PEARSON

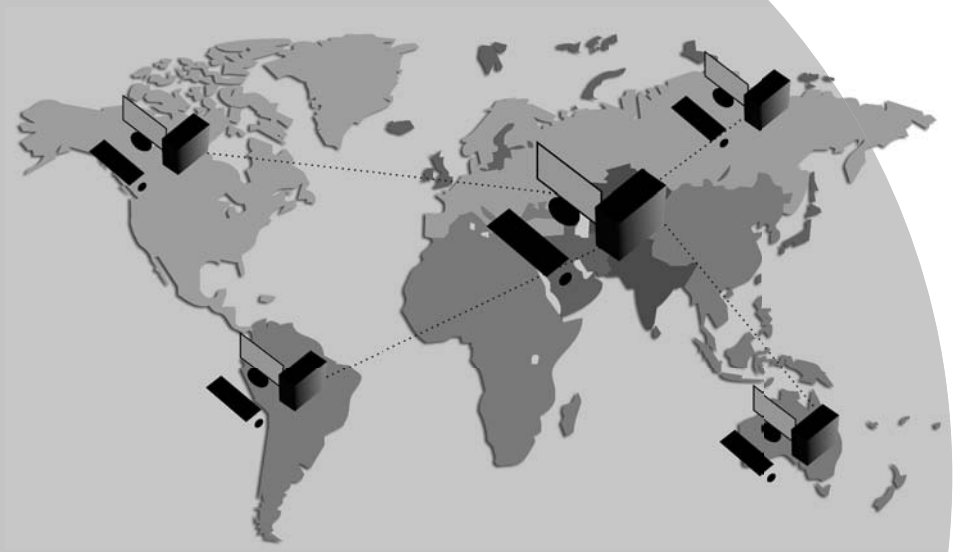
DISTRIBUTED DATABASE SYSTEMS

CHHANDA RAY

Distributed Database Systems

This page is intentionally left blank

Distributed Database Systems



Chhanda Ray
Assistant Professor and Head
Department of Computer Application
RCC Institute of Information Technology
Kolkata, West Bengal

PEARSON
Delhi • Chennai • Chandigarh

ISBN 978-81-317-2718-8

First Impression

Published by Dorling Kindersley (India) Pvt. Ltd., licensees of Pearson Education in South Asia

Head Office: 7th Floor, Knowledge Boulevard, A-8 (A), Sector 62, NOIDA, 201 309, UP, India.
Registered Office: 14 Local Shopping Centre, Panchsheel Park, New Delhi 110 017, India

Typeset by Integra Software Services Pvt. Ltd., Pondicherry, India.
Printed in India at Kumar Offset Printers.

Dedicated to:
My Parents
Husband and Children
Ankita and Krittik

This page is intentionally left blank

Contents

This page is intentionally left blank

Preface

The need for managing large volumes of data has led to the evolution of database technology. In recent years, the amount of data handled by a database management system (DBMS) has been increasing continuously, and it is no longer unusual for a DBMS to manage data sizes of several hundred gigabytes to terabytes. This is owing to the growing need for DBMSs to exhibit more sophisticated functionality such as the support of object-oriented, deductive, and multimedia-based applications. Also, the evolution of the Internet and the World Wide Web has increased the number of DBMS users tremendously. The motivation behind the development of centralized database systems was the need to integrate the operational data of an organization, and to provide controlled access to the data centrally. As the size of the userbase and data increases, traditional DBMSs that run on a single powerful mainframe system face difficulty in meeting the I/O and CPU performance requirements, as it cannot scale up well to the changes.

To achieve the performance requirements, database systems are increasingly required to make use of parallelism, which results in one of the two DBMS architectures: parallel DBMS or distributed DBMS. A parallel DBMS can be defined as a DBMS implemented on a tightly coupled multiprocessor system. The goals of parallelism are two-fold: speedup and scaleup. A distributed DBMS is a software system that provides the functionality for managing distributed databases and is implemented on a loosely coupled multiprocessor system. A distributed database is a logically interrelated collection of shared data that is physically distributed over a computer network. In the 1980s, a series of critical social and technological changes had taken place that affected the design and development of database technology. During recent times, the rapid developments in network and data communication technologies have changed the mode of working from a centralized to a decentralized manner. Currently, distributed DBMS is gaining acceptance as a more sophisticated modern computing environment.

This book focuses on the key concepts of the distributed database technology. In this book, all concepts related to distributed DBMS such as distributed database fundamentals, distributed database design, distributed DBMS architectures, distributed transaction management, distributed concurrency control, deadlock handling in distributed system, distributed recovery management, distributed query processing, data security and catalog management in distributed system and related concepts are briefly presented. Two popular distributed database systems, R* and SDD-1, are discussed as case studies. Moreover, the basic concepts of mobile databases and object-oriented DBMS are introduced. A full chapter is devoted to data warehousing, data mining and online analytical processing (OLAP) technology. In this book, each topic is illustrated with suitable examples. This book is intended for those who are professionally interested in distributed data processing including students and teachers of computer science and information technology, researchers, application developers, analysts and programmers.

Chapter 1 introduces the fundamentals of relational databases. The relational data model terminology, relational algebra, normalization and other related concepts are discussed in this chapter. Chapter 2 introduces the evaluation of distributed database technology and some key concepts of parallel database systems. Benefits of parallel databases and different alternative architectural model for parallel databases, namely, shared-memory, shared-disk, shared-nothing and hierarchial are described. In this chapter, parallel database design has also been discussed. The fundamentals of distributed database systems are presented in Chapter 3. This chapter introduces the benefits and limitations of distributed systems over centralized systems, objectives of a distributed system, components of a distributed system, types of distributed DBMS, and the functionality provided by a distributed database system.

In Chapter 4, an overview of computer networking is presented. In a distributed system, communications between different sites are established via high-speed computer networks. Thus, Chapter 4 presents different types of communication networks, network topologies, and different types of network protocols that are required for transfer of data among multiple sites. Chapter 5 focuses on distributed database design. In distributed database design, three important issues are considered, namely, fragmentation, replication and allocation. To improve availability, reliability and parallelism data items are replicated and stored at multiple sites of a distributed system. Different types of fragmentation, benefits of fragmentation, algorithms for fragmentation, data replication, and alternative strategies for data allocation are clearly explained in Chapter 5. One of the main objectives of distributed database design is to achieve transparency. Hence, different kinds of transparencies in distributed database design are also discussed.

The architecture of a system reflects the structure of the underlying system. Owing to the versatility of distributed database systems it is very difficult to describe a general architecture for distributed DBMS. In Chapter 6, distributed data independence and the reference architectures of distributed database systems such as client/server, peer-to-peer and multidatabase systems (MDBS) are briefly presented. This chapter also highlights the differences between federated MDBS and non-federated MDBS and the corresponding reference architectures. Chapter 7 focuses on distributed transaction management. In a distributed DBMS, it is necessary to maintain ACID property of local transactions as well as global transactions to ensure data integrity. A framework for transaction management in distributed system is illustrated in Chapter 7. Moreover, this chapter also introduces the different categories of transactions.

In a distributed environment, concurrent accesses to data are allowed in order to increase performance, but the database may become inconsistent owing to concurrent accesses. To ensure consistency of data, distributed concurrency control techniques are described in Chapter 8. In this chapter, degrees of consistency and distributed serializability are also discussed. Different locking protocols and timestamp-based protocols used for achieving serializability are clearly explained in Chapter 8. Chapter 9 focuses on deadlock handling in a distributed environment. In this chapter, different deadlock detection algorithms and recovery from deadlock are presented in detail.

In a distributed environment, different kinds of failures may occur such as node failure, link failure and network partition, in addition to the other possible failures in a centralized environment. Two popular distributed recovery protocols, namely, two-phase commit protocol (2PC) and three-phase commit protocol (3PC), which ensure the consistency of data in spite of failures, are introduced in Chapter 10. Check pointing mechanism and cold restart are also described in Chapter 10. Chapter 11 presents distributed query processing techniques. This chapter briefly introduces distributed query transformation, distributed query optimization, and the different join strategies and distributed query optimization algorithms used for distributed query processing.

Chapter 12 focuses on distributed security control mechanisms used for ensuring security of distributed data. In this chapter, various security control techniques, such as authorization and protection and view management are described in detail. In addition, semantic data control and global system catalog management are also discussed in Chapter 12. Chapter 13 focuses on the preliminary concepts of mobile databases and object-oriented database management systems. Chapter 14 introduces the concepts of R* distributed database system and SDD-1 distributed database system, case studies from emerging technologies. The concluding chapter, Chapter 15, is devoted to the fundamentals of data warehousing, data mining and OLAP technologies.

Chapter summary, review questions, and exercises (including multiple choice questions) are included at the end of each chapter to aid easy referencing and recollection of the main points.

Chhanda Ray

1 Overview of Relational DBMS

This chapter presents an overview of relational databases. Normalization is a very important concept in relational databases. In this chapter, the process of normalization is explained with suitable examples. Integrity constraints, another important concept in relational data model, are also briefly discussed here. Relational algebra and the overall concept of relational database management system have also been illustrated in this chapter with appropriate examples.

The organization of this chapter is as follows: Section 1.1 represents the basic concepts of relational databases. In Section 1.2, integrity constraints are described with examples. The process of normalization is illustrated with examples in Section 1.3. Section 1.4 introduces the formal database language relational algebra, and overall concept of relational DBMS is discussed in Section 1.5.

1.1 Concepts of Relational Databases

Database management systems (DBMSs) have evolved to overcome the limitations of file-based approach, and now it is an integral part of our day-to-day life. Database management systems have wideranging applications in almost all areas, which include business, education, engineering, medicine, law, and modern computing environments. A **database** is a collection of interrelated data, and a **DBMS** is a set of programs that are used to store, retrieve, and manipulate the data in the database. The main objective of a DBMS is to integrate organizational data into a centralized location and to provide secure shared access to data. The integration of organizational data at a central location controls data redundancy and thereby ensures data consistency. An important benefit of DBMS is that it provides **data independence**. In traditional file-based approach, data and programs that access the data are dependent on each other; thus, changing of one affects the other. In a DBMS, data independence is provided by its three-tier architecture.

The **relational data model** was first introduced by E.F. Codd of IBM Research in 1970. The relational model is based on the concept of mathematical relation, and it uses set theory and first-order predicate logic as its theoretical basis. In relational data model, the database is expressed as a collection of **relations**. A relation is physically represented as a **table** in which each row corresponds to an individual **record**, and each column corresponds to an individual **attribute**, where attributes can appear in any order within the table. In formal relational data model terminology, a row is called a **tuple**, a column header is called an attribute, and the table is called a relation. The **degree** of a relation is represented by the number of attributes it contains, while the **cardinality** of a relation is represented by the number of tuples it contains. For each attribute in the table there is

a set of permitted values, known as the **domain** of the attribute. Domains may be distinct for each attribute or two or more attributes may be defined in the same domain. A named relation defined by a set of attributes and domain name pairs is called **relational schema**. Formally, a relation can be defined as follows:

A relation R defined over a set of n attributes $A_1, A_2,\ldots, A_n$ with domains $D_1, D_2,\ldots, D_n$ is a set of n tuples denoted by $<d_1, d_2,\ldots, d_n>$ such that $d_1 \in D_1, d_2 \in D_2,\ldots, d_n \in D_n$. Hence, the set $\{A_1{:}D_1, A_2{:}D_2,\ldots, A_n{:}D_n\}$ represents the relational schema.

By definition, all tuples within a relation should be distinct. To maintain the uniqueness of tuples in a relation, the concept of different relational keys is introduced into the relational data model terminology. The **key** of a relation is the non-empty subset of its attributes that is used to identify each tuple uniquely in a relation. The attributes of a relation that make up the key are called **prime attributes** or **key attributes**. In contrast, the attributes of a relation that do not participate in key formation are called **non-prime attributes** or **non-key attributes**. The superset of a key is usually known as **superkey**. A superkey uniquely identifies each tuple within a relation, but a superkey may contain additional attributes. A **candidate key** is a superkey such that no proper subset of it is a superkey within the relation. Thus, a minimal superkey is called candidate key. In general, a relational schema can have more than one candidate key among which one is chosen as **primary key** during design time. The candidate keys that are not selected as primary key are called **alternate keys**.

The information stored in a table may vary over time, thereby producing many instances of the same relation. Thus, the term relation refers to a **relation instance**. The value of an attribute in a relation may be undefined. This special value of the attribute is generally expressed as **null value**. The concept of relational database is illustrated in the following example.

Example 1.1

Let us consider an organization that has a number of departments where each department has a number of employees. The entities that are considered here are employee and department. The information that is to be stored in the database for each employee are employee code (unique for each employee), employee name, designation of the employee, salary, department number of the employee and voter identification number (unique for each employee). Similarly, department number (unique for each department), department name and department location for each department are to be stored in the database. The relational schemas for this database can be defined as follows:

Employee (*emp-id*, ename, designation, salary, deptno, voter-id)

Department (*deptno*, dname, location)

In Employee relation there are six attributes, and in Department relation there are three attributes. Thus, the degree of the relations Employee and Department are 6 and 3, respectively. For each attribute in the above relations, there is a corresponding domain, which need not necessarily be distinct. If D_1 is the domain of the attribute emp-id in Employee relation, then all values of emp-id in employee records must come from the domain D_1. The primary keys for the relations Employee and Department are emp-id and deptno, respectively. In Employee relation, there are two candidate keys: emp-id and voter-id. In this case, voter-id is called alternate key. These relations are physically represented in tabular format in figure 1.1.

Employee

emp-id	ename	Designation	salary	deptno	voter-id
E01	J. Lee	Manager	80,000	10	V01
E02	S. Smith	Asst. Manager	60,000	20	V03
E03	A. Roy	Analyst	70,000	10	V04
E04	B. Sharma	Programmer	20,000	30	V02
E05	D. Davis	Manager	75,000	30	V05
E06	M. Jones	Consultant	85,000	40	V07
E07	A. Sasmal	Manager	80,000	20	V06
E08	R. Miller	Consultant	75,000	10	V08
E09	S. Sen	Analyst	72,000	40	V10
E10	D. Dastidar	Asst. Manager	50,000	40	V09

Department

deptno	dname	Location
10	Development	Lake side
20	Production	Near playground
30	Maintenance	Lake side
40	HR	Near garden
50	Sales	Near playground

Fig. 1.1 Sample Database Schemas

1.2 Integrity Constraints

The term data integrity, which is expressed by a set of integrity rules or constraints, refers to the correctness of data. A constraint specifies a condition and a proposition that must be maintained as true. For instance, a **domain constraint** imposes a restriction on the set of values of an attribute, which indicates that the values must be projected only from the associated domain. In relational data model, there are two principal integrity constraints known respectively as entity integrity constraint and referential integrity constraint.

Entity integrity constraint specifies that each attribute of a primary key in a relation must be not null in a relational data model. This integrity rule ensures that no subset of the primary key is sufficient to provide uniqueness of tuples. To define referential integrity constraint it is necessary to define the foreign key first. A **foreign key** is an attribute or a set of attributes within one relation that matches the candidate key of some other relation (possibly the same relation). In relational databases, sometimes it requires to ensure that a value that appears in one relation for a given attribute (or a given set of attributes) also appears for a certain attribute (or a certain set of attributes) in another relation. This is known as **referential integrity constraint**, which is expressed in terms of foreign key. Referential integrity constraint specifies that if a foreign key exists in a relation either the foreign

key value must match a candidate key value of some tuples in another relation (may be the same relation), or the foreign key value must be null. The relation that contains the foreign key is called **referencing relation**, and the relation that contains the candidate key is called **referenced relation**.

Database modifications may violate the referential integrity constraint. The following anomalies can occur in a relational database owing to referential integrity constraint:

- **Repetition anomaly** – If a foreign key exists between two relations certain information may be repeated unnecessarily. This is known as repetition anomaly. Repetition anomaly results in wastage of storage space.
- **Insertion anomaly** – This anomaly may occur during the insertion of new records into the referencing relation. Therefore, to preserve referential integrity constraint, it must be ensured that the foreign key values of new records in the referencing relation are matched with the candidate key values in the referenced relation.
- **Update anomaly** – This anomaly may occur during updating of existing records in both referencing and referenced relations. Hence, before updating, it must be ensured that there is no violation of referential integrity constraint.
- **Deletion anomaly** – During deletion of existing records from referenced relation, this anomaly may occur. To maintain referential integrity, either all records from the referencing relation whose foreign key values match with candidate key values of deleted records should be deleted or foreign key values should be updated.

Example 1.2

Let us consider the same relations Employee and Department, which were described in example 1.1. In this case, the attribute deptno, candidate key of the Department relation, is a foreign key of the Employee relation, and it references the attribute deptno in Employee relation. Hence, Employee relation is called referencing relation and Department relation is called referenced relation.

To preserve referential integrity between these two relations, during insertion of new records into the Employee relation it must be ensured that the department numbers mentioned in these new employee records are valid and they match with the existing department numbers of the Department relation. While deleting records from the Department relation, either all records in the Employee relation whose department numbers match with department numbers of the deleted records of Department relation should be deleted, or department numbers of those records in Employee relation should be updated. While updating records in Employee relation, it must be ensured that the department numbers specified in employee records are valid. Similarly, no violations of referential integrity constraint must be made while updating records in Department relation.

1.3 Normalization

The process of normalization was first developed by E.F. Codd in 1972. In relational databases, **normalization** is a step-by-step process of replacing a given relation with a collection of successive relations to achieve simpler and better data representation. The main objective of normalization is to eliminate different anomalies that may occur owing to referential integrity constraint [described in **Section 1.2**] and to identify a suitable set of relations in database design. In normalization, a series of tests are performed to determine whether a relation satisfies the requirements of a given

normal form or not. Initially, three normal forms were introduced, namely, 1NF (First normal form), 2NF (Second normal form) and 3NF (Third normal form). In 1974, a stronger version of 3NF called Boyce–Codd normal form (BCNF) was developed. All these normal forms are based on functional dependencies among the attributes in a relation. Later, higher normal forms such as 4NF (Fourth normal form) and 5NF (Fifth normal form) were introduced. 4NF is based on multi-valued dependencies, while 5NF is based on join dependencies.

The process of normalization is a formal method in which a single relation is decomposed into a number of relations based on the primary key or candidate key and the functional dependencies among the attributes. The single relation for the entire database is called **global relation** or **Universal relation**. In normalization process, two criteria are to be satisfied during decomposition. The first criterion is **lossless** decomposition, and the second is **dependency-preserving** decomposition. The first criterion ensures no loss of information during decomposition. If it is possible to reconstruct the original relation by joining decomposed relations, then it is called lossless decomposition. Decomposition is said to be dependency-preserving, if the union of the functional dependencies in decomposed relations is equivalent to the closure of the functional dependencies of the original relation.

1.3.1 Functional Dependencies

Functional dependencies are used to decompose relations in the normalization process. Formally, a **functional dependency** can be defined as follows:

In a relation R, an attribute or a set of attributes Y is said to be functionally dependent on another attribute or a set of attributes X, denoted by $X \rightarrow Y$, if each value of X is associated with exactly one value of Y. Thus, X functionally determines Y.

In a relation, non-prime attributes are functionally dependent on the key attribute. In relational data model, various types of functional dependencies may exist, which are listed in the following:

(i) Full functional dependency – In a relation R, a functional dependency of the form $X \rightarrow Y$ is said to be full, if the dependency no more sustains when any attribute is removed from X (left-hand side of the arrow of functional dependency).

