SATHYABAMA
INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)
Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE
www.sathyabama.ac.in

SCHOOL OF COMPUTING

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

UNIT – I – DISTRIBUTED DATABASE – SCS1613

UNIT - I

INTRODUCTION TO DISTRIBUTED DATABASE

Introduction of Distributed Databases-Features of Distributed Databases-Distributed databases versus Centralized Databases- Principles—Levels Of Distribution-Transparency-Reference Architecture- Types of Data Fragmentation- Integrity Constraints in Distributed Databases- Architectural Issues- Alternative Client/Server Architecture.

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 ANSISPARC 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.

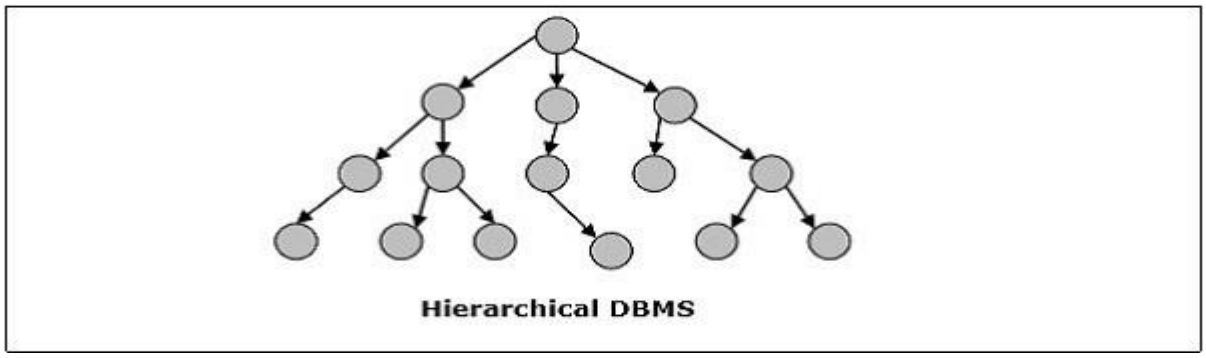


Figure 1.1 Hierarchical DBMS

Network DBMS

Network DBMS in 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.

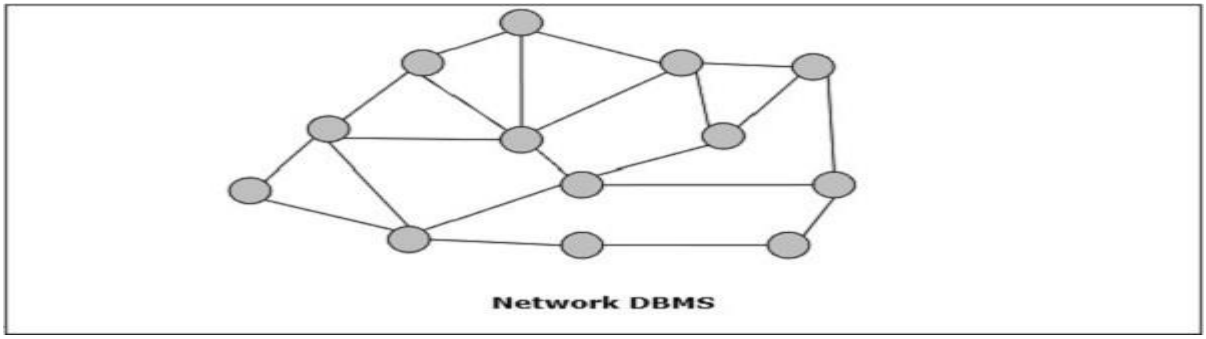
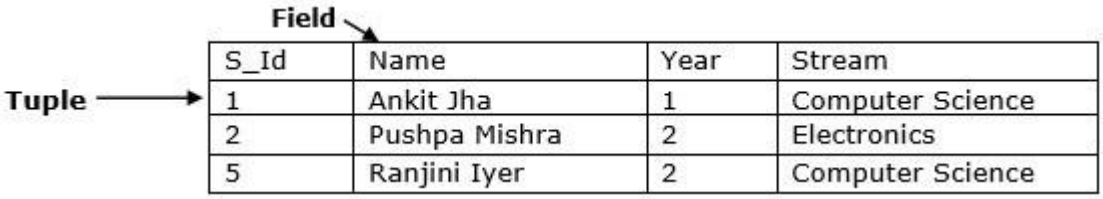


Figure 1.2 Network DBMS

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

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

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

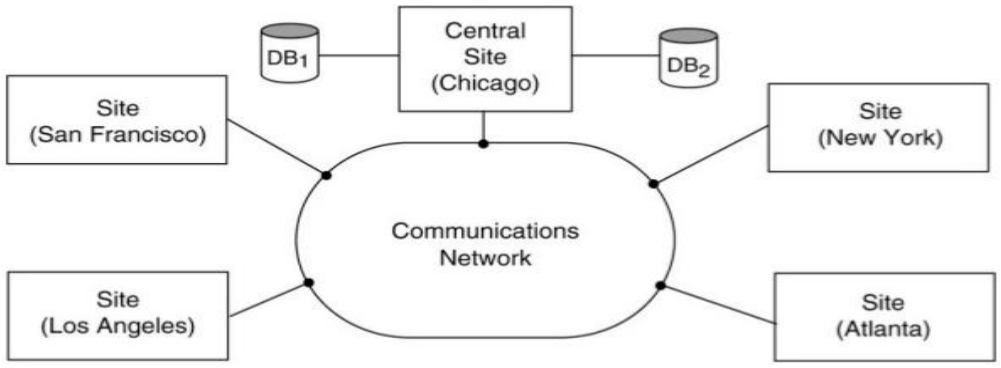


Figure 1.5 Centralized database

Distributed database

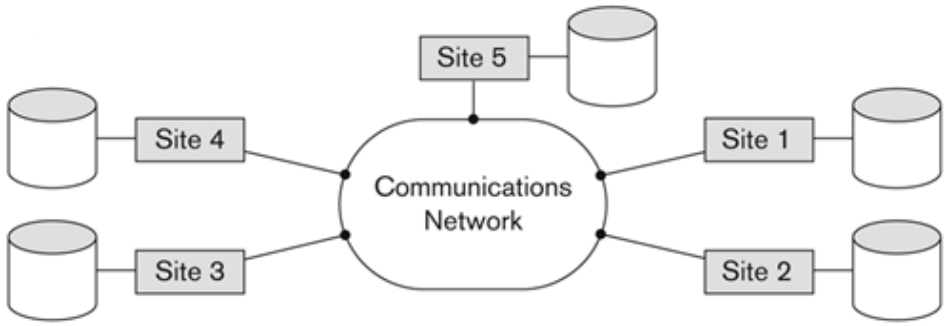


Figure1. 6 Distributed database

Types of Distributed Databases

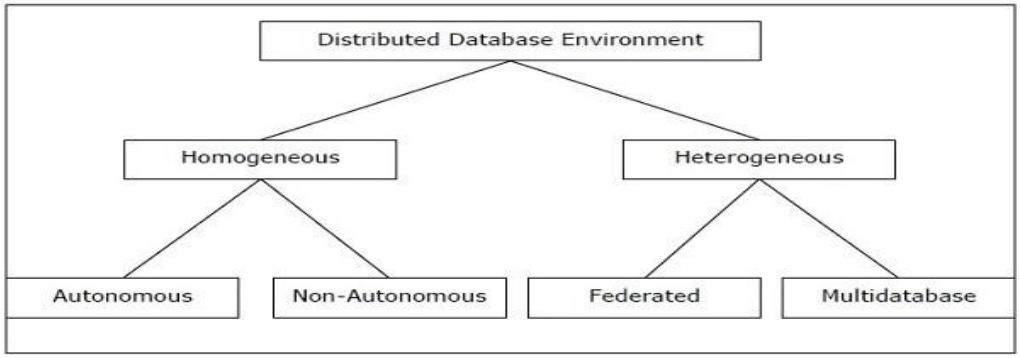


Figure 1.7 Types of Distributed Databases

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.

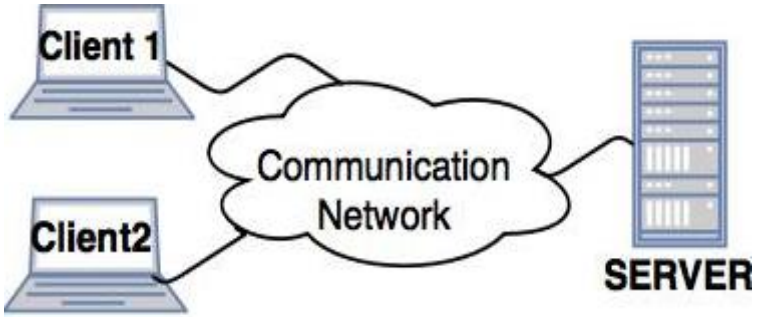


Figure1. 8 Single Server Multiple Client

Multiple Server Multiple Client

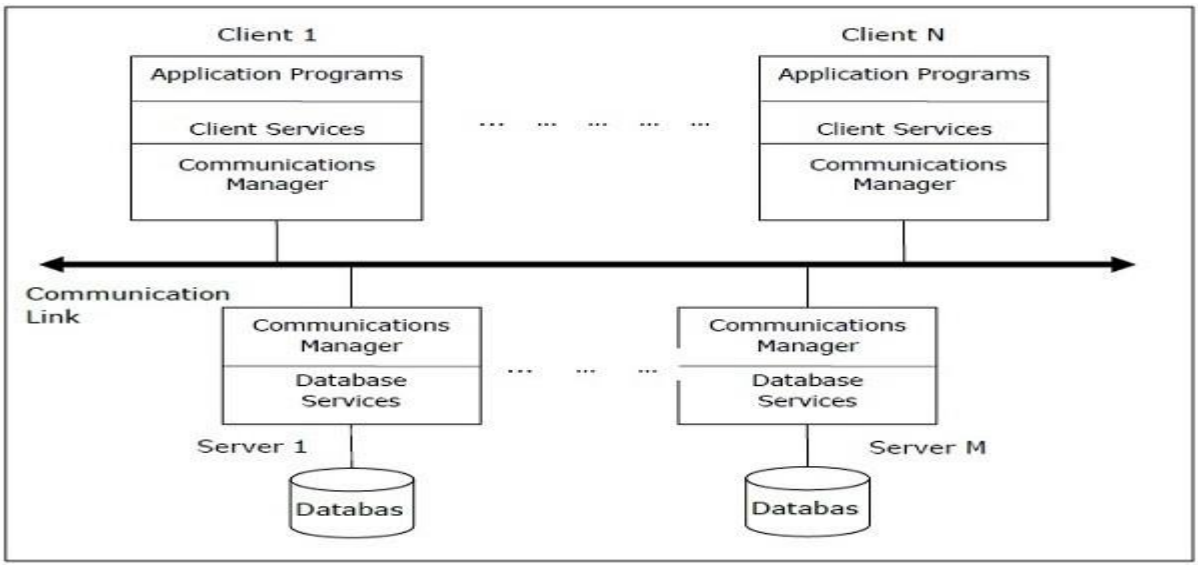


Figure 1. 9 Multiple Servers accessed by multiple clients

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

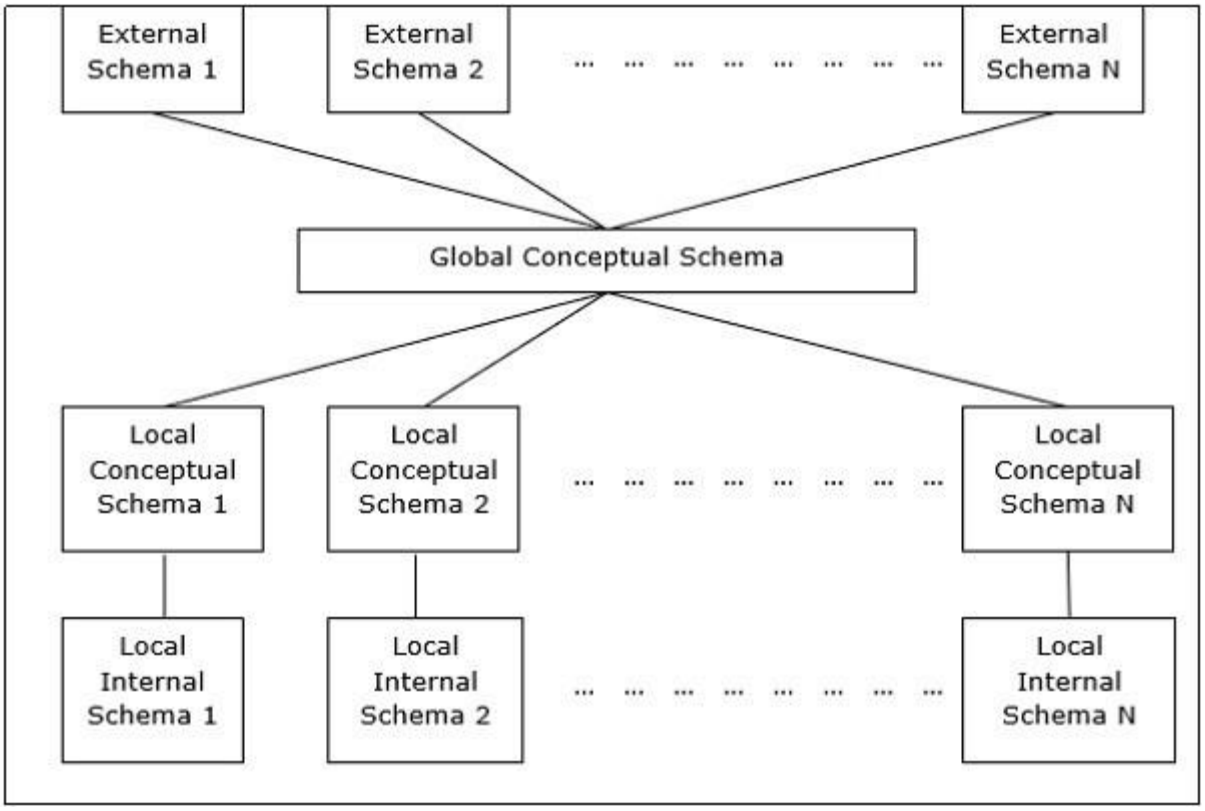


Figure 1.10 Frame Work

- 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

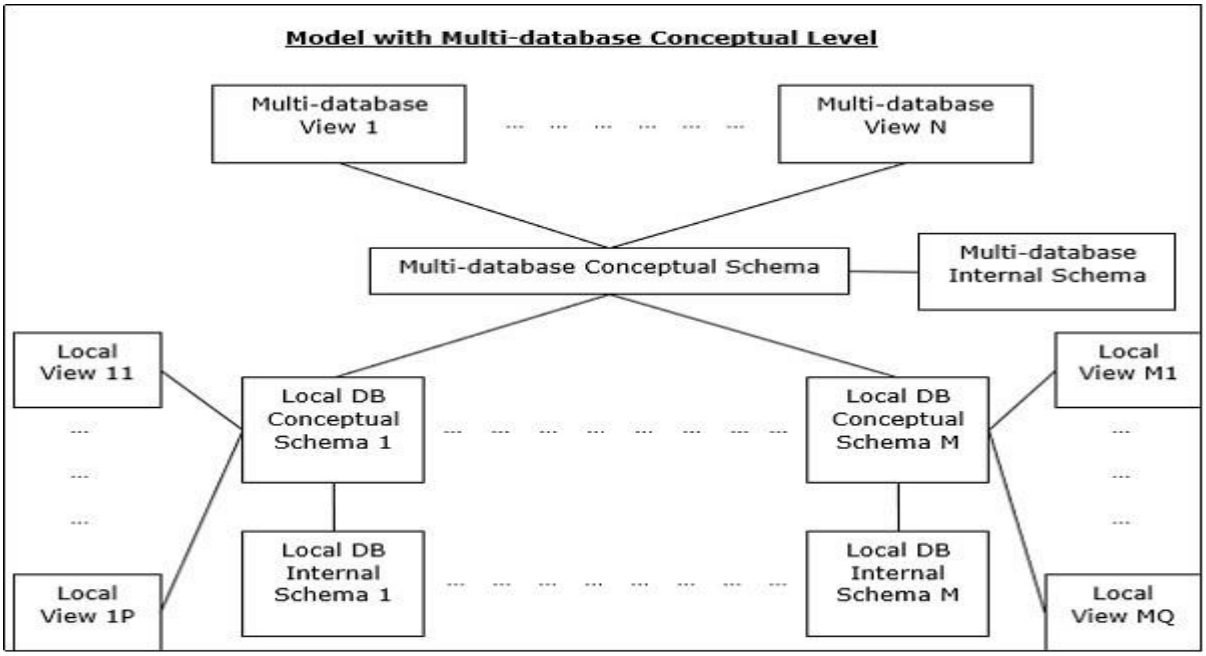


Figure 1.11 Models Using a Global Conceptual Schema

- 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 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

