

## INTRODUCTION TO DISTRIBUTED DATABASE

A **database** is an ordered collection of related data that is built for a specific purpose. A database may be organized as a collection of multiple tables, where a table represents a real world element or entity. Each table has several different fields that represent the characteristic features of the entity.

For example, a company database may include tables for projects, employees, departments, products and financial records. The fields in the Employee table may be Name, Company\_Id, Date\_of\_Joining, and so forth.

A **database management system** is a collection of programs that enables creation and maintenance of a database. DBMS is available as a software package that facilitates definition, construction, manipulation and sharing of data in a database. Definition of a database includes description of the structure of a database. Construction of a database involves actual storing of the data in any storage medium. Manipulation refers to the retrieving information from the database, updating the database and generating reports. Sharing of data facilitates data to be accessed by different users or programs.

### Examples of DBMS Application Areas

Automatic Teller Machines Train Reservation System Employee Management System  
Student Information System

### Examples of DBMS Packages

MySQL

Oracle

SQL Server dBASE

FoxPro PostgreSQL, etc.

### Database Schemas

A database schema is a description of the database which is specified during database design and subject to infrequent alterations. It defines the organization of the data, the relationships among them, and the constraints associated with them. Databases are often represented through the three-schema architecture or ANSI/SPARC architecture. The goal of this architecture is to separate the user application from the physical database. The three levels are –

Internal Level having Internal Schema – It describes the physical structure, details of internal storage and access paths for the database.

Conceptual Level having Conceptual Schema – It describes the structure of the whole database while hiding the details of physical storage of data. This illustrates the entities, attributes with their data types and constraints, user operations and relationships.

External or View Level having External Schemas or Views – It describes the portion of a database relevant to a particular user or a group of users while hiding the rest of database.

## **Types of DBMS**

### **Hierarchical DBMS**

In hierarchical DBMS, the relationships among data in the database are established so that one data element exists as a subordinate of another. The data elements have parent-child relationships and are modelled using the “tree” data structure. These are very fast and simple.

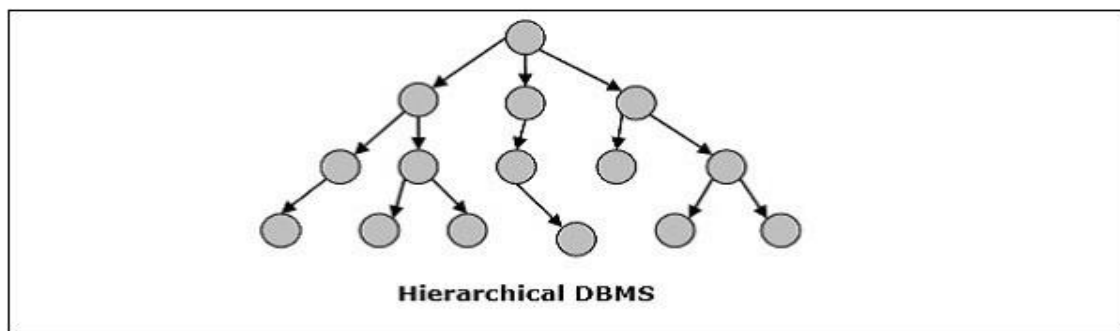


Figure 1.1 Hierarchical DBMS

### **Network DBMS**

Network DBMS is one where the relationships among data in the database are of type many-to-many in the form of a network. The structure is generally complicated due to the existence of numerous many-to-many relationships. Network DBMS is modelled using “graph” data structure.

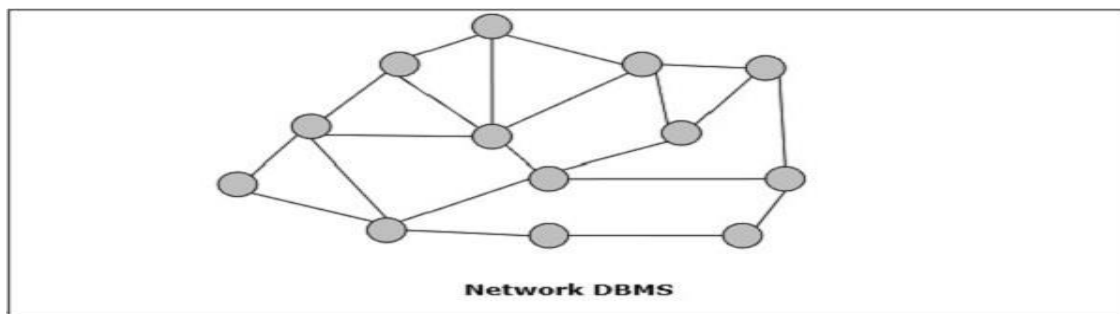
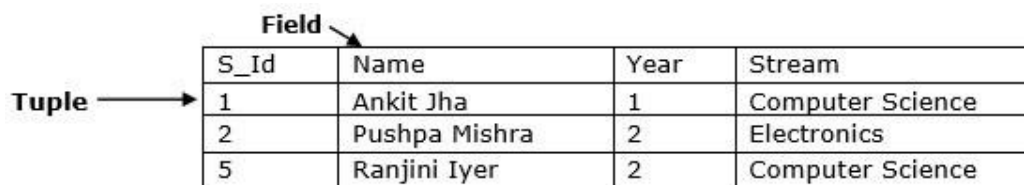


Figure 1.2 Network DBMS

### **Relational DBMS**

In relational databases, the database is represented in the form of relations. Each relation models an entity and is represented as a table of values. In the relation or table, a row is called a tuple and denotes a single record. A column is called a field or an attribute and denotes a characteristic property of the entity. RDBMS is the most popular database management system.

For example – A Student Relation –



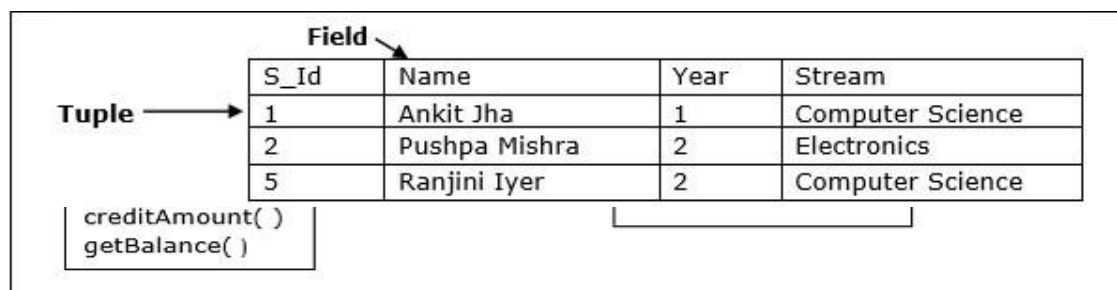
S_Id	Name	Year	Stream
1	Ankit Jha	1	Computer Science
2	Pushpa Mishra	2	Electronics
5	Ranjini Iyer	2	Computer Science

Figure 1.3 A Student Relation

## Object Oriented DBMS

Object-oriented DBMS is derived from the model of the object-oriented programming paradigm. They are helpful in representing both consistent data as stored in databases, as well as transient data, as found in executing programs. They use small, reusable elements called objects. Each object contains a data part and a set of operations which works upon the data. The object and its attributes are accessed through pointers instead of being stored in relational table models.

For example – A simplified Bank Account object-oriented database –



S_Id	Name	Year	Stream
1	Ankit Jha	1	Computer Science
2	Pushpa Mishra	2	Electronics
5	Ranjini Iyer	2	Computer Science

`creditAmount( )`  
`getBalance( )`

Figure 1.4 A simplified Bank Account object-oriented database

## Distributed DBMS

A distributed database is a set of interconnected databases that is distributed over the computer network or internet. A Distributed Database Management System (DDBMS) manages the distributed database and provides mechanisms so as to make the databases transparent to the users. In these systems, data is intentionally distributed among multiple nodes so that all computing resources of the organization can be optimally used.

A **distributed database** is a collection of multiple interconnected databases, which are spread physically across various locations that communicate via a computer network.

### Features

- Databases in the collection are logically interrelated with each other. Often they represent a single logical database.
- Data is physically stored across multiple sites. Data in each site can be managed by a DBMS independent of the other sites.
- The processors in the sites are connected via a network. They do not have any multiprocessor configuration.

- A distributed database is not a loosely connected file system.
- A distributed database incorporates transaction processing, but it is not synonymous with a transaction processing system.

### **Distributed Database Management System**

A distributed database management system (DDBMS) is a centralized software system that manages a distributed database in a manner as if it were all stored in a single location.

#### **Features**

- It is used to create, retrieve, update and delete distributed databases.
- It synchronizes the database periodically and provides access mechanisms by the virtue of which the distribution becomes transparent to the users.
- It ensures that the data modified at any site is universally updated.
- It is used in application areas where large volumes of data are processed and accessed by numerous users simultaneously.
- It is designed for heterogeneous database platforms.
- It maintains confidentiality and data integrity of the databases.

#### **Factors Encouraging DDBMS**

- **Distributed Nature of Organizational Units** – Most organizations in the current times are subdivided into multiple units that are physically distributed over the globe. Each unit requires its own set of local data. Thus, the overall database of the organization becomes distributed.
- **Need for Sharing of Data** – The multiple organizational units often need to communicate with each other and share their data and resources. This demands common databases or replicated databases that should be used in a synchronized manner.
- **Support for Both OLTP and OLAP** – Online Transaction Processing (OLTP) and Online Analytical Processing (OLAP) work upon diversified systems which may have common data. Distributed database systems aid both these processing by providing synchronized data.
- **Database Recovery** – One of the common techniques used in DDBMS is replication of data across different sites. Replication of data automatically helps in data recovery if database in any site is damaged. Users can access data from other sites while the damaged site is being reconstructed. Thus, database failure may become almost inconspicuous to users.
- **Support for Multiple Application Software** – Most organizations use a variety of application software each with its specific database support. DDBMS provides a uniform functionality for using the same data among different platforms.