(ii) Partial functional dependency – In contrast with full functional dependency, in a relation R, a functional dependency of the form $X \rightarrow Y$ is said to be partial, if the dependency persists even when any attribute is removed from X.

(iii) Trivial functional dependency – In a relation R, a functional dependency of the form $X \rightarrow Y$ is called trivial functional dependency, if Y is a subset of X (not necessarily a proper subset), that is, $Y \subseteq X$. For instance, $A \rightarrow A$ is a trivial functional dependency.

(iv) Non-trivial functional dependency – In a relation R, a functional dependency of the form $X \rightarrow Y$ is said to be non-trivial, if Y is not a subset of X, that is, $Y \not\subset X$.

(v) Transitive functional dependency – Consider a relation R in which Y is functionally dependent on X, and Z is functionally dependent on Y, that is, $X \rightarrow Y$ and $Y \rightarrow Z$. Then, Z is functionally dependent on X via rules of transitivity. In this case, the functional dependency $X \rightarrow Z$ is called transitive functional dependency.

(vi) Multi-valued dependency – Consider a relation R in which X, Y and Z are three different attributes (or sets of attributes) such that $X \subset R$, $Y \subset R$ and $Z \subset R$. In this relation R, if for every value of X there exists a set of values for Y and Z, but the set of values for Y and Z are not dependent on each other, then Y and Z are multi-valued dependent on X. That is, $X \twoheadrightarrow Y$ and

$X \twoheadrightarrow Z$. Multi-valued dependency is further classified into trivial and non-trivial categories. In a relation R, a multi-valued dependency of the form $X \twoheadrightarrow Y$ is said to be a **trivial multi-valued dependency**, if Y is a subset of X or $X \cup Y = R$. On the other hand, a multi-valued dependency of the form $X \twoheadrightarrow Y$ is said to be a **non-trivial multi-valued dependency** if $Y \not\subset X$ and $X \cup Y \neq R$.

In this context, it is necessary to mention that every multi-valued dependency is also a functional dependency, but the reverse is not necessarily true. The concept of join dependencies is not discussed here.

Example 1.3

In this example, the **Employee** relation, which was defined in example 1.1, is considered. Assume that (*emp-id*, *voter-id*) is a composite key of the Employee relation. Hence, all non-prime attributes of the Employee relation are functionally dependent on the key attribute (*emp-id*, *voter-id*).

Thus, emp-id,voter-id → ename, designation, salary, deptno.

The above functional dependency is not a full functional dependency, because even if voter-id or emp-id attribute is removed from the composite key attribute, the dependency persists. In this case, the functional dependency is partial. The above functional dependency is not trivial, because the attributes on the right-hand side of the arrow is not a subset of the attributes on the left-hand side of the arrow. Hence, the above functional dependency is non-trivial.

Example 1.4

Let us consider the following relational schema, which represents the course information offered by an institution:

Course (*course-id*, course-name, duration, fees, coordinator)

In the above Course relation, course-id is the key attribute. In this relation, course fees are dependent on course durations and course durations are dependent on course names.

Therefore, course-name → duration and

duration → fees

Hence, the attribute fees is functionally dependent on the attribute course-name via rules of transitivity. In this case, course-name → fees is a transitive functional dependency.

Example 1.5

In this example, the relational schema Emp-proj is considered, which is defined as follows:

Emp-proj (emp-id, project-no, hobby)

Assume that in the above Emp-proj relation, one employee can work in multiple projects, and one employee can have more than one hobby. Hence, for each value of the attribute emp-id, there exists a set of values for the attributes project-no and hobby in the Emp-proj relation, but the set of values for these multi-valued attributes are not dependent on each other. In this case, the attributes project-no and hobby are multi-valued dependent on the attribute emp-id. These multi-valued dependencies are listed in the following:

emp-id ↠ project-no and

emp-id ↠ hobby.

The Emp-proj relation is illustrated in figure 1.2.

Emp-proj

Emp-id	project-no	Hobby
E01	P01	Reading
E01	P01	Singing
E02	P02	Swimming
E02	P04	Reading
E02	P02	Reading
E02	P04	Swimming

Fig. 1.2 An example of Multi-valued Dependency

1.3.2 Normal Forms

In each normal form, there are some known requirements, and a hierarchical relationship exists among the normal forms. Therefore, if a relation satisfies the requirements of 2NF, then it also satisfies the requirements of 1NF. A relation is said to be in **1NF**, if domains of the attributes of the relation are atomic. In other words, a relation that is in 1NF should be flat with no repeating groups. A relation in 1NF must not contain any composite attribute or multi-valued attribute. To convert a relation to 1NF, composite attributes are to be represented by their component attributes, and multi-valued attributes are to be represented by using separate tables.

A relation is in **2NF**, if it is in 1NF, and all non-prime attributes of the relation are fully functionally dependent on the key attribute. To convert a relation from 1NF to 2NF all partial functional dependencies are to be removed.

A relation is said to be in **3NF**, if it is in 2NF, and no non-prime attribute in the relation is transitively dependent on the key attribute. To convert a relation from 2NF to 3NF all transitive dependencies are to be removed.

A stronger version of 3NF is BCNF. Before defining BCNF, it is necessary to introduce the concept of determinant. **Determinant** refers to the attribute or group of attributes on the left-hand side of the arrow of a functional dependency. For example, in the functional dependency $X \rightarrow YZ$, X is called the determinant. A relation is in **BCNF**, if it is in 3NF, and every determinant is a candidate key.

A relation is said to be in **4NF**, if it is in BCNF and contains no non-trivial multi-valued dependency. To achieve 4NF, all non-trivial multi-valued dependencies are to be converted to trivial multi-valued dependencies.

Relations that satisfy the requirements of 5NF are very rare in real-life applications. So, these are not discussed here.

Example 1.6

Let us consider the following relational schema, which represents the student information:

Student (*sreg-no*, sname, address, course-id, cname, duration, fees)

In this relation, sreg-no is the key attribute and address is a composite attribute that consists of the simple attributes street and city. Further, each course has a unique course identification number and a number of students can enroll for each course. The set of functional dependencies that exists among the attributes of the student relation are listed below:

sreg-no → sname, address, course-id

course-id → cname, duration, fees

The above relation does not satisfy the requirements of 1NF, because it contains a composite attribute address. To transform this relation into 1NF, the composite attribute address is to be represented by its components, street and city. The student table depicted in figure 1.3 is in 1NF.

Student

sreg-no	sname	street	city	course-id	cname	duration	Fees
S01	Amit	Rowdon st.	Kolkata	C01	B.Tech	4	120,000
S02	John	Circus Row	Delhi	C01	B.Tech	4	120,000
S03	Rohit	Park st.	Kolkata	C03	B.Pharm	4	100,000
S04	Ankit	Lake Road	Kolkata	C04	M.Tech	2	50,000
S05	Sachin	Akbar Road	Delhi	C02	MBA	3	150,000
S06	Anuj	Main Road	Chennai	C04	M.Tech	2	50,000
S07	Sagar	Jhu Road	Mumbai	C02	MBA	3	150,000

Fig. 1.3 Normalized Student Relation in 2NF

The above relation is also in 2NF because all non-prime attributes are fully functionally dependent on the key attribute sreg-no. Now, the Student relation does not satisfy the requirements of 3NF, because it contains transitive dependencies. To convert this relation into 3NF, it is decomposed into two relations STU_1 and STU_2, respectively as shown in figure 1.4

STU_1

sreg-no	sname	street	city	course-id
S01	Amit	Rowdon st.	Kolkata	C01
S02	John	Circus Row	Delhi	C01
S03	Rohit	Park st.	Kolkata	C03
S04	Ankit	Lake Road	kolkata	C04
S05	Sachin	Akbar Road	Delhi	C02
S06	Anuj	Main Road	Chennai	C04
S07	Sagar	Jhu Road	Mumbai	C02

STU_2

course-id	cname	duration	fees
C01	B.Tech	4	120,000
C02	MBA	3	150,000
C03	B.Pharm	4	100,000
C04	M.Tech	2	50,000

Fig. 1.4 Normalized Student Relation in BCNF

The course-id attribute of STU_2 relation is a foreign key of the relation STU_1. After decomposition, the above relations are also in BCNF, because the determinants sreg-no and course-id are candidate keys.

Example 1.7

In this example, the Emp-proj relation of Example 1.5 is considered. This relation contains two multi-valued dependencies as follows:

emp-id ↠ project-no and

emp-id ↠ hobby.

By definition of 4NF, the Emp-proj relation is not in 4NF because it contains non-trivial multi-valued dependencies. To convert this relation into 4NF, it is decomposed into two relations PROJ1 and PROJ2 as illustrated in figure 1.5.

PROJ1

emp-id	**project-no**
E01	P01
E02	P02
E02	P04

PROJ2

emp-id	**hobby**
E01	Reading
E01	Singing
E02	Swimming
E02	Reading

Fig. 1.5 An example of 4NF

In this case, the attribute emp-id of PROJ2 relation is a foreign key of the relation PROJ1.

1.4 Relational Algebra

Relational algebra is a high-level procedural language that is derived from mathematical set theory. It consists of a set of operations that are used to produce a new relation from one or more existing relations in a database. Each operation takes one or more relations as input and produces a new relation as output, which can be used further as input to another operation. This allows expressions to be nested in the relational algebra, which is known as **closure.**

In 1972, Codd originally proposed eight operations of relational algebra, but several others have also been developed subsequently. The five fundamental operations of relational algebra are selection, projection, Cartesian product, union and set difference. The first two among these are unary operations and the last three are binary operations. In addition, there are three derived operations of relational algebra, namely, join, intersection and division, which can be represented in terms of the five fundamental operations.

1.4.1 Selection Operation

The selection operation operates on a single relation and selects those tuples from the relation that satisfies a given predicate. It is denoted by the symbol σ (sigma), and the predicate appears as a subscript of it. The predicate may be simple or complex. Complex predicates can be represented by using the logical connectors ∧(AND), ∨(OR) and ¬ (NOT). The comparison operators that are used in selection operation are <, >, =, ≠, ≤ and ≥.

Example 1.8

Consider the query "Retrieve all those students whose city is Kolkata and enrolled for the courses B.Tech or M.Tech", which involves the Student relation of Example 1.6. Using selection operation the above query can be represented as follows:

$$\sigma_{(city = \text{'Kolkata'}) \wedge ((cname = \text{'B.Tech'}) \vee (cname = \text{'M.Tech'}))} \textbf{(Student)}$$

The result of the query is shown in figure 1.6.

Student

sreg-no	sname	street	city	course-id	cname	duration	fees
S01	Amit	Rowdon st.	Kolkata	C01	B.Tech	4	120,000
S04	Ankit	Lake Road	Kolkata	C04	M.Tech	2	50,000

Fig. 1.6 Result of Selection Operation

1.4.2 Projection Operation

The projection operation works on a single relation and defines a new relation that contains a vertical subset of the attributes of the input relation. This operation extracts the values of specified attributes and eliminates duplicates. It is denoted by the symbol ∏.

Example 1.9

By using projection operation, the query "Retrieve registration number, name and course name for all students" can be expressed as:

$$\prod_{\text{sreg-no, sname, cname}} \textbf{(Student)}$$

The result of the query is shown in figure 1.7.

Student

sreg-no	sname	cname
S01	Amit	B.Tech
S02	John	B.Tech
S03	Rohit	B.Pharm
S04	Ankit	M.Tech
S05	Sachin	MBA
S06	Anuj	M.Tech
S07	Sagar	MBA

Fig. 1.7 Result of Projection Operation

1.4.3 Union Operation

The union operation is a binary operation, and it is denoted by the symbol $\cup$. The union operation of two relations R and S selects all tuples that are in R, or in S, or in both, eliminating duplicate tuples. Hence, R and S must be union-compatible. Two relations are said to be **union-compatible**, if they have the same number of attributes and each pair of corresponding attributes have the same domain.

1.4.4 Set Difference Operation

The set difference operation of two relations R and S, denoted by $R - S$, defines a new relation that consists of the tuples that are in R but not in S. Hence, R and S must be union-compatible. It is to be noted that $R - S \neq S - R$. This operation allows deletion of tuples from a relation.

1.4.5 Cartesian Product Operation

The Cartesian product operation is a binary operation, and it is denoted by the symbol $\times$. The Cartesian product of two relations R and S, denoted by $R \times S$, defines a new relation, which is the concatenation of each tuple of relation R with each tuple of relation S. If the relation R has n attributes and the relation S has m attributes, then there should be $(n + m)$ attributes in the resultant relation. It is possible that the two relations have attributes with the same name. In this case, to maintain the uniqueness within a relation, the attribute names are to be prefixed with the original relation names.

1.4.6 Intersection Operation

The intersection operation of two relations R and S, denoted by $R \cap S$, defines a new relation consisting of the set of tuples that are in both R and S. Hence, R and S must be union-compatible. In terms of the set difference operation, the intersection operation, $R \cap S$, can be represented as $R - (R - S)$.

1.4.7 Join Operation

The join operation is derived from the Cartesian product operation of relational algebra. We can say join operation is a combination of the operations selection, projection and Cartesian product. There are various forms of join operation such as theta join, equi join, natural join, outer join and semijoin. The outer join operation is further classified into full outer join, left outer join and right outer join.

(i) Theta join (θ) – The most common join operation is theta join. The theta join operation of two relations *R* and *S* defines a relation that contains tuples satisfying a join predicate from the Cartesian product of *R* and *S*. The join predicate is defined as $R.A1\ \theta\ S.B1$, where θ may be one of the comparison operator <, >, =, ≠, ≤ and ≥. In terms of selection and Cartesian product operations, the theta join operation can be represented as follows:

$R \bowtie_F S = \sigma_F (R \times S)$.

(ii) Equi join – Equi join operation is a theta join operation where the comparison operator equality (=) is used in the join predicate.

(iii) Natural join – The natural join operation is an equi join operation performed over all the attributes in two relations that have the same name. One occurrence of each common attribute is eliminated from the resultant relation.

(iv) Outer join – Generally, a join operation selects tuples from two relations that satisfies the join predicate. In outer join operation, those tuples that do not satisfy the join predicate also appear in the resultant relation. The **left outer join** operation on two relations *R* and *S* selects unmatched tuples from the relation *R* along with the matched tuples from both the relations *R* and *S* that satisfy the join predicate. Missing values in the relation *S* are set to null. The **right outer join** operation on two relations *R* and *S* selects unmatched tuples from the relation *S* along with the matched tuples from both the relations *R* and *S* that satisfy the join predicate. In the case of right outer join, the missing values in the relation *R* are set to null. In the case of **full outer join**, unmatched tuples from both the relations *R* and *S* appear in the resultant relation along with the matched tuples.

(v) Semijoin – The semijoin operation on two relations *R* and *S* defines a new relation that contains the tuples of *R* that participate in the join of *R* with *S*. Hence, only the attributes of relation *R* are projected in the resultant relation. The advantage of semijoin operation is that it decreases the number of tuples that are required to be handled to form the join. The semijoin operation is very useful for performing join operations in distributed databases. The semijoin operation can be rewritten as follows using the projection and join operations of relational algebra:

$R \rhd_F S = \prod_A (R \bowtie_F S)$, where *A* is the set of all attributes for *R*.

Example 1.10

The result of natural join operation that involves the Employee and Department relations of Example 1.1 is depicted in figure 1.8.

Employee $\bowtie_{\text{deptno}}$ Department

emp-id	ename	designation	salary	deptno	voter-id	dname	location
E01	J. Lee	Manager	80,000	10	V01	Development	Lake side
E02	S. Smith	Asst. Manager	60,000	20	V03	Production	Near playground
E03	A. Roy	Analyst	70,000	10	V04	Development	Lake side
E04	B. Sharma	Programmer	20,000	30	V02	Maintenance	Lake side
E05	D. Davis	Manager	75,000	30	V05	Maintenance	Lake side
E06	M. Jones	Consultant	85,000	40	V07	HR	Near Garden
E07	A. Sasmal	Manager	80,000	20	V06	Production	Near playground
E08	R. Miller	Consultant	75,000	10	V08	Development	Lake side
E09	S. Sen	Analyst	72,000	40	V10	HR	Near Garden
E10	D. Dastidar	Asst. Manager	50,000	40	V09	HR	Near Garden

Fig. 1.8 The Result of Natural Join

Example 1.11

The result of semijoin operation that involves the Employee and Department relations is illustrated in figure 1.9.

Employee $\ltimes_{\text{deptno}}$ Department

emp-id	ename	designation	salary	deptno	voter-id
E01	J. Lee	Manager	80,000	10	V01
E02	S. Smith	Asst. Manager	60,000	20	V03
E03	A. Roy	Analyst	70,000	10	V04
E04	B. Sharma	Programmer	20,000	30	V02
E05	D. Davis	Manager	75,000	30	V05
E06	M. Jones	Consultant	85,000	40	V07
E07	A. Sasmal	Manager	80,000	20	V06
E08	R. Miller	Consultant	75,000	10	V08
E09	S. Sen	Analyst	72,000	40	V10
E10	D. Dastidar	Asst. Manager	50,000	40	V09

Fig. 1.9 The Result of Semijoin

1.4.8 Division Operation

Consider two relations R and S defined over the attribute sets A and B respectively such that $B \subseteq A$ and $C = A - B$. The division operation of the two relations R and S defines a relation over the attribute set C that consists of the set of tuples from R that matches the combination of every tuple in S. In terms of fundamental operations in relational algebra, the division operation can be represented as follows:

$$R \div S = \prod_C(R) - \prod_C ((S \times \prod_C(R)) - R).$$

Example 1.12

Let us consider the two relations Studentinfo and Courseinfo [in **figure 1.10**] where one student can take admission in multiple courses and more than one student can enrol in each course. Using division operation the query "Retrieve all those student names who have taken admission in all courses where the course duration is greater than 2 years" will produce the result as shown in figure 1.10.

Studentinfo

sreg-no	sname	course-id
S01	Ankit	C01
S02	John	C01
S03	Sachin	C03
S01	Ankit	C02
S03	Sachin	C02
S02	John	C02
S01	Ankit	C03

Result

sname
Ankit

Courseinfo

course-id	cname	duration	fees
C01	PGDBM	4	150,000
C02	PGCACS	3	100,000
C03	CFA	4	120,000

Fig. 1.10 Example of Division Operation

1.5 Relational Database Management System

A relational database management system (RDBMS) is a software package that stores data into the database based on the relational data model and supports relational database languages for retrieving and manipulating data in the database. An RDBMS provides all the functionalities of

a DBMS. In a top-down manner, the different levels of these functionalities can be expressed as **application interface, security control, compilation, execution, data access** and **consistency control.** The application interface level manages interfaces to the user's applications. The security control level is responsible for maintaining data integrity and authorization checking. The query processing module of an RDBMS is responsible for processing queries and retrieving data from the database. This task is performed collectively by three different levels, namely, *compilation, execution* and *data access.* The consistency control level ensures the correctness of the data in spite of parallel or concurrent access to the data.

CHAPTER SUMMARY

- A database is a collection of interrelated data and a database management system (DBMS) is a set of programs that are used to store, retrieve and manipulate data in the database.
- A relational database is a collection of relations where each relation is represented by a two-dimensional table. Each row of the table is known as a tuple, and each column of the table is called an attribute.
- There are two principal integrity constraints in relational data model: entity integrity constraint and referential integrity constraint. Entity integrity constraint requires that the primary key of a relation must be not null. Referential integrity constraint is expressed in terms of foreign key.
- Normalization is a step-by-step process to achieve better data representation in relational database design. There are different normal forms such as 1NF, 2NF, 3NF, BCNF, 4NF and 5NF, and each normal form has some requirements. Functional dependency is a very important concept in normalization.
- The relational algebra is a high-level procedural language that is based on mathematical set theory. The five fundamental operations of relational algebra are selection, projection, Cartesian product, union and set difference. In addition, there are three derived operations, namely, join, intersection and division, which can be represented in terms of the five fundamental operations.
- An RDBMS is a software package that supports relational data model and relational languages.