$R_i = \sigma_{Fi}(R), \quad 1 <= i <= w$

Fi selection formula used to obtain fragment R_i

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

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

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

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

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

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

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);
```

SATHYABAMA
INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)
Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE
www.sathyabama.ac.in

SCHOOL OF COMPUTING

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

UNIT – II – DISTRIBUTED DATABASE – SCS1613

UNIT - II

QUERIES AND OPTIMAZATION

Global Queries to Fragment Queries-Equivalence Transformations for Queries-Distributed Grouping and Aggregate Function Evaluation-Parametric Queries-Optimization of Access Strategies-Framework for Query Optimization-Join Queries- General Queries-Introduction to Distributed Transactions.

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 −

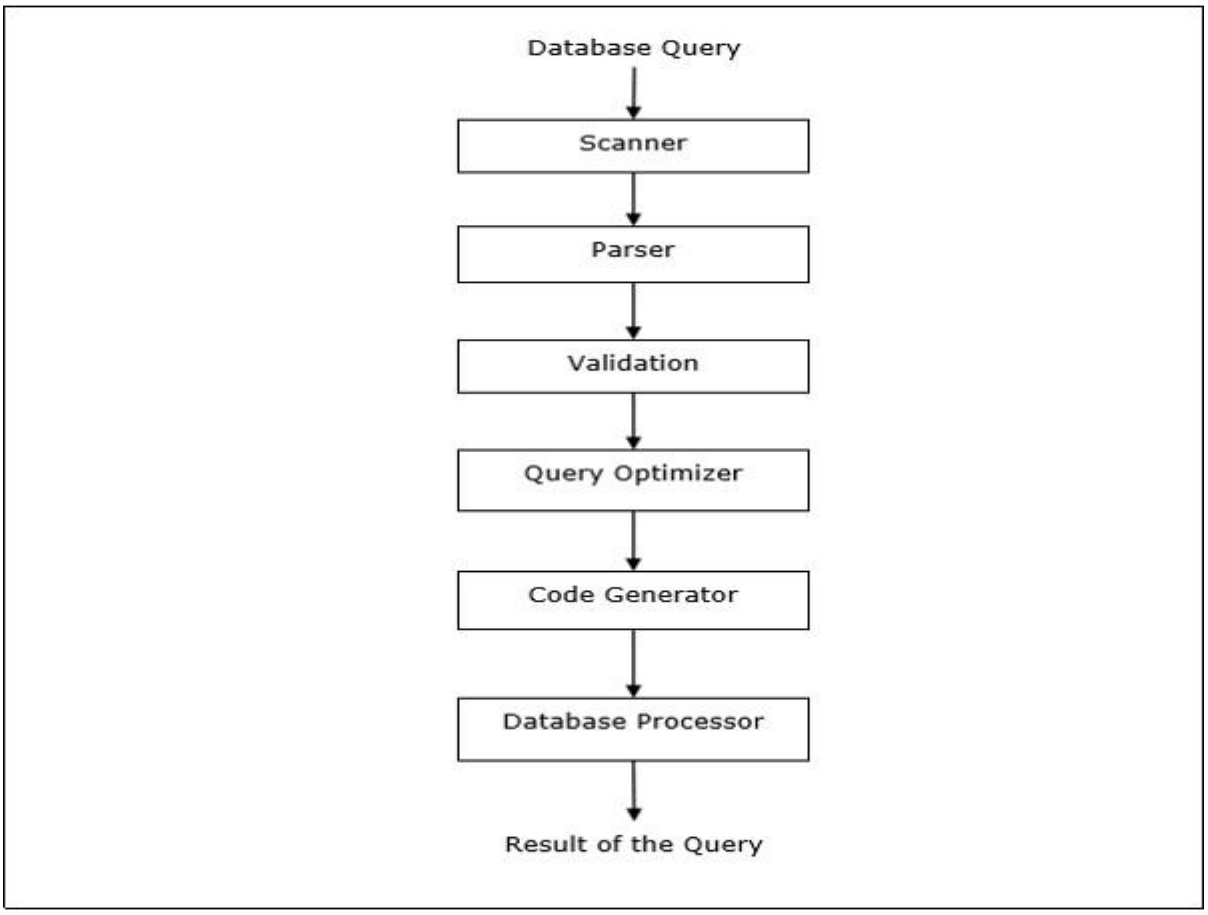


Figure 2.1 step in query processing

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

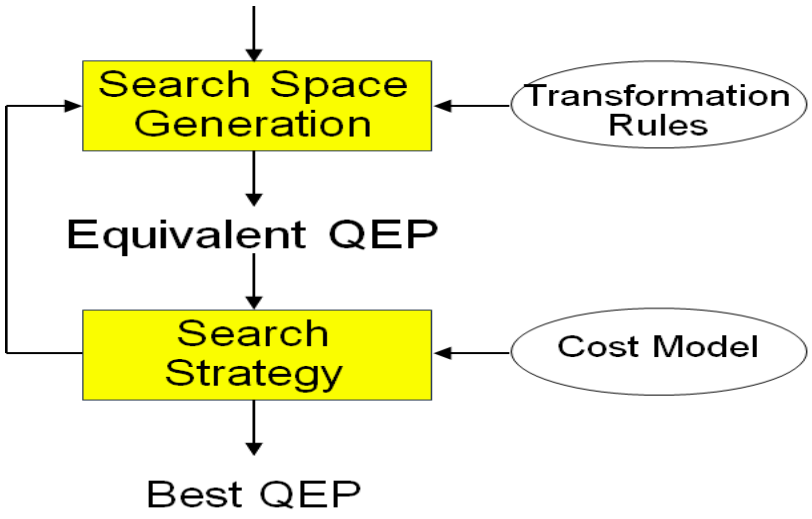


Figure 2.2 Query Optimization Process

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 $S = \{s_1, s_2, \ldots, s_n\}$ 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 $S_q = \{s_1, s_2, \ldots, s_k\}$ 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 load(s_1)=2
- Iteration 2: select q_2, allocate to s_2, set load(s_2)=3
- Iteration 3: select q_3, allocate to s_1, set 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.

SATHYABAMA

INSTITUTE OF SCIENCE AND TECHNOLOGY

(DEEMED TO BE UNIVERSITY)

Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE

www.sathyabama.ac.in

SCHOOL OF COMPUTING

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

UNIT – III – DISTRIBUTED DATABASE – SCS1613

Unit III

MANAGEMENT OF DISTRIBUTED TRANSACTIONS

Management of Distributed Transactions- Framework for Transaction Management- Supporting Atomicity of Distributed Transactions- Concurrency Control for Distributed Transactions- Architectural Aspects of Distributed Transactions-Concurrency Control- Foundation of Distributed Concurrency Control- Distributed Deadlocks-Concurrency Control based on Timestamps- Optimistic Methods for Distributed Concurrency Control

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 alocal transaction manager(LTM) which is capable of implementing local transactions.

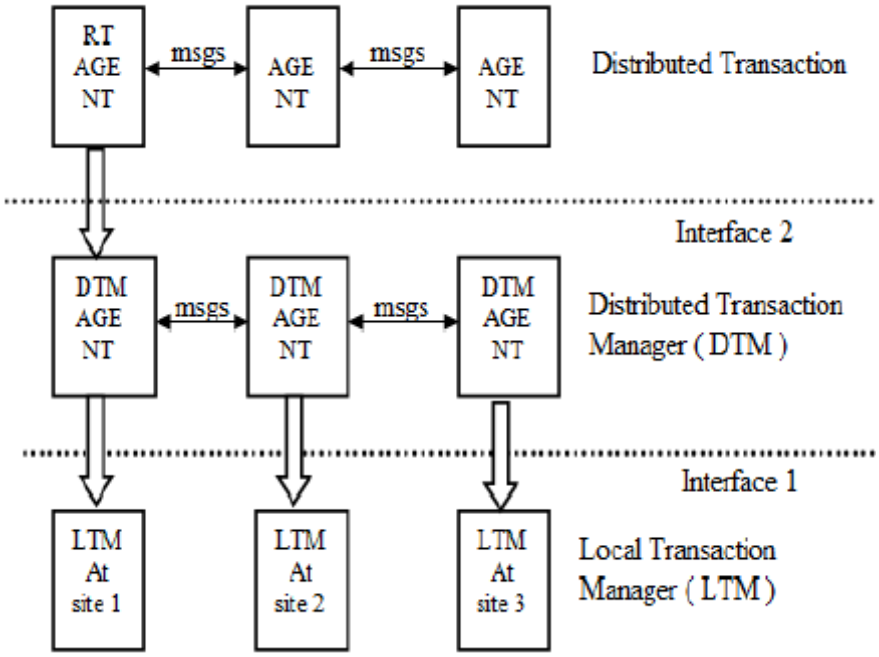


Figure 3.1 Reference Model of istributed transaction recovery

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 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.

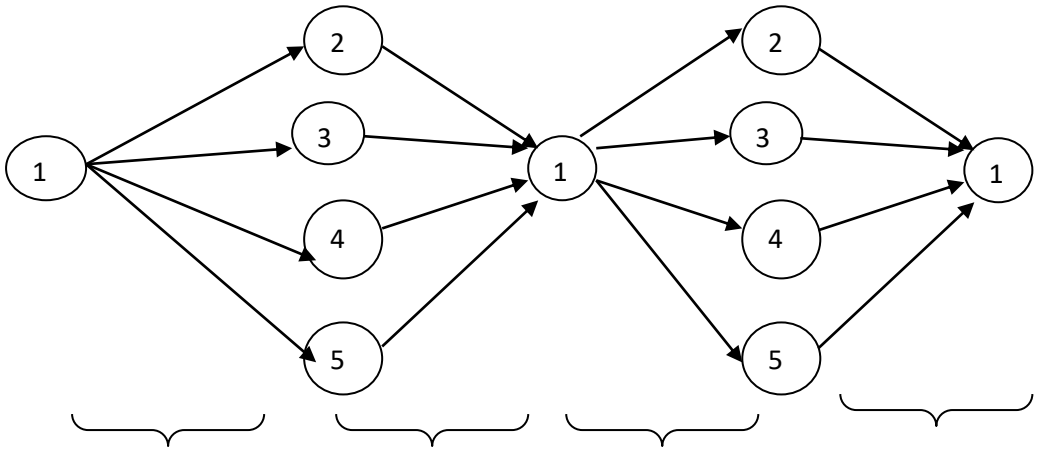
Architectural Aspects of Distributed Transactions

- 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 oerformed 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**

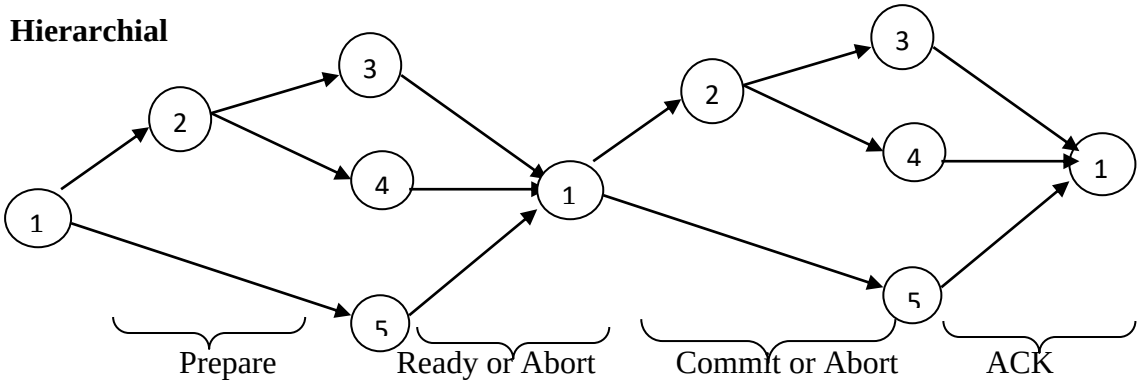


Figure 3.3 **Hierarchial**

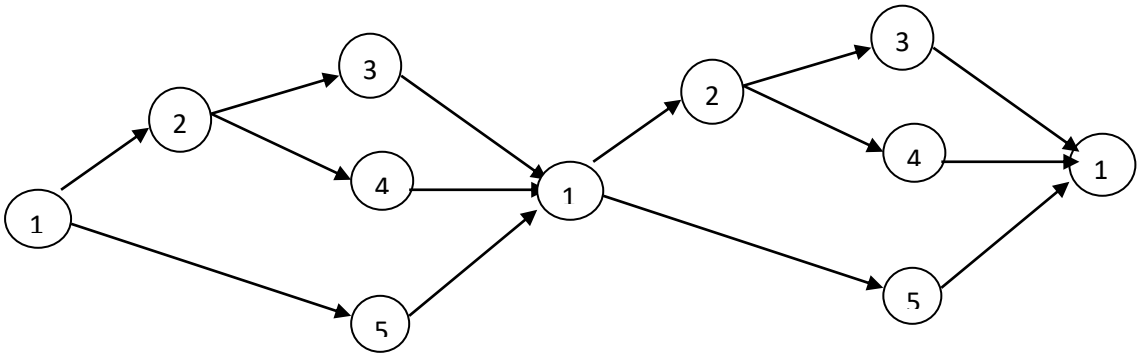
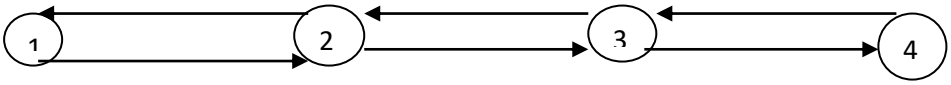


Figure 3.4 **Hierarchial**

- **Linear**

(Commit or Abort**)**



(Prepare or Ready)

Figure 3.5 **Linear**

Ordering is defined

- Distributed

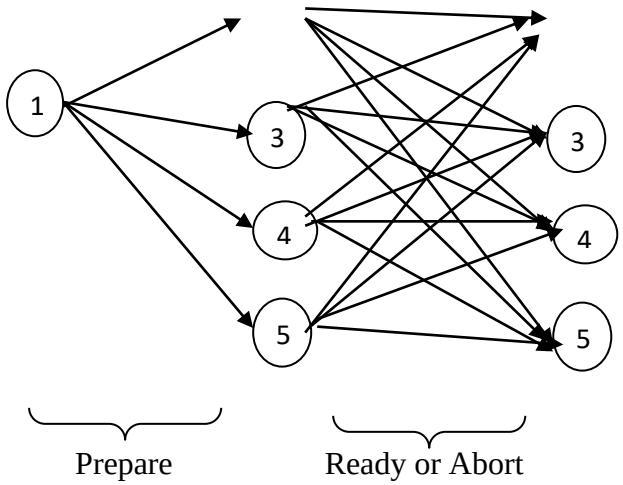


Figure 3.6 **Distributed**

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 T1, T2, T3……..Tn; for any transaction Ti, the operations in Ti 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 −

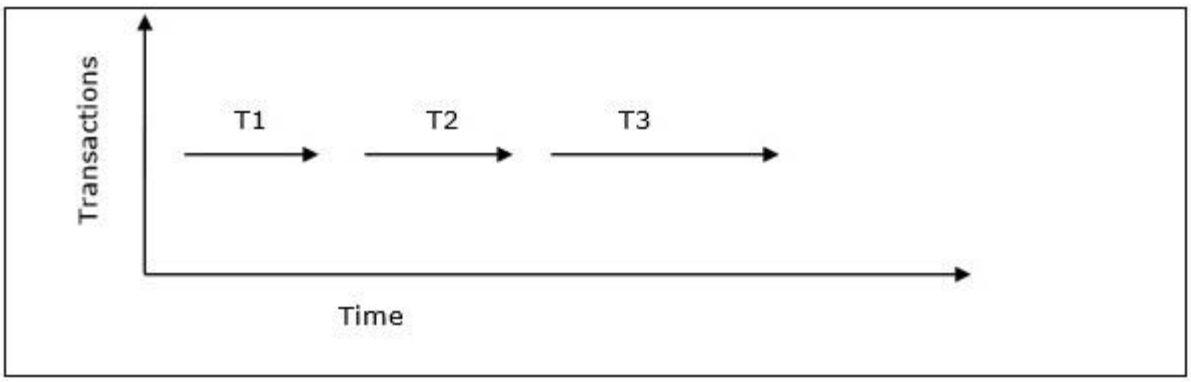


Figure 3.7 Serial Schedules

- **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 −

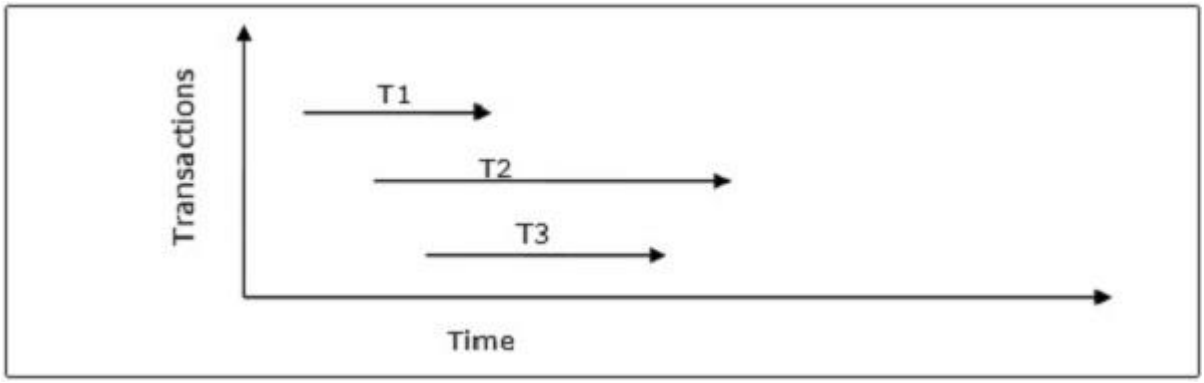


Figure 3.8 Parallel Schedules

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 are 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 operation**s $R_2(A)$ 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, s**wapping 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->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 describe 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) <= 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 twp conflicting operation that occur in incorrect order, it rejects the later of the two operation 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 must also be aborted and rolled back. So the problem of Cascading aborts still prevails.
Let's gist the Advantages and Disadvantages of Basic TO protocol:

- Timestamp Ordering protocol ensures serializablity
- 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.

SATHYABAMA

INSTITUTE OF SCIENCE AND TECHNOLOGY

(DEEMED TO BE UNIVERSITY)

Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE

www.sathyabama.ac.in

SCHOOL OF COMPUTING

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

UNIT – IV – DISTRIBUTED DATABASE – SCS1613

UNIT IV

RELIABILITY AND PROTECTION

Reliability- Basic Concepts- Reliability and concurrency Control- Determining a Consistent View of the NetworkDetection and Resolution of Inconsistency- Checkpoints and Cold Restart- Distributed Database AdministrationCatalog Management in Distributed Databases-Authorization and Protection

RELIABILITY

Reliability is defined as a measure of the success with which the system conforms to some authoritative specification of its behavior. When the behavior deviates from that which is specified for it, this is called as **Failure**. The reliability of the system is inversely related to the frequency of failures.

The reliability of a system can be measured in several ways, which are based on the incidence of failures. Measures include Mean Time Between Failure(MTBF), Mean Time To Repair(MTTR), and availability, defined as the fraction of the time that the system meets its specification. MTBF is the amount of time between system failures in a network. MTTR is the amount of time system takes to repair the failed systems.

BASIC CONCEPTS

In a database system application, the highest level specification is application-dependent. It is convenient to split the reliability problem into two separate parts, an application-dependent part and an application-independent part.

We emphasize two aspects of reliability: correctness and availability. It is important not only that a system behave correctly, i.e., in accordance with the specification, but also that it be available when necessary.

In some applications, like banking applications, correctness is an absolute requriment, and errors which may corrupt the consistency of the database cannot be tolerated. Other applications may tolerate the risk of inconsistencies in order to achieve a greater availability.

When a communication network fails the following problems may arise,

1. Commitment of transactions

2. Multiple copies of data and robusteness of concurrency control
3. Determining the state of the network
4. Detection and resolution of inconsistencies.
5. Checkpoints and cold restart.

NONBLOCKING COMMITMENT PROTOCOLS

A Commitment protocol is called **blocking**if the occurrence of some kinds of failure forces some of the participating sites to wait until the failure is repaired before terminating the transaction. A transaction that cannot be terminated at a site is called **pending** at this site. State diagrams are used for describing the evolution of the coordinator and participants during the execution of a protocol.

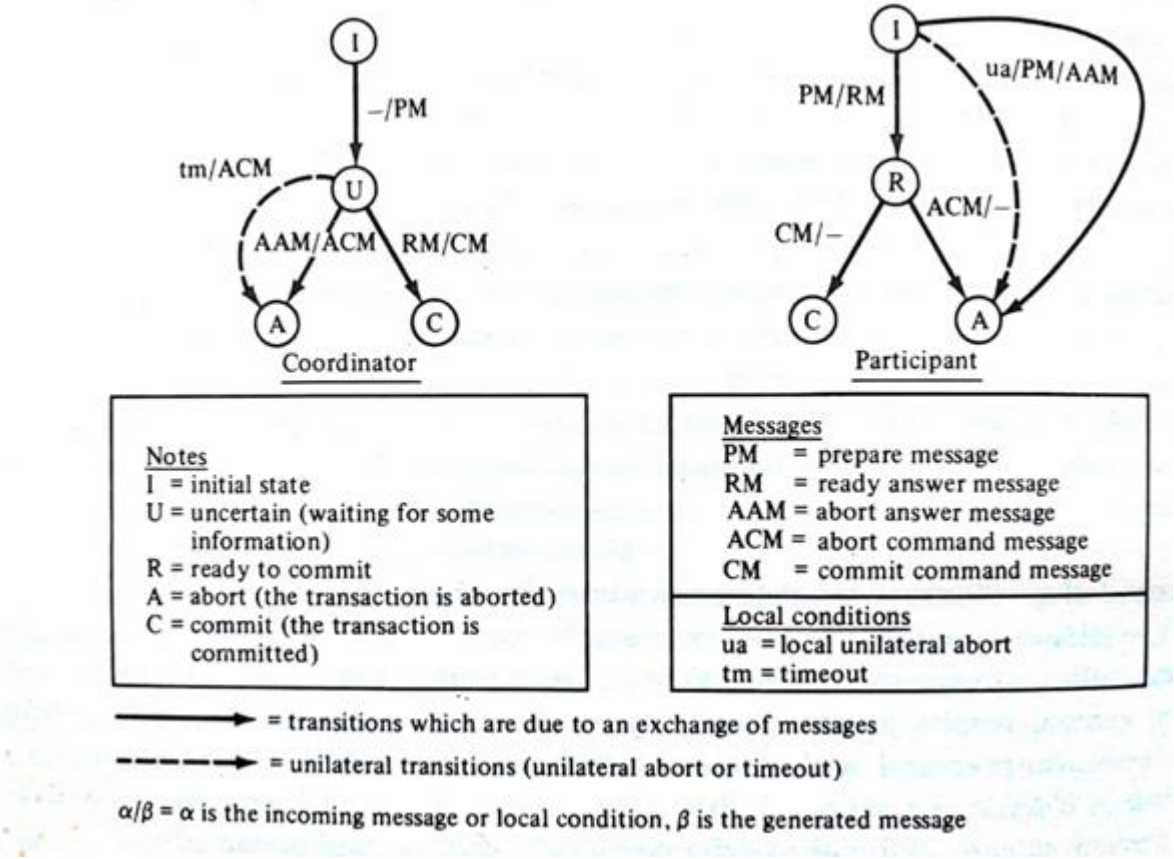


Figure 4.1 two state diagrams for the 2-phase-commitment protocol

The above figure shows the two state diagrams for the 2-phase-commitment protocol without the ACK messages. For each transition, an input message and an output message are indicated. A transition occurs when an input message arrives and causes the output message to be sent.

State information must be recorded into stable storage for recovery purposes. This helps in writing appropriate records in the logs.

Consider a transition from state X to state Y with input I and output O. The following behavior is assumed:

1. The input message I is received.

2. The new state Y is recorded on stable storage.
3. The output message O is sent.

If the site fails between the first and the second event, the state remains X, and the input message is lost. If the site fails between the second and third event, then the site reaches state Y, but the output message is not sent.

1. NONBLOCKING COMMITMENT PROTOCLS WITH SITE FAILURE

The termination protocol for the 2-phase-commitment protocol must allow the transactions to be terminated at all operational sites when a failure of the coordinator site occurs. This is possible in the following two cases:

1. At least one of the participants has received the command. In this case, the other participants can be told by this participant of the outcome of the transactions and can terminate it.
2. None of the participants has received the command, and only the coordinator site has crashed, so that all participants are operational. In this case, the participants can elect a new coordinator and resume the protocol.

In above cases, the transactions can be correctly terminated at all operational sites. Termination is impossible when no operational participants has received the command and at least one participant failed, because the operational participants cannot know the failed participant has done and cannot take an independent decision. So, if a coordinator fails termination is impossible.

This problem can be eliminated by modifying the 2-phase-commitment protocol in the 3-phase-commitment protocol.

The 3-phase-commitment protocol

In this protocol, the participants do not directly commit the transactions during the second phase of commitment, instead they reach in this phase a new prepared-to-commit(PC) state. So an additional third phase is required for actually committing the transactions.

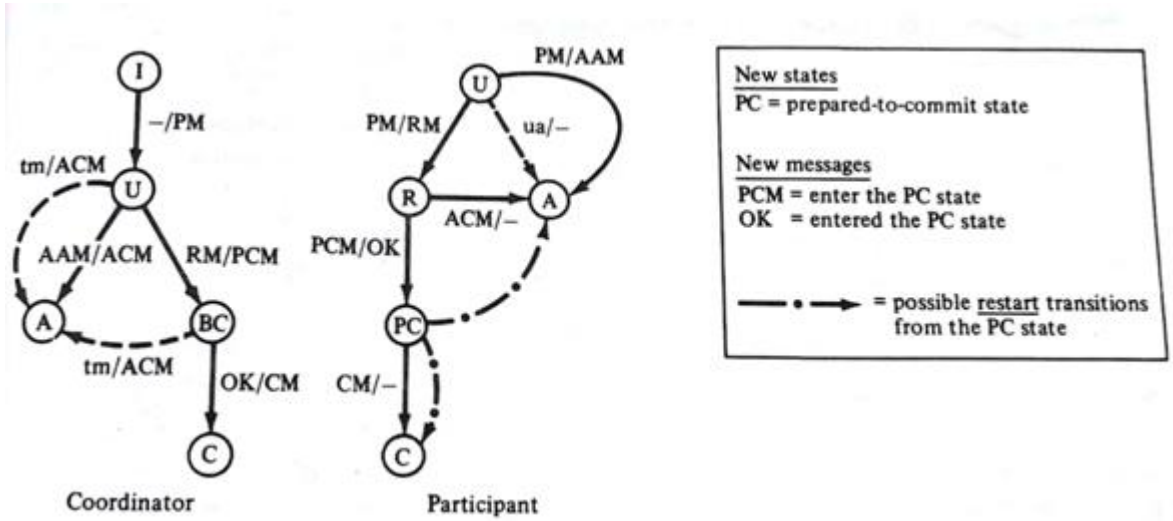


Figure 4.2 two state diagrams for the 3-phase-commitment protocol

This protocol eliminates the blocking problem of the 2-phase-commitment protocol because,

1. If one of the operational participants has received the command and the command was ABORT, then the operational participant can abort the transactions. The failed participant will abort the transaction at restart if it has not done it already.
2. If one of the operational participants has received the command and the command was ENTER-PREPARED-STATE, then all the operational participants can commit the transactions, terminating the second phase if necessary m performing the third phase.
3. If none of the operational participants has received the ENTER-PREPARED-STATE command, 2-phase-commitment protocol cannot be terminated. But with our new protocol, the operational participants can abort the transactions, because the failed participants has not committed. The failed transactions therefore abort the transactions at restart.

This new protocol requires three phases for committing a transaction and two phases for aborting it.

Termination protocol for 3-phase-commitment

"If at least one operational participant has not entered the prepared-to-commit state, then the transactions can be aborted. If at least one operational participant has entered the prepared-to-commit state, then the transactions can be safely committed."

Since the above two condition are not mutually exclusive, in several cases the termination protocol can decide whether to commit or abort the transactions. The protocol which always commits the transactions when both cases are possible is called progressive.

The simplest termination protocol is the centralized, nonprogressive protocol. First a new coordinator is elected by the operational participants. Then the new coordinator behaves as follows:

1. If the new coordinator is in the prepared-to-commit state, it issues to all operational participants the command to enter also in this state. When it has received all the OK messages, it issues the COMMIT command.
2. If the new coordinator is in commit state, i.e. it has committed the transactions, it issues the COMMIT command to all participants.
3. If the new coordinator is in the abort state, it issues the ABORT command to all participants.
4. Otherwise, the new coordinator orders all participants to go back to a state previous to the prepared-to-commit and after it has received all the acknowledgements, it issues the ABORT command.

2. COMMITMENT PROTOCOLS AND NETWORK PARTITIONS

Existence of nonblocking protocols for partitions

The main problem of the existence of nonblocking protocols is, some protocol allows independent recovery in case of site failures.

The protocol we design must work as the following example. Suppose that we can build a protocol such that if one site, say site2, fails, then

1. The other site, site1, terminates the transactions.
2. Site2 at restart terminates the transactions correctly without requiring any additional information from site1.

So we make 4 propositions for the nonblocking commitment protocol, they are,

1. Independent recovery protocols exist only for single-site failures; however there exists no independent recovery protocol which is resilient to multiple-site failures.
2. There exists no nonblocking protocol that is resilient to a network partition if messages are lost when the partition occurs.
3. There exist nonblocking protocols which are resilient to a single network partition if all undeliverable messages are returned to the sender.
4. There exists no nonblocking protocol which is resilient to a multiple partition.

Protocols which can deal with partitions

It is convenient to allow the termination of the transactions by at least one group of sites, possible the largest group so that blocking is minimized. But it is not possible to determine the largest group, because it does not know the size of the other groups.

There are two approaches to this problem, the primary site approach and the majority approach.

In primary site approach, a site is designated as the primary site and the group that contains it is allowed to terminate the transactions.

In majority approach, only the group which contains a majority of ites can terminate the transactions. Here it is possible that no single group reaches a majority, in this case, all groups are blocked.

A. Primary Site Approach

If the 2-phase-commitment protocol is used together with a primary site approach, then it is possible to terminate all the transactions of the group of the primary site(the primary group),if and only if the coordinators of all pending transactions belong to this group. This can be achieved by assigning to the primary site the coordinator function for all transactions.

This approach is inefficient in most types of networks and it is very vulnerable to primary site failure. To avoid this condition we can use 3-phase-commitment protocol can be used in primary group.

B. Majority Approach and Quorum-Based Protocols

The majority approach avoids the disadvantages of the primary site approach. The basic idea is that a majority of sites must agree on the abort or commit of a transaction before the

transaction is aborted or committed. A majority approach requires a specialized commitment protocol. It cannot be applied with the standard 2-phase-commitment.

A straightforward generalization of the basic majority approach consists of assigning different weights to the sites. The protocol which use a weighted majority are called **quorum-based protocols**. The weights which are assigned to the sites are usually called **votes**, since they are used when a site "votes" on the commit or abort of a transaction.

The basic rules of a quorum-based protocol are:

1. Each site I has associated with it a number of votes V_i, $V_i$being a positive integer.
2. Let V indicate the sum of the votes of all sites of the network.
3. A transaction must collect a commit quorum V_c before committing.
4. A transaction must collect an abortquorum V_a before aborting.
5. $V_a + V_c > V$.

Rule 5 ensures that a transaction is either committed or aborted by implementing the basic majority idea. In practice, the choice $V_a + V_c = V + 1$ is the most convenient one.

A commitment protocol which implements this rule must guarantee that at one time a number of sites such that the sum of their votes is greater than V_c agree to commit. It means these sites have entered a prepared-to-commit state. Therefore a quorum based commitment protocol can be obtained from the 3-phase-commitment protocol implementing the quorum requirement.

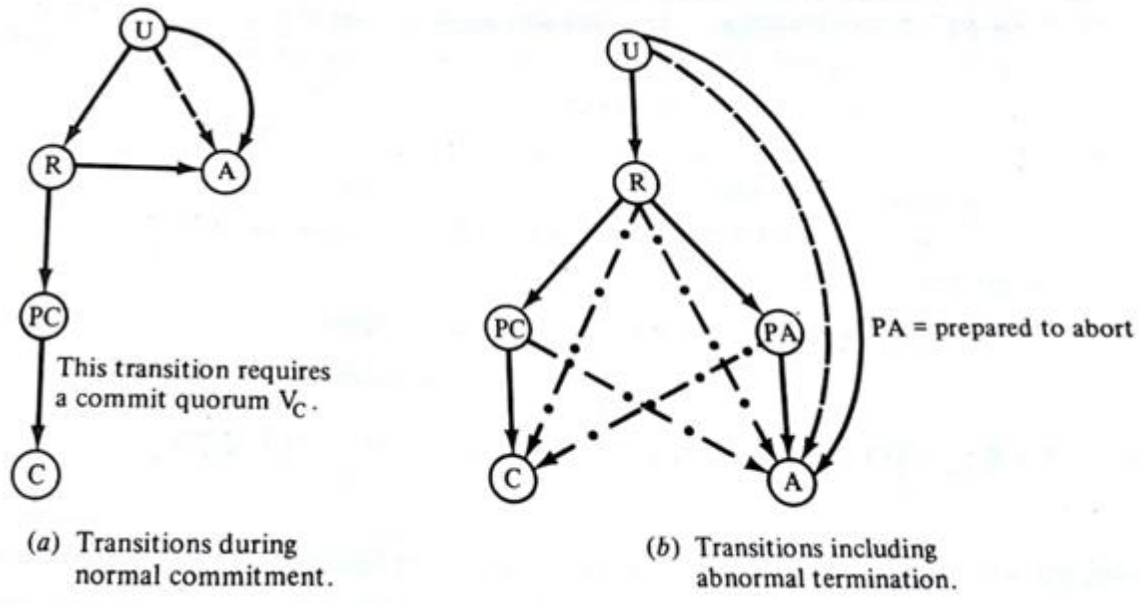


Figure 4.3 Quorum based 3 phase commitment protocol

Termination and restart are more complex in this protocol. So once a site has participated in building a commit (abort) quorum, it cannot participate in an abort (commit) quorum. Since a site cab fail after participating in building a quorum, its participation must be recorded in stable storage.

A centralized termination protocol for the quorum-based 3-phase-commitment has the following structure.

1. A new coordinator is elected.
2. The coordinator collects state information and acts according to the following rules:
   a. If at least one site has committed (aborted), send a COMMIT (ABORT) command to the other sites.
   b. If the number of votes of sites that reached the prepare-to-commit state is greater than or equal to V_c , send a COMMIT command.
   c. If the number of votes of sites in prepare to abort state reaches the abort quorum, send an ABORT command.
   d. If the number of votes of sites that reached the prepare-to-commit state plus the number of votes of uncertain sites is greater than or equal to V_c, send a PREPARE-TO-COMMIT command to uncertain sites, and wait for condition 2b to occur.
   e. If the number of votes of sites that reached the prepare-to-abort plus the number of votes of uncertain sites is greater than or equal to V_a, send a PREPARE-TO-ABORT command to uncertain sites, and wait for condition 2c to occur.
   f. Otherwise, wait for the repair of some failure.

RELIABILITY AND CONCURRENCY CONTROL

The problem arises when a failure happens is addressed here. We have to maximize the number of transactions which are executed during this failure by the operational part of the system.

Consider a transaction T having read-set RS(T) and write-set WS(T) and suppose that we want to run Talone, so that no concurrency control is needed. In order to run T it is necessary that at least one copy of each data item x belonging to RS(T) be available. If this elementary necessary condition is not satisfied, T cannot be executed, because it lacks input data. The availability of the data items of the write-set of T is not strictly required if we run T alone during a failure, because a list of deferred updates can be produced which will be applied to the database when the failure is repaired. Deferred updates can be implemented using “spooler” method.

The availability of a system which allows only one transaction to be run during failure is not satisfactory; therefore, concurrency control must be taken in account. The strongest limitations on the execution of transactions in the presence of failures are due to the need for concurrency control.

A. NONREDUNDANT DATABASES

If the database is nonredundant, then it is very simple to determine which transactions can be executed. Let us consider 2-phase-locking is used for concurrency control. A transaction tries to lock all data items of its readand write-sets before commitment. As there is only one copy of some data item, this copy is either available or not. If the unique copy of some data item of the read or write-set is not available, the transaction cannot commit and must therefore be aborted.

