DISTRIBUTED DATABASES (23IT512)

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.

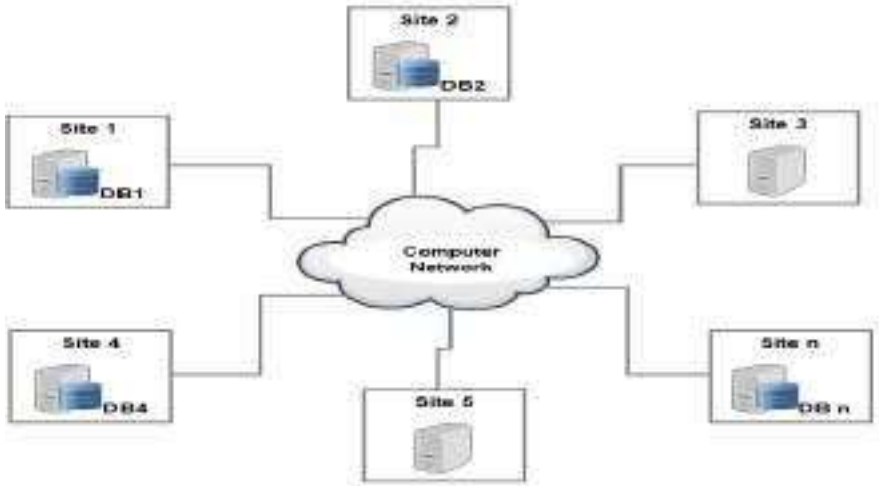


Fig: Distributed Database System

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.

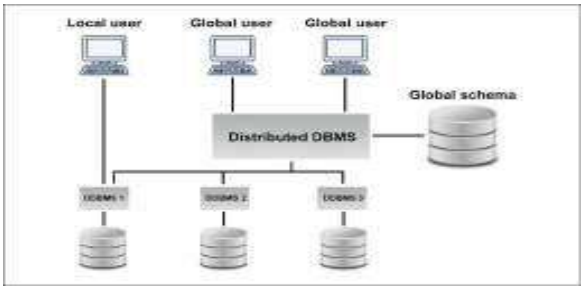
Distributed Database Types

There are two types of distributed databases:

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

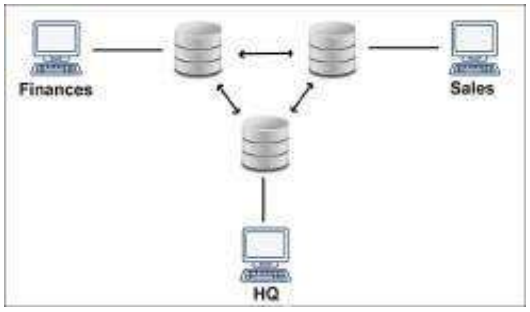
Homogeneous: A homogenous distributed database is a network of identical databases stored on multiple sites. The sites have the same operating system, DDBMS, and data structure, making them easily manageable.

- 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:



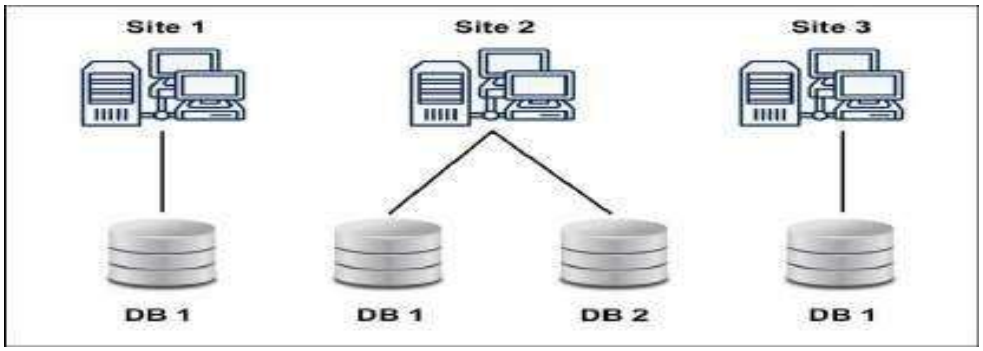
Distributed Database Storage

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

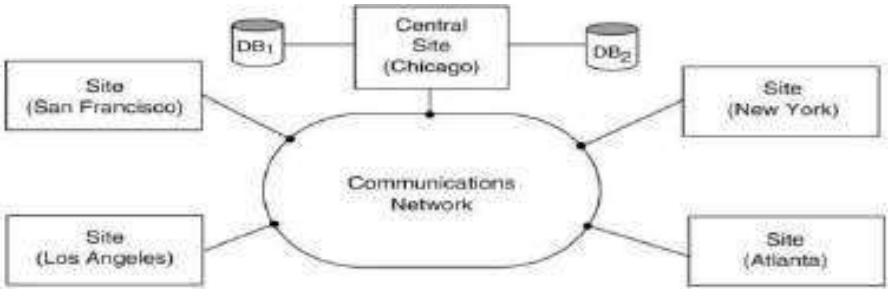


Fig : Centralized database

Distributed database

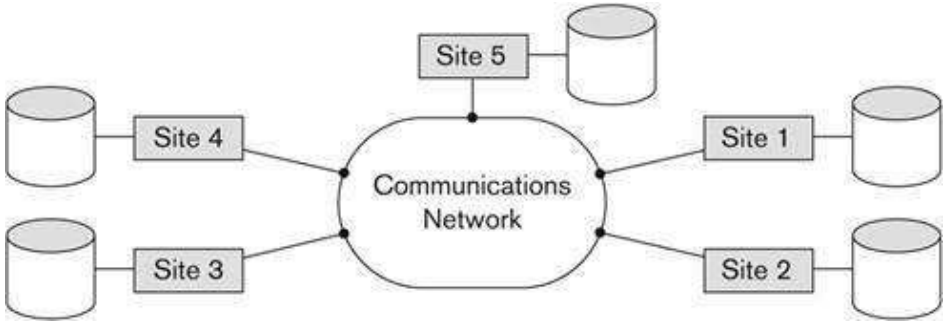


Fig : Distributed database

Types of Distributed Databases

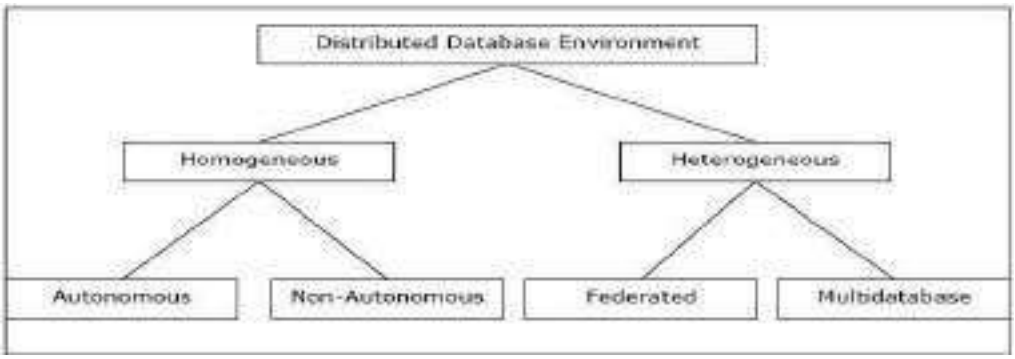


Fig : Types of Distributed Databases

Examples of distributed databases: Some common examples of distributed databases include:

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

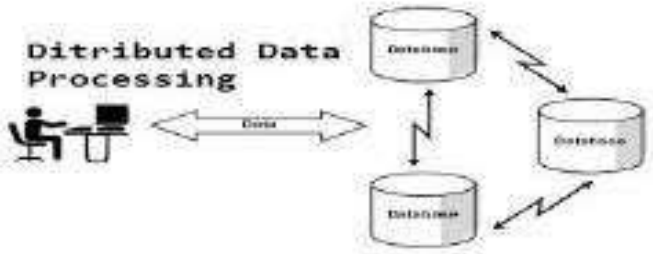
Distributed data processing

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.

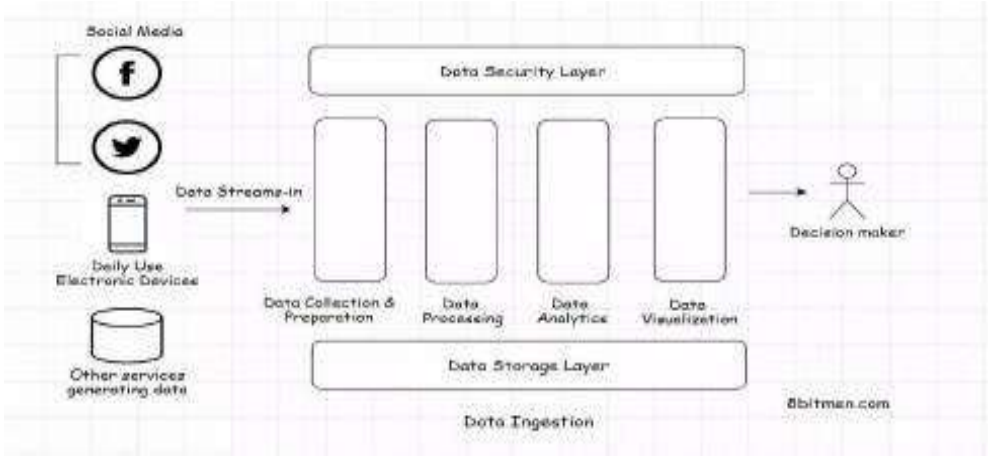


Fig : Data Ingestion

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
- Easier system 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 store this 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.***
- Sometimes 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.

