one of these cases is detected, the query is simply returned to the user with an explanation. Otherwise, query processing is continued.

Example: The following SQL query on the engineering database is type incorrect for two reasons.

```
SELECT E# FROM      EMP   WHERE  ENAME > 200
```

First, attribute E# is not declared in the schema.

Second, the operation ">200" is incompatible with the type string of ENAME.

iii. Elimination of Redundancy

- Here, the query should be simplified means eliminating the redundant predicates.

Example :

```
SELECT TITLE FROM EMP

WHERE  (NOT (TITLE = "Programmer")
AND (TITLE = "Programmer"  OR   TITLE = "Elect. Eng.")
AND NOT (TITLE = "Elect. Eng.")) OR ENAME = "J. Doe"
```

can be simplified using the previous rules to become

```
SELECT TITLEFROM  EMP  WHERE  ENAME = "J. Doe"
```

The simplification proceeds as follows. Let p_1 be TITLE = "Programmer", p_2 be TITLE = "Elect. Eng.", and p_3 be ENAME = "J. Doe". The query qualification is

$$(\neg p_1 \wedge (p_1 \vee p_2) \wedge \neg p_2) \vee p_3$$

The disjunctive normal form for this qualification is

$$(\neg p_1 \wedge ((p_1 \wedge \neg p_2) \vee (p_2 \wedge \neg p_2))) \vee p_3$$

iv. Rewriting

- The last step of query decomposition rewrites the query in relational algebra.
- To represent the relational algebra query graphically by an *operator tree*.
- An operator tree is a tree in which a leaf node is a relation stored in the database, and a non-leaf node is an intermediate relation produced by a relational algebra operator. The sequence of operations is directed from the leaves to the root, which represents the answer to the query.

Example : The query "Find the names of employees other than J. Doe who worked on the CAD/CAM project for either one or two years" whose SQL expression is

```
SELECT ENAME
FROM PROJ, ASG, EMP WHERE
ASG.ENO = EMP.ENO
AND ASG.PNO = PROJ.PNO
AND ENAME!= "J. Doe"
AND PROJ.PNAME = "CAD/CAM"
AND (DUR = 12 OR DUR = 24)
```

- Select operation (σ) : It selects the tuples that satisfy the given predicate from a relation.
- Project operation (π) : It projects the columns that satisfy the given predicate.
- Cartesian Product (x) : Combines information of two different relations into one.

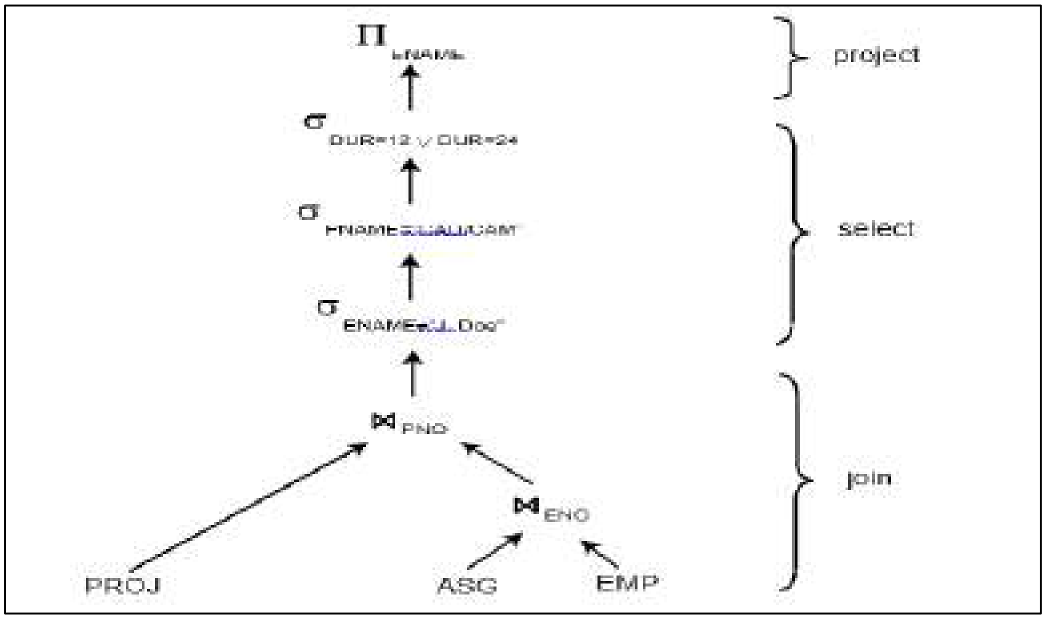


Fig : Example of Operator Tree

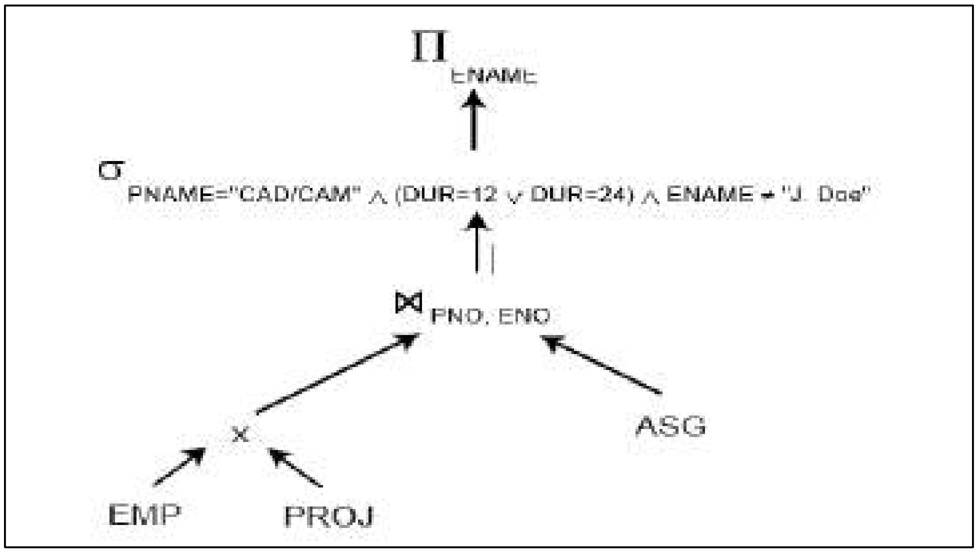


Fig : Equivalent Operator Tree

2.Data Localization

- The input to the second layer is an algebraic query on global relations. The main role of the second layer is to localize the query's data using data distribution information in the fragment schema.
- This layer determines which fragments are involved in the query and transforms the distributed query into a query on fragments.
- A global relation can be reconstructed by applying the fragmentation rules, and then deriving a program, called a localization program, of relational algebra operators, which then act on fragments.

3.Global Query Optimization

- The input to the third layer is an algebraic query on fragments. The goal of query optimization is to find an execution strategy for the query which is close to optimal.
- The previous layers have already optimized the query, for example, by eliminating redundant expressions. However, this optimization is independent of fragment characteristics such as fragment allocation and cardinalities.
- Query optimization consists of finding the "best" ordering of operators in the query, including communication operators that minimize a cost function.
- The output of the query optimization layer is a optimized algebraic query with communication operators included on fragments. It is typically represented and saved (for future executions) as a distributed query execution plan.

4.Distributed Query Execution

- The last layer is performed by all the sites having fragments involved in the query.
- Each sub query executing at one site, called a local query, is then optimized using the local schema of the site and executed.

Localization of Distributed Data

- The localization layer translates an algebraic query on global relations into an algebraic query expressed on physical fragments. Localization uses information stored in the fragment schema.
- The general techniques for decomposing and restructuring queries expressed in relational calculus.
- These global techniques apply to both centralized and distributed DBMSs and do not take into account the distribution of data. This is the role of the localization layer.
- the localization layer translates an algebraic query on global relations into an algebraic query expressed on physical fragments. Localization uses information stored in the fragment schema.
- Fragmentation is defined through fragmentation rules, which can be expressed as relational queries.
- a global relation can be reconstructed by applying the reconstruction (or reverse fragmentation) rules and deriving a relational algebra program whose operands are the fragments. We call this a *localization program*.

- In each type of fragmentation, we present *reduction techniques* that generate simpler and optimized queries.
  - ❖ Reduction for Primary Horizontal Fragmentation
    - Reduction with Selection
    - Reduction with Join
  - ❖ Reduction for Vertical Fragmentation
  - ❖ Reduction for Derived Fragmentation
  - ❖ Reduction for Hybrid Fragmentation

- **The Horizontal fragmentation** function distributes a relation based on selection predicates.

Example : Relation EMP(ENO, ENAME, TITLE) of Figure 2.3 can be split into three horizontal fragments EMP_1, EMP_2, and EMP_3, defined as follows:

$$EMP_1 = \sigma_{ENO \leq "E3"}(EMP)$$

$$EMP2 = \sigma "E3" < ENO \leq "E6"(EMP)$$

$$EMP_3 = \sigma_{ENO > "E6"}(EMP)$$

The localization program for an horizontally fragmented relation is the union of the fragments. In our example we have

$$EMP = EMP_1 \cup EMP_2 \cup EMP_3$$

- **The vertical fragmentation** function distributes a relation based on projection attributes. Since the reconstruction operator for vertical fragmentation is the join, the localization program for a vertically fragmented relation consists of the join of the fragments on the common attribute.

Example : Relation EMP can be divided into two vertical fragments where the key attribute ENO is duplicated:

$$EMP1 = \Pi\ ENO, ENAME(EMP)$$

$$EMP2 = \Pi\ ENO,TITLE(EMP)$$

The localization program is

$$EMP = EMP_1\ 0_{ENO}\ EMP_2$$

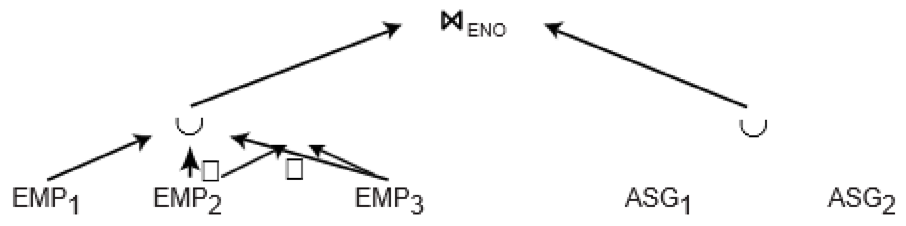
- **The Derived horizontal fragmentation** is another way of distributing two relations so that the joint processing of select and join is improved.

Example : Given a one-to-many relationship from EMP to ASG, relation ASG(ENO, PNO, RESP, DUR) can be indirectly fragmented according to the following rules:

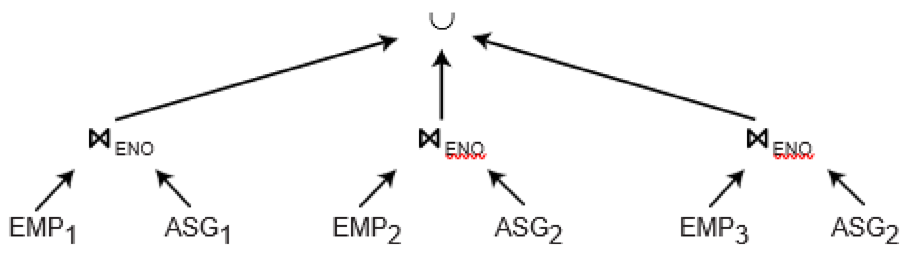
$$ASG_1 = ASG\ n_{ENO}\ EMP_1 ASG_2 = ASG\ n_{ENO}\ EMP_2$$

The localization program for a horizontally fragmented relation is the union of the fragments. In our example, we have

$$ASG = ASG_1 \cup ASG_{2.}$$



(a) Localized query



(b) Reduced query

- **The Hybrid fragmentation** is obtained by combining the fragmentation (horizontal & vertical) functions. The goal of hybrid fragmentation is to support, efficiently, queries involving projection, selection, and join.

  hybrid fragmentation based on selection-projection will make selection only, or projection only, less efficient than with horizontal fragmentation (or vertical fragmentation). The localization program for a hybrid fragmented relation uses unions and joins of fragments.

Example : Here is an example of hybrid fragmentation of relation EMP:

$$EMP1 = \sigma_{ENO \leq "E4"}(\Pi_{ENO,ENAME}(EMP))$$

$$EMP2 = \sigma_{ENO > "E4"}(\Pi_{ENO,ENAME}(EMP))$$

$$EMP3 = \Pi_{ENO,TITLE}(EMP)$$

In our example, the localization program is

$$EMP = (EMP_1 \cup EMP_2)\ 0_{ENO}\ EMP_3$$

Queries on hybrid fragments can be reduced by combining the rules used, respectively, in primary horizontal, vertical, and derived horizontal fragmentation.

These rules can be summarized as follows:

1. Remove empty relations generated by contradicting selections on horizontal fragments.
2. Remove useless relations generated by projections on vertical fragments.
3. Distribute joins over unions in order to isolate and remove useless joins.

Distributed Query Optimization

Query Optimization

- Query optimization refers to the process of producing a query execution plan (QEP) which represents an execution strategy for the query.
- This QEP minimizes an objective cost function. A query optimizer, the software module that performs query optimization.
- usually it consisting of three components:
    - a search space,
    - a cost model, and
    - a search strategy
- The ***search space*** is the set of alternative execution plans that represent the input query.
- These plans are equivalent, in the sense that they yield the same result, but they differ in the execution order of operations and the way these operations are implemented, and therefore in their performance.
- The ***cost model*** predicts the cost of a given execution plan. To be accurate, the cost model must have good knowledge about the distributed execution environment.
- The ***search strategy*** explores the search space and selects the best plan, using the cost model. It defines which plans are examined and in which order.
- The details of the environment (centralized versus distributed) are captured by the search space and the cost model.
- Query execution plans are typically abstracted by means of operator trees, which define the order in which the operations are executed.
- They are enriched with additional information, such as the best algorithm chosen for each operation.
- For a given query, the search space can thus be defined as the set of equivalent operator trees that can be produced using transformation rules.
- To characterize query optimizers, it is useful to concentrate on *join trees*, which are operator trees whose operators are join or Cartesian product.
- This is because permutations of the join order have the most important effect on performance of relational queries.

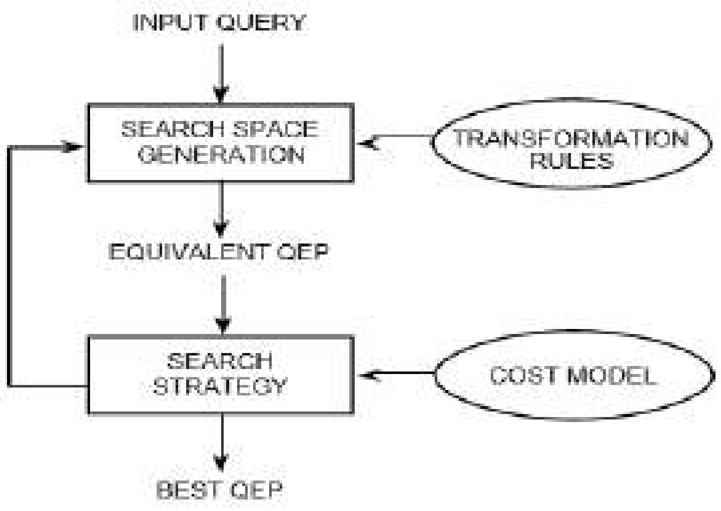


Fig : Query Optimization Process

Example : Consider the following query:

```
SELECT ENAME,  RESP FROM EMP, ASG, PROJ
WHERE EMP.ENO=ASG.ENO
AND ASG.PNO=PROJ.PNO
```

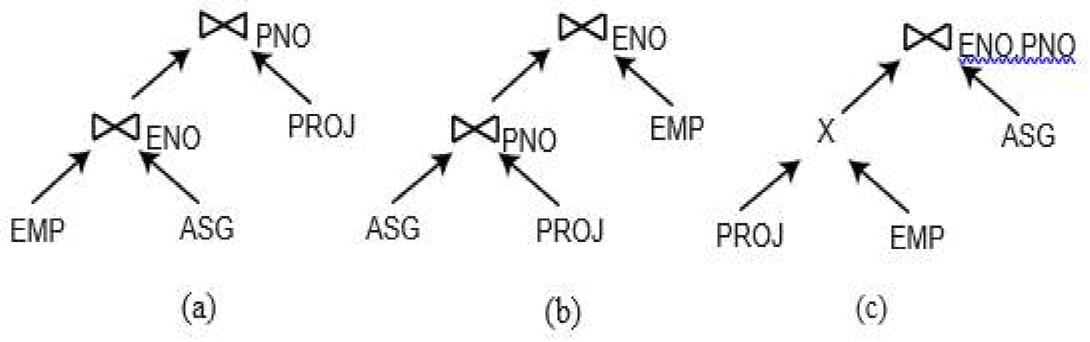


Fig : Equivalent Join Trees

- Each of these join trees can be assigned a cost based on the estimated cost of each operator.
- Join tree (c) which starts with a Cartesian product may have a much higher cost than the other join trees.
- If a query is large we use restrictions.
- The first restriction is to use heuristics. The most common heuristic is to perform selection and projection when accessing base relations.
- Another common heuristic is to avoid Cartesian products that are not required by the query.
- operator tree (c) would not be part of the search space considered by the optimizer.
- Another important restriction is with respect to the shape of the join tree.
- Two kinds of join trees are usually distinguished: linear versus bushy trees.

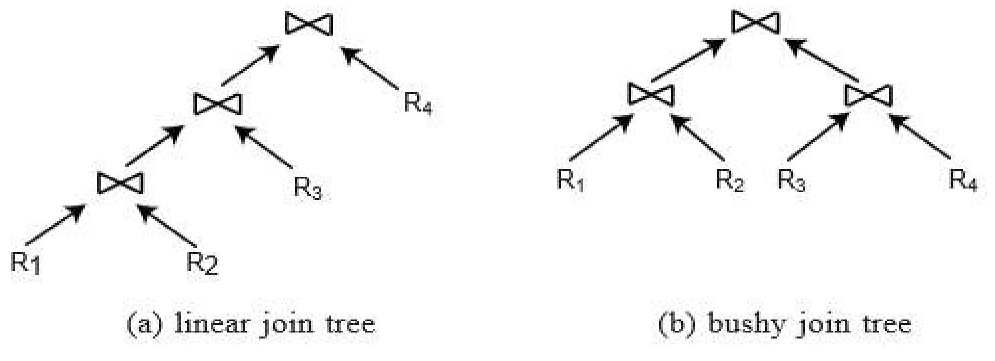


Fig : The Two Major Shapes of Join Trees

- A *linear tree* is a tree such that at least one operand of each operator node is a base relation.
- A *bushy tree* is more general and may have operators with no base relations as operands (i.e., both operands are intermediate relations).
- By considering only linear trees, the size of the search space is reduced to $O(2^N)$.
- In a distributed environment, bushy trees are useful in exhibiting parallelism.
- For example, in join tree operations R_1 *0* R_2 and R_3 *0* R_4 can be done in parallel.

Cost Functions

- The cost of a distributed execution strategy can be expressed with respect to either the total time or the response time.
- The cost is in terms of execution time, so a cost function represents the execution time of a query.
- The total time is the sum of all time (also referred to as cost) components, while the response time is the elapsed time from the initiation to the completion of the query.
- A general formula for determining the total time can be specified as follows :

  - $$\textit{Total time} = T_{CPU} * \#insts + T_{I/O} * \#I/Os + T_{MSG} * \#msgs + T_{TR} * \#bytes$$

- The two first components measure the local processing time.
- T_{CPU} is the time of a CPU instruction and $T_{I/O}$ is the time of a disk I/O. The communication time is depicted by the two last components.
- T_{MSG} is the fixed time of initiating and receiving a message.
- T_{TR} is the time it takes to transmit a data unit from one site to another.
- The data unit is given here in terms of bytes (*#bytes* is the sum of the sizes of all messages), but could be in different units (e.g., packets).
- A typical assumption is that T_{TR} is constant. This might not be true for wide area networks, where some sites are farther away than others. However, this assumption greatly simplifies query optimization.
- Thus the communication time of transferring *#bytes* of data from one site to another is assumed to be a linear function of *#bytes*:

$$CT(\#bytes) = T_{MSG} + T_{TR} * \#bytes$$

- Costs are generally expressed in terms of time units, which in turn, can be translated into other units (e.g., dollars).

Centralized Query Optimization

- There are three reasons to understanding distributed query optimization.
- First, a distributed query is translated into local queries, each of which is processed in a centralized way.
- Second, distributed query optimization techniques are often extensions of the techniques for centralized systems.
- Finally, centralized query optimization is a simpler problem
- The minimization of communication costs makes distributed query optimization more complex.
- The optimization timing, which can be dynamic, static or hybrid, is a good basis for classifying query optimization techniques

Dynamic Query Optimization

- Dynamic query optimization combines the two phases of query decomposition and optimization with execution.

- The QEP is dynamically constructed by the query optimizer which makes calls to the DBMS execution engine for executing the query's operations.
- Thus, there is no need for a cost model.
- Dynamic query optimization, done at run-time
- The most popular dynamic query optimization algorithm is that of INGRES

Static Query Optimization

- With static query optimization, there is a clear separation between the generation of the QEP at compile-time and its execution by the DBMS execution engine.
- Thus, an accurate cost model is key to predict the costs of candidate QEPs.
- Static query optimization, done at compilation time
- The most popular static query optimization algorithm is that of System R.

Hybrid Query Optimization

- Hybrid query optimization attempts to provide the advantages of static query optimization while avoiding the issues generated by inaccurate estimates.
- The approach is basically static, but further optimization decisions may take place at run time.

Distributed Query Optimization Algorithms

Dynamic Approach :

INGRES Algorithm

- The algorithm of Distributed **INGRES** illustrates the **dynamic approach**.
- It recursively breaks a query into smaller pieces
- The objective function of the algorithm is to minimize a combination of both the communication time and the response time. The algorithm also takes advantage of fragmentation, but only horizontal fragmentation is handled for simplicity.
- INGRES is a popular relational DB system and it has a distributed version whose optimization algorithms are extensions of the centralized versions.
- It uses a dynamic query optimization algorithm that recursively breaks-up a calculus query into smaller pieces.
- INGRES combines calculus-algebra decomposition and optimization.
- A query is first decomposed into a subsequence of queries having a unique relation in common.
- Then each mono relation query is processed by a **One - Variable Query Processor (OVQP).**
- The OVQP optimizes the access to a single-relation by selecting the best access method to that relation.
- (Eg : index, sequential scan).
- Given an N - relation query q, the INGRES query processor decompose q into n subqueries
- q1->q2->…..->qi.
- This decomposition uses two basic techniques: detachments and substitutions.
- Each qi is a mono relation query; the output of qi is consumed by qi+1. Detachment is used first.
- There is a processor that can efficiently process mono-relation queries optimizes each query independently for the access to single relation.

Static Approach :

R* Algorithm

- It uses **static optimization algorithm** based on an exhaustive search of solution space.
- It uses database statistics to determine the total cost involved.
- Under this strategy all the candidate tree are given particular cost and lowest cost tree is retained.
- The total number of trees possible is determined using dynamic programming under which those paths, which are not optimal are not considered and also which includes Cartesian product is not considered.
- The disadvantage of this algorithm is that it is costly and does not deal with fragments.

Semijoin Algorithm:

Hill-Climbing query optimization algorithm

- Refinements of an initial feasible solution are recursively computed until no more cost improvements can be made
- Semijoins, data replication, and fragmentation are not used
- Devised for wide area point-to-point networks
- The first distributed query processing algorithm

SDD-1 Algorithm

The SDD-1 query optimization algorithm improves the Hill-Climbing algorithm in a number of directions:

– Semijoins are considered

– More elaborate statistics

– Initial plan is selected better

– Post-optimization step is introduced

Hill Climbing using Semijoin

Initialization

Step 1: In the execution strategy (call it ES), include all the local processing

Step 2: Reflect the effects of local processing on the database profile

Step 3: Construct a set of beneficial semijoin operations (BS) as follows :

```
BS = Ø
For each semijoin SJi
```

BS ← BS ∪ SJi if cost(SJi) < benefit(SJi)

Iterative Process

Step 4: Remove the most beneficial SJi from BS and append it to ES

Step 5: Modify the database profile accordingly

Step 6: Modify BS appropriately

– compute new benefit/cost values

– check if any new semijoin needs to be included in BS

Step 7: If BS ≠ Ø, go back to Step 4

Assembly Site Selection

Step 8: Find the site where the largest amount of data resides and select it as the assembly site

Postprocessing

Step 9: For each Ri at the assembly site, find the semijoins of $\bowtie$ the type Ri Rj.

where the total cost of ES without this semijoin is smaller than the cost with it and remove the semijoin from ES.

Step 10: Permute the order of semijoins, if doing so would improve the total cost of ES.

UNIT – III

Transaction Management : Definition, properties of transaction, types of transactions, Distributed concurrency control: serializability, concurrency control mechanisms & algorithms, time - stamped & optimistic concurrency control Algorithms, deadlock Management.

Definition of a Transaction

- Transaction consists of a set of operations that performs a single logical unit of work in database environment.
- The *transaction* is a basic unit of consistent and reliable computing.
- A Transaction takes a database, performs an action on it and generate a new version of the database, causing state transition.
- This is similar to what a query does, except that if the database was consistent before the execution of the transaction, we can now guarantee that it will be consistent at the end of its execution regardless of the fact that

  (1) The transaction may have been executed concurrently with others, and
  (2) Failures may have occurred during its execution.

- A Transaction is considered to be made up of a sequence of read and write operations on the database together with computation steps.
- A Transaction may be thought of a program with embedded database access queries.

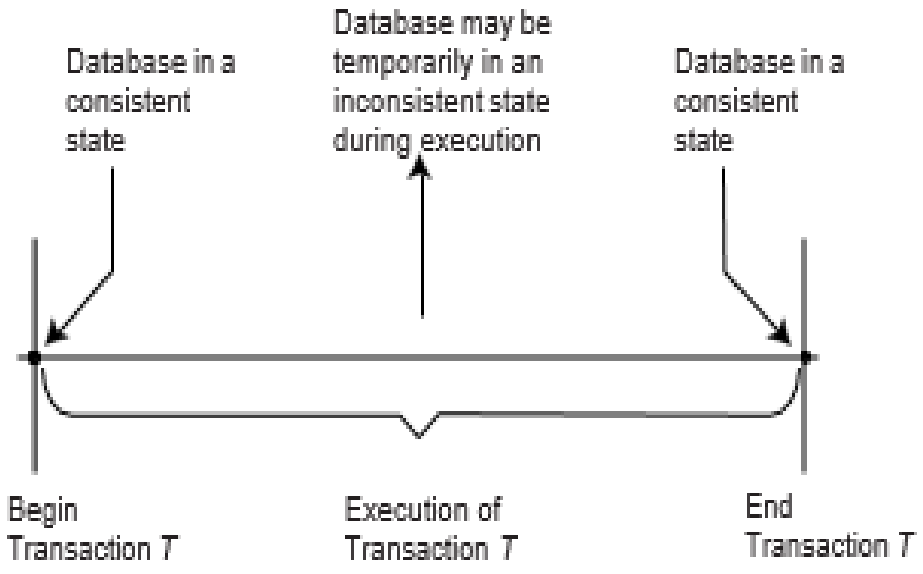

Database in a consistent state
Database may be temporarily in an inconsistent state during execution
Database in a consistent state
Begin Transaction T
Execution of Transaction T
End Transaction T

Fig : Transaction Model

Example : Airline Reservation System

FLIGHT relation that records the data about each flight

CUST relation for the customers who book flights

FC relation indicating which customers are on what flights.

- Let us also assume that the relation definitions are as follows (where the underlined attributes constitute the keys):
  FLIGHT(FNO, DATE, SRC, DEST, STSOLD, CAP)
  CUST(CNAME, ADDR, BAL)
  FC(FNO, DATE, CNAME, SPECIAL)
- FNO is the flight number, DATE denotes the flight date, SRC and DEST indicate the source and destination for the flight, STSOLD indicates the number of seats that have been sold on that flight, CAP denotes the passenger capacity on the flight, CNAME indicates the customer name whose address is stored in ADDR and whose account balance is in BAL, and SPECIAL corresponds to any special requests that the customer may have for a booking.
- Let us consider a simplified version of a typical reservation application, where a travel agent enters the flight number, the date, and a customer name, and asks for a reservation. The transaction to perform this function can be implemented as follows, where database accesses are specified in embedded SQL notation:

```
Begin transaction Reservation
begin
    input(flight no, date, customer name);              (1)
      EXEC SQL UPDATE FLIGHT                            (2)
           SET    STSOLD = STSOLD + 1
      WHERE FNO = flight no AND DATE = date;
      EXEC SQL INSERT                                   (3)
           INTO FC(FNO,DATE,CNAME,SPECIAL)
            VALUES (flight no, date, customer name, null);
    output("reservation completed")                     (4)
 end.
end.
```

Explanation : First a point about notation. Even though we use embedded SQL, we do not follow its syntax very strictly. The lowercase terms are the program variables; the uppercase terms denote database relations and attributes as well as the SQL statements. Numeric constants are used as they are, whereas character constants are enclosed in quotes. Keywords of the host language are written in boldface, and *null* is a keyword for the null string.

Line (1) is to input the flight number, the date, and the customer name.

Line (2) updates the number of sold seats on the requested flight by one.

Line (3) inserts a tuple into the FC relation.

Here we assume that the customer is an old one, so it is not necessary to have an insertion into the CUST relation, creating a record for the client.

The keyword *null* in line (3) indicates that the customer has no special requests on this flight. Finally,

Line (4) reports the result of the transaction to the agent's terminal.

Characterization of Transactions

- Transactions read and write some data. This has been used as the basis for characterizing a transaction.
- The data items that a transaction reads are said to constitute its *read set* (*RS*).
- The data items that a transaction writes are said to constitute its *write set* (*WS*).
- The read set and write set of a transaction need not be mutually exclusive.
- The union of the read set and write set of a transaction constitutes its *base set* ($BS = RS \cup WS$).

Example :

RS[Reservation] = {FLIGHT.STSOLD, FLIGHT.CAP}

WS[Reservation] = {FLIGHT.STSOLD, FC.FNO, FC.DATE, FC.CNAME, FC.SPECIAL}

BS[Reservation] = {FLIGHT.STSOLD, FLIGHT.CAP, FC.FNO, FC.DATE, FC.CNAME, FC.SPECIAL}

Formalization of the Transaction Concept

- The meaning of a transaction should be intuitively clear.
- To reason about transactions and about the correctness of the management algorithms, it is necessary to define the concept formally.

Example : Consider a simple transaction *T* that consists of the following steps:

```
Read($x$)
Read($y$)
$x \leftarrow x + y$
Write($x$)
Commit
```

- One advantage of defining a transaction as a partial order is its correspondence to a **directed acyclic graph (DAG).**
- Thus a transaction can be specified as a DAG whose vertices are the operations of a transaction and whose arcs indicate the ordering relationship between a given pair of operations.

Example: The transaction can be represented as a DAG as depicted in Figure:

$$T = \{R(x), R(y), W(x), C\}$$

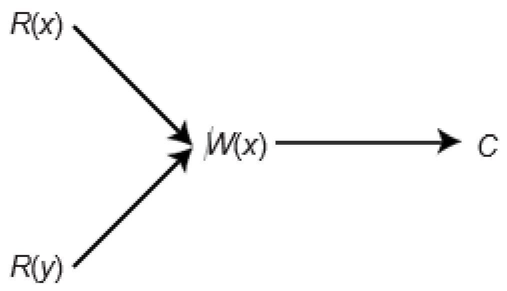
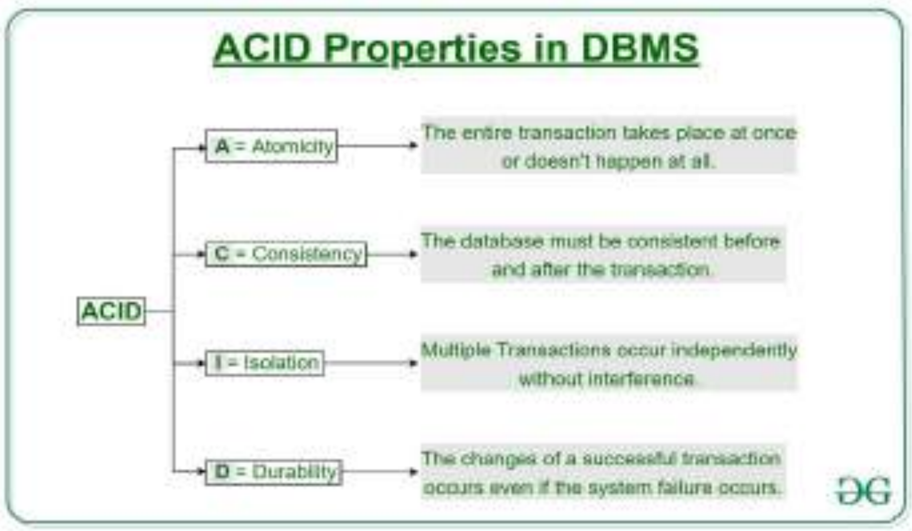


Fig: DAG Representation of a Transaction

Properties of Transactions

The consistency and reliability aspects of transactions are due to four properties: Together, these are commonly referred to as the ACID properties of transactions.