### Advantages of Distributed Databases

- **Modular Development** – If the system needs to be expanded to new locations or new units, in centralized database systems, the action requires substantial efforts and disruption in the existing functioning. However, in distributed databases, the work simply requires adding new computers and local data to the new site and finally connecting them to the distributed system, with no interruption in current functions.
- **More Reliable** – In case of database failures, the total system of centralized databases comes to a halt. However, in distributed systems, when a component fails, the functioning of the system continues may be at a reduced performance. Hence DDBMS is more reliable.
- **Better Response** – If data is distributed in an efficient manner, then user requests can be met from local data itself, thus providing faster response. On the other hand, in centralized systems, all queries have to pass through the central computer for processing, which increases the response time.
- **Lower Communication Cost** – In distributed database systems, if data is located locally where it is mostly used, then the communication costs for data manipulation can be minimized. This is not feasible in centralized systems.

### Adversities of Distributed Databases

- **Need for complex and expensive software** – DDBMS demands complex and often expensive software to provide data transparency and co-ordination across the several sites.
- **Processing overhead** – Even simple operations may require a large number of communications and additional calculations to provide uniformity in data across the sites.
- **Data integrity** – The need for updating data in multiple sites pose problems of data integrity.
- **Overheads for improper data distribution** – Responsiveness of queries is largely dependent upon proper data distribution. Improper data distribution often leads to very slow response to user requests.

### Distributed Database Vs Centralized Database

<b><i>Centralized DBMS</i></b>	<b><i>Distributed DBMS</i></b>
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

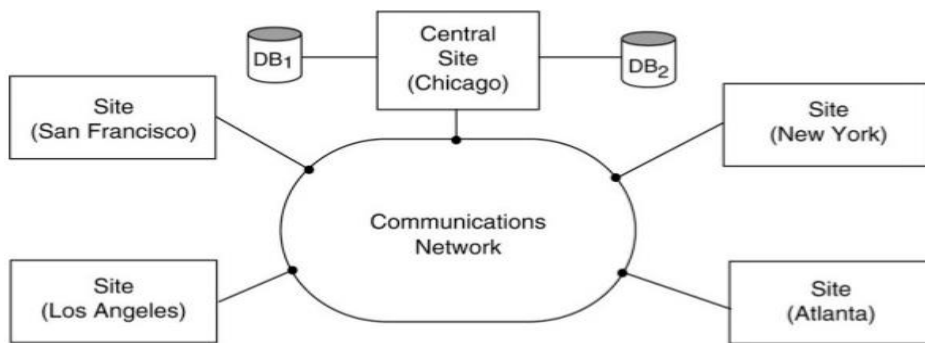


Figure 1.5 Centralized database

### Distributed database

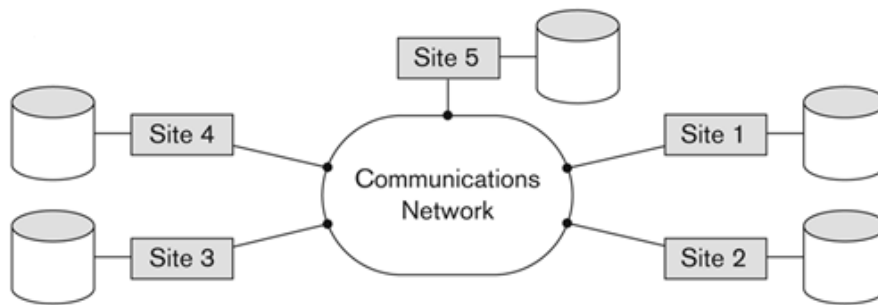


Figure1. 6 Distributed database

### Types of Distributed Databases

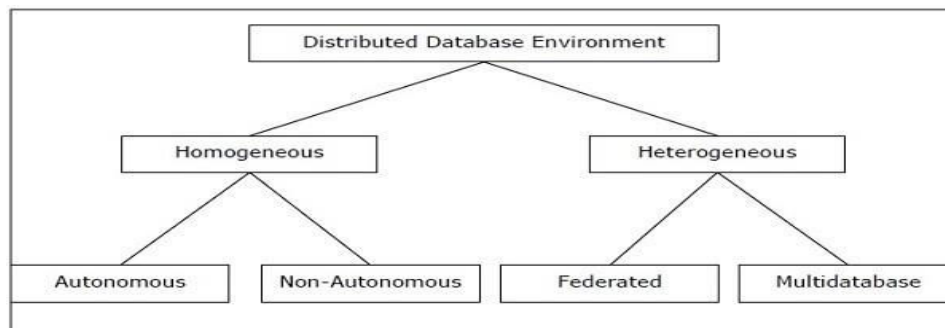


Figure 1.7 Types of Distributed Databases

Distributed databases can be broadly classified into homogeneous and heterogeneous distributed database environments

### **Homogeneous Distributed Databases**

In a homogeneous distributed database, all the sites use identical DBMS and operating systems. Its properties are –

- The sites use very similar software.
- The sites use identical DBMS or DBMS from the same vendor.
- Each site is aware of all other sites and cooperates with other sites to process user requests.
- The database is accessed through a single interface as if it is a single database.

#### Types of Homogeneous Distributed Database

There are two types of homogeneous distributed database –

**Autonomous** – Each database is independent that functions on its own. They are integrated by a controlling application and use message passing to share data updates.

**Non-autonomous** – Data is distributed across the homogeneous nodes and a central or master DBMS co-ordinates data updates across the sites.

### **Heterogeneous Distributed Databases**

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

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

#### Types of Heterogeneous Distributed Databases

**Federated** – The heterogeneous database systems are independent in nature and integrated together so that they function as a single database system.

**Un-federated** – The database systems employ a central coordinating module through which the databases are accessed.

### **Distributed DBMS Architectures**

DDBMS architectures are generally developed depending on three parameters –

- **Distribution** – It states the physical distribution of data across the different sites.

- Autonomy – It indicates the distribution of control of the database system and the degree to which each constituent DBMS can operate independently.
- Heterogeneity – It refers to the uniformity or dissimilarity of the data models, system components and databases.

### **Architectural Models**

- Client - Server Architecture for DDBMS
- Peer - to - Peer Architecture for DDBMS
- Multi - DBMS Architecture

### **Client - Server Architecture for DDBMS**

This is a two-level architecture where the functionality is divided into servers and clients. The server functions primarily encompass data management, query processing, optimization and transaction management. Client functions include mainly user interface. However, they have some functions like consistency checking and transaction management.

Distinguish the functionality and divide these functions into two classes, server functions and client functions.

Server does most of the data management work

- query processing
- data management
- Optimization
- Transaction management etc

Client performs

- Application
- User interface
- DBMS Client model

The two different client - server architecture are –

Single Server Multiple Client

Single Server accessed by multiple clients

- A client server architecture has a number of clients and a few servers connected in a network.
- A client sends a query to one of the servers. The earliest available server solves it and replies.
- A Client-server architecture is simple to implement and execute due to centralized server system.



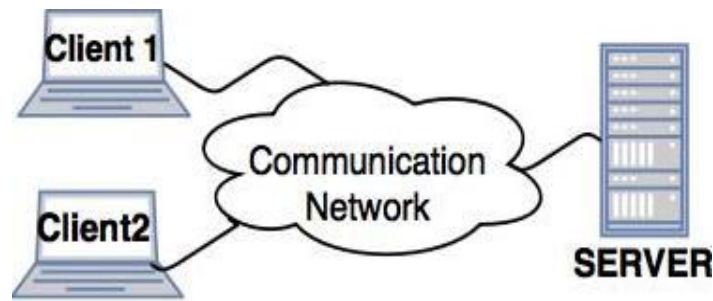


Figure1. 8 Single Server Multiple Client

### Multiple Server Multiple Client

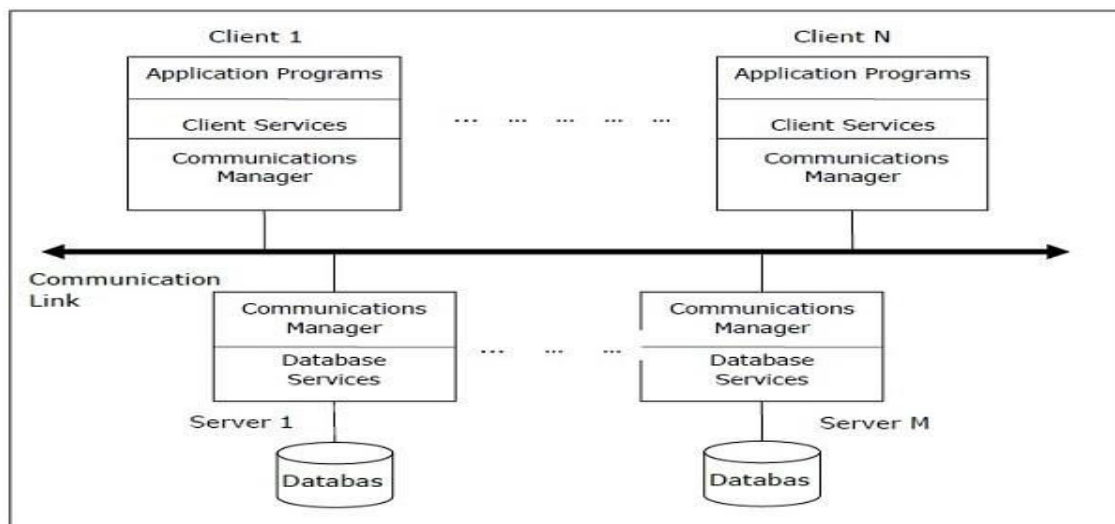


Figure 1. 9 Multiple Servers accessed by multiple clients

### Peer- to-Peer Architecture for DDBMS

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 –

#### Schemas Present

Individual internal schema definition at each site, *local internal schema*

Enterprise view of data is described the *global conceptual schema*.

Local organization of data at each site is describe in the *local conceptual schema*.

User applications and user access to the database is supported by *external schemas*

Local conceptual schemas are mappings of the global schema onto each site.

Databases are typically designed in a top-down fashion, and, therefore all external view definitions are made globally.