EXERCISES

Multiple Choice Questions

(i) Relational data model is a
   a. Formal model
   b. Non-formal model
   c. Semi-formal model
   d. None of the above.

(ii) The degree of a relation is
   a. The total number of candidate keys in the relation
   b. The total number of attributes in the relation
   c. The total number of tuples in the relation
   d. None of the above.

(iii) The cardinality of a relation represents
   a. The total number of attributes in the relation
   b. The total number of tuples in the relation
   c. The total number of super keys in the relation
   d. None of the above.

(iv) Which of the following statements is false?
   a. A relation can have more than one candidate keys
   b. A relation can have only one primary key
   c. A relation can have several super keys
   d. None of the above
   e. All of the above.

(v) Null value represents
   a. Zero value
   b. Blank value
   c. Undefined value
   d. None of the above.

(vi) Entity integrity constraint stipulates that
   a. For each attribute there must be a set of permitted values
   b. The value of each attribute in a primary key must be not null
   c. All of the above
   d. None of the above.

(vii) Which of the following statements is false?
   a. Primary keys are candidate keys
   b. Alternate keys are candidate keys
   c. Candidate keys are super keys
   d. None of the above.

(viii) Which of the following is a trivial functional dependency?
   a. $X \rightarrow Y$
   b. $Y \rightarrow X$
   c. $XY \rightarrow Y$
   d. None of the above.

(ix) Tuples selection based on predicate is done by using
   a. Selection operation
   b. Projection operation
   c. Cartesian Product
   d. All of the above

(x) Which of the following statements is true?
   a. An equi-join is a theta join
   b. A natural join is an equi-join
   c. A natural join is a theta join
   d. All of the above.

(xi) Which of the following statements is true?
   a. A semijoin operation requires no join attribute
   b. A semijoin operation reduces the size of the resultant relation compared to a join operation
   c. A semijoin operation in not applicable for centralized DBMS
   d. None of the above.

(xii) Which of the following anomalies occurs owing to referential integrity constraints?
   a. Repetition anomaly
   b. Insertion anomaly
   c. Deletion anomaly
   d. All of the above.

(xiii) In a relation R, a multi-valued dependency of the form $X \twoheadrightarrow Y$ is called trivial multi-valued dependency if
   a. $Y \subset X$ and $X \cup Y = R$
   b. $Y \subset X$ or $X \cup Y = R$
   c. All of the above
   d. None of the above.

(xiv) Which of the following the statement is true?
   a. A left outer join retrieves tuples from both relations that match the join attribute value only.
   b. A left outer join retrieves tuples from both relations that match the join attribute value and unmatched tuples from both relations.
   c. A left outer join retrieves tuples from both the relations that match the join attribute value and unmatched tuples from the left relation.
   d. A left outer join retrieves tuples from both the relations that match the join attribute value and unmatched tuples from the right relation.

(xv) Which of the following statements is true?
   a. If $X \twoheadrightarrow Y$, then $X \rightarrow Y$
   b. If $X \twoheadrightarrow Y$, then $X \nrightarrow Y$
   c. If $X \twoheadrightarrow Y$, then $X \subset Y$
   d. If $X \twoheadrightarrow Y$, then $Y \twoheadrightarrow X$.

Review Questions

1. Why table is called a relation? Explain your answer with an example.
2. Define referential integrity constraint and give an example.

3. Discuss different types of anomalies that may occur in relational data model owing to referential integrity constraint.

4. What are the objectives of normalization?

5. What is multi-valued dependency? Give an example.

6. Prove that if a relation is in 4NF, then it is also in BCNF.

7. Is semijoin operation beneficial over join operation? Justify your answer.

8. Describe different types of join operation with examples.

9. Explain the concept of transitive dependency and discuss how it is related to 3NF. Give an example to illustrate your answer.

10. Differentiate between 3NF and BCNF with examples.

11. Describe different types of outer join with examples.

12. Explain division operation with an example.

13. Consider the Employee and Department relations in Example 1.1. Write the relational algebraic expression for the following queries:
    (a) List all those employee names whose department location is "Lake side".
    (b) List all those employee names and employee codes whose designation is "Manager".
    (c) List all those employee names and their department names whose salary is between 60,000 and 80,000 including both.
    (d) List all those employee names and their department numbers whose voter identification number is either "V01" or "V09".
    (e) List all those department names and department locations whose department number is greater than 20.

14. Consider the relation $R = \{A, B, C, D, E, F, G, H, I, J\}$ and the set of functional dependencies $\{AB \rightarrow C, A \rightarrow DE, B \rightarrow F, F \rightarrow GH, D \rightarrow IJ\}$. What is the key attribute for the relation R? Decompose the relation R into 3NF.

15. Consider the relation $R = \{A, B, C, D, E\}$ and the set of functional dependencies $\{AB \rightarrow C, CD \rightarrow E, DE \rightarrow B\}$. Which one is the key attribute for the relation, AB or ABD? Justify your answer.

16. Consider the relational schema BOOK (title, author, booktype, price, affiliation, publishers) and the set of functional dependencies {title → publishers booktype, booktype → price, author → affiliation}. In which normal form the relation is? Explain your answer.

17. Consider the relational schema Employee (empid, name, designation, phoneno, projectno) and the following set of functional and multi-valued dependencies.

    empid ↠ phoneno,
    empid ↠ projectno,
    empid → name designation.

    Is the relation in 4NF or not? If not, convert the relation into 4NF.

18. Consider the following relational schemas Student and Course.

Student

sreg-no	sname	street	city	course-id
S01	Amit	Rowdon st.	Kolkata	C01
S02	John	Circus Row	Delhi	C01
S03	Rohit	Park st.	Kolkata	C03
S05	Sachin	Akbar Road	Delhi	C02
S07	Sagar	Jhu Road	Mumbai	C02
S08	Anuj	Elite Road	Jaipur	C03

Course

course-id	Cname	duration	Fees
C01	B.Tech	4	120,000
C02	MBA	3	150,000
C03	B.Pharm	4	100,000
C04	M.Tech	2	50,000
C05	PGDBM	2	100,000

Write down the resultant relations for natural join operation, semijoin operation and full outer join operation between Student and Course.

2 Review of Database Systems

This chapter introduces the evolution of distributed database technology and some key concepts of parallel database systems. This chapter mainly focuses on the benefits of parallel databases and different types of architecture for parallel database systems. Data partitioning techniques for parallel database design are also discussed with examples.

The outline of this chapter is as follows. The evolution of distributed database system is represented in Section 2.1. In Section 2.2, an overview of parallel processing system is presented, and different architectures for parallel database systems are described in Section 2.2.3. The data partitioning techniques for parallel database design are discussed in Section 2.3.

2.1 Evolution of Distributed Database System

Database technology has evolved motivated by a requirement to manage large volumes of data. Management of data involves both defining structures for storage of data and providing mechanisms for the efficient manipulation of data. During the 1970s, centralized database management systems were implemented to overcome the drawbacks of traditional file-based systems and to satisfy structured information requirements. The motivation behind the development of centralized database systems was the need to integrate the operational data of an organization and to provide controlled access to data centrally.

A major limitation of centralized database system is that all information must be stored in a single central location, usually in a mainframe computer or a minicomputer. The performance may degrade if the central site becomes a bottleneck. Moreover, the centralized approach fails to satisfy the faster response time and quick access to information. In the 1980s, a series of critical social and technological changes had occurred that affected the design and development of database technology. Business environment became decentralized geographically and required a dynamic environment in which organizations had to respond quickly under competitive and technological pressures. Over the past two decades, advancements in microelectronic technology have resulted in the availability of fast, inexpensive processors. During recent times, rapid development in network and data communication technology, characterized by the internet, mobile and wireless computing, has resulted in the availability of cost-effective and highly efficient computer networks. The merging of computer and network technologies changed the mode of working from a centralized to a decentralized manner. This decentralized approach simulates the organizational structure of many companies that are logically distributed into divisions, departments, projects and so on and physically distributed into offices, plants, depots and factories, where each unit maintains its own operational data [Date, 2000]. Database systems had addressed the challenges of distributed computing from the very early days. In this context, distributed DBMS has been gaining acceptance as

a more sophisticated modern computing environment. Starting from the late 1970s, a significant amount of research work has been carried out both in universities and in industries in the area of distributed systems. These research activities provide us with the basic concepts for designing distributed systems.

Computer system architectures consisting of interconnected multiple processors are basically classified into two different categories: loosely coupled systems and tightly coupled systems. These are described in the following:

(i) Tightly coupled systems – In these systems, there is a single systemwide global primary memory (address space) that is shared by all processors connected to the system. If any processor writes some information into the global memory, it can be shared by all other processors in the system. For example, if a processor writes the value 200 to a memory location *y*, any other processor subsequently reading from the location *y* will get the value 200. Thus, in these systems, any communication between the processors usually takes place through the shared memory. The tightly coupled system is illustrated in figure 2.1.

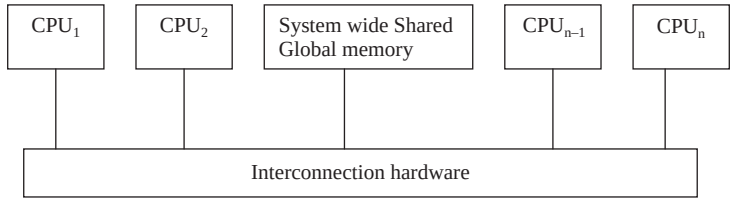


Fig. 2.1 A Tightly Coupled Multiprocessor System

(ii) Loosely coupled systems – In these systems, processors do not share memory (clocks and system buses also), and each processor has its own local memory. In this case, if a processor writes the value 200 to a memory location *y*, this write operation only changes the content of its own local memory and does not affect the content of the memory of any other processor. In such systems, all physical communications between the processors are established by passing messages across the network that interconnects the processors of the system. The loosely coupled system is depicted in figure 2.2.

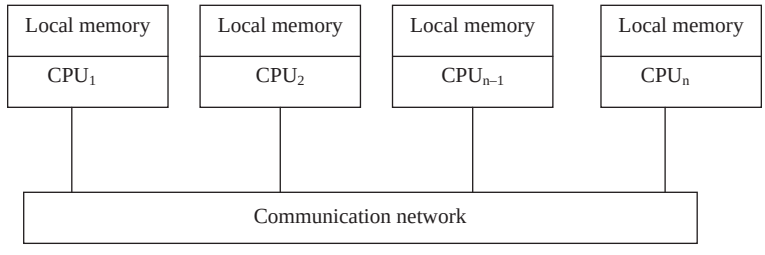


Fig. 2.2 A Loosely Coupled Multiprocessor System

Usually, tightly coupled systems are referred to as **parallel processing systems**, and loosely coupled systems are referred to as **distributed computing systems**, or simply **distributed systems**.

2.2 Overview of Parallel Processing System

A **parallel processing system** involves multiple processes that are active simultaneously and solving a given problem, generally on multiple processors or nodes. The **parallel processing** technique divides a large task into several subtasks (smaller tasks) and executes these subtasks concurrently on several processors. Since the subtasks can execute on different processors in parallel, the given task is completed more quickly. A parallel processing system has the following characteristics:

- All processors in the system can perform their tasks concurrently.
- Tasks may require to be synchronized.
- Nodes (processors) usually share resources such as data, disks and other devices.

Parallel processing happens in real-life situations also. For example, in a banking organization if a number of employees provide services to several customers simultaneously, the service becomes faster. On the other hand, there are some real-life situations where parallel processing is not possible. Effective implementation of a parallel processing system involves two challenges:

- Structuring of tasks so that several tasks can be executed at the same time "in parallel".
- Preserving task sequencing so that tasks can be executed serially.

The parallel processing technique increases the system performance in terms of two important properties. These are speedup and scaleup as described below.

(a) Speedup – Speedup is the extent to which more hardware can perform the same task in less time than the original system. With added hardware, speedup holds the task constant and measures time saving. With good speedup, additional processors reduce system response time. For example, if one processor requires n units of time to complete a given task, n processors require 1 unit of time to complete the same task if parallel execution is possible for the given task. Speedup can be calculated using the following formula:

speedup = time_original / time_parallel,

where time_parallel is the elapsed time spent by a larger parallel system on the same task.

(b) Scaleup – Scaleup is the factor that represents how much more work can be done in the same time period by a larger system. With added hardware, scaleup holds time as a constant and measures the increased size of the job that can be done within that constant period of time. If transaction volumes grow and the system has good scaleup, the response time can be kept constant by adding more hardware resources (such as CPUs). Scaleup can be computed using the following formula:

scaleup = volume_parallel / volume_original,

where volume_parallel is the transaction volume (amount of job) processed in a given amount of time on a parallel system.

2.2.1 Parallel Databases

In the case of database applications, some tasks can be divided into several subtasks effectively, and the parallel execution of these subtasks reduces the total processing time by surprisingly large amounts, thereby improving system performance. Features such as online backup, data replication, portability,

interoperability and support for a wide variety of client tools can enable a parallel server to support application integration, distributed operations and mixed-application workloads. A variety of computer system architectures allow sharing of resources such as data, software and peripheral devices among multiple processors. Parallel databases are designed to take advantage of such architectures.

A parallel database is designed to take advantage of executing operations in parallel, by running multiple instances that share a single physical database. A parallel server processes transactions in parallel by servicing a stream of transactions using multiple processors on different sites, where each processor processes an entire transaction. To improve system performance, a parallel database system allows multiple users to access a single physical database from multiple machines. To balance the workload among processors, parallel databases provide concurrent access to data and preserve data integrity.

2.2.2. Benefits of Parallel Databases

Parallel database technology provides a number of benefits, which are listed in the following:

Better Performance – In a parallel database system, for processing an application higher speedup and scaleup can be attained by involving a number of processors. The improvement in performance depends on the degree of inter-node locking and synchronization activities. The volume of lock operations and database contention as well as the throughput and performance of the Distributed Lock Manager (DLM) ultimately determines the scalability of the system.

Higher Availability – In a parallel database system, processors are isolated from each other, so a failure of one node does not imply the failure of the entire system. One of the surviving nodes recovers the failed node while the other nodes in the system continue to provide data access to users. Therefore, data availability in a parallel database system is much higher than that in a centralized database system in case of failure.

Greater Flexibility – Parallel server environments (such as Oracle Parallel Server) are extremely flexible. One can allocate or deallocate instances as necessary. For example, one can temporarily allocate more instances as demand on database increases. When they are no longer required, these instances can be deallocated and used for other purposes.

Serves more users – In a parallel database technology system, it is possible to overcome memory limits; thus, a single system can serve thousands of users.

2.2.3 Parallel Database Architectures

There are several architectural models for parallel databases, which are listed in the following:

- **Shared-memory** – Shared-memory is a tightly coupled architecture in which multiple processors within a single system share a single system memory.
- **Shared-disk** – Shared-disk is a loosely coupled architecture in which multiple processors share a common set of disks. Shared-disk systems are sometimes called **clusters**.
- **Shared-nothing** – Shared-nothing, often known as massively parallel processing (MPP), is a multiple processor architecture in which processors do not share a common memory or a common set of disks.
- **Hierarchical** – Hierarchical architectural model is a hybrid of shared-memory, shared-disk and shared-nothing architectures.

Shared-memory architecture

Shared memory architecture is a tightly coupled architecture in which multiple processors within a single system share a common memory, typically via a bus or through an interconnection network. This approach is known as **symmetric multiprocessing (SMP)**. SMP has become popular on platforms ranging from personal workstations that support a few microprocessors in parallel to large RISC-(reduced instruction set computer) based machines, all the way up to the largest mainframes. In shared-memory architecture, a processor can send messages to other processors using memory writes (which usually take less than a microsecond), which is much faster than sending a message through a communication mechanism. This architecture provides high-speed data access for a limited number of processors, but it is not scalable beyond about 64 processors, when the interconnection network becomes a bottleneck. Shared-memory architecture is illustrated in figure 2.3.

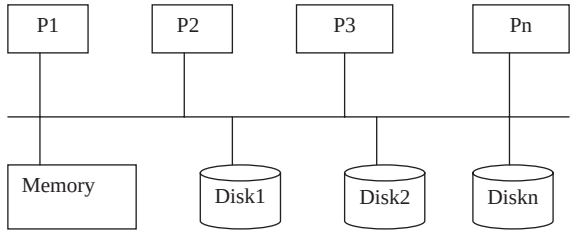


Fig. 2.3 Shared-memory Architecture

Shared-memory architecture provides a number of advantages such as simplicity and load sharing. In this architecture, as all the information is shared by all processors, developing database software is not very difficult, and also it does not vary too much from the database software that is designed for a single-processor system. Load sharing is excellent as it can be achieved at runtime using the shared memory.

Shared-memory architecture has a number of disadvantages also. These are cost, limited extensibility and low availability. High cost is incurred owing to the interconnection that links each processor to each memory module or disk. Usually, Shared-memory architectures have large memory caches at each processor, so that referencing of the shared memory is avoided whenever possible. Moreover, the caches need to be kept coherent; that is, if a processor performs a write operation to a memory location, the data in that memory location should be either updated or removed from any processor where the data is cached. Maintaining cache-coherency becomes an increasing overhead with increasing number of processors. Consequently, shared-memory architecture is not scalable beyond about 64 processors when the interconnection network becomes a bottleneck; hence, extensibility is limited. Finally, in shared-memory architecture memory space is shared by all processors in the system; therefore, a memory fault may affect most of the processors, thereby limiting data availability.

Examples of shared-memory parallel database system include XPRS [Hong, 1992], DBS3 [Bergsten et al., 1991], and Volcano [Graefe, 1990], as well as portings of major commercial DBMSs on shared-memory multiprocessors.

Shared-disk architecture

Shared-disk is a loosely coupled architecture in which all processors share a common set of disks. Shared-disk systems are sometimes called **clusters**. In this architecture, each processor can

access all disks directly via an interconnection network, but each processor has its own private memory. Shared-disk architecture can be optimized for applications that are inherently centralized and require high availability and performance.

Shared-disk architecture provides several advantages over shared-memory architecture such as cost, performance, availability, extensibility and easy migration from uniprocessor systems. The cost of interconnection network is significantly less than that in shared-memory architecture, because standard bus technology can be used. In shared-disk architecture, as each processor has its own local memory, the memory bus is not a bottleneck. Moreover, it offers a cheap way to provide a degree of fault tolerance. In this architecture, if a processor (or its memory) fails, the other processors can take over its tasks, as the database resides on disks and is accessible from all processors; thus, availability increases. The interference on the shared-disk can be minimized, as each processor has its own private memory, thereby improving extensibility. Finally, migrating from a centralized system to a shared-disk system is relatively straightforward, as the data on disk need not be reorganized.

The disadvantages of shared-disk architecture are complexity and lack of scalability. It is more complex than shared-memory architecture, as it requires distributed database system protocols such as distributed locking and two-phase commit. Furthermore, high communication overhead is incurred to maintain the coherency of copies. Although the memory bus is no longer a bottleneck in shared-disk system, the interconnection to the disk subsystem is a bottleneck. Shared-disk systems can scale-up to a large number of processors, but communication across processors is slower, as it has to go through a communication network. Shared-disk architecture is illustrated in figure 2.4.