(1) atomicity,
(2) consistency,
(3) isolation, and
(4) durability.



ACID Properties in DBMS

ACID

A = Atomicity → The entire transaction takes place at once or doesn't happen at all.

C = Consistency → The database must be consistent before and after the transaction.

I = Isolation → Multiple Transactions occur independently without interference.

D = Durability → The changes of a successful transaction occurs even if the system failure occurs.

1. Atomicity

- *Atomicity* refers to the fact that a transaction is treated as a unit of operation.
- Refers either all the transaction's actions are completed, or none of them are.
- This is also known as the "**all-or-nothing property**".
- Atomicity requires that if the execution of a transaction is interrupted by any sort of failure, the DBMS will be responsible for determining what to do with the transaction upon recovery from the failure.
- There are, of course, two possible courses of action:
    - it can either be terminated by completing the remaining actions,

  - or it can be terminated by undoing all the actions that have already been executed.
- One can generally talk about two types of failures.
  - A transaction itself may fail due to input data errors, deadlocks, or other factors. In these cases either the transaction aborts itself, or the DBMS may abort it while handling deadlocks.
    Maintaining transaction atomicity in the presence of this type of failure is commonly called the ***transaction recovery***.
  - The second type of failure is caused by system crashes, such as media failures, processor failures, communication link breakages, power outages, and so on.
    Ensuring transaction atomicity in the presence of system crashes is called ***crash recovery*****.**
- An important difference between the two types of failures is that during some types of system crashes, the information in volatile storage may be lost or inaccessible. Both types of recovery are parts of the reliability issue.

2. Consistency

- Each user is responsible to ensure that their transaction would leave the database in a consistent state.
- Transaction consistency is the responsibility of the user not the DBMS.
- The *consistency* of a transaction is simply its correctness.
- A Transaction is a correct program that maps one consistent database state to another.
- This classification groups databases into four levels of consistency.
- Then, based on the concept of dirty data, the four levels are defined as follows:

- Degree 3: Transaction *T* sees *degree 3 consistency* if:
  1. *T* does not overwrite dirty data of other transactions.
  2. *T* does not commit any writes until it completes all its writes [i.e., until the end of transaction (EOT)].
  3. *T* does not read dirty data from other transactions.
  4. Other transactions do not dirty any data read by *T* before *T* completes.

- Degree 2: Transaction *T* sees *degree 2 consistency* if:
  1. *T* does not overwrite dirty data of other transactions.
  2. *T* does not commit any writes before EOT.
  3. *T* does not read dirty data from other transactions.

- Degree 1: Transaction *T* sees *degree 1 consistency* if:
  1. *T* does not overwrite dirty data of other transactions.
  2. *T* does not commit any writes before EOT.

- Degree 0: Transaction *T* sees *degree 0 consistency* if:
  1. *T* does not overwrite dirty data of other transactions.”

- Of course, it is true that a higher degree of consistency encompasses all the lower degrees.
- The point in defining multiple levels of consistency is to provide application programmers the flexibility to define transactions that operate at different levels.
- Consequently, while some transactions operate at Degree 3 consistency level, others may operate at lower levels and may see, for example, dirty data.

3. Isolation

- *Isolation* is the property of transactions that requires each transaction to see a consistent database at all times.
- In other words, an executing transaction cannot reveal its results to other concurrent transactions before its commitment.
- There are a number of reasons for insisting on isolation.
- One has to do with maintaining the interconsistency of transactions.
- If two concurrent transactions access a data item that is being updated by one of them, it is not possible to guarantee that the second will read the correct value.

Example : Consider the following two concurrent transactions (T_1 and T_2), both of which access data item x. Assume that the value of x before they start executing is 50.

T_1: Read(x)	T_2: Read(x)
$x \leftarrow x+1$	$x \leftarrow x+1$
Write(x)	Write(x)
Commit	Commit

The following is one possible sequence of execution of the actions of these transactions:

T_1: Read(x)
T_1: $x \leftarrow x+1$
T_1: Write(x)
T_1: Commit
T_2: Read(x)
T_2: $x \leftarrow x+1$
T_2: Write(x)
T_2: Commit

In this case, there are no problems; transactions T_1 and T_2 are executed one after the other and transaction T_2 reads 51 as the value of x. Note that if, instead, T_2 executes before T_1, T_2 reads 51 as the value of x. So, if T_1 and T_2 are executed one after the other (regardless of the order), the second transaction will read 51 as the value of x and x will have 52 as its value at the end of execution of these two transactions. However, since transactions are executing concurrently,

The following execution sequence is also possible:

T_1: Read(x)
T_1: $x \leftarrow x + 1$
T_2: Read(x)
T_1: Write(x)
T_2: $x \leftarrow x + 1$
T_2: Write(x)
T_1: Commit
T_2: Commit

- In this case, transaction T_2 reads 50 as the value of x. This is incorrect since T_2 reads x while its value is being changed from 50 to 51. Furthermore, the value of x is 51 at the end of execution of T_1 and T_2 since T_2's Write will overwrite T_1's Write.
- Ensuring isolation by not permitting incomplete results to be seen by other transactions, as the previous example shows, solves the ***lost updates*** problem.
- This type of isolation has been called ***cursor stability***. In the example above, the second execution sequence resulted in the effects of T_1 being lost.
- A second reason for isolation is *cascading aborts*. If a transaction permits others to see its incomplete results before committing and then decides to abort, any transaction that has read its incomplete values will have to abort as well. This chain can easily grow and impose considerable overhead on the DBMS.
- As we move up the hierarchy of consistency levels, there is more isolation among transactions.
- Degree 0 provides very little isolation other than preventing lost updates. Transactions commit write operations before the entire transaction is completed (and committed), if an abort occurs after some writes are committed to disk, the updates to data items that have been committed will need to be undone. Since at this level other transactions are allowed to read the dirty data, it may be necessary to abort them as well.
- Degree 2 consistency avoids cascading aborts.
- Degree 3 provides full isolation which forces one of the conflicting transactions to wait until the other one terminates. Such execution sequences are called *strict.*

- ANSI, as part of the SQL2 (also known as SQL-92) standard specification, has defined a set of isolation levels. SQL isolation levels are defined on the basis of what ANSI call *phenomena* which are situations that can occur if proper isolation is not maintained. Three phenomena are specified:

Dirty Read:

- As defined earlier, dirty data refer to data items whose values have been modified by a transaction that has not yet committed.
- Consider the case where transaction T_1 modifies a data item value, which is then read by another transaction T_2 before T_1 performs a Commit or Abort.

- In case T_1 aborts, T_2 has read a value which never exists in the database.
- A precise specification of this phenomenon is as follows (where subscripts indicate the transaction identifiers)

$\ldots, W_1(x), \ldots, R_2(x), \ldots, C_1(\text{or } A_1), \ldots, C_2(\text{or } A_2)$

or

$\ldots, W_1(x), \ldots, R_2(x), \ldots, C_2(\text{or } A_2), \ldots, C_1(\text{or } A_1)$

Non-repeatable or Fuzzy read:

- Transaction T_1 reads a data item value. Another transaction T_2 then modifies or deletes that data item and commits.
- If T_1 then attempts to reread the data item, it either reads a different value or it can't find the data item at all; thus two reads within the same transaction T_1 return different results.
- A precise specification of this phenomenon is as follows:

$\ldots, R_1(x), \ldots, W_2(x), \ldots, C_1(\text{or } A_1), \ldots, C_2(\text{or } A_2)$

or

$\ldots, R_1(x), \ldots, W_2(x), \ldots, C_2(\text{or } A_2), \ldots, C_1(\text{or } A_1)$

Phantom:

- The phantom condition that was defined earlier occurs when T_1 does a search with a predicate and T_2 inserts new tuples that satisfy the predicate.
- Again, the precise specification of this phenomenon is (where P is the search predicate)

$\ldots, R_1(P), \ldots, W_2(y \text{ in } P), \ldots, C_1(\text{or } A_1), \ldots, C_2(\text{or } A_2)$

or

$\ldots, R_1(P), \ldots, W_2(y \text{ in } P), \ldots, C_2(\text{ or } A_2), \ldots, C_1(\text{or } A_1)$

- Based on these phenomena, the isolation levels are defined as follows. The objective of defining multiple isolation levels is the same as defining multiple consistency levels.
- **Read uncommitted :** For transactions operating at this level all three phenomena are possible.
- **Read committed :** Fuzzy reads and phantoms are possible, but dirty reads are not.
- **Repeatable read :** Only phantoms are possible.
- **Anomaly serializable :** None of the phenomena are possible.
- ANSI SQL standard uses the term "serializable" rather than "anomaly serializable."
- One non-serializable isolation level that is commonly implemented in commercial products is *snapshot isolation* Snapshot isolation provides repeatable reads, but not serializable isolation.
- Each transaction "sees" a snapshot of the database when it starts and its reads and writes are performed on this snapshot – thus the writes are not visible to other transactions and it does not see the writes of other transactions.

4. Durability

- *Durability* refers to that property of transactions which ensures that once a transaction commits, its results are permanent and cannot be erased from the database.
- Therefore, the DBMS ensures that the results of a transaction will survive subsequent system failures.
- The durability property brings forth the issue of ***database recovery***, that is, how to recover the database to a consistent state where all the committed actions are reflected.

Types of Transactions

Transactions have been classified according to a number of criteria.

1. First one is the duration of transactions (Timing).
2. Second is with respect to the organization of the read and write actions
3. Third one is according to their structure.

First one criterion is the duration of transactions

- Accordingly, transactions may be classified as *online* or *batch*. These two classes are also called *short-life* and *long-life* transactions.
- **Online transactions** are characterized by very short execution/response times (typically, on the order of a couple of seconds) and by access to a relatively small portion of the database.
- This class of transactions probably covers a large majority of current transaction applications.
- Examples : Banking transactions and Airline reservation transactions.
- **Batch transactions** take longer to execute (response time being measured in minutes, hours, or even days) and access a larger portion of the database.
- Typical applications that might require batch transactions are design databases.
- Examples : statistical applications, report generation, complex queries, and image processing. Along this dimension, one can also define a *conversational* transaction, which is executed by interacting with the user issuing it.

Second is with respect to the organization of the read and write actions

- Another classification that has been proposed is with respect to the organization of the read and write actions.
- Examples : considered so far intermix their read and write actions without any specific ordering. We call this type of transactions *general*.
- If the transactions are restricted so that all the read actions are performed before any write action, the transaction is called a *two-step* transaction.
- Similarly, if the transaction is restricted so that a data item has to be read before it can be updated (written), the corresponding class is called *restricted* (or *read-before-write*). If a transaction is both two- step and restricted, it is called a *restricted two-step* transaction.

Third one is according to their structure.

- Finally, there is the *action* model of transactions, which consists of the restricted class with the further restriction that each ⟨read, write⟩ pair be executed atomically.
- ***Example:*** The following are some examples of the above-mentioned models. We omit the declaration and commit commands.

General:

$T_1 : \{R(x), R(y), W(y), R(z), W(x), W(z), W(w), C\}$

Two-step:

$T_2 : \{R(x), R(y), R(z), W(x), W(z), W(y), W(w), C\}$

Restricted:

$T_3 : \{R(x), R(y), W(y), R(z), W(x), W(z), R(w), W(w), C\}$

Note that T_3 has to read w before writing.

Two-step restricted:

$T_4 : \{R(x), R(y), R(z), R(w), W(x), W(z), W(y), W(w), C\}$

Action:

$T_5 : \{[R(x), W(x)], [R(y), W(y)], [R(z), W(z)], [R(w), W(w)], C\}$

Note that each pair of actions within square brackets is executed atomically.

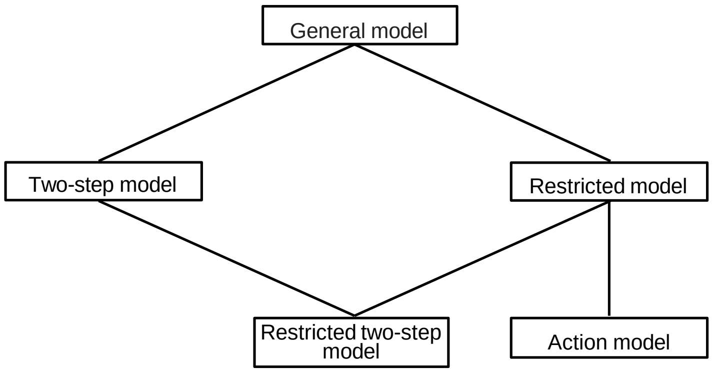


Fig : Various transaction Models

Transactions can also be classified according to their structure. We distinguish three broad categories in increasing complexity:

1. *Flat transactions,*
2. *Nested transactions,*
3. *Workflow models.*

Flat Transactions

- A client makes requests to multiple servers in a flat transaction
- Flat transactions have a single start point (**Begin transaction**) and a single termination point (**End transaction**).
- Transaction T, for example, is a flat transaction that performs operations on objects in servers X, Y, and Z.
  Before moving on to the next request, a flat client transaction completes the previous one.
- As a result, each transaction visits the server object in order. A transaction can only wait for one object at a time when servers utilize locking.
- They are usually very simple and are generally used for short activities rather than larger ones.
- It is beneficial to commit partial results. Bulk updates can be expensive to undo all updates.

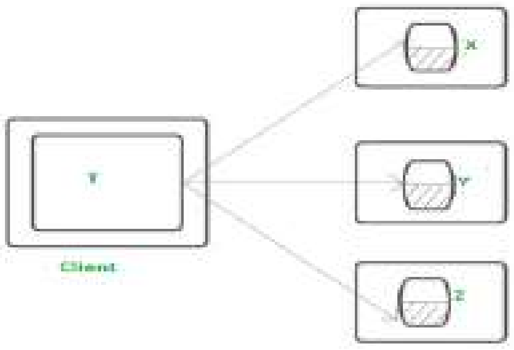


Fig : Flat Transaction

Nested Transactions

- In this Transaction model is to permit a transaction to include other transactions with their own begin and commit points. Such transactions are called *nested* transactions.
- These transactions that are embedded in another one are usually called *subtransactions*.
- *Example :* Let us extend the reservation transaction. Most travel agents will make reservations for hotels and car rentals in addition to the flights. If one chooses to specify all of this as one transaction, the reservation transaction would have the following structure:

```
Begin transaction Reservation
begin
   Begin transaction Airline
           ...
   end. {Airline}
   Begin transaction Hotel
           ...
```

```
            end. {Hotel}
-           Begin transaction Car
                    ...
            end. {Car}
    end.
end.
```

- The level of nesting is generally open, allowing subtransactions themselves to have nested transactions. This generality is necessary to support application areas where transactions are more complex than in traditional data processing.
- Based on the termination characteristics Nested Transaction is classified into :
    - ✓ *Closed Nested Transaction*
    - ✓ *Open Nested Transaction*

Closed nested transactions : Commit in a bottom-up fashion through the root. Thus, a nested subtransaction begins *after* its parent and finishes *before* it, and the commitment of the subtransactions is conditional upon the commitment of the parent. The semantics of these transactions enforce atomicity at the top-most level.

Open nesting transactions : relaxes the top-level atomicity restriction of closed nested transactions. Therefore, an open nested transaction allows its partial results to be observed outside the transaction. Sagas and split transactions are examples of open nesting. A saga is a "sequence of transactions that can be interleaved with other transactions".

The advantages of nested transactions are the following.

- First, they provide a higher-level of concurrency among transactions. Since a transaction consists of a number of other transactions, more concurrency is possible within a single transaction.
- A second argument in favor of nested transactions is related to recovery. It is possible to recover independently from failures of each subtransaction. This limits the damage to a smaller part of the transaction, making it less costly to recover.
- Finally, it is possible to create new transactions from existing ones simply by inserting the old one inside the new one as a subtransaction.

Workflows

- Flat transactions model relatively simple and short activities very well. However, they are less appropriate for modeling longer and more elaborate activities. That is the reason for the development of the various nested transaction models.
- These extensions are not sufficiently powerful to model business activities: "after several decades of data processing, we have learned that we have not won the battle of modeling and automating complex enterprises".
- To meet these needs, more complex transaction models which are combinations of open and nested transactions have been proposed. There are well-justified arguments for not calling these transactions, since they hardly follow any of the

ACID properties; a more appropriate name that has been proposed is a *workflow*.

- A working definition is that a workflow is “a collection of *tasks* organized to accomplish some business process.” Three types of workflows are identified:

❖ ***Human-oriented workflows***, which involve humans in performing the tasks. The system support is provided to facilitate collaboration and coordination among humans, but it is the humans themselves who are ultimately responsible for the consistency of the actions.

❖ ***System-oriented workflows*** are those that consist of computation-intensive and specialized tasks that can be executed by a computer. The system support in this case is substantial and involves concurrency control and recovery, automatic task execution, notification, etc.

❖ ***Transactional workflows*** range in between human-oriented and system- oriented workflows and borrow characteristics from both. They involve “coordinated execution of multiple tasks that
  - may involve humans,
  - require access to HAD [heterogeneous, autonomous, and/or distributed] systems, and
  - support selective use of transactional properties [i.e., ACID properties] for individual tasks or entire workflows.”

- Among the features of transactional workflows, the selective use of transactional properties is particularly important as it characterizes possible relaxations of ACID properties.

Example: Let us further extend the reservation transaction of Example 10.3. The entire reservation activity consists of the following taks and involves the follow- ing data:

- Customer request is obtained (task T_1) and Customer Database is accessed to obtain customer information, preferences, etc.;
- Airline reservation is performed (T_2) by accessing the Flight Database;
- Hotel reservation is performed (T_3), which may involve sending a message to the hotel involved;
- Auto reservation is performed (T_4), which may also involve communication with the car rental company;
- Bill is generated (T_5) and the billing info is recorded in the billing database.
- Figure depicts this workflow where there is a serial dependency of T_2 on T_1, and T_3, T_4 on T_2; however, T_3 and T_4 (hotel and car reservations) are performed in parallel and T_5 waits until their completion.

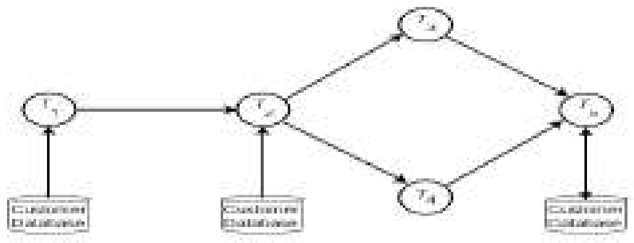
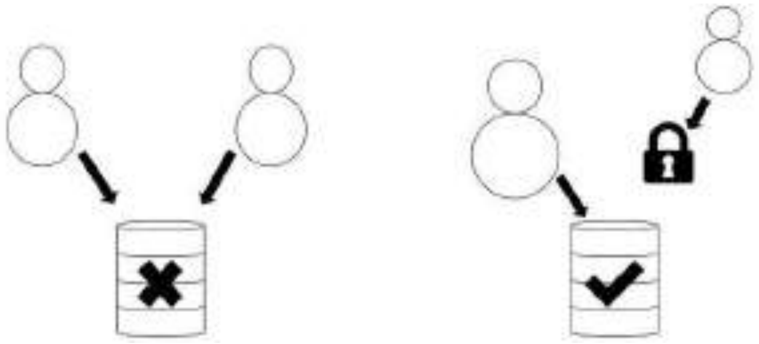


Fig : Workflow

Distributed Concurrency Control

- A Situation in which two or more persons access the same records simultaneously is called concurrency.
- Concurrency control involve the synchronization of access to the distributed database, such that the integrity of the database is maintained.
- Concurrency control deals with the isolation and consistency properties of transactions.
- The distributed concurrency control mechanism of a distributed DBMS ensures that the consistency of the database and it is maintained in a multiuser distributed environment.
- The level of concurrency (i.e., the number of concurrent transactions) is probably the most important parameter in distributed systems.
- The concurrency control mechanism attempts to find a suitable trade-off between maintaining the consistency of the database and maintaining a high level of concurrency.
- Serializability is the most widely accepted correctness criterion for concurrency control algorithms.



Serializability Theory

- The primary function of a concurrency controller is to generate a serializable history for the execution of pending transactions.
- To devise algorithms that are guaranteed to generate only serializable histories.
- Serializability theory extends in a straightforward manner to the non-replicated (or partitioned) distributed databases.
- The history of transaction execution at each site is called a *local history*.
- If the database is not replicated and each local history is serializable, their union (called the *global history*) is also serializable as long as local serialization orders are identical.

Example : We will give a very simple example to demonstrate the point. Consider two bank accounts, x (stored at Site 1) and y (stored at Site 2), and the following two transactions where T_1 transfers \$100 from x to y, while T_2 simply reads the balances of x and y:

T_1: Read(x)	T_2: Read(x)
$x \leftarrow x - 100$	Read(y)

Write(x)	Commit
Read(y)	
$y \leftarrow y + 100$	
Write(y)	
Commit	

Obviously, both of these transactions need to run at both sites. Consider the following two histories that may be generated locally at the two sites (H_i is the history at Site i):

$$H_1 = \{R_1(x), W_1(x), R_2(x)\}$$
$$H_2 = \{R_1(y), W_1(y), R_2(y)\}$$

- Both of these histories are serializable; indeed, they are serial. Therefore, each represents a correct execution order.
- The serialization order for both are the same $T_1 \rightarrow T_2$. Therefore, the global history that is obtained is also serializable with the serialization order $T_1 \rightarrow T_2$.
- However, if the histories generated at the two sites are as follows, there is a problem:

$$H_1' = \{R_1(x), W_1(x), R_2(x)\}$$
$$H_2' = \{R_2(y), R_1(y), W_1(y)\}$$

- Although each local history is still serializable, the serialization orders are different: H' serializes T_1 before T_2 while H' serializes T_2 before T_1. Therefore, there can be no global history that is serializable.
- A weaker version of serializability that has gained importance in recent years is *snapshot isolation* that is now provided as a standard consistency criterion in a number of commercial systems.
- Snapshot isolation allows read transactions (queries) to read stale data by allowing them to read a snapshot of the database that reflects the committed data at the time the read transaction starts.
- Consequently, the reads are never blocked by writes, even though they may read old data that may be dirtied by other transactions that were still running when the snapshot was taken.
- Hence, the resulting histories are not serializable, but this is accepted as a reasonable tradeoff between a lower level of isolation and better performance.

Concurrency Control Mechanisms & Algorithms

- There are a number of ways that the concurrency control approaches can be classified.
- One obvious classification criterion is the mode of database distribution.
- Some algorithms that have been proposed require a fully replicated database, while others can operate on partially replicated or partitioned databases.
- The concurrency control algorithms may also be classified according to network

topology, such as those requiring a communication subnet with broadcasting capability or those working in a star-type network or a circularly connected network.

- The corresponding breakdown of the concurrency control algorithms results in two classes:
    - those algorithms that are based on mutually exclusive access to shared data (locking),
    - those that attempt to order the execution of the transactions according to a set of rules (protocols).
- However, these primitives may be used in algorithms with two different viewpoints: the pessimistic view that many transactions will conflict with each other, or the optimistic view that not too many transactions will conflict with one another.
- Group the concurrency control mechanisms into two broad classes:
    - Pessimistic concurrency control methods
    - Optimistic concurrency control methods.
- ***Pessimistic* algorithms** synchronize the concurrent execution of transactions early in their execution life cycle.
- ***Optimistic* algorithms** delay the synchronization of transactions until their termination.
- The pessimistic group consists of *locking-based* algorithms, *ordering* (or *transaction ordering*) *based* algorithms, and *hybrid* algorithms.
- The optimistic group can, similarly, be classified as locking-based or timestamp ordering-based. This classification is depicted in Figure

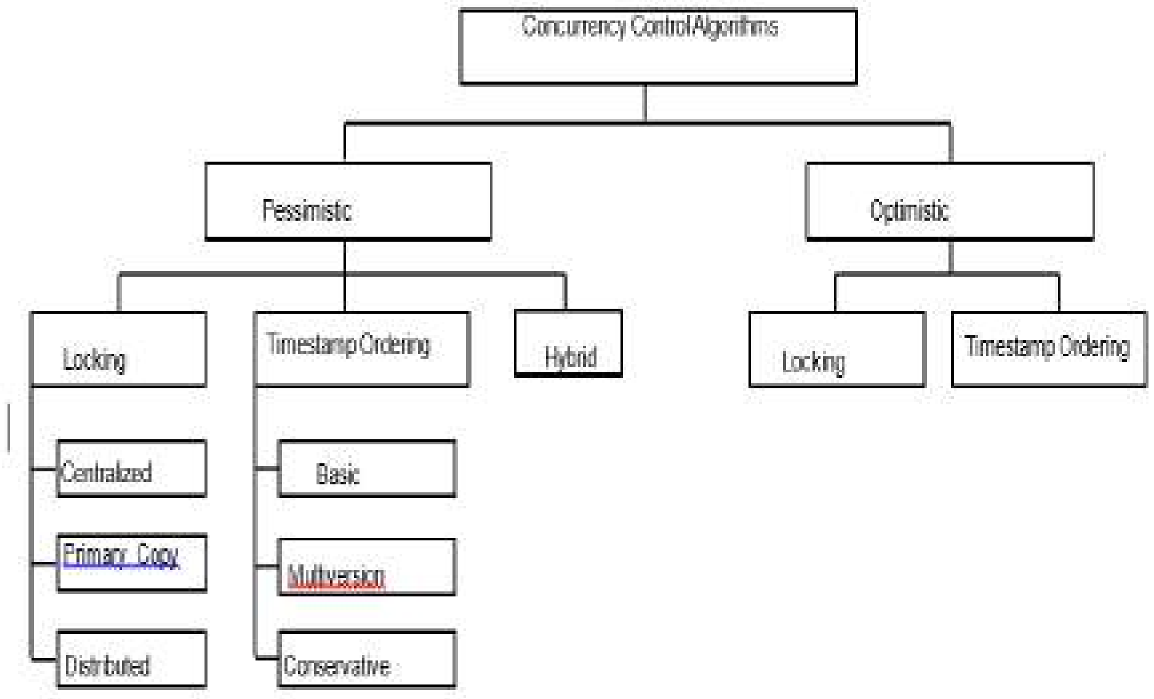


Fig : Classification of Concurrency Control Algorithms

- In the *locking-based* approach, the synchronization of transactions is achieved by employing physical or logical locks on some portion or granule of the database.
- This class is subdivided further according to where the lock management activities are performed: *centralized* and *decentralized* (or *distributed*) *locking*.
- The *timestamp ordering* (TO) class involves organizing the execution order of transactions so that they maintain transaction consistency.
- This ordering is maintained by assigning timestamps to both the transactions and the data items that are stored in the database. These algorithms can be *basic TO*, *multiversion TO*, or *conservative TO*.
- We should indicate that in some locking-based algorithms, timestamps are also used. This is done primarily to improve efficiency and the level of concurrency. We call these *hybrid* algorithms.

Locking-Based Concurrency Control Algorithms

- The main idea of locking-based concurrency control is to ensure that a data item that is shared by conflicting operations is accessed by one operation at a time. This is accomplished by associating a "lock" with each lock unit.
- This lock is set by a transaction before it is accessed and is reset at the end of its use.
- Obviously a lock unit cannot be accessed by an operation if it is already locked by another.
- There are two types of locks (commonly called *lock modes*) associated with each lock unit:
  - ✓ *read lock* (*rl*)
  - ✓ *write lock* (*wl*).
- A transaction T_i that wants to read a data item contained in lock unit *x* obtains a read lock on *x* [denoted $rl_i(x)$].
- The same happens for write operations.
- Two lock modes are *compatible* if two transactions that access the same data item can obtain these locks on that data item at the same time. read locks are compatible, whereas read-write or write-write locks are not. Therefore, it is possible, for example, for two transactions to read the same data item concurrently.

	$rl_i(x)$	$wl_j(x)$
$rl_j(x)$	compatible	not compatible
$wl_j(x)$	not compatible	not compatible

Fig : Compatibility Matrix of Lock Modes

- The distributed DBMS not only manages locks but also handles the lock management responsibilities on behalf of the transactions.
- It means users do not need to specify when a data item needs to be locked; the distributed DBMS takes care of that every time the transaction issues a read or write operation.

Example : Consider the following two transactions:

T_1: Read(x)	T_2: Read(x)
$x \leftarrow x+1$	$x \leftarrow x*2$
Write(x)	Write(x)
Read(y)	Read(y)
$y \leftarrow y-1$	$y \leftarrow y*2$
Write(y)	Write(y)
Commit	Commit

- The following is a valid history that a lock manager employing the locking algorithm may generate:

$$H = \{wl_1(x), R_1(x), W_1(x), lr_1(x), wl_2(x), R_2(x), w_2(x), lr_2(x), wl_2(y), R_2(y), W_2(y), lr_2(y), wl_1(y), R_1(y), W_1(y), lr_1(y)\}$$

- where $lr_i(z)$ indicates the release of the lock on z that transaction T_i holds.
- The problem with history H is the locking algorithm releases the locks that are held by a transaction (say, T_i) as soon as the associated database command (read or write) is executed, and that lock unit (say x) no longer needs to be accessed.
- However, the transaction itself is locking other items (say, y), after it releases its lock on x. Even though this may seem to be advantageous from the viewpoint of increased concurrency, it permits transactions to interfere with one another, resulting in the loss of isolation and atomicity.
- Hence, we go for *two-phase locking* (2PL).
- The two-phase locking rule simply states that no transaction should request a lock after it releases one of its locks.
- 2PL algorithms execute transactions in two phases. Each transaction has a *growing phase*, where it obtains locks and accesses data items, and a *shrinking phase,* during which it releases locks.
- The *lock point* is the moment when the transaction has achieved all its locks but has not yet started to release any of them.
- Thus the lock point determines the end of the growing phase and the beginning of the shrinking phase of a transaction.
- It has been proven that any history generated by a concurrency control algorithm that obeys the 2PL rule is serializable.

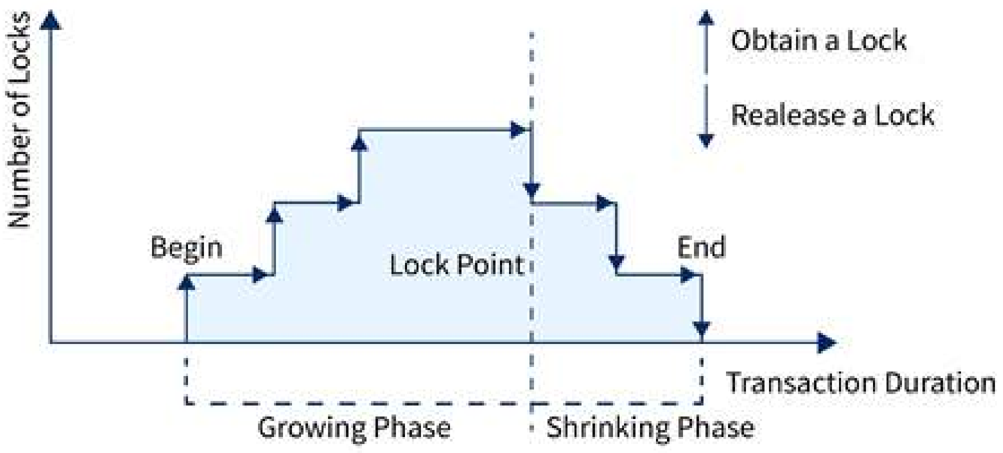


Fig 1 : 2PL Lock Graph

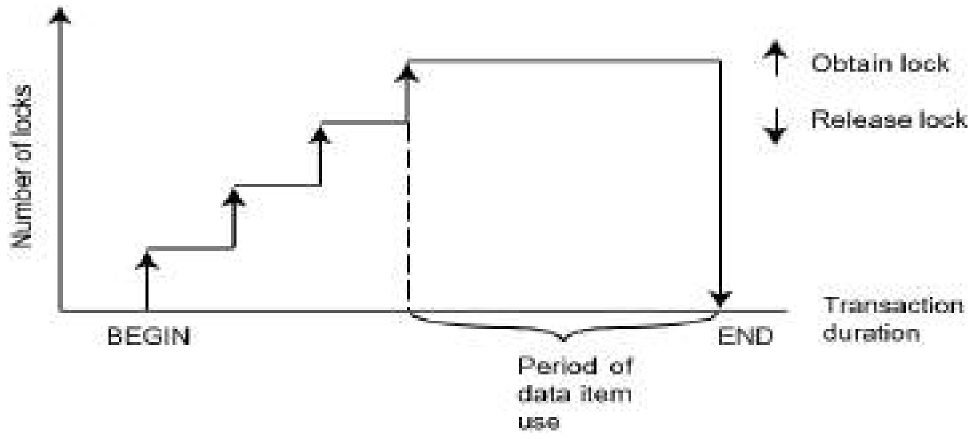


Fig 2: Strict 2PL Lock Graph.