### Major Components of a Peer-to-Peer System

- User Processor
- Data processor

#### User Processor

- User-interface handler
- responsible for interpreting user commands, and formatting the result data
- Semantic data controller
- checks if the user query can be processed.
- Global Query optimizer and decomposer
- determines an execution strategy
- Translates global queries into local one.
- Distributed execution
- Coordinates the distributed execution of the user request

#### Data processor

- Local query optimizer
- Acts as the access path selector
- Responsible for choosing the best access path
- Local Recovery Manager

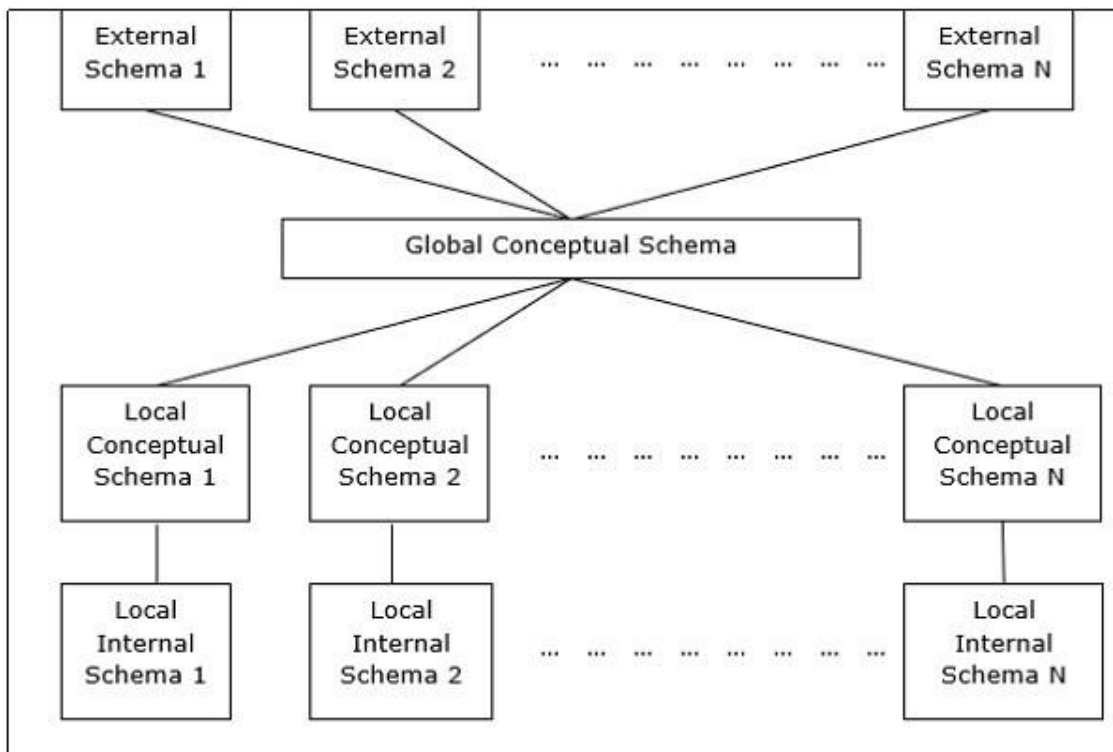


Figure 1.10 Frame Work

- Makes sure local database remains consistent
- Run-time support processor
- Is the interface to the operating system and contains the database buffer
- Responsible for maintaining the main memory buffers and managing the data access.

### Multi - DBMS Architectures

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 –

- Multi-database View Level – Depicts multiple user views comprising of subsets of the integrated distributed database.
- Multi-database Conceptual Level – Depicts integrated multi-database that comprises of global logical multi-database structure definitions.
- Multi-database Internal Level – Depicts the data distribution across different sites and multi-database to local data mapping.
- Local database View Level – Depicts public view of local data.
- Local database Conceptual Level – Depicts local data organization at each site.
- 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.

#### *Models Using a Global Conceptual Schema*

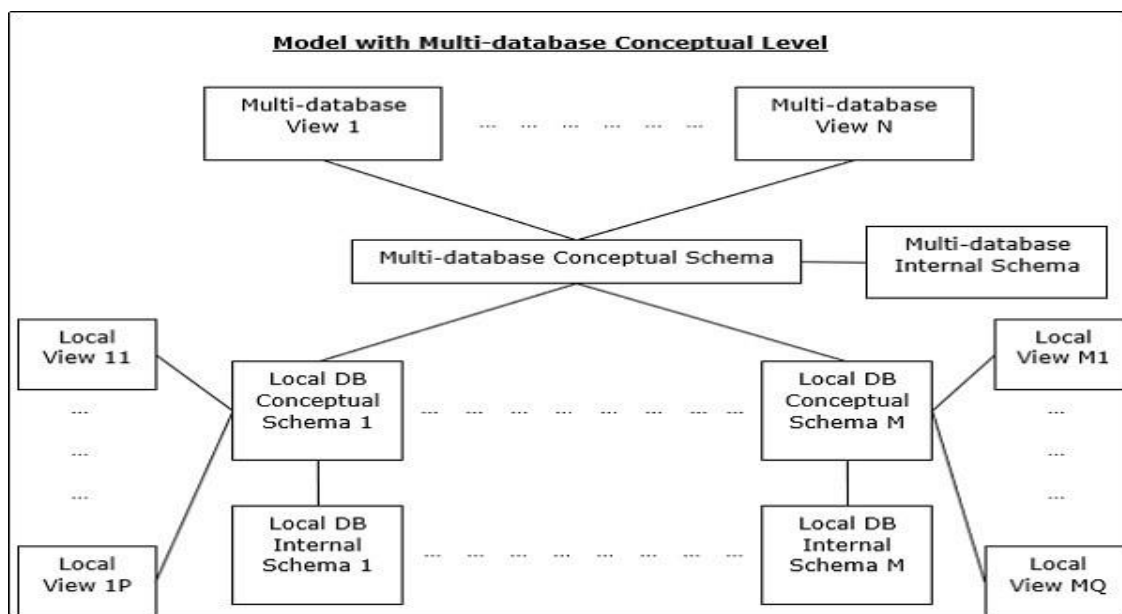


Figure 1.11 Models Using a Global Conceptual Schema

- 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 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

### **Global Directory Issues**

Global Directory is an extension of the normal directory, including information about the location of the fragments as well as the makeup of the fragments, for cases of distributed DBMS or a multi-DBMS, that uses a global conceptual schema,

- Relevant for distributed DBMS or a multi-DBMS that uses a global conceptual schema
- Includes information about the location of the fragments as well as the makeup of fragments.
- Directory is itself a database that contains meta-data about the actual data stored in database.

### **Three issues**

- A directory may either be global to the entire database or local to each site.
- Directory may be maintained centrally at one site, or in a distributed fashion by distributing it over a number of sites.
  - If system is distributed, directory is always distributed

- Replication may be single copy or multiple copies.
  - Multiple copies would provide more reliability

## **Organization of Distributed systems**

Three orthogonal dimensions

- 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

## **Top Down Design**

- Suitable for applications where database needs to be build from scratch
- Activity begins with requirement analysis
- Requirement document is input to two parallel activities:
  - view design activity, deals with defining the interfaces for end users
  - conceptual design, process by which enterprise is examined
    - Can be further divided into 2 related activity groups
    - Entity analyses, concerned with determining the entities, attributes and the relationship between them
    - Functional analyses, concerned with determining the fun
  - Distributed design activity consists of two steps
    - Fragmentation
    - Allocation

## **Bottom-Up Approach**

- Suitable for applications where database already exists
- Starting point is individual conceptual schemas
- Exists primarily in the context of heterogeneous database.

## **Design Alternatives**

The distribution design alternatives for the tables in a DDBMS are as follows –

Non-replicated and non-fragmented

Fully replicated

Partially replicated Fragmented

Mixed

### **Non-replicated & Non-fragmented**

In this design alternative, different tables are placed at different sites. Data is placed so that it is at a close proximity to the site where it is used most. It is most suitable for database systems where the percentage of queries needed to join information in tables placed at different sites is low. If an appropriate distribution strategy is adopted, then this design alternative helps to reduce the communication cost during data processing.

### **Fully Replicated**

In this design alternative, at each site, one copy of all the database tables is stored. Since, each site has its own copy of the entire database, queries are very fast requiring negligible communication cost. On the contrary, the massive redundancy in data requires huge cost during update operations. Hence, this is suitable for systems where a large number of queries is required to be handled whereas the number of database updates is low.

### **Partially Replicated**

Copies of tables or portions of tables are stored at different sites. The distribution of the tables is done in accordance to the frequency of access. This takes into consideration the fact that the frequency of accessing the tables vary considerably from site to site. The number of copies of the tables (or portions) depends on how frequently the access queries execute and the site which generate the access queries.

### **Fragmented**

In this design, a table is divided into two or more pieces referred to as fragments or partitions, and each fragment can be stored at different sites. This considers the fact that it seldom happens that all data stored in a table is required at a given site. Moreover, fragmentation increases parallelism and provides better disaster recovery. Here, there is only one copy of each fragment in the system, i.e. no redundant data.

The three fragmentation techniques are –

- Vertical fragmentation
- Horizontal fragmentation
- Hybrid fragmentation

**Mixed Distribution:** This is a combination of fragmentation and partial replications. Here, the tables are initially fragmented in any form (horizontal or vertical), and then these fragments are partially replicated across the different sites according to the frequency of accessing the fragments.

## **Design Strategies**

In the last chapter, we had introduced different design alternatives. In this chapter, we will study the strategies that aid in adopting the designs. The strategies can be broadly divided into replication and fragmentation. However, in most cases, a combination of the two is used.

### **Data Replication**

Data replication is the process of storing separate copies of the database at two or more sites. It is a popular fault tolerance technique of distributed databases.

#### **Advantages of Data Replication**

- Reliability – In case of failure of any site, the database system continues to work since a copy is available at another site(s).
- Reduction in Network Load – Since local copies of data are available, query processing can be done with reduced network usage, particularly during prime hours. Data updating can be done at non-prime hours.
- Quicker Response – Availability of local copies of data ensures quick query processing and consequently quick response time.
- Simpler Transactions – Transactions require less number of joins of tables located at different sites and minimal coordination across the network. Thus, they become simpler in nature.

#### **Disadvantages of Data Replication**

- Increased Storage Requirements – Maintaining multiple copies of data is associated with increased storage costs. The storage space required is in multiples of the storage required for a centralized system.
- Increased Cost and Complexity of Data Updating – Each time a data item is updated, the update needs to be reflected in all the copies of the data at the different sites. This requires complex synchronization techniques and protocols.
- Undesirable Application – Database coupling – If complex update mechanisms are not used, removing data inconsistency requires complex co-ordination at application level. This results in undesirable application – database coupling.