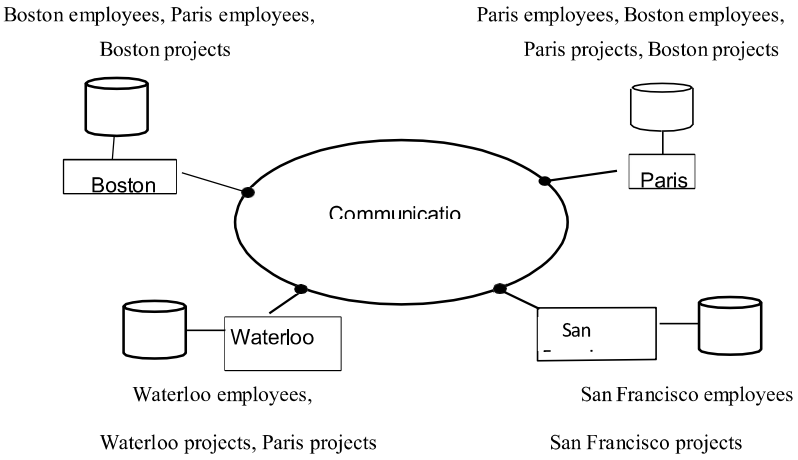


Fig. A Distributed Application

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.
- Theformer 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**

       Logicaldata 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

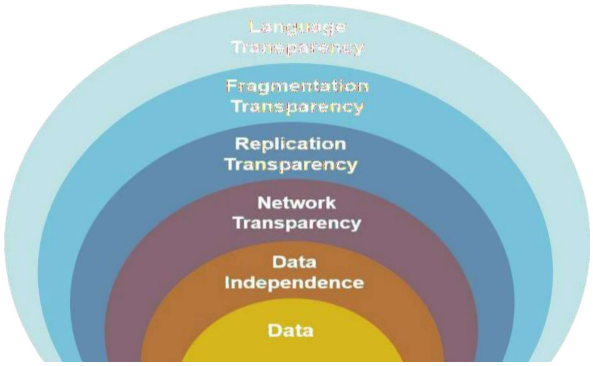


Fig : Levels of Transparency

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. Multi database 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.

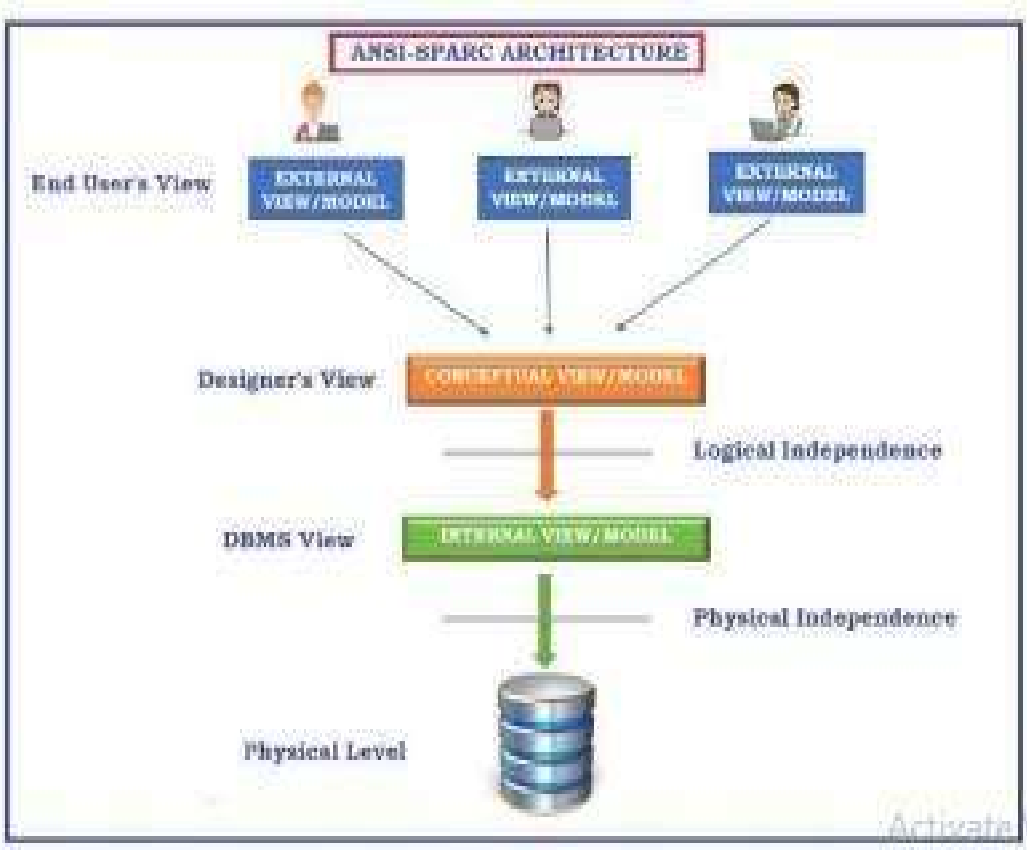


Fig : ANSI/ SPARC ARCHITECTURE

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.