If we assume that only site crashes occur but no partitions, then the availability of the items which belong only one to the write-set is not required, and it is possible to spool the update messages for these items. All transactions which have their read-set available executed completely, including commitment; but the updates affecting sites which are down are stored at spooler sites. When recovery happens, the restart procedures of the failed sites will receive this list of deferred updates and execute them. We consider a crashed site as exclusively locked for the transaction.No other transaction can read the values of data items which are stored here. In the case of partitions the differed updated will cause inconsistent results to be produced- the failure is catastrophic.

In conclusion, if the database is nonredundant, there is not very much to do in order to increase its availability in the presence of failures. Therefore, most reliability techniques consider the case of redundant databases.

B. REDUNDANT DATABASE

The reasons why redundancy is introduced in a distributed database are twofold:

1. To increase the locality of reads, especially in those applications where the number of reads is much larger than the number of writes
2. To increase the availability and reliability of the system.

We deal here essentially with the second aspect; however, in designing reliable concurrency control methods for replicated data the first goal also should be kept in mind.

There are three main approaches to concurrency control based on 2-phase-locking in a redundant database: write-locks-all, majority locking, and primary copy locking.

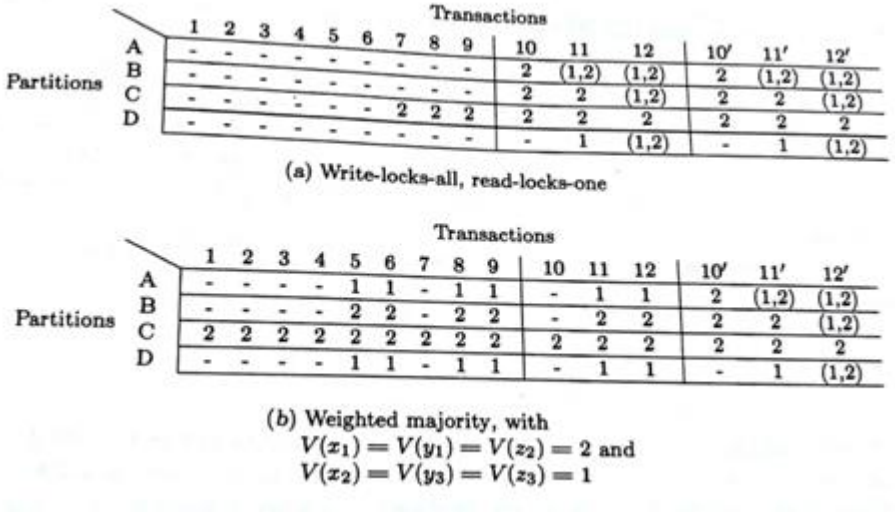
I. WRITE-LOCK-ALL

For transaction with a small write-set and especially for read-only transactions, the system is much more available than for transaction with a large write-set. For read-only transactions sometimes can run in more than one group ,because if a data item has two copies in two copies in two different groups, then no update transaction can write on it and read-only transaction can use each copy consistently.

If we make the assumption that no partitions occur, but only site crashes, then the same approach can be used as with a nonredundant database i.e., the updates of unavailable copies of data items can be spooled. In this case, the availability of the database for update transactions increases very much. In fact, since only the read-set matters in the case, transaction 1,4and 7 have the same availability as transaction 10; transactions 2, 5 and 8 as transaction 11; and transaction 3,6 and 9 as transaction 12. So the example must be carefully interpreted. The fact that a transaction can run in a given group means now that it can be run if all other sites are down, instead of building separate groups. The high increase in availability is obtained at the risk of catastrophic partitions.

Requests to lock or unlock a data item and the messages of the 2-phase-commitment protocol are required for the control of transactions. Control messages carry information and are short. Data messages contain database information and can be long. With the write locks-all approach, we have:

1. **Benefit** - For each transaction executed at site i having x in its read-set, one lock message and one data message are saved.
2. **Cost** - for each transaction which is not executed at site i and has x in its write-set, one lock message and one data message are required, plus the messages required by the commitment protocol.



Transactions

Partitions	1	2	3	4	5	6	7	8	9	10	11	12	10′	11′	12′
A	-	-	-	-	-	-	-	-	-	2	(1,2)	(1,2)	2	(1,2)	(1,2)
B	-	-	-	-	-	-	-	-	-	2	2	(1,2)	2	2	(1,2)
C	-	-	-	-	-	-	2	2	2	2	2	2	2	2	2
D	-	-	-	-	-	-	-	-	-	-	1	(1,2)	-	1	(1,2)

(a) Write-locks-all, read-locks-one

Transactions

Partitions	1	2	3	4	5	6	7	8	9	10	11	12	10′	11′	12′
A	-	-	-	-	1	1	-	1	1	-	1	1	2	(1,2)	(1,2)
B	-	-	-	-	2	2	-	2	2	-	2	2	2	2	(1,2)
C	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
D	-	-	-	-	1	1	-	1	1	-	1	1	-	1	(1,2)

(b) Weighted majority, with $V(x_1) = V(y_1) = V(z_2) = 2$ and $V(x_2) = V(y_3) = V(z_3) = 1$

Figure 4.4 Availability of Transaction

II. WEIGHTED MAJORITY LOCKING

The pure majority locking approach is not very suitable for our example, because two copies of each data item exist; hence to lock a majority we must lock both. So consider a weighted majority approach, or quorum approach, which adopts the same rules which have been used for quorum-based commitment and termination protocols.

These rules, applied to the locking problem, consist of assigning to each data item x a total number of votes V(x), and assigning votes $V(x_i)$ to each copy x_i in such a way that V(x) is the sum of all $V(x_i)$. A read quorum $V_r(x)$ and a write quorum $V_w(x)$ are then determined, such that:

$$V_r(x) + V_w(x) > V(x)$$
$$V_w(x) > V(x)/2$$

A transaction can read(write)x if it obtains read(write) locks on so many copies of x that the sum of their votes is greater than or equal to $V_r(x)(V_w(x))$. Due to the first condition, all conflicts between read and write operations are determined, because two transactions which perform a read and a write operation on x cannot reach the read and write quorum using two disjoint subsets of copies. Likewise, because of the second condition, all conflicts between write operations are determined. Notice that the second condition can be omitted if transactions read all data items which are written.

Let us assign votes for the copies of data items of Figure in the following way:

$$V(x) = V(y) = V(z) = 3$$
$$V(x1) = V(y1) = V(z2) = 2$$
$$V(x2) = V(y3) = V(z3) = 1$$

With this assignment we can now consider the availability of the system in the case of partitions. We choose the read and write quorums to be 2 for all data items. The availability for the 12 transaction is shown in the figure. The following can be observed:

1. Transaction 1,2,3,4,7 and 10 have all the same availability. They are characterized by the fact that they access all three data items either for reading or for writing or for both. Since the read quorum is equal to the write quorum, it makes no difference whether the data item is read or written from the viewpoint of availability. For the same reason, transaction 5,6,8and 11, which access only data items x and y, have the same availability. Also, transactions 9 and 12 have the same availability of the latter group, because the copy with highest weight for y.
2. The availability for update transactions is grater with the weighted majority approach than with write-locks-all, while the availability for read-only transactions is smaller.
3. With this method , read-only transaction increases their availability if they can read an inconsistent database, i.e., if they do not need to lock items, in fact, columns 10',11,and 12, are the same for the majority approach as for the write-locks-all approach .

With the majority approach it is not reasonable to consider the assumption that partitions do not occur. Notice that if we assume the absence of partitions, then the majority approach is dominated by the write-locks-all approach(an approach is dominated by another one if it is worse under all circumstances). In fact, we have seen that the majority and quorum ideas have been developed essentially for dealing with partitions.

Consider now the locality aspect. A transaction reads a data item x at its site of origin, if a copy is locally available. Hence, also in this case a data message is saved if a local copy is available .However, read locks must be obtained at a number of copies corresponding to the read quorum. Therefore, the addition of a copy of x can also force transactions which read x to request more read locks at remote sites. This additional cost is incurred by transactions which have x in their write-set, which must obtain write locks at a number of sites corresponding to the write quorum. Moreover, they have to send a data message to all the sites where there are copies of x.

It is clear that, considering only data messages, the same advantages and disadvantage exist for the majority and the write-locks-all method. When control message are also considered, then the situation is more complex; however, some of the locality motivations for read-only transaction are lost.

III.PRIMARY COPY LOCKING

In the primary copy locking approach, all locks for a data item x are requested at the site of the primary copy. We will assume first that also all he read and write operations are performed on this copy; however, write are then propagated to all other copies.

Several enhancements of the primary copy approach exist which it more attractive. The principal ones are:

1. Allowing consistent reads at different copies than the primary, even if real locks are requested only at the primary; this enhances the locality of reads.
2. Allowing the migration of the primary copy if a site crash makes it unavailable; this enhances availability.
3. Allowing the migration of the primary copy depending on its usage pattern. This also enhances the locality aspect.

The first point deserves a comment. In order to obtain consistent reads at different copies from the primary one, we should use the primary copy method for synchronization, but perform the write and read operations according to the "write all/read one" method. In this approach, the locks are all requested at the primary copy. So, at commitment all copies are updated before releasing the write lock. A read can be performed in this way at any copy, obtaining consistent data.

DETERMINING A CONSISTANT VIEW OF THE NETWORK

There are two aspects of this problem: Monitoring the state of the network, so that state transitions of a site are discovered as soon as possible, and propagating new state information to all sites consistently. Normally we use timeouts in the algorithms in order to discover if a site was down. The use of timeouts can lead to an inconsistent view of the network. Consider the following example in a 3-site network: site 1 sends a message to site2 requesting an answer. If no answer arrives before a given timeout, site 1 sends assumes that sites 2 is down. If site 2 was just slow, then site 1 has a wrong view of the state of site2, which is inconsistent with the view of site 2 about itself. Moreover, a third site 3 could try the same operation at the same time as site 1, obtain an answer within the timeout, and assume that site 2 is up. So it has different view that site1.

A generalized network wide mechanism is built such that all higher-level programs are provided with the following facilities:

1. There is at each site a **state table**containing an entry for eachsite. The entry can be up or down. A program can send an inquiry to the state table for state information.
2. Any program can set a "watch" on any site, so that it receives an interrupt when the site changes state.

The meaning of the state table and of a consistent view in the presence of partitions failures is defined as follow: A site considers up only those sites with which it can communicate.So all crashed sites and all sites which belong to a different group in case of partitions are considered down.A consistent view can be achieved only between sites of the same group.Incase of a partition there are as many consistent views as there are isolated groups of sites. The consistency requirement is therefore that a site has the same state table as all other sites which are up in its state table.

I.Monitoring the state of the network

The basic mechanism for deciding whether a site is up or down is to request a message from it and to wait for a timeout. The requesting site is called controller and the other site is called controlled site. In a generalized monitoring algorithm, instead of having the controller request message from the controlled site, it is more convenient to have the controlled site send I-AM-UP messages periodical to the controller and the controlled site.

Note that if only site crashes are considered, the monitoring function essentially has to detect transitions from up to down states, because the opposite transaction is detected by the site which performs recovery and restart; this site will inform all the others. If, however, partitions also are considered, then the monitoring function has also to determine transitions from down to up states. When a partition is repaired, sites of one group must detect that sites of the other group must detect that sites of the group become available.

Using this mechanism for detecting whether a site is up or down the problem consists of assigning controllers to each site so that the overall message overhead is minimized and the algorithm survives correctly the failure of a controller. The latter requirement is of extreme importance, since in a distributed approach each site is controlled and at the same time performs the function of controller of some other site.

A possible solution is to assign circular ordering to the sites and to assign to each site the function of controller of its predecessor. In the absence of failures, each site periodically sends an I-AM-UP message to its successor and controls that the I-AM-UP message from its predecessor arrives in time. If the I-AM-UP message from the predecessor does not arrive in time, then the controller assumes that the controlled site has failed, updates the state table and broadcasts the updated state table to all other sites.

If the predecessor of a site is down,then the site also has to control its predecessor, and if this one is also down, the predecessor of the predecessor, and so on backward in the ordering until an up site is found is isolated or all other sites have crashed; this does not invalidate the algorithm). In this way, each operational site always has a controller. For example, in site k controls site k-3; i.e., it responsible for discovering that sites k-1 and k-2 recover from down to up. Symmetrically, if the successor of a site is down, then this site has as a controller the first operational site following it in the ordering. For example, site k-3 has site k as controller. Note that in the **FIG sites k-1 and k-2** is not necessarily crashed; they could belong to a different group after a partition.Therefore, the view of the network of sites k and k-3 is not necessarily the “real” state.

Broadcasting a New State

Each time that the monitor function detects a stage change, this function is activated. The purpose of this function is to broadcast the new state table so that all sites of the same group have the same state table so that all sites of the same group have the same state table. Since this function could be activated by several sites in parallel, some mechanism is needed to control interference. A possible mechanism is to attach a globally unique timestamp to each new version of a state table. By including the version number of the current state table in the I-am-up message all sites in the same group can check that they have a consistent view.

The site which starts the propagation of a new state table first performs a synchronization step in order to obtain a timestamp and then sends the state table to all sites which have answered.

DETECTION AND RESOLUTION OF INCONSISTENCY

When a partition of the network occurs, transaction should be run at most in one group of sites if we want to preserve strictly the consistency of the database. In some application it is acceptable to lose consistency in order to achieve more availability. In this case, transaction is allowed to run in all partitions where there is at least one copy of the necessary data. When the failure is repaired, one can try to eliminate the inconsistencies

which have been introduced into the database. For this purpose it is necessary first to discover which portion of the data has become inconsistent, and then to assign to these portions a value which is the most reasonable in consideration of what has happened. The first problem is called the **detection ofinconsistencies.**The second is called the **resolution** of the inconsistencies. While exact solutions can be found for the detection problem, the resolution problem has no general solution, because transaction has serializable way. Therefore the word “reasonable” and not the word”correct” is used for the value which is assigned by the resolution procedure.

DETECTION OF INCONSISTENCIES

Let us assume that, during a partition, transaction have been executed in two or more groups of sites, and that independent updates may have been performed on different copies of the same fragment. Let us first observe that the most naïve solution,consisting of comparing the contents of the copies to check that they are identical, is not only inefficient, but also not correct in general. For example consider an airline reservation system. If, during the partition, reservation for the same flight independently on different copies until the maximum number is reached, then all copies might have the same value for the number of reservation; however, the flight would be overbooked in this case.

A correct approach to the detection of inconsistencies can be based on version number .Assume that one of the approaches is used for determining for each data item, the one group of sites which is allowed to operate on it. The copies of the data item which are stored at the sites of this group are called **master copies**; the others are called **isolated copies.**

During normal operation all copies are master copies and are mutually consistent. For each copy an original **version number** and a **current version number** are maintained .Initially the original version number is set to 0, and the current version number is set to 1; only the current version number is incremented each time that an update is performed on the copy. When a partition occurs, the original version number of each isolated copy is set to the value of its current version number. In this way, the original version number is not altered until the partition is repaired. At this time, the comparison of the current and original version numbers of all copies reveals inconsistencies.

Let us consider an example of this method. Assume that copies x1, x2 and x3 of data item x are stored at three different sites. Let V1,V2 and V3 be the version numbers of x1, x2 and x3. Each Vi is in fact a pair with the original and current version number. Initially all three copies are consistently updated .Suppose that one update has been performed, so that the situation is

$$V1=(0,2),\ V2=(0,2),\ V3=(0,2)$$

Now a partition occurs separating x3 from the other two copies. A majority algorithm is used which chooses x1 and x2 as major copies. The version numbers become now

$$V1=(0,2),\ \ V2=(0,2)\ ,\ V3=(2,2)$$

Suppose now that only the master copies are updated during the partitions. The version numbers become

$$V1=(0,3),\ V2=(0,3),\ V3=(2,2)$$

And after the repair it is possible to see that x3 has not been modified, since the current and original version numbers are equal. In this case, no inconsistency has occurred and it is sufficient to perform the updated during the partition. We have

$$V1=(0,2),\ V2=(0,2),\ V3=(2,3)$$

Since the original version number of x3 is equal to the current version number of x1 and x2 , the master copies have not been updated. If there are no other copies, then we can simply apply to the master copies the updates of x3, since the situation is exactly symmetrical to the previous one. If there are other isolated copies, for example x4 with V4=(2,3), we cannot tell whether x4 was updated consistency with x3 even if version numbers are the same, hence we have to assume inconsistency.