- Figure 1 : it indicates that the lock manager releases locks as soon as access to that data item has been completed. This permits other transactions awaiting access to go ahead and lock it, thereby increasing the degree of concurrency.
- Figure 2 : if the transaction aborts after it releases a lock, it may cause other transactions that may have accessed the unlocked data item to abort as well. This is known as *cascading aborts*. These problems may be overcome by *strict two-phase locking*, which releases all the locks together when the transaction terminates (commits or aborts).

Centralized 2PL

- One way of doing this is to delegate lock management responsibility to a single site only. This means that only one of the sites has a lock manager; the transaction managers at the other sites communicate with it rather than with their own lock managers. This approach is also known as the *primary site* 2PL algorithm
- The communication between the cooperating sites in executing a transaction

according to a centralized 2PL (C2PL) algorithm.

- This communication is between the transaction manager at the site where the transaction is initiated (called the *coordinating* TM), the lock manager at the central site, and the data processors (DP) at the other participating sites.
- The participating sites are those that store the data item and at which the operation is to be carried out. The order of messages is denoted in the figure.

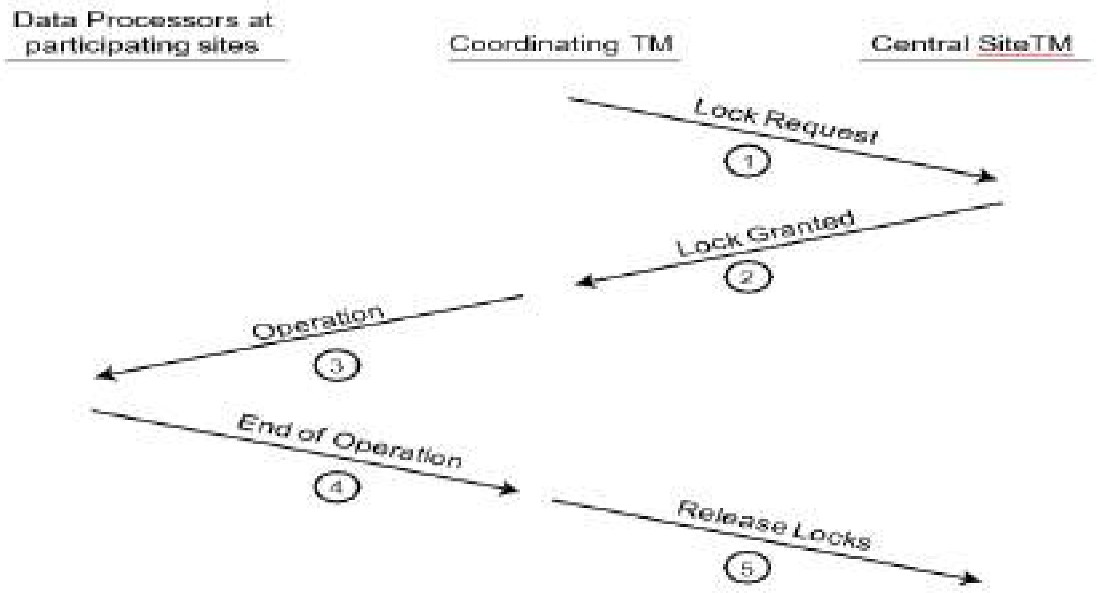


Fig : Communication Structure of Distributed 2PL

Distributed 2PL

- Distributed 2PL (D2PL) requires the availability of lock managers at each site.
- The communication between cooperating sites that execute a transaction according to the distributed 2PL protocol.
- The distributed 2PL transaction management algorithm is similar to the C2PL-TM, with two major modifications. The messages that are sent to the central site lock manager in C2PL-TM are sent to the lock managers at all participating sites in D2PL-TM.
- The second difference is that the operations are not passed to the data processors by the coordinating transaction manager, but by the participating lock managers.
- This means that the coordinating transaction manager does not wait for a "lock request granted" message.

Timestamp-Based Concurrency Control Algorithms

- To establish timestamp ordering, the transaction manager assigns each transaction T_i a unique timestamp, $ts(T_i)$, at its initiation.
- A timestamp is a simple identifier that serves to identify each transaction uniquely and is used for ordering.
- Uniqueness is only one of the properties of timestamp generation.
- The second property is monotonicity. Two timestamps generated by the same transaction manager should be monotonically increasing.

- There are a number of ways that timestamps can be assigned. To maintain uniqueness, each site appends its own identifier to the counter value. Thus the timestamp is a two-tuple of the form ⟨local counter value, site identifier⟩.
- It is simple to order the execution of the transactions' operations according to their timestamps. Formally, the timestamp ordering (TO) rule can be specified as follows:
- **TO Rule.** Given two conflicting operations O_{ij} and O_{kl} belonging, respectively, to transactions T_i and T_k, O_{ij} is executed before O_{kl} if and only if $ts(T_i) < ts(T_k)$. In this case T_i is said to be the *older* transaction and T_k is said to be the *younger* one.
- A scheduler that enforces the TO rule checks each new operation against conflicting operations that have already been scheduled. If the new operation belongs to a transaction that is younger than all the conflicting ones that have already been scheduled, the operation is accepted; otherwise, it is rejected, causing the entire transaction to restart with a *new* timestamp.
- A timestamp ordering scheduler that operates in this fashion is guaranteed to generate serializable histories. However, this comparison between the transaction timestamps can be performed only if the scheduler has received all the operations to be scheduled. If operations come to the scheduler one at a time (which is the realistic case), it is necessary to be able to detect, in an efficient manner, if an operation has arrived out of sequence. To facilitate this check, each data item x is assigned two timestamps:
- a *read timestamp* [$rts(x)$], which is the largest of the timestamps of the transactions that have read x, and
- a *write timestamp* [$wts(x)$], which is the largest of the timestamps of the transactions that have written (updated) x. It is now sufficient to compare the timestamp of an operation with the read and write timestamps of the data item that it wants to access to determine if any transaction with a larger timestamp has already accessed the same data item.
- The transaction manager is responsible for assigning a timestamp to each new transaction and attaching this timestamp to each database operation that it passes on to the scheduler.
- The latter component is responsible for keeping track of read and write timestamps as well as performing the serializability check.

Basic TO Algorithm

- The basic TO algorithm is a straightforward implementation of the TO rule.
- The coordinating transaction manager assigns the timestamp to each transaction, determines the sites where each data item is stored, and sends the relevant operations to these sites.
- The histories at each site simply enforce the TO rule.
- A transaction one of whose operations is rejected by a scheduler is restarted by the transaction manager with a new timestamp. This ensures that the transaction has a chance to execute in its next try.
- Since the transactions never wait while they hold access rights to data items, the basic TO algorithm never causes deadlocks.

- The penalty of deadlock freedom is potential restart of a transaction numerous times.
- There is an alternative to the basic TO algorithm that reduces the number of restarts.
- When an accepted operation is passed on to the data processor, the scheduler needs to refrain from sending another conflicting, but acceptable operation to the data processor until the first is processed and acknowledged.
- This is a requirement to ensure that the data processor executes the operations in the order in which the scheduler passes them on. Otherwise, the read and write timestamp values for the accessed data item would not be accurate.
- ***Example*** *:.* Assume that the TO scheduler first receives $W_i(x)$ and then receives $W_j(x)$, where $ts(T_i) < ts(T_j)$. The scheduler would accept both operations and pass them on to the data processor. The result of these two operations is that $wts(x) = ts(T_j)$, and we then expect the effect of $W_j(x)$ to be represented in the database. However, if the data processor does not execute them in that order, the effects on the database will be wrong.
- The scheduler can enforce the ordering by maintaining a queue for each data item that is used to delay the transfer of the accepted operation until an acknowledgment is received from the data processor regarding the previous operation on the same data item.
- Such a complication does not arise in 2PL-based algorithms because the lock manager effectively orders the operations by releasing the locks only after the operation is executed. In one sense the queue that the TO scheduler maintains may be thought of as a lock. However, this does not imply that the history generated by a TO scheduler and a 2PL scheduler would always be equivalent. There are some histories that a TO scheduler would generate that would not be admissible by a 2PL history.
- Remember that in the case of strict 2PL algorithms, the releasing of locks is delayed further, until the commit or abort of a transaction. It is possible to develop a strict TO algorithm by using a similar scheme. For example, if $W_i(x)$ is accepted and released to the data processor, the scheduler delays all $R_j(x)$ and $W_j(x)$ operations
- (for all T_j) until T_i terminates (commits or aborts).

Conservative TO Algorithm

- We indicated in the preceding section that the basic TO algorithm never causes operations to wait, but instead, restarts them. We also pointed out that even though this is an advantage due to deadlock freedom, it is also a disadvantage, because numerous restarts would have adverse performance implications.
- The conservative TO algorithms attempt to lower this system overhead by reducing the number of transaction restarts.
- Let us first present a technique that is commonly used to reduce the probability of restarts.
- Remember that a TO scheduler restarts a transaction if a younger conflicting transaction is already scheduled or has been executed.
- **Example :** one site is comparatively inactive relative to the others and does not

issue transactions for an extended period. In this case its timestamp counter indicates a value that is considerably smaller than the counters of other sites.

- If the TM at this site then receives a transaction, the operations that are sent to the histories at the other sites will almost certainly be rejected, causing the transaction to restart.
- Furthermore, the same transaction will restart repeatedly until the timestamp counter value at its originating site reaches a level of parity with the counters of other sites.
- Each transaction manager can send its remote operations, rather than histories, to the transaction managers at the other sites.
- The receiving transaction managers can then compare their own counter values with that of the incoming operation. Any manager whose counter value is smaller than the incoming one adjusts its own counter to one more than the incoming one.
- if system clocks are used instead of counters, this approximate synchronization may be achieved automatically as long as the clocks are of comparable speeds.
- We will consider one possible implementation of the conservative TO algorithm ;
- Assume that each scheduler maintains one queue for each transaction manager in the system. The scheduler at site *i* stores all the operations that it receives from the transaction manager at site *j* in queue Q_{ij}.
- Scheduler *i* has one such queue for each *j*. When an operation is received from a transaction manager, it is placed in its appropriate queue in increasing timestamp order.
- The histories at each site execute the operations from these queues in increasing timestamp order.
- This scheme will reduce the number of restarts, but it will not guarantee that they will be eliminated completely. Consider the case where at site *i* the queue for site *j* (Q_{ij}) is empty. The scheduler at site *i* will choose an operation [say, $R(x)$] with the smallest timestamp and pass it on to the data processor.
- However, site *j* may have sent to *i* an operation [say, $W(x)$] with a smaller timestamp which may still be in transit in the network. When this operation reaches site *i*, it will be rejected since it violates the TO rule: it wants to access a data item that is currently being accessed (in an incompatible mode) by another operation with a higher timestamp.
- It is possible to design an extremely conservative TO algorithm by insisting that the scheduler choose an operation to be sent to the data processor only if there is at least one operation in each queue.
- This guarantees that every operation that the scheduler receives in the future will have timestamps greater than or equal to those currently in the queues.

Multiversion TO Algorithm

- Multiversion TO is another attempt at eliminating the restart overhead cost of transactions.
- Most of the work on multiversion TO has concentrated on centralized databases, so we present only a brief overview.
- We should indicate that multiversion TO algorithm would be a suitable

concurrency control mechanism for DBMSs that are designed to support applications that inherently have a notion of versions of database objects (e.g., engineering databases and document databases).

- In multiversion TO, the updates do not modify the database; each write operation creates a new version of that data item. Each version is marked by the timestamp of the transaction that creates it.
- Thus the multiversion TO algorithm trades storage space for time.
- The existence of versions is transparent to users who issue transactions simply by referring to data items, not to any specific version.
- The transaction manager assigns a timestamp to each transaction, which is also used to keep track of the timestamps of each version.
- The operations are processed by the histories as follows:

1. A $R_i(x)$ is translated into a read on one version of x. This is done by finding a version of x (say, x_v) such that $ts(x_v)$ is the largest timestamp less than $ts(T_i)$. $R_i(x_v)$ is then sent to the data processor to read x_v. This case is depicted in

   Figure - a, which shows that R_i can read the version (x_v) that it would have read had it arrived in timestamp order.
2. A $W_i(x)$ is translated into $W_i(x_w)$ so that $ts(x_w) = ts(T_i)$ and sent to the data processor if and only if no other transaction with a timestamp greater than $ts(T_i)$ has read the value of a version of x (say, x_r) such that $ts(x_r) > ts(x_w)$.
   In other words, if the scheduler has already processed a $R_j(x_r)$ such that

   $$ts(T_i) < ts(x_r) < ts(T_j)$$

   then $W_i(x)$ is rejected. This case is depicted in Figure - b, which shows that if W_i is accepted, it would create a version (x_w) that R_j should have read, but did not since the version was not available when R_j was executed – it, instead, read version x_k, which results in the wrong history.

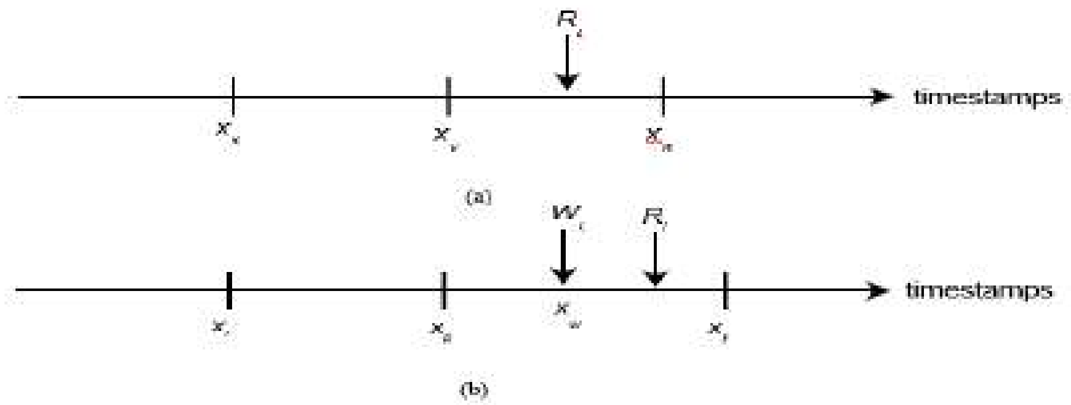


Fig. 11.11 Multiversion TO Cases

- A scheduler that processes the read and the write requests of transactions according to the rules noted above is guaranteed to generate serializable histories.
- To save space, the versions of the database may be purged from time to time. This should be done when the distributed DBMS is certain that it will no longer receive a transaction that needs to access the purged versions.

Optimistic Concurrency Control Algorithms

- The concurrency control algorithms are pessimistic in nature. In other words, they assume that the conflicts between transactions are quite frequent and do not permit a transaction to access a data item if there is a conflicting transaction that accesses that data item.
- The execution of any operation of a transaction follows the sequence of phases: validation (V), read (R), computation (C), write (W) (Figure 11.12). Generally, this sequence is valid for an update transaction as well as for each of its operations.

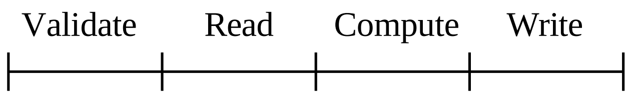


Fig. 11.12 Phases of Pessimistic Transaction Execution

- Optimistic algorithms, on the other hand, delay the validation phase until just before the write phase (Figure 11.13). Thus an operation submitted to an optimistic scheduler is never delayed.
- The read, compute, and write operations of each transaction are processed freely without updating the actual database.
- Each transaction initially makes its updates on local copies of data items. The validation phase consists of checking if these updates would maintain the consistency of the database.
- If the answer is affirmative, the changes are made global (i.e., written into the actual database). Otherwise, the transaction is aborted and has to restart.

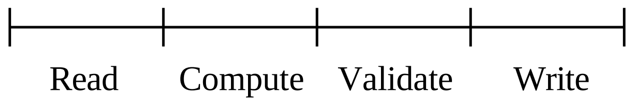


Fig. 11.13 Phases of Optimistic Transaction Execution

- It is possible to design locking-based optimistic concurrency control algorithms.
- The original optimistic proposals are based on timestamp ordering. Therefore, we describe only the optimistic approach using timestamps.
- It differs from pessimistic TO-based algorithms not only by being optimistic but also in its assignment of timestamps. Timestamps are associated only with transactions, not with data items (i.e., there are no read or write timestamps).
- Furthermore, timestamps are not assigned to transactions at their initiation but at the beginning of their validation step. This is because the timestamps are needed only during the validation phase, and as we will see shortly, their early assignment may cause unnecessary transaction rejections.
- Each transaction T_i is subdivided (by the transaction manager at the originating site) into a number of subtransactions, each of which can execute at many sites. Notationally, let us denote by T_{ij} a subtransaction of T_i that executes at site j.
- Until the validation phase, each local execution follows the sequence depicted in

Figure 11.13. At that point a timestamp is assigned to the transaction which is copied to all its subtransactions.

- The local validation of T_{ij} is performed according to the following rules, which are mutually exclusive.

Rule 1. If all transactions T_k where $ts(T_k) < ts(T_{ij})$ have completed their write phase before T_{ij} has started its read phase (Figure 11.14a),[3] validation succeeds, because transaction executions are in serial order.

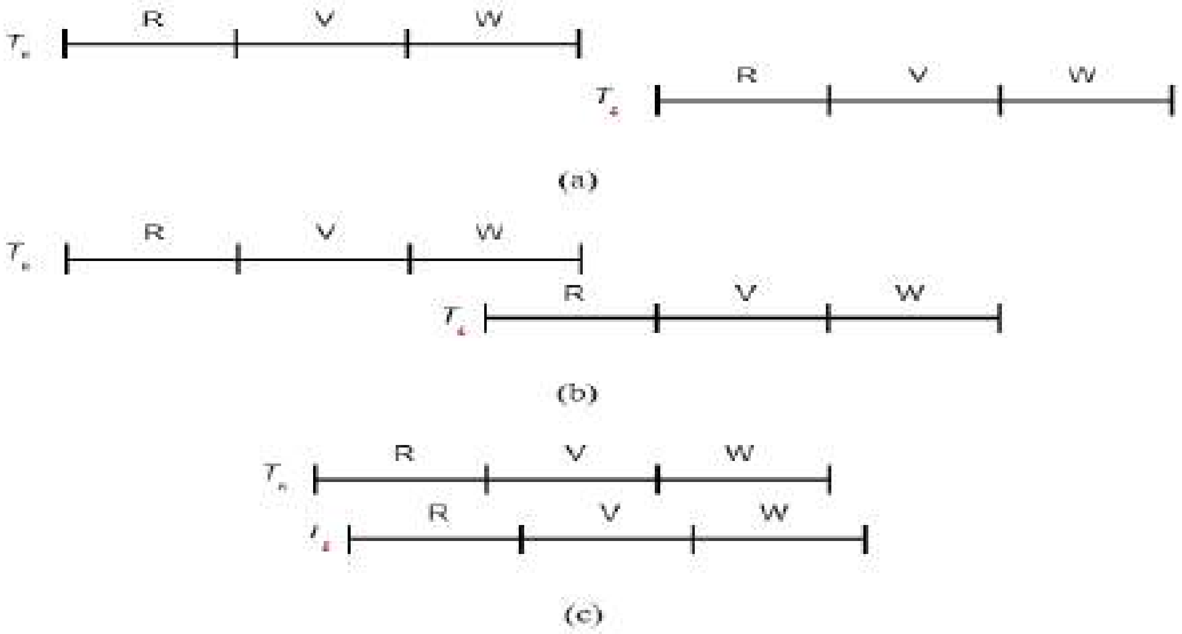


Fig. 11.14 Possible Execution Scenarios

Rule 2. If there is any transaction T_k such that $ts(T_k) < ts(T_{ij})$, and which completes its write phase while T_{ij} is in its read phase (Figure 11.14b), the validation succeeds if $WS(T_k) \cap RS(T_{ij}) = \emptyset$.

Rule 3. If there is any transaction T_k such that $ts(T_k) < ts(T_{ij})$, and which completes its read phase before T_{ij} completes its read phase (Figure 11.14c), the validation succeeds if $WS(T_k) \cap RS(T_{ij}) = \emptyset$, and $WS(T_k) \cap WS(T_{ij}) = \emptyset$.

- Rule 1 is obvious; it indicates that the transactions are actually executed serially in their timestamp order. Rule 2 ensures that none of the data items updated by T_k. are read by T_{ij} and that T_k finishes writing its updates into the database before T_{ij} starts writing.

- Thus the updates of T_{ij} will not be overwritten by the updates of T_k. Rule 3 is similar to Rule 2, but does not require that T_k finish writing before T_{ij} starts writing. It simply requires that the updates of T_k not affect the read phase or the write phase of T_{ij}.

- **An advantage** of optimistic concurrency control algorithms is their potential to allow a higher level of concurrency. It has been shown that when transaction conflicts are very rare, the optimistic mechanism performs better than locking.

- **A major problem** with optimistic algorithms is the higher storage cost. To validate a transaction, the optimistic mechanism has to store the read and the write sets of several other terminated transactions.
- Another problem is starvation. Consider a situation in which the validation phase of a long transaction fails. In subsequent trials it is still possible that the validation will fail repeatedly.

Deadlock Management

- Any locking-based concurrency control algorithm may result in deadlocks, since there is mutual exclusion of access to shared resources (data) and transactions may wait on locks.
- Some TO-based algorithms that require the waiting of transactions (e.g., strict TO) may also cause deadlocks.
- **A deadlock can occur because transactions wait for one another.**
- A deadlock situation is a set of requests that can never be granted by the concurrency control mechanism.

- ***Example :*** Consider two transactions T_i and T_j that hold write locks on two entities x and y [i.e., $wl_i(x)$ and $wl_j(y)$]. Suppose that T_i now issues a $rl_i(y)$ or a $wl_i(y)$. Since y is currently locked by transaction T_j, T_i will have to wait until T_j releases its write lock on y.

- However, if during this waiting period, T_j now requests a lock (read or write) on x, there will be a deadlock. This is because, T_i will be blocked waiting for T_j to release its lock on y while T_j will be waiting for T_i to release its lock on x. In this case, the two transactions T_i and T_j will wait indefinitely for each other to release their respective locks

- A deadlock is a permanent phenomenon. If one exists in a system, it will not go away unless outside intervention takes place. This outside interference may come from the user, the system operator, or the software system.
- A useful tool in analyzing deadlocks is a *wait-for graph* (WFG). A WFG is a directed graph that represents the wait-for relationship among transactions.
- The nodes of this graph represent the concurrent transactions in the system. An edge $T_i \rightarrow T_j$ exists in the WFG if transaction T_i is waiting for T_j to release a lock on some entity.

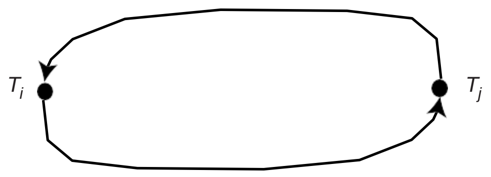


Fig. A WFG Example

- Using the WFG, it is easier to indicate the condition for the occurrence of a deadlock. A deadlock occurs when the WFG contains a cycle.

- We should indicate that the formation of the WFG is more complicated in distributed systems, since two transactions that participate in a deadlock condition may be running at different sites. We call this situation a *global deadlock*.
- In distributed systems, then, it is not sufficient that each local distributed DBMS form a *local wait-for graph* (LWFG) at each site; it is also necessary to form a *global wait-for graph* (GWFG), which is the union of all the LWFGs.

Example : Consider four transactions T_1, T_2, T_3, and T_4 with the following wait- for relationship among them: $T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow T_4 \rightarrow T_1$. If T_1 and T_2 run at site 1 while T_3 and T_4 run at site 2, the LWFGs for the two sites are shown in Figure a. Notice that it is not possible to detect a deadlock simply by examining the two LWFGs, because the deadlock is global. The deadlock can easily be detected, however, by examining the GWFG where intersite waiting is shown by dashed lines (Figure b).

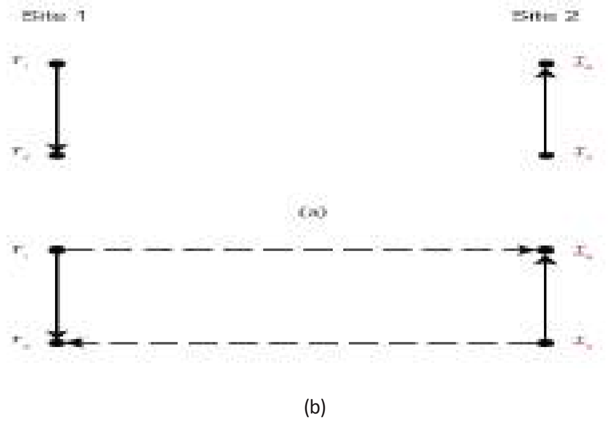
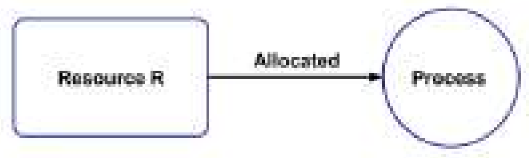


Fig. Difference between LWFG and GWFG

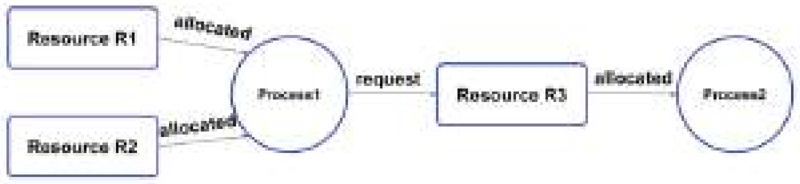
Necessary Conditions of Deadlock

There are four different conditions that result in Deadlock. These four conditions are also known as Coffman conditions and these conditions are not mutually exclusive. Let's look at them one by one.

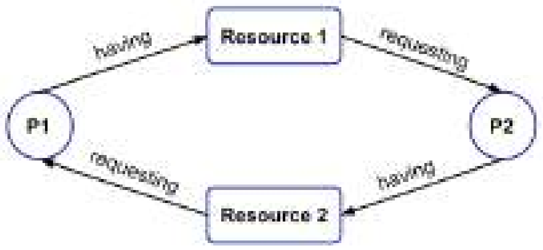
- **Mutual Exclusion:** A resource can be held by only one process at a time. In other words, if a process P1 is using some resource R at a particular instant of time, then some other process P2 can't hold or use the same resource R at that particular instant of time. The process P2 can make a request for that resource R but it can't use that resource simultaneously with process P1.



- **Hold and Wait:** A process can hold a number of resources at a time and at the same time, it can request for other resources that are being held by some other process. For example, a process P1 can hold two resources R1 and R2 and at the same time, it can request some resource R3 that is currently held by process P2.



- **No preemption:** A resource can't be preempted from the process by another process, forcefully. For example, if a process P1 is using some resource R, then some other process P2 can't forcefully take that resource. If it is so, then what's the need for various scheduling algorithm. The process P2 can request for the resource R and can wait for that resource to be freed by the process P1.
- **Circular Wait:** Circular wait is a condition when the first process is waiting for the resource held by the second process, the second process is waiting for the resource held by the third process, and so on. At last, the last process is waiting for the resource held by the first process. So, every process is waiting for each other to release the resource and no one is releasing their own resource. Everyone is waiting here for getting the resource. This is called a circular wait.



- There are three known methods for handling deadlocks:
  - ✓ Deadlock Prevention
  - ✓ Deadlock Avoidance
  - ✓ Deadlock Detection and Resolution

Deadlock Prevention

- Deadlock prevention methods guarantee that deadlocks cannot occur in the first place.
- Thus the transaction manager checks a transaction when it is first initiated and does not permit it to proceed if it may cause a deadlock.
- To perform this check, it is required that all of the data items that will be accessed by a transaction be predeclared.
- The transaction manager then permits a transaction to proceed if all the data items that it will access are available. Otherwise, the transaction is not permitted to proceed.

- There are two approaches for deadlock prevention:
  1. Non-Preemptive approach (Process not Interrupted)
  2. Preemptive approach (Process can Interrupt)

Deadlock Avoidance

- Deadlock avoidance schemes either employ concurrency control techniques that will never result in deadlocks or require that potential deadlock situations are detected in advance and steps are taken such that they will not occur. We consider both of these cases.
- The simplest means of avoiding deadlocks is to order the resources and insist that each process request access to these resources in that order.
- The lock units in the distributed database are ordered and transactions always request locks in that order. This ordering of lock units may be done either globally or locally at each site.
- Another alternative is to make use of transaction timestamps to prioritize transactions and resolve deadlocks by aborting transactions with higher (or lower) priorities.
- To implement this type of prevention method, the lock manager is modified as follows. If a lock request of a transaction T_i is denied, the lock manager does not automatically force T_i to wait.
- If the test is passed, T_i is permitted to wait for T_j; otherwise, one transaction or the other is aborted.
- Examples of this approach is the WAIT-DIE and WOUND-WAIT algorithms. These algorithms are based on the assignment of timestamps to transactions.

- **WAIT-DIE is a non-preemptive algorithm** in that if the lock request of T_i is denied because the lock is held by T_j, it never preempts T_j, following the rule:

- **WAIT-DIE Rule.** If T_i requests a lock on a data item that is already locked by T_j, T_i is permitted to wait if and only if T_i is older than T_j. If T_i is younger than T_j, then T_i is aborted and restarted with the same timestamp.

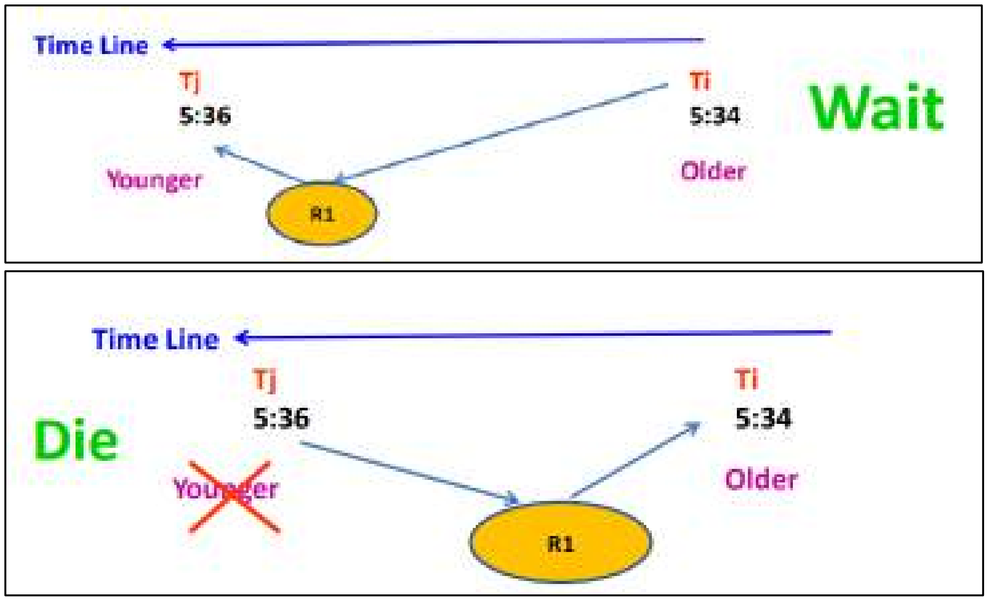
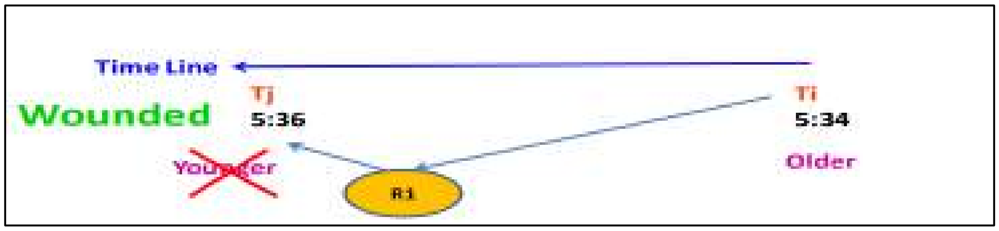


Fig : Non-Preemptive Algorithm (Wait – Die Rule)

- **A preemptive version of the same idea is the WOUND-WAIT algorithm**, which follows the rule:
- **WOUND-WAIT Rule.** If T_i requests a lock on a data item that is already locked by T_j, then T_i is permitted to wait if only if it is younger than T_j; otherwise, T_j is aborted and the lock is granted to T_i.