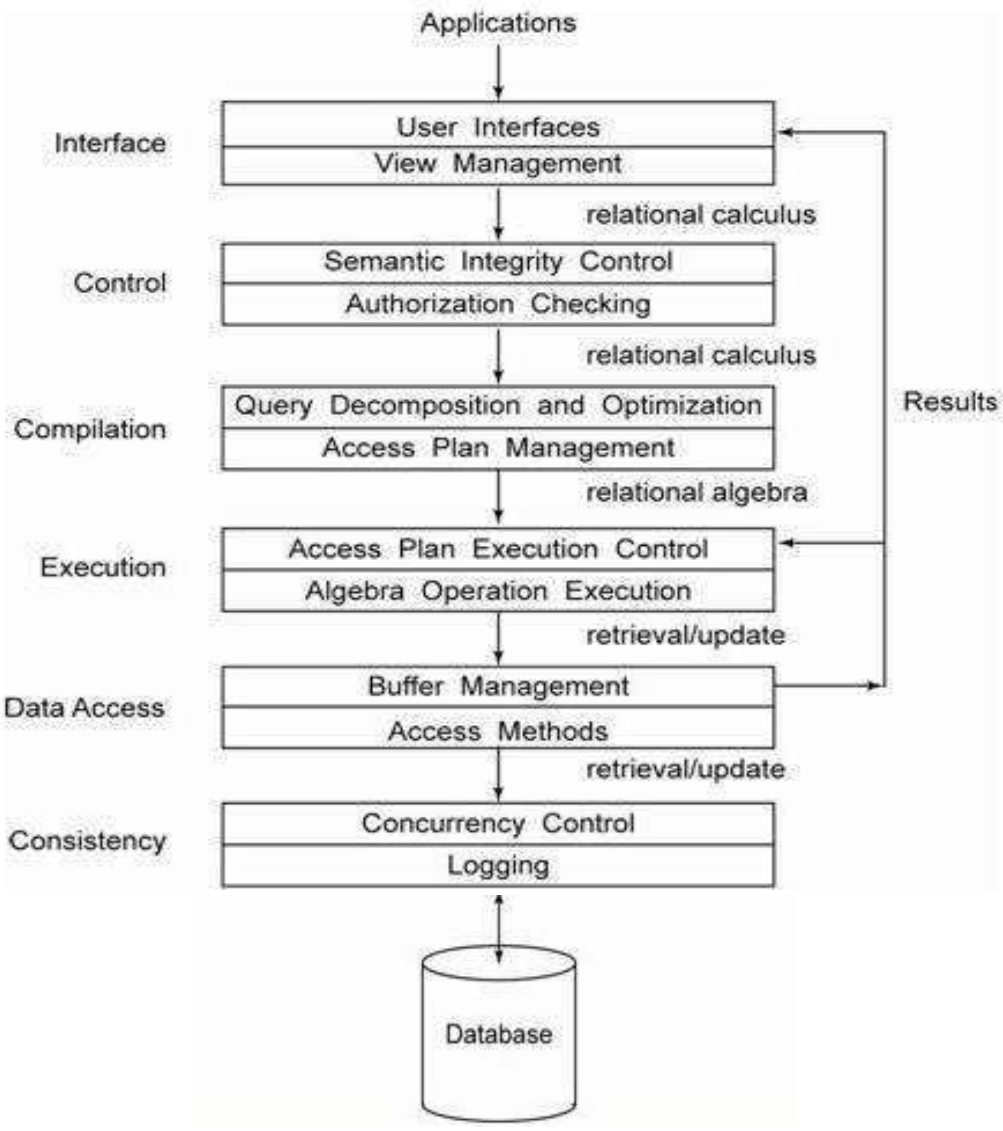


Fig : Centralized DBMS Architecture

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

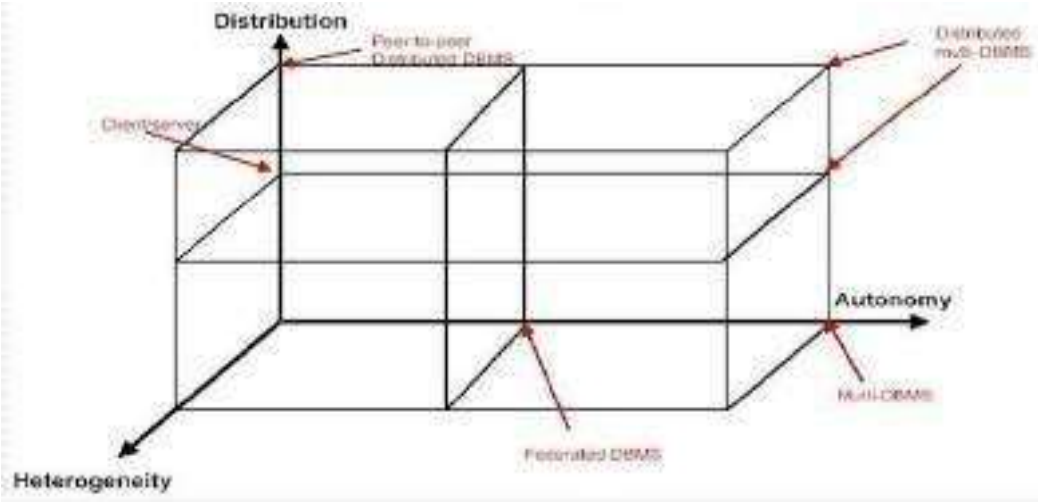


Fig : Architectural Models for DDBMS

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 tomanage 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 theuser 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.