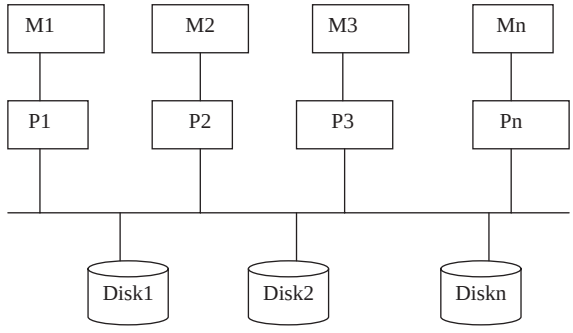


Fig. 2.4 Shared-disk Architecture

DEC (Digital Equipment Corporation) cluster running Rdb was one of the early commercial users of the shared-disk database architecture. Rdb is now owned by Oracle and is called Oracle Rdb.

Shared-nothing architecture

Shared-nothing architecture, often known as **MPP**, is a multiprocessor architecture in which each processor is part of a complete system with its own memory and disk storage. The communication between the processors is incorporated via high-speed interconnection network. The database is partitioned among all disks associated with the system, and data is transparently available to all users in the system. Each processor functions as a server for the data that is stored on its own disk (or disks). The shared-nothing architecture is depicted in figure 2.5.

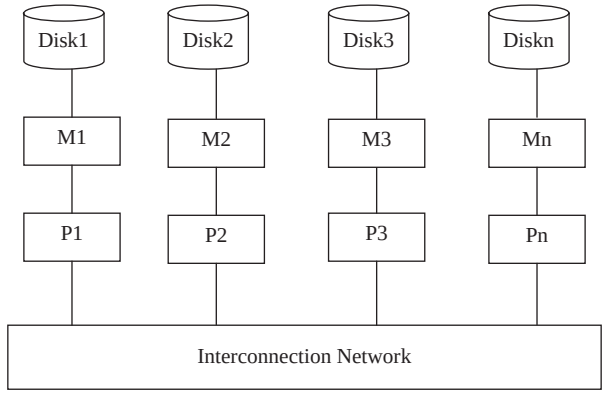


Fig. 2.5 Shared-Nothing Architecture

The advantages of shared-nothing architecture are better communication, scalability, extensibility and availability. In shared-nothing architecture local disk references are serviced by local disks at each processor; thus, it overcomes the disadvantage of requiring all I/O to go through a single interconnection network. Only queries accessing the non-local disks and result relations pass through the interconnection network. This architecture is usually designed to be scalable, so that its transmission capacity increases as more processors are added. Consequently, shared-nothing architecture is more scalable than shared-memory architecture and can easily support a large number of processors. By implementing a distributed database design that supports the smooth incremental growth of the system by the addition of new nodes, extensibility can be improved. Linear speedup and linear scaleup can be achieved for simple workloads by careful partitioning of data on multiple disks. With replication of data on multiple nodes, high availability can also be achieved.

Shared-nothing architecture has several disadvantages also. The main drawback of shared-nothing architecture is the cost of communication for accessing non-local disk, which is much higher than that of other parallel system architectures as sending data involves software interaction at both ends. Shared-nothing architecture entails more complexity than shared-memory architecture, as necessary implementation of distributed database functions – assuming large numbers of nodes – is very complicated. Although shared-nothing systems are more scalable, the performance is optimal only when requested data is available in local storage. Furthermore, the addition of new nodes in the system presumably requires reorganization of the database to deal with the load-sharing issues.

The Teradata database machine was among the earliest commercial systems using shared-nothing database architecture. The Grace and the Gamma research prototypes also made use of shared-nothing architecture.

Hierarchical architecture

The hierarchical architecture is a combination of shared-memory, shared-disk and shared-nothing architectures. At the top level, the system consists of a number of nodes connected by an interconnection network, and the nodes do not share disks or memory with one another. Thus, top level takes the form of shared-nothing architecture. Each node of the system is a shared-memory system with a few processors. Alternatively, each node can be a shared-disk system, and each of these subsystems sharing a set of disks can be a shared-memory system. Thus, hierarchical system is built as

a hierarchy, with shared-memory architecture with a few processors at the base, and shared-nothing architecture at the top, with possibly shared-disk architecture in the middle. The hierarchical architecture is illustrated in figure 2.6.

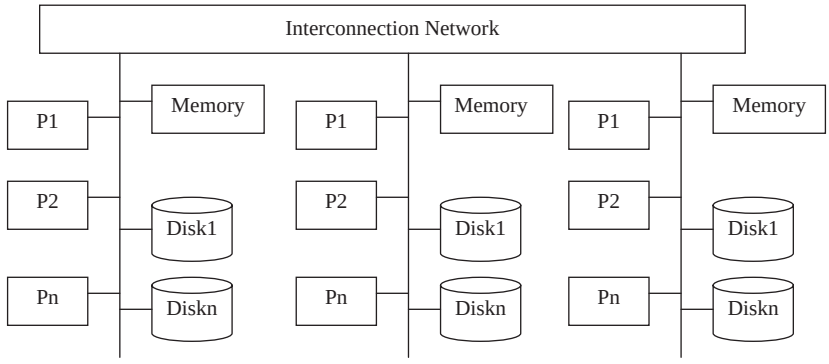


Fig. 2.6 Hierarchical Architecture

The advantages of hierarchical architecture are evident. It provides all the advantages of shared-memory, shared-disk and shared-nothing architectures such as flexibility and better performance of shared-memory architecture, higher data availability of shared-disk architecture and high extensibility of shared-nothing architecture.

The major disadvantage of hierarchical architecture is complexity. To reduce the complexity of programming, such systems have yielded distributed virtual memory architecture, where logically there is a single shared memory, but physically there are multiple disjoint memory systems. The virtual memory mapping hardware, coupled with system software, allows each processor to view the disjoint memories as a single virtual memory. As access speeds differ depending on whether the page is available locally or not, such architecture is also referred to as **non-uniform memory architecture (NUMA)**.

Today, several commercial parallel database systems run on these hierarchical architectures.

2.3 Parallel Database Design

The crucial issues in parallel database design include data partitioning, parallel query processing, query optimization and parallel transaction management. Parallel database systems support various types of parallelism both between and within queries (inter- and intra-query parallelism). An efficient exploitation of parallelism requires sophisticated load-balancing techniques to distribute the workload across the resources (such as CPUs, disks, main memory and network) of a parallel system. Load balancing must be supported by adequate data-partitioning methods that distribute the data across multiple disks connected to the system so as to enable efficient parallel I/O.

2.3.1 Data Partitioning

Data partitioning has its origins in centralized system where it is used to partition files, either because the file is too big for one disk or because the file access rate cannot be supported by a

single disk. **Data partitioning** allows parallel database systems to exploit the I/O parallelism of multiple disks by reading and writing on them in parallel. **I/O parallelism** refers to reducing the time required to retrieve relations from the disk by partitioning the relations on multiple disks. Partitioning of a relation involves distributing its tuples over several disks, so that each tuple resides on one disk. This most common form of data partitioning in a parallel database environment is known as **horizontal partitioning**. A number of data partitioning techniques for parallel database system have been projected, which are listed in the following.

Round-robin

The simplest partitioning strategy distributes tuples among the fragments in a **round-robin** fashion. This is the partitioned version of the classic entry-sequence file. Assume that there are n disks, $D_1, D_2, \ldots, D_n$, among which the data are to be partitioned. This strategy scans the relation in some order and sends the ith tuple to disk number $D_{i \bmod n}$. The round-robin scheme ensures an even distribution of tuples across disks.

Round-robin partitioning is excellent for applications that wish to read the entire relation sequentially for each query. The problem with round-robin partitioning is that it is very difficult to process point queries and range queries. A **point query** involves the retrieval of tuples from a relation that satisfies a particular attribute value whereas a **range query** involves the retrieval of tuples from a relation that satisfies the attribute value within a given range. For example, the retrieval of all tuples from the Student relation **[Chapter 1, Example 1.6]** with the value "Kolkata" for the attribute "city" is a point query. An example of a range query is retrieving all tuples from the Employee relation **[Chapter 1, Example 1.1]** that satisfy the criterion "the value of the salary attribute is within the range 50,000 to 80,000". With round-robin partitioning scheme, processing of range queries as well as point queries is complicated, as it requires the searching of all disks.

Example 2.1

Let us consider the Employee relation **[Chapter 1, Example 1.1]** with 1,000 tuples that is to be distributed among 10 disks. Using round-robin data partitioning technique, tuple number 1 will be stored on disk number 1 and tuple number 2 will be stored on disk number 2. All those tuples that have the same digit in the right most position of the tuple number will be stored on the same disk. In this case, tuple numbers 1, 11, 21, 31, 41, …, 991 of the Employee relation will be stored on disk number 1, because "tuple number mod 10" produces the result 1 for all these tuples. The data distribution on disks is uniform here, and each disk will store 100 tuples. Hence, processing of the point query "designation = 'Manager' " for the relation is very difficult. Similarly, the range query "salary between 50,000 and 80,000" is also very difficult to process when using round-robin partitioning technique.

Hash partitioning

This strategy chooses one or more attributes from the given relational schema as the partitioning attributes. A hash function is chosen whose range is within 0 to $n-1$, if there are n disks. Each tuple of the original relation is hashed based on the partitioning attributes. If the hash function returns i then the tuple is placed on disk D_i.

Hash Partitioning is ideally suited for applications that want only sequential and associative accesses to the data. Tuples are stored by applying a *hashing* function to an attribute. The hash function specifies the placement of the tuple on a particular disk. Associative access (point query) to the tuples with a specific attribute value can be directed to a single disk, avoiding the overhead

of starting queries on multiple disks. If the hash function is a good randomizing function and the partitioning attributes form a key of the relation, then the number of tuples in each disk is almost the same without much variance. In this case, the time required to scan the entire relation is $1/n$ of the time required to scan the same relation in a single-disk system, because it is possible to scan multiple disks in parallel. However, hash partitioning is not well-suited for point queries on non-partitioning attributes. This partitioning technique is not suitable for range queries also.

Database systems pay considerable attention to cluster related data together in physical storage. If a set of tuples is routinely accessed together, the database system attempts to store them on the same physical page. Hashing tends to randomize data rather than cluster it. Hash partitioning mechanisms are provided by Arbre, Bubba [Copeland, Alexander, Bougherty and Keller, 1988], Gamma, and Teradata.

Example 2.2

Assume that the relational schema **Student (rollno, name, course-name, year-of-admission)** has 1,000 tuples and is to be partitioned among 10 disks. Further assume that, the attribute **rollno** is the hash partitioning attribute and the hash function is R mod n, where R represents the values of the attribute rollno in student tuples and n represents the total number of disks. Using hash partitioning technique, all those tuples of the Student relation that have the same digit in the right most position of the roll number will be stored into the same disk. Hence, all tuples whose roll numbers belong to the set $A = \{1, 11, 21, 31, 41, \ldots, 991\}$ will be stored on disk number 1. Similarly, all those student records whose roll numbers belong to the set $B = \{2, 12, 22, 32, \ldots, 992\}$ will be stored on disk number 2. In this case, it is easier to process point queries and range queries that involve the attribute rollno. If rollno is the key of the relation Student, then tuples distribution among disks is also uniform. However, processing of point queries and range queries that involve non-partitioning attributes such as name, course-name and year-of-admission is difficult.

Range partitioning

Range partitioning clusters together tuples with similar attribute values in the same partition. This strategy distributes contiguous attribute-value ranges to each disk. It chooses a partitioning attribute as a partitioning vector. Using range-partitioning technique, a relation is partitioned in the following way. Let $\{v_0, v_1, \ldots, v_{n-2}\}$ denote the partition vector of the partitioning attribute such that if $i < j$, then $v_i < v_j$. Consider a tuple t such that $t[A] = x$. If $x < v_0$, then t goes on disk D_0 and if $x >= v_{n-2}$, then t goes on disk D_{n-1}. If $v_i \leq x < v_{i+1}$, then t goes on disk D_{i+1}.

Range partitioning is good for sequential and associative access, and is also good for clustering data. It derives its name from the typical SQL range queries such as "salary between 50,000 and 80,000". The problem with range partitioning is that it risks data skew, where all the data is placed in one partition, and execution skew, in which all the execution occurs in one partition.

Hashing and round-robin are less susceptible to these skew problems. Range partitioning can minimize skew by picking non-uniformly distributed partitioning criteria. Bubba uses this concept by considering the access frequency *(heat)* of each tuple when creating partitions of a relation.

Example 2.3

Let us assume that the Department relation **[Chapter 1, Example 1.1]** has 50 tuples and is to be distributed among 3 disks. Further assume that deptno is the partitioning attribute and $\{<20, \geq 20$ and $\leq 40, >40\}$ is the partition vector. Using range-partitioning technique, all tuples that has deptno

values less than 20 will be stored on disk number 1, all tuples that has deptno values between 20 and 40 will be stored on disk number 2 and all tuples that has deptno values greater than 40 will be stored on disk number 3. In this case, processing of point queries and range queries that involve the deptno attribute is easier. However, processing of point queries and range queries that involve non-partitioning attributes is difficult.

CHAPTER SUMMARY

» Computer architectures comprising of interconnected multiple processors are basically classified into two categories: loosely coupled systems and tightly coupled systems.

» In loosely coupled systems, processors do not share memory, but in tightly coupled systems processors share memory. Loosely coupled systems are referred to as distributed systems, whereas tightly coupled systems are referred to as parallel processing systems.

» Parallel processing systems divide a large task into many smaller tasks, and execute these smaller tasks concurrently on several processors.

» Parallel processing increases the system performance in terms of two important properties, namely, scaleup and speedup.

» Alternative architectural models for parallel databases are shared-memory, shared-disk, shared-nothing and hierarchical.

> **Shared-memory** – Shared-memory is a tightly coupled architecture in which multiple processors within a single system share system memory.

> **Shared-disk** – Shared-disk is a loosely coupled architecture in which multiple processors share a common set of disks. Shared-disk systems are sometimes called clusters.

> **Shared-nothing** – Shared-nothing, often known as MPP, is a multiple processor architecture in which processors do not share common memory or common set of disks.

> **Hierarchical** – This model is a hybrid of shared memory, shared-disk and shared-nothing architectures.

» **Data partitioning** allows parallel database systems to exploit the I/O parallelism of multiple disks by reading and writing on them in parallel. Different data-partitioning techniques are round-robin, hash partition and range partition.

EXERCISES

Multiple Choice Questions

(i) Shared memory is
a. Loosely coupled architecture
b. Tightly coupled architecture
c. Both a and b
d. None of the above.

(ii) Shared-nothing is also known as
a. Clusters
b. Massively parallel processing system
c. Both a and b
d. None of the above.

(iii) NUMA (non-uniform memory architecture) is
a. Shared-memory architecture
b. Shared-disk architecture
c. Shared-nothing architecture
d. Hierarchical architecture.

(iv) Symmetric multiprocessing (SMP) is used in
a. Shared-memory architecture
b. Shared-disk architecture
c. Shared-nothing architecture
d. Hierarchical architecture.

(v) Which of the following statements is true?
   a. Data partitioning is a major drawback of parallel processing systems
   b. Data partitioning is applicable for centralized databases
   c. Data partitioning is beneficial for I/O parallelism
   d. All of the above.

(vi) Hash function is used in
   a. Range partitioning
   b. Round-robin
   c. Both in range partitioning and round robin
   d. None of the above.

(vii) A point query retrieves the tuples from a relation
   a. Based on a particular attribute value within a given range
   b. Based on the primary key value of the relation
   c. Based on a particular attribute value
   d. All of the above.

(viii) Even distribution of tuples across disks is ensured by
   a. Hash partitioning technique
   b. Range partitioning technique
   c. Round-robin partitioning technique
   d. All of the above.

(ix) A range query retrieves the tuples from a relation
   a. Based on a particular attribute value within a given range
   b. Based on the primary key value of the relation
   c. Based on the value of a particular attribute
   d. None of the above.

(x) Which of the following statements is false?
   a. Speedup and scaleup is associated with parallel processing systems
   b. Speedup and scaleup is associated with distributed processing systems
   c. Speed-up and scale-up is associated with centralized databases
   d. Both a. and b.

Review Questions

1. Differentiate between loosely coupled systems and tightly coupled systems.
2. What is a parallel processing system? How does it differ from a distributed processing system?
3. Define parallel databases. Discuss scaleup and speedup.
4. Discuss the benefits of parallel databases.
5. Compare and contrast a distributed DBMS with a parallel DBMS. Under what circumstances would you choose a distributed DBMS over a parallel DBMS?
6. Discuss the relative advantages and disadvantages of different parallel database architectures.
7. Define data partitioning for parallel databases.
8. Differentiate between range query and point query with examples.
9. Describe the different data-partitioning techniques with examples.
10. Compare and contrast different data-partitioning techniques for parallel database design.

3 Distributed Database Concepts

In the 1980s, distributed database systems had evolved to overcome the limitations of centralized database management systems and to cope up with the rapid changes in communication and database technologies. This chapter introduces the fundamentals of distributed database systems. Benefits and limitations of distributed DBMS over centralized DBMS are briefly discussed. The objectives of a distributed system, the components of a distributed system, and the functionality provided by a distributed system are also described in this chapter.

This chapter is organized as follows. Section 3.1 represents the fundamentals of distributed databases, and the features of distributed DBMSs are described in Section 3.2. In Section 3.3, pros and cons of distributed DBMSs are discussed, and an example of a distributed database system is represented in Section 3.4. The classification of distributed DBMSs is explained in Section 3.5, and the functions of distributed DBMSs are introduced in Section 3.6. Section 3.7 illustrates the components of a distributed database system, and Date's 12 objectives for distributed database system are discussed in Section 3.8.

3.1 Fundamentals of Distributed Databases

In recent years, the distributed database system has been emerging as an important area of information processing, and its popularity is increasing rapidly. A **distributed database** is a database that is under the control of a central DBMS in which not all storage devices are attached to a common CPU. It may be stored on multiple computers located in the same physical location, or may be dispersed over a network of interconnected computers. Collections of data (e.g., in a database) can be distributed across multiple physical locations. In short, a **distributed database** is a logically interrelated collection of shared data, and a description of this data is physically distributed over a computer network. A distributed database must ensure the following:

- **The distribution is transparent** – users must be able to interact with the system as if it is a single logical system. This applies to the system performance and method of accessing amongst other things.
- **The transactions are transparent** – each transaction must maintain database integrity across multiple databases. Transactions may also be divided into subtransactions; each subtransaction affects one database system.

A distributed database management system (DDBMS) can be defined as follows.

A DDBMS consists of a single logical database that is split into a number of **partitions** or **fragments**. Each partition is stored on one or more computers under the control of a separate DBMS,

with the computers connected by a communication network. Each site may have a substantial degree of independence. Each site in the distributed system is capable of independently processing user requests that require access to local data as well as it is capable of processing user requests that require access to remote data stored on other computers in the network. The sites in a distributed system have agreed to work together so that a user can access any data from anywhere in the network exactly as if the data are stored at the user's own site. A **distributed database system** allows applications to access various data items from local and remote databases. Applications are classified into two categories depending on whether the transactions access data from local site or remote sites.

- **Local applications** – These applications require access to local data only and do not require data from more than one site.
- **Global applications** – These applications require access to data from other remote sites in the distributed system.

3.2 Features of a Distributed DBMS

A Distributed DBMS may have a number of local applications, but it has at least one global application. Thus, a distributed DBMS has the following features:

(i) A distributed DBMS is a collection of logically related shared data.