Some commonly used replication techniques are

Snapshot replication

Near-real-time replication

Pull replication

### **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). Horizontal fragmentation can further be classified into two techniques: primary horizontal fragmentation and derived horizontal fragmentation.

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 be 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.

### **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

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

fragment

### Horizontal Fragmentation

Horizontal fragmentation groups the tuples of a table in accordance to values of one or more fields. Horizontal fragmentation should also confirm to the rule of reconstructiveness. Each horizontal fragment must have all columns of the original base table.

- 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

$$R_i = \sigma_{F_i}(R), \quad 1 \leq i \leq w$$

$F_i$  selection formula used to obtain fragment  $R_i$

The example mentioned in slide 20, can be represented by using the above formula as

$$Emp_1 = \sigma_{Sal \leq 20K}(Emp)$$

$$Emp_2 = \sigma_{Sal > 20K}(Emp)$$

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

$$R_i = R \alpha S_i, \quad 1 \leq i \leq w$$

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

### 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.

Hybrid fragmentation can be done in two alternative ways –

At first, generate a set of horizontal fragments; then generate vertical fragments from one or more of the horizontal fragments.

At first, generate a set of vertical fragments; then generate horizontal fragments from one or more of the vertical fragments.

### Transparency

**Transparency** in DBMS stands for the separation of high level semantics of the system from the low-level implementation issue. High-level semantics stands for the endpoint user, and low level implementation concerns with complicated hardware implementation of data or how the data has been stored in the database. Using data independence in various layers of the database, transparency can be implemented in DBMS.

Distribution transparency is the property of distributed databases by the virtue of which the internal details of the distribution are hidden from the users. The DDBMS designer may choose to fragment tables, replicate the fragments and store them at different sites. However, since users are oblivious of these details, they find the distributed database easy to use like any centralized database.

Unlike normal DBMS, DDBMS deals with communication network, replicas and fragments of data. Thus, transparency also involves these three factors.

Following are three types of transparency:

1. Location transparency
2. Fragmentation transparency
3. Replication transparency

### Location Transparency

Location transparency ensures that the user can query on any table(s) or fragment(s) of a table as if they were stored locally in the user's site. The fact that the table or its fragments

are stored at remote site in the distributed database system, should be completely oblivious to the end user. The address of the remote site(s) and the access mechanisms are completely hidden. In order to incorporate location transparency, DDBMS should have access to updated and accurate data dictionary and DDBMS directory which contains the details of locations of data.

### **Fragmentation Transparency**

Fragmentation transparency enables users to query upon any table as if it were unfragmented. Thus, it hides the fact that the table the user is querying on is actually a fragment or union of some fragments. It also conceals the fact that the fragments are located at diverse sites. This is somewhat similar to users of SQL views, where the user may not know that they are using a view of a table instead of the table itself.

### **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. Replication transparency is associated with concurrency transparency and failure transparency. Whenever a user updates a data item, the update is reflected in all the copies of the table. However, this operation should not be known to the user. This is concurrency transparency. Also, 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.

### **Combination of Transparencies**

In any distributed database system, the designer should ensure that all the stated transparencies are maintained to a considerable extent. The designer may choose to fragment tables, replicate them and store them at different sites; all oblivious to the end user. However, complete distribution transparency is a tough task and requires considerable design efforts.

### **Database Control**

Database control refers to the task of enforcing regulations so as to provide correct data to authentic users and applications of a database. In order that correct data is available to users, all data should conform to the integrity constraints defined in the database. Besides, data should be screened away from unauthorized users so as to maintain security and privacy of the database. Database control is one of the primary tasks of the database administrator (DBA).

The three dimensions of database control are –

- Authentication
- Access Control
- Integrity Constraints

### **Authentication**

In a distributed database system, authentication is the process through which only legitimate users can gain access to the data resources.

Authentication can be enforced in two levels –

**Controlling Access to Client Computer** – At this level, user access is restricted while login to the client computer that provides user-interface to the database server. The most common method is a username/password combination. However, more sophisticated methods like biometric authentication may be used for high security data.

**Controlling Access to the Database Software** – At this level, the database software/administrator assigns some credentials to the user. The user gains access to the database using these credentials. One of the methods is to create a login account within the database server.

### **Access Rights**

A user's access rights refers to the privileges that the user is given regarding DBMS operations such as the rights to create a table, drop a table, add/delete/update tuples in a table or query upon the table.

In distributed environments, since there are large number of tables and yet larger number of users, it is not feasible to assign individual access rights to users. So, DDBMS defines certain roles. A role is a construct with certain privileges within a database system. Once the different roles are defined, the individual users are assigned one of these roles. Often a hierarchy of roles are defined according to the organization's hierarchy of authority and responsibility.

For example, the following SQL statements create a role "Accountant" and then assigns this role to user "ABC".

```
CREATE ROLE ACCOUNTANT;  
  
GRANT SELECT, INSERT, UPDATE ON EMP_SAL TO ACCOUNTANT; GRANT INSERT, UPDATE,  
DELETE ON TENDER TO ACCOUNTANT; GRANT INSERT, SELECT ON EXPENSE TO  
ACCOUNTANT;  
  
COMMIT;
```

### **Semantic Integrity Control**

Semantic integrity control defines and enforces the integrity constraints of the database system.

The integrity constraints are as follows –

Data type integrity constraint

Entity integrity constraint

Referential integrity constraint

#### **Data Type Integrity Constraint**

A data type constraint restricts the range of values and the type of operations that can be applied to the field with the specified data type.

For example, let us consider that a table "HOSTEL" has three fields - the hostel number, hostel name and capacity. The hostel number should start with capital letter "H" and cannot be NULL, and the capacity should not be more than 150. The following SQL command can be used for data definition –

```
CREATE TABLE HOSTEL (  
H_NO VARCHAR2(5) NOT NULL, H_NAME VARCHAR2(15), CAPACITY INTEGER,  
CHECK ( H_NO LIKE 'H%'), CHECK ( CAPACITY <= 150)  
);
```

### Entity Integrity Control

Entity integrity control enforces the rules so that each tuple can be uniquely identified from other tuples. For this a primary key is defined. A primary key is a set of minimal fields that can uniquely identify a tuple. Entity integrity constraint states that no two tuples in a table can have identical values for primary keys and that no field which is a part of the primary key can have NULL value.

For example, in the above hostel table, the hostel number can be assigned as the primary key through the following SQL statement (ignoring the checks) –

```
CREATE TABLE HOSTEL (  
H_NO VARCHAR2(5) PRIMARY KEY, H_NAME VARCHAR2(15),  
CAPACITY INTEGER);
```

### Referential Integrity Constraint

Referential integrity constraint lays down the rules of foreign keys. A foreign key is a field in a data table that is the primary key of a related table. The referential integrity constraint lays down the rule that the value of the foreign key field should either be among the values of the primary key of the referenced table or be entirely NULL.

For example, let us consider a student table where a student may opt to live in a hostel. To include this, the primary key of hostel table should be included as a foreign key in the student table. The following SQL statement incorporates this –

```
CREATE TABLE STUDENT (  
S_ROLL INTEGER PRIMARY KEY, S_NAME VARCHAR2(25) NOT NULL, S_COURSE  
VARCHAR2(10),  
S_HOSTEL VARCHAR2(5) REFERENCES HOSTEL);
```

## **Global Queries to Fragment Queries**

When a query is placed, it is at first scanned, parsed and validated. An internal representation of the query is then created such as a query tree or a query graph. Then alternative execution strategies are devised for retrieving results from the database tables. The process of choosing the most appropriate execution strategy for query processing is called query optimization.

## **Query Optimization Issues in DDBMS**

In DDBMS, query optimization is a crucial task. The complexity is high since number of alternative strategies may increase exponentially due to the following factors –

- The presence of a number of fragments.
- Distribution of the fragments or tables across various sites.
- The speed of communication links.
- Disparity in local processing capabilities.

Hence, in a distributed system, the target is often to find a good execution strategy for query processing rather than the best one. The time to execute a query is the sum of the following

- Time to communicate queries to databases.
- Time to execute local query fragments.
- Time to assemble data from different sites.
- Time to display results to the application.

## **Query Processing**

Query processing is a set of all activities starting from query placement to displaying the results of the query. The steps are as shown in the following diagram –

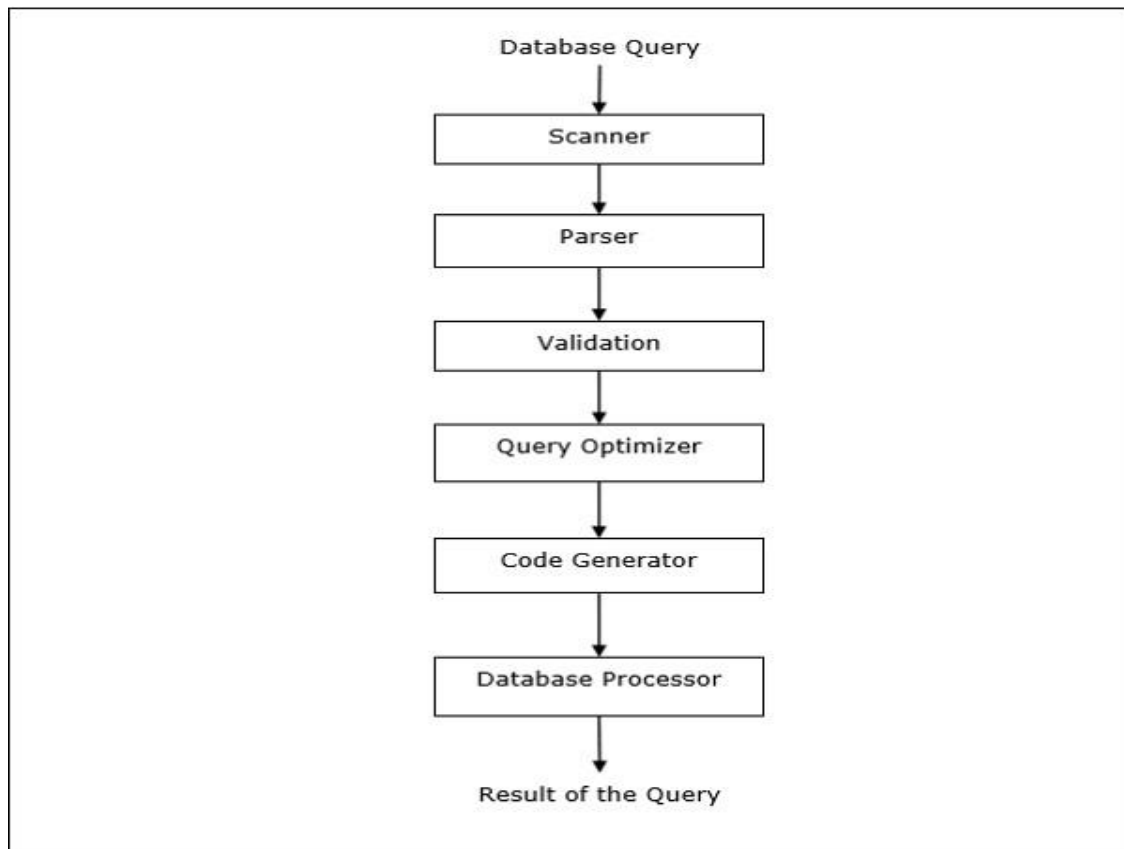


Figure 2.1 step in query processing

### Global Query Optimization

Input: Fragment query

- Find the *best* (not necessarily optimal) global schedule
  - ➔ Minimize a cost function
  - ➔ Distributed join processing
    - ◆ Bushy vs. linear trees
    - ◆ Which relation to ship where?
    - ◆ Ship-whole vs ship-as-needed
  - ➔ Decide on the use of semijoins

- ◆ Semijoin saves on communication at the expense of more local processing.