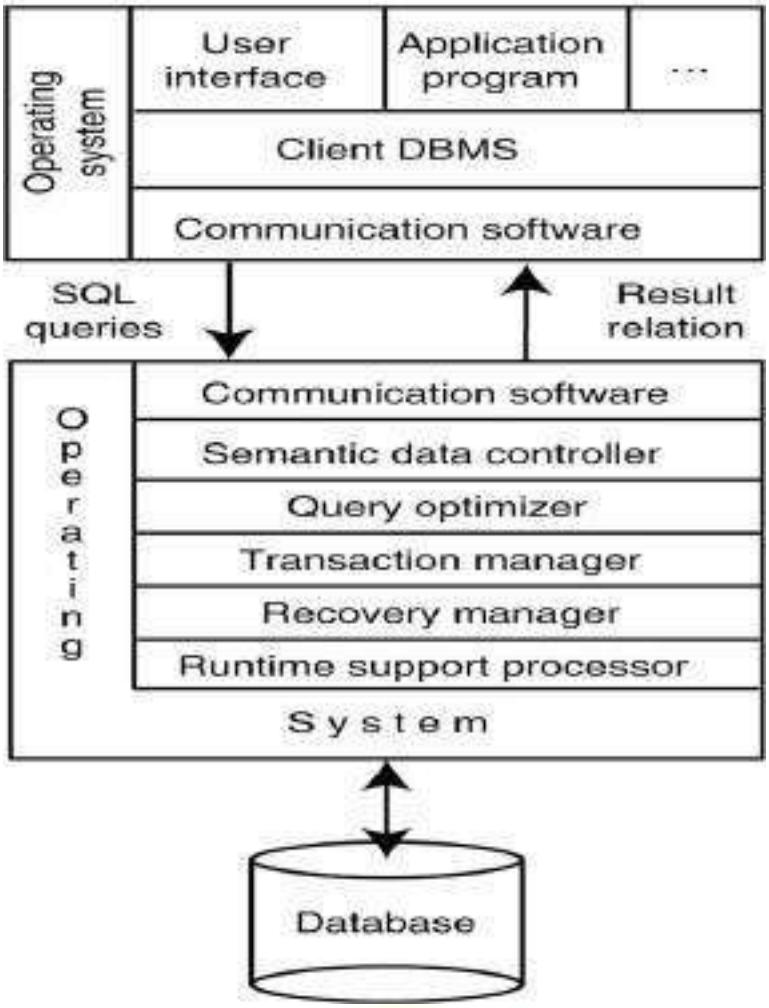


Fig: Client/ Server Reference Architecture

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

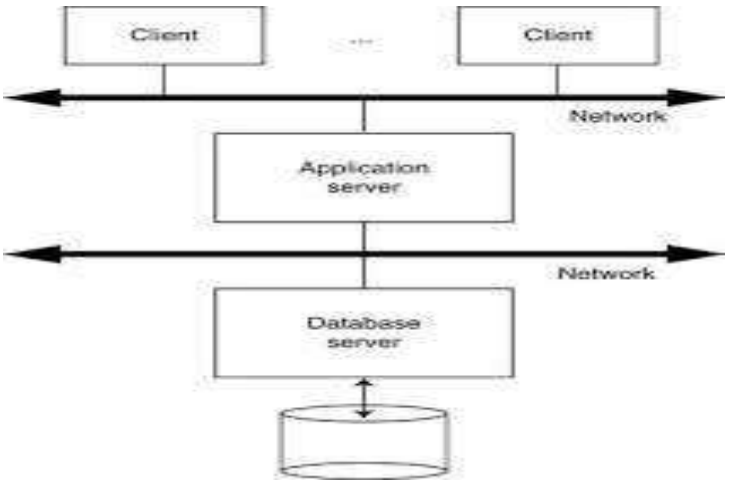


Fig : Database Server Approach

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 databasemanagement 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 applicationservers can be done in classical client/server architectures.

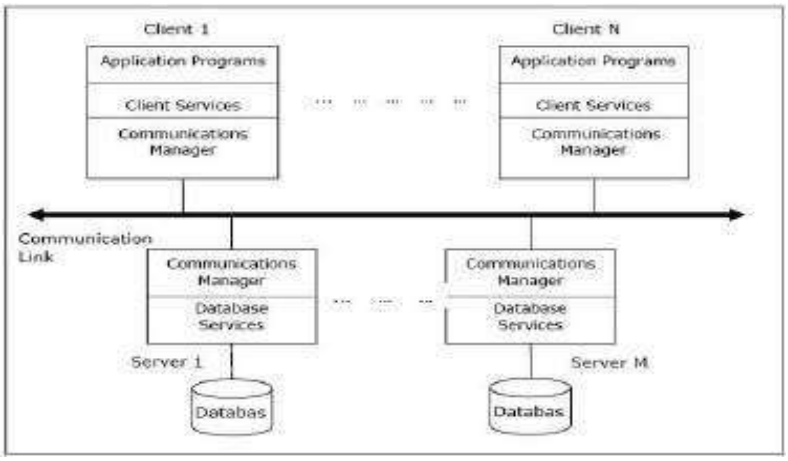


Fig : Distributed Database Servers

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.