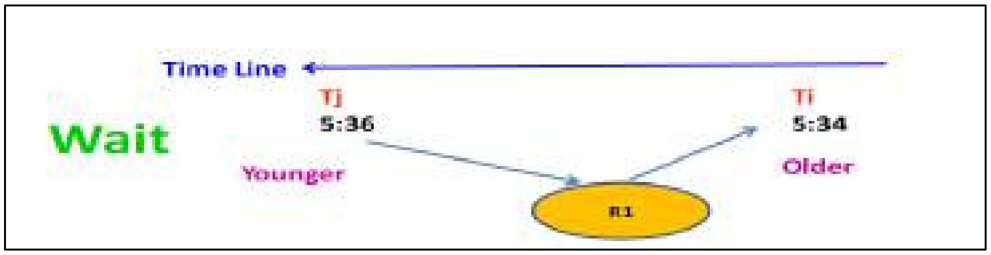


Fig : Preemptive Algorithm (Wound – Wait Rule)

- The rules are specified from the viewpoint of T_i: T_i waits, T_i dies, and T_i wounds T_j. In fact, the result of wounding and dying are the same: the affected transaction is aborted and restarted. With this perspective, the two rules can be specified as follows:

  if $ts(T_i) < ts(T_j)$ **then** T_i waits **else** T_i dies (WAIT-DIE)

  if $ts(T_i) < ts(T_j)$ **then** T_j is wounded **else** T_i waits (WOUND-WAIT)

- Notice that in both algorithms the younger transaction is aborted. The difference between the two algorithms is whether or not they preempt active transactions.
- Also note that the WAIT-DIE algorithm prefers younger transactions and kills older ones. Thus an older transaction tends to wait longer and longer as it gets older.
- The WOUND-WAIT rule prefers the older transaction since it never waits for a younger one. One of these methods, or a combination, may be selected in implementing a deadlock prevention algorithm.
- Deadlock avoidance methods are more suitable than prevention schemes for timestamp.
- Their fundamental drawback is that they require run-time support for deadlock management, which adds to the run-time overhead of transaction execution.

Deadlock Detection and Resolution

- Deadlock detection and resolution is the most popular and best-studied method. Detection is done by studying the GWFG for the formation of cycles.
- Resolution of deadlocks is typically done by the selection of one or more *victim* transaction(s) that will be preempted and aborted in order to break the cycles in the GWFG.

- Under the assumption that the cost of preempting each member of a set of deadlocked transactions is known, the problem of selecting the minimum total-cost set for breaking the deadlock cycle has been shown to be a difficult (NP-complete) problem. However, there are some factors that affect this choice :

1. The amount of effort that has already been invested in the transaction. This effort will be lost if the transaction is aborted.
2. The cost of aborting the transaction. This cost generally depends on the number of updates that the transaction has already performed.
3. The amount of effort it will take to finish executing the transaction. The scheduler wants to avoid aborting a transaction that is almost finished. To do this, it must be able to predict the future behavior of active transactions (e.g., based on the transaction's type).
4. The number of cycles that contain the transaction. Since aborting a transaction breaks all cycles that contain it, it is best to abort transactions that are part of more than one cycle (if such transactions exist).

- Now we can return to deadlock detection. There are three fundamental methods of detecting distributed deadlocks, referred as *centralized, distributed,* and *hierarchical deadlock detection*.

Centralized Deadlock Detection

- In the centralized deadlock detection approach, one site is designated as the deadlock detector for the entire system. Periodically, each lock manager transmits its LWFG to the deadlock detector, which then forms the GWFG and looks for cycles in it. Actually, the lock managers need only send changes in their graphs (i.e., the newly created or deleted edges) to the deadlock detector.
- The length of intervals for transmitting this information is a system design decision: the smaller the interval, the smaller the delays due to undetected deadlocks, but the larger the communication cost.
- Centralized deadlock detection has been proposed for distributed INGRES. This method is simple and would be a very natural choice if the concurrency control algorithm were centralized 2PL.
- However, the issues of vulnerability to failure, and high communication overhead, must also be considered.

Hierarchical Deadlock Detection

- An alternative to centralized deadlock detection is the building of a hierarchy of deadlock detectors.
- Deadlocks that are local to a single site would be detected at that site using the LWFG. Each site also sends its LWFG to the deadlock detector at the next level. Thus, distributed deadlocks involving two or more sites would be detected by a

deadlock detector in the next lowest level that has control over these sites.

- For example, a deadlock at site 1 would be detected by the local deadlock detector (*DD*) at site 1 (denoted DD_{21}, 2 for level 2, 1 for site 1). If, however, the deadlock involves sites 1 and 2, then DD_{11} detects it. Finally, if the deadlock involves sites 1 and 4, DD_{0x} detects it, where x is either one of 1, 2, 3, or 4.
- The hierarchical deadlock detection method reduces the dependence on the central site, thus reducing the communication cost.
- It is, however, considerably more complicated to implement and would involve non-trivial modifications to the lock and transaction manager algorithms.

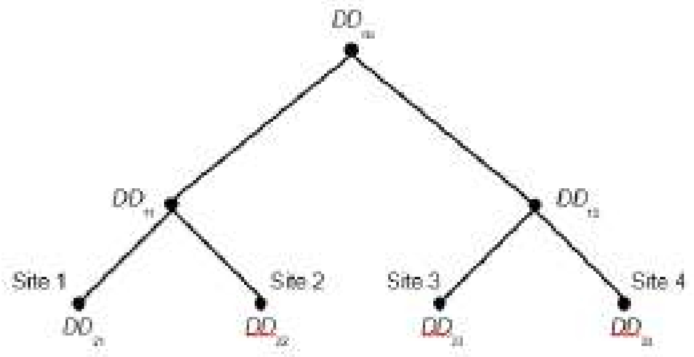


Fig. Hierarchical Deadlock Detection

Distributed Deadlock Detection

- Distributed deadlock detection algorithms delegate the responsibility of detecting deadlocks to individual sites.
- Thus, as in the hierarchical deadlock detection, there are local deadlock detectors at each site that communicate their LWFGs with one another (in fact, only the potential deadlock cycles are transmitted).
- Among the various distributed deadlock detection algorithms, the one implemented in System R* seems to be the more widely known and referenced.
- The LWFG at each site is formed and is modified as follows:

1. Since each site receives the potential deadlock cycles from other sites, these edges are added to the LWFGs.
2. The edges in the LWFG which show that local transactions are waiting for transactions at other sites are joined with edges in the LWFGs which show that remote transactions are waiting for local ones.

- Local deadlock detectors look for two things. If there is a cycle that does not include the external edges, there is a local deadlock that can be handled locally.
- If, on the other hand, there is a cycle involving these external edges, there is a potential distributed deadlock and this cycle information has to be communicated to other deadlock detectors.
- Obviously, it can be transmitted to all deadlock detectors in the system. In the

absence of any more information, this is the only alternative, but it incurs a high overhead. If, however, one knows whether the transaction is ahead or behind in the deadlock cycle, the information can be transmitted forward or backward along the sites in this cycle.

- The receiving site then modifies its LWFG as discussed above, and checks for deadlocks. Obviously, there is no need to transmit along the deadlock cycle in both the forward and backward directions.
- In the case of, site 1 would send it to site 2 in both forward and backward transmission along the deadlock cycle.
- The distributed deadlock detection algorithms require uniform modification to the lock managers at each site. This uniformity makes them easier to implement. However, there is the potential for excessive message transmission.
- Let the path that has the potential of causing a distributed deadlock in the local WFG of a site be $T_i \rightarrow \cdots \rightarrow T_j$. A local deadlock detector forwards the cycle information only if $ts(T_i) < ts(T_j)$. This reduces the average number of message transmissions by one-half.

DISTRIBUTED DATABASES (R22CSE3146)

III B.Tech I Semester (Information Technology)

UNIT – IV

Distributed DBMS Reliability : Reliability concepts and measures, fault-tolerance in distributed systems, failures in Distributed DBMS, local & distributed reliability protocols, site failures and network partitioning.

Parallel Database Systems : Parallel database system architectures, parallel data placement, parallel query processing, load balancing, database clusters.

Distributed DBMS Reliability

- A **reliable DDBMS** is one that can continue to process user requests even when the underlying system is unreliable, i.e., failures occur.

  Data replication + Easy scaling = **Reliable system**
- **Distribution** enhances system reliability (not enough). Need number of protocols to be implemented to exploit distribution and replication.
- Reliability is closely related to the problem of how to maintain the **atomicity** and **durability** properties of transactions.

Reliability Concepts and Measures

- A reliable distributed database management system is one that can continue to process user requests even when the underlying system is unreliable.
- When components of the distributed computing environment fail, a reliable distributed DBMS should be able to continue executing user requests without violating database consistency.

1. System, State, and Failure

- Reliability refers to a ***system*** that consists of a set of ***components*.**
- The system has a ***state***, which changes as the system operates.
- The behavior of the system in providing response to all the possible external stimuli is laid out in an authoritative ***specification*** of its behavior.
- The specification indicates the valid behavior of each system state.
- Any deviation of a system from the behavior described in the specification is considered a ***failure*.**
- For example, in a distributed transaction manager the specification may state that only serializable schedules for the execution of concurrent transactions should be generated.
- If the transaction manager generates a non-serializable schedule, we say that it has **failed.**
- Each failure obviously needs to be traced back to its cause.
- Failures in a system can be attributed to deficiencies either in the components that make it up, or in the design, that is, how these components are put together.
- Each state that a reliable system goes through is valid in the sense that the state fully meets its specification.

- However, in an unreliable system, it is possible that the system may get to an internal state that may not obey its specification.
- Further transitions from this state would eventually cause a system **failure.**
- Such internal states are called ***erroneous states***; the part of the state that is incorrect is called an ***error*** in the system.
- Any error in the internal states of the components of a system or in the design of a system is called a ***fault*** in the system.
- Thus, a fault causes an error that results in a system failure.

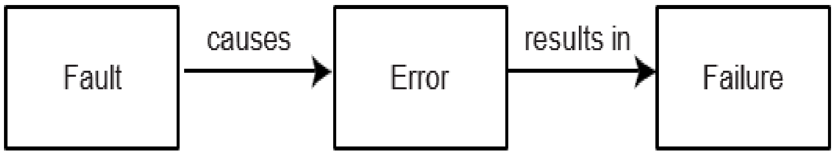


Fig. Chain of Events Leading to System Failure

- We differentiate between errors (or faults and failures) that are permanent and those that are not permanent. (Types of Faults)

Hard faults

- Permanent
- Resulting failures are called hard failures
- Permanence can apply to a failure, a fault, or an error, although we typically use the term with respect to faults.
- A *permanent fault*, also commonly called a *hard fault*, is one that reflects an irreversible change in the behavior of the system.
- Permanent faults cause permanent errors that result in permanent failures.
- The characteristics of these failures is that recovery from them requires intervention to "repair" the fault.

Soft faults.

- Transient or intermittent
- Account for more than 90% of all failures
- Resulting failures are called soft failures
- An intermittent fault refers to a fault that demonstrates itself occasionally due to unstable hardware or varying hardware or software states.
- A typical example is the faults that systems may demonstrate when the load becomes too heavy.
- On the other hand, a transient fault describes a fault that results from temporary environmental conditions.
- A transient fault might occur, for example, due to a sudden increase in the room temperature.
- The transient fault is therefore the result of environmental conditions that may be impossible to repair.
- An intermittent fault, on the other hand, can be repaired since the fault can be traced to a component of the system.
- Remember that we have also indicated that system failures can be due to design faults. Design faults together with unstable hardware cause intermittent errors that result in system failure.

- A final source of system failure that may not be attributable to a component fault or a design fault is operator mistakes.

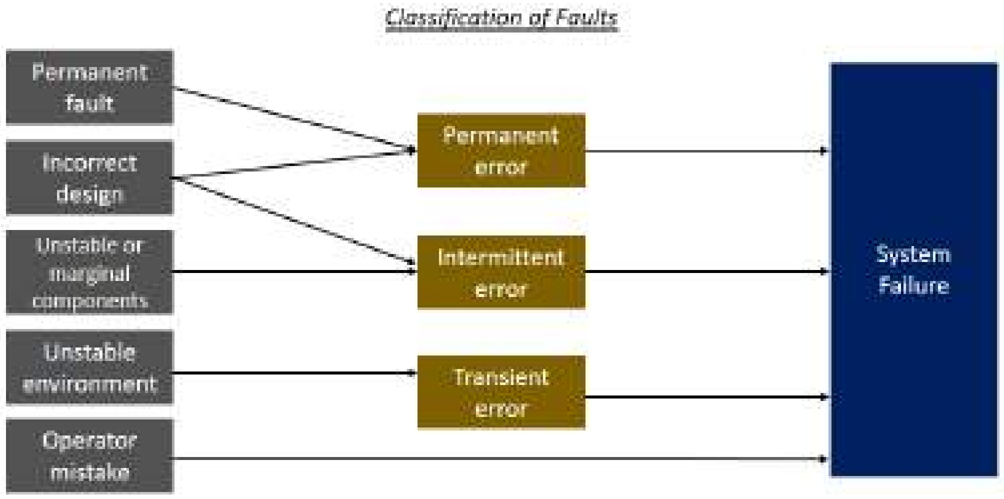


Fig : Sources of System Failure

2. **Reliability and Availability**

   - **Reliability**:
     - A measure of success with which a system conforms to some authoritative specification of its behavior.
     - Probability that the system has not experienced any failures within a given time period.

   - **Availability**:
     - The fraction of the time that a system meets its specification.
     - The probability that the system is operational at a given time *t*.

- ***Reliability*** refers to the probability that the system under consideration does not experience any failures in a given time interval. It is typically used to describe systems that cannot be repaired (as in space-based computers), or where the operation of the system is so critical that no downtime for repair can be tolerated.
- Formally, the reliability of a system, $R(t)$, is defined as the following conditional probability:

$$R(t) = \Pr\{0 \text{ failures in time } [0,t] \mid \text{no failures at } t = 0\}$$

- If we assume that failures follow a Poisson distribution (which is usually the case for hardware), this formula reduces to

$$R(t) = \Pr\{0 \text{ failures in time } [0,t]\}$$

- Under the same assumptions, it is possible to derive that

$$\Pr\{k \text{ failures in time } [0,\ t]\} = \frac{e^{-m(t)}[m(t)]^k}{k!}$$

- where $m(t) = \int_0^t z(x)\,dx$. Here $z(t)$ is known as the *hazard function*, which gives the time-dependent failure rate of the specific hardware component under consideration.

- The probability distribution for $z(t)$ may be different for different electronic components. The expected (mean) number of failures in time $[0, t]$ can then be computed as

$$E[k] = \sum_{k=0}^{\infty} k \frac{e^{-m(t)}[m(t)]^k}{k!} = m(t)$$

and the variance as

$$Var[k] = E[k^2] - (E[k])^2 = m(t)$$

Given these values, $R(t)$ can be written as

$$R(t) = e^{-m(t)}$$

- Note that the reliability equation above is written for one component of the system. For a system that consists of n non-redundant components (i.e., they all have to function properly for the system to work) whose failures are independent, the overall system reliability can be written as

$$R_{sys}(t) = \prod_{i=1}^{n} R_i(t)$$

- ***Availability*, $A(t)$,** refers to the probability that the system is operational according to its specification at a given point in time t.
- A number of failures may have occurred prior to time t, but if they have all been repaired, the system is available at time t. Obviously, availability refers to systems that can be repaired.
- If one looks at the limit of availability as time goes to infinity, it refers to the expected percentage of time that the system under consideration is available to perform useful computations.
- Availability can be used as some measure of "goodness" for those systems that can be repaired and which can be out of service for short periods of time during repair.
- Reliability and availability of a system are considered to be contradictory objectives.
- It is usually accepted that it is easier to develop highly available systems as opposed to highly reliable systems.
- If we assume that failures follow a Poisson distribution with a failure rate λ, and that repair time is exponential with a mean repair time of $1/\mu$, the steady-state availability of a system can be written as

$$A = \frac{\mu}{\lambda + \mu}$$

3. Mean Time between Failures/Mean Time to Repair

- Two single-parameter measures have become more popular than the reliability and availability functions given above to model the behavior of systems.
- These two measures used are
- *Mean Time Between Failures* (MTBF)
- *Mean Time To Repair* (MTTR).
- MTBF is the expected time between subsequent failures in a system with repair.
- MTBF can be calculated either from empirical data or from the reliability function as

$$\text{MTBF} = \int_0^{\infty} R(t)\, dt$$

- Since $R(t)$ is related to the system failure rate, there is a direct relationship between MTBF and the failure rate of a system.
- MTTR is the expected time to repair a failed system. It is related to the repair rate as MTBF is related to the failure rate.
- Using these two metrics, the steady-state availability of a system with exponential failure and repair rates can be specified as

$$A = \frac{\text{MTBF}}{\text{MTBF} + \text{MTTR}}$$

- System failures may be *latent*, in that a failure is typically detected some time after its occurrence.
- This period is called *error latency*, and the average error latency time over a number of identical systems is called *mean time to detect* (MTTD).

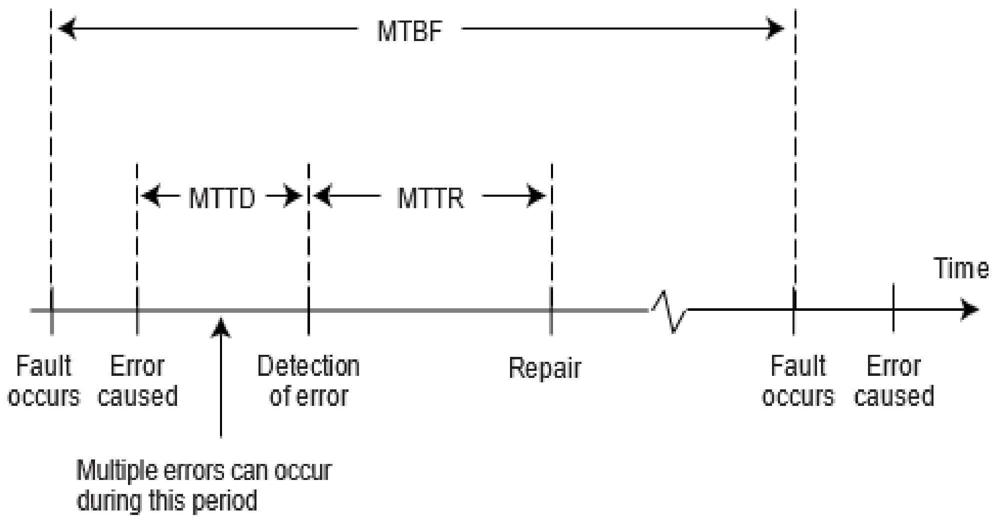


Fig. Occurrence of Events over Time

Fault-Tolerance in Distributed Systems

In distributed systems, three types of problems occur. All these three types of problems are related.

- **Fault:** Fault is defined as a weakness or shortcoming in the system or any hardware and software component. The presence of fault can lead to error and failure.
- **Errors:** Errors are incorrect results due to the presence of faults.
- **Failure:** Failure is the outcome where the assigned goal is not achieved.

Types of Faults

There are three types of Faults :

1. **Transient Faults**
2. **Intermittent Faults**
3. **Permanent Faults**

Transient Faults:

- Transient Faults are the type of faults that occur once and then disappear.
- These types of faults do not harm the system to a great extent but are very difficult to find or locate.
- Processor fault is an example of transient fault.

Intermittent Faults:

- Intermittent Faults are the type of faults that come again and again.
- Such as once the fault occurs it vanishes upon itself and then reappears again.
- An example of intermittent fault is when the working computer hangs up.

Permanent Faults:

- Permanent Faults are the type of faults that remain in the system until the component is replaced by another.
- These types of faults can cause very severe damage to the system but are easy to identify.
- A burnt-out chip is an example of a permanent Fault.

What is Fault Tolerance?

- Fault Tolerance is defined as the ability of the system to function properly even in the presence of any failure.
- Distributed systems consist of multiple components due to which there is a high risk of faults occurring.
- Due to the presence of faults, the overall performance may degrade.

Need for Fault Tolerance in Distributed Systems

Fault Tolerance is required in order to provide below four features.

1. **Availability:** Availability is defined as the property where the system is readily available for its use at any time.

2. **Reliability:** Reliability is defined as the property where the system can work continuously without any failure.

3. **Safety:** Safety is defined as the property where the system can remain safe from unauthorized access even if any failure occurs.

4. **Maintainability:** Maintainability is defined as the property states that how easily and fastly the failed node or system can be repaired.

Phases of Fault Tolerance in Distributed Systems

- In order to implement the techniques for fault tolerance in distributed systems, the design, configuration and relevant applications need to be considered.
- Below are the phases carried out for fault tolerance in any distributed systems.

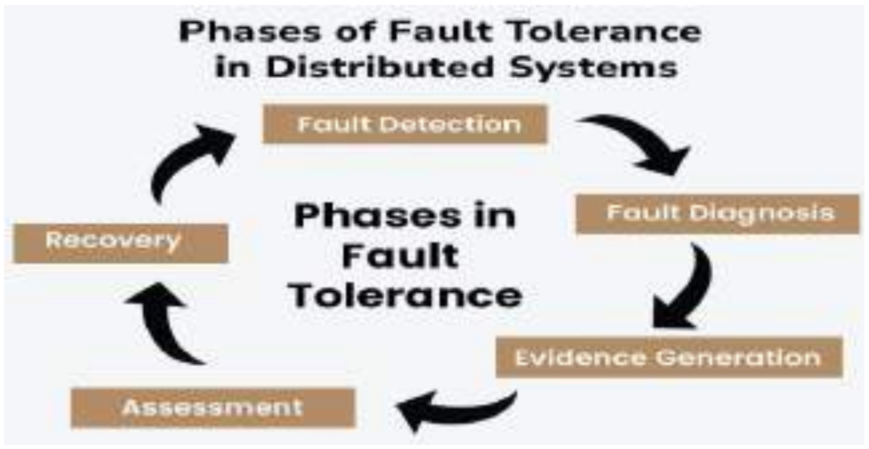

Phases of Fault Tolerance in Distributed Systems
Fault Detection
Fault Diagnosis
Evidence Generation
Assessment
Recovery
Phases in Fault Tolerance

Fig : Phases of Fault Tolerance in Distributed Systems

1. Fault Detection

- Fault Detection is the first phase where the system is monitored continuously.
- The outcomes are being compared with the expected output.
- During monitoring if any faults are identified they are being notified.
- These faults can occur due to various reasons such as hardware failure, network failure, and software issues.
- The main aim of the first phase is to detect these faults as soon as they occur so that the work being assigned will not be delayed.

2. Fault Diagnosis

- Fault diagnosis is the process where the fault that is identified in the first phase will be diagnosed properly in order to get the root cause and possible nature of the faults.
- Fault diagnosis can be done manually by the administrator or by using automated Techniques in order to solve the fault and perform the given task.

3. Evidence Generation

- Evidence generation is defined as the process where the report of the fault is prepared based on the diagnosis done in an earlier phase.
- This report involves the details of the causes of the fault, the nature of faults, the solutions that can be used for fixing, and other alternatives and preventions that need to be considered.

4. Assessment

- Assessment is the process where the damages caused by the faults are analyzed.
- It can be determined with the help of messages that are being passed from the component that has encountered the fault.
- Based on the assessment further decisions are made.

5. Recovery

- Recovery is the process where the aim is to make the system fault free.
- It is the step to make the system fault free and restore it to state forward recovery and backup recovery.
- Some of the common recovery techniques such as reconfiguration and resynchronization can be used.

Types of Fault Tolerance in Distributed Systems

1. **Hardware Fault Tolerance:**
   - Hardware Fault Tolerance involves keeping a backup plan for hardware devices such as memory, hard disk, CPU, and other hardware peripheral devices.
   - Hardware Fault Tolerance is a type of fault tolerance that does not examine faults and runtime errors but can only provide hardware backup.
   - The two different approaches that are used in Hardware Fault Tolerance are fault-masking and dynamic recovery.

2. **Software Fault Tolerance:**
   - Software Fault Tolerance is a type of fault tolerance where dedicated software is used in order to detect invalid output, runtime, and programming errors.
   - Software Fault Tolerance makes use of static and dynamic methods for detecting and providing the solution.
   - Software Fault Tolerance also consists of additional data points such as recovery rollback and checkpoints.

3. **System Fault Tolerance:**
   - System Fault Tolerance is a type of fault tolerance that consists of a whole system.
   - It has the advantage that it not only stores the checkpoints but also the memory block, and program checkpoints and detects the errors in applications automatically.
   - If the system encounters any type of fault or error it does provide the required mechanism for the solution.
   - Thus system fault tolerance is reliable and efficient.

Failures in Distributed DBMS

- Designing a reliable system that can recover from failures requires identifying the types of failures with which the system has to deal.
- In a distributed database system, we need to deal with four types of failures:
  1. Transaction failures (aborts)
  2. Site (system) failures
  3. Media (disk) failures
  4. Communication line failures

1. Transaction Failures (Aborts)

- Transactions can fail for a number of reasons.
- Failure can be due to an error in the transaction caused by incorrect input data as well as the detection of a present or potential deadlock.
- Some concurrency control algorithms do not permit a transaction to proceed or even to wait if the data that they attempt to access are currently being accessed by another transaction. This might also be considered a failure.
- The usual approach to take in cases of transaction failure is to *abort* the transaction, thus resetting the database to its state prior to the start of this transaction.

2. **Site (System) Failures**

- The reasons for system failure can be traced back to a hardware or to a software failure.
- The system failure is always assumed to result in the loss of main memory contents.
- Therefore, any part of the database that was in main memory buffers is lost as a result of a system failure.
- The database that is stored in secondary storage is assumed to be safe and correct.
- In distributed database terminology, system failures are typically referred to as *site failures*, since they result in the failed site being unreachable from other sites in the distributed system.
- We typically differentiate between partial and total failures in a distributed system.
- *Total failure* refers to the simultaneous failure of all sites in the distributed system;
- *partial failure* indicates the failure of only some sites while the others remain operational.

3. **Media (Disk) Failures**

- *Media failure* refers to the failures of the secondary storage devices that store the database.
- Such failures may be due to operating system errors, as well as to hardware faults such as head crashes or controller failures.
- The important point from the perspective of DBMS reliability is that all or part of the database that is on the secondary storage is considered to be destroyed and inaccessible.
- Media failures are frequently treated as problems local to one site and therefore not specifically addressed in the reliability mechanisms of distributed DBMSs.

4. **Communication line failures**

- The three types of failures described above are common to both centralized and distributed DBMSs.
- Communication failures are unique to the distributed case.
- There are a number of types of communication failures.
- The most common ones are the errors in the messages, improperly ordered messages, lost (or undeliverable) messages, and communication line failures.
- If a communication line fails, in addition to losing the message(s) in transit, it may also divide the network into two or more disjoint groups.
- This is called ***network partitioning***. If the network is partitioned, the sites in each partition may continue to operate.
- Executing transactions that access data stored in multiple partitions becomes a major issue.
- The term for the failure of the communication network to deliver messages and the confirmations within this period is ***performance failure***.
- It needs to be handled within the reliability protocols for distributed DBMSs.

Local & Distributed Reliability Protocols

Local Reliability Protocols

- Here, we discuss the functions performed by the **Local Recovery Manager** (LRM) that exists at each site.

- These functions maintain the atomicity and durability properties of local transactions.
- They relate to the execution of the commands that are passed to the LRM, which are **begin transaction**, **read**, **write**, **commit**, and **abort**.
- When the LRM wants to read a page of data on behalf of a transaction it uses a fetch command, indicating the page that it wants to read.

1. Architectural Considerations

Volatile database

- The database buffer manager keeps some of the recently accessed data in main memory buffers.
- This is done to enhance access performance.
- Typically, the buffer is divided into pages that are of the same size as the stable database pages.
- The part of the database that is in the database buffer is called the ***volatile database***.
- It is important to note that the LRM executes the operations on behalf of a transaction only on the volatile database, which, at a later time, is written back to the stable database.

Stable database

- We assume that the database is stored permanently on secondary storage, which in this context is called the *stable storage*.
- The stability of this storage medium is due to its robustness to failures.
- A stable storage device would experience considerably less-frequent failures than would a non-stable storage device.
- In today's technology, stable storage is typically implemented by means of duplexed magnetic disks which store duplicate copies of data that are always kept mutually consistent (i.e., the copies are identical).
- We call the version of the database that is kept on stable storage the ***stable database***.
- The unit of storage and access of the stable database is typically a *page*.

Buffer Manager

- The buffer manager acts as a conduit for all access to the database via the buffers that it manages.
- It provides this function by fulfilling three tasks:

  1. *Searching* the buffer pool for a given page;
  2. If it is not found in the buffer, *allocating* a free buffer page and *loading* the buffer page with a data page that is brought in from secondary storage;
  3. If no free buffer pages are available, choosing a buffer page for *replacement*.

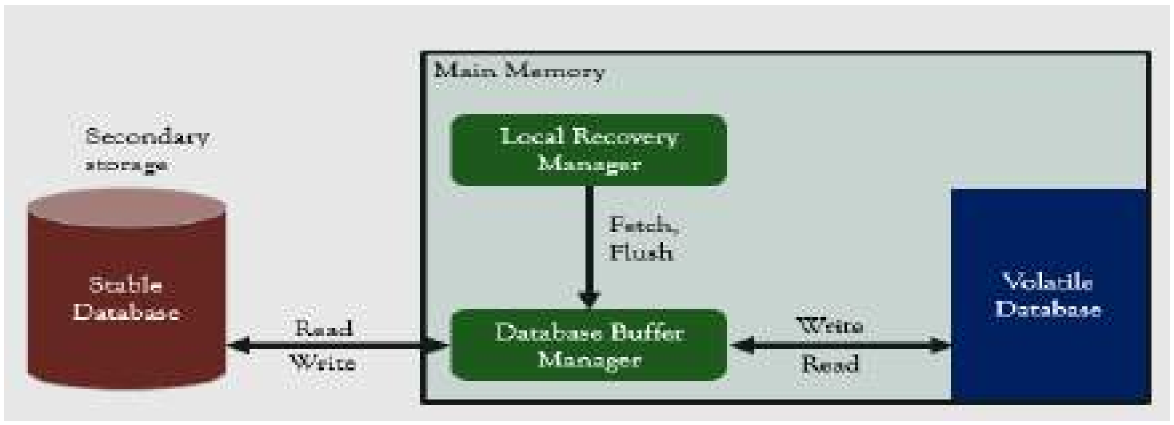


Fig. Interface Between the Local Recovery Manager and the Buffer Manager

2. **Recovery Information**

- When a system failure occurs, the volatile database is lost.
- Therefore, the DBMS has to maintain some information about its state at the time of the failure in order to be able to bring the database to the state that it was in when the failure occurred. We call this information the ***recovery information***.
- The recovery information that the system maintains is dependent on the method of executing updates.
- Two possibilities are in-place updating and out-of-place updating.
- ***In-place updating*** physically changes the value of the data item in the stable database. As a result, the previous values are lost.
- ***Out-of-place updating*** does not change the value of the data item in the stable database but maintains the new value separately.

In-Place Update Recovery Information

Logging:

- Since in-place updates cause previous values of the affected data items to be lost, it is necessary to keep enough information about the database state changes to facilitate the recovery of the database to a consistent state following a failure.
- This information is typically maintained in a ***database log***.
- Thus each update transaction not only changes the database but the change is also recorded in the database log.
- The log contains information necessary to recover the database state following a failure.