(ii) The data in a distributed DBMS is split into a number of fragments or partitions.

(iii) Fragments may be replicated in a distributed system.

(iv) Fragments/replicas are allocated to different sites.

(v) In a distributed system, the sites are linked by communications network.

(vi) The data at each site is under the control of a DBMS.

(vii) The DBMS at each site has its own right, that is, it can handle local applications independently.

(viii) Each DBMS in a distributed system participates in at least one global application.

Every site in a distributed DBMS may have its own local database depending on the topology of the Distributed DBMS.

3.3 Advantages and Disadvantages of Distributed DBMS

A distributed DBMS has several advantages over traditional centralized DBMSs. These are listed in the following.

Sharing of information The major advantage in a distributed database system is the provision for sharing information. Users at one site in a distributed system may be able to access data residing at other sites. For example, consider an organization that has a number of branches throughout the country. Each branch stores its own data locally. Similarly, a user in one branch can access data from another branch; thus, information sharing is possible in a distributed system.

Faster data access End-users in a distributed system often work with only a subset of the entire data. If such data are locally stored and accessed, data accessing in distributed database system will be much faster than it is in a remotely located centralized system.

Speeding up of query processing A distributed database system makes it possible to process data at several sites simultaneously. If a query involves data stored at several sites, it may be possible to split the query into a number of subqueries that can be executed in parallel. Thus, query processing becomes faster in distributed systems.

Increased local autonomy In a distributed system, the primary advantage of sharing data by means of data distribution is that each site is able to retain a degree of control over data that are stored locally. In a centralized system, the database administrator is responsible for controlling the entire data in the database system. In a distributed system, there is a global database administrator responsible for the entire system. A part of these responsibilities is delegated to local database administrators residing at each site. Each local administrator may have a different degree of **local autonomy** or **independence** depending on the design of the distributed database system. The possibility of local autonomy is often a major advantage of distributed databases.

Increased availability If one site fails in a distributed system, the remaining sites may be able to continue the transactions of the failed site. In a distributed system, data may be replicated at several sites; therefore, a transaction requiring a particular data item from a failed site may find it at other sites. Thus, the failure of one site does not necessarily imply the shutdown of the system. In distributed systems, some mechanism is required to detect failures and to recover from failures. The system must no longer use the services of the failed site. Finally, when the failed site recovers, it must get integrated and smoothly come back into the system using appropriate mechanisms. Although recoveries in distributed systems are more complex than in centralized systems, the ability to continue normal execution in spite of the failure of one site increases availability. Availability is crucial for database systems used in real-time applications.

Increased reliability As data may be replicated in distributed systems, a single data item may exist at several sites. If one site fails, a transaction requiring a particular data item from that site may access it from other sites. Therefore, the failure of a node or a communication link does not necessarily make the data inaccessible; thus, reliability increases.

Better performance The data in a distributed system are dispersed to match business requirements; therefore, data are stored near the site where the demand for them is the greatest. As the data are located near the greatest-demand site and given the inherent parallelism of distributed DBMSs, speed of database access may be better than in a remote centralized database. Moreover, as each site handles only a part of the entire database, there may not be the same level of contention for CPU and I/O services that characterizes a centralized DBMS.

Reduced operating costs In the 1960s, computing power used to be estimated as being proportional to the square of the cost of equipments; hence, three times the cost would provide nine times the power. This was known as **Grosch's Law**. It is now generally accepted that the cost will be much lesser to create a system of smaller computers with computing power equivalent to that of a single large computer. Thus, it is more cost-effective for corporate divisions and departments to obtain separate computers. It is also much more cost-effective to add workstations to a network than to update a mainframe system. The second potential cost savings occurs where databases are geographically remote and the applications require access to distributed data. In such cases, owing to the relative expense of transmitting data across the network as opposed to the cost of

local access, it may be much more economic to partition the application and perform the processing locally at each site.

Distributed system reflects organizational structure Many organizations are by their nature distributed over several locations. For example, a banking organization can have several branches in different locations throughout the country. In such applications, it is beneficial if data are distributed over these locations and maintained locally. At the same time, the banking organization may wish to make global queries involving the access of data at all or a number of branches. In such organizations, building a distributed system is more beneficial as it reflects the organizational structure.

Integration of existing databases When several databases already exist in an organization and the necessity of performing global applications arises, distributed database is the natural solution. In this case, the distributed database is created from the pre-existing local databases using bottom-up approach. Although this process requires a certain degree of local restructuring, the effort required for the creation of a completely new centralized database is much higher than the effort required for local restructuring.

Processor independence A distributed system may contain multiple copies of a particular data item. Thus, in a distributed system, users can access any available copy of the data item, and user-requests can be processed by the processor at the data location. Hence, user-requests do not depend on a specific processor.

Modular extensibility Modular extension in a distributed system is much easier. New sites can be added to the network without affecting the operations of other sites. Such flexibility allows organizations to extend their system in a relatively rapid and easier way. Increasing data size can usually be handled by adding processors and storage power to the network.

A distributed DBMS has a number of disadvantages also. These are listed in the following.

Increased complexity Management of distributed data is a very complicated task than the management of centralized data. A distributed DBMS that hides the distributed nature from the users and provides an acceptable level of performance, reliability and availability is inherently more complex. In a distributed system, data may be replicated, which adds additional complexity to the system. All database related tasks such as transaction management, concurrency control, query optimization and recovery management are more complicated than in centralized systems.

Increased maintenance and communication cost The procurement and maintenance costs of a distributed DBMS are much higher than those of a centralized system, as complexity increases. Moreover, a distributed DBMS requires a network to afford communication among different sites. An additional ongoing communication cost is incurred owing to the use of this network. There are also additional labour costs to manage and maintain the local DBMSs and the underlying network.

Security As data in a distributed DBMS are located at multiple sites, the probability of security lapses increases. Further, all communications between different sites in a distributed DBMS are conveyed through the network, so the underlying network has to be made secure to maintain system security.

Lack of standards A distributed system can consist of a number of sites that are heterogeneous in terms of processors, communication networks (communication mediums and communication protocols) and DBMSs. This lack of standards significantly limits the potential of distributed DBMSs. There are also no tools or methodologies to help users to convert a centralized DBMS into a distributed DBMS.

Increased storage requirements Data replication in distributed DBMS requires additional disk storage space. This disadvantage is a minor one, because disk storage space is relatively cheap, and it is becoming cheaper. However, disk access and storage management in a widely dispersed data-storage environment are more complex than they would be in a centralized database.

Maintenance of integrity is very difficult Database integrity refers to the validity and consistency of stored data. Database integrity is usually expressed in terms of constraints. A constraint specifies a condition and a proposition that the database is not permitted to violate. Enforcing integrity constraints generally requires access to a large amount of data that define the constraint but are not involved in the actual update operation itself. The communication and processing costs that are required to enforce integrity constraints in a distributed DBMS may be prohibitive.

Lack of experience Distributed DBMSs have not been widely accepted, although many protocols and problems are well-understood. Consequently, we do not yet have the same level of experience in industry as we have with centralized DBMSs. For a prospective adopter of this technology, this may be a significant restriction.

Database design more complex The design of a distributed DBMS involves fragmentation of data, allocation of fragments to specific sites and data replication, besides the normal difficulties of designing a centralized DBMS. Therefore, the database design for a distributed system is much more complex than for a centralized system.

3.4 An Example of Distributed DBMS

Assume that an IT company has a number of branches in different cities throughout the country. Each branch has its own local system that maintains information regarding all the clients, projects and employees in that particular branch. Each such individual branch is termed a site or a node. All sites in the system are connected via a communication network. Each site maintains three relational schema: Project for project information, Client for client information and Employee for employee information, which are listed in the following.

Project = (project-id, project-name, project-type, project-leader-id, branch-no, amount)

Client = (client-id, client-name, client-city, project-id)

Employee = (emp-id, emp-name, designation, salary, emp-branch, project-no)

Here the underlined attributes represent the primary keys for the corresponding relations. There also exists one single site that maintains the information about all branches of the company. The single site containing information about all the clients, projects and employees in all branches of the company maintains an additional relational schema Branch, which is defined as follows:

Branch = (branch-no, branch-name, branch-city, no-of-projects, total-revenue)

Each local branch can access its local data via local applications as well as it can access data from other branches via global applications. During normal operations that are requested from a branch, local applications need only to access the database of that particular branch. These applications are completely executed by the processor of the local branch where they are initiated, and are therefore called local applications. Similarly, there exist many applications that require projects, clients and employees information from other branches also. These applications are called global applications or distributed applications.

3.5 Homogeneous and Heterogeneous Distributed DBMSs

A distributed DBMS may be classified as homogeneous or heterogeneous. In an ideal distributed database system, the sites would share a common global schema (although some relations may be stored only at some local sites), all sites would run the same database management software and the sites are aware of the existence of other sites. In a distributed system, if all sites use the same DBMS product, it is called a **homogenous distributed database system**. However, in reality, a distributed database has to be constructed by linking multiple already-existing database systems together, each with its own schema and possibly running different database management software. Such systems are called **heterogeneous distributed database systems**. In a heterogeneous distributed database system, sites may run different DBMS products that need not be based on the same underlying data model, and thus, the system may be composed of relational, network, hierarchical, and object-oriented DBMSs.

Homogeneous distributed DBMS provides several advantages such as simplicity, ease of designing and incremental growth. It is much easier to design and manage a homogeneous distributed DBMS than a heterogeneous one. In a homogeneous distributed DBMS, making the addition of a new site to the distributed system is much easier, thereby providing incremental growth. These systems also improve performance by exploiting the parallel processing capability of multiple sites.

Heterogeneous systems are usually constructed over a number of existing individual sites where each site has its own local databases and local DBMS software, and integration is considered at a later stage. To allow the necessary communications among the different sites in a heterogeneous distributed system, interoperability between different DBMS products is required. If a distributed system provides DBMS transparency, users are unaware of the different DBMS products in the system, and therefore it allows users to submit their queries in the language of the DBMS at their local sites. Achieving DBMS transparency increases system complexity. Heterogeneity may occur at different levels in a distributed database system as listed in the following.

- » different hardware
- » different DBMS software
- » different hardware and different DBMS software

If the hardware is different but the DBMS software is the same in a distributed system, the translation is straightforward between sites involving the change of codes and word lengths. Moreover, the differences among the sites at lower levels in a distributed system are usually managed by the communication software. Therefore, homogeneous distributed DBMS refers to a DDBMS with the same DBMS at each site, even if the processors and/or the operating systems are not the same.

If the DBMS software is different at different sites, the execution of global transactions becomes very complicated, as it involves the mapping of data structures in one data model to the equivalent data structures in another data model. For example, relations in the relational data model may be mapped into records and sets in the network data model. The translation of query languages is also necessary. If both the hardware and DBMS software are different at different sites in a distributed system, the processing of global transactions becomes extremely complex, because translations of both hardware and DBMS software are required. The provision of a common conceptual schema, which is formed by integrating individual local conceptual schemas, adds extra complexity to the distributed processing. The integration of data models in different sites can be very difficult owing to the semantic heterogeneity.

Some relational systems that are part of a heterogeneous distributed DBMS use **gateways**, which convert the language and data model of each different DBMS into the language and data model of the relational system. **Gateways** are mechanisms that provide access to other systems. In a gateway, one vendor (e.g., Oracle) provides single direction access through its DBMS to another database managed by a different vendor's DBMS (e.g., IBM DB2). The two DBMSs need not share the same data model. For example, many RDBMS vendors provide gateways to hierarchical and network DBMSs. However, the gateway approach has some serious limitations as listed below.

- The gateway between two systems may be only a query translator, that is, it may not support transaction management even for a pair of systems.
- The gateway approach generally does not address the issue of homogenizing the structural and representational differences between different database schemas; that is, it is only concerned with the problem of translating a query expressed in one language into an equivalent expression in another language.

In this context, one solution is a **multi-database system (MDBS)** that resides on top of existing databases and file systems and provides a single logical database to its users. An MDBS maintains only the global schema against which users issue queries and updates. [MDBSs are discussed in detail in **Chapter 6**.]

3.6 Functions of Distributed DBMS

A distributed DBMS manages the storage and processing of logically related data on interconnected computers wherein both data and processing functions are distributed among several sites. Thus, a distributed DBMS has at least all the functionality of a centralized DBMS. In addition, it must provide the following functions to be classified as distributed.

Application interfaces A distributed DBMS provides application interfaces to interact with the end-users or application programs and with remote databases within the distributed system. This feature is most important and is provided by all systems that are distributed in nature.

Validation and transformation techniques A distributed DBMS must contain necessary validation techniques to analyze data requests. It must also contain transformation techniques to determine which data request components are distributed and which ones are local.

Distribution transparency One major objective of the distributed system is to achieve distribution transparency; that is, users should be unaware of the distributed nature of the system. This feature is supported to a different extent by different distributed systems, because there is a strong trade-off between distribution transparency and performance.

Mapping and I/O interfaces A distributed DBMS must provide mapping techniques to determine the data location of local and remote fragments. It must also provide I/O interfaces to read or write data from or to permanent local storage.

Management of replicated data To maintain the consistency of replicated data items, a distributed DBMS must have the ability to decide which copy of a replicated data item is to be selected while executing a data request.

Extended communication services A distributed DBMS supports extended communication services to provide access to remote sites and allows the transfer of queries and data among the sites using a network.

Extended query processing and optimization The distributed DBMS supports query processing techniques to retrieve answers of local queries as well as global queries. It also provides query optimization both for local and global queries to find the best access strategy. In the case of global query optimization, the global query optimizer will determine which database fragments are to be accessed in order to execute global queries.

Distributed transaction management Transaction management facilities in distributed DBMSs ensure that the local and global transactions move data from one consistent state to another. Consistency of data should not be violated by local and distributed transactions. This consistency maintenance activity includes the synchronization of local and remote transactions as well as transactions across multiple distributed segments.

Distributed backup and recovery services Backup services in distributed DBMSs ensure the availability and reliability of a database in case of failures. Recovery services in distributed DBMSs take account of failures of individual sites and of communication links and preserve the database in the consistent state that existed prior to the failure.

Distributed concurrency control Concurrency control techniques manage simultaneous data access and ensure data consistency across database fragments in the distributed DBMS. They ensure that consistency of replicated data is maintained in spite of concurrent execution of local and global transactions in the distributed DBMS.

Support for global system catalog A distributed DBMS must contain a global system catalog to store data distribution details for the system. This feature includes tools for monitoring the database, gathering information about database utilization and providing a global view of data files existing at the various sites.

Support for global database administrator Global database administrator in distributed DBMS is responsible for maintaining the overall control of data and programs accessing the data in the distributed system.

Distributed security control A distributed DBMS offers security services to provide data privacy at both local and remote databases. This feature is used to maintain appropriate authorization/access privileges to the distributed data.

3.7 Components of a Distributed DBMS

A Distributed DBMS controls the storage and efficient retrieval of logically interrelated data that are physically distributed among several sites. Therefore, a distributed DBMS includes the following components.

(a) Computer workstations (sites or nodes) – A distributed DBMS consists of a number of computer workstations that form the network system. The distributed database system must be independent of the computer system hardware.

(b) Network components (both hardware and software) – Each workstation in a distributed system contains a number of network hardware and software components. These components allow

each site to interact and exchange data with each other site. Network system independence is a desirable property of the distributed system.

(c) Communication media – In a distributed system, any type of communication (data transfer, information exchange) among nodes is carried out through communication media. This is a very important component of a distributed DBMS. It is desirable that a distributed DBMS be communication media independent, that is, it must be able to support several types of communication media.

(d) Transaction processor (TP) – A TP is a software component that resides in each computer connected with the distributed system and is responsible for receiving and processing both local and remote applications' data requests. This component is also known as the **application processor (AP)** or the **transaction manager (TM)**.

(e) Data processor (DP) – A DP is also a software component that resides in each computer connected with the distributed system and stores and retrieves data located at that site. The DP is also known as the **data manager (DM)**. In a distributed DBMS, a DP may be a centralized DBMS.

The communication between TP and DP is incorporated through a specific set of rules or protocols used by the distributed DBMS. [The details about network protocols are introduced in **Chapter 4**.]

3.8 Date's 12 Objectives for Distributed Database Systems

Date identified 12 fundamental principles or objectives for distributed systems. He [Date] has stated that these 12 objectives are not all independent of one another, they are not necessarily exhaustive, nor are they all equally significant. Date's 12 commandments are very useful as a basis for understanding distributed database technology and as a framework for characterizing the functionalities of a specific distributed system. The discussion about distributed system will not be complete until Date's 12 commandments about distributed system are discussed, which are listed in the following.

(i) Local Autonomy – The sites in a distributed system should be autonomous. Local autonomy means that all operations at a given site should be managed by the particular site and no site should depend on some other site for its successful operation. Local autonomy also implies that local data is locally owned and managed with local accountability.

(ii) No Reliance on a Central Site – This feature implies that no site in the network relies on a central site, that is, there should be no one site in the distributed system without which the system cannot operate. There should be no central servers in a distributed system for services such as transaction management, deadlock detection, query optimization and management of the global system catalog.

(iii) Continuous Operations – Ideally, there should never be a requirement for a planned system shutdown for operations such as adding or removing a site from the distributed system, upgrading the DBMS at an existing site to a new release level, or the dynamic creation and deletion of fragments at one or more sites.

(iv) Location Independence – The basic idea of location independence is that users should not have any idea regarding the physical storage of data, but users must be able to access data

from all sites, no matter where it is physically stored. Location independence is equivalent to location transparency.

(v) **Fragmentation Independence** – Fragmentation independence implies that users should not be aware of the data fragmentation. Users must be able to access all data no matter how it is fragmented.

(vi) **Replication Independence** – Replication independence means that users should be unaware that data has been replicated. Thus, users must not be able to access a particular copy of a data item directly, nor they should specifically update all copies of a data item.

(vii) **Distributed Query Processing** – Distributed systems must be able to process distributed queries that references data from more than one site. Query optimization is performed transparently by the distributed DBMS.

(viii) **Distributed Transaction Processing** – A distributed DBMS must support transaction as the unit of recovery. In a distributed DBMS, both local and global transactions should ensure data consistency, that is, the system must ensure that both global and local transactions conform to the ACID property of transactions, namely, atomicity, consistency, isolation and durability.

(ix) **Hardware Independence** – Hardware independence ensures that it is possible to run the distributed DBMS on a variety of hardware platforms.

(x) **Operating System Independence** – This rule ensures that the distributed DBMS runs on a variety of operating systems.

(xi) **Network Independence** – Network independence ensures that it is possible to run the distributed DBMS on a variety of disparate communication networks.

(xii) **Database Independence** – This rule ensures that it is possible to have a distributed DBMS made up of different local DBMSs, perhaps supporting different underlying data models. In other words, the distributed system must support heterogeneity.

CHAPTER SUMMARY

» A DDBMS consists of a single logical database that is split into a number of **partitions** or **fragments**. Each partition is stored on one or more computers under the control of a separate DBMS, with the computers connected by a communication network.

» A distributed DBMS provides a number of advantages over centralized DBMS, but it has several disadvantages also.

» A distributed system can be classified as homogeneous distributed DBMS or heterogeneous distributed DBMS.

> In a distributed system, if all sites use the same DBMS product, it is called a **homogenous distributed database system**.

> In a heterogeneous distributed system, different sites may run different DBMS products, which need not be based on the same underlying data model.

» A distributed DBMS consists of the components computer workstations, computer network, communication media, transaction processor and data processor.