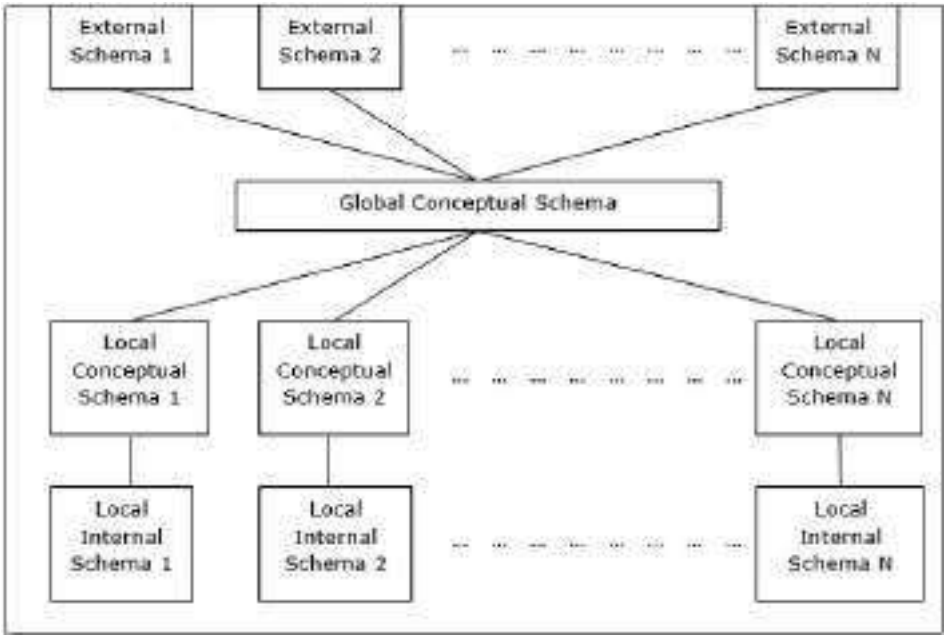


Fig : Distributed Database Reference Architecture

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

Global conceptual schema (GCS) : The enterpriseview 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:

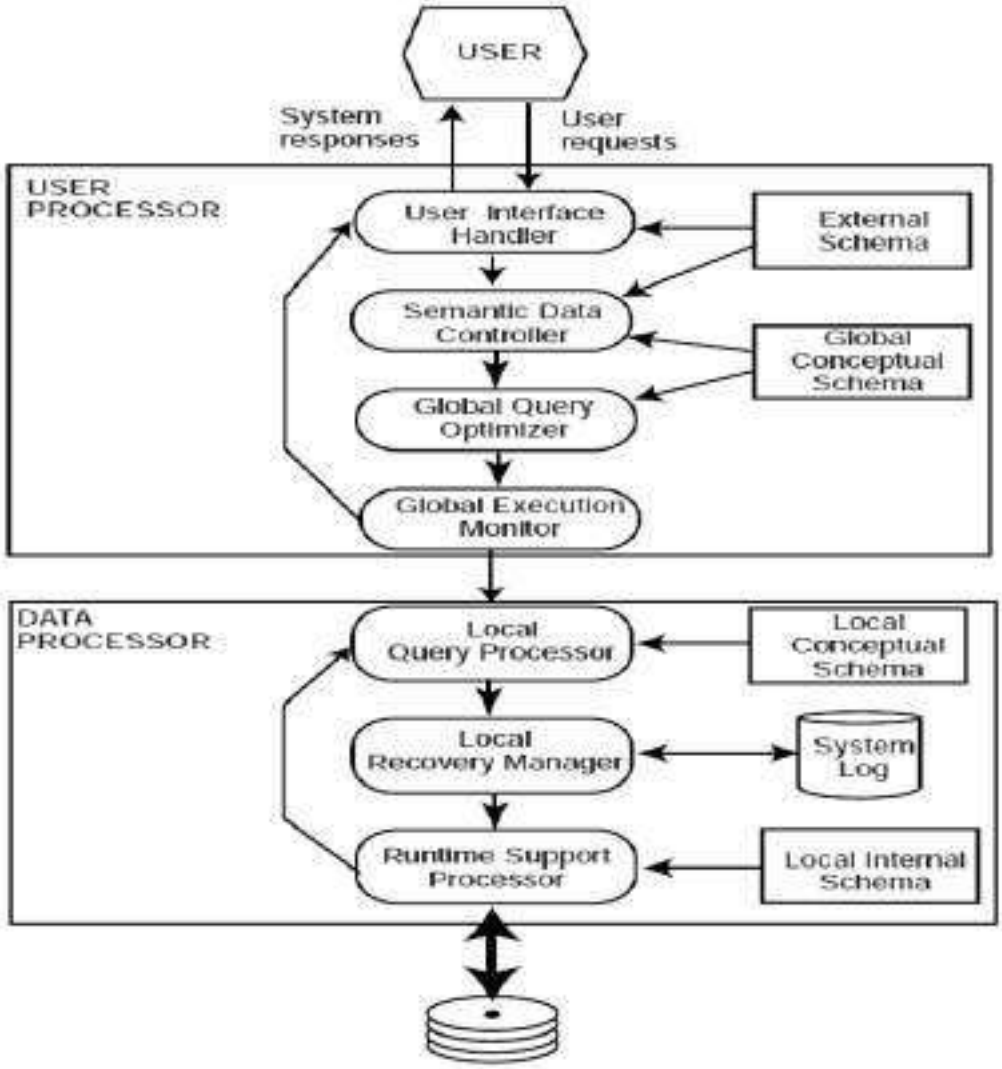


Fig : Components of a Distributed DBMS

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 userquery 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.