Finally , if both the master and the isolated copies have been updated , which also reveals an inconsistency, then the original and the current version number of the isolated copy are different, and the original version number of the isolated copy is also different from the current version number of the master copies; for example

$$V1=(0,3),\ V2=(0,3),\ V3=(2,3)$$

RESOLUTION OF INCONSISTENCIES

After a partition has been repaired and an inconsistency has been detected a common value must be assigned to all copies of a same data item. The problem of resolution of inconsistency is the determination of this value.

Since in the different group transaction have been executed without mutual synchronization, it seems correct to assign as a common value the one which would be produced by some serializable execution of these same transactions. However besides the difficulty of obtaining this new value, this is not a satisfactory solution, because the transactions which have been executed have produced effects outside of the system which cannot be undone and cannot be simply ignored.

Note that the transaction requiring the high degree of availability which motivates the acceptances of inconsistencies is exactly those which perform effects outside of the system. For example, take the airline reservation example considered before. The reason for running transaction while the system is partitioned is to tell the customers that flight are available; otherwise , it would be simpler to collect the customer request and to apply them to the database after the failure has been repaired.

However, if overbooking has occurred during the partition, then forcing the system to serializable execution would force the system to perform arbitrary cancellation. From the view point of the application, it might be better to keep the over bookings and let normal user cancellations reduce the number of reservations. A possible way of reducing or eliminating overbooking due to partitions is to assign to each site a number of reservations which is smaller than the total number. This number could be proportional to the size of each group or to some other application dependent value.

The above example shows that the resolution of inconsistencies is in general, application-dependent, and hence within the scope of this book, which deals with generalized mechanisms.

CHECKPONTS AND RESTART

There are two types for errors: Omission errors and Commission errors. Omission errors occur when a action (commit/abort) is left out of the transactions being executed. Commission errors occur when a action (commit/abort) is incorrectly included in the transactionexecuted. An error of omission in one transaction will be counted as an error in commission in another transaction.

Cold restart is required after some catastrophic failure which has caused the low of log information on stable storage, so that the current consistent state of the database cannot be reconstructed and a previous consistent state must be restored. A previous consistent state is marked by a checkpoint.

In a distributed database, the problem of cold restart is worse than in a centralized one; this is because if one site has to establish an earlier state then all other sites also have to establish earlier states which are consistent with the one of the site, so that the global state of the distributed database as a whole is consistent. This means that the recovery process is intrinsically global, affecting all sites of the database, although the failure which caused the cold restart is typically local.

A consistent global state C is characterized by the following two properties:

1. For each transaction T, C contains the updates performed by all subtransactions of T at any site, of it does not contain any of them; in the former case we say that T is contained in C.
2. If a transaction T is contained in C, then all conflicting transactions which have preceded T in the serialization order are also contained in C.

Property 1 is related to the atomicity to the transactions: either all effects of T or none of them can appear in a consistent state. Property 2 is related to the serializability of transactions: if a conflicting transaction T' has preceded T, then the updates performed by T' have affected the execution of T; Hence, if we keep the effects of T , we must keep also all the effects of T' . Note that durability of transaction cannot be ensured if we are forced to a cold restart; the effect of some transactions is lost.

The simplest way to reconstruct a global consistent state in a distributed database is to use local dumps, local logs, and **global checkpoints**. A global checkpoint is a set of local checkpoints which are performed all sites of the network and are synchronized by the following condition: if a subtransaction of a transaction T is contained in the local checkpoint at some site, then all other subtransactions of T must be contained in the corresponding local checkpoint at other sites.

If global checkpoints are available, the reconstruction problem is relatively easy. First, at the failed site the latest local checkpoint which can be considered safe is determined; this determines which earlier global state has to be reconstructed. Then all other sites are required to reestablish the local states of the corresponding local checkpoints.

The main problem with the above approach consists in recording global checkpoints. It is not sufficient for one site to broadcast a "write checkpoints" message to all other sites, because it is possible that the situation of Fig arises; in this situation, T2 and T3 are

subtransactions of the same transaction T, and the local checkpoint C2 does not contain subtransaction T2, while the local checkpoint C3 contains sub transaction T3, thus violating the basic requirement for global checkpoints. FIGURE shows also that the fact that T performs a 2- phase-commitment does not eliminate this problem, because the synchronization of subtransactions during 2-phase-commitment and of sites during recording of the global checkpoint is independent.

The simplest way to avoid the above problem is to require that all sites become inactive before each other records its local checkpoint. Note that all sites must remain inactive simultaneously,and therefore coordination is required. A protocol which is very similar to 2-phase-commitment can be used for this purpose; a coordinator broadcasts " prepare for checkpoint" to all sites, each site terminates the execution of subtransactions and then answers READY, and then the coordinator broadcasts " perform checkpoint". This type of method is unacceptable in practice because of the inactivity which is required all the sites. A site has to remain inactive not only for the time required to record its checkpoints, but until all other sites have finished their active transactions. Three more efficient solutions are possible:

1. To find less expensive ways to record global checkpoints, so called **loosely synchronized checkpoints.** All sites are asked by a coordinator to record a global checkpoint; however, they are free to perform it within a large time interval. The responsibility of guaranteeing that all subtransaction of the same transaction are contained in the local checkpoints corresponding to the same global checkpoint is left to transaction management. If the root agent of transaction T starts after checkpoint C_i and before checkpoint C_{i+1} , then each other subtransaction at a different site can be started only after C_i has been recorded at its sites and before C_{i+1} has been recorded . Observing the first condition may force a subtransaction to wait; observing the second condition can cause transaction aborts and restarts.
2. To avoid building global checkpoints at all, let the recovery procedure take the responsibility of reconstructing a consistent global state at cold restart. With this approach, the notion of global checkpoint is abandoned. Each site records its local checkpoints independently from other sites, and the whole effort of building a consistent global state is therefore performed by the cold restart procedure.
3. To use the 2-phase-commitment protocol for guaranteeing that the local checkpoints created by each sites are ordered in a globally uniform way. The basic ideas is to modify the 2-phase-commitment protocol so that the check points idea is to modify the 2-phase-commitment protocol so that the checkpoints of all subtransactions which belong to two distributed transaction T and T^1 are recorded in the same order at all sites where both transaction T and T' are recorded in the same order at all sites where both transactions are executed. Let T_i and T_j be subtransactions T'_i and T_j'be subtransactions of T'. If at site i the checkpoint of subtransaction $T_i$proceeds the checkpoint of T_J should precede the checkpoint of subtransaction T'_j.

DISTRIBUTED DATABASE ADMIBISTRATION

Database administration refers to a variety of activities for the development, control, maintenance, and testing of the software of the database application. Database administration

is not only a technical problem, since it involves the statement of policies under which users can access the database, which is clearly also an organization problem.

The technical aspects of database administration in a distributed environment focus on the following problems:

1. The content and management of the catalogs with this name, we designate the information which is required by the system for accessing the database. In distributed systems, catalogs include the description of fragmentation and allocation of data and the mapping to local names.
2. The extension of protection and authorization mechanisms to distributed systems.

CATALOG MANAGEMENT IN DISTRIBUTED DATABASES

Catalogs of distributed databases store all the information which is useful to the system for accessing data correctly and efficiently and for verifying that users have the appropriate access rights to them.

Catalogs are used for:

1. **Translating applications** - Data referenced by applications at different levels of transparency are mapped to physical data (physical images in our reference architecture).
2. **Optimizing applications -** Data allocation, access methods available at each site, and statistical information (recorded in the catalogs) are required for producing an access plan.
3. **Executing applications** - Catalog information is used to verify that access plans are valid and that the users have the appropriate access rights.

Catalogs are usually updated when the users modify the data definition.It happens when global relations, fragments, or images are created or moved, local access structures are modified, or authorization rules are changed.

I. CONTENT OF CATALOGS

Several classifications of the information which is typically stored in distributed database catalogs are possible.

1. **Global schema description -**It includes the name of global relations and of attributes.
2. **Fragmentation description -**In horizontal fragmentation, it includes the qualification of fragments. In vertical fragmentation, it includes the attributes which belong to each fragment.In mixed fragmentation, it includes both the fragmentation tree and the description of the fragmentation corresponding to each nonleaf node of the tree.
3. **Allocation description -** It gives the mapping between fragments and physical images.
4. **Mappings to local names -**It is used for binding the names of physical images to the names of local data stored at each site.
5. **Access method description -**It describes the access methods which are locally available at each site. For instance, in the case of a relational system, it includes the number and types of indexes available.
6. **Statistics on the database** - They include the profiles of the database

7. **Consistency information (protection and integrity constraints) -** It includes information about the users' authorization to access the database, or integrity constraints on the allowed values of data.
   Examples of authorization rules are:
   a. Assessing the rights of users to perform specific actions on data. The typical actions considered are: read, insert, delete, update, move.
   b. Giving to users the possibility of granting to other users the above rights. Some references in the literature also include in the catalog content state information (such as locking or recovery information); it seems more appropriate to consider this information as part of a system's data structure and not of the catalog's content.

II. THE DISTRIBUTION OFCATALOGS

When catalogs are used for the translation, optimization, and execution of applications, their information is only retrieved.When they are used in conjunction with a change in data definitions, they are updated. In a few systems, statistics are updated after each execution, but typically updates to statistics are batched. In general, retrieval usage is quantitatively the most important, and therefore the ratio between updates and queries is small.

Solutions given to catalog management with and without site autonomy are very different. Catalogs can be allocated in the distributed database in many different ways. The three basic alternatives are:

Centralized catalogs

The complete catalog is stored at one site.This solution has obvious limitations, such as the loss of locality of applications which are not at the central site and the loss of availability of the system, which depends on this single central site.

Fully replicated catalogs

Catalogs are replicated at each site.This solution makes the read-only use of the catalog local to each site, but increases the complexity of modifying catalogs, since this requires updating catalogs at all sites.

Local catalogs

Catalogs are fragmented and allocated in such a way that they are stored at the same site as the data to which they refer.

A practical solution which is used in several systems consists of periodically caching catalog information which is not locally stored. This solution differs from having totally replicated catalogs, because cached information is not kept up-to-date.

If an application is translated and optimized with a different catalog version than the up-to-date one, this is revealed by the difference in the version numbers. This difference can be observed either at the end of compilation, when the access plan is transmitted to remote sites, or at execution time.

In the design of catalogs for Distributed-INGRES, five alternatives were considered,

1. The centralized approach

2. The full replication of items 1, 2, and 3 of catalog content and the local allocation of remaining items
3. The full replication of items 1, 2, 3, 4, and 5 of catalog contentand the local allocation of remaining items
4. The full replication of all items 5 of catalog content.
5. The local allocation of all items with remote "caching"

SDD-1 considers catalog information as ordinary user data; therefore an arbitrary level of redundancy is supported. Security, concurrency, and recovery mechanisms of the system are also used for catalog management.

III. OBJECT NAMING AND CATALOG MANAGEMENT WITH SITE AUTONOMY

We now turn our attention to the different problems which arise when site autonomy is required. The major requirement is to allow each local user to create and name his or her local data independently from any global control, at the same time allowing several users to share data.

1. Data definition should be performed locally.
2. Different users should be able, independently, to give the same name to different data.
3. Different users at different sites should be able to reference the same data.

In the solution given to these problems in R*prototype, two types of names is used:

1. **System wide names** are unique names given to each object in the system.
   They have four components:
   a. The identifier of the user who creates the object
   b. The site of that user
   c. The object name
   d. The birth site of the object, i.e., the site at which the object was created.

   An example of a systemwide name is

   User_ 1 @San_ Jose.EMP@Zurich

   where the symbol @ is a separator which precedes site names.
   Here, User_ 1 from San Jose has created a global relation EMP at Zurich. The same user name at different sites corresponds to different users (i.e., JohnOSF is not the same as JohnOLA). This allows creating user names independently.
2. **Print names** are shorthand names for systemwide names. Sine in systemwide names a, b, and d part can be omitted, name resolution is made by context, where a context is defined as the current user at the local site.

   a. A missing user identifier is replaced by the identifier of the current user.
   b. A missing user site or object site is replaced by the current site.

It is also possible for each user to define synonyms, which map simple names to systemwide names. Synonyms are created for a specific user at a specific site.Synonym mapping of a simple name to a systemwide name is attempted before name resolution.

Catalog management in R* satisfies the following requirements:

1. Global replication of a catalog is unacceptable, since this would violate the possibility of autonomous data definition.
2. No site should be required to maintain catalog information of objects which are not stored or created there.
3. The name resolution should not require a random search of catalog entries in the network.
4. Migration of objects should be supported without requiring any change in programs.

The above requirements are met by storing catalog entries of each object as follows:

1. One entry is stored at the birth site of the object, until the object is destroyed. If the object is still stored at its birth site, the catalog contains all the information; otherwise, it indicates the sites at which there are copies of the object.
2. One entry is stored at every site where there is a copy of the object.

The catalog content in R* includes relation names, column names and types, authorization rules, low-level objects' names, available access paths, and profiles. R* supports the "caching" of catalogs, using version numbers to verify the validity of cached information.

AUTHORIZATION AND PROTECTION

I. Site-to-Site Protection

The first security problem which arises in a distributed database is initiating and protecting intersite communication. When two database sites communicate, it is important to make sure that:

1. At the other side of the communication line is the intended site (and not an intruder).
2. No intruder can either read or manipulate the messages which are exchanged between the sites.

The first requirement can he accomplished by establishing an identification protocol between remote sites. When two remote databases communicate with each other, on the first request they also send each other a password. When two sites decide to share some data they follow R* mechanism.

The second requirement is to protect the content of transmitted messages once the two identified sites start to communicate. Messages in a computer network are typically routed

along paths which involve several intermediate nodes and trans-missions, with intermediate buffering.

The best solution to this problem consists of using cryptography, a standard technique commonly used in distributed information systems, for instance for protecting communications between terminals and processing units. Messages ("plaintext") are initially encoded into cipher messages ("ciphertext") at the sender site, then transmitted in the network, and finally decoded at the receiver site.

II. User Identification