EXERCISES

Multiple Choice Questions

(i) Which of the following is a component of a distributed DBMS?
   a. Server
   b. Client
   c. Network
   d. All of the above.

(ii) Which of the following is a function of a distributed DBMS?
   a. Distributed query processing
   b. Replicated data management
   c. Distributed data recovery
   d. All of these.

(iii) Local autonomy means
   a. Local query will be processed locally
   b. Local data will be accessed by local site only
   c. Local DBA is sole authority of local data
   d. Each site is a DBMS in its own right.

(iv) A transaction processor is responsible for
   a. Receiving and processing only local applications' data requests
   b. Receiving and processing only remote applications' data requests
   c. Receiving and processing both local and remote applications' data requests
   d. None of the above.

(v) Which of the following statements is true?
   a. In a heterogeneous distributed DBMS, each site must have a different DBMS product
   b. In a heterogeneous distributed DBMS, at least more than half of the sites must have different DBMS products
   c. In a heterogeneous distributed DBMS, at least two sites must have different DBMS products
   d. None of the above.

(vi) Location independence represents
   a. Users are aware of the location of the data
   b. Users are unaware of the location of the data
   c. Users are aware of the physical names of database objects
   d. None of the above.

(vii) Which of the following statements is true?
   a. Network independence indicates that a distributed DBMS cannot work on a variety of disparate communication networks
   b. Network independence indicates that a distributed DBMS can work only in a homogeneous network environment
   c. Network independence indicates that a distributed DBMS can work on a variety of disparate communication networks
   d. All of the above
   e. None of the above.

(viii) Which of the following refers to the operation of copying and maintaining database objects in multiple physical databases belonging to a distributed system?
   a. Backup
   b. Recovery
   c. Replication
   d. All of these.

(ix) In a heterogeneous distributed DBMS
   a. Two different sites can use different DBMS products, but data model must be the same
   b. Two different sites can use different data model, but DBMS product must be the same
   c. Two different sites can use both different DBMS products and different data models
   d. Two different sites can use different DBMS products but database languages must be same.

(x) Which of the following is true?
   a. Global applications access only remote data
   b. Global applications access only local data
   c. Global applications can access both local and remote data
   d. All of the above.

(xi) Which of the following is not a benefit of site autonomy?
   a. A global catalog is not necessary to access local data
   b. Nodes can upgrade software independently
   c. Administrators can recover from isolated system failures independently
   d. There is no need for backup and recovery.

(xii) Which of the following statements is true?
   a. Remote data accessing and distributed database are the same
   b. Remote data accessing and parallel database are the same
   c. Remote data accessing and distributed processing are the same
   d. All of the above
   e. None of the above.

(xiii) Which of the following statements is correct?
a. In a homogeneous DDBMS, all sites must have same operating system and same DBMS
b. In a homogeneous DDBMS, all sites must have the same DBMS, but operating system may be different
c. All of the above
d. None of the above.

(xiv) Which of the following statements is false?
a. A transaction manager is responsible for receiving and processing both local and remote applications' data requests
b. A data processor stores and retrieves data located at that site
c. The DM and the TM both have the same responsibility
d. None of the above.

Review Questions

1. Define distributed database and distributed DBMS.
2. Discuss the circumstances and reasons when and why an organization with a centralized database system would prefer to move on to a distributed database environment. What are the possible disadvantages of such a decision?
3. Describe the relative advantages and disadvantages of distributed DBMSs.
4. Differentiate between reliability and availability.
5. Write down the differences between centralized and distributed DBMSs with respect to DBA, redundancy, indexing, reliability and performance.
6. What are the features of a distributed database?
7. What functionalities are provided by a distributed DBMS?
8. Explain the difference between distributed DBMS and distributed processing.
9. Differentiate between homogeneous and heterogeneous distributed DBMSs with examples.
10. Discuss the heterogeneity of local systems in a DDBMS from at least three different aspects.
11. Describe the different components of a distributed DBMS.
12. Discuss Date's 12 rules for distributed DBMSs.

4 Overview of Computer Networking

This chapter presents the fundamentals of computer networking. Different types of communication networks and different network topologies are briefly described. Network protocol is a set of rules that govern the data transmission between the nodes within a network. A wide range of network protocols that are used to transmit data within connection-oriented as well as within connectionless computer networks are represented. The concept of the internet and the world-wide web has also been introduced in this chapter.

The outline of this chapter is as follows. Section 4.1 introduces the fundamentals of computer networks. The types of computer networks are described in Section 4.2, and Section 4.3 introduces different communication schemes. In Section 4.4, network topologies are discussed, and the OSI model is represented in Section 4.5. The network protocols are briefly described in Section 4.6, and Section 4.7 represents the concept of the internet and the world-wide web.

4.1 Introduction to Networking

Data communications and networking is the fastest growing technology today. The rapid development in this technology changes the business scenario and the modern computing environment. Technological advancement in the network field merges data processing technology with data communication technology. The development of the internet and the world-wide web has made it possible to share information among millions of users throughout the world with thousands of machines connected to the network.

Data communication is the exchange of data between two devices via some form of transmission medium. To exchange data, the communicating devices must be part of some communication system that is made up of a combination of hardware and software. A data communication system is made up of five components. These are message, sender, receiver, medium and protocol. **Message** is the information to be communicated. **Sender** is the device that sends the message and **receiver** is the device that receives the message. **Transmission medium** is the physical path through which a message travels from the sender to the receiver. The transmission medium may be cables (twisted-pair wire, coaxial cable, fibre-optic cable), laser or radio waves. **Protocol** is a set of rules that determine how information is to be sent, interpreted and processed between the nodes in a network.

A **computer network** is the collection of devices referred to as nodes (or sites or hosts) connected by transmission media. A **node** can be a computer, a printer or any other device that is capable of receiving and/or sending information that is generated by other nodes in the network and/or by itself. The transmission media is often called communication channel or communication link. Computer networks use distributed processing in which a task is divided into several subtasks

that are executed in parallel on multiple computers connected to the network. Therefore, in the context of distributed database systems, it is necessary to introduce the fundamentals of computer networks.

4.2 Types of Computer Networks

Computer networks are broadly classified into three different categories. These are local area networks (LANs), metropolitan area networks (MANs) and wide area networks (WANs). This classification is done based on the geographical area (i.e., distance) covered by a network and the physical structure of the network.

Local Area Network (LAN) – A LAN usually connects the computers within a single office, building or campus. A LAN covers a small geographical area of a few kilometers. LANs are designed to share resources between personal computers or workstations. The resources that are shared within a LAN may be hardware, software or data. In addition to geographical area, LANs are distinguishable from other types of networks by their communication media and topology. The data transfer rates for LANs are 10 to 2,500 Mbps, and LANs are highly reliable.

Wide Area Network (WAN) – A WAN allows sharing of information over a large geographical area that may comprise a country, a continent or even the whole world. Since WANs cover a large geographical area, the communication channels in a WAN are relatively slow and less reliable than those in LANs. The data transfer rate for a WAN generally ranges from 33.6 kbps to 45 Mbps. WANs may utilize public, leased or private communication devices for sharing information, thereby covering a large geographical area. A WAN that is totally owned and used by a single company is called an **enterprise network.**

Metropolitan Area Network (MAN) – A MAN is a special case of the WAN that generally covers the geographical area within a city. A MAN may be a single network (e.g., a cable television network) or it may consist of several LANs so that information can be shared LAN-to-LAN as well as device-to-device. A MAN may be totally owned by a private company or it may be a service provided by a public company (e.g., local telephone company).

4.3 Communication Schemes

In terms of physical communication pathway, networks can be classified into two different categories: point-to-point (also called unicast) networks and broadcast (also called multi-point) networks.

A **point-to-point** network provides a dedicated link between each pair of nodes in the network. The entire capacity of the channel is reserved for the transmission between the two nodes. Hence, the receiver and the sender are identified by their addresses that are included in the frame header. Data transfer between two nodes follows one of the many possible links, some of which may involve visiting other intermediate nodes. The intermediate node checks the destination address in the frame header and passes it to the next node. This is known as **switching.** The most common communication media for point-to-point networks are coaxial cables or fiber-optic cables.

In **broadcast networks** a common communication channel is shared by all the nodes in the network. In such networks, the capacity of the channel is shared either spatially or temporarily among the nodes. If several nodes use the communication channel simultaneously, then it is called

a spatially shared broadcast network. A special case of broadcasting is multicasting where the message is sent to a subset of all the nodes in the network.

4.4 Network Topologies

The term **network topology** refers to the physical layout of a computer network. The topology of a network is the geometric representation of the relationships between all communication links and nodes. Five basic network topologies are star, ring, bus, tree and mesh.

Star Topology – In star topology, each node has a dedicated point-to-point connection only to a central controller node, known as the **hub;** but the nodes are not directly connected to each other. If one node wants to send some information to another node in the network, it sends the information to the central controller node, and the central controller node passes it to the destination node. The major advantage of the star topology is that it is less expensive and easier to install and reconfigure. Another benefit is that even if one node fails in star topology, others remain active; thus, it is easier to identify the fault and isolate the faulty node. However, possible failure of the central controller node is a major disadvantage. The star topology is illustrated in figure 4.1.

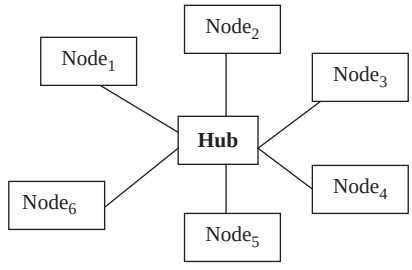


Fig. 4.1 Star Topology

Ring Topology – In ring topology, each node has a dedicated connection with two neighboring nodes on either side of it. The data transfer within a ring topology is unidirectional. Each node in the ring incorporates a repeater, and when a signal passes through the network, it receives the signal, checks the address and copies the message if it is the destination node or retransmits it. In ring topology, the data transmission is controlled by a **token.** The token circulates in the ring all the time, and the bit pattern within the token represents whether the network is free or in use. If a node wants to transmit data it receives the token and checks the bit pattern of it. If it indicates that the network is free, the node changes the bit pattern of the token and then puts the data on the ring. After transmitting the data to the receiver, the token again comes back to the sender, and the sender changes the bit pattern of the token to indicate that the network is free for further transmission. A ring topology is relatively easy to install and reconfigure. However, unidirectional traffic can be a major disadvantage. Further, a break in the ring can disable the entire network. This problem can be solved by using a dual ring or a switch. The ring topology is depicted in figure 4.2.

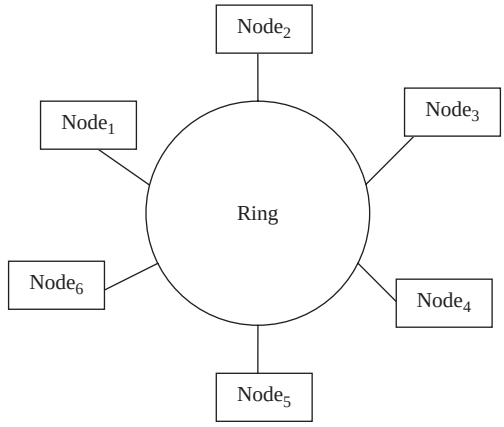


Fig. 4.2 Ring Topology

Bus Topology – In bus topology, all the nodes in the network are connected by a common communication channel, known as the **backbone.** Bus topology uses multi-point communication scheme. In this topology, the nodes are connected to the backbone by drop lines and taps. A drop line is a connection between a node and the backbone. A tap is a connector that is used to make contact with the metallic core. The advantage of bus topology is its ease of installation. The major disadvantage of this topology is that it is very difficult to isolate a fault and reconfigure the network. Furthermore, a fault or break in the backbone stops all transmission within the network. The bus topology is illustrated in figure 4.3.

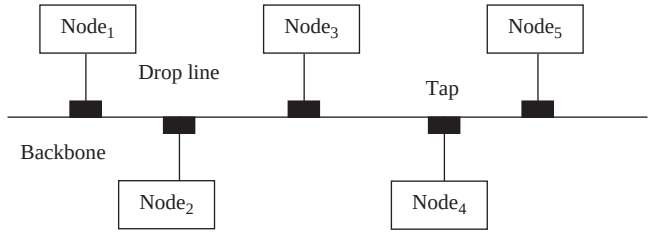


Fig. 4.3 Bus Topology

Tree Topology – Tree topology is a variation of star topology. In tree topology, every node is not directly connected to the central hub. A majority of the nodes are connected to the central hub via a secondary hub. The central hub is the **active hub** and it contains a repeater that regenerates the bit patterns of the received signal before passing them out. The secondary hubs may be active or passive. A passive hub simply provides a physical connection between the attached nodes. The addition of secondary hubs provides extra advantages. First, it increases the number of nodes that can be attached to the central hub. Second, it allows the network to isolate and prioritize communications from different computers. The tree topology is illustrated in the figure 4.4.

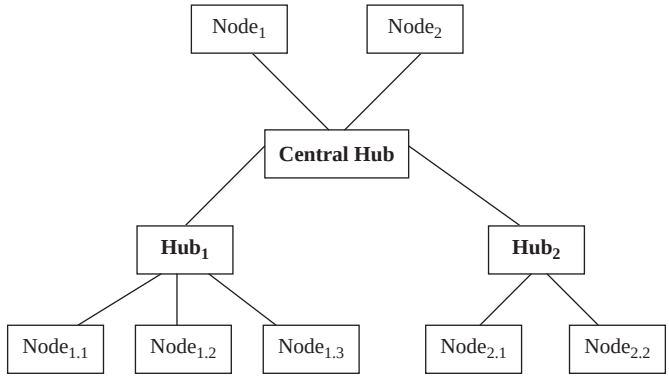


Fig. 4.4 Tree Topology

Mesh Topology – In mesh topology, each node has a dedicated point-to-point connection with every other node in the network. Therefore, a fully connected mesh network with n nodes has $n\,(n-1)/2$ physical communication links. Mesh topology provides several advantages over other topologies. First, the use of dedicated links guarantees that each link carries only its own data load, thereby eliminating traffic problems. Second, mesh topology is robust; even if one link fails, other links remain active. Third, in a mesh network, as data are transmitted through dedicated links, privacy and security are inherently maintained. Finally, fault detection and fault isolation are much easier in mesh topology. The main disadvantage of mesh topology is that it requires a large number of cables and I/O ports; thus, installation and reconfiguration are very difficult. Another disadvantage is that it is more costly than other topologies. Hence, mesh networks are implemented in limited applications. The mesh topology is depicted in figure 4.5.

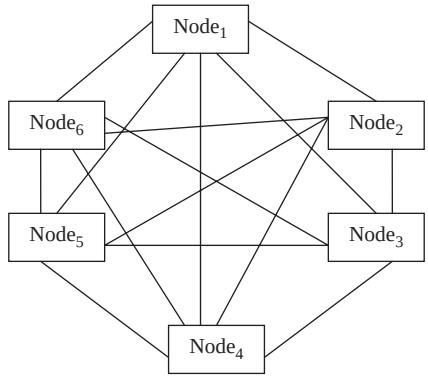


Fig. 4.5 Mesh Topology

In large networks, hybrid topologies, which are combinations of several topologies, are also adopted.

4.5 The OSI Model

The International Organization for Standardization has defined a protocol that determines how two different systems will communicate with each other regardless of their underlying architecture. The protocol is known as **open systems interconnection model (OSI model),** in which the network is divided into a series of layers. The OSI model consists of seven ordered layers. These are **application layer, presentation layer, session layer, transport layer, network layer, data link layer** and **physical layer.** The lowest three layers, the network, data link and physical layers are together known as the **communication subnet.** The passing of the data and network information between the layers is facilitated by an **interface** between each pair of adjacent layers. The architecture of the OSI model is illustrated in figure 4.6.

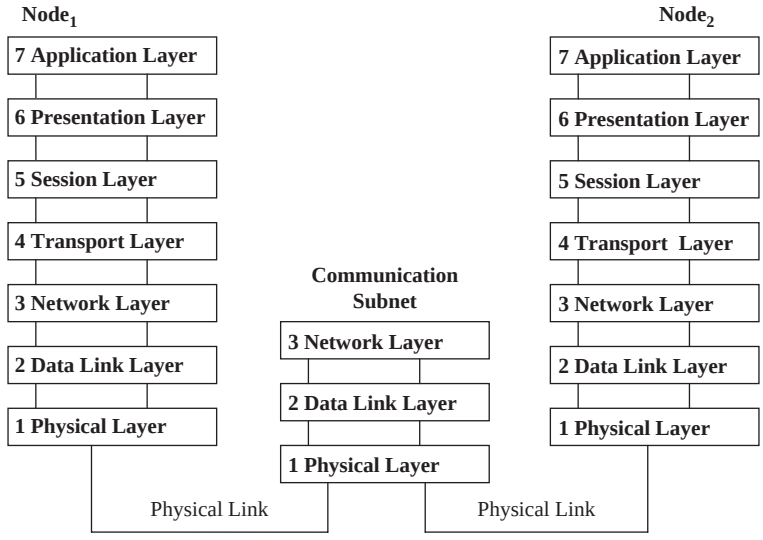


Fig. 4.6 The OSI Model Architecture

Each layer in the OSI model provides a particular service to its upper layer, hiding the detailed implementation. The application layer provides user interfaces and allows users to access network resources. It supports a number of services such as electronic mail, remote-file access, distributed information services and shared database management. The presentation layer deals with the syntax and semantics of the information that is to be exchanged between two systems. The responsibility of this layer is to transform, encrypt and compress data. Session establishment and termination are controlled by the session layer. The transport layer is responsible for the delivery of the entire message from source to destination, and the network layer is responsible for the delivery of packets from source to destination across multiple networks. The data link layer organizes bits into frames and provides node-to-node delivery. The physical layer deals with the mechanical and electrical specifications of the interface and the communication medium that are required to transmit a bit stream over a physical medium. The details of these layers are not discussed here. Interested readers can consult [Tanenbaum, 1997] or any other network books available in the market.

4.6 Network Protocols

A network protocol is a set of rules that determine how information between nodes are sent, interpreted and processed. It represents an agreement between the communicating nodes. The key elements of a protocol are syntax, semantics and timing. The **syntax** of a protocol represents the structure or format of the data that is to be communicated. The **semantics** of a protocol refers the meaning of each section of bits, and the **timing** of a protocol represents when data should be sent and how fast it should be sent. Several network protocols are described in the following section.

4.6.1 TCP/IP (Transmission Control Protocol/Internet Protocol)

The TCP and IP are a pair of protocols that allow one subnet to communicate with another (each part of the internet). TCP/IP was originally developed by the US Defense Advanced Research Projects Agency (**DARPA**). Their objective was to connect a number of universities and other research establishments to DARPA. The resultant network is now known as the **internet**. The internet uses TCP/IP as a standard protocol to transfer data. Each node on the internet is assigned a unique network address, called an **IP address.** A typical IP address consists of two fields: left field and right field. The left field (or the network number) identifies the network and the right field (or the host number) identifies the particular host within that network. The IP address is 32 bits long, and it can address over four billion physical addresses. TCP/IP is composed of the following layers:

- **IP** – IP is responsible for moving packets from node to node. IP forwards each packet based on a four-byte destination address (the IP number). The internet authorities assign ranges of numbers to different organizations. The organizations assign groups of their numbers to departments. IP operates on gateway machines that move data from department to organization, organization to region, and then around the world.
- **TCP** – TCP is responsible for verifying the correct delivery of data from client to server. Data can be lost in the intermediate network. TCP can detect errors and loss of data and retransmit the data until the data is correctly and completely received by the destination node.
- **Sockets** – Sockets is the name given to the package of subroutines that provide access to TCP/IP on most systems.