➔ Join methods

- ◆ nested loop vs ordered joins (merge join or hash join)

### **Cost-Based Optimization**

- Solution space
  - ➔ The set of equivalent algebra expressions (query trees).
- Cost function (in terms of time)
  - ➔ I/O cost + CPU cost + communication cost
  - ➔ These might have different weights in different distributed environments (LAN vs WAN).
  - ➔ Can also maximize throughput
- Search algorithm
  - ➔ How do we move inside the solution space?
  - ➔ Exhaustive search, heuristic algorithms (iterative improvement, simulated annealing, genetic,...)

### **Query Optimization Process**



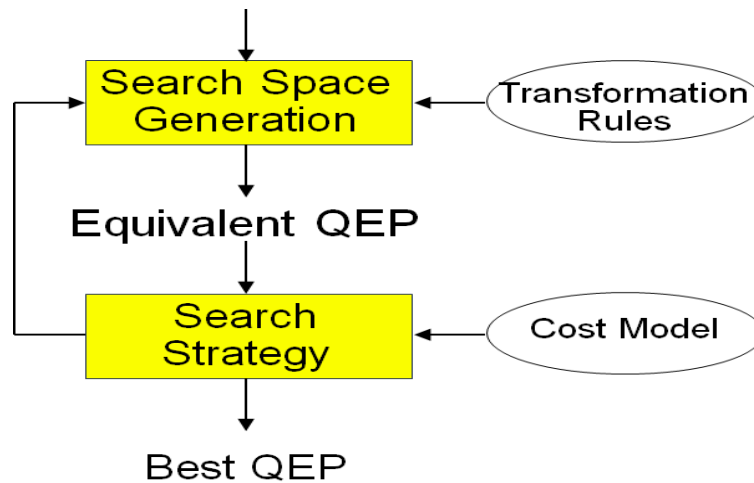


Figure 2.2 Query Optimization Process

### Search Space

- Search space characterized by alternative execution
- Focus on join trees
- For  $N$  relations, there are  $O(N!)$  equivalent join trees that can be obtained by applying commutativity and associativity rules

```

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

### Cost Functions

- Total Time (or Total Cost)
  - ➔ Reduce each cost (in terms of time) component individually
  - ➔ Do as little of each cost component as possible
  - ➔ Optimizes the utilization of the resources

Increases system throughput

- Response Time
  - ➔ Do as many things as possible in parallel
  - ➔ May increase total time because of increased total activity
- Summation of all cost factors
- Total cost = CPU cost + I/O cost + communication cost
- CPU cost = unit instruction cost \* no.of instructions
- I/O cost = unit disk I/O cost \* no. of disk I/Os
- communication cost = message initiation + transmission

## 2-Step – Problem Definition

- Given
  - ➔ A set of sites  $S = \{s_1, s_2, \dots, s_n\}$  with the load of each site
  - ➔ A query  $Q = \{q_1, q_2, q_3, q_4\}$  such that each subquery  $q_i$  is the maximum processing unit that accesses one relation and communicates with its neighboring queries
  - ➔ For each  $q_i$  in  $Q$ , a feasible allocation set of sites  $S_{q_i} = \{s_1, s_2, \dots, s_k\}$  where each site stores a copy of the relation in  $q_i$
- The objective is to find an optimal allocation of  $Q$  to  $S$  such that
  - ➔ the load unbalance of  $S$  is minimized
  - ➔ The total communication cost is minimized
- For each  $q$  in  $Q$  compute load ( $S_q$ )
- While  $Q$  not empty do
  - ➔ Select subquery  $a$  with least allocation flexibility
  - ➔ Select best site  $b$  for  $a$  (with least load and best benefit)

➔ Remove  $a$  from  $Q$  and recompute loads if needed

## 2-Step Algorithm Example

- Let  $Q = \{q_1, q_2, q_3, q_4\}$  where  $q_1$  is associated with  $R_1$ ,  $q_2$  is associated with  $R_2$  joined with the result of  $q_1$ , etc.
- Iteration 1: select  $q_4$ , allocate to  $s_1$ , set  $\text{load}(s_1)=2$
- Iteration 2: select  $q_2$ , allocate to  $s_2$ , set  $\text{load}(s_2)=3$
- Iteration 3: select  $q_3$ , allocate to  $s_1$ , set  $\text{load}(s_1)=3$
- Iteration 4: select  $q_1$ , allocate to  $s_3$  or  $s_4$

## Relational Algebra :

- The Relational Algebra is used to define the ways in which relations (tables) can be operated to manipulate their data.
- This Algebra is composed of Unary operations (involving a single table) and Binary operations (involving multiple tables).
- Join, Semi-join these are Binary operations in Relational Algebra.

### Join

- Join is a binary operation in Relational Algebra.
- It combines records from two or more tables in a database.
- A join is a means for combining fields from two tables by using values common to each.

### Semi-Join

- A Join where the result only contains the columns from one of the joined tables.
- Useful in distributed databases, so we don't have to send as much data over the network.
- Can dramatically speed up certain classes of queries.

*What is "Semi-Join" ?*

Semi-join strategies are technique for query processing in distributed database systems. Used for reducing communication cost.

A semi-join between two tables returns rows from the first table where one or more matches are found in the second table.

The difference between a semi-join and a conventional join is that rows in the first table will be returned at most once. Even if the second table contains two matches for a row in the first table, only one copy of the row will be returned.

Semi-joins are written using EXISTS or IN.

A Simple Semi-Join Example “Give a list of departments with at least one employee.” Query written with a conventional join:

```
SELECT D.deptno, D.dname FROM dept D, emp E WHERE E.deptno = D.deptno  
ORDER BY D.deptno;
```

- A department with N employees will appear in the list N times.
- We could use a DISTINCT keyword to get each department to appear only once.

A Simple Semi-Join Example “Give a list of departments with at least one employee.” Query written with a semi-join:

```
SELECT D.deptno, D.dname FROM dept D WHERE EXISTS (SELECT 1 FROM  
emp E WHERE E.deptno = D.deptno) ORDER BY D.deptno;
```

- No department appears more than once.
- Oracle stops processing each department as soon as the first employee in that department is found.

A **transaction** is a program including a collection of database operations, executed as a logical unit of data processing. The operations performed in a transaction include one or more of database operations like insert, delete, update or retrieve data.

- **read\_item()** – reads data item from storage to main memory.
- **modify\_item()** – change value of item in the main memory.
- **write\_item()** – write the modified value from main memory to storage.

### Transaction Operations

The low level operations performed in a transaction are –

- **begin\_transaction** – A marker that specifies start of transaction execution.
- **read\_item or write\_item** – Database operations that may be interleaved with main memory operations as a part of transaction.
- **end\_transaction** – A marker that specifies end of transaction.
- **commit** – A signal to specify that the transaction has been successfully completed in its entirety and will not be undone.

- **rollback** – A signal to specify that the transaction has been unsuccessful and so all temporary changes in the database are undone. A committed transaction cannot be rolled back.

#### Desirable Properties of Transactions

Any transaction must maintain the ACID properties, viz. Atomicity, Consistency, Isolation, and Durability.

- **Atomicity** – This property states that a transaction is an atomic unit of processing, that is, either it is performed in its entirety or not performed at all. No partial update should exist.
- **Consistency** – A transaction should take the database from one consistent state to another consistent state. It should not adversely affect any data item in the database.
- **Isolation** – A transaction should be executed as if it is the only one in the system. There should not be any interference from the other concurrent transactions that are simultaneously running.
- **Durability** – If a committed transaction brings about a change, that change should be durable in the database and not lost in case of any failure.

#### States of a transaction

**Active:** Initial state and during the execution

**Partially committed:** After the final statement has been executed

**Committed:** After successful completion

**Failed:** After the discovery that normal execution can no longer proceed

**Aborted:** After the transaction has been rolled back and the DB restored to its state prior to the start of the transaction. Restart it again or kill it.

#### Goal:

The **goal of transaction management** in a distributed database is to control the execution of **transactions** so that: 1. **Transactions** have atomicity, durability, serializability and isolation properties.

- CPU and main memory utilization
- Control messages
- Response time
- Availability

#### Distributed Transactions

A distributed transaction is a database transaction in which two or more network hosts are involved. Usually, hosts provide transactional resources, while the transaction manager is responsible for creating and managing a global transaction that encompasses all operations against such resources.

#### Supporting Atomicity of Distributed Transactions

**Logs:**

A log contains information for undoing or redoing all actions which are performed by transactions. The log record contains

- Identifier of the transaction
- Identifier of the record

**Type of action**(insert,delete, modify)

- Old record value
- New record value
- Auxiliary information for the recovery procedure

#### **Recovery procedures:**

When a failure occurs a recovery procedure reads the log file and performs the following operations,

- Determine all noncommitted transactions that have to be undone
- Determine all transactions which need to be redone.
- Undo the transactions determined at step 1 and redo the transactions determined at step 2.

#### Recovery of distributed transactions

Each site have a local transaction manager(LTM) which is capable of implementing local transactions.

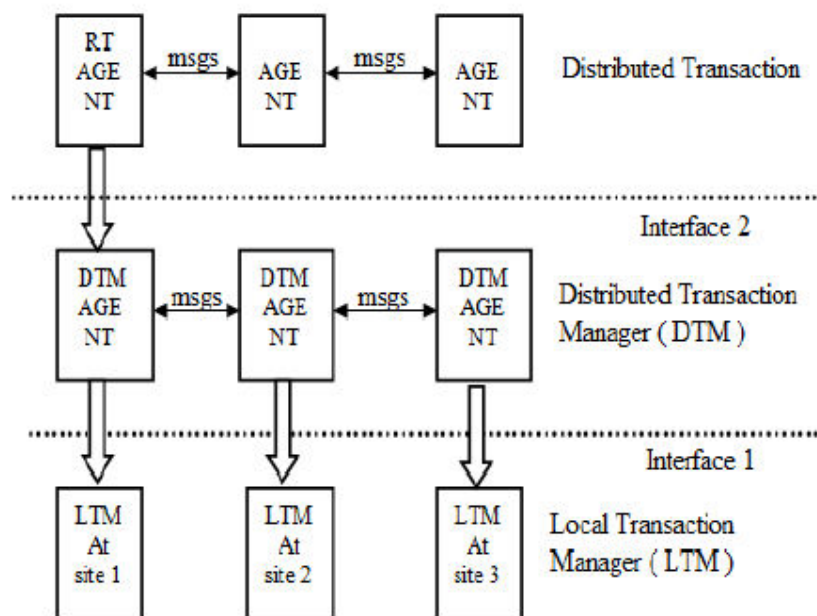


Figure 3.1 Reference Model of istributed transaction recovery

The relationship between distributed transaction management and local transaction management is represented in the reference model. At the bottom level we have the local transaction managers which do not need communication between them. The LTM's implement interface Local\_begin, Local\_commit, and local\_abort. At the next higher level we have the distributed transaction manager. DTM is by its nature a distributed level;DTM will be implemented by a set of local DTM agents which exchanges messages between them. DTM implements interface begin\_transasction , commit, abort, and create.

At the higher level we have the distributed transaction , constituted by the root agent and the other agents.

### **The 2-phase commit protocol**

Distributed two-phase commit reduces the vulnerability of one-phase commit protocols. The steps performed in the two phases are as follows –

#### **Phase 1: Prepare Phase**

- After each slave has locally completed its transaction, it sends a “DONE” message to the controlling site. When the controlling site has received “DONE” message from all slaves, it sends a “Prepare” message to the slaves.
- The slaves vote on whether they still want to commit or not. If a slave wants to commit, it sends a “Ready” message.
- A slave that does not want to commit sends a “Not Ready” message. This may happen when the slave has conflicting concurrent transactions or there is a timeout.

#### **Phase 2: Commit/Abort Phase**

- After the controlling site has received “Ready” message from all the slaves –
  - The controlling site sends a “Global Commit” message to the slaves.
  - The slaves apply the transaction and send a “Commit ACK” message to the controlling site.
  - When the controlling site receives “Commit ACK” message from all the slaves, it considers the transaction as committed.
- After the controlling site has received the first “Not Ready” message from any slave –
  - The controlling site sends a “Global Abort” message to the slaves.
  - The slaves abort the transaction and send a “Abort ACK” message to the controlling site.
  - When the controlling site receives “Abort ACK” message from all the slaves, it considers the transaction as aborted.

### **Concurrency control for distributed Transactions**

#### **Locking Based Concurrency Control Protocols**

Locking-based concurrency control protocols use the concept of locking data items. A **lock** is a variable associated with a data item that determines whether read/write



operations can be performed on that data item. Generally, a lock compatibility matrix is used which states whether a data item can be locked by two transactions at the same time.

Locking-based concurrency control systems can use either one-phase or two-phase locking protocols.

### One-phase Locking Protocol

In this method, each transaction locks an item before use and releases the lock as soon as it has finished using it. This locking method provides for maximum concurrency but does not always enforce serializability.

### Two-phase Locking Protocol

In this method, all locking operations precede the first lock-release or unlock operation. The transaction comprise of two phases. In the first phase, a transaction only acquires all the locks it needs and do not release any lock. This is called the **expanding** or the **growing phase**. In the second phase, the transaction releases the locks and cannot request any new locks. This is called the **shrinking phase**.

Every transaction that follows two-phase locking protocol is guaranteed to be serializable. However, this approach provides low parallelism between two conflicting transactions.

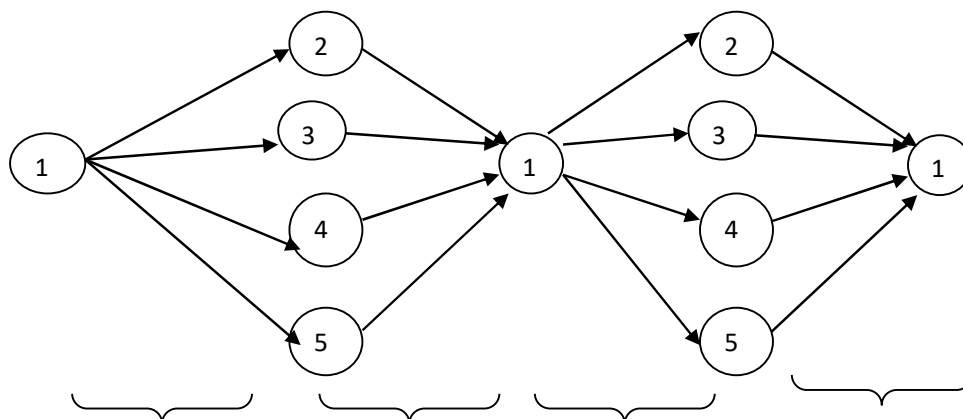
### Architectural Aspects of Distributed Transactions

- Structure of the computation
- Communication of a distributed transactions
- Sessions and datagrams:

The communications between processes or servers can be performed through sessions and datagrams. Sessions have a basic advantage: the authentication and identification functions need to be performed only once and then messages can be exchanged without repeating these operations.

### **Communication structure for commit protocols**

- **Centralized**



Prepare      Ready or Abort      Commit or Abort      ACK

Figure3. 2 **Centralized**

- **Hierarchial**

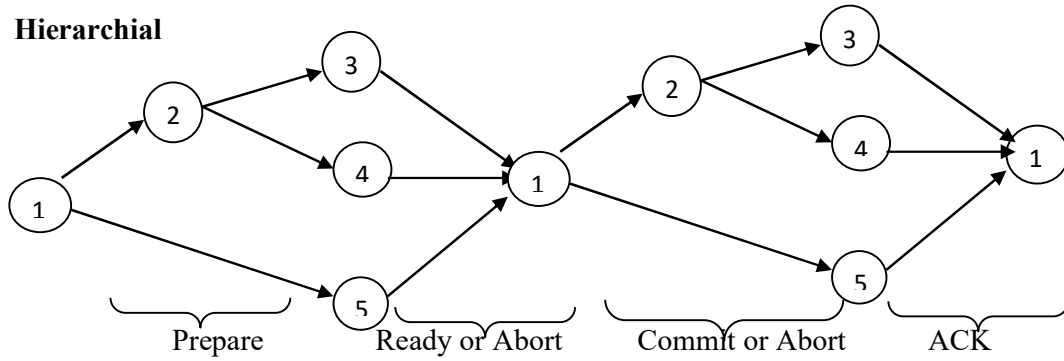


Figure 3.3 **Hierarchial**

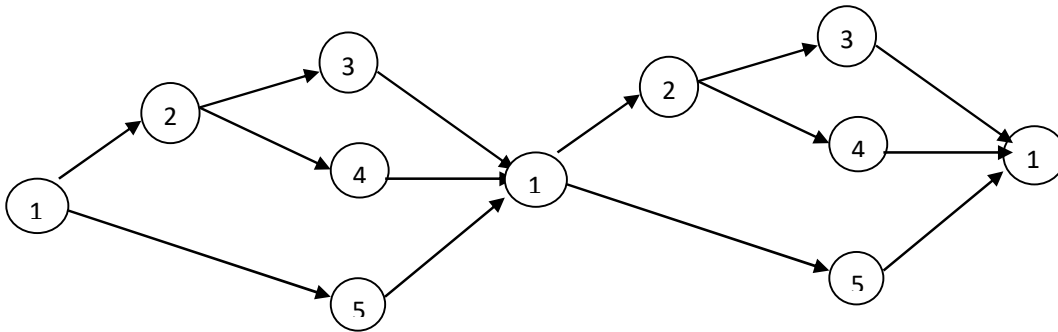


Figure 3.4 **Hierarchial**

- **Linear**

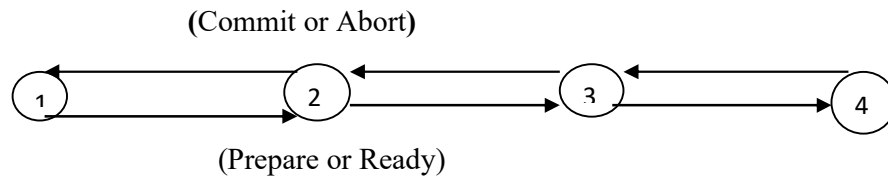


Figure 3.5 **Linear**

Ordering is defined

- **Distributed**



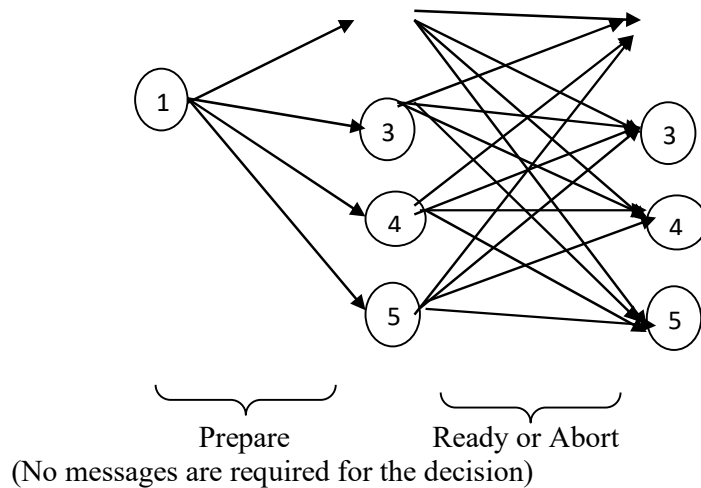


Figure 3.6 Distributed

### Concurrency Control

Concurrency controlling techniques ensure that multiple transactions are executed simultaneously while maintaining the ACID properties of the transactions and serializability in the schedules.

### Serializability in distributed database

In a system with a number of simultaneous transactions, a **schedule** is the total order of execution of operations. Given a schedule  $S$  comprising of  $n$  transactions, say  $T_1, T_2, T_3, \dots, T_n$ ; for any transaction  $T_i$ , the operations in  $T_i$  must execute as laid down in the schedule  $S$ .

### Types of Schedules

There are two types of schedules –