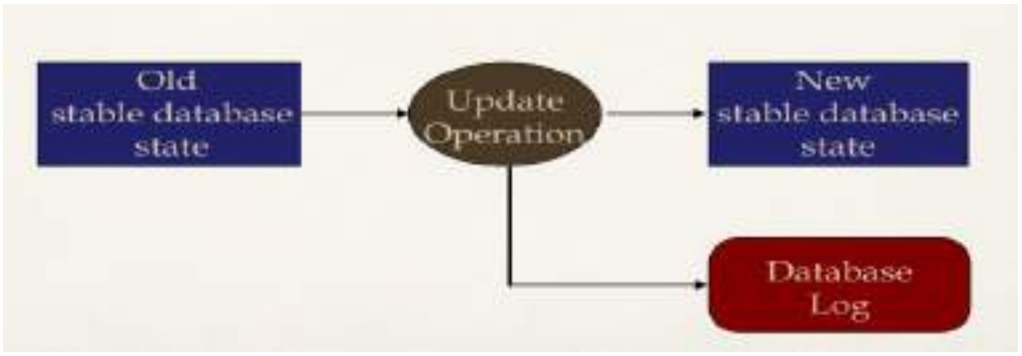


Fig. Update Operation Execution

- Assume that the LRM and buffer manager algorithms are such that the buffer pages are written back to the stable database only when the buffer manager needs new buffer space.
- The **flush** command is not used by the LRM and the decision to write back the pages into the stable database is taken at the discretion of the buffer manager.
- Now consider that a transaction T_1 had completed (i.e., committed) before the failure occurred.
- The durability property of transactions would require that the effect os T_1 be reflected in the database.
- It is possible that the volatile database pages that have been updated by T_1 may not have been written back to the stable database at the time of the failure.
- Therefore, upon recovery, it is important to be able to *redo* the operations of T_1.
- This requires some information to be stored in the database log about the effects of T_1.
- Given this information, it is possible to recover the database from its "old" state to the "new" state that reflects the effects of T_1.

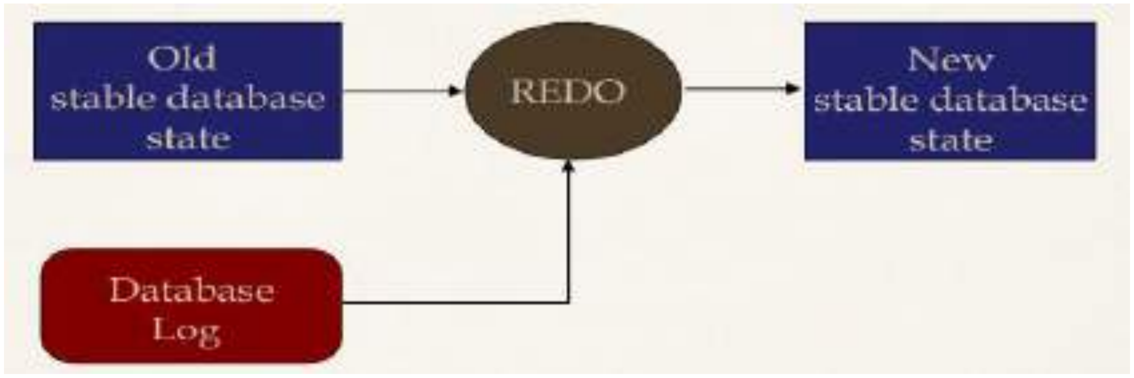


Fig. REDO Action

- Now consider another transaction, T_2, that was still running when the failure occurred.
- The atomicity property would dictate that the stable database not contain any effects of T_2.
- It is possible that the buffer manager may have had to write into the stable database some of the volatile database pages that have been updated by T_2.
- Upon recovery from failures it is necessary to *undo* the operations of T_2.
- Thus the recovery information should include sufficient data to permit the undo by taking the "new" database state that reflects partial effects of T_2 and recovers the "old" state that existed at the start of T_2.
- We should indicate that the undo and redo actions are assumed to be idempotent.
- Their repeated application to a transaction would be equivalent to performing them once.
- The undo/redo actions form the basis of different methods of executing the commit commands.

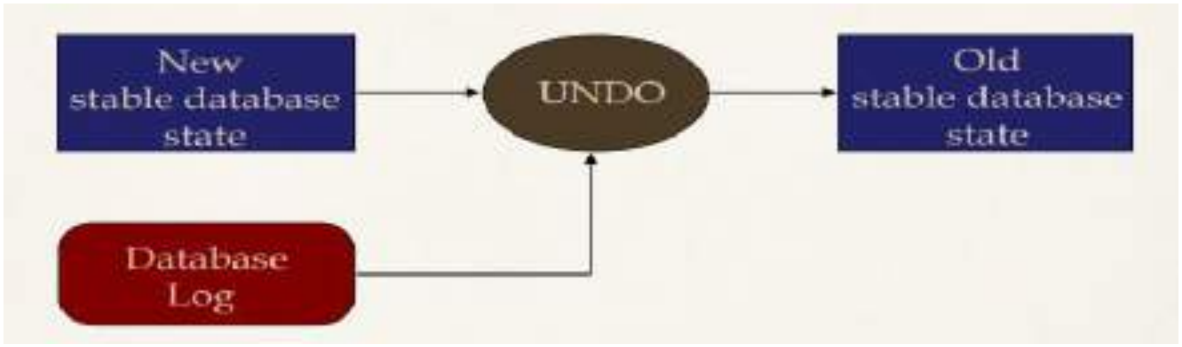


Fig. UNDO Action

- The contents of the log may differ according to the implementation.
- The following minimal information for each transaction is contained in almost all database logs:
- A begin transaction record, the value of the data item before the update (called the *before image*)
- The updated value of the data item (called the *after image*)
- A termination record indicating the transaction termination condition (commit, abort).
- The granularity of the before and after images may be different, as it is possible to log entire pages or some smaller unit.
- As an alternative to this form of *state logging*, *operational logging*, may be supported where the operations that cause changes to the database are logged rather than the before and after images.
- The log is also maintained in main memory buffers (called *log buffers*) and written back to stable storage (called *stable log*) similar to the database buffer pages.
- The log pages can be written to stable storage in one of two ways.
- They can be written *synchronously* (more commonly known as *forcing a log*) where the addition of each log record requires that the log be moved from main memory to stable storage.
- They can also be written *asynchronously*, where the log is moved to stable storage either at periodic intervals or when the buffer fills up.
- When the log is written synchronously, the execution of the transaction is suspended until the write is complete. This adds some delay to the response-time performance of the transaction.
- On the other hand, if a failure occurs immediately after a forced write, it is relatively easy to recover to a consistent database state.

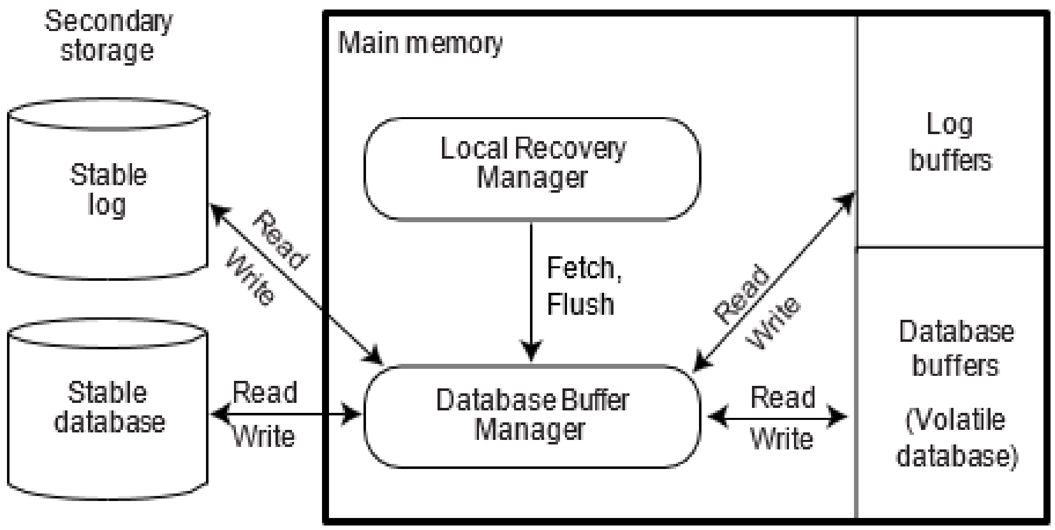


Fig. Logging Interface

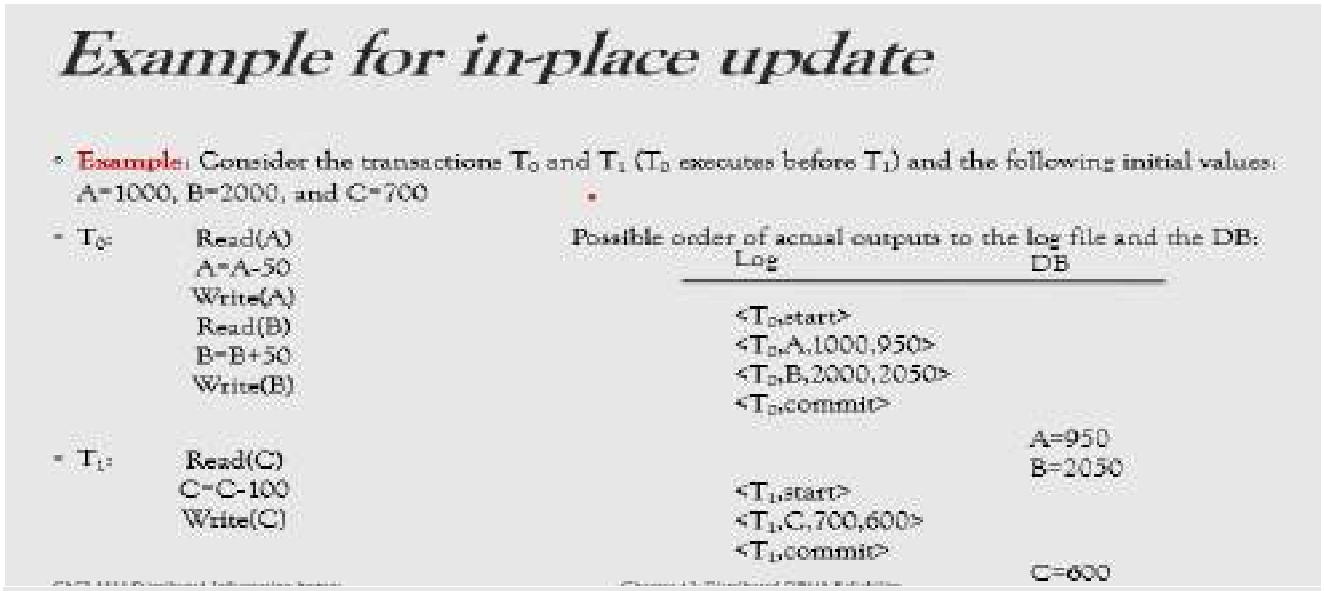
- Whether the log is written synchronously or asynchronously, one very important protocol has to be observed in maintaining logs.
- Consider a case where the updates to the database are written into the stable storage before the log is modified in stable storage to reflect the update.
- If a failure occurs before the log is written, the database will remain in updated form, but the log will not indicate the update that makes it impossible to recover the database to a consistent and up-to-date state. Therefore, the stable log is always updated prior to the updating of the stable database.
- This is known as the *write-ahead logging* (WAL) protocol and can be precisely specified as follows:

Write-Ahead Log protocol

1. Before a stable database is updated (perhaps due to actions of a yet uncommitted transaction), the before images should be stored in the stable log. This facilitates undo.
2. When a transaction commits, the after images have to be stored in the stable log prior to the updating of the stable database. This facilitates redo.

Notice:

- If a system crashes before a transaction is committed, then all the operations must be undone. Only need the before images (*undo portion* of the log).
- Once a transaction is committed, some of its actions might have to be redone. Need the after images (*redo portion* of the log).



Example for in-place update

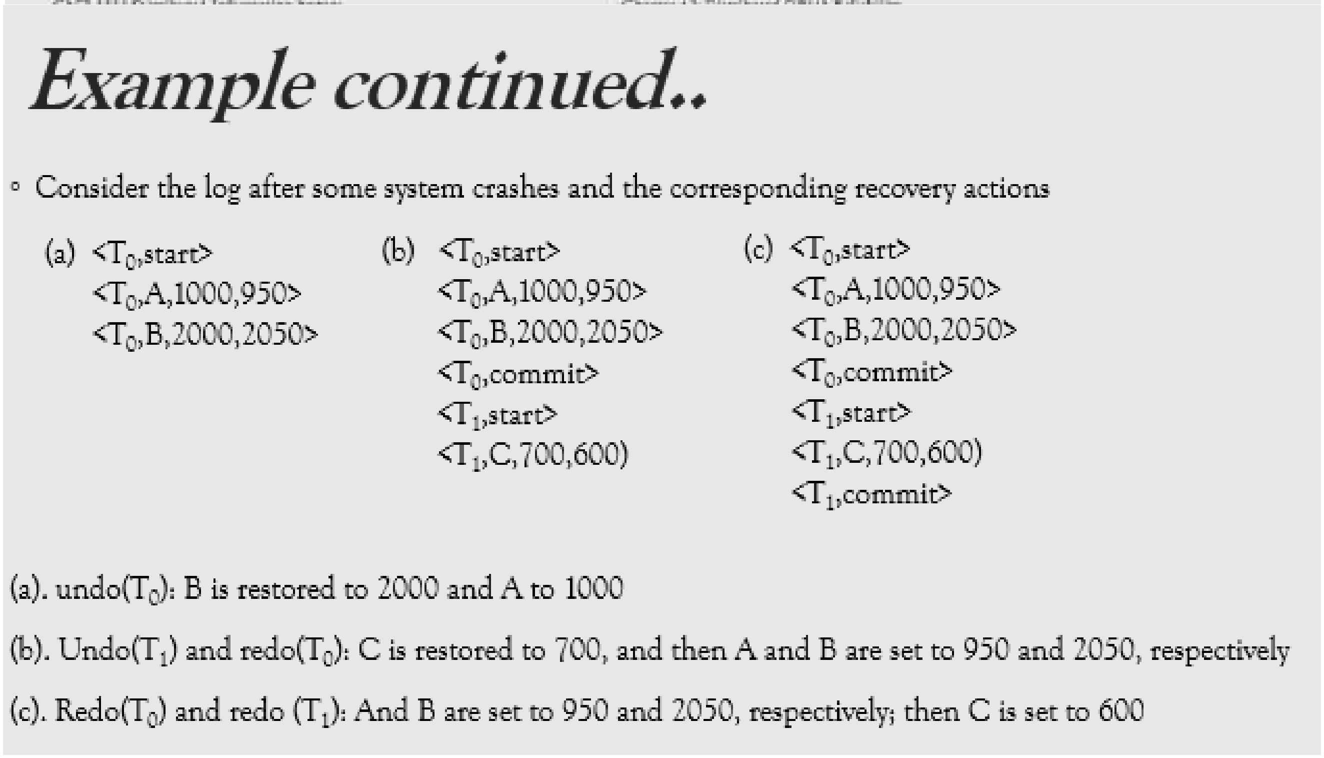
- Example: Consider the transactions T_0 and T_1 (T_0 executes before T_1) and the following initial values: A=1000, B=2000, and C=700

```
- T0:   Read(A)
        A=A-50
        Write(A)
        Read(B)
        B=B+50
        Write(B)

- T1:   Read(C)
        C=C-100
        Write(C)
```

Possible order of actual outputs to the log file and the DB:

Log	DB
$<T_0$,start>	
$<T_0$,A,1000,950>	
$<T_0$,B,2000,2050>	
$<T_0$,commit>	
	A=950
	B=2050
$<T_1$,start>	
$<T_1$,C,700,600>	
$<T_1$,commit>	
	C=600



Example continued..

- Consider the log after some system crashes and the corresponding recovery actions

(a)	(b)	(c)
$<T_0$,start>	$<T_0$,start>	$<T_0$,start>
$<T_0$,A,1000,950>	$<T_0$,A,1000,950>	$<T_0$,A,1000,950>
$<T_0$,B,2000,2050>	$<T_0$,B,2000,2050>	$<T_0$,B,2000,2050>
	$<T_0$,commit>	$<T_0$,commit>
	$<T_1$,start>	$<T_1$,start>
	$<T_1$,C,700,600)	$<T_1$,C,700,600)
		$<T_1$,commit>

(a). undo(T_0): B is restored to 2000 and A to 1000

(b). Undo(T_1) and redo(T_0): C is restored to 700, and then A and B are set to 950 and 2050, respectively

(c). Redo(T_0) and redo (T_1): And B are set to 950 and 2050, respectively; then C is set to 600

Out-of-Place Update Recovery Information

Shadowing:

- Typical techniques for out-of-place updating are *shadowing* and *differential files*.
- Shadowing uses duplicate stable storage pages in executing updates.
- Thus every time an update is made, the old stable storage page, called the *shadow page*, is left intact and a new page with the updated data item values is written into the stable database.
- The access path data structures are updated to point to the new page, which contains the current data so that subsequent accesses are to this page.
- The old stable storage page is retained for recovery purposes (to perform undo).
- Recovery based on shadow paging is implemented in System R's recovery manager.
- This implementation uses shadowing together with logging.

The Differential File

- The method maintains each stable database file as a read-only file.
- In addition, it maintains a corresponding read-write differential file that stores the changes to that file.
- Given a logical database file *F*, let us denote its read-only part as *FR* and its corresponding differential file as *DF*.
- *DF* consists of two parts:
- Insertions part, which stores the insertions to *F* , denoted DF^+.
- Deletions part, denoted DF^-.
- All updates are treated as the deletion of the old value and the insertion of a new one.
- Thus each logical file *F* is considered to be a view defined as $F = (FR \cup DF^+) - DF^-$.
- Periodically, the differential file needs to be merged with the read-only base file.

3. Execution of LRM Commands

- Recall that there are five commands that form the interface to the LRM.
- These are the **begin transaction, read, write, commit**, and **abort** commands.
- In this section we introduce a sixth interface command to the LRM: **recover**.
- The **recover** command is the interface that the operating system has to the LRM.
- It is used during recovery from system failures when the operating system asks the DBMS to recover the database to the state that existed when the failure occurred.
- Now we present the execution methods of the **begin transaction, read**, **write, commit** and **recovery** commands.

Begin transaction, Read, and Write Commands

Begin transaction.

- Assume that it causes the LRM to write a begin transaction record into the log.
- This is an assumption made for convenience of discussion; in reality, writing of the begin transaction record may be delayed until the first **write** to improve performance by reducing I/O.

Read.

- The **read** command specifies a data item.
- The LRM tries to read the specified data item from the buffer pages that belong to the transaction.
- If the data item is not in one of these pages, it issues a **fetch** command to the buffer manager in order to make the data available.
- Upon reading the data, the LRM returns it to the scheduler.

Write.

- The **write** command specifies the data item and the new value.
- As with a read command, if the data item is available in the buffers of the transaction, its value is modified in the database buffers (i.e., the volatile database).
- If it is not in the private buffer pages, a **fetch** command is issued to the buffer manager, and the data is made available and updated.
- The before image of the data page, as well as its after image, are recorded in the log.
- The local recovery manager then informs the scheduler that the operation has been completed successfully.

Execution strategies

Dependent upon

- Can the buffer manager decide to write some of the buffer pages being accessed by a transaction into stable storage or does it wait for LRM to instruct it?
  - fix/no-fix decision
- Does the LRM force the buffer manager to write certain buffer pages into stable database at the end of a transaction's execution?
  - flush/no-flush decision

Possible execution strategies:

- no-fix/no-flush
- no-fix/flush
- fix/no-flush
- fix/flush

No-fix/No-flush

This type of LRM algorithm is called a redo/undo algorithm since it requires performing both the redo and undo operations upon recovery.

Abort

- Buffer manager may have written some of the updated pages into stable database
- LRM performs **transaction undo** (or **partial undo**)

Commit

- LRM writes an “end_of_transaction” record into the log.

Recover

- For those transactions that have both a “begin_transaction” and an “end_of_transaction” record in the log, a partial redo is initiated by LRM
- For those transactions that only have a “begin_transaction” in the log, a **global undo** is executed by LRM

No-fix/flush

The LRM algorithms that use this strategy are called undo/no-redo.

Abort

- Buffer manager may have written some of the updated pages into stable database
- LRM performs transaction undo (or partial undo)

Commit

- LRM issues a **flush** command to the buffer manager for all updated pages
- LRM writes an “end_of_transaction” record into the log.

Recover

- No need to perform redo
- Perform global undo

Fix/No-Flush

In this case the LRM controls the writing of the volatile database pages into stable storage. This precludes the need for a global undo operation and is therefore called a redo/no-undo.

Abort

- None of the updated pages have been written into stable database
- Release the fixed pages

Commit

- LRM writes an “end_of_transaction” record into the log.
- LRM sends an **unfix** command to the buffer manager for all pages that were previously **fixed**

Recover

- Perform partial redo
- No need to perform global undo

Fix/Flush

This is the case where the LRM forces the buffer manager to write the updated volatile database pages into the stable database at precisely the commit point—not before and not after. This strategy is called no-undo/no-redo.

Abort

- None of the updated pages have been written into stable database
- Release the fixed pages

Commit (the following must be done atomically)

- LRM issues a **flush** command to the buffer manager for all updated pages

  - LRM sends an **unfix** command to the buffer manager for all pages that were previously **fixed**
  - LRM writes an "end_of_transaction" record into the log.

Recover

  - No need to do anything

4. Checkpointing

- In most of the LRM implementation strategies, the execution of the recovery action requires searching the entire log.
- This is a significant overhead because the LRM is trying to find all the transactions that need to be undone and redone.
- The overhead can be reduced if it is possible to build a wall which signifies that the database at that point is up-to-date and consistent.
- In that case, the redo has to start from that point on and the undo only has to go back to that point. This process of building the wall is called *checkpointing*.
- Checkpointing is achieved in three steps :

  1. Write a begin checkpoint record into the log.
  2. Collect the checkpoint data into the stable storage.
  3. Write an end checkpoint record into the log.

- The first and the third steps enforce the atomicity of the checkpointing operation.
- If a system failure occurs during check pointing, the recovery process will not find an end checkpoint record and will consider checkpointing not completed.
- There are a number of different alternatives for the data that is collected in Step 2, how it is collected, and where it is stored.
- We will consider one example here, called *transaction-consistent checkpointing*.
- The checkpointing starts by writing the begin checkpoint record in the log and stopping the acceptance of any new transactions by the LRM.
- Once the active transactions are all completed, all the updated volatile database pages are flushed to the stable database followed by the insertion of an end checkpoint record into the log.
- In this case, the redo action only needs to start from the end checkpoint record in the log.
- The undo action can go the reverse direction, starting from the end of the log and stopping at the end checkpoint record.
- Transaction-consistent checkpointing is not the most efficient algorithm, since a significant delay is experienced by all the transactions.
- There are alternative check- pointing schemes such as action-consistent checkpoints, fuzzy checkpoints, and others.

5. Handling Media Failures

- As we mentioned before, the previous discussion on centralized recovery considered non-media failures, where the database as well as the log stored in the stable storage survive the failure.
- Media failures may either be quite catastrophic, causing the loss of the stable database or of the stable

log, or they can simply result in partial loss of the database or the log.

- The methods that have been devised for dealing with this situation are again based on duplexing.
- To cope with catastrophic media failures, an *archive* copy of both the database and the log is maintained on a different (tertiary) storage medium, which is typically the magnetic tape or CD-ROM.
- Thus the DBMS deals with three levels of memory hierarchy: the main memory, random access disk storage, and magnetic tape.
- To deal with less catastrophic failures, having duplicate copies of the database and log may be sufficient.
- When a media failure occurs, the database is recovered from the archive copy by redoing and undoing the transactions as stored in the archive log.
- The real question is how the archive database is stored.
- If we consider the large sizes of current databases, the overhead of writing the entire database to tertiary storage is significant.
- Two methods that have been proposed for dealing with this are
  - To perform the archiving activity concurrent with normal processing
  - To archive the database incrementally as changes occur so that each archive version contains only the changes that have occurred since the previous archiving.

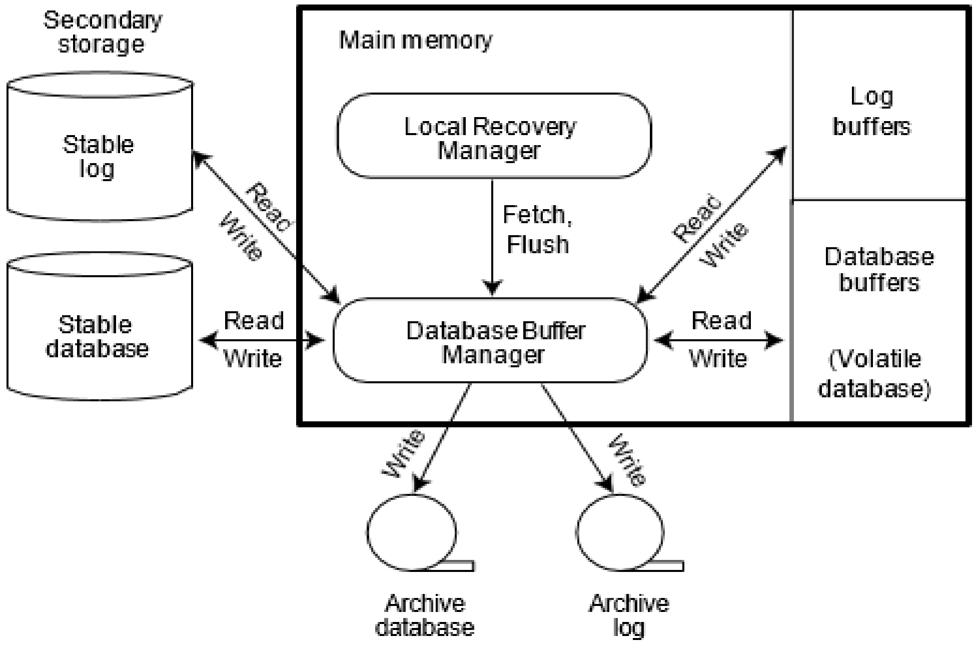


Fig : Full Memory Hierarchy Managed by LRM and BM

Distributed Reliability Protocols

- The distributed versions aim to maintain the atomicity and durability of distributed transactions that execute over a number of databases.
- The protocols address the distributed execution of the **begin transaction, read, write, abort, commit**, and **recover** commands.
- The distributed reliability protocols are implemented between the coordinator and the participants.

They are :

1. Components of Distributed Reliability Protocols
2. Two-Phase Commit Protocol

1. Components of Distributed Reliability Protocols:

- Reliability techniques consist of commit, termination, and recovery protocols
- Commit and recover commands executed differently in a distributed DBMS than centralized
- Termination protocols are unique to distributed systems
- Termination vs. recovery protocols
    - Opposite faces of recovery problem
    - Given a site failure, termination protocols address how the operational sites deal with the failure
    - Recovery protocols address procedure the process at the failed site must go through to recover its state
- Commit protocols must maintain atomicity of distributed transactions
- Ideally recovery protocols are independent – no need to consult other sites to terminate transaction

2. Two-Phase Commit Protocol

- Insists that all sites involved in the execution of a distributed transaction agree to commit the transaction before its effects are made permanent
- Scheduler issues, deadlocks necessitate synchronization
- Global commit rule:
    - If even one participant votes to abort the transaction, the coordinator must reach a global abort decision
    - If all the participants vote to commit the transaction, the coordinator must reach a global commit decision
- A participant may unilaterally abort a transaction until an affirmative vote is registered
- A participant's vote cannot be changed
- Timers used to exit from wait states

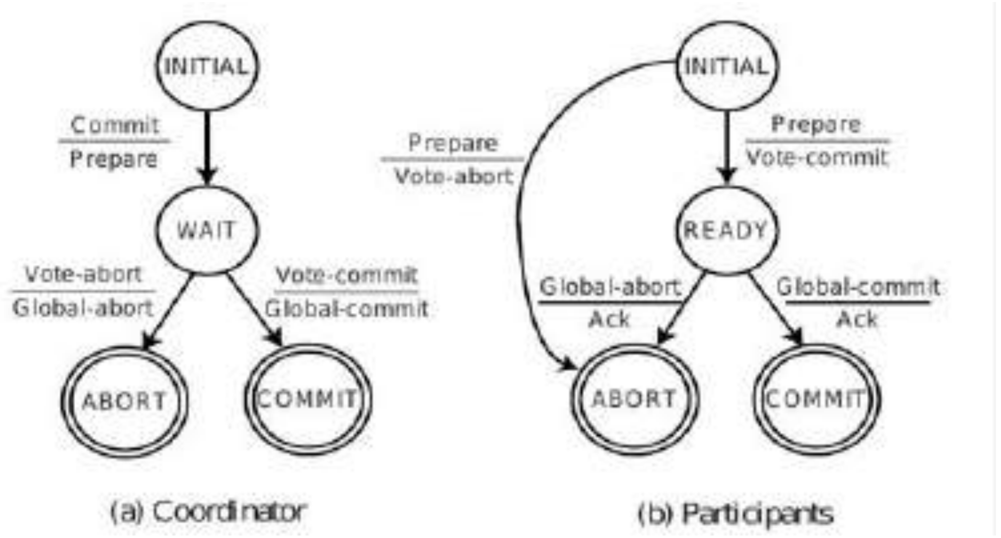


Fig : State Transitions in 2PC Protocol

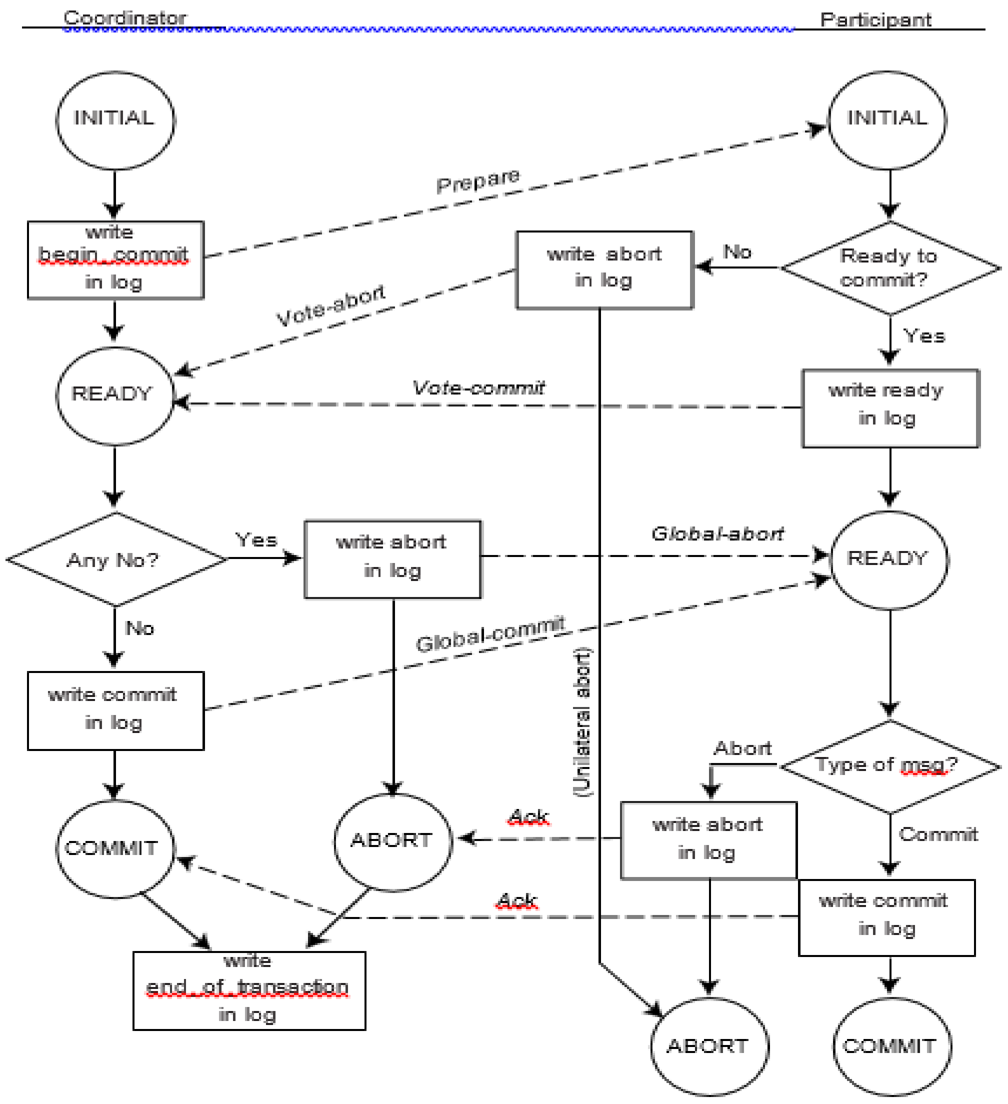


Fig.: 2PC Protocol Actions

Centralized vs. Linear/nested 2PC

- Centralized – participants do not communicate among themselves
- Linear – participants can communicate with one another
- Sites in linear system become numbered, pass messages to one another with "**vote-commit**" or "**vote-abort**" messages
- If last node decides to commit, send message back to coordinator node-by-node
- Distributed 2PC eliminates second phase of linear protocol
- Coordinator sends Prepare message to all participants
- Each participant sends its decision to all other participants and coordinator
- Participants must know identity of other participants for linear and distributed, not centralized

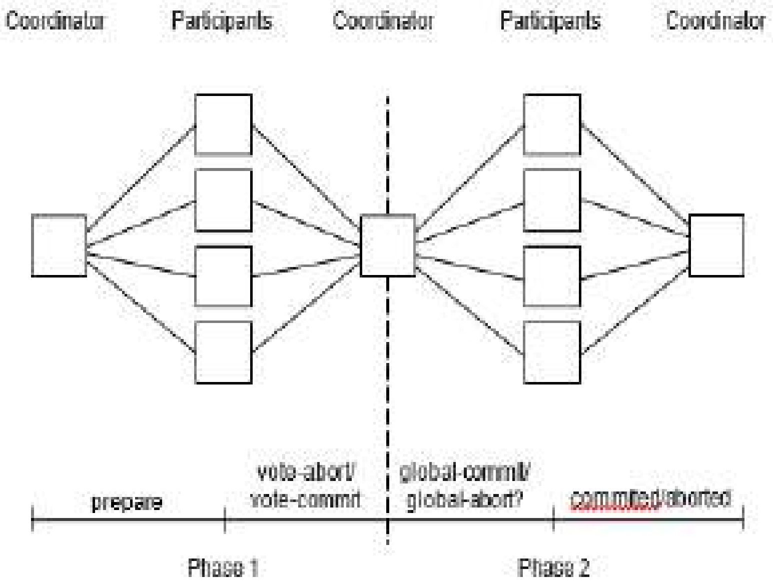


Fig. Centralized 2PC Communication Structure

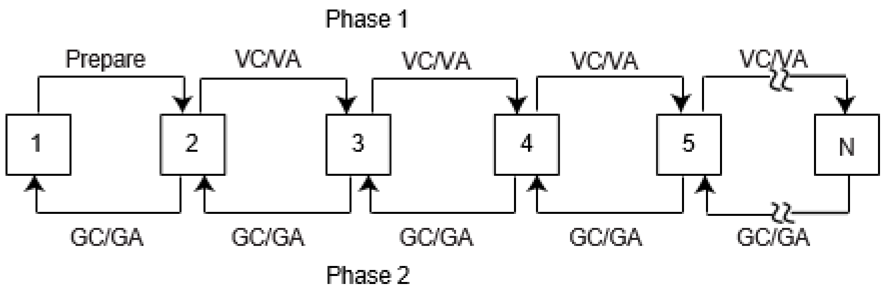


Fig. Linear 2PC Communication Structure. VC, vote.commit; VA, vote.abort; GC, global.commit; GA, global.abort.)