The TCP part and the IP part correspond to the transport layer and the network layer, respectively, of the OSI model. In TCP/IP communications, the common applications are remote login and file transfer.

4.6.2 SPX/IPX (Sequence Packet Exchange/Internetwork Packet Exchange)

SPX/IPX is a routable protocol, and it can be used for small and large networks. SPX/IPX was developed by Novell and it was derived from the Xerox XNS network protocol family. SPX is the transport layer protocol, and IPX is the network layer protocol. SPX, like TCP, is a connection-oriented protocol that ensures correct data delivery. It controls the integrity of packets and acknowledges packets received. When no acknowledgement is received, a packet is retransmitted until a fixed number of retransmission has been done.

IPX is peer-to-peer, and an IPX network is a server-based network. It provides fast, unreliable communication with network nodes using a connectionless datagram service. IPX is responsible

for network addressing (both internetwork addressing and intranode addressing) and routing. This protocol completely relies on the network hardware for the actual node addressing. IPX uses an 80-bit address space with a 32-bit network portion and a 48-bit host portion. One major advantage of this protocol is automatic host addressing. IPX is best suited for mobile users.

4.6.3 NetBIOS (Network Basic Input/Output System)

In 1983, NetBIOS protocol was developed by IBM and Sytek. It is effectively an application interface to access LAN facilities for personal computers. In 1985, NetBEUI (NetBIOS extended user interface) was designed by IBM as the network protocol that completes the requirements to transport NetBIOS sessions across the network. Originally NetBIOS and NetBEUI were considered as one protocol. Nowadays, NetBIOS sessions can be transported over TCP/IP and SPX/IPX protocols. NetBEUI is a small, fast and efficient protocol, but it is not routable.

NetBIOS supports both connection-oriented and connectionless communications using broadcasts and multi-casts. It is peer-to-peer, and sessions can occur between any two NetBIOS nodes on a network; there is no hierarchy. NetBIOS supports the following services:

- **Naming services** – Names are necessary for identification, can be specific to the node (**unique name**) or for a group of nodes (**group name**) and can be changed. Naming services are provided by **name management protocol (NMP).**
- **Session services** – Session services provide a connection-oriented, reliable, full duplex message service to user process using **session management protocol (SMP).**
- **Datagram services** – Using **user datagram protocol (UDP),** datagrams can be sent to a specific name or all members of a group, or can be broadcast to the entire LAN.
- **Diagnostic services** – Diagnostic services provide the ability to query the status of nodes on the network using **diagnostic and monitoring protocol (DMP).**

4.6.4 APPC (Advanced Program-to-Program Communications)

APPC is a high-level communication protocol that allows one program to interact with another program across the network. It allows user-written programs to perform transactions in a client-server and distributed computing environment by providing a common programming interface across all IBM platforms. APPC is a useful tool to achieve efficient connectivity for mainframe computers such as CICS, MVS "batch" and AS/400s. It combines small operations into larger messages and thereby reduces overhead and increases throughput.

APPC provides commands for managing a session, sending and receiving data, and transaction management using a two-phase commit protocol. Instead of establishing a new session for every request, in APPC communication is managed through a subsystem. The subsystem maintains a queue of long-running sessions between the user machine and subsystems on the server machine. In most of the cases, a new request is sent through an existing session. However, there is some overhead as programs communicate to the subsystem, but it is much lesser than the cost of creating and ending new sessions constantly. The smallest APPC transaction consists of two operations: allocate and deallocate. The **allocate** operation acquires temporary ownership of one of the existing sessions on the server node. The **deallocate** operation frees the session and ends the conversation. Such a transaction requires that a program be run on the server node, but it provides no data and

does not wait for a response. APPC programs use two statements to send and receive data. These are **Send_Data** and **Receive_and_Wait. LU 6.2** is a set of SNA parameters used to support APPC when it runs on IBM's System Network Architecture (SNA) network; thus, sometimes APPC and LU 6.2 are considered identical.

4.6.5 DECnet

DECnet is a group of data communication products including a protocol suite, developed and supported by Digital Equipment Corporation (Digital). The first version of DECnet was released in 1975, which allowed two directly attached PDP-11 minicomputers to communicate. In recent years, Digital has started supporting several non-proprietary protocols. DECnet is a series of products that conform to DEC's Digital Network Architecture (DNA). DNA supports a variety of media and link implementations such as Ethernet and X.25. DNA also offers a proprietary point-to-point link-layer protocol called Digital Data Communications Message Protocol (DDCMP) and a 70-Mbps bus used in the VAXcluster called the computer-room interconnect bus (CI bus).

DECnet routable protocol supports two different types of nodes. These are end nodes and routing nodes. Both end nodes and routing nodes can send and receive network information, but routing nodes can provide routing services for other DECnet nodes. Unlike TCP/IP and some other network protocols, DECnet addresses are not associated with the physical network to which the nodes are connected. Instead, DECnet locates hosts using area and node address pairs. Areas can span many routers, and a single cable can support many areas. DECnet is currently in its fifth major product release, which is known as Phase V and DECnet/OSI.

4.6.6 AppleTalk

In the early 1980s, AppleTalk, a protocol suite, was developed by Apple Computer in conjunction with the Macintosh Computer. AppleTalk is one of the early implementations of a distributed client/server networking system. AppleTalk is Apple's LAN routable protocol that supports Apple's proprietary LocalTalk access method as well as Ethernet and token ring technologies.

AppleTalk networks are arranged hierarchically. The four basic components that form the basis of an AppleTalk network are sockets, nodes, networks and zones. AppleTalk utilizes addresses to identify and locate devices on a network in a manner similar to the one utilized by the common protocols such as TCP/IP and IPX. The AppleTalk network manager and the LocalTalk access method are built into all Macintoshes and Laser Writers.

4.6.7 WAP (Wireless Application Protocol)

In 1997, **Wireless Application Protocol Forum (WAP Forum)** was established by mobile equipments manufactures to define an architecture that extends the internet technology for mobile devices. The WAP architecture provides an optimized protocol stack for communicating over wireless channels, an application environment for mobile phone applications, a content description language and a miniature browser interface. It has grown into a complex specification within a short period of time, with many overlaps with the global Internet architecture. The WAP architecture is similar to OSI model architecture. This layered architecture model specifies how information flows from a web application to an application residing on a mobile phone.

The WAP protocol stack is designed to operate with a variety of bearer services with emphasis on low-speed mobile communication, but is most suited for packet-switched bearer services. The wireless datagram protocol (WDP) operates above the bearer services and provides a connectionless unreliable datagram service similar to UDP, transporting data from a sender to a receiver. The wireless transaction protocol (WTP) in a WAP stack can be considered as equivalent to the TCP layer in the IP stack, but it is optimized for low bandwidth. To run a WAP application, a complex infrastructure consisting of a mobile client, a public land–mobile network (such as GSM), a public switched telephony network (such as ISDN), a WAP gateway, an IP network and a WAP application server are required in addition to the system components WAP portals, proxy servers, routers and firewalls. WAP also provides a couple of advanced security features that are not available in the IP environment. Wireless markup language (WML) is an XML-based markup language that is specially designed for small mobile devices. In short, WAP provides a complete environment for wireless applications.

4.7 The Internet and the World-Wide Web (WWW)

The Internet consists of a number of separate but interconnected networks that belong to different commercial, educational and government organizations throughout the world. In other words, the Internet is a network of networks. Each of these separate networks has its own characteristics and set of rules. Thus, the Internet is a heterogeneous combination of networks. The Internet provides several services such as electronic mail, video conferencing, chat, file transfer and the ability to access remote computers. In the late 1960s and early 1970s, US Department of Defense introduced the Internet as ARPANET (Advanced Research Projects Agency NETwork). From 1982, TCP/IP was adopted as the standard communication protocol for the Internet. Today, TCP/IP refers to the entire Internet suite of protocols that are commonly run on TCP/IP such as FTP (file transfer protocol), SMTP (simple mail transfer protocol), Telnet (telecommunication network) and DNS (domain name service).

The World-Wide Web (WWW or the Web) is a hyper media-based system, which is a repository of information spread all over the world and linked together. The WWW is a distributed client–server service where clients can access a service by using a **web browser.** The service is provided by a number of **web servers** that are distributed over many locations. A web browser is the software that provides the facility to access web resources. The information on the Web is stored in the form of **web pages,** which are a collection of text, graphics, pictures, sound and video. Web pages also contain links to other web pages, called **hyperlinks.** A **website** is a collection of several web pages. The information on web pages is generally represented by a special language other than simple text, called **hypertext markup language (HTML).** The protocol that determines the exchange of information between the web server and the web browser is called **hypertext transfer protocol (HTTP).** The location or address of each resource on the Web is unique and it is represented by a **uniform resource locator (URL).** The World-Wide Web has established itself as the most popular way of accessing information via the Internet.

CHAPTER SUMMARY

» **Data communication** is the exchange of data between two devices via some form of transmission medium. A data communication system is made up of five components, namely, message, sender, receiver, medium and protocol.

» A **computer network** is a collection of nodes (or sites or hosts) connected by transmission media that are capable of receiving and/or sending information that are generated by other nodes in the network and/or itself.

» Computer networks are broadly classified into three different categories: LANs, WANs and MANs. Depending on the communication pathway, networks can also be classified into two different categories: point-to-point networks and broadcast networks.

» Network topology represents the physical layout of a computer network. There are five different network topologies: star, bus, ring, tree and mesh. Sometimes, hybrid topologies are also adopted in specific networks.

» A protocol is a set of rules that governs the data transmission between the nodes in a network. **The OSI Model** is a protocol that determines how two different systems will interact with each other regardless of their underlying architecture.

» **The Internet** consists of a number of separate but interconnected networks. **The World-Wide Web (WWW)** is a repository of information spread all over the world and linked together.

EXERCISES

Multiple Choice Questions

(i) Which of the following statements is correct?
   a. In a point-to-point network one common communication channel is shared by all nodes
   b. In a broadcast network, there is a dedicated connection between each pair of nodes
   c. In a point-to-point network, there is a dedicated connection between each pair of nodes
   d. All of the above.

(ii) An enterprise network is
   a. A LAN owned by a particular organization
   b. A WAN owned by a particular organization
   c. A MAN owned by a particular organization
   d. All of the above
   e. None of the above.

(iii) Which of the following statements is true?
   a. In ring topology all connections are established via a central hub
   b. In bus topology there is a dedicated connection between each pair of nodes
   c. In star topology there is a dedicated connection with neighboring nodes
   d. In star topology each connection is made via a central hub.

(iv) Which of the following statements is false?
   a. In tree topology there may be more than one hub
   b. In mesh topology there is a dedicated connection between each pair of nodes
   c. In ring topology, there is a dedicated connection with each neighboring node
   d. None of these
   e. All of these.

(v) Multi-casting is a special type of
   a. Point-to-point network
   b. Broadcasting network
   c. Uni-casting network
   d. None of the above.

(vi) Token is used in
   a. Bus topology
   b. Star topology
   c. Tree topology
   d. Ring topology.

(vii) Backbone is used in
   a. Bus topology
   b. Star topology
   c. Mesh topology
   d. Ring topology.

(viii) The total number of communication links with n nodes in a mesh topology is
   a. $n * n$
   b. $2n$
   c. $n * (n-1)/2$
   d. None of these.

(ix) In the OSI model, the communication subnet consists of
   a. Application layer, network layer and physical layer
   b. Session layer, application layer and presentation layer
   c. Network layer, data link layer and physical layer
   d. Application layer, network layer and data link layer.

(x) Which of the following statements is incorrect?
   a. The network layer is responsible for delivery of packets

b. The data link layer provides user interfaces
c. The physical layer deals with the mechanical and electrical specifications of the interface and the communication medium
d. The transport layer is responsible for delivery of messages.

(xi) The key elements of a network protocol are
a. Syntax, network topology and network category
b. Syntax, semantics and timing
c. Semantics, network topology and network category
d. Timing, network topology and syntax.

(xii) Which of the following protocols is not routable?
a. SPX/IPX
b. TCP/IP
c. NetBIOS
d. None of these.

(xiii) Which of the following statements is true?
a. IP address is 48 bits long
b. IP address is 16 bits long
c. IP address is 24 bits long
d. IP address is 32 bits long.

(xiv) Sockets is the name of a
a. Secure communication channel
b. Network protocol
c. The package of subroutines that provide access to TCP/IP
d. None of these.

(xv) On the Internet, services are provided by
a. Web sites
b. Web browsers
c. Web pages
d. None of these.

(xvi) Which of the following services is provided by NetBIOS protocol?
a. Naming services
b. Datagram services
c. Session Services
d. All of these.

(xvii) Which of the following statements is false?
a. URL represents the unique address of each resource on the Internet
b. A web site is a collection of several wed pages
c. Web servers are not physically distributed
d. None of these.

(xviii) WML is
a. WAP Manipulation Language
b. WAP Markup Language
c. Wireless Manipulation Language
d. Wireless Markup Language.

Review Questions

1. Describe the components of a data communication system.
2. Compare and contrast LAN, MAN and WAN.
3. Define network topology. Compare and contrast different network topologies.
4. Differentiate between point-to-point and multi-point networks.
5. Explain the key elements of a protocol. Describe TCP/IP protocol.
6. What are the different services provided by a NetBIOS protocol?
7. Explain how World-Wide Web is related to the Internet.
8. Write a short note on Wireless Application Protocol.

5 Distributed Database Design

This chapter introduces the basic principles of distributed database design and related concepts. All distributed database design concepts, such as fragmentation, replication, and data allocation are discussed in detail. The different types of fragmentations are illustrated with examples. The benefits of fragmentation, objectives of fragmentation, different allocation strategies, and allocation of replicated and non-replicated fragments are explained here briefly. Different types of distribution transparencies have also been focused in this chapter.

The outline of this chapter is as follows. Section 5.1 represents the basic concepts of distributed database design. The objectives of data distribution are introduced in Section 5.2. In Section 5.3, data fragmentation – one important issue in distributed database design – is explained briefly with examples. Section 5.4 focuses on the allocation of fragments, and the measure of costs and benefits of fragment allocation. In Section 5.5, different types of distribution transparencies are represented.

5.1 Distributed Database Design Concepts

In a distributed system, data are physically distributed among several sites but it provides a view of single logical database to its users. Each node of a distributed database system may follow the three-tier architecture like the centralized database management system (DBMS). Thus, the design of a distributed database system involves the design of a global conceptual schema, in addition to the local schemas, which conform to the three-tier architecture of the DBMS in each site. The design of computer network across the sites of a distributed system adds extra complexity to the design issue. The crucial design issue involves the distribution of data among the sites of the distributed system. Therefore, the design and implementation of the distributed database system is a very complicated task and it involves three important factors as listed in the following.

- **Fragmentation** – A global relation may be divided into several non-overlapping subrelations called fragments, which are then distributed among sites.
- **Allocation** – Allocation involves the issue of allocating fragments among sites in a distributed system. Each fragment is stored at the site with optimal distribution.
- **Replication** – The distributed database system may maintain several copies of a fragment at different sites.

The definition and allocation of fragments must be based on how the database is to be used. After designing the database schemas, the design of application programs is required to access

and manipulate the data into the distributed database system. In the design of a distributed database system, precise knowledge of application requirements is necessary, since database schemas must be able to support applications efficiently. Thus, the database design should be based on both quantitative and qualitative information, which collectively represents application requirements. Quantitative information is used in allocation, while qualitative information is used in fragmentation. The quantitative information of application requirements may include the following:

- The frequency with which a transaction is run, that is, the number of transaction requests in the unit time. In case of general applications that are issued from multiple sites, it is necessary to know the frequency of activation of each transaction at each site.
- The site from which a transaction is run (also called site of origin of the transaction).
- The performance criteria for transactions.

The qualitative information of application requirements may include the following information about the transactions that are executed:

- The relations, attributes, and tuples accessed by the transactions.
- The type of access (read or write).
- The predicates of read operations.

Characterizing these features is not trivial. Moreover, this information is typically given for global relation and must be properly translated into terms of all fragmentation alternatives that are considered during database design.

5.1.1 Alternative Approaches for Distributed Database Design

The distributed database design involves making decisions on the fragmentation and placement of fragmented data across the sites of a computer network. Two alternative design issues have been identified for the distributed database design, namely, top-down and bottom-up design process.

Top-down design process – In this process, the database design starts from the global schema design and proceeds by designing the fragmentation of the database, and then by allocating the fragments to the different sites, creating the physical images. The process is completed by performing the physical design of the data at each site, which is allocated to it. The global schema design involves both designing of global conceptual schema and global external schemas (view design). In global conceptual schema designing step, the user needs to specify the data entities and to determine the applications that will run on the database as well as statistical information about these applications. At this stage, the design of local conceptual schemas is considered. The objective of this step is to design local conceptual schemas by distributing the entities over the sites of the distributed system. Rather than distributing relations, it is quite common to partition relations into subrelations, which are then distributed to different sites. Thus, in a top-down approach, the distributed database design involves two phases, namely, **fragmentation** and **allocation.**

The fragmentation phase is the process of clustering information in fragments that can be accessed simultaneously by different applications, whereas the allocation phase is the process of distributing

the generated fragments among the sites of a distributed database system. In the top-down design process, the last step is the physical database design, which maps the local conceptual schemas into physical storage devices available at corresponding sites. Top-down design process is the best suitable for those distributed systems that are developed from scratch.

Bottom-up design process – In the bottom-up design process, the issue of integration of several existing local schemas into a global conceptual schema is considered to develop a distributed system. When several existing databases are aggregated to develop a distributed system, the bottom-up design process is followed. This process is based on the integration of several existing schemas into a single global schema. It is also possible to aggregate several existing heterogeneous systems for constructing a distributed database system using the bottom-up approach. Thus, the bottom-up design process requires the following steps:

- The selection of a common database model for describing the global schema of the database
- The translation of each local schema into the common data model
- The integration of the local schemas into a common global schema.

Any one of the above design strategies is followed to develop a distributed database system.

5.2 Objectives of Data Distribution

In a distributed system, data may be fragmented, and each fragment can have a number of replicas to increase data availability and reliability. The following objectives must be considered while designing the fragmentation and allocation of these fragments to different sites in a distributed system (i.e., during the design of data distribution).

(a) Locality of reference – To maximize the locality of references, whenever possible, data should be stored close to where it is used during data distribution. If a fragment is used at several sites, it may be beneficial to store copies of the fragment at different sites.

(b) Improved availability and reliability of distributed data – During data distribution, reliability and availability are improved by replication. A higher degree of availability can be achieved by storing multiple copies of the same information in different sites. Reliability is also achieved by storing multiple copies of the same information, since it is possible to continue the normal operations of a particular site in case of site failure, by referencing the copy of the same information from other sites.

(c) Workload distribution and improved performance – Bad allocation may result in underutilization of resources, and thereby system performance may degrade. Distributing the workload over the sites is an important feature of distributed database system. Workload distribution should be done to take advantage of system resources at each site and to maximize the degree of parallelism of execution of transactions. Since workload distribution might negatively affect processing locality, it is necessary to consider the trade-off between them during the design of data distribution.

(d) Balanced storage capacities and costs – Database distribution must reflect the cost and availability of storage at different sites. The availability and cost of storage at each site must be considered so that cheap mass storage can be used, whenever possible. This must be balanced against locality of reference.

(e) Minimal communication costs – The cost of processing remote requests must be considered during data distribution. Retrieval cost is minimized when locality of reference is maximized or when each site has its own copy of the data. However, when replicated data are updated, the update has to be performed at all sites holding a replica, thwereby increasing the communication costs.