When a user connects to the database system, they must be identified by the system.The identification is a crucial aspect of preserving security, because if an intruder could pretend to be a valid user, then security would be violated.

In a distributed database, users could identify themselves at any site of the distributed database. However, this feature can be implemented in two ways which both show negative aspects.

1. Passwords could be replicated at all the sites of the distributed database. This would allow user identification to be performed locally at each site, but would also compromise the security of passwords, since it would they easier for an intruder to access them.
2. Users could each have a "home" site where their identification is performed; in this scenario, a user connecting to a different site would be identified by sending a request to the home site and letting this site perform the identification.

A reasonable solution is to restrict each user to identifying themselves at the home site. This solution is consistent with the idea that users seem to be more "static" than, for instance, data or programs. A "pass-through" facility could be used to allow users at remote sites to connect their terminals to their "home" sites in order to identify themselves.

III. Enforcing Authorization Rules

Once users are properly identified, database systems can use authorization rules to regulate the actions performed upon database objects by them. In a distributed environment, additional problems include the allocation of these rules, which are part of the catalog, and the distribution of the mechanisms used for enforcing them. Two alternative, possible solutions are:

1. Full replication of authorization rules. This solution is consistent with having fully replicated catalogs, and requires mechanisms for distributing online updates to them. But, this solution allows authorization to be checked either at the beginning of compilation or at the beginning of execution.
2. Allocation of authorization rules at the same sites as the objects to which they refer. This solution is consistent with local catalogs and does not incur the update overhead as in the first case.

The second solution is consistent with site autonomy, while the first is consistent with considering a distributed database as a single system.

The authorizations that can be given to users of a centralized database include the abilities of reading, inserting, creating, and deleting object instances (tuples) and of creating and deleting objects (relations or fragments).

IV. Classes of Users

For simplifying the mechanisms which deal with authorization and the amount of stored information, individual users are grouped into classes, which are all granted the same privileges.

In distributed databases, the following considerations apply to classes of users:

1. A "natural" classification of users is the one which is induced by the distribution of the database to different sites.It is likely that "all users at site x" have some common properties from the viewpoint of authorization. An explicit naming mechanism for this class should be provided.
2. Several interesting problems arise when groups of users include users from multiple sites. Problems are particularly complex when multiple-site user groups are considered in the context of site autonomy. So, mechanisms involve the consensus of the majority or of the totality of involved sites, or a decision made by a higher-level administrator. So, multiple-site user groups contrast with pure site autonomy.

SATHYABAMA
INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)
Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE
www.sathyabama.ac.in

SCHOOL OF COMPUTING

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

UNIT – V – DISTRIBUTED DATABASE – SCS1613

UNIT 5

DATABASE INTEGRATION AND MANAGEMENT

Database Integration- Scheme Translation- Scheme Integration- Query Processing Query Processing Layers in Distributed Multi-DBMSs- Query Optimization Issues- Transaction Management Transaction and Computation ModelMultidatabase Concurrency Control- Multidatabase Recovery- Object Orientation And Interoperability- Object Management Architecture - Distributed Component Model.

DATABASE INTEGRATION

Database integration means that multiple different applications have their data stored in a specific database – the integration database – so that data is available across all of these different applications. In other words, the data is available between two different parties and therefore, can be easily accessed and implemented into a different application without having to transfer to a different database.

For the database integration to work successfully, it needs to have a plan that allows for all of the client applications to be taken into account. Whether the scheme is more complex, general or both is irrelevant because a separate group controls the database to negotiate between the numerous different applications and the database group. In other words, this plan makes it possible for all the applications to be grouped together into that one database group.

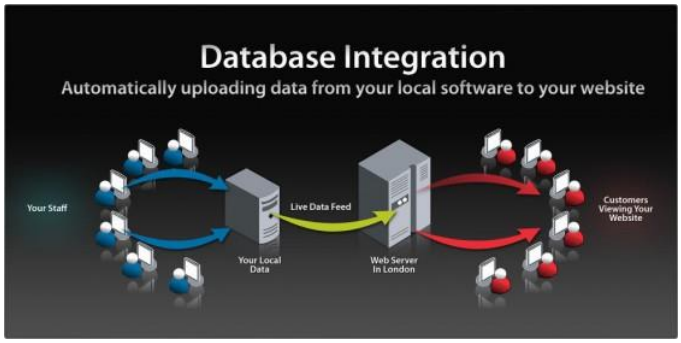


Figure 5.1 Database Integration

NECESSARY OF DATABASE INTEGRATION

The fundamental reason that database integration is necessary is because it allows for data to be shared throughout an organization without there needing to be another set of integration services on each application. It would be a tremendous waste of resources if each application needed something to convert the data into data it can read. By using database integration, it allows for the information to automatically be integrated so if, at any time the data is needed, it can be pulled up and accessed.

On top of that, it helps when two companies that are merging have their data integrated because when their databases come together, the data can mesh easily. If the data wasn't integrated, an server manager would have to go in and manually integrate everything which can become a hassle and, as previous mentioned, result in a waste of resources. Therefore, integrating before a merger is definitely ideal.

Another application that database integration can be used for is in the scientific community. When collecting data, a scientist might use one application for one bit of data. Then, he'll go

to a different application for a different bit of data. By having database integration, the data becomes readily available across the spectrum without thereneeding to be any wasted time. This results in more successful experiments.

All in all, database integration is becoming a technology that more companies are investing in, especially as the quantity and connectivity of data increases. As people need to access more data and share data between departments, companies have realized that have all the data integrated on a database is an incredible time saver.

DATABASE INTEGRATION = TRANSLATION AND SCHEMA INTEGRATION

- **Database integration** is done in most cases in two-steps : **schema translation** (or simply **translation**) and **schema integration**.
- **Scheme translation** means the translation of the participating local schemes into a **common intermediate canonical representation.**

  > e.g. if a network and a relational model is used, then an intermediate data model should be chosen, if it is the relational one, the database scheme formulated in the network model is translated into a scheme based on the relational model.

- Scheme transformation is of course only necessary if different data models are involved.
- The scheme integration integrates each intermediate schemes into a **global conceptual scheme**.
- All intermediate schema base on the same data model, the so called **target model**, which is of course the data model for the global conceptual scheme.

THE EXAMPLE FOR THE TRANSLATION AND THE SCHEMA INTEGRATION

- We consider the following three local schema. The first one is based on the relational model, the second one on the network model (the CODEASYL network) and the third one on the entity-relationship data model.
- **First** scheme, the **Relational Engineering Database Representation** :

  E(ENO, ENAME, TITLE) *Each Engineer Description*
  J(JNO, CNAME, JNAME, BUDGET, LOC) *Job Description*
  G(ENO, JNO, RESP ,DUR) *Engineer to Job relation description*
  S(TITLE, SALARY) *Salary description*

- **Second** scheme : the **CODEASYL Network Definition of the Employee Database** :

  Two records : DEPARTMENT and EMPLOYEE and their attributes DEP-NAME and so on. One link between the records with →, named employs which links the two corresponding records. It can model only one-to-many relationships. The schema representation looks like :

DEPARTMENT : DEP-NAME BUDGET MANAGER →(employs)
EMPLOYEE : E# NAME ADDRESS TITLE SALARY

SCHEMA TRANSLATION :

- Schema translation is the task of mapping one schema to another.
- Requires the specification of the **target data model** for the global conceptual schema definition.
- Some rare approaches did merge the translation and integration phase, but increases the complexity of the whole process.
- In the example, the Entity-Relationship model is chosen as the target model.
- The first scheme translation is the CODEASYL network schema to an E-R-scheme one.

SCHEME TRANSLATION 1 : CODEASYL SCHEMA TO E-R SCHEMA

- One entity is created for each record. Thus, an EMPLOYEE and one DEPARTMENT entity is created.
- The attributes of the records are taken directly into the E-R scheme.
- Finally, the links employs becomes a many-to-one relationship from the EMPLOYEE entity to the DEPARTMENT entity. The final model looks like :

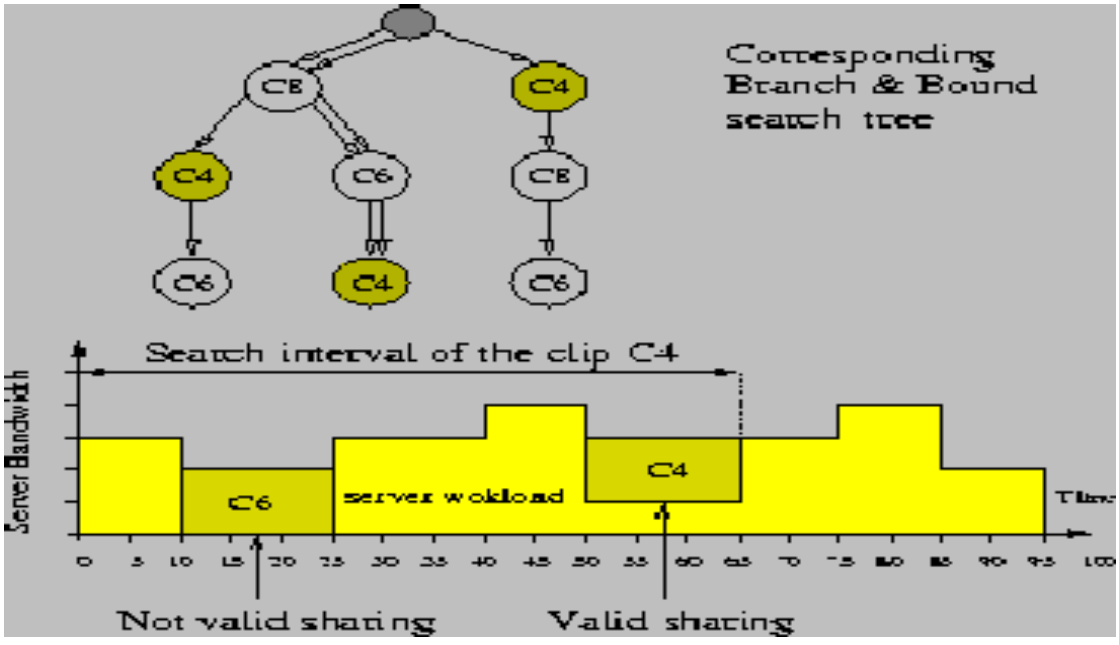


Figure 5.2 Schema Translation 1

Remark : Many-to-many relationships modelled in the network by some intermediate records can be represented directly by one many-to-many relationship (→ translation should be optimized).

SCHEME TRANSLATION 2 : RELATIONAL SCHEMA TO E-R SCHEMA

- The example relational model of the engineering database consists of four relations :

E(ENO, ENAME, TITLE) *Each Engineer Description*
J(JNO, CNAME, JNAME, BUDGET, LOC) *Job Description*
G(ENO, JNO, RESP ,DUR) *Engineer to Job relation description*
S(TITLE, SALARY) *Salary description*

- Identify the base relations : E and J clearly corresponds to an entity.
- Identify the relationships : G corresponds to a relationship, ENO and JNO are foreign keys, thus a relationship between J and E can be identified.
- Handling of S is difficult.
- First it can be a entity. In such a case a relationship between S and E must be established (this would be a many-to-one relationship, e.g. pay between S and E). No relation is specified for this relationship.

  An employee could have only one salary, but a salary can belong to many employees.

- Second salary could be an attribute of E, cleaner, but the relationship between the title and salary is lost.
    - See below the result E-R scheme, with SAL as attribute of E.

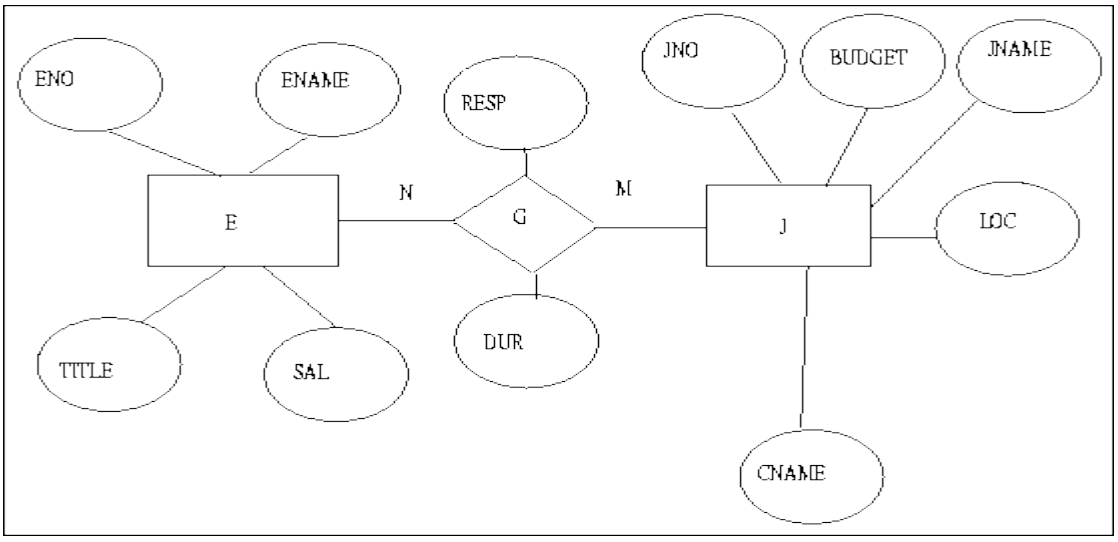


Figure 5.3 Schema Translation 2

SCHEME INTEGRATION :

- All local scheme are now translated to an intermediate scheme based on the target model. The task of the schema integration is now to generate the **global conceptual schema (CGS)**, which can be queried by the user of the MDBMS.
- Ozsu defines the schema integration, as the process of **identifying** the components of a database which are related to one another, **selecting** the best representation for the global conceptual schema and finally **integrating** the components of each intermediate schema.
- Integration methodologies are either **binary** or u***n*ary**

  Binary integration methodologiesinvolves the manipulation of two schema at a time. These occurs either ladder (linear tree !) or purely binary (bushy tree !).

- Binary are either one-shot (integration of all schema) or interactive (integration of 2,3,4 .. at a time). Binary approaches are a special case of the latter.

  In general, the one-shot approach is very complex and rarely used, mostly the binary approach is used (Determine the best ordering!).

- Very good graphical tools exists now which help the identification and integration approach.

OVERVIEW OVER THE SCHEMA INTEGRATION :

- **Preintegration** : identify the keys and defines the ordering of the binary processing approach.
- **Comparison** : Identification of naming and structural conflicts.
- **Conformation** : Resolution of the naming and structural conflicts.
- **Restructeration and Merging** of the different intermediate schema to the global conceptual scheme (GCS).
- Interaction with an integrator is absolutely necessary.

Preintegration

- **Preintegration** establishes the rules of the integration process, i.e. the integration method is selected (e.g. binary iterative n-ary) and then the **order of the schema integration** (i.e. which intermediate schema is integrated with which one first).
- **Candidate keys** are determined. Here for each of the entities in all intermediate schemes, the keys are determined.
- Potentially **equivalent domains** of attributes are detected and transformation rules between the domains should be determined (e.g. one scheme defines the attribute temperature in Grad Celsius, the other one in Fahrenheit, transformation rules between the different domains should be prepared for further integration).

Comparison

- The comparison phase detects **naming conflicts**, **relationships between schemes** and **structural conflicts**.

- Naming conflicts are either the **synonym** or the **homonyms** problem.
- Two identical entities which have different names are **synonyms**, and two different entities that have identical names, but are not identical entities, are **homonyms**.
- Example 1 : ENGINEER and E are synonyms and they both refer to an engineer entity.
- Title in the network model refers to an employee and is different from the title related to an engineer, thus these are homonyms.

Relationship between the schemes

- The determination of the relationship bases on the recognition of the synonyms as described before.
- There are four possibilities of relationships between schemes

  1) Equivalent
  2) One is subset of the other
  3) Some components from any may occur in the other
  4) Completely no overlap.

Structural conflicts

- **Type Conflicts** : Type conflict happens if the same object is represented by an attribute in the one intermediate scheme and by an entity in another scheme.
- **Dependency Conflicts** : This conflict occurs, when the different relationship modes (e.g. one-to-one versus many-to-many) are used to represent the same thing in different schemas.
- **Key Conflicts** : This conflict happens, if different candidate keys are available and different primary keys are selected in different scheme.
- **Behavorial Conflicts** are implied by the modeling process. For example deleting the last employee out of the employee record can result in an empty department, as for the engineers this may not be allowed).

Conformation

- **Conformation** is the resolution of the conflicts that are determined at the comparison phase.
- **Naming conflicts** are resolved by simply renaming conflicting ones. In the case of **homonyms**, the identical entities or attributes are extended with the name of the entity and the name of the scheme it belongs to.
- **Structural conflicts** are resolved by transforming entities/attributes or relationships between them.

Resolving structural conflicts

- **Resolving** attribute to represent it.
- **Remark** : Key attributed to Entities require supplemental steps.
- **Dependency Conflicts** will be resolved by choosing the most general relationship.
- The restructuring is virtually an art rather than a science. Semantic knowledge about the all intermediate schemas is repaired, which makes an automatic resolution difficult. There exists many supporting tools.

- structural conflicts means the **restructuring** of some schemes to eliminate the conflicts.
- **Attribute &Entity** : A non-key attribute can be transformed into an entity by creating an intermediate relationship connecting the new entity and a new **Merging and Restructuration**

- All modified and non-conflicting schemes must be **first merged** into a single database schema and **secondly restructured** to create the 'best' (see later) one.
- The **merging** follows the integration order ones fixed in the Preintegration. The merging should be **complete**, i.e. all components of all the intermediate schema should be find their place in the merged one.
- Now a **Restructuration** would take place which searches for the minimal one, thus the redundant relationships are removed.
- Finally, the scheme could be re-transformed to be more **understandable**. This process is in its great parts autonomous and this mechanism ignores all kind of understandability, it is often necessary by the integrator to rebuild or extend some relationships (here the minimalist can be lost) in a way that the user can understand the scheme and thus formulate correct queries.

ACID PROPERTIES : atomicity, consistency, isolation, and durability.

Atomicity

A transaction's changes to the state are atomic: either all happen or none happen. These changes include database changes, messages, and actions on transducers.

Consistency

A transaction is a correct transformation of the state. The actions taken as a group do not violate any of the integrity constraints associated with the state.

Isolation

Even though transactions execute concurrently, it appears to each transaction T, that others executed either before T or after T, but not both.

Durability

Once a transaction completes successfully (commits), its changes to the database survive failures and retain its changes.

QUERY PROCESSING :

• Query Processing Overview
• Query Optimization

• Distributed Query Processing Steps

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

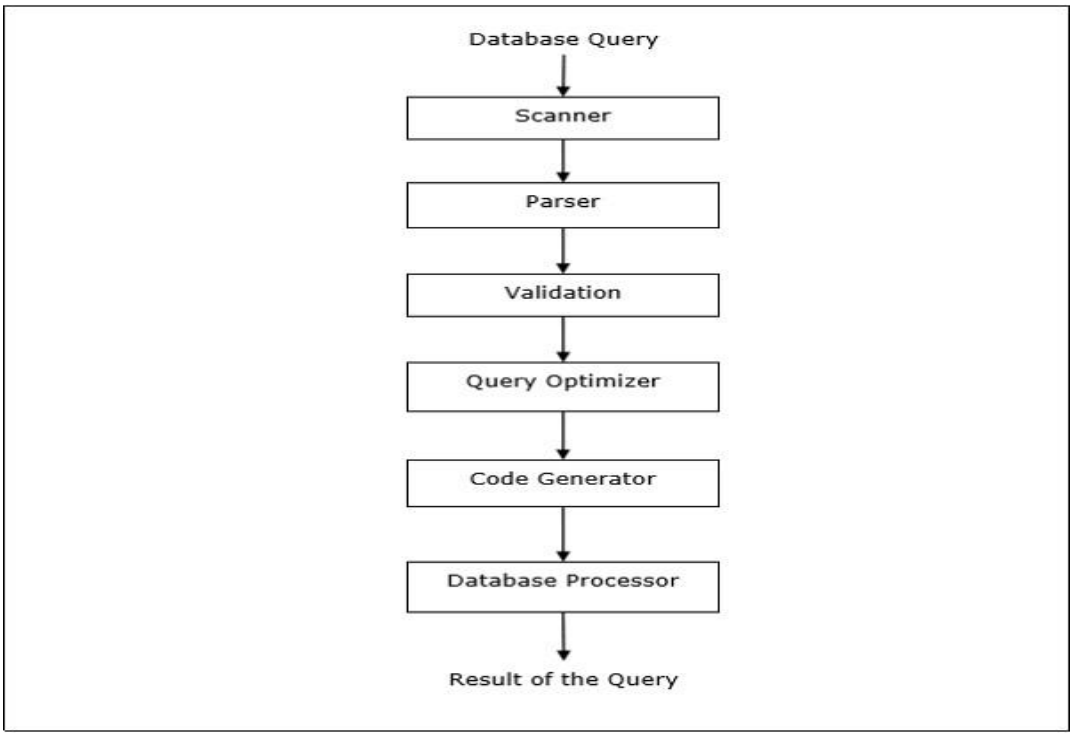


Figure 5.4 Query Processing

RELATIONAL ALGEBRA

Relational algebra defines the basic set of operations of relational database model. A sequence of relational algebra operations forms a relational algebra expression. The result of this expression represents the result of a database query.

The basic operations are −

- Projection
- Selection
- Union
- Intersection
- Minus
- Join

Projection

Projection operation displays a subset of fields of a table. This gives a vertical partition of the table.

Syntax in Relational Algebra

π<AttributeList>(<TableName>)π<AttributeList>(<TableName>)

For example, let us consider the following Student database −

STUDENT				
Roll_No	**Name**	**Course**	**Semester**	**Gender**
2	Amit Prasad	BCA	1	Male
4	Varsha Tiwari	BCA	1	Female
5	Asif Ali	MCA	2	Male
6	Joe Wallace	MCA	1	Male
8	Shivani Iyengar	BCA	1	Female

Table 5.1 Student Data

If we want to display the names and courses of all students, we will use the following relational algebra expression −

πName,Course(STUDENT)πName,Course(STUDENT)

Selection

Selection operation displays a subset of tuples of a table that satisfies certain conditions. This gives a horizontal partition of the table.

SYNTAX IN RELATIONAL ALGEBRA

σ<Conditions>(<TableName>)σ<Conditions>(<TableName>)

For example, in the Student table, if we want to display the details of all students who have opted for MCA course, we will use the following relational algebra expression −

σCourse="BCA"(STUDENT)σCourse="BCA"(STUDENT)

Combination of Projection and Selection Operations

For most queries, we need a combination of projection and selection operations. There are two ways to write these expressions −

- Using sequence of projection and selection operations.