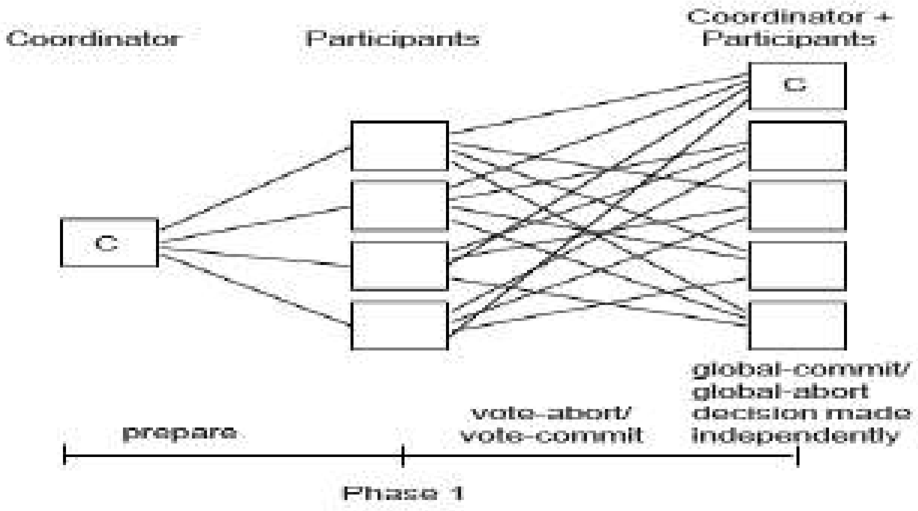

Coordinator
Participants
Coordinator + Participants
C
C
global-commit/ global-abort decision made independently
prepare
vote-abort/ vote-commit
Phase 1

Fig. Distributed 2PC Communication Structure

Variations of 2PC

- Presumed Abort 2PC Protocol
  - Participant polls coordinator and there is no information about transaction, aborts
  - Coordinator can forget about transaction immediately after aborting
  - Expected to be more efficient, saves message transmission between coordinator and participants
- Presumed Commit 2PC Protocol
  - Likewise, if no information about transaction exists, should be considered committed
  - Most transactions are expected to commit
  - Could cause inconsistency – must create a collecting record

Site Failures and Network Partitioning

Dealing with Site Failures

- Let us first set the boundaries for the existence of non-blocking termination and independent recovery protocols in the presence of site failures.
- It can formally be proven that such protocols exist when a single site fails.
- In the case of multiple site failures, however, the prospects are not as promising.
- A negative result indicates that it is not possible to design independent recovery protocols (and, therefore, non- blocking termination protocols) when multiple sites fail.
- We first develop termination and recovery protocols for the 2PC algorithm and show that 2PC is inherently blocking.
- We then proceed to the development of atomic commit protocols which are non-blocking in the case of single site failures.

1. Termination and Recovery Protocols for 2PC

Termination Protocols

- The termination protocols serve the timeouts for both the coordinator and the participant processes.
- A timeout occurs at a destination site when it cannot get an expected message from a source site within the expected time period. In this section we consider that this is due to the failure of the source site.
- The method for handling timeouts depends on the timing of failures as well as on the types of failures. We therefore need to consider failures at various points of 2PC execution.
- State transition diagram :
- The states are denoted by circles and the edges represent the state transitions.
- The terminal states are depicted by concentric circles.
- The interpretation of the labels on the edges is as follows:
- the reason for the state transition, which is a received message, is given at the top, and the message that is sent as a result of state transition is given at the bottom.

Coordinator Timeouts.

- There are three states in which the coordinator can timeout: WAIT, COMMIT, and ABORT.
- Timeouts during the last two are handled in the same manner.
- So we need to consider only two cases:

1. *Timeout in the WAIT state*. In the WAIT state, the coordinator is waiting for the local decisions of the participants. The coordinator cannot unilaterally commit the transaction since the global commit rule has not been satisfied. However, it can decide to globally abort the transaction, in which case it writes an abort record in the log and sends a “global-abort” message to all the participants.
2. *Timeout in the COMMIT or ABORT states*. In this case the coordinator is not certain that the commit or abort procedures have been completed by the local recovery managers at all of the participant sites. Thus the coordinator repeatedly sends the “global-commit” or “global-abort” commands to the sites that have not yet responded, and waits for their acknowledgement.

Participant Timeouts.

- A participant can time out in two states: INITIAL and READY.
- Let us examine both of these cases.

1. *Timeout in the INITIAL state*. In this state the participant is waiting for a “prepare” message. The coordinator must have failed in the INITIAL state. The participant can unilaterally abort the transaction following a timeout. If the “prepare” message arrives at this participant at a later time, this can be handled in one of two possible ways. Either the participant would check its log, find the abort record, and respond with a “vote-abort,” or it can simply ignore the “prepare” message. In the latter case the coordinator would time out in the WAIT state and follow the course we have discussed above.

2. *Timeout in the READY state*. In this state the participant has voted to commit the transaction but does not know the global decision of the coordinator. The participant cannot unilaterally make a decision. Since it is in the READY state, it must have voted to commit the transaction. Therefore, it cannot now change its vote and unilaterally abort it. On the other hand, it cannot unilaterally decide to commit it since it is possible that another participant may have voted to abort it. In this case the participant will remain blocked until it can learn from someone (either the coordinator or some other participant) the ultimate fate of the transaction.

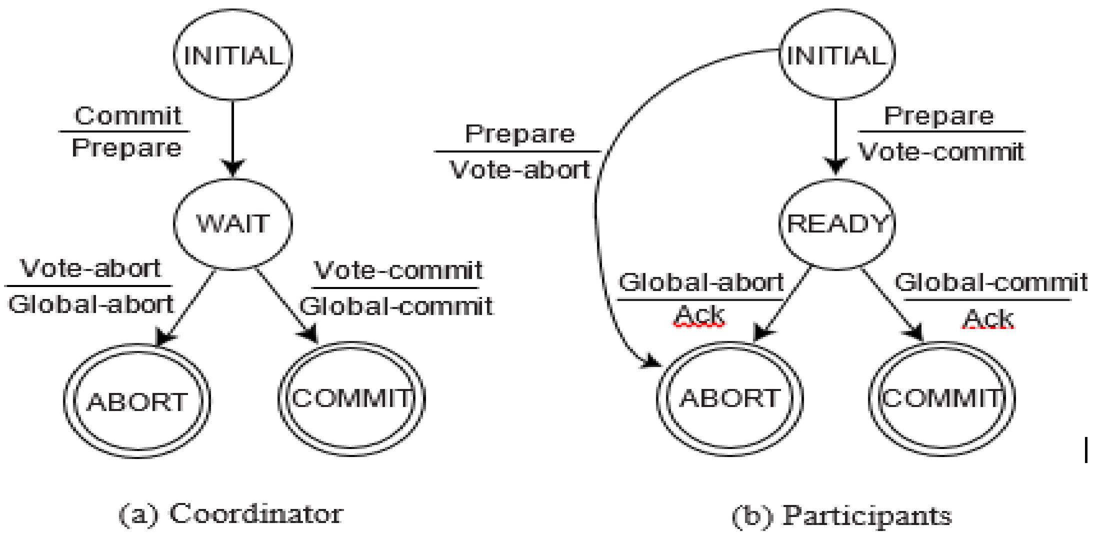
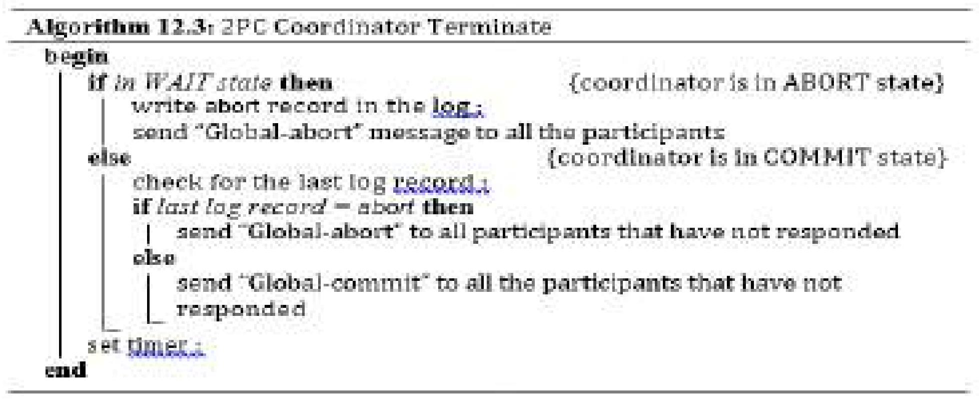


Fig. State Transitions in 2PC Protocol

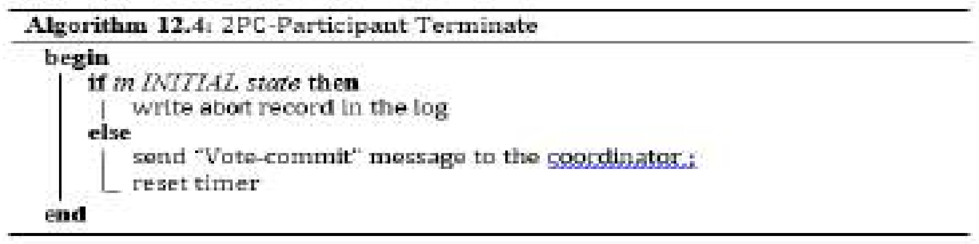
Recovery Protocols

- In the preceding section, we discussed how the 2PC protocol deals with failures from the perspective of the operational sites.
- In this section, we take the opposite viewpoint: we are interested in investigating protocols that a coordinator or participant can use to recover their states when their sites fail and then restart.
- Remember that we would like these protocols to be independent.
- In general, it is not possible to design protocols that can guarantee independent recovery while maintaining the atomicity of distributed transactions.
- This is not surprising given the fact that the termination protocols for 2PC are inherently blocking.



Algorithm 12.3: 2PC Coordinator Terminate

```
begin
    if in WAIT state then                    {coordinator is in ABORT state}
        write abort record in the log ;
        send "Global-abort" message to all the participants
    else                                     {coordinator is in COMMIT state}
        check for the last log record ;
        if last log record = abort then
            send "Global-abort" to all participants that have not responded
        else
            send "Global-commit" to all the participants that have not
            responded
    set timer ;
end
```



Algorithm 12.4: 2PC-Participant Terminate

```
begin
    if in INITIAL state then
        write abort record in the log
    else
        send "Vote-commit" message to the coordinator ;
        reset timer
end
```

Coordinator Site Failures

The following cases are possible:

1. *The coordinator fails while in the INITIAL state*. This is before the coordinator has initiated the commit procedure. Therefore, it will start the commit process upon recovery.
2. *The coordinator fails while in the WAIT state*. In this case, the coordinator has sent the "prepare" command. Upon recovery, the coordinator will restart the commit process for this transaction from the beginning by sending the "prepare" message one more time.
3. *The coordinator fails while in the COMMIT or ABORT states*. In this case, the coordinator will have informed the participants of its decision and terminated the transaction. Thus, upon recovery, it does not need to do anything if all the acknowledgments have been received. Otherwise, the termination protocol is involved.

Participant Site Failures

There are three alternatives to consider:

1. *A participant fails in the INITIAL state*. Upon recovery, the participant should abort the transaction unilaterally. Let us see why this is acceptable. Note that the coordinator will be in the INITIAL or WAIT state with respect to this transaction. If it is in the INITIAL state, it will send a "prepare" message and then move to the WAIT state. Because of the participant site's failure, it will not receive the participant's decision and will time out in that state. We have already discussed how the coordinator would handle timeouts in the WAIT state by globally aborting the transaction.
2. *A participant fails while in the READY state*. In this case the coordinator has been informed of the failed site's affirmative decision about the transaction before the failure. Upon recovery, the participant at the failed site can treat this as a timeout in the READY state and hand the incomplete transaction over to its termination protocol.
3. *A participant fails while in the ABORT or COMMIT state*. These states represent the termination conditions, so, upon recovery, the participant does not need to take any special action.

2. Three-Phase Commit Protocol

- The three-phase commit protocol (3PC) is designed as a non-blocking protocol.
- It is indeed non-blocking when failures are restricted to site failures.
- Let us first consider the necessary and sufficient conditions for designing non- blocking atomic commitment protocols.
- A commit protocol that is synchronous within one state transition is non-blocking if and only if its state transition diagram contains neither of the following:

    1. No state that is "adjacent" to both a commit and an abort state.
    2. No non-committable state that is "adjacent" to a commit state.

- The term *adjacent* here means that it is possible to go from one state to the other with a single state transition.

- Consider the COMMIT state in the 2PC protocol. If any process is in this state, we know that all the sites have voted to commit the transaction. Such states are called *committable*. There are other states in the 2PC protocol that are *non-committable*.
- The one we are interested in is the READY state, which is non-committable since the existence of a process in this state does not imply that all the processes have voted to commit the transaction.
- It is obvious that the WAIT state in the coordinator and the READY state in the participant 2PC protocol violate the non-blocking conditions we have stated above.
- Therefore, one might be able to make the following modification to the 2PC protocol to satisfy the conditions and turn it into a non-blocking protocol.
- We can add another state between the WAIT (and READY) and COMMIT states which serves as a buffer state where the process is ready to commit (if that is the final decision) but has not yet committed.
- This is called the three-phase commit protocol (3PC) because there are three state transitions from the INITIAL state to a COMMIT state.
- Observe that 3PC is also a protocol where all the states are synchronous within one state transition. Therefore, the foregoing conditions for non-blocking 2PC apply to 3PC.

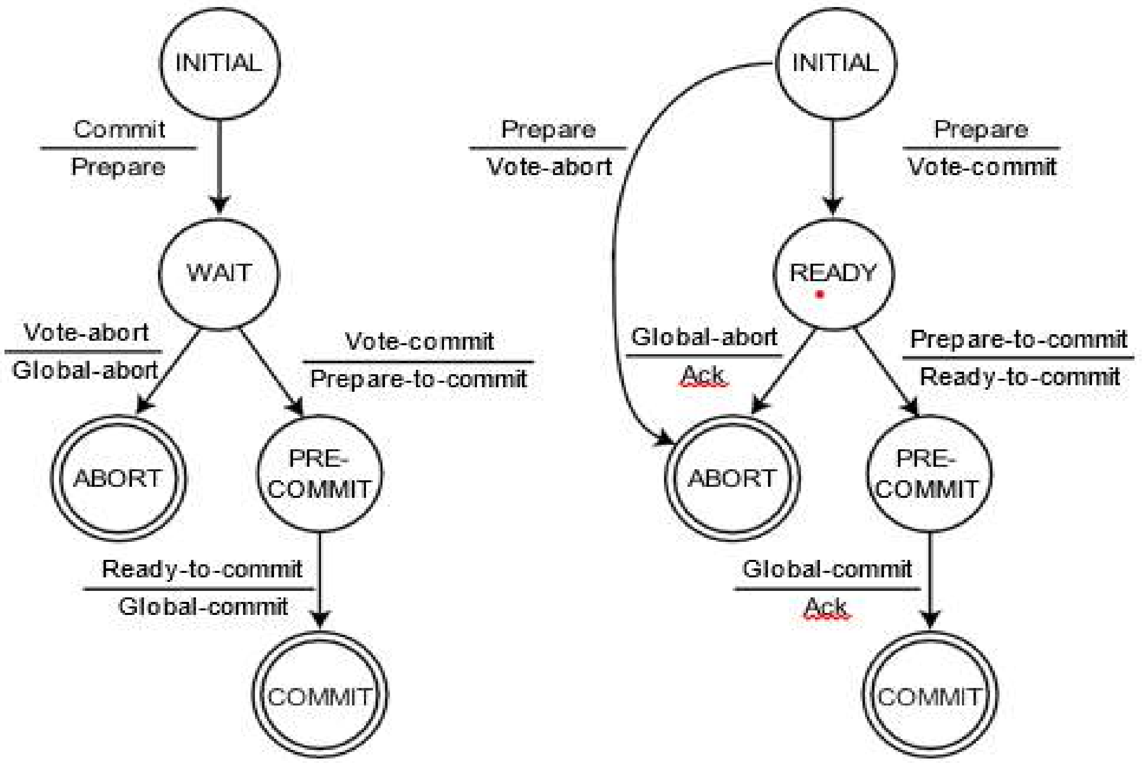


Fig. State Transitions in 3PC Protocol

- It is possible to design different 3PC algorithms depending on the communication topology.
- It is also straightforward to design a distributed 3PC protocol.

Termination Protocol

- As we did in discussing the termination protocols for handling timeouts in the 2PC protocol, let us investigate timeouts at each state of the 3PC protocol.

***Coordinator Timeouts*.**

- In 3PC, there are four states in which the coordinator can time out: WAIT, PRECOM- MIT, COMMIT, or ABORT.

1. *Timeout in the WAIT state*. This is identical to the coordinator timeout in the WAIT state for the 2PC protocol. The coordinator unilaterally decides to abort the transaction. It therefore writes an abort record in the log and sends a "global-abort" message to all the participants that have voted to commit the transaction.
2. *Timeout in the PRECOMMIT state*. The coordinator does not know if the non-responding participants have already moved to the PRECOMMIT state. However, it knows that they are at least in the READY state, which means that they must have voted to commit the transaction. The coordinator can therefore move all participants to PRECOMMIT state by sending a "prepare-to-commit" message go ahead and globally commit the transaction by writing a commit record in the log and sending a "global-commit" message to all the operational participants.
3. *Timeout in the COMMIT (or ABORT) state*. The coordinator does not know whether the participants have actually performed the commit (abort) command. However, they are at least in the PRECOMMIT (READY) state (since the protocol is synchronous within one state transition) and can follow the termination protocol as described in case 2 or case 3 below. Thus the coordinator does not need to take any special action.

Participant Timeouts.

- A participant can time out in three states: INITIAL, READY, and PRECOMMIT. Let us examine all of these cases.

1. *Timeout in the INITIAL state*. This can be handled identically to the termina- tion protocol of 2PC.
2. *Timeout in the READY state*. In this state the participant has voted to commit the transaction but does not know the global decision of the coordinator. Since communication with the coordinator is lost, the termination protocol proceeds by electing a new coordinator, as discussed earlier. The new coordinator then terminates the transaction according to a termination protocol that we discuss below.
3. *Timeout in the PRECOMMIT state*. In this state the participant has received the "prepare-to-commit" message and is awaiting the final "global-commit" message from the coordinator. This case is handled identically to case 2 above.

Recovery Protocols

- There are some minor differences between the recovery protocols of 3PC and those of 2PC.
- We only indicate those differences.

1. *The coordinator fails while in the WAIT state*. This is the case we discussed at length in the earlier section on termination protocols. The participants have already terminated the transaction. Therefore, upon recovery, the coordinator has to ask around to determine the fate of the transaction.
2. *The coordinator fails while in the PRECOMMIT state*. Again, the termination protocol has guided the operational participants toward termination. Since it is now possible to move from the PRECOMMIT state to the ABORT state during this process, the coordinator has to ask around to determine the fate of the transaction.
3. *A participant fails while in the PRECOMMIT state*. It has to ask around to determine how the other participants have terminated the transaction.

- One property of the 3PC protocol becomes obvious from this discussion.
- When using the 3PC protocol, we are able to terminate transactions without blocking.
- However, we pay the price that fewer cases of independent recovery are possible.
- This also results in more messages being exchanged during recovery.

Network Partitioning

- Network partitions are due to communication line failures and may cause the loss of messages, depending on the implementation of the communication subnet.
- A partitioning is called a ***simple partitioning*** if the network is divided into only two components; otherwise, it is called ***multiple partitioning***.
- No **non-blocking** atomic commitment protocol exists that is resilient to multiple partitioning.
- Possible to design non-blocking atomic commit protocol resilient to simple partitioning.
- Concern is with termination of transactions that were active at time of partitioning.
- Strategies: permit all partitions to continue normal operations, accept possible inconsistency of database; or block operation in some partitions to maintain consistency.
- This decision problem is the premise of a classification of partition handling strategies.
- We can classify the strategies as *pessimistic* or *optimistic*
- **Pessimistic strategies** emphasize the consistency of the database, and would therefore not permit transactions to execute in a partition if there is no guarantee that the consistency of the database can be maintained.
- **Optimistic approaches**, on the other hand, emphasize the availability of the database even if this would cause inconsistencies.
- All the known termination protocols that deal with network partitioning in the case of non-replicated databases are pessimistic.
- Since the pessimistic approaches emphasize the maintenance of database consistency, the fundamental issue that we need to address is which of the partitions can continue normal operations.
- We consider two approaches.
    1. Centralized Protocols
    2. Voting-based Protocols

1.Centralized Protocols

- Centralized termination protocols are based on the centralized concurrency control algorithms.
- It makes sense to permit the operation of the partition that contains the central site, since it manages the lock tables.
- Primary site techniques are centralized with respect to each data item.
- In this case, more than one partition may be operational for different queries.
- For any given query, only the partition that contains the primary site of the data items that are in the write set of that transaction can execute that transaction.
- Both of these are simple approaches that would work well, but they are dependent on the concurrency control mechanism employed by the distributed database manager.
- Furthermore, they expect each site to be able to differentiate network partitioning from site failures properly.
- This is necessary since the participants in the execution of the commit protocol react differently to the different types of failures.

2.Voting-based Protocols

Voting as a technique for managing concurrent data accesses has been proposed by a number of researchers.
The fundamental idea is that a transaction is executed if a majority of the sites vote to execute it.
The idea of majority voting has been generalized to voting with *quorums*.
Quo- rum-based voting can be used as a replica control method, as well as a commit method to ensure transaction atomicity in the presence of network partitioning.
In the case of non-replicated databases, this involves the integration of the voting principle with commit protocols.
Every site in the system is assigned a vote V_i. Let us assume that the total number of votes in the system is V, and the abort and commit quorums are V_a and V_c, respectively.
Then the following rules must be obeyed in the implementation of the commit protocol:

1. $V_a + V_c > V$, where $0 \leq V_a, V_c \leq V$.
2. Before a transaction commits, it must obtain a commit quorum V_c.
3. Before a transaction aborts, it must obtain an abort quorum V_a.

- The first rule ensures that a transaction cannot be committed and aborted at the same time. The next two rules indicate the votes that a transaction has to obtain before it can terminate one way or the other.
- The integration of these rules into the 3PC protocol requires a minor modification of the third phase.
- For the coordinator to move from the PRECOMMIT state to the COMMIT state, and to send the “global-commit” command, it is necessary for it to have obtained a commit quorum from the participants. This would satisfy rule 2.
- Note that we do not need to implement rule 3 explicitly. This is due to the fact that a transaction which is in the WAIT or READY state is willing to abort the transaction. Therefore, an abort quorum already exists.
- Let us now consider the termination of transactions in the presence of failures.

- When a network partitioning occurs, the sites in each partition elect a new coordinator, similar to the 3PC termination protocol in the case of site failures.
- There is a fundamental difference, however. It is not possible to make the transition from the WAIT or READY state to the ABORT state in one state transition, for a number of reasons.
- Depending on the responses, it terminates the transaction as follows:

1. If at least one participant is in the COMMIT state, the coordinator decides to commit the transaction and sends a "global-commit" message to all the participants.
2. If at least one participant is in the ABORT state, the coordinator decides to abort the transaction and sends a "global-abort" message to all the participants.
3. If a commit quorum is reached by the votes of participants in the PRECOM- MIT state, the coordinator decides to commit the transaction and sends a "global-commit" message to all the participants.
4. If an abort quorum is reached by the votes of participants in the PREABORT state, the coordinator decides to abort the transaction and sends a "global- abort" message to all the participants.
5. If case 3 does not hold but the sum of the votes of the participants in the PRECOMMIT and READY states are enough to form a commit quorum, the coordinator moves the participants to the PRECOMMIT state by sending a "prepare-to-commit" message. The coordinator then waits for case 3 to hold.
6. Similarly, if case 4 does not hold but the sum of the votes of the participants in the PREABORT and READY states are enough to form an abort quorum, the coordinator moves the participants to the PREABORT state by sending a "prepare-to-abort" message. The coordinator then waits for case 4 to hold.

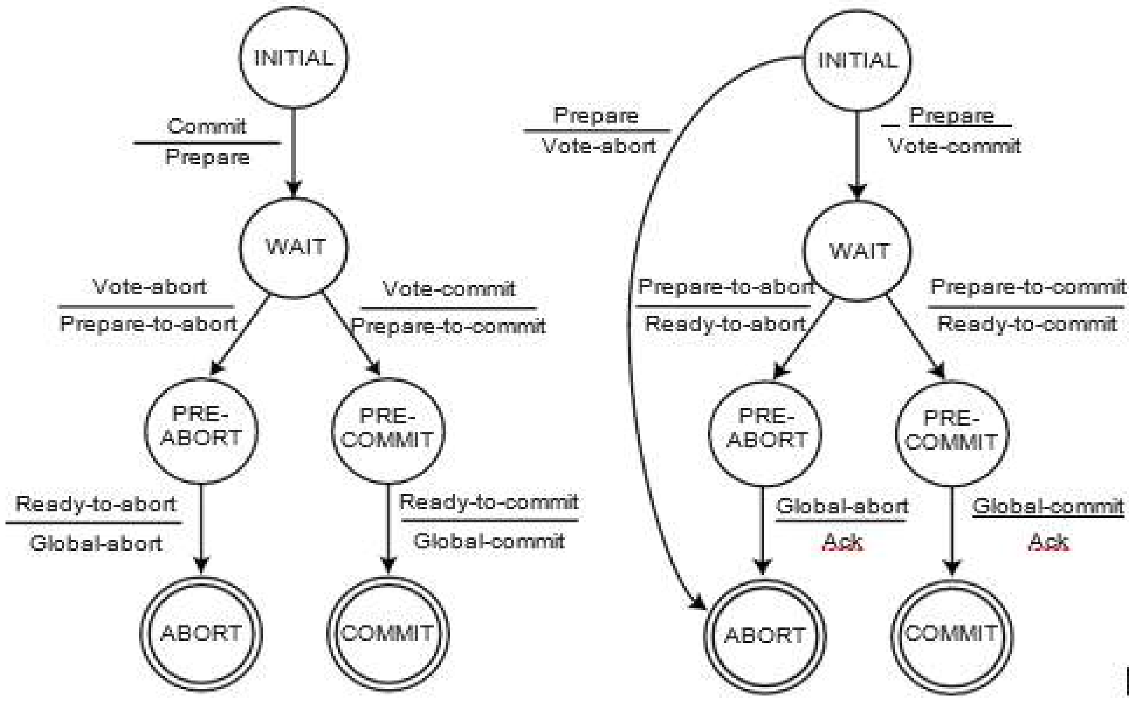


Fig. State Transitions in Quorum 3PC Protocol

- Two points are important about this quorum-based commit algorithm.
- First, it is blocking; the coordinator in a partition may not be able to form either an abort or a commit quorum if messages get lost or multiple partitioning occur. This is hardly surprising given the theoretical bounds that we discussed previously.
- The second point is that the algorithm is general enough to handle site failures as well as network partitioning. Therefore, this modified version of 3PC can provide more resiliency to failures.
- The recovery protocol that can be used in conjunction with the above-discussed termination protocol is very simple.
- When two or more partitions merge, the sites that are part of the new larger partition simply execute the termination protocol.
- That is, a coordinator is elected to collect votes from all the participants and try to terminate the transaction.

Parallel Database Systems

- Many data-intensive applications require support for very large databases (e.g., hundreds of terabytes or petabytes).
- Examples of such applications are e-commerce, data warehousing, and data mining.
- Very large databases are typically accessed through high numbers of concurrent transactions (e.g., performing on-line orders on an electronic store) or complex queries (e.g., decision-support queries). The first kind of access is representative of **On-Line Transaction Processing (OLTP)** applications while the second is representative of **On-Line Analytical Processing (OLAP)** applications.
- Supporting very large databases efficiently for either OLTP or OLAP can be addressed by combining parallel computing and distributed database management.
- Data distribution can be exploited to increase performance (through parallelism) and availability (through replication).
- This principle can be used to implement ***parallel database systems***, i.e., database systems on parallel computers.
- Parallel database systems can exploit the parallelism in data management in order to deliver high-performance and high-availability database servers.
- Thus, they can support very large databases with very high loads.

Parallel Database System Architectures

- We present and compare the main architectures: shared-memory, shared-disk, shared-nothing and hybrid architectures.

Objectives

- A parallel database system can be loosely defined as a DBMS implemented on a parallel computer.
- The objectives of parallel database systems are covered by those of distributed DBMS (performance, availability, extensibility).

- A parallel database system should provide the following advantages.
    1. **High-performance**
    2. **High-availability.**
    3. **Extensibility**

High-performance.

- This can be obtained through several complementary solutions: database-oriented operating system support, parallel data management, query optimization, and load balancing
- Parallelism can increase throughput, using inter-query parallelism, and decrease transaction response times, using intra-query parallelism.
- Therefore, it is crucial to optimize and parallelize queries in order to minimize the overhead of parallelism, e.g., by constraining the degree of parallelism for the query.
- *Load balancing* is the ability of the system to divide a given workload equally among all processors.
- Depending on the parallel system architecture, it can be achieved statically by appropriate physical database design or dynamically at run-time.

High-availability.

- Because a parallel database system consists of many redundant components, it can well increase data availability and fault-tolerance.
- A fault-tolerance technique that enables automatic redirection of transactions from a failed node to another node that stores a copy of the data.
- This provides uninterrupted service to users. However, it is essential that a node failure does not crate load imbalance, e.g., by doubling the load on the available copy.
- Solutions to this problem require partitioning copies in such a way that they can also be accessed in parallel.

Extensibility.

- In a parallel system, accommodating increasing database sizes or increasing performance demands (e.g., throughput) should be easier. Extensibility is the ability to expand the system smoothly by adding processing and storage power to the system.
- Ideally, the parallel database system should demonstrate two extensibility advantages : *linear speedup* and *linear scaleup.*
- Linear speedup refers to a linear increase in performance for a constant database size while the number of nodes (i.e., processing and storage power) are increased linearly.
- Linear scaleup refers to a sustained performance for a linear increase in both database size and number of nodes. Furthermore, extending the system should require minimal reorganization of the existing database.