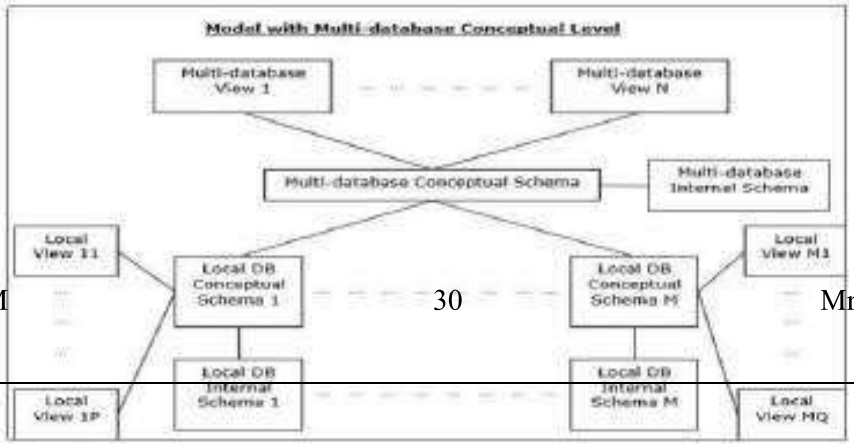
The run-time support processor is the interface to the operating system and contains the *database buffer* (or *cache*) *manager*, which is responsible for maintaining the main memory buffers and managing the data accesses.

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

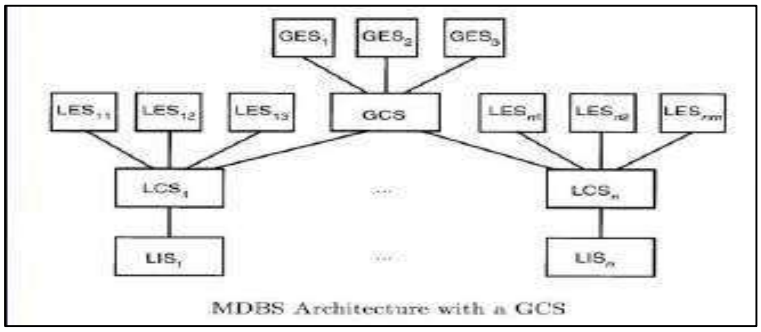
- GCS is defined by integrating either the external schemas of local autonomous databases or parts of their local conceptual schema
- Users of a local DBMS define their own views on the local database.
- If heterogeneity exists in the system, then two implementation alternatives exist: unilingual and 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

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



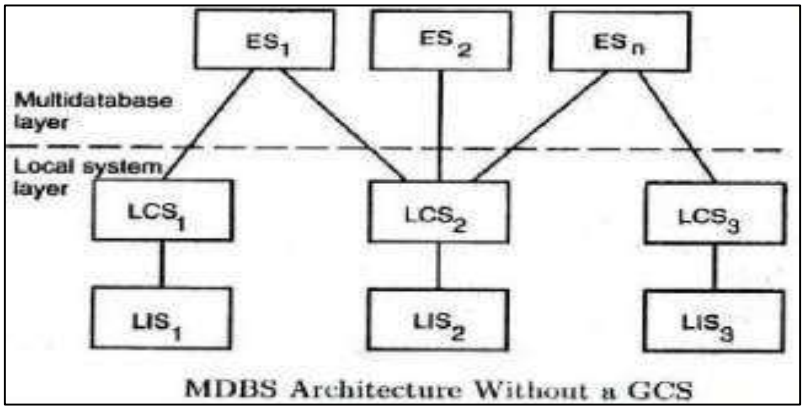
MDBS Architecture with a GCS

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

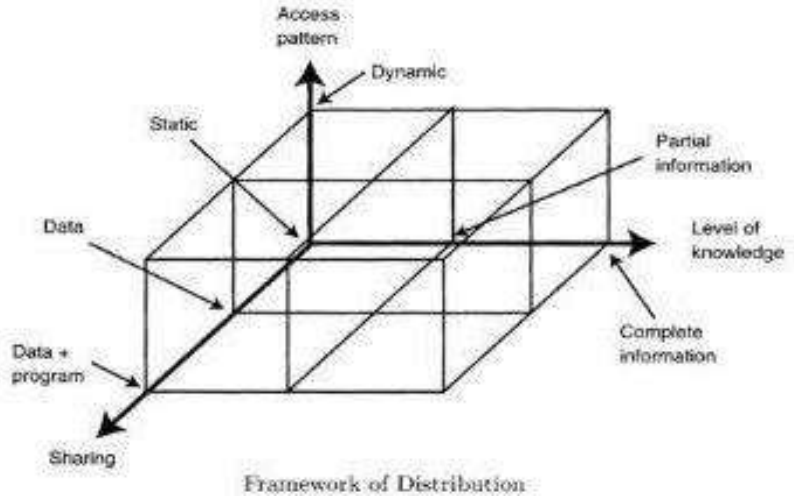


MDBS Architecture Without a GCS

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.