- Using rename operation to generate intermediate results.

For example, to display names of all female students of the BCA course −

- Relational algebra expression using sequence of projection and selection operations

πName(σGender="Female"ANDCourse="BCA"(STUDENT))πName(σGender="Female"ANDCourse="BCA"(STUDENT))

- Relational algebra expression using rename operation to generate intermediate results

FemaleBCAStudent←σGender="Female"ANDCourse="BCA"(STUDENT)FemaleBCAStudent←σGender="Female"ANDCourse="BCA"(STUDENT)

Result←πName(FemaleBCAStudent)Result←πName(FemaleBCAStudent)

Union

If P is a result of an operation and Q is a result of another operation, the union of P and Q (pUQpUQ) is the set of all tuples that is either in P or in Q or in both without duplicates.

For example, to display all students who are either in Semester 1 or are in BCA course −

Sem1Student←σSemester=1(STUDENT)Sem1Student←σSemester=1(STUDENT)

BCAStudent←σCourse="BCA"(STUDENT)BCAStudent←σCourse="BCA"(STUDENT)

Result←Sem1StudentUBCAStudentResult←Sem1StudentUBCAStudent

Intersection

If P is a result of an operation and Q is a result of another operation, the intersection of P and Q (p∩Qp∩Q) is the set of all tuples that are in P and Q both.

For example, given the following two schemas −

EMPLOYEE

EmpID	Name	City	Department	Salary

PROJECT

PId	City	Department	Status

To display the names of all cities where a project is located and also an employee resides −

CityEmp←πCity(EMPLOYEE)CityEmp←πCity(EMPLOYEE)

CityProject←πCity(PROJECT)CityProject←πCity(PROJECT)

Result←CityEmp∩CityProjectResult←CityEmp∩CityProject

Minus

If P is a result of an operation and Q is a result of another operation, P - Q is the set of all tuples that are in P and not in Q.

For example, to list all the departments which do not have an ongoing project (projects with status = ongoing) −

AllDept←πDepartment(EMPLOYEE)AllDept←πDepartment(EMPLOYEE)

ProjectDept←πDepartment(σStatus="ongoing"(PROJECT))ProjectDept←πDepartment(σStatus="ongoing"(PROJECT))

Result←AllDept−ProjectDeptResult←AllDept−ProjectDept

Join

Join operation combines related tuples of two different tables (results of queries) into a single table.

For example, consider two schemas, Customer and Branch in a Bank database as follows −

CUSTOMER

CustID	AccNo	TypeOfAc	BranchID	DateOfOpening

BRANCH

BranchID	BranchName	IFSCcode	Address

To list the employee details along with branch details −

Result←CUSTOMER⋈Customer.BranchID=Branch.BranchIDBRANCHResult←CUSTOMER⋈Customer.BranchID=Branch.BranchIDBRANCH

LAYERS OF QUERY PROCESSING :

.

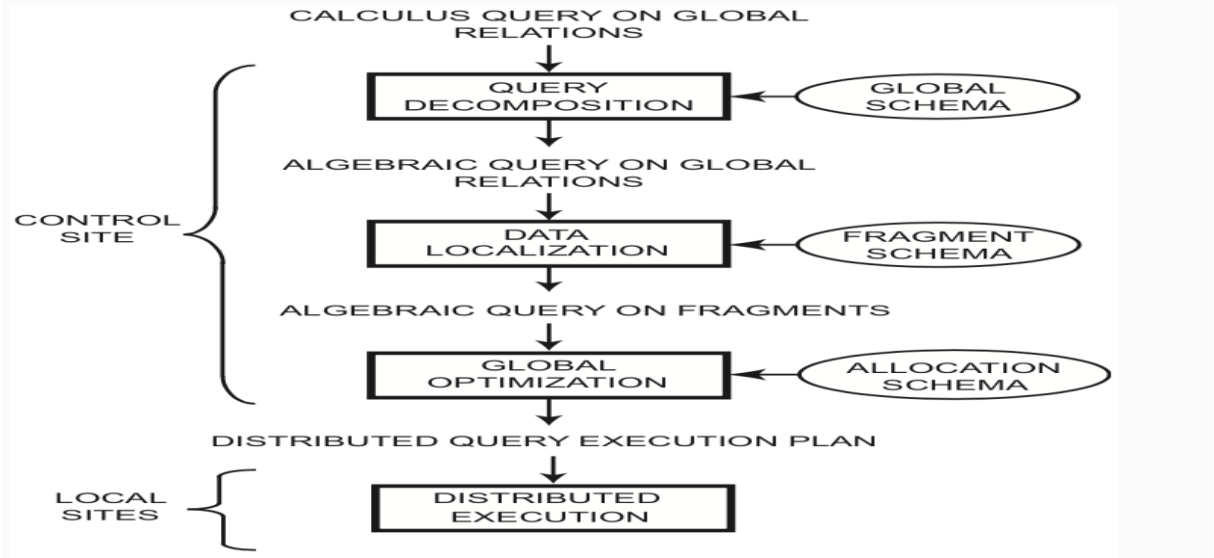


Figure 5.5 layers of query processing

The problem of query processing can itself be decomposed into several subproblems, corresponding tovarious layers. A generic layering scheme for query processing is shown where each layer solves a

well-defined subproblem. To simplify the discussion, let us assume a static and semicentralized query processor that does not exploit replicated fragments. The input is a query on global data expressed in relational calculus. This query is posed on global (distributed) relations, meaning that data distribution is hidden. Four main layers are involved in distributed query processing. The first three layers map the input query into an optimized distributed query execution plan. They perform the functions of query decomposition, data localization, and global query optimization. Query decomposition and data localization correspond to query rewriting. The first three layers are performed by a central control site and use schema information stored in the global directory. The fourth layer performs distributed queryexecution by executing the plan and returns the answer to the query. It is done by the local sites and the control site

GENERIC LAYERING SCHEME FOR DISTRIBUTED QUERY PROCESSING

QUERY DECOMPOSITION

The first layer decomposes the calculus query into an algebraic query on global relations. The information needed for this transformation is found in the global conceptual schema describing the global relations. However, the information about data distribution is not used here but in the next layer. Thus the techniques used by this layer are those of a centralized DBMS.

Query decomposition can be viewed as four successive steps. First, the calculus query is rewritten in a normalized form that is suitable for subsequent manipulation. Normalization of a query generally involves the manipulation of the query quantifiers and of the query qualification by applying logical operator priority.

Second, the normalized query is analyzed semantically so that incorrect queries are detected and rejected as early as possible. Techniques to detect incorrect queries exist only for a subset of relational calculus. Typically, they use some sort of graph that captures the semantics of the query.

Third, the correct query (still expressed in relational calculus) is simplified. One way to simplify a query is to eliminate redundant predicates. Note that redundant queries are likely to arise when a query is the result of system transformations applied to the user query. Such transformations are used for performing semantic data control (views, protection, and semantic integrity control).

Fourth, the calculus query is restructured as an algebraic query**.** That several algebraic queries can be derived from the same calculus query, and that some algebraic queries are “better” than others. The quality of an algebraic query is defined in terms of expected performance. The traditional way to do this transformation toward a “better” algebraic specification is to start with an initial algebraic query and transform it in order to find a “good” one. The initial algebraic query is derived immediately from the calculus query by translating the predicates and the target statement into relational operators as they appear in the query. This directly translated algebra query is then restructured through transformation rules. The algebraic query generated by this layer is good in the sense that the worse executions are typically avoided. For instance, a relation will be accessed only once, even if there are several select predicates. However, this query is generally far from providing an optimal execution, since information about data distribution and fragment allocation is not used at this layer.

DATA LOCALIZATION :

The input to the second layer is an algebraic query on global relations. The main role of the second layer is to localize the query's data using data distribution information in the fragment schema. We saw that relations are fragmented and stored in disjoint subsets, called fragments, each being stored at a different site. This layer determines which fragments are involved in the query and transforms the distributed query into a query on fragments. Fragmentation is defined by fragmentation predicates that can be expressed through relational operators. A global relation can be reconstructed by applying the fragmentation rules, and then deriving a program, called a localization program, of relational algebra operators, which then act on fragments. Generating a query on fragments is done in two steps. First, the query is mapped into a fragment query by substituting each relation by its reconstruction program (also called materialization program). Second, the fragment query is simplified and restructured to produce another "good" query. Simplification and restructuring may be done according to the same rules used in the decomposition layer. As in the decomposition layer, the final fragment query is generally far from optimal because information regarding fragments is not utilized.

GLOBAL QUERY OPTIMIZATION :

The input to the third layer is an algebraic query on fragments. The goal of query optimization is to find an execution strategy for the query which is close to optimal. Remember that finding the optimal solution is computationally intractable. An execution strategy for a distributed query can be described with relational algebra operators and *communication primitives* (send/receive operators) for transferring data between sites. The previous layers have already optimized the query, for example, by eliminating redundant expressions. However, this optimization is independent of fragment characteristics such as fragment allocation and cardinalities. In addition, communication operators are not yet specified. By permuting the ordering of operators within one query on fragments, many equivalent queries may be found.

Query optimization consists of finding the "best" ordering of operators in the query, including communication operators that minimize a cost function. The cost function, often defined in terms of time units, refers to computing resources such as disk space, disk I/Os, buffer space, CPU cost, communication cost, and so on. Generally, it is a weighted combination of I/O, CPU, and communication costs. Nevertheless, a typical simplification made by the early distributed DBMSs, as we mentioned before, was to consider communication cost as the most significant factor. This used to be valid for wide area networks, where the limited bandwidth made communication much more costly than local processing. This is not true anymore today and communication cost can be lower than I/O cost. To select the ordering of operators it is necessary to predict execution costs of alternative candidate orderings. Determining execution costs before query execution (i.e., static optimization) is based on fragment statistics and the formulas for estimating the cardinalities of results of relational operators. Thus the optimization decisions depend on the allocation of fragments and available statistics on fragments which are recorder in the allocation schema.

An important aspect of query optimization is join ordering, since permutations of the joins within the query may lead to improvements of orders of magnitude. One basic technique for optimizing a sequence of distributed join operators is through the semijoin operator. The

main value of the semijoin in a distributed system is to reduce the size of the join operands and then the communication cost. However, techniques which consider local processing costs as well as communication costs may not use semijoins because they might increase local processing costs. The output of the query optimization layer is a optimized algebraic query with communication operators included on fragments. It is typically represented and saved (for future executions) as a distributed query execution plan.

DISTRIBUTED QUERY EXECUTION :

The last layer is performed by all the sites having fragments involved in the query. Each subquery executing at one site, called a local query, is then optimized using the local schema of the site and executed. At this time, the algorithms to perform the relational operators may be chosen. Local optimization uses the algorithms of centralized systems.

The goal of distributed query processing may be summarized as follows: given a calculus query on a distributed database, find a corresponding execution strategy that minimizes a system cost function that includes I/O, CPU, and communication costs. An execution strategy is specified in terms of relational algebra operators and communication primitives (send/receive) applied to the local databases (i.e., the relation fragments). Therefore, the complexity of relational operators that affect the performance of query execution is of major importance in the design of a query processor.

TRANSACTION AND COMPUTATION MODEL

- Page Model
- Object Model

Page Model

Syntax

Atransactiont is a partial order of steps (actions) of the formr(x) or w(x), where x∈D and reads and writes as well as multiple writes applied to the same object are ordered.

We write t = (op, <),

for transaction t with step set op and partial order <.

Example:r(s) w(s) r(t) w(t)

Semantics

Interpretation of j^{th} step, p_j , of t:

If p_j =r(x), then interpretation is assignment v_j:= x to local variable v_j.

If p_j=w(x), then interpretation is assignment x := $f_j(v_{j1}, ..., v_{jk})$.

with unknown function $f_j$and j_1, ..., j_k denoting t‘s prior read steps.

Object Model

A transaction t is a (finite) tree of labeled nodes with

- the transaction identifier as the label of the root node,
- the names and parameters of invoked operations as labels of inner nodes, and

- page-model read/write operations as labels of leaf nodes, along with a partial order < on the leaf nodes such that for all leaf-node operations p and q with p of the form w(x) and q of the form r(x) or w(x) or vice versa, we have

$$p<q \quad \vee \quad q<p$$

Special case: layered transactions(all leaves have same distance from root)
Derived inner-node ordering: a < b ifall leaf-node descendants of a precede all leaf-node descendants of b
Example: DBS Internal Layers

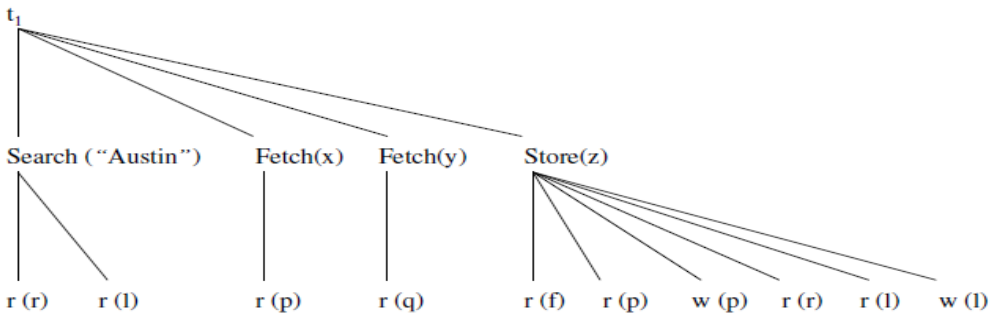


Figure 5.6 DBS Internal Layers

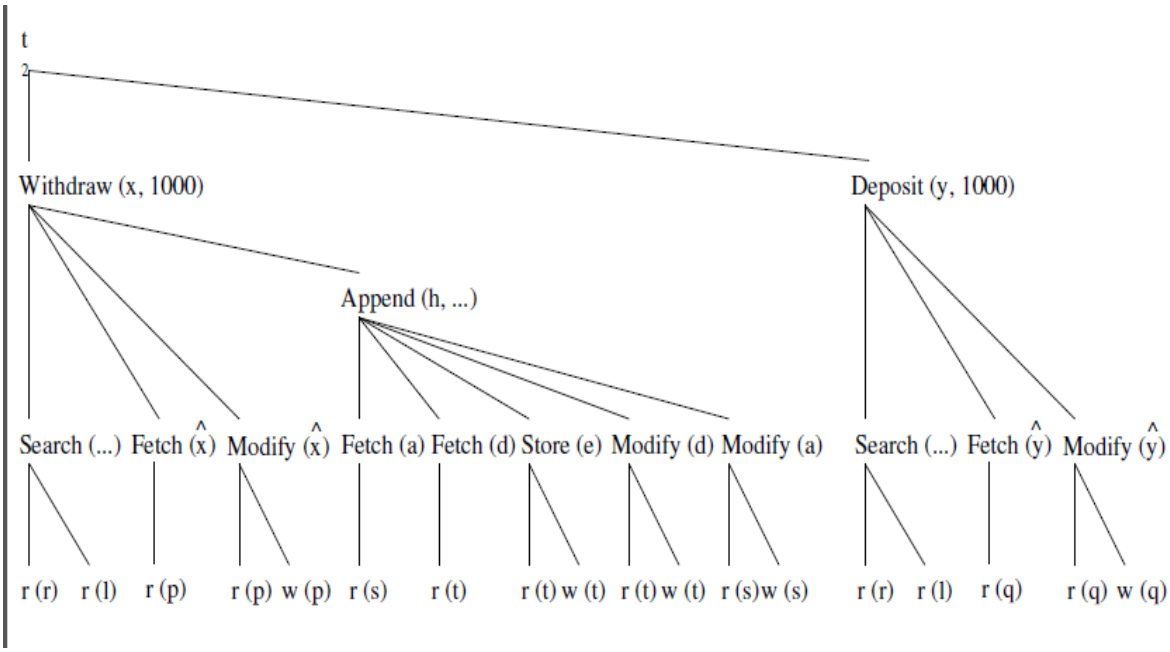


Figure 5.7 Business Objects

MULTIDATABASE CONCURRENCY CONTROL :

Concurrency controlin hierarchical MDBSs .In this section, we present a framework for the design of concurrency control mechanisms for hierarchical MDBSs. In a hierarchical MDBS, for the global schedule S to be serializable, the projection of S onto data items in each domain $D \in \Delta$ (that is, S D) must be serializable. However, as illustrated in the following example, ensuring serializability of S D, for each $D \in \Delta$, is not sufficient to ensure global serializability.

For example, in a schedule generated by a serialization-graph-testing (SGT) scheduler, it may not be possible to associate a serialization function with transactions. However, in such schedules, serialization functions can be introduced by forcing direct conflicts between transactions.

Let $\tau' \subseteq \tau$ be a set of transactions in a schedule S. If each transaction in τ' executed a conflicting operation (say a write operation on data item ticket) in S, then the functions that maps a transaction Ti $\in \tau'$ to its write operation on ticket is the serialization function for the transactions in S with respect to the set of transactions τ'. Associating serialization functions with global transactions makes the task of ensuring serializability of S D relatively simple. Since at each local DBMS the order in which transactions that are global with respect to the local DBMSs are serialized is consistent with the order in which their serSk operations execute, serializability of S D can be ensured by simply controlling the execution order of the serSk operations belonging to the transactions global with respect to the local DBMSs. To see how this can be achieved, for a global transaction Ti , let us denote its projection to its serialization function values over the local DBMSs as a transaction T˜ D i .

Formally, T˜ D i is defined as follows.

Definition 1. Let Ti be a transaction and D be a simple domain such that global(Ti , DBk), for some DBk, where child(DBk, D), T˜ D i is a restriction of Ti consisting of all the operations in the set {serSk (Ti) | Ti executes in DBk, and child(DBk, D)} Further, for the global schedule S, we define a schedule S˜D to be the restriction of S consisting of the set of operations belonging to transactions T˜ D i . Thus, S˜D = (τS˜D , $\prec$S˜D), where τS˜D = {T˜ D i | global(Ti , DBk) for some DBk, where child(DBk, D)}, and for all operations oq, or in S˜D, oq $\prec$S˜D or, iff oq $\prec$S or. In the schedule S˜D the conflict between operations is defined as follows:

Definition 2. Let S be a global schedule. Operations Sk (Ti) and Sl (Tj) in schedule S˜D, Ti / Tj , are said to conflict if and only if k = l. It is not too difficult to show that the serializability of the schedule S D can be ensured by ensuring the serializability of the schedule S˜D. Essentially, ensuring serializability of S˜D enforces a total order over global transactions (with respect to the local DBMSs), such that if Ti occurs before Tj in the total order, then serSk operation of Ti occurs before serSk operation of Tj for all sites sk at which they execute in common, thereby ensuring serializability of S D

Notice that operations in the schedule S˜D consist of only global transactions. Thus, since global transactions execute under the control of the MDBS software, the MDBS software can control the execution of the operations in S˜D to ensure its serializability, thereby ensuring serializability of S D. How this can be achieved – that is, how the MDBS software can ensure serializability of S˜D is a topic of the next section. Recall that the above-described mechanism for ensuring serializability of S D has been developed under the assumption that D is a simple domain. In the remainder of this section, we extend the mechanism suitably to ensure serializability of the schedule S D for an arbitrary domain D. One way we can extend the mechanism to arbitrary domains in hierarchical MDBSs is by suitably extending the notion of the serialization function to the set of domains.

Definition 3. Let D be any arbitrary domain in Δ. An extended serialization function is a function sf(Ti , D) that maps a given transaction Ti , and a domain D, to some operation of Ti that executes in D such that the following holds. For all Ti , Tj , if global(Ti , D), global(Tj , D), and Ti $*$ SD Tj , then sf(Ti , D) $\prec$SD sf(Tj , D). We refer to sf(Ti , D) as a serialization function of transaction Ti with respect to the domain D. To see how such a serialization function will aid us in ensuring serializability within a domain, consider a domain D /= DBk,

k = 1, 2, . . . , m. To develop the intuition, let us assume that the above-defined serialization function exists for transactions in every child domain of D, that is, for every Dk, where child(Dk, D). If such a serialization function can be associated with the child domains, we can simply use the mechanism developed for simple domains to ensure serializability of S D.

We will, however, have to appropriately extend our definitions of the transaction T˜ D i , and the schedule S˜D with respect to the newly defined serialization function. This is done below.

Definition 4. Let Ti be a transaction and D be a domain such that global(Ti , Dk) for some Dk, where child(Dk, D). T˜ D i is a restriction of Ti consisting of all the operations in the set {sf(Ti , Dk) | Ti executes in Dk, and child(Dk, D) }. As before, schedule S˜D is simply the schedule consisting of the operations in the transactions T˜ D i . That is, S˜D = (τS˜D , ≺S˜D), where τS˜D = {T˜ D i | global(Ti , Dk) for some Dk, where child(Dk, D)}, 158 and for all operations oq, or in S˜D, oq ≺S˜D or, iff oq ≺S or. Similar to the case of simple domain, two operations in S˜D, where D is an arbitrary domain, conflict if they are both serialization function values of different transactions over the same child domain.