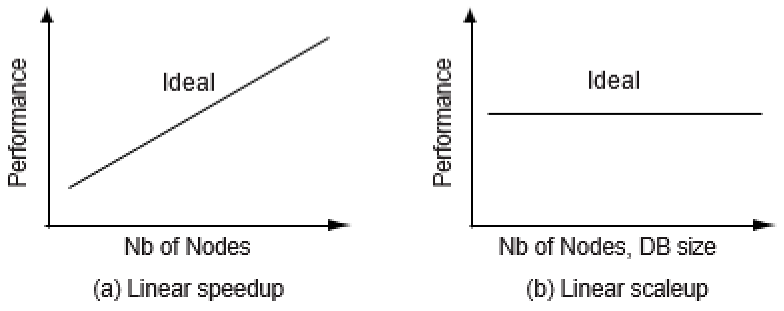


Fig. Extensibility Metrics

Functional Architecture

- Assuming a client/server architecture, the functions supported by a parallel database system can be divided into three subsystems much like in a typical DBMS.
- The differences, though, have to do with implementation of these functions, which must now deal with parallelism, data partitioning and replication, and distributed transactions.
- Depending on the architecture, a processor node can support all (or a subset) of these subsystems.

1. **Session Manager.**
   - It plays the role of a transaction monitor, providing support for client interactions with the server. In particular, it performs the connections and disconnections between the client processes and the two other subsystems.
   - Therefore, it initiates and closes user sessions (which may contain multiple transactions). In case of OLTP sessions, the session manager is able to trigger the execution of pre-loaded transaction code within data manager modules.
2. **Transaction Manager.**
   - It receives client transactions related to query compilation and execution. It can access the database directory that holds all meta-information about data and programs.
   - The directory itself should be managed as a database in the server. Depending on the transaction, it activates the various compilation phases, triggers query execution, and returns the results as well as error codes to the client application.
   - Because it supervises transaction execution and commit, it may trigger the recovery procedure in case of transaction failure. To speed up query execution, it may optimize and parallelize the query at compile-time.
3. **Data Manager.**
   - It provides all the low-level functions needed to run compiled queries in parallel, i.e., database operator execution, parallel transaction support, cache management, etc.
   - If the transaction manager is able to compile dataflow control, then synchronization and communication among data man- ager modules is possible.
   - Otherwise, transaction control and synchronization must be done by a transaction manager module.

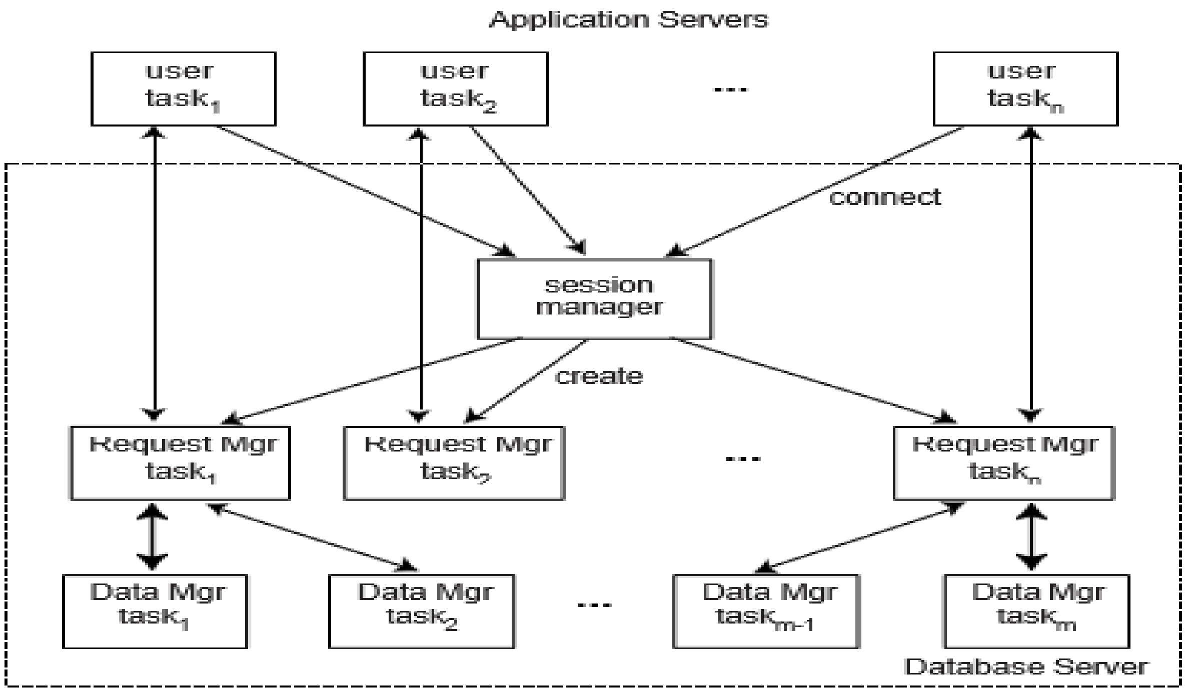


Fig. General Architecture of a Parallel Database System

Parallel DBMS Architectures

- There are three basic parallel computer architectures depending on how main memory or disk is shared:
  1. *shared-memory*
  2. *shared-disk*
  3. *shared-nothing*
- Hybrid architectures such as NUMA or *cluster* try to combine the benefits of the basic architectures.
- We focus on the four main hardware elements: interconnect, processors (P), main memory (M) and disks.
- We ignore other elements such as processor cache and I/O bus.

Shared-Memory

- In the shared-memory approach, any processor has access to any memory module or disk unit through a fast interconnect (e.g., a high-speed bus or a cross-bar switch).
- All the processors are under the control of a single operating system.
- All shared-memory parallel database products today can exploit inter-query parallelism to provide high transaction throughput and intra-query parallelism to reduce response time of decision-support queries.
- Current mainframe designs and symmetric multiprocessors (SMP) follow this approach.
- Since meta-information (directory) and control information (e.g., lock tables) can be shared by all processors.

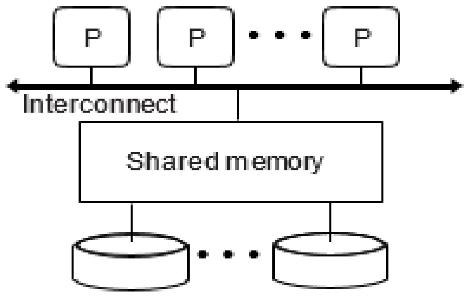


Fig. Shared-Memory Architecture

Shared-memory has two strong advantages: simplicity and load balancing.
Shared-memory has three problems: high cost, limited extensibility and low availability.

Shared-Disk

- In the shared-disk approach, any processor has access to any disk unit through the interconnect but exclusive (non-shared) access to its main memory.
- Each processor-memory node is under the control of its own copy of the operating system.
- Then, each processor can access database pages on the shared disk and cache them into its own memory.
- The first parallel DBMS that used shared-disk is Oracle with an efficient implementation of a distributed lock manager for cache consistency.
- Other major DBMS vendors such as IBM, Microsoft and Sybase provide shared-disk implementations.

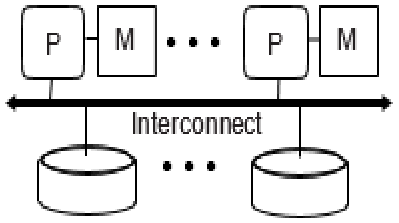


Fig. Shared-Disk Architecture

Shared-disk has a number of advantages: lower cost, high extensibility, load balancing, availability, and easy migration from centralized systems.
Shared-disk suffers from two problems: higher complexity and potential performance

Shared-Nothing

- In the shared-nothing approach each processor has exclusive access to its main memory and disk unit(s).
- Similar to shared-disk, each processor- memory-disk node is under the control of its own copy of the operating system.
- Then, each node can be viewed as a local site (with its own database and software) in a distributed database system.
- As opposed to SMP, this architecture is often called Massively Parallel Processor (MPP).

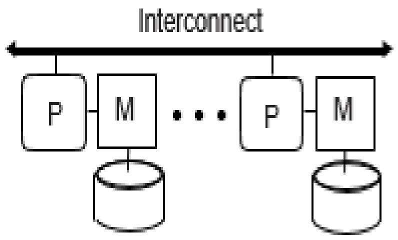


Fig. Shared-Nothing Architecture

Advantages : lower cost, high extensibility, and high availability.
Problems : Shared-nothing is much more complex to manage than either shared-memory or shared-disk.

Hybrid Architectures

- Hybrid architectures try to obtain the advantages of different architectures: typically the efficiency and simplicity of shared-memory and the extensibility and cost of either shared disk or shared nothing.
- There are two popular hybrid architectures: NUMA and cluster.

NUMA

- The objective of NUMA is to provide a shared-memory programming model and all its benefits, in a scalable architecture with distributed memory.
- The term NUMA reflects the fact that an access to the (virtually) shared memory may have a different cost depending on whether the physical memory is local or remote to the processor.
- The most successful class of NUMA multiprocessors is Cache Coherent NUMA (CC-NUMA).

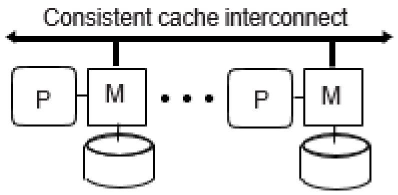


Fig. Cache coherent NUMA (CC-NUMA)

Cluster

- A cluster is a set of independent server nodes interconnected to share resources and form a single system. The shared resources, called *clustered* resources, can be hardware such as disk or software such as data management services.
- A cluster architecture has important advantages. It combines the flexibility and performance of shared-memory at each node with the extensibility and availability of shared-nothing or shared-disk.

Parallel Data Placement

- Data placement in a parallel database system exhibits similarities with data fragmentation in distributed databases (see Chapter 3). An obvious similarity is that fragmentation can be used to increase parallelism.
- we use the terms *partitioning* and *partition* instead of horizontal fragmentation and horizontal fragment, respectively
- To contrast with the alternative strategy, which consists of *clustering* a relation at a single node. The term *declustering* is sometimes used to mean partitioning.
- Vertical fragmentation can also be used to increase parallelism and load balancing much as in distributed databases.
- Data placement must be done so as to maximize system performance, which can be measured by combining the total amount of work done by the system and the response time of individual queries.
- An alternative solution to data placement is *full partitioning*, whereby each relation is horizontally fragmented across *all* the nodes in the system.
- There are three basic strategies for data partitioning:
    1. Round-Robin Partitioning
    2. Hash Partitioning
    3. Range Partitioning

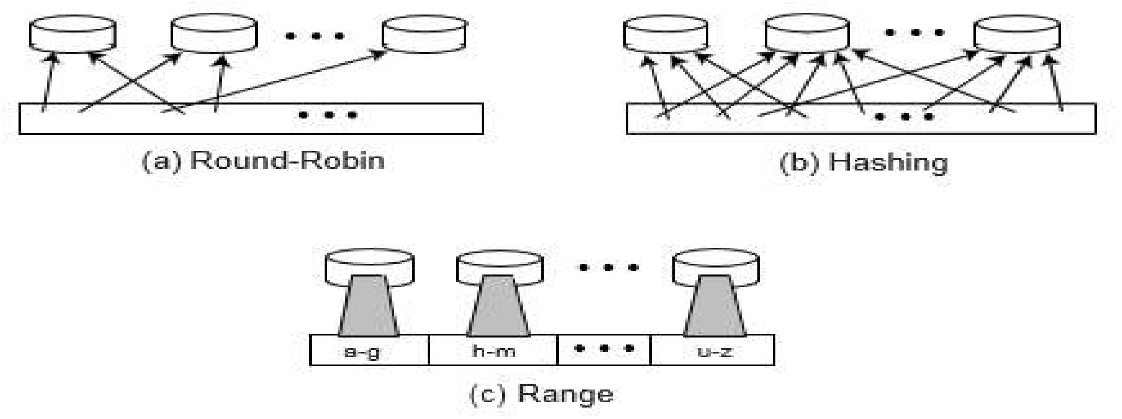


Fig. Different Partitioning Schemes

1. ***Round-robin partitioning***
    - *Round-robin partitioning* is the simplest strategy, it ensures uniform data distribution.
    - This strategy enables the sequential access to a relation to be done in parallel.
    - The direct access to individual tuples, based on a predicate, requires accessing the entire relation.
2. ***Hash partitioning***
    - *Hash partitioning* applies a hash function to some attribute that yields the partition number.
    - This strategy allows exact-match queries on the selection attribute to be processed by exactly one node and all other queries to be processed by all the nodes in parallel.

3. ***Range partitioning***
   - *Range partitioning* distributes tuples based on the value intervals (ranges) of some attribute.
   - In addition to supporting exact-match queries (as in hashing), it is well-suited for range queries.
   - For instance, a query with a predicate "*A* between A_1 and A_2" may be processed by the only node(s) containing tuples whose *A* value is in range $[A_1, A_2]$. However, range partitioning can result in high variation in partition size.

Parallel Query Processing

- The objective of parallel query processing is to transform queries into execution plans that can be efficiently executed in parallel.
- This is achieved by exploiting parallel data placement and the various forms of parallelism offered by high-level queries.
- The various forms of parallelism.
  - o Query parallelism
  - o Parallel algorithms for data processing
  - o Parallel query optimization.

1.Query Parallelism

- Parallel query execution can exploit two forms of parallelism: inter- and intra-query *parallelism.*
- **Inter-query parallelism** which gives each machine different queries to work on so that the system can achieve a high throughput and complete as many queries as possible.
- **Intra-query parallelism** attempts to make one query run as fast as possible by spreading the work over multiple computers. We can further divide Intra-query parallelism into two classes: intra-operator and inter-operator.

Intra-operator Parallelism

- Intra-operator parallelism is based on the decomposition of one operator in a set of independent sub-operators, called *operator instances*.
- This decomposition is done using static and/or dynamic partitioning of relations.

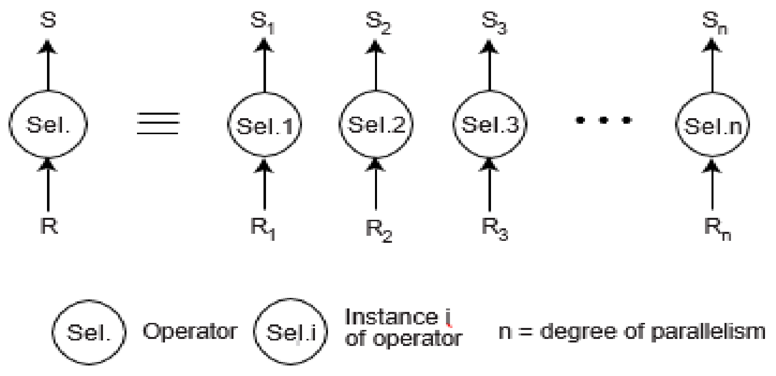


Fig. Intra-operator Parallelism

Inter-operator Parallelism

- Two forms of inter-operator parallelism can be exploited.
- *pipeline parallelism*, several operators with a producer-consumer link are executed in parallel. The select operator will be executed in parallel with the join operator.
- The advantage of such execution is that the intermediate result is not materialized, thus saving memory and disk accesses.
- *Independent parallelism* is achieved when there is no dependency between the operators that are executed in parallel. For instance, the two select operators can be executed in parallel.
- This form of parallelism is very attractive because there is no interference between the processors.

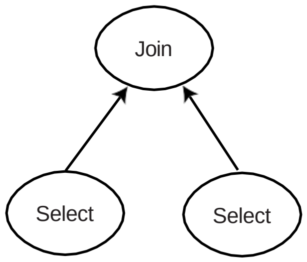


Fig. Inter-operator Parallelism

2.Parallel Algorithms for Data Processing

- Parallel data processing should exploit intra-operator parallelism. i.e, parallel algorithms for database operators on the select and join operators.
- The processing of the select operator in a partitioned data placement context is identical to that in a fragmented distributed database.
- The parallel processing of join is significantly more involved than that of select. The distributed join algorithms designed for high-speed networks can be applied successfully in a partitioned database context.
- There are three basic parallel join algorithms for partitioned databases:
    1. The Parallel Nested Loop (PNL) Algorithm
    2. The Parallel Associative Join (PAJ) Algorithm
    3. The Parallel Hash Join (PHJ) Algorithm.

- We describe each using a pseudo-concurrent programming language with three main constructs: **parallel- do, send**, and **receive**. **Parallel-do** specifies that the following block of actions is executed in parallel.
- For example,
  for i from 1 to *n* in parallel do action *A*
- Indicates that action *A* is to be executed by *n* nodes in parallel.
- **Send** and **receive** are the basic communication primitives to transfer data between nodes.

- To summarize, the parallel nested loop algorithm can be viewed as replacing the operator $R\ 0\ S$ by $\text{U}_{i=1}^{n}$ $(R\ 0\ S_i)$.

```
Algorithm 14.1: PNL Algorithm
  Input: $R_1, R_2, \ldots, R_m$: fragments of relation $R$;
  $S_1, S_2, \ldots, S_n$: fragments of relation $S$;
  $JP$: join predicate
  Output: $T_1, T_2, \ldots, T_n$: result fragments
  begin
      for i from 1 to m in parallel do          {send R entirely to each S-node}
          send $R_i$ to each node containing a fragment of $S$
      for j from 1 to n in parallel do          {perform the join at each S-node}
          $R \leftarrow \bigcup_{i=1}^{m} R_i$;      {receive $R_i$ from R-nodes; R is fully replicated at S-nodes}
          $T_j \leftarrow R\ 0_{JP}\ S_j$
  end
```

***Example* : The application of the parallel nested loop algorithm with *m* = *n* = 2.**

3.Parallel Query Optimization

- Parallel query optimization exhibits similarities with distributed query processing.
- It focuses much more on taking advantage of both intra-operator parallelism and inter-operator parallelism.
- A parallel query optimizer can be seen as three components: a search space, a cost model, and a search strategy.

Search Space

- Execution plans are abstracted by means of operator trees, which define the order in which the operators are executed.

Cost Model

- Cost model is responsible for estimating the cost of a given execution plan.
- It consists of two parts: architecture-dependent and architecture- independent.

Search Strategy

- The search strategy does not need to be different from either centralized or distributed query optimization.
- The search space tends to be much larger because there are more parameters that impact parallel execution plans, in particular, pipeline and store annotations.
- Thus, randomized search strategies generally outperform deterministic strategies in parallel query optimization.

Load Balancing

- Good load balancing is crucial for the performance of a parallel system.
- The response time of a set of parallel operators is that of the longest one.
- Minimizing the time of the longest one is important for minimizing response time.
- Balancing the load of different transactions and queries among different nodes is also essential to maximize throughput.
- Solutions to these problems can be obtained at the intra- and inter-operator levels

Intra-Operator Load Balancing

- Good intra-operator load balancing depends on the degree of parallelism and the allocation of processors for the operator.
- The skew problem makes it hard for a parallel query optimizer to make this decision statically (at compile-time) as it would require a very accurate and detailed cost model.
- Therefore, the main solutions rely on adaptive or specialized

Inter-Operator Load Balancing

- In order to obtain good load balancing at the inter-operator level, it is necessary to choose, for each operator, how many and which processors to assign for its execution.
- This should be done taking into account pipeline parallelism, which requires inter- operator communication.

Database Clusters

- Database clustering is the process of connecting more than one single database instance or server to your system.
- In most common database clusters, multiple database instances are usually managed by a single database server called the master.

Database Cluster Architecture

Database cluster is in two types of architectures

1. shared-disk architecture
2. shared-nothing architecture

Shared-Nothing Architecture

- To build a shared-nothing database architecture each database server must be independent of all other nodes.
- Meaning that each node has its own database server to store and access data from. In this type of architecture, no single database server is master.
- Meaning that there is no one central database node that monitors and controls the access of data in the system.

- Note that a shared-nothing architecture offers great horizontal scalability as no resources are being shared between either nodes or database servers.

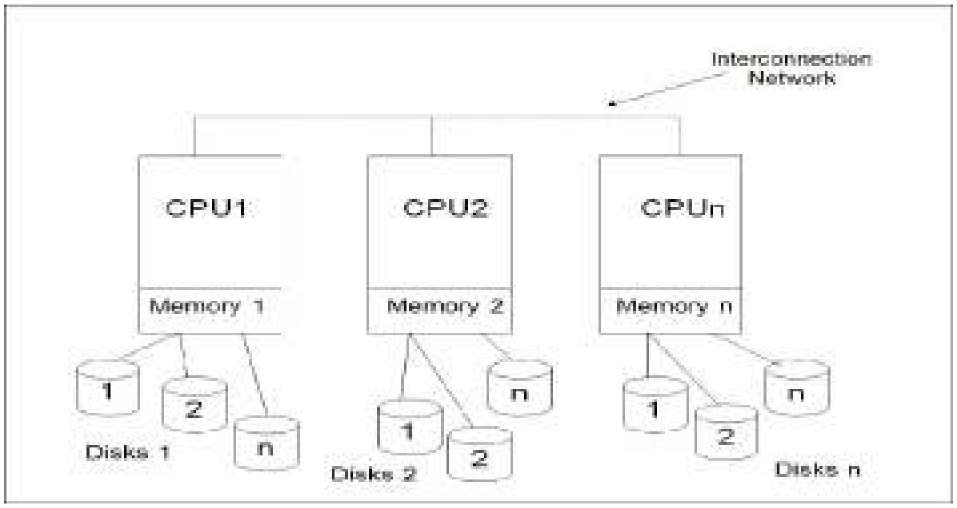


Fig : Shared-Nothing Architecture

Shared-Disk Architecture

- In this architecture, all nodes(CPU) share access to all the database servers available, subsequently having access to all the system's data.
- Unlike the shared-nothing architecture, the interconnection network layer is between the CPU and the database servers allowing for multiple database servers' access.
- It is worth noting that a shared disk cluster does not offer much scalability when compared to the shared-nothing architecture, as if all nodes share access to the same data a controlling node is required to monitor the data flow in the system.
- The issue is that after exceeding a certain number of slave nodes, the master node would be unable to monitor and control all the slave nodes efficiently.

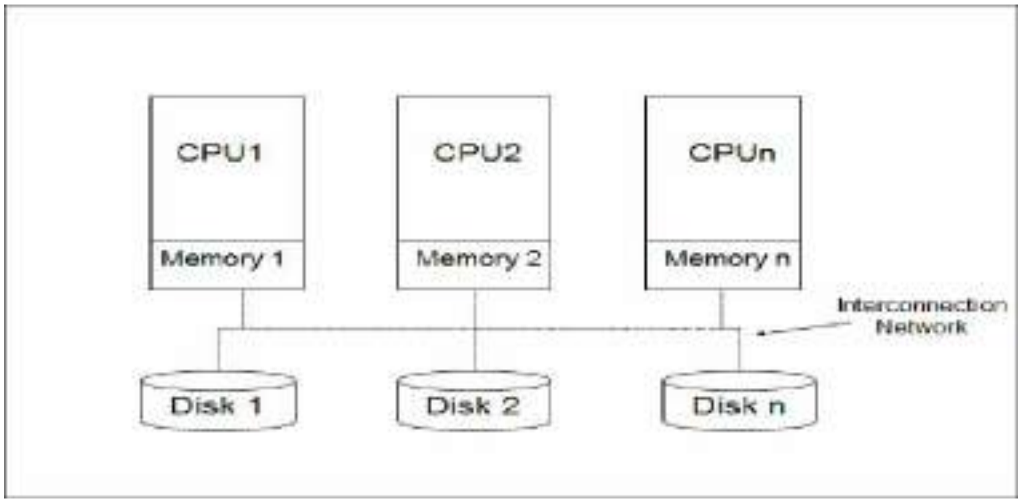


Fig : Shared Disk Architecture

DISTRIBUTED DATABASES (R22CSE3146)

III B.Tech I Semester (Information Technology)

UNIT – V

Distributed Object Database Management Systems:
Fundamental object concepts and models, object distributed design, architectural issues, object management, distributed object storage, object query processing.
Object Oriented Data Model:
Inheritance, object identity, persistent programming languages, persistence of objects, comparison OODBMS and ORDBMS.

Distributed Object Database Management System

Fundamental Object Concepts and Object Models

- An object DBMS is a system that uses an "object" as the fundamental modeling, in which information is represented in the form of objects.

Object

- All object DBMSs are built around the fundamental concept of an *object*.
- An object represents a real entity in the system that is being modeled.
- It is represented as a tuple $\langle \text{OID}, \text{ state}, \text{ interface} \rangle$
- In which OID is the objectidentifier, the corresponding state is some representation of the current state of the object, and the interface defines the behavior of the object.
- Object identifier is an invariant property of an object which permanently distinguishes it logically and physically from all other objects, regardless of its state.
- The *state* of an object is commonly defined as either an atomic value or a constructed value (e.g., tuple or set).

A *value* is defined as follows:

1. An element of D is a value, called an *atomic value*.
2. $[a_1 : v_1,\ldots,a_n : v_n]$, in which a_i is an element of A and v_i is either a value oran element of I, is called a *tuple value*. [ ] is known as the tuple constructor.
3. $\{v_1,\ldots,v_n\}$, in which v_i is either a value or an element of I, is called a *setvalue*. { } is known as the set constructor.

- These models consider object identifiers as values (similar to pointers in programming languages).
- Set and tuple are data constructors that we consider essential for database applications. Other constructors, such as list or array, could also be added to increase the modeling power.

Example *:* Consider the following objects:

(i_1, 231)
(i_2, S70)
(i_3, {i_6,i_{11})
(i_4, {1,3,5})
(i_5, [LF: i_7, RF: i_8, LR: i_9, RR: i_10])

- Objects i_1 and i_2 are atomic objects and i_3 and i_4 are constructed objects. i_3 is the OID of an object whose state consists of a set.
- The same is true of i_4. The difference between the two is that the state of i_4 consists of a set of values, while that of i_3 consists of a set of OIDs. Thus, object i_3 references other objects.
- By considering object identifiers (e.g., i_6) as values in the object model, arbitrarily complex objects may be constructed.
- Object i_5 has a tuple valued state consisting of four attributes (or instance variables), the values of each being another object.

Example : Consider the following objects:

(i_1, Volvo)
(i_2, [name: John, mycar: i_1])
(i_3, [name: Mary, mycar: i_1])

- Here, John and Mary share the object denoted by i_1 (they both own Volvo cars). Changing the value of object i_1 from "Volvo" to "Chevrolet" is automatically seen by both objects i_2 and i_3.

Types and Classes

- The term "class" refers to the specific object model construct and the term "type" to refer to a domain of objects (e.g., integer, string).
- A class is a template for a group of objects, thus defining a common type for these objects that conform to the template.
- We don't make a distinction between primitive system objects (i.e., values), structural (tuple or set) objects, and user-defined objects.
- A class describes the type of data by providing a domain of data with the same structure, as well as methods applicable to elements of that domain.
- The abstraction capability of classes, commonly referred to as *encapsulation*, hides the implementation details of the methods, which can be written in a general-purpose programming language.
- Some (possibly proper) subset of its class structure and methods make up the publicly visible interface of objects that belong to that class.

Example : In this example, demonstrates the power of object models.

- We will model a car that consists of various parts (engine, bumpers, tires) and will store other information such as make, model, serial number, etc.
- The type definition of Car can be as follows using this abstract syntax:

```
type Car
  attributes
        engine : Engine
        bumpers : {Bumper}
        tires : [lf: Tire, rf: Tire, lr: Tire, rr: Tire]
        make : Manufacturer
        model : String
        year : Date
        serial_no : String
        capacity : Integer
  methods
        age: Real
        replaceTire(place, tire)
```

- The class definition specifies that Carhas eight attributes and two method. Fourof the attributes (model, year, serial no, capacity) are value-based, while the others(engine, bumpers, tires and make) are object-based (i.e., have other objects as theirvalues).
- Attribute bumpers is set valued (i.e., uses the set constructor), and attribute tires is tuple-valued where the left front (lf), right front (rf), left rear (lr) and right rear (rr) tires are individually identified.
- Incidentally, we follow a notation where the attributes are lower case and types are capitalized. Thus, engine is an attribute and Engineis a type in the system.

Composition (Aggregation)

- Composition is one of the most powerful features of object models.
- It allows sharingof objects, commonly referred to as *referential sharing*, since objects "refer" to each other by their OIDs as values of object-based attributes.

Example : Assume that c_1 is one instance of Cartype. If the following is true:

(i_2, [name: John, mycar: c_1])
(i_3, [name: Mary, mycar: c_1])

- Then this indicates that John and Mary own the same car.
- The composite object relationship between types can be represented by a *composition (aggregation) graph* (or *composition (aggregation) hierarchy* in the case of complex objects).
- There is an edge from instance variable I of type T_1 to type T_2 if the domain of I is T_2.

Subclassing and Inheritance

- Object systems provide extensibility by allowing user-defined classes to be defined and managed by the system.
- This is accomplished in two ways: by the definition of classes using type constructors or by the definition of classes based on existing classes through the process of *subclassing*.
- Subclassing is based on the *specialization* relationship among classes (or types that they define).
- A class A is a *specialization* of another class B if its interface is a superset of B's interface. Thus, a specialized class is more defined (or more specified) than the class from which it is specialized.
- A class may be a specialization of a number of classes; it is explicitly specified as a *subclass* of a subset of them.
- Some object models require that a class is specified as a subclass of only one class, in which case the model supports *single subclassing*; others allow *multiple subclassing*, where a class may be specified as a subclass of more than one class.
- Subclassing and specialization indicate an **is-a** relationship between classes (types).
- In the above example, A **is-a** B, resulting in *substitutability*: an instance of a subclass (A) can be substituted in place of an instance of any of its *superclasses* (B) in any expression.

Example : Consider the Cartype we defined earlier.

- A car can be modeledas a special type of Vehicle. Thus, it is possible to define Car as a subtype of Vehiclewhose other subtypes may be Motorcycle, Truck, and Bus.
- In thiscase, Vehiclewould define the common properties of all of these:

```
type Vehicle as Objectattributes

engine : Engine
make : Manufacturer
model : String
year : Date
serial_no : String
methods age: Real
```

- Vehicleis defined as a subclass of Objectthat we assume is the root of theclass lattice with common methods such as Put or Store.
- Vehicle is defined with five attributes and one method that takes the date of manufacture and today's date (both of which are of system-defined type Date) and returns a real value.
- Obviously, Vehicle is a generalization of Car that we defined in Example
- Car can nowbe defined as follows:

```
type Car as Vehicle

attributes

  bumpers : {Bumper}
  tires : [LF: Tire, RF: Tire, LR: Tire, RR: Tire]
  capacity : Integer
```

- Even though Car is defined with only two attributes, its interface is the same as the definition given in Example.
- This is because Car **is-a** Vehicle, and therefore inherits the attributes and methods of Vehicle.

Features of Object DBMS :

1. Support for complex objects
2. Support for object identity
3. Data persistence must be provided.
4. OODMS must support concurrent users.
5. OODMS must capable of recovery from hardware and software failure.
6. Support for dynamic binding.

Drawbacks of Object Database:

1. Object DB's are not as popular as RDBMS. It is difficult to find object DB developers.
2. Not many programming languages support object databases.
3. Object DB do not have a standard query language.
4. Object DB are difficult to learn for Non-programmers.

Object Distribution Design

- The two important aspects of distribution design are **fragmentation and allocation.**
- Distribution design in the object world brings new complexities due to the encapsulation of methods together with object state.
- An object is defined by its state and its methods. We can fragment the state, the method definitions, and the method implementation.
- Furthermore, the objects in a class extent can also be fragmented and placed at different sites. Each of these raise interesting problems and issues.

Fragmentation

- There are three fundamental types of fragmentation:horizontal, vertical, and hybrid.
- In addition to these two fundamental cases, Horizontal Partitioning , Vertical Partitioning , and Path Partitioning have been defined.

Horizontal Class Partitioning

- Partitioning of a class arising from the fragmentation of its subclasses. This occurs when a more specialized class is fragmented, so the results of this fragmentation should be reflected in the more general case.
- The fragmentation of a complex attribute may affect the fragmentation of its containing class.
- Fragmentation of a class based on a method invocation sequence from one class to another may need to be reflected in the design.

- Class C for partitioning, we create classes $C_1,\ldots,C_n$, each of which takes the instances of C that satisfy the particular partitioning predicate.
- If these predicates are mutually exclusive, then classes $C_1,\ldots,C_n$ are disjoint.

Example : Consider the definition of the Engine class

Class Engine as Object

attributes

no_cylinder : Integer

capacity : Real

horsepower: Integer

In this simple definition of Engine, all the attributes are simple.

Consider the partitioning predicates

p_1: horsepower ≤ 150
p_2: horsepower > 150

- In this case, Engine can be partitioned into two classes, Engine1 and Engine2, which inherit all of their properties from the Engine class, which is redefined as an abstract class (i.e,. a class that cannot have any objects in its shallow extent).
- The objects of Engineclass are distributed to the Engine1and Engine2classes based on the value of their horsepower attribute value.