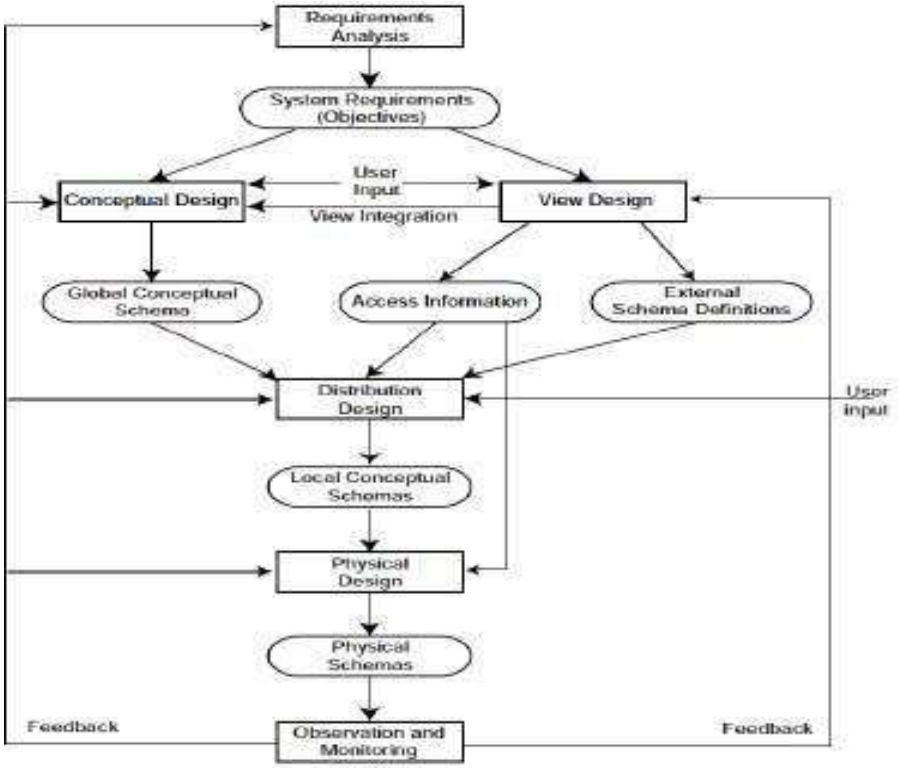


Fig : Top-Down Design Process

- 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 becross-referenced to get a better understanding of which functions deal with whichentities.
- 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 conceptualschemas (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 designactivity 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

ROJ1

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

   $R = Q R_i, \ \forall R_i \in F_R$
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 semi joins.
- 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.

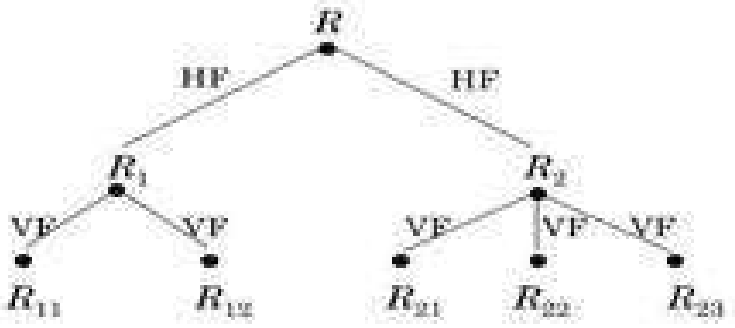


Fig : Hybrid Fragmentation

- 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.

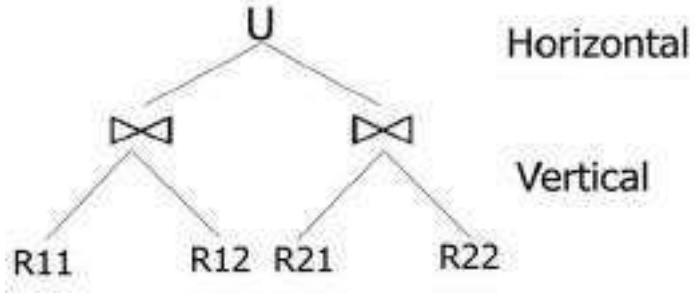


Fig : Reconstruction of Hybrid Fragmentation

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 $F = \{F_1, F_2, ..., F_n\}$ and a distributed system consisting of sites $S = \{S_1, S_2, ..., S_m\}$ on which a set of applications $Q = \{q_1, q_2, ..., q_q\}$ 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:

$u_{ij} =$ 1 if query q_i updates fragment F_j
0 otherwise

$r_{ij} =$ 1 if query q_i retrieves from fragment F_j
0 otherwise

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 $g_{i\,j}$ denotes the communication cost per frame between sites S_i and S_j. To enable the calculation of the number of messages, we use *f size* 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.