Definition 5.Let S be a global schedule. Operations sf(Ti , Dk) and sf(Tj , Dl) in schedule S˜D, Ti / Tj , are said to conflict if and only if k = l. It it not difficult to see that similar to the case of simple domains, serializability of S D can be ensured, where D is an arbitrary domain, by ensuring the serializability of the schedule S˜D, under the assumption that, for all child domains Dk of D, the schedule S Dk is serializable and further a serialization function sf can be associated with transactions that are global with respect to Dk (see Lemma 1 in the appendix for a formal proof). In fact, this result can be applied recursively over the domain hierarchy to ensure serializability of the schedules S D for arbitrary domains D in hierarchical MDBSs. To see this, consider a hierarchical MDBS shown in Fig. 4. To ensure serializability of S D3 , it suffices to ensure serializability of the schedule S˜D3 , under the assumption that S D1 and S D2 are serializable and further that an appropriate serialization function sf can be associated with transactions that are global with respect to D1 and D2. In turn, serializability of S D1 (S D2) can be ensured by ensuring that the schedule S˜D1 (S˜D2) is serializable, under the assumption that S DB1 and S DB2 (S DB3 and S DB4) are serializable and further that an appropriate serialization function sf can be associated with transactions that are global with respect to DB1 and DB2 (DB3 and DB4). The recursion ends when D is a simple domain, since the child domains are local DBMSs and by assumption the schedule at each local DBMS is serializable. Thus, if we can associate an appropriate serialization function sf with transactions in each domain D ∈ Δ, we can ensure serializability of S D, by ensuring serializability of S˜D for all domains D ∈ Δ. Note that, for a domain D = DBk, the function sf is simply the function serSk introduced earlier. We now define the function sf for an arbitrary domain D ∈ Δ, which is done recursively over the domain ordering relation.

Definition 6. Let D be a domain and Ti be a transaction such that global(Ti , D). The serialization function for transaction Ti in domain D is defined as follows: sf(Ti , D) = serSk (Ti), if for some DBk, D = DBk. serS˜D (T˜ D i), if for all DBk, D /= DBk Let us illustrate the above definition of the serialization function using the following example. Example 3. Consider an MDBS environment consisting of local databases: DBMS1 with data item a, DBMS2 with data item b, DBMS3 with data item c, and DBMS4 with data item d. Let the domain ordering relation be as illustrated in Fig. 4. The set of domains: Δ = {DB1, DB2, DB3, DB4, D1, D2, D3}

MULTIDATABASE RECOVERY :

ReMT - A Recovery Strategy for MDBSs As already mentioned, reliability in MDBSs requires the design of two different types of protocols: commit and recovery protocols. A

commit protocol which enforces commit atomicity of global transactions. In this section, we will present a strategy, called ReMT, for recovering multidatabase consistency after failures, without human intervention. In MDBSs, recovering multidatabase consistency has a twofold meaning. First, for global transaction aborts, recovering multidatabase consistency means to undo the effects of locally committed subsequences belonging to the aborted global transactions from a semantic point of view. In addition, the effects of transactions which have accessed objects updated by aborted global transactions should be preserved (recall that, after the last operation of a subsequence, all locks held by the subsequence are released). For the other types of failures, recovering multi database consistency means to restore the most recent global transaction-consistent state. We say that a multi database is in a global transaction-consistent state, if all local DBMSs the effects of locally-committed subsequences. The ReMT strategy consists of a collection of recovery protocols which are distributed among the components of an MDBS. Hence, some of them are performed by the GRM, some by the servers and some are provided by the LDBMSs. We assume that every participating LDBMS provides its own recovery mechanism. Local recovery mechanisms should be able to restore the most recent transaction-consistent state of local databases after local failures. For each type of failure, we propose a specic recovery scheme. 6.1 Transaction Failures As seen before, we identify different kinds of transaction failures which may occur in a multidatabase environment. Each of them can be dealt with in a different manner. First, a particular global transaction may fail. This can be caused by a decision of the GTM or can be requested by the transaction itself. Second, a given subsequence of a global transaction may fail. In the following, we will propose recovery procedures to cope with failures of global transactions and subsequences.

A global transaction failure may occur for two reasons. The abort can be requested by the transaction or it occurs on behalf of the MDBS. The GTM can identify the reason which has caused the abort. This is because the GTM receives an abort operation from the transaction, whenever the abort is required by the transaction. We have observed that the recovery protocol for global transaction failures can be optimized if the following design decision is used: specific recovery actions should be dined for each situation in which a global transaction abort occurs. Therefore, we have designed recovery actions which should be triggered when the global transaction requires the abort, and recovery actions for coping with aborts which occur on behalf of the MDBS. Aborts Required by Transactions Since we assume that updates of a global transaction Gi may be viewed by other transactions, we cannot restore the database state which existed before the execution Gi, if Gi aborts. This implies that the standard transaction undo action cannot be used in such a situation. However, the effects of a global transaction must be somehow removed from the database, if it aborts. For that reason, we need a more adequate recovery paradigm for such an abort scenario. This new recovery paradigm should primarily focus on the fact that the effects of transactions which have accessed the objects updated by an aborted global transaction Gi and database consistency should be preserved, when removing the effects of Gi from the database. The key to this new recovery paradigm is the notion of compensating transactions. A compensating transaction CT \undoes" the effect of a particular transaction T from a semantic point of view. That means, CT does not restore the physical database state which existed before the execution of the transaction T . The compensation guarantees that a consistent (in the sense that all integrity constraints are preserved) database state is established based on semantic information, which is application-specific.

By definition, a compensating transaction CTi should be associated with a transaction Ti and may only be executed within the context of Ti. That means that the existence of CTi depends on Ti. In other words, CTi may only be executed, if Ti has been executed before. Hence, CTi must be serialized after Ti. We will assume that persistence of compensation is guaranteed, that is, once the compensating action has been started, it is completed successfully. For our purpose the concept of compensation is realized as follows. For a given transaction Gi consisting of subsequences SUBi;1, SUBi;2, :::, SUBi;n, a global compensating transaction CTi is dened, which in turn consists of a collection of local compensating transactions CTi;k, 0 < k n. Each local compensating transaction CTi;k is associated to the corresponding subsequence SUBi;k of transaction Gi. Of course, CTi;k must be performed at the same local site as does SUBi;k and must be serialized after SUBi;k. Now, we are in a position to describe the recovery strategy for aborts required by transactions. When the GS receives an abort request from a global transaction Gi, the GS forwards this operation to the GRM. The GRM reads the global log in order to identify which subsequences of Gi are still active. For each active subsequence, the GRM sends a local abort operation to the servers responsible for the execution of the subsequence. The GRM then waits for an acknowledgment from these servers conrming that the subsequences were aborted. After that, the GRM triggers the corresponding local compensating transactions for every subsequence which has already been locally committed. This information can be retrieved from the global log le. Operations of the compensating transactions are scheduled by the GS. Therefore, the execution of local compensating transactions will undo the effect of committed subsequences from a semantic point of view. Since we have assumed that the LDBMSs implement 2PL to enforce local serializability, the compensation mechanism described above satisfies the following requirement. A particular transaction T (subsequence or local transaction) running at a local system either views a database state reacting the effects of an updating subsequence SUBi;k or it accesses a state produced by the compensating transaction of SUBi;k, namely CTi;k. In other words, T cannot access objects updated by SUBi;k and by CTi;k. Such a constraint is required for preserving local database consistency. Thus far, we have assumed that the effect of any transaction can be removed from the database by means of a compensating transaction. However, not all transactions are compensatable. There are some actions, classified by Gray as real actions ,which present the following property: once they are done, they cannot be undone anymore. For some of these actions, the user does not know how they can be compensated, that is, the semantic of such compensating transactions is unknow. For instance, the action ring a missile cannot be undone. Moreover, the semantic of a compensating transaction for this action cannot be defined. For that reason, we say that transactions involving such real actions are not compensatable. In order to overcome this problem, we propose the following mechanism. The execution of local commit operations for non-compensatable subsequences should be delayed until the GTM receives a commit for the global transaction containing the non-compensatable subsequences. This mechanism requires that the following two conditions are satisfied. First, the user should specify which subsequences of a global transaction are non-compensatable3 . 3When it is not specified that a subsequence is non-compensatable, it is assumed that the subsequence is compensatable. This is a reasonable requirement, since our recovery strategy relies on a compensation mechanism. This latter mechanism presumes that the user defines compensation transactions, when he or she is designing transactions. Hence, the user can identify at this point, which subsequences of a global transaction may not be compensatable. Second, the information

identifying which subsequences are non-compensatable should be made available to the GTM. For instance, the GTM can be designed to receive this information as an input parameter of subsequences. The procedure of delaying the execution of local commit operations for non-compensatable subsequences can be realized according to the following protocol: 1. When the GTM receives the rst operation of a particular subsequence, it must identify whether the subsequence is compensatable. If the subsequence is non-compensatable, the GTM saves this information in the log record of the subsequence. The log record should be stored in the global log le. 2. If the GTM receives a local commit operation for a non-compensatable subsequence, it marks the log record of the subsequence stored in the global log with a ag. This ag captures the information that the local commit operation for the subsequence can be processed when the global transaction is to be committed. 3. Whenever the GTM receives a commit operation for a given global transaction Gi, it verifies in the global log if there are local commit operations to be processed for subsequences of Gi. This can be realized by reading the log records of all subsequences belonging to Gi. Following this protocol, we ensure that the effects of non-compensatable subsequences are reacted in the local databases only when the global transaction is to be committed. This eliminates the possibility of undoing the effect of such subsequences. Unfortunately, this mechanism has the following disadvantage. Locks held by non-compensatable subsequences can only be released when the global transaction completes its execution. Another drawback of the compensation approach is the specification of compensating transactions for interactive transactions as, for instance, design activities. As a solution for overcoming such a problem, we propose the following strategy. When an interactive global transaction G has to be aborted and G has some locally committed subsequences, the GTM reads the global log le in order to identify which subsequences of G were already locally committed. After that, the GTM notifies the user that the effects of some subsequences of G must be \manually" undone. The GRM informs which subsequences should be undone and what operations these subsequences have executed. Moreover, the GRM informs the user on which objects these operations have been performed. The user then starts another transaction in order to undo the effect of such subsequences. Objects updated by these subsequences may have been viewed by other global transactions. For that reason, the user must know which global transactions have read these objects. With this knowledge the user can notify other designers that the values of the objects x,y,z they have read (the GRM has provided this information) are invalid. Aborts on Behalf of the MDBS Usually, such aborts occur when global transactions are involved in deadlocks. Deadlocks are provoked by transactions trying to access the same objects with connecting locks. Committed subsequences have already released their locks. Besides this, they are not competing for locks anymore. Hence, operations of such subsequences can neither provoke nor be involved in deadlocks. This observation has an important impact in designing recovery actions to cope with transaction aborts required by the MDBS. It is not necessary to abort entire global transactions to resolve deadlock situations. Aborting active subsequences is sufficient. However, we need to replay the execution of the aborted subsequences in order to ensure commit atomicity. This implies that new results may be produced by the resubmission of the subsequences. In such a situation, the user must be noticed that the subsequences were aborted and, for that reason, they must be replayed, which may produce different results from those he/she has already received. With this knowledge, the user can decide to accept the new results or to abort the entire global transaction. Observe that, if the original values read by the

failed subsequences were not communicated to other subsequences (those reads may be invalid), the resubmission of the aborted subsequences will produce no inconsistency in the execution of entire global transaction. Such a requirement is reasonable in a multidatabase environment. Based on these observations, we propose the following strategy for dealing with global transaction aborts which occur on behalf of the MDBS. When the GTM (or another component of the MDBS) decides to abort a transaction Gi, the GRM must be informed that Gi has to be aborted. When the GRM receives this signal, it verifies in the global log which subsequences of Gi are still active. For each active subsequence, the GRM sends a local abort operation to the servers (through the GS, of course) responsible for the submission of these subsequences to the local systems. In the meantime, the GRM waits for an acknowledgment from the servers confrming the local aborts of the subsequences. Furthermore, the GRM sends to the user responsible for the execution of Gi the notification informing that some subsequences of Gi have to be aborted and they will be replayed. The GRM is able to inform the user which operations have to be reexecuted. The user can then decide to wait for the resubmission of the aborted subsequences or to abort the entire global transaction. If the user decides to abort the entire global transaction, the process of replaying the subsequences is cancelled and the recovery protocol for global transaction failure requested by the transaction is triggered. Otherwise, the recovery protocol for global transaction aborts which occur on behalf of the MDBS goes on as described below. When the GRM has received the acknowledgments that the subsequences were aborted in the local DBMSs, the GRM starts to replay the execution of each aborted subsequence SUBi;k. For that purpose, the GRM must read from the global log le the log record which contains information about the installation point of each subsequence to be replayed. This record can be identified by the elds SUBID and LRT. Observe that LRT must have the value `IP'.

As mentioned before, a subsequence of a particular global transaction may abort for many reasons. However, there are two situations of subsequence aborts which should be handled in a different manner. The rest situation is when the subsequence is aborted on behalf of the local DBMS. The second situation is when the subsequence decides to abort its execution. In this section, we describe a recovery method to deal with these two subsequence abort situations. Aborts on Behalf of the Local DBMS Typically, DBMSs decide to abort subsequences, when such subsequences are involved in local deadlocks. After such aborts, the effect of failed subsequences are undone by the LDBMSs. Locks held by the aborted subsequences are released. As soon as the server recognizes that a particular subsequence has been aborted by the local DBMS, the server reads the server log le and retrieves the log records of the aborted subsequence. The server stores a new log record for the subsequence with LRT=`ST' in the server log le. Moreover, the server sends a message to the GTM reporting that the subsequence has been aborted by the LDBMS. The GRM forces a record log of the failed subsequence to the global log le. By doing this, the new state of the subsequence is stored in the global log le as well. After that, the server forces a log record with the new state of the subsequence to the server log and starts the resubmission of the aborted subsequence. As already seen, new results may be produced by such a resubmission. However, we propose a notification mechanism which gives the user the necessary support to decide for accepting the new results or for aborting the entire global transaction. It is important to notice here that a given subsequence SUBi;k belonging to a global transaction Gi may have more than one log record with LRT=`ST' (in each log le) during the execution of Gi. For such a subsequence, only the last record with LRT=`ST' should be considered.

Aborts Required by Subsequences When the subsequence identifies some internal error condition (e.g., violation of some integrity constraints or bad input), it aborts its execution. Sometimes the resubmission of the subsequence is sufficient to overcome the error situation. However, we cannot guarantee that the subsequence will be committed after being resubmitted a certain number of times.