Vertical Class Partitioning

- Vertical fragmentation is considerably more complicated. Given a class C, fragmenting it vertically into $C_1,\ldots,C_m$ produces a number of classes, each of which contains some of the attributes and some of the methods.
- Thus, each of the fragments is less defined than the original class. Issues that must be addressed include the subtyping between the original class' superclasses and subclasses and the fragment classes, the relationship of the fragment classes among themselves, and the location of the methods.
- If all the methods are simple, then methods can be partitioned easily. However, when this is not the case, the location of these methods becomes a problem.
- Adaptations of the affinity-based relational vertical fragmentation approaches have been developed for object databases.
- However, the break-up of encapsulation during vertical fragmentation has created significant doubts as to the suitability of vertical fragmentation in object DBMSs.

Path Partitioning

- The composition graph presents a representation for composite objects. For many applications, it is necessary to access the complete composite object.
- Path partitioning is a concept describing the clustering of all the objects forming a composite object into a partition.
- A path partition consists of grouping the objects of all the domain classes that correspond to all the instance variables in the subtree rooted at the composite object.
- A path partition can be represented as a hierarchy of nodes forming a structural index.
- Each node of the index points to the objects of the domain class of the component object.
- The index thus contains the references to all the component.

Class Partitioning Algorithms

- The main issue in class partitioning is to improve the performance of user queries and applications by reducing the irrelevant data access.
- Thus, class partitioning is a logical database design technique that restructures the object database schema based on the application semantics.
- It should be noted that class partitioning is morecomplicated than relation fragmentation, and is also NP-complete.
- The algorithms for class partitioning are based on affinity-based and cost-driven approaches.

Affinity-based Approach

- Affinity among attributes is used to vertically fragment relations. Similarly, affinity among instance variables and methods, and affinity among multiple methods can be used for horizontal and vertical class partitioning.
- Horizontal and vertical class partitioning algorithms have been developed that are based on classifying instance variables and methods as being either simple or complex.
- A complex instance variable is an object-based instance variable and is part of the class composition hierarchy.
- An alternative is a method- induced partitioning scheme, which applies the method semantics and appropriatelygenerates fragments that match the methods data requirements.

Cost-Driven Approach

- Though the affinity-based approach provides "intuitively" appealing partitioning schemes, it has been shown that these partitioning schemes do not always result in the greatest reduction of disk accesses required to process a set of applications.
- Therefore, a cost model for the number of disk accesses for processing both queries and methods on an object oriented database has been developed.
- Further, an heuristic "hill-climbing" approach that uses both the affinity approach (for initial solution) and the cost-driven approach (for further refinement) has been proposed.

- This work also develops structural join index hierarchies for complex object retrieval, and studies its effectiveness against pointer traversal and other approaches, such as join index hierarchies, multi-index and access support relations (see next section).

Allocation

- The data allocation problem for object databases involves allocation of both methods and classes.
- The method allocation problem is tightly coupled to the class allocation problem because of encapsulation.
- Therefore, allocation of classes will imply allocation of methods to their corresponding home classes. But since applications on object-oriented databases invoke methods, the allocation of methods affects the performance of applications. However, allocation of methods that need to access multiple classes at different sites is a problem that has been not yet been tackled.
- Four alternatives can be identified :

1. **Local behavior – local object.** This is the most straightforward case and is included to form the baseline case. The behavior, the object to which it isto be applied, and the arguments are all co-located. Therefore, no special mechanism is needed to handle this case.
2. **Local behavior – remote object.** This is one of the cases in which the behavior and the object to which it is applied are located at different sites. There are two ways of dealing with this case. One alternative is to move the remote object to the site where the behavior is located. The second is to ship the behavior implementation to the site where the object is located. This is possible if the receiver site can run the code.
3. **Remote behavior – local object.** This case is the reverse of case (2).
4. **Remote function – remote argument.** This case is the reverse of case (1).

Replication

- Replication adds a new dimension to the design problem. Individual objects, classes of objects, or collections of objects (or all) can be units of replication.
- Undoubtedly, the decision is at least partially object-model dependent.
- Whether or not type specifications are located at each site can also be considered a replication problem.

Architectural Issues

- The preferred architectural model for object DBMSs has been client/server.
- The design issues related to these systems are somewhat more complicated due to the characteristics of object models. The major concerns are listed below.

1. Since data and procedures are encapsulated as objects, the unit of communication between the clients and the server is an issue. The unit can be a page, an object, or a group of objects.

2. Closely related to the above issue is the design decision regarding the functions provided by the clients and the server. This is especially important since objects are not simply passive data, and it is necessary to consider the sites where object methods are executed.
3. In relational client/server systems, clients simply pass queries to the server, which executes them and returns the result tables to the client. This is referred to as *function shipping*. In object client/server DBMSs, this may not be the best approach, as the navigation of composite/complex object structures by the application program may dictate that data be moved to the clients (called *data shipping systems*). Since data are shared by many clients, the management of client cache buffers for data consistency becomes a serious concern. Client cache buffer management is closely related to concurrency control, since data that are cached to clients may be shared by multiple clients, and this has to be controlled. Most commercial object DBMSs use locking for concurrency control, so a fundamental architectural issue is the placement of locks, and whether or not the locks are cached to clients.
4. Since objects may be composite or complex, there may be possibilities for prefetching component objects when an object is requested. Relational client/server systems do not usually prefetch data from the server, but this may be a valid alternative in the case of object DBMSs.

- These considerations require revisiting some of the issues common to all DBMSs, along with several new ones. We will consider these issues in three sections: those directly related to architectural design (architectural alternatives, buffer management, and cache consistency).

Alternative Client/Server Architectures

- Two main types of client/server architectures have been proposed: object servers and page servers.
- The distinction is partly based on the granularity of data that are shipped between the clients and the servers, and partly on the functionality provided to the clients and servers.

Object Server :

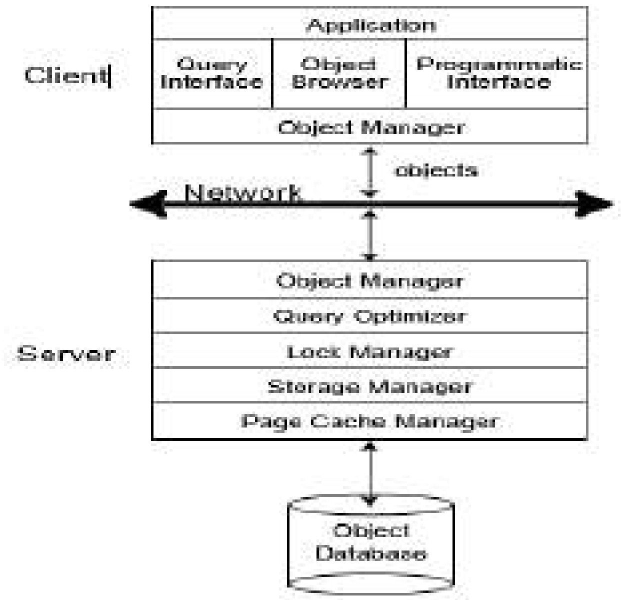


Fig. Object Server Architecture

- The first alternative is that clients request “objects” from the server, which retrieves them from the database and returns them to the requesting client. These systems are called *object servers*.
- In object servers, the server undertakes most of the DBMS services, with the client providing basically an execution environment for the applications, as well as some level of object management functionality.
- The object management layer is duplicated at both the client and the server in order to allow both to perform object functions. Object manager serves a number of functions.
- Object manager also deals with the implementation of the object identifier (logical, physical, or virtual) and the deletion of objects (either explicit deletion or garbage collection).
- The object managers at the client and the server implement an object cache (in additionto the page cache at the server). Objects are cached at the client to improve system performance by localizing accesses.
- The optimization of user queries and the synchronizationof user transactions are all performed in the server, with the client receiving the resulting objects.

Page Server :

- An alternative organization is a *page server* client/server architecture, in which the unit of transfer between the servers and the clients is a physical unit of data, suchas a page or segment, rather than an object.
- Page server architectures split the object processing services between the clients and the servers.
- Page servers simplify the DBMS code, since both the server and the client maintainpage caches, and the representation of an object is the same all the way from the disk to the user interface.
- Thus, updates to the objects occur only in client caches and these updates are reflected on disk when the page is flushed from the client to the server.
- The server performs a limited set of functions andcan therefore serve a large number of clients.
- Page servers can also exploit operating systems and even hardware functionality to deal with certain problems, such as pointer swizzling.

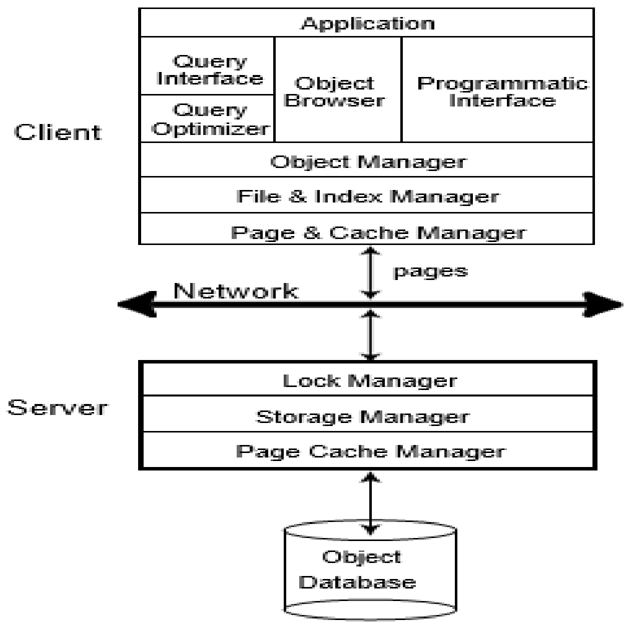


Fig. Page Server Architecture

Client Buffer Management

- The clients can manage either a page buffer, an object buffer, or a dual (i.e., page/object) buffer.
- If clients have a page buffer, then entire pages are read or written from the server every time a page fault occurs or a page is flushed. Object buffers can read/write individual objects and allow the applications object-by-object access.
- A page buffer does not encounter this problem, but if the data clustering on the disk does not match the application data access pattern, then the pages contain a great deal of unaccessed objects that use up valuable client buffer space.
- In these situations, buffer utilization of a page buffer will be lower than the buffer utilization of an object buffer.

Server Buffer Management

- The server buffer management issues in object client/server systems are not much different than their relational counterparts, since the servers usually manage a page buffer.
- We nevertheless discuss the issues here briefly in the interest of completeness.
- The pages from the page buffer are, in turn, sent to the clients to satisfy their data requests.
- A grouped object-server constructs its object groups by copying the necessary objects from the relevant server buffer pages, and sends the object group to the clients.
- In addition to the page level buffer, the servers can also maintain a modified object buffer (MOB).
- A MOB stores objects that have been updated and returned by the clients. These updated objects have to be installed onto their corresponding data pages, which may require installation reads as described earlier.
- Finally, the modified page has to be written back to the disk.
- A MOB allows the server to amortize its disk I/O costs by batching the installation read and installation write operations.
- In a client/server system, since the clients typically absorb most of the data requests (i.e., the system has a high cache hit rate), the server buffer usually behaves more as a staging buffer than a cache.

Cache Consistency

- Cache consistency is a problem in any data shipping system that moves data to the clients. However, the problems arise in unique ways in object DBMSs.
- The DBMS cache consistency algorithms can be classified as avoidance-based or detection-based.
- ***Avoidance-based algorithms*** prevent the access to stale cache data[5] by ensuring that clients cannot update an object if it is being read by other clients. So they ensure that stale data never exists in client caches.
- ***Detection-based algorithms*** allow access of stale cache data, because clients can update objects that are being read by other clients.
- However, the detection-based algorithms perform a validation step at commit time to satisfy data consistency requirements.

- We discuss each of the alternatives in the design space and comment on their performance characteristics.
- **Avoidance-based synchronous:** Callback-Read Locking (CBL) is the most common synchronous avoidance-based cache consistency algorithm [Franklin and Carey, 1994]. In this algorithm, the clients retain read locks across transactions, but they relinquish write locks at the end of the transaction. The clients send lock requests to the server and they block until the server responds. If the client requests a write lock on a page that is cached at other clients, the server issues callback messages requesting that the remote clients relinquish their read locks on the page. Callback-Read ensures a low abort rate and generally outperforms deferred avoidance-based, synchronous detection-based, and asynchronous detection-based algorithms.
- **Avoidance-based asynchronous:** Asynchronous avoidance-based cache conistency algorithms (AACC) do not have the message blocking overhead present in synchronous algorithms. Clients send lock escalation messages to the server and continue application processing. Normally, optimistic approaches such as this face high abort rates, which is reduced in avoidance-based algorithms by immediate server actions to invalidate stale cache objects at remote clients as soon as the system becomes aware of the update. Thus, asynchronous algorithms experience lower deadlock abort rates than deferred avoidance-based algorithms, which are discussed next.
- **Avoidance-based deferred:** Optimistic Two-Phase Locking (O2PL) family of cache consistency are deferred avoidance-based algorithms [Franklin and Carey, 1994]. In these algorithms, the clients batch their lock escalation requests and send them to the server at commit time. The server blocks the updating client if other clients are reading the updated objects. As the data contention level increases, O2PL algorithms are susceptible to higher deadlock abort rates than CBL algorithms.
- **Detection-based synchronous:** Caching Two-Phase Locking (C2PL) is a synchronous detection-based cache consistency algorithm [Carey et al., 1991]. In this algorithm, clients contact the server whenever they access a page in their cache to ensure that the page is not stale or being written to by other clients. C2PL's performance is generally worse than CBL and O2PL algorithms, since it does not cache read locks across transactions.
- **Detection-based asynchronous:** No-Wait Locking (NWL) with Notification is an asynchronous detection-based algorithm. In this algorithm, the clients send lock escalation requests to the server, but optimistically assume that their requests will be successful. After a client transaction commits, the server propagates the updated pages to all the other clients that have also cached the affected pages. It has been shown that CBL outperforms the NWL algorithm.
- **Detection-based deferred:** Adaptive Optimistic Concurrency Control (AOCC) is a deferred detection-based algorithm. It has been shown that AOCC can outperform callback locking algorithms even while encountering a higher abort rate if the client transaction state (data and logs) completely fits into the client cache, and all application processing is strictly performed at the clients (purely data-shipping architecture). Since AOCC uses deferred messages, its messaging overhead is less than CBL. Furthermore, in a purely data-shipping client/server environment, the impact of an aborting client on the performance of other clients is quite minimal. These factors contribute to AOCC's superior performance.

Object Management

- Object management includes tasks such as object identifier management, pointer swizzling, object migration, deletion of objects, method execution, and some storage management tasks at the server.

Object Identifier Management

- Object identifiers (OIDs) are system-generated and used to uniquely identify every object (transient or persistent, system-created or user-created) in the system.
- The implementation of persistent object identifier has two common solutions, based on either physical or logical identifiers, with their respective advantages and shortcomings.
- The physical identifier (POID) approach equates the OID with the physical address of the corresponding object. The address can be a disk page address and an offset from the base address in the page.
- The advantage is that the object can be obtained directly from the OID.
- The drawback is that all parent objects and indexes must be updated whenever an object is moved to a different page.
- The logical identifier (LOID) approach consists of allocating a system-wide unique OID (i.e., a surrogate) per object.
- LOIDs can be generated either by using a system-wide unique counter (called pure LOID) or by concatenating a server identifier with a counter at each server (called pseudo-LOID).

Pointer Swizzling

- In object systems, one can navigate from one object to another using *path expressions* that involve attributes with object-based values. For example, if object c is of type Car, then c.engine.manufacturer.name is a path expression. These are basically pointers.
- Usually on disk, object identifiers are used to represent these pointers.
- However, in memory, it is desirable to use in-memory pointers for navigating from one object to another.
- The process of converting a disk version of the pointer to an in-memory version of a pointer is known as “pointer-swizzling”.
- Hardware-based and software-based schemes are two types of pointer-swizzling mechanisms.
- In hardware-based schemes, the operating system’s page-fault mechanism is used; when a page is brought into memory, all the pointers in it are swizzled.

Object Migration

- One aspect of distributed systems is that objects move, from time to time, between sites.
- This raises a number of issues. First is the unit of migration.
- It is possible to move the object’s state without moving its methods.
- Even if individual objects are units of migration, their relocation may move them away from their type specifications and one has to decide whether types are duplicated at every site where instances reside or the types are accessed remotely when behaviors or methods are applied to objects.

- Three alternatives can be considered for the migration of classes (types):

  1. The source code is moved and recompiled at the destination,
  2. The compiled version of a class is migrated just like any other object, or
  3. The source code of the class definition is moved, but not its compiled opera-tions, for which a lazy migration strategy is used.

- Another issue is that the movements of the objects must be tracked so that they can be found in their new locations.
- A common way of tracking objects is to leave *surrogates*, or *proxy objects*.
- These are place-holder objects left at the previous site of the object, pointing to its new location. Accesses to the proxy objects are directed transparently by the system to the objects themselves at the new sites.
- The migration of objects can be accomplished based on their current state.
- Objects can be in one of four states:

  1. **Ready :** Ready objects are not currently invoked, or have not received a mes-sage, but are ready to be invoked to receive a message.
  2. **Active :** Active objects are currently involved in an activity in response to an invocation or a message.
  3. **Waiting :** Waiting objects have invoked (or have sent a message to) another object and are waiting for a response.
  4. **Suspended :** Suspended objects are temporarily unavailable for invocation.

- Objects in active or waiting state are not allowed to migrate, since the activity they are currently involved in would be broken.
- The migration involves two steps:

  1. shipping the object from the source to the destination, and
  2. creating a proxy at the source, replacing the original object.

- Two related issues must also be addressed here.
  1. One relates to the maintenance of the system directory.
  2. The second issue is that, in a highly dynamic environment where objects move frequently, the surrogate or proxy chains may become quite long.

Distributed Object Storage

- Among the many issues related to object storage, two are particularly relevant in a distributed system:
  1. Object Clustering
  2. Distributed Garbage Collection.

- For clustering data on disk such that the I/O cost of retrieving them is reduced.
- Garbage collection is a problem that arises in object databases due to reference-based sharing. Indeed, in many object DBMSs, the only way to delete an object is to delete all references to it.

Object Clustering

- An object model is essentially conceptual, and should provide high physical data independence to increase programmer productivity. The mapping of this conceptual model to a physical storage is a classical database problem.
- Object clustering refers to the grouping of objects in physical containers (i.e., disk extents) according to common properties, such as the same value of an attribute or sub-objects of the same object.
- Thus, fast access to clustered objects can be obtained.
- Object clustering is difficult for two reasons.
- First, it is not orthogonal to object identifier implementation (i.e, LOID vs. POID). LOIDs incur more overhead (an indirection table), but enable vertical partitioning of classes. POIDs yield more efficient direct object access, but require each object to contain all inherited attributes.
- Second, the clustering of complex objects along the composition relationship is more involved because of object sharing (objects with multiple parents).
- There are three basic storage models for object clustering:

1. The ***decomposition storage model*** (DSM) partitions each object class into binary relations (OID, attribute) and therefore relies on logical OID. The advantage of DSM is simplicity.
2. The ***normalized storage model*** (NSM) stores each class as a separate relation. It can be used with logical or physical OID. However, only logical OID allows the vertical partitioning of objects along the inheritance relationship.
3. The ***direct storage model*** (DSM) enables multi-class clustering of complex objects based on the composition relationship. This model generalizes the techniques of hierarchical and network databases, and works best with physical OID.

Distributed Garbage Collection

- The generality of distributed object-based systems calls for automatic distributed garbage collection.
- The basic garbage collection algorithms can be categorized as ***Reference counting*** **or Tracing-based.**

Reference counting

- In a reference counting system, each object has an associated count of the references to it.
- Each time a program creates an additional reference that points to an object, the object’s count is incremented.
- When an existing reference to an object is destroyed, the corresponding count is decremented.

- The memory occupied by an object can be reclaimed when the object’s count drops to zero and become unreachable (at which time, the object is garbage).
- In reference counting, a problem can arise where two objects only refer to each other but not referred to by anyone else; in this case, the two objects are basically unreachable (except from each other) but their reference count has not dropped to zero.

Tracing-based

- *Tracing-based* collectors are divided into *mark and sweep* and *copy-based* algorithms.
- *Mark and sweep* collectors are two-phase algorithms.
- The first phase, called the “mark” phase, starts from the root and marks every reachable object (for example, by setting a bit associated to each object). This mark is also called a “color”, and the collector is said to color the objects it reaches.
- The mark bit can be embedded in the objects themselves or in *color maps* that record, for every memory page, the colors of the objects stored in that page.
- Once all live objects are marked, the memory is examined and unmarked objects are reclaimed. This is the “sweep” phase.

Object Query Processing

- There has been significant amount of work on object query processing and optimization, these have primarily focused on centralized systems.
- Almost all object query processors and optimizers that have been proposed to date use techniques developed for relational systems.
- It is possible to claim that distributed object query processing and optimization techniques require the extension of centralized object.
- Although most object query processing proposals are based on their relational counterparts, there are a number of issues that make query processing and optimization more difficult in object DBMSs
- Relational query languages operate on very simple type systems consisting of a single type.
- Relational query optimization depends on knowledge of the physical storage of data (access paths) that is readily available to the query optimizer.
- Objects can (and usually do) have complex structures whereby the state of an object references another object.

Object Oriented Data Model

- In Object Oriented Data Model, data and their relationships are contained in a single structure which is referred as object in this data model.
- In this, real world problems are represented as objects with different attributes.
- All objects have multiple relationships between them.
- It is combination of Object Oriented programming and Relational Database Model
- It is clear from the following figure :

Object Oriented Data Model = Combination of Object Oriented Programming + Relational database model

Components of Object Oriented Data Model :

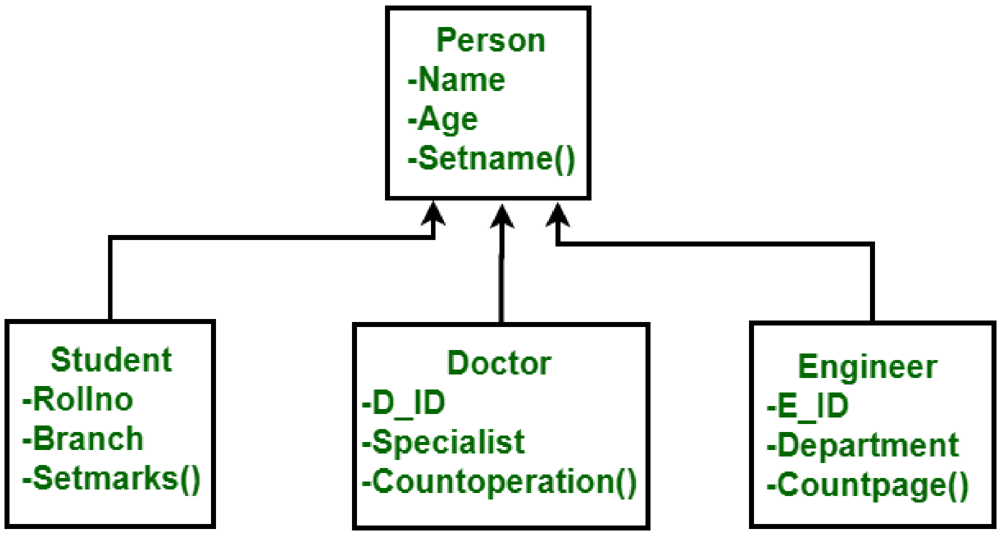


Fig : Basic Object Oriented Data Model / Hierarchy of classes

Objects

- An object is an abstraction of a real world entity or we can say it is an instance of class.
- Objects encapsulates data and code into a single unit which provide data abstraction by hiding the implementation details from the user.
- For example: Instances of student, doctor, engineer in above figure.

Attribute

- An attribute describes the properties of object.
- For example: Object is STUDENT and its attribute are Roll no, Branch, Setmarks() in the Student class.

Methods

- Method represents the behavior of an object. Basically, it represents the real-world action.
- For example: Finding a STUDENT marks in above figure as Setmarks().

Class

- A class is a collection of similar objects with shared structure i.e. attributes and behavior i.e. methods.
- An object is an instance of class.
- For example: Person, Student, Doctor, Engineer in above figure.

```
class student
```

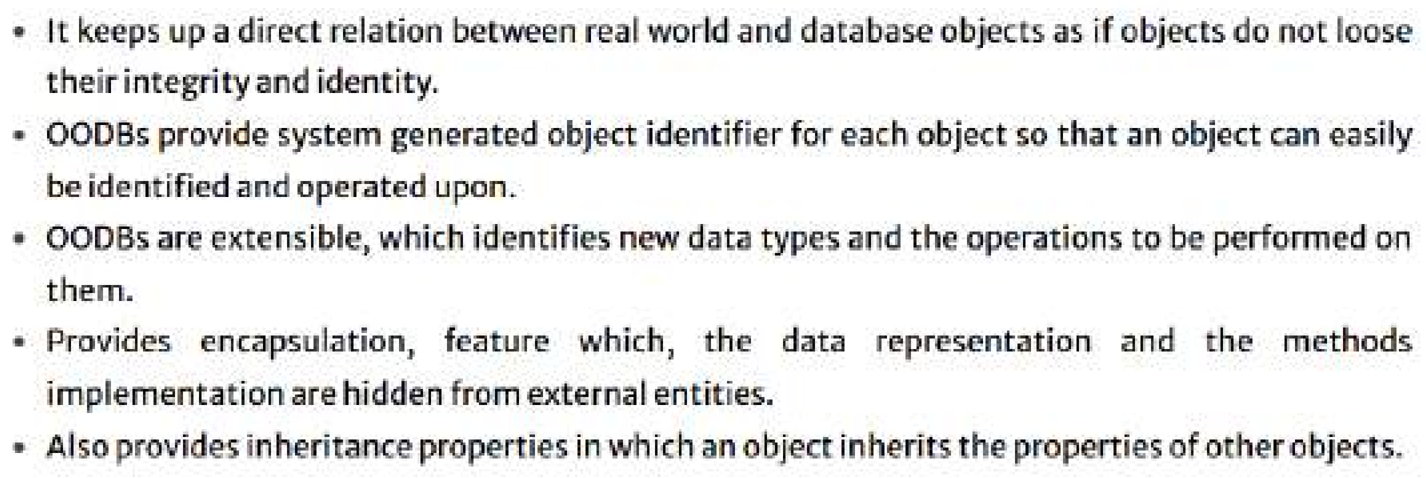
```
{
    char Name[20];
    int roll_no;
    --
    --
    public:
    void search();
    void update();
}
```

In this example, students refers to class and S1, S2 are the objects of class which can be created in main function.

Inheritance

- By using inheritance, new class can inherit the attributes and methods of the old class i.e. base class.
- For example: as classes Student, Doctor and Engineer are inherited from the base class Person.

Characteristics of an Object Oriented Data Model



- It keeps up a direct relation between real world and database objects as if objects do not loose their integrity and identity.
- OODBs provide system generated object identifier for each object so that an object can easily be identified and operated upon.
- OODBs are extensible, which identifies new data types and the operations to be performed on them.
- Provides encapsulation, feature which, the data representation and the methods implementation are hidden from external entities.
- Also provides inheritance properties in which an object inherits the properties of other objects.

Advantages of Object Oriented Data Model :

- Codes can be reused due to inheritance.
- Easily understandable.
- Cost of maintenance can reduced due to reusability of attributes and functions because of inheritance.

Disadvantages of Object Oriented Data Model :

- It is not properly developed so not accepted by users easily.

Inheritance

- Inheritance creates a hierarchical relationship between related classes while making parts of code reusable.
- Defining new types inherits all the existing class fields and methods plus further extends them.
- The existing class is the parent class, while the child class extends the parent.
- Inheritance can be of various type
    - Substitution inheritance
    - Inclusion inheritance
    - Constraint inheritance
    - Specialization inheritance

Substitution inheritance

- ✓ Is based on behavior and not on values
- ✓ t' can be substituted anywhere for t.

Inclusion inheritance

- ✓ Denotes classification
- ✓ It is based on structure and not operations

Constraint inheritance

- ✓ Is a sub-case of inclusion
- ✓ Satisfies certain constraint (retired person sub type of person with constraint)

Specialization inheritance

- ✓ With more specific information (student object is a person with some extra attributes as course no)

Object Identity

- The basic idea behind object identity is that objects should be identified by system-generated object identifiers (OIDs) rather than by the values of their properties.
- This is in sharp contrast to the relational model, where for instance tuples are identified by the value of their primary key.
- Object identity provides a unique identifier for each object in an object database.
- This identifier distinguishes one object from another, even if they share the same attributes. Object identity ensures data integrity and consistency.
- You can reference objects using their unique identifiers.
- This capability simplifies data retrieval and manipulation. Object identity plays a crucial role in managing complex data structures.
- This has two implications:

- ✓ One is object sharing and
- ✓ The other one is object updates.

Object sharing:

- It is an identity-based model, two objects can share a component.
- Thus, the pictorial representation of a complex object is a graph, while it is limited to be a tree in a system without object identity.
- Consider the following example: a Person has a name, an age and a set of children. Assume Peter and Susan both have a 15-year-old child named John. In real life, two situations may arise: Susan and Peter are parent of the same child or there are two children involved. In a system without identity, Peter is represented by:

  (peter, 40, {(john, 15, {})})

  and Susan is represented by:

  (susan, 41, {(john, 15, {})}).

- Thus, there is no way of expressing whether Peter and Susan are the parents of the same child. In an identity-based model, these two structures can share the common part (john, 15, {}) or not, thus capturing either situations.

Object updates:

- The update item (or object) operation updates an existing item or adds a new item to the table if it does not already exist.
- Assume that Peter and Susan are indeed parents of a child named John. In this case, all updates to Susan's son will be applied to the object John and, consequently, also to Peter's son.
- In a value-based system, both sub-objects must be updated separately. Object identity is also a powerful data manipulation primitive that can be the basis of set, tuple and recursive complex object manipulation.
- Supporting object identity implies offering operations such as object assignment, object copy (both deep and shallow copy) and tests for object identity and object equality (both deep and shallow equality).

Persistent Programming Language

- Programming languages that natively and seamlessly allow objects to continue existing after the program has been closed down are called **persistent programming languages**.
- A persistent programming language is a programming language extended with constructs to handle persistent data.
- In a persistent programming language:
  - The query language is fully integrated with the host language and both share the same type system.

- Any format changes required between the host language and the database are carried out transparently.

- Using Embedded SQL, a programmer is responsible for writing explicit code to fetch data into memory or store data back to the database.

- In a persistent programming language, a programmer can manipulate persistent data without having to write such code explicitly.

- The drawbacks of persistent programming languages include:

  - While they are powerful, it is easy to make programming errors that damage the database.
  - It is harder to do automatic high-level optimization.
  - They do not support declarative querying well.

Persistence of Object

- **Persistence** denotes a process or an object that continues to exist even after its parent process or object ceases, or the system that runs it is turned off.
- Types of persistent : There are two types of persistence:
  1. Object Persistence
  2. Process persistence.
- **Object persistence** refers to an object that is not deleted until a need emerges to remove it from the memory. Some database models provide mechanisms for storing persistent data in the form of objects.
- **Process persistence**, processes are not killed or shut down by other processes and exist until the user kills them. For example, all of the core processes of a computer system are persistent in enabling the proper functioning of the system. Persistent processes are stored in non-volatile memory. They do not need special databases like persistent objects.

Comparison of OODBMS and ORDBMS

OODBMS	ORDBMS
It stands for Object Oriented Database Management System.	It stands for Object Relational Database Management System.
In the Object Oriented Database, the data is stored in the form of objects.	In Relational Database, data is stored in the form of tables, which contains rows and columns.

OODBMS	**ORDBMS**
In OODBMS, relationships are represented by references via the object identifier (OID).	In ORDBMS, connections between two relationships are represented by foreign key attributes.
It handles larger and complex data than RDBMS.	It handles comparatively simpler data.
In OODBMS, the data management language is typically incorporated into a programming languages such as C++, C#.	In relational database systems there are data manipulation language such as SQL.
Stores data entries are described as objects.	Stores data in entries is described as tables.
Object-oriented databases, like Object Oriented Programming, represent data in the form of objects and classes.	An object-relational database is one that is based on both the relational and object-oriented database models.
OODBMSs support ODL/OQL.	ORDBMS adds object-oriented functionalities to SQL.
Every object-oriented system has a different set of constraints that it can accommodate.	Keys, entity integrity, and referential integrity are constraints of an object-oriented database.
The efficiency of query processing is low.	Processing of queries is quite effective.