- **Serial Schedules** – In a serial schedule, at any point of time, only one transaction is active, i.e. there is no overlapping of transactions. This is depicted in the following graph –

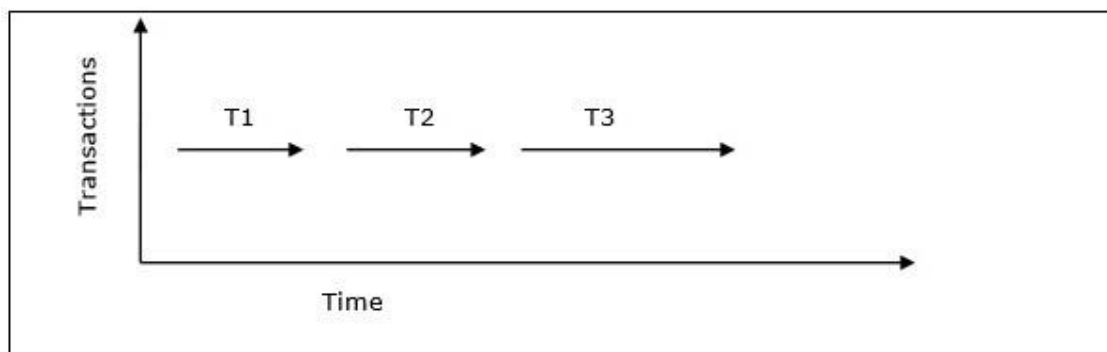


Figure 3.7 Serial Schedules

- **Parallel Schedules** – In parallel schedules, more than one transactions are active simultaneously, i.e. the transactions contain operations that overlap at time. This is depicted in the following graph –

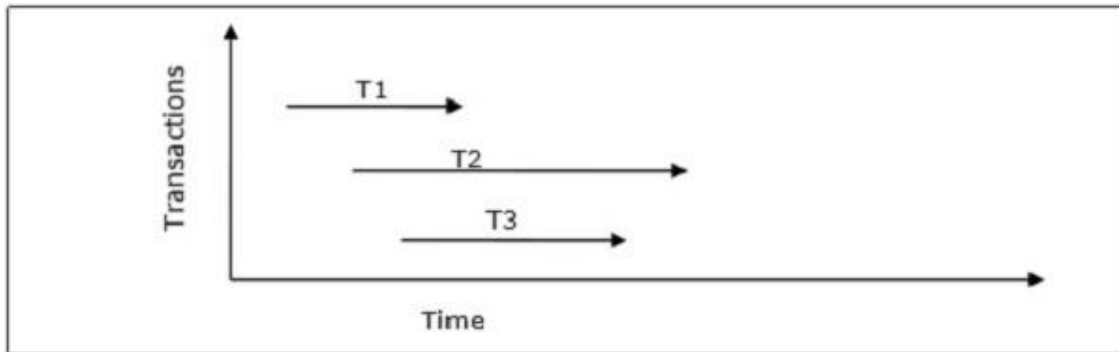


Figure 3.8 Parallel Schedules

### Conflicts in Schedules

In a schedule comprising of multiple transactions, a **conflict** occurs when two active transactions perform non-compatible operations. Two operations are said to be in conflict, when all of the following three conditions exists simultaneously –

- The two operations are parts of different transactions.
- Both the operations access the same data item.
- At least one of the operations is a write\_item() operation, i.e. it tries to modify the data item.

### Serializability

A **serializable schedule** of 'n' transactions is a parallel schedule which is equivalent to a serial schedule comprising of the same 'n' transactions. A serializable schedule contains the correctness of serial schedule while ascertaining better CPU utilization of parallel schedule.

### Equivalence of Schedules

Equivalence of two schedules can be of the following types –

- **Result equivalence** – Two schedules producing identical results are said to be result equivalent.
- **View equivalence** – Two schedules that perform similar action in a similar manner are said to be view equivalent.
- **Conflict equivalence** – Two schedules are said to be conflict equivalent if both contain the same set of transactions and has the same order of conflicting pairs of operations.

Serial schedules have less resource utilization and low throughput. To improve it, two or more transactions are run concurrently. But concurrency of transactions may lead to inconsistency in database. To avoid this, we need to check whether these concurrent schedules are serializable or not.

**Conflict Serializable:** A schedule is called conflict serializable if it can be transformed into a serial schedule by swapping non-conflicting operations.

**Conflicting operations:** Two operations are said to be conflicting if all conditions satisfy:

- They belong to different transaction
- They operation on same data item
- At Least one of them is a write operation

Example: –

- **Conflicting** operations pair  $(R_1(A), W_2(A))$  because they belong to two different transactions on same data item A and one of them is write operation.
- Similarly,  $(W_1(A), W_2(A))$  and  $(W_1(A), R_2(A))$  pairs are also **conflicting**.
- On the other hand,  $(R_1(A), W_2(B))$  pair is **non-conflicting** because they operate on different data item.
- Similarly,  $((W_1(A), W_2(B)))$  pair is **non-conflicting**.

Consider the following schedule:

S1:  $R_1(A), W_1(A), R_2(A), W_2(A), R_1(B), W_1(B), R_2(B), W_2(B)$

If  $O_i$  and  $O_j$  are two operations in a transaction and  $O_i < O_j$  ( $O_i$  is executed before  $O_j$ ), same order will follow in schedule as well. Using this property, we can get two transactions of schedule S1 as:

T1:  $R_1(A), W_1(A), R_1(B), W_1(B)$

T2:  $R_2(A), W_2(A), R_2(B), W_2(B)$

**Possible Serial Schedules are: T1->T2 or T2->T1**

-> **Swapping non-conflicting operations**  $R_2(A)$  and  $R_1(B)$  in S1, the schedule becomes,

**S11:**  $R_1(A), W_1(A), R_1(B), W_2(A), R_2(A), W_1(B), R_2(B), W_2(B)$

-> Similarly, **swapping non-conflicting operations**  $W_2(A)$  and  $W_1(B)$  in S11, the schedule becomes,

**S12:**  $R_1(A), W_1(A), R_1(B), W_1(B), R_2(A), W_2(A), R_2(B), W_2(B)$

S12 is a serial schedule in which all operations of T1 are performed before starting any operation of T2. Since S has been transformed into a serial schedule S12 by swapping non-conflicting operations of S1, S1 is conflict serializable.

Let us take another Schedule:

S2:  $R_2(A), W_2(A), R_1(A), W_1(A), R_1(B), W_1(B), R_2(B), W_2(B)$

Two transactions will be:

T1:  $R_1(A), W_1(A), R_1(B), W_1(B)$

T2:  $R_2(A), W_2(A), R_2(B), W_2(B)$

**Possible Serial Schedules are: T1->T2 or T2->T1**

Original Schedule is:

**S2:**  $R_2(A), W_2(A), R_1(A), W_1(A), R_1(B), W_1(B), R_2(B), W_2(B)$

Swapping non-conflicting operations  $R_1(A)$  and  $R_2(B)$  in S2, the schedule becomes,

**S21:**  $R_2(A), W_2(A), R_2(B), W_1(A), R_1(B), W_1(B), R_1(A), W_2(B)$

Similarly, swapping non-conflicting operations  $W_1(A)$  and  $W_2(B)$  in S21, the schedule becomes,

**S22:**  $R_2(A), W_2(A), R_2(B), W_2(B), R_1(B), W_1(B), R_1(A), W_1(A)$

In schedule S22, all operations of T2 are performed first, but operations of T1 are not in order (order should be  $R_1(A)$ ,  $W_1(A)$ ,  $R_1(B)$ ,  $W_1(B)$ ). So S2 is not conflict serializable.

**Conflict Equivalent:** Two schedules are said to be conflict equivalent when one can be transformed to another by swapping non-conflicting operations. In the example discussed above, S11 is conflict equivalent to S1 (S1 can be converted to S11 by swapping non-conflicting operations). Similarly, S11 is conflict equivalent to S12 and so on.

*Note 1:* Although S2 is not conflict serializable, but still it is conflict equivalent to S21 and S21 because S2 can be converted to S21 and S22 by swapping non-conflicting operations.

*Note 2:* The schedule which is conflict serializable is always conflict equivalent to one of the serial schedule. S1 schedule discussed above (which is conflict serializable) is equivalent to serial schedule ( $T1 \rightarrow T2$ ).

### Distributed deadlocks

**Distributed deadlocks** can occur in **distributed** systems when **distributed** transactions or concurrency control is being used. **Distributed deadlocks** can be detected either by constructing a global wait-for graph from local wait-for graphs at a **deadlock** detector or by a **distributed** algorithm like edge chasing.

Transaction processing in a distributed database system is also distributed, i.e. the same transaction may be processing at more than one site. The two main deadlock handling concerns in a distributed database system that are not present in a centralized system are **transaction location** and **transaction control**. Once these concerns are addressed, deadlocks are handled through any of deadlock prevention, deadlock avoidance or deadlock detection and removal.

### **Transaction Location**

Transactions in a distributed database system are processed in multiple sites and use data items in multiple sites. The amount of data processing is not uniformly distributed among these sites. The time period of processing also varies. Thus the same transaction may be active at some sites and inactive at others. When two conflicting transactions are located in a site, it may happen that one of them is in inactive state. This condition does not arise in a centralized system. This concern is called transaction location issue.

This concern may be addressed by Daisy Chain model. In this model, a transaction carries certain details when it moves from one site to another. Some of the details are the list of tables required, the list of sites required, the list of visited tables and sites, the list of tables and sites that are yet to be visited and the list of acquired locks with types. After a transaction terminates by either commit or abort, the information should be sent to all the concerned sites.

### **Transaction Control**

Transaction control is concerned with designating and controlling the sites required for processing a transaction in a distributed database system. There are many options regarding

the choice of where to process the transaction and how to designate the center of control, like –

- One server may be selected as the center of control.
- The center of control may travel from one server to another.
- The responsibility of controlling may be shared by a number of servers.

### **Distributed Deadlock Prevention**

Just like in centralized deadlock prevention, in distributed deadlock prevention approach, a transaction should acquire all the locks before starting to execute. This prevents deadlocks.

The site where the transaction enters is designated as the controlling site. The controlling site sends messages to the sites where the data items are located to lock the items. Then it waits for confirmation. When all the sites have confirmed that they have locked the data items, transaction starts. If any site or communication link fails, the transaction has to wait until they have been repaired.

Though the implementation is simple, this approach has some drawbacks –

- Pre-acquisition of locks requires a long time for communication delays. This increases the time required for transaction.
- In case of site or link failure, a transaction has to wait for a long time so that the sites recover. Meanwhile, in the running sites, the items are locked. This may prevent other transactions from executing.
- If the controlling site fails, it cannot communicate with the other sites. These sites continue to keep the locked data items in their locked state, thus resulting in blocking.

### **Distributed Deadlock Avoidance**

As in centralized system, distributed deadlock avoidance handles deadlock prior to occurrence. Additionally, in distributed systems, transaction location and transaction control issues needs to be addressed. Due to the distributed nature of the transaction, the following conflicts may occur –

- Conflict between two transactions in the same site.
- Conflict between two transactions in different sites.