This is because the abort is caused when some internal error condition occurs (e.g. division by 0). Hence, it is impossible to predict whether or not the same problem will occur in a repeated execution of the subsequence. In this case, the solution is to abort the complete global transaction. The user or the GTM should be able to make such a decision. Observe that, when an internal error occurs, it is necessary that the subsequence reads new values (new input) and produces new results in order to overcome the internal error condition. Based on this observation, we propose the following actions for dealing with aborts required by subsequences. When the subsequence decides to abort its execution, an explicit abort operation is submitted to the GS, which in turn sends this operation to the GRM. The GRM then writes a new log record with LRT=`ST' for the subsequence in order to re ect its new state. Thereafter, the GRM forwards the abort operation to the server. In turn, the server forces a log record with the new state of the subsequence to the server log and submits the abort operation to the LDBMS. After the subsequence is aborted by the LDBMS, the GTM resubmits the aborted subsequence to the LDBMS. 6.2 Local System Failures Local DBSs reside in heterogeneous and autonomous computer systems (sites). When a system failure occurs at a particular site, we assume that the LDBMS is able to perform recovery actions in order to restore the most recent transaction-consistent state. These actions are executed outside the control of the MDBS. After an LDBMS completes the recovery actions, the interface server assumes the control of the recovery processing. While the server is executing its recovery actions, no local transaction can be submitted to the restarted DBMS. Before describing the strategy for recovering from local system failures, we need to defined states of a subsequence in a given server. A subsequence may present four different states in a server. A subsequence is said to be active, when no termination operation for the subsequence has been submitted to the local DBMS by the server. When the server submits a commit operation, the subsequence enters the to-be-committed state. If the commit operation submitted by the server has been successfully executed by the local DBMS, the subsequences enters the locally-committed state. When the subsequence aborts, it enters the locally-aborted state.

We assume that the GTM can identify when a given server has failed. The protocol for handling server failures is the following:

1.When the GTM recognizes that a server has failed, it aborts the execution of all active subsequences which were being executed in the failed server. Log records (with LRT=`ST') for the aborted subsequences are forced to the global log le in order to store the information that these subsequences have passed from the active to the aborted state. Moreover, the GTM stops submitting operations to that server. In order to decide what kind of recovery actions should be performed for to-be-committed subsequences, the GTM must wait until the server has been restarted, since the GTM must know whether the subsequence was successfully committed by the local DBMS.

2. After the server is restarted, it should trigger the following recovery procedures:

(a) The server log is sequentially read. For each subsequence which was active immediately before the occurrence of the failure, the server sends an abort operation to the local DBMS. If the subsequence was to-be-committed, the server may query the external interface of the local DBMS in order to know whether or not the subsequence was successfully committed by the local DBMS. The server then forwards this information to the GTM.

(b) The server log must be updated. For instance, if a particular to-be-committed subsequence was aborted by the local DBMS before the occurrence of the failure, the server writes a record in the server log le in order to capture this information.

(c) After the server log is read and updated, the server sends a message to the GTM informing that it is in operation. 3. When the GRM receives a message from the server reporting that it is operational, the GRM replays the aborted subsequences. After that, the recovery procedure for server failure is completed.

Communication Failures The components of an MDBS are interconnected via communication links. Typically, communication failures break the communication among some of the components of an MDBS. According to Figure 2, there may be two types of communication links in MDBSs. One type of link, which we call Server-LDBS link, connects servers to local systems. If the interface servers are not integrated with the GTM, that is, each server resides at a different site from the GTM site, the other type of link connects the GTM to servers. Such a communication link is denoted GTM-Server link. We propose different recovery strategies for handling failures in each type of communication link. In order to enable MDBSs to cope with communication failures, the following requirement must be satisfied. Each server in an MDBS must know the timeout period of the local DBMS with which the server is associated. We also assume that each server has its own timeout period and this timeout period is larger than the timeout period of the respective local DBMS. Failures in Server-LDBS links In such failures, the link between a particular local system and a server is broken. The local system and the server will continue to work correctly. Such a situation can lead the local system to abort the execution of some subsequences (which are being executed at the local system) by timeout. For coping with communication failures between a server and a local system, we propose the following strategy. If the communication link is reestablished before the timeout period of the local DBMS is reached, no recovery action is necessary. This is because no subsequence was aborted by timeout. In the case that the communication link is reestablished after the timeout period of the local DBMS is reached, but before the timeout period of the server, the following recovery actions should be performed by the server: 1. The server scans the server log le. During the scan process, the following recovery actions should be performed.

(a) For each subsequence which was active before the occurrence of the failure, the server executes recovery actions, since such subsequences were aborted by the LDBMS by timeout. These recovery actions are the same as those which should be performed for recovering from subsequence failures required by LDBMSs

(b) If the subsequence was to-be-committed, the server may query the external interface of the LDBMS in order to know whether the subsequence was successfully committed. In this case, the server performs actions to confirm the fact that the subsequence was committed (for instance, log records with LRT=`ST' must be forced to the server log and global log les). Otherwise, it considers the subsequence as locally aborted and performs actions for

recovering from subsequence failures required by LDBMSs. If the timeout period of the server is reached before the communication link is reestablished, the server sends a message to the GTM reporting that it cannot process subsequences anymore. After that, the GTM aborts the execution of all subsequences which were being executed in the failed server. Log records for the aborted subsequences are stored in the global log le with their new state (aborted). The GTM stops submitting operations to that server. If the communication link is reestablished before the timeout period of the GTM is reached, recovery actions for recovering from server failures are executed. If the timeout period of the GTM is reached before the Server-LDBS link is reestablished, the global log le is sequentially read. For each global transaction which has submitted a subsequence to the server whose Server-DBMS link is broken, the subsequence's log record with LRT=`ST' is read. If the subsequence is active or to-be-committed, the GRM aborts the global transaction. In this case, recovery actions for global transaction recovery should be triggered. Observe that a subsequence which was submitted to the server with a broken Server-LDBS link and has a to-be-committed state in the global log may have been committed by the local DBMS. In this case, after the link is reestablished, the server must be able to query the external interface of the LDBMS to know whether or not the subsequence was successfully committed. If the subsequence was committed, a compensating transaction for such a subsequence should be executed. Failures in GTM-Server links Of course, such a failure has only to be considered, if it is assumed that the interface servers reside at different sites from the GTM's site. When a failure in the GTM-Server link occurs, the link between the GTM and a server is broken. In order to enable MDBSs to cope with failures in GTM-Server links, we propose the strategy described below. Without loss of generality, consider that the link between the GTM and the server Serverk is broken. Serverk is associated with local system LDBSk. If the communication link is reestablished after the timeout period of LDBSk is reached, but before the timeout period of the server, the following actions are performed: The server log is sequentially read.

1. For each subsequence which was active before the failure, the server executes recovery actions for subsequence failures required by local DBMSs, since such transactions were aborted by the local DBMS (timeout).

2. If the subsequence was to-be-committed, the server may query the external interface of the local DBMS in order to know whether the subsequence was committed. In this case, the server performs the actions to react the fact that the subsequence was locally committed. Otherwise, it performs actions for recovering from subsequence failures required by local DBMSs. If the link is reestablished after the timeout period of Serverk, but before the timeout period of the GTM is reached, actions for recovering from server failures are started. If the timeout period of the GTM is reached before the link is reestablished, the GRM reads the global log in order to identify active global transactions which have submitted a subsequence to Serverk whose link with the GTM is broken. For each global transaction satisfying this condition, the GRM verifies the state of the subsequence submitted to Serverk. If the subsequence was active or to-be-committed, the GRM aborts the global transaction. Recovery actions for global transaction recovery should be triggered. A subsequence which has a to-be-committed state in the global log may have been committed by the local DBMS. In this case, after the communication link is reestablished, the server must be able to query the external interface of the LDBMS in order to know whether or not the subsequence was

successfully committed. If the subsequence was committed, a compensating transaction for such a subsequence should be executed.

OBJECT ORIENTATION AND INTEROPERABILITY:

Interoperating applications are often developed independently of each other in environments that may differ in the following dimensions:

• Locations

• Machine architectures

• Operating systems

• Programming languages

• Models of information. Applications can interoperate along the following dimensions:

• "Horizontal" peer-to-peer sharing of services and information, such as an editor invoking a spreadsheet processor to embed a spreadsheet in a document.

• "Vertical" cascading through levels of implementation. A student registration service may use a database service which in turn uses a file manager which uses a device driver.

• "Time-line" through the life cycle of an application. Enterprise modeling may be done in terms of one set of constructs which are translated into constructs of the application programming language which are compiled into constructs of the run-time environment. Or, a graphical language used to capture a user's conceptual model of a business domain is translated into a computer-executable simulation language, with the results of the simulation then being input either to an analysis tool to allow refinement of the simulation, or to a report generator to produce the final result. Internal Accession Date Only 2

• Others, e.g., the "viewpoints" of the ISO/CCITT Reference Model for Open Distributed Processing (RM-ODP) – Enterprise viewpoint – Information viewpoint – Computational viewpoint – Engineering viewpoint – Technology viewpoint Interoperation is concerned with such things as:

• Application interconnection: – Finding services and information in a distributed environment. – Coping with operational differences between requesters and providers of services, such as interface/communication protocols, synchronization, exception handling, work coordination, resource management, etc.

• Information compatibility.

OMA (OBJECT MANAGEMENT ARCHIRECTURE):

OMA is an architecture developed by the OMG (Object Management Group) that provides an industry standard for developing object-oriented applications to run on distributed networks. The goal of the OMG is to provide a common architectural framework for object-oriented applications based on widely available interface specifications.

The OMA reference model identifies and characterizes components, interfaces, and protocols that comprise the OMA. It consists of components that are grouped into application-oriented

interfaces, industry-specific vertical applications, object services, and ORBs (object request brokers). The ORB defined by the OMG is known more commonly as CORBA (Common Object Request Broker Architecture).

The Common Object Request Broker Architecture (CORBA) is a specification developed by the Object Management Group (OMG). CORBA describes a messaging mechanism by which objects distributed over a network can communicate with each other irrespective of the platform and language used to develop those objects.

There are two basic types of objects in CORBA. The object that includes some functionality and may be used by other objects is called a service provider. The object that requires the services of other objects is called the client. The service provider object and client object communicate with each other independent of the programming language used to design them and independent of the operating system in which they run. Each service provider defines an interface, which provides a description of the services provided by the client.

CORBA enables separate pieces of software written in different languages and running on different computers to work with each other like a single application or set of services. More specifically, CORBA is a mechanism in software for normalizing the method-call semantics between application objects residing either in the same address space (application) or remote address space (same host, or remote host on a network).

CORBA applications are composed of objects that combine data and functions that represent something in the real world. Each object has multiple instances, and each instance is associated with a particular client request. For example, a bank teller object has multiple instances, each of which is specific to an individual customer. Each object indicates all the services it provides, the input essential for each service and the output of a service, if any, in the form of a file in a language known as the Interface Definition Language (IDL). The client object that is seeking to access a specific operation on the object uses the IDL file to see the available services and marshal the arguments appropriately.

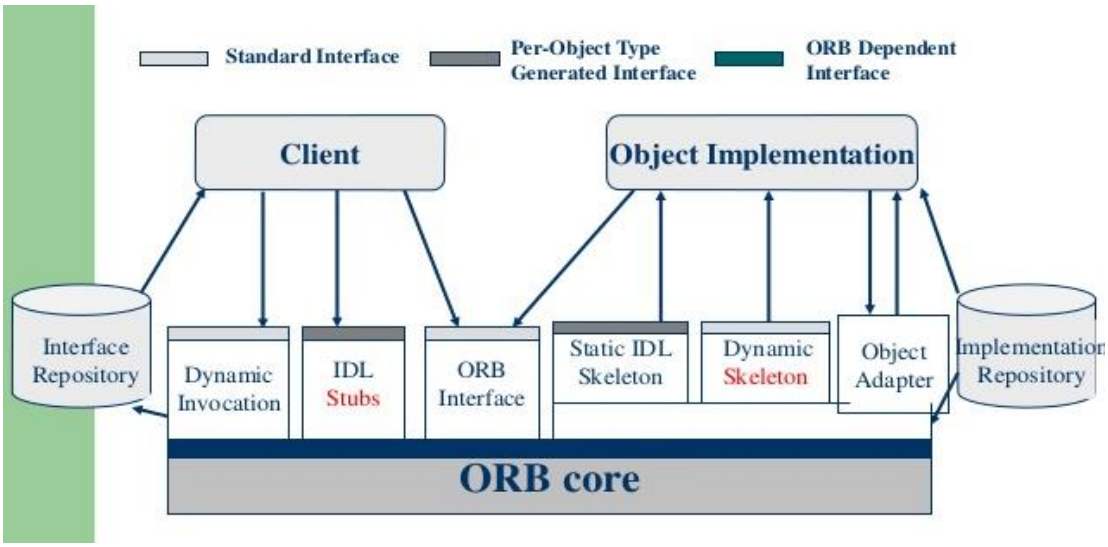


Figure 5.8 Object Management Archirecture

The CORBA specification dictates that there will be an object request broker (ORB) through which an application interacts with other objects. In practice, the application simply initializes the ORB, and accesses an internal object adapter, which maintains things like reference counting, object (and

reference) instantiation policies, and object lifetime policies. The object adapter is used to register instances of the generated code classes. Generated code classes are the result of compiling the user IDL code, which translates the high-level interface definition into an OS- and language-specific class base to be applied by the user application. This step is necessary in order to enforce CORBA semantics and provide a clean user process for interfacing with the CORBA infrastructure.

DISTRIBUTED COMPONENT MODEL:

DCOM is a programming construct that allows a computer to run programs over the network on a different computer as if the program was running locally. DCOM is an acronym that stands for Distributed Component Object Model. DCOM is a proprietary Microsoft software component that allows COM objects to communicate with each other over the network.

An extension of COM, DCOM solves a few inherent problems with the COM model to better use over a network:

Marshalling: Marshalling solves a need to pass data from one COM object instance to another on a different computer – in programming terms, this is called "passing arguments."

For example, if I wanted Zaphod's last name, I would call the COM Object LastName with the argument of Zaphod. The LastName function would use a Remote Procedure Call (RPC) to ask the other COM object on the target server for the return value for LastName(Zaphod), and then it would send the answer – Beeblebrox – back to the first COM object.

Distributed Garbage Collection: Designed to scale DCOM in order to support high volume internet traffic, Distributed Garbage Collection also addresses away to destroy and reclaim completed or abandoned DCOM objects to avoid blowing up the memory on webservers. In turn, it communicates with the other servers in the transaction chain to let them know they can get rid of the objects related to a transaction.Using DCE/RPC as the underlying RPC mechanism: To achieve the previous items and to attempt to scale to support high volume web traffic, Microsoft implemented DCE/RPC as the underlying technology for DCOM – which is where the D in DCOM came from.

How Does DCOM Work?

In order for DCOM to work, the COM object needs to be configured correctly on both computers – in our experience they rarely were, and you had to uninstall and reinstall the objects several times to get them to work.

The Windows Registry contains the DCOM configuration data in 3 identifiers:

CLSID – The Class Identifier (CLSID) is a Global Unique Identifier (GUID). Windows stores a CLSID for each installed class in a program. When you need to run a class, you need the correct CLSID, so Windows knows where to go and find the program.

PROGID – The Programmatic Identifier (PROGID) is an optional identifier a programmer can substitute for the more complicated and strict CLSID. PROGIDs are usually easier to read and understand. A basic PROGID for our previous example could be Hitchiker.LastName. There are no restrictions on how many PROGIDs can have the same name, which causes issues on occasion.

APPID – The Application Identifier (APPID) identifies all of the classes that are part of the same executable and the permissions required to access it. DCOM cannot work if the APPID isn't correct. You will probably get permissions errors trying to create the remote object, in my experience.

A basic DCOM transaction looks like this:

The client computer requests the remote computer to create an object by its CLSID or PROGID. If the client passes the APPID, the remote computer looks up the CLSID using the PROGID.

The remote machine checks the APPID and verifies the client has permissions to create the object.

DCOMLaunch.exe (if an exe) or DLLHOST.exe (if a dll) will create an instance of the class the client computer requested.

Communication is successful!

The Client can now access all functions in the class on the remote computer.

If the APPID isn't configured correctly, or the client doesn't have the correct permissions, or the CLSID is pointing to an old version of the exe or any other number of issues, you will likely get the dreaded "Can't Create Object" message.

DCOM vs. CORBA

Common Object Request Broker Architecture (CORBA) is a JAVA based application and functions basically the same as DCOM. Unlike DCOM, CORBA isn't tied to any particular Operating System (OS), and works on UNIX, Linux, SUN, OS X, and other UNIX-based platforms.

Neither proved secure or scalable enough to become a standard for high volume web traffic. DCOM and CORBA didn't play well with firewalls, so HTTP became the default standard protocol for the internet.