In case of conflict, one of the transactions may be aborted or allowed to wait as per distributed wait-die or distributed wound-wait algorithms.

Let us assume that there are two transactions, T1 and T2. T1 arrives at Site P and tries to lock a data item which is already locked by T2 at that site. Hence, there is a conflict at Site P. The algorithms are as follows –

- **Distributed Wound-Die**

- If T1 is older than T2, T1 is allowed to wait. T1 can resume execution after Site P receives a message that T2 has either committed or aborted successfully at all sites.
- If T1 is younger than T2, T1 is aborted. The concurrency control at Site P sends a message to all sites where T1 has visited to abort T1. The controlling site notifies the user when T1 has been successfully aborted in all the sites.

- **Distributed Wait-Wait**

- If T1 is older than T2, T2 needs to be aborted. If T2 is active at Site P, Site P aborts and rolls back T2 and then broadcasts this message to other relevant sites. If T2 has left Site P but is active at Site Q, Site P broadcasts that T2 has been aborted; Site L then aborts and rolls back T2 and sends this message to all sites.
- If T1 is younger than T1, T1 is allowed to wait. T1 can resume execution after Site P receives a message that T2 has completed processing.

### **Distributed Deadlock Detection**

Just like centralized deadlock detection approach, deadlocks are allowed to occur and are removed if detected. The system does not perform any checks when a transaction places a lock request. For implementation, global wait-for-graphs are created. Existence of a cycle in the global wait-for-graph indicates deadlocks. However, it is difficult to spot deadlocks since transaction waits for resources across the network.

Alternatively, deadlock detection algorithms can use timers. Each transaction is associated with a timer which is set to a time period in which a transaction is expected to finish. If a transaction does not finish within this time period, the timer goes off, indicating a possible deadlock.

Another tool used for deadlock handling is a deadlock detector. In a centralized system, there is one deadlock detector. In a distributed system, there can be more than one deadlock detectors. A deadlock detector can find deadlocks for the sites under its control. There are three alternatives for deadlock detection in a distributed system, namely.

- **Centralized Deadlock Detector** – One site is designated as the central deadlock detector.



- **Hierarchical Deadlock Detector** – A number of deadlock detectors are arranged in hierarchy.
- **Distributed Deadlock Detector** – All the sites participate in detecting deadlocks and removing them.

### Time and time stamps in a distributed database

**Timestamp** is a unique identifier created by the DBMS to identify a transaction. They are usually assigned in the order in which they are submitted to the system. Refer to the timestamp of a transaction  $T$  as  $TS(T)$ . For basics of Timestamp you may refer here.

#### **Timestamp Ordering Protocol –**

The main idea for this protocol is to order the transactions based on their Timestamps. A schedule in which the transactions participate is then serializable and the only *equivalent serial schedule permitted* has the transactions in the order of their Timestamp Values. Stating simply, the schedule is equivalent to the particular *Serial Order* corresponding to the *order of the Transaction timestamps*. Algorithm must ensure that, for each items accessed by *Conflicting Operations* in the schedule, the order in which the item is accessed does not violate the ordering. To ensure this, use two Timestamp Values relating to each database item  $X$ .

- $W\_TS(X)$  is the largest timestamp of any transaction that executed **write(X)** successfully.
- $R\_TS(X)$  is the largest timestamp of any transaction that executed **read(X)** successfully.

#### **Timestamp Concurrency Control Algorithms**

Timestamp-based concurrency control algorithms use a transaction's timestamp to coordinate concurrent access to a data item to ensure serializability. A timestamp is a unique identifier given by DBMS to a transaction that represents the transaction's start time.

These algorithms ensure that transactions commit in the order dictated by their timestamps. An older transaction should commit before a younger transaction, since the older transaction enters the system before the younger one.

Timestamp-based concurrency control techniques generate serializable schedules such that the equivalent serial schedule is arranged in order of the age of the participating transactions.

Some of timestamp based concurrency control algorithms are –

- Basic timestamp ordering algorithm.
- Conservative timestamp ordering algorithm.
- Multiversion algorithm based upon timestamp ordering.

Timestamp based ordering follow three rules to enforce serializability –

- **Access Rule** – When two transactions try to access the same data item simultaneously, for conflicting operations, priority is given to the older transaction. This causes the younger transaction to wait for the older transaction to commit first.
- **Late Transaction Rule** – If a younger transaction has written a data item, then an older transaction is not allowed to read or write that data item. This rule prevents the older transaction from committing after the younger transaction has already committed.
- **Younger Transaction Rule** – A younger transaction can read or write a data item that has already been written by an older transaction.

### Basic Timestamp Ordering –

Every transaction is issued a timestamp based on when it enters the system. Suppose, if an old transaction  $T_i$  has timestamp  $TS(T_i)$ , a new transaction  $T_j$  is assigned timestamp  $TS(T_j)$  such that  $TS(T_i) < TS(T_j)$ . The protocol manages concurrent execution such that the timestamps determine the serializability order. The timestamp ordering protocol ensures that any conflicting read and write operations are executed in timestamp order. Whenever some Transaction  $T$  tries to issue a  $R\_item(X)$  or a  $W\_item(X)$ , the Basic TO algorithm compares the timestamp of  $T$  with  $R\_TS(X)$  &  $W\_TS(X)$  to ensure that the Timestamp order is not violated. This describes the Basic TO protocol in following two cases.

1. Whenever a Transaction  $T$  issues a **W\_item(X)** operation, check the following conditions:
  1. If  $R\_TS(X) > TS(T)$  or if  $W\_TS(X) > TS(T)$ , then abort and rollback  $T$  and reject the operation. else,
    - Execute  $W\_item(X)$  operation of  $T$  and set  $W\_TS(X)$  to  $TS(T)$ .
2. Whenever a Transaction  $T$  issues a **R\_item(X)** operation, check the following conditions:
  - If  $W\_TS(X) > TS(T)$ , then abort and reject  $T$  and reject the operation, else
  - If  $W\_TS(X) \leq TS(T)$ , then execute the  $R\_item(X)$  operation of  $T$  and set  $R\_TS(X)$  to the larger of  $TS(T)$  and current  $R\_TS(X)$ .

Whenever the Basic TO algorithm detects two conflicting operations that occur in incorrect order, it rejects the later of the two operations by aborting the Transaction that issued it. Schedules produced by Basic TO are guaranteed to be *conflict serializable*. Already discussed that using Timestamp, can ensure that our schedule will be *deadlock free*.

One drawback of Basic TO protocol is that it **Cascading Rollback** is still possible. Suppose we have a Transaction  $T_1$  and  $T_2$  has used a value written by  $T_1$ . If  $T_1$  is aborted and resubmitted to the system then,  $T_2$  must also be aborted and rolled back. So the problem of Cascading aborts still prevails.

Let's give the Advantages and Disadvantages of Basic TO protocol:

- Timestamp Ordering protocol ensures serializability
- Timestamp protocol ensures freedom from deadlock as no transaction ever waits.
- But the schedule may *not be cascade free*, and may not even be recoverable.

Optimistic Concurrency Control Algorithm

In systems with low conflict rates, the task of validating every transaction for serializability may lower performance. In these cases, the test for serializability is postponed to just before commit. Since the conflict rate is low, the probability of aborting transactions which are not serializable is also low. This approach is called optimistic concurrency control technique.

In this approach, a transaction's life cycle is divided into the following three phases –

- **Execution Phase** – A transaction fetches data items to memory and performs operations upon them.
- **Validation Phase** – A transaction performs checks to ensure that committing its changes to the database passes serializability test.
- **Commit Phase** – A transaction writes back modified data item in memory to the disk.

This algorithm uses three rules to enforce serializability in validation phase –

**Rule 1** – Given two transactions  $T_i$  and  $T_j$ , if  $T_i$  is reading the data item which  $T_j$  is writing, then  $T_i$ 's execution phase cannot overlap with  $T_j$ 's commit phase.  $T_j$  can commit only after  $T_i$  has finished execution.

**Rule 2** – Given two transactions  $T_i$  and  $T_j$ , if  $T_i$  is writing the data item that  $T_j$  is reading, then  $T_i$ 's commit phase cannot overlap with  $T_j$ 's execution phase.  $T_j$  can start executing only after  $T_i$  has already committed.

**Rule 3** – Given two transactions  $T_i$  and  $T_j$ , if  $T_i$  is writing the data item which  $T_j$  is also writing, then  $T_i$ 's commit phase cannot overlap with  $T_j$ 's commit phase.  $T_j$  can start to commit only after  $T_i$  has already committed.