5.2.1 Alternative Strategies for Data Allocation

Four alternative strategies have been identified for data allocation. This section describes these different data allocation strategies and also draws a comparison between them.

(i) Centralized – In this strategy, the distributed system consists of a single database and DBMS is stored at one site with users distributed across the communication network. Remote users can access centralized data over the network; thus, this strategy is similar to **distributed processing.**

In this approach, locality of reference is the lowest at all sites, except the central site where the data are stored. The communication cost is very high since all users except the central site have to use the network for all types of data accesses. Reliability and availability are very low, since the failure of the central site results in the loss of entire database system.

(ii) Fragmented (or Partitioned) – This strategy partitions the entire database into disjoint fragments, where each fragment is assigned to one site. In this strategy, fragments are not replicated.

If fragments are stored at the site where they are used most frequently, locality of reference is high. As there is no replication of data, storage cost is low. Reliability and availability are also low but still higher than centralized data allocation strategies, as the failure of a site results in the loss of local data only. In this case, communication costs are incurred only for global transactions. However, in this approach, performance should be good, and communication costs are low if the data distribution is designed properly.

(iii) Complete replication – In this strategy, each site of the system maintains a complete copy of the entire database. Since all the data are available at all sites, locality of reference, availability and reliability, and performance are maximized in this approach.

Storage costs are very high in this case, and hence, no communication costs are incurred due to global transactions. However, communication costs for updating data items are the most expensive. To overcome this problem, **snapshots** are sometimes used. A snapshot is a copy of the data at a given time. The copies are updated periodically, so they may not be always up-to-date. Snapshots are also sometimes used to implement views in a distributed database, to reduce the time taken for performing a database operation on a view.

(iv) Selective replication – This strategy is a combination of centralized, fragmented, and complete replication strategies. In this approach, some of the data items are fragmented and allocated to the sites where they are used frequently, to achieve high localization of reference. Some of the data items or fragments of the data items that are used by many sites simultaneously but not frequently updated, are replicated and stored at all these different sites. The data items that are not used frequently are centralized.

The objective of this strategy is to utilize all the advantages of all other strategies but none of the disadvantages. This strategy is used most commonly because of its flexibility [see **table 5.1].**

Table 5.1 Comparison of Strategies for Data Allocation

	Locality of reference	Reliability and availability	Workload distribution and performance	Storage costs	Communication costs
Centralized	Lowest	Lowest	Poor	Lowest	Highest
Fragmented	High	Low for data item, high for system	Satisfactory	Lowest	Low
Complete replication	Highest	Highest	Best for reading	Highest	High for updating low for reading
Selective replication	High	Low for data item, high for system	Satisfactory	Average	Low

5.3 Data Fragmentation

In a distributed system, a global relation may be divided into several non-overlapping subrelations and allocated to different sites, called **fragments.** This process is called **data fragmentation.** The objective of data fragmentation design is to determine non-overlapping fragments, which are logical units of allocation. Fragments can be designed by grouping a number of tuples or attributes of relations. Each group of tuples or attributes that constitute a fragment has the same properties.

5.3.1 Benefits of Data Fragmentation

Data fragmentation provides a number of advantages that are listed in the following.

(i) **Better usage** – In general, applications work with views, rather than the entire relation. Therefore, it seems to be beneficial to fragment the relations into subrelations and store them into different sites as units of distribution in data distribution.

(ii) **Improved efficiency** – Fragmented data can be stored close to where it is most frequently used. In addition, data that are not required by local applications are not stored locally, which may result in faster data access, thereby increasing the efficiency.

(iii) **Improved parallelism or concurrency** – With a fragment as the unit of distribution, a transaction can be divided into several subtransactions that operate on different fragments in parallel. This increases the degree of concurrency or parallelism in the system, thereby allowing transactions to execute in parallel in a safe way.

(iv) **Better security** – Data that are not required by local applications are not stored locally and, consequently, are not available to unauthorized users of the distributed system, thus, improving the security.

Fragmentation provides several disadvantages also. Owing to data fragmentation, performance can degrade, and maintaining integrity may become difficult.

(i) Performance degradation – The performance of global applications that require data from several fragments located at different sites may become slower and, therefore, performance may degrade. For example, in case of global applications, it may be necessary to retrieve data from more than one fragment located at different sites. In this case, performing either union or join of these fragments becomes slower and costly.

(ii) Maintaining integrity becomes difficult – Owing to data fragmentation, data and functional dependencies may be decomposed into different fragments that might be allocated to different sites across the network of a distributed system. In this situation, checking functional dependencies is very complicated and, consequently, maintaining integrity becomes very difficult.

5.3.2 Correctness Rules for Data Fragmentation

To ensure no loss of information and no redundancy of data (i.e., to ensure the correctness of fragmentation), there are three different rules that must be considered during fragmentation. These correctness rules are listed below.

(a) Completeness – If a relation instance R is decomposed into fragments $R_1, R_2, \ldots, R_n$, each data item in R must appear in at least one of the fragments R_i. This property is identical to the lossless decomposition property of normalization and it is necessary in fragmentation to ensure that there is no loss of data during data fragmentation.

(b) Reconstruction – If a relation R is decomposed into fragments $R_1, R_2, \ldots, R_n$, it must be possible to define a relational operation that will reconstruct the relation R from the fragments $R_1, R_2, \ldots, R_n$. This rule ensures that constraints defined on the data in the form of functional dependencies are preserved during data fragmentation.

(c) Disjointness – If a relation instance R is decomposed into fragments $R_1, R_2, \ldots, R_n$, and if a data item is found in the fragment R_i, then it must not appear in any other fragment. This rule ensures minimal data redundancy. In case of vertical fragmentation, primary key attribute must be repeated to allow reconstruction and to preserve functional dependencies. Therefore, in case of vertical fragmentation, disjointness is defined only on non-primary key attributes of a relation.

5.3.3 Different Types of Fragmentation

This section introduces different types of fragmentation. There are two main types of fragmentation: **horizontal** and **vertical**. Horizontal fragments are subsets of tuples, whereas vertical fragments are subsets of attributes. There are also two other types of fragmentation: **mixed** and **derived,** a type of horizontal fragmentation. All these different types of fragmentation have been described briefly in the following.

Horizontal fragmentation

Horizontal fragmentation partitions a relation along its tuples, that is, horizontal fragments are subsets of the tuples of a relation. A horizontal fragment is produced by specifying a predicate that performs a restriction on the tuples of a relation. In this fragmentation, the predicate is defined by using the selection operation of the relational algebra. For a given relation R, a horizontal fragment is defined as

$\sigma_{\rho}(R)$

where ρ is a predicate based on one or more attributes of the relation *R*.

In some cases, the choice of horizontal fragmentation strategy, that is, the predicates or search conditions for horizontal fragmentation is obvious. However, in some cases, it is very difficult to choose the predicates for horizontal fragmentation and it requires a detailed analysis of the application. The predicates may be **simple** involving single attribute, or may be **complex** involving multiple attributes. Further, the predicates for each attribute may be single or multi-valued, and the values may be discrete or may involve a range of values. Rather than complicated predicates, the fragmentation strategy involves finding **midterm** predicates or **minimal** predicates that can be used as the basis for fragmentation schema. A minimal predicate is the conjunction of simple predicates, which is **complete** and **relevant**. A set of predicates is complete if and only if any two tuples in the same fragment are referenced with the same probability by any transaction. A predicate is relevant if there is at least one transaction that accesses the resulting fragments differently.

Horizontal fragments can be generated by using the following methods:

- Consider a predicate *P*1 which partitions the tuples of a relation *R* into two parts which are referenced differently by at least one application. Assume that $P = P1$.
- Consider a new simple predicate *Pi* that partitions at least one fragment of *P* into two parts which are referenced in a different way by at least one application; Set $P \leftarrow P \cup Pi$. Non-relevant predicate should be eliminated from *P* and this procedure is to be repeated until the set of the midterm fragments of *P* is complete.

Example 5.1

Let us consider the relational schema Project [**Chapter 3, Section 3.1.3**] where project-type represents whether the project is an inside project (within the country) or abroad project (outside the country). Assume that P1 and P2 are two horizontal fragments of the relation Project, which are obtained by using the predicate "whether the value of project-type attribute is 'inside' or 'abroad' ", as listed in the following:

P1: $\sigma_{\text{project-type} = \text{“inside”}}(\text{Project})$

P2: $\sigma_{\text{project-type} = \text{“abroad”}}(\text{Project})$

The descriptions of the Project relation and the horizontal fragments of this relation are illustrated in figure 5.1.

These horizontal fragments satisfy all the correctness rules of fragmentation as shown below:

- **Completeness** – Each tuple in the relation Project appears either in fragment P1 or P2. Thus, it satisfies completeness rule for fragmentation.
- **Reconstruction** – The Project relation can be reconstructed from the horizontal fragments P1 and P2 by using the union operation of relation algebra, which ensures the reconstruction rule.

  Thus, $P1 \cup P2 = \text{Project}$.
- **Disjointness** – The fragments P1 and P2 are disjoint, since there can be no such project whose project-type is both "inside" and "abroad".

In this example, the predicate set {project-type = "inside", project-type = "abroad"} is complete.

Project

Project-id	Project-name	Project-type	Project-leader-id	Branch-no	Amount
P01	Inventory	Inside	E001	B10	$1000000
P02	Sales	Inside	E001	B20	$300000
P03	R&D	Abroad	E004	B70	$8000000
P04	Educational	Inside	E003	B20	$400000
P05	Health	Abroad	E005	B60	$7000000

P1

Project-id	Project-name	Project-type	Project-leader-id	Branch-no	Amount
P01	Inventory	Inside	E001	B10	$1000000
P02	Sales	Inside	E001	B20	$300000
P04	Educational	Inside	E003	B20	$400000

P2

Project-id	Project-name	Project-type	Project-leader-id	Branch-no	Amount
P03	R&D	Abroad	E004	B70	$8000000
P05	Health	Abroad	E005	B60	$7000000

Figure 5.1 Horizontal fragmentation of the relation Project

Example 5.2

Let us consider the distributed database of a manufacturing company that has three sites in eastern, northern, and southern regions. The company has a total of 20 products out of which the first 10 products are produced in the eastern region, the next five products are produced in the northern region, and the remaining five products are produced in the southern region. The global schema of this distributed database includes several relational schemas such as **Branch, Product, Supplier,** and **Sales.** In this example, the horizontal fragmentation of Sales and Product has been considered.

Assume that there are two values for region attribute, "eastern" and "northern", in the relational schema **Sales (depo-no, depo-name, region).** Let us consider an application that can generate from any site of the distributed system and involves the following SQL query.

Select depo-name from Sales where depo-no = $1

If the query is initiated at site 1, it references Sales whose region is "eastern" with 80 percent probability. Similarly, if the query is initiated at site 2, it references Sales whose region is "northern" with 80 percent probability, whereas if the query is generated at site 3, it references Sales of "eastern" and "northern" with equal probability. It is assumed that the products produced in a region come to the nearest sales depot for sales. Now, the set of predicates is {$p1$, $p2$}, where

p1: region = "eastern" and p2: region = "northern".

Since the set of predicates {$p1$, $p2$} is complete and minimal, the process is terminated. The relevant predicates cannot be deduced by analysing the code of an application. In this case, the midterm predicates are as follows:

X1: (region = "eastern") AND (region = "northern")

X2: (region = "eastern") AND NOT (region = "northern")

X3: NOT (region = "eastern") AND (region = "northern")

X4: NOT (region = "eastern") AND NOT (region = "northern")

Since (region = "eastern") ⇒ NOT (region = "northern") and (region = "northern") ⇒ NOT (region = "eastern"), *X*1 and *X*4 are contradictory and *X*2 and *X*3 reduce to the predicates *p*1 and *p*2.

For the global relation **Product (product-id, product-name, price, product-type)**, the set of predicates are as follows:

P1: product-id ≤ 10

P2: 10 < product-id ≤ 15

P3: product-id > 15

P4: product-type = "consumable"

P5: product-type = "Non-consumable"

It is assumed that applications are generated at site 1 and site 2 only. It is further assumed that the applications that involve queries about consumable products are issued at site 1, while the applications that involve queries about non-consumable products are issued at site 2. In this case, the fragments after reduction with the minimal set of predicates are listed in the following:

F1: product-id ≤ 10

F2: (10 < product-id ≤ 15) AND (product-type = "consumable")

F3: (10 < product-id ≤ 15) AND (product-type = "Non-consumable")

F4: product-id > 15.

The allocation of fragments is introduced in Section 5.4.

Vertical fragmentation

Vertical fragmentation partitions a relation along its attributes, that is, vertical fragments are subsets of attributes of a relation. A vertical fragment is defined by using the projection operation of relational algebra. For a given relation *R*, a vertical fragment is defined as

$$\Pi_{a_1, a_2, \ldots, a_n}(R)$$

where $a_1, a_2, \ldots, a_n$, are attributes of the relation *R*.

The choice of vertical fragmentation strategy is more complex than that of horizontal fragmentation, since a number of alternatives are available. One solution is to consider the affinity of one attribute to another. Two different types of approaches have been identified for attribute partitioning in vertical fragmentation of global relations as listed in the following:

(a) Grouping – Grouping is started by assigning each attribute to one fragment, and at each step, joining of some of the fragments are done until some criteria is satisfied. Grouping was first suggested in Hammer and Niamir [1979] for centralized databases and was used for distributed databases later on [Sacca and Wiederhold, 1985].

(b) Splitting – Splitting starts with a relation and decides on beneficial partitioning, based on the access behaviour of applications to the attributes. One way to do this is to create a matrix that shows the number of accesses that refer to each attribute pair. For example, a transaction that accesses attributes a_1, a_2, a_3, and a_4 of relation R can be represented by the following matrix:

	a_1	a_2	a_3	a_4
a_1		1	0	1
a_2			0	1
a_3				0
a_4				

In this process, a matrix is produced for each transaction and an overall matrix is produced showing the sum of all accesses for each attribute pair. Pairs with high affinity should appear in the same vertical fragment and pairs with low affinity may appear in different fragments. This technique was first proposed for centralized database design [Hoffer and Severance, 1975] and then it was extended for distributed environment [Navathe et al., 1984].

Example 5.3

In this case, the Project relation is partitioned into two vertical fragments V1 and V2, which are described below [**figure 5.2**]:

V1

Project-id	Project-leader-id	Branch-no
P01	E001	B10
P02	E001	B20
P03	E004	B70
P04	E003	B20
P05	E005	B60

V2

Project-id	Project-name	Project-type	Amount
P01	Inventory	Inside	$1000000
P02	Sales	Inside	$300000
P03	R&D	Abroad	$8000000
P04	Educational	Inside	$400000
P05	Health	Abroad	$7000000

Figure 5.2 Vertical fragmentation of the relation Project

V1: $\Pi_{\text{Project-id, branch-no, project-leader-id}}$(Project)

V2: $\Pi_{\text{Project-id, Project-name, Project-type, amount}}$(Project).

Hence, primary key for the relation Project is **Project-id,** which is repeated in both vertical fragments V1 and V2 to reconstruct the original base relation from the fragments.

Hence, vertical fragmentation also ensures the correctness rules for fragmentation.

- **Completeness** – Each attribute in the relation Project appears either in fragment V1 or V2, which satisfies the completeness rule for fragmentation.
- **Reconstruction** – The Project relation can be reconstructed from the vertical fragments V1 and V2 by using the Natural join operation of relational algebra, which ensures the reconstruction rule.

  Thus, V1 $\times$ V2 = Project.
- **Disjointness** – The vertical fragments V1 and V2 are disjoint, except for the primary key project-id, which is repeated in both fragments and is necessary for reconstruction. Hence, the primary key Project-id of the Project relation appears in both vertical fragments V1 and V2.

Bond energy algorithm The Bond Energy Algorithm (BEA) is the most suitable algorithm for vertical fragmentation [Navathe et al., 1984]. The bond energy algorithm uses attribute affinity matrix (AA) as input and produces a clustered affinity matrix (CA) as output by permuting rows and columns of AA. The generation of CA from AA involves three different steps: initialization, iteration, and row ordering, which are illustrated in the following:

- **Initialization** – In this step, one column from AA is selected and is placed into the first column of CA.
- **Iteration** – In this step, the remaining $n - i$ columns are taken from AA and they are placed in one of the possible $i + 1$ positions in CA that makes the largest contribution to the global neighbour affinity measure. It is assumed that i number of columns are already placed into CA.
- **Row ordering** – In this step, rows are ordered in the same way as columns are ordered. The contribution of a column A_k, which is placed between A_i and A_j, can be represented as follows:

  $$\text{cont}(A_i, A_k, A_j) = \text{bond}(A_i, A_k) + \text{bond}(A_k, A_j) - \text{bond}(A_i, A_j)$$

  where

  $$\text{bond}(A_x, A_y) = \sum_{z=1, 2, \ldots, n} \text{aff}(A_x, A_z)\ \text{aff}(A_z, A_y).$$

Now, for a given set of attributes many orderings are possible. For example, for n number of attributes n orderings are possible. One efficient algorithm for ordering is searching for clusters. The BEA proceeds by linearly traversing the set of attributes. In each step, one of the remaining attributes is added and is inserted in the current order of attributes in such a way that the maximal contribution is achieved. This is first done for the columns. Once all the columns are determined, the row ordering is adapted to the column ordering, and the resulting affinity matrix exhibits the desired clustering. To compute the contribution to the global affinity value, the loss incurred through separation of previously joint columns is subtracted from the gain, obtained by adding a new column.

The contribution of a pair of columns is the scalar product of the columns, which is maximal if the columns exhibit the same value distribution.

Example 5.4

Consider $Q = \{Q_1, Q_2, Q_3, Q_4\}$ as a set of queries, $A = \{A_1, A_2, A_3, A_4\}$ as a set of attributes for the relation R, and $S = \{S_1, S_2, S_3\}$ as a set of sites in the distributed system. Assume that A_1 is the primary key of the relation R, and the following matrices represent the attribute usage values of the relation R and application access frequencies at different sites:

	A_1	A_2	A_3	A_4
Q_1	0	1	1	0
Q_2	1	1	1	0
Q_3	1	0	0	1
Q_4	0	0	1	0

	S_1	S_2	S_3	Sum
Q_1	10	20	0	30
Q_2	5	0	10	15
Q_3	0	35	5	40
Q_4	0	10	0	10

Since A_1 is the primary key of the relation R, the following attribute affinity matrix is considered here:

	A_2	A_3	A_4
A_2	45	45	0
A_3	45	55	0
A_4	0	0	40

Now,

$\text{bond}(A_2, A_3) = 45*45 + 45*55 + 0 * 0 = 4{,}500.$

$\text{bond}(A_2, A_4) = 45 * 0 + 45 * 0 + 0 * 40 = 0.$

$\text{bond}(A_3, A_4) = 45 * 0 + 55 * 0 + 0 * 40 = 0.$

The contributions of the columns depending on their position are as follows:

For A_4-A_2-A_3, $\text{cont}(\_, A_4, A_2) = \text{bond}(\_, A_4) + \text{bond}(A_4, A_2) - \text{bond}(\_, A_2) = 0.$

For A_2-A_4-A_3, $\text{cont}(A_2, A_4, A_3) = \text{bond}(A_2, A_4) + \text{bond}(A_4, A_3) - \text{bond}(A_2, A_3) = 0 + 0 - 4{,}500 = -4{,}500.$

For A_2-A_3-$A4$, $\text{cont}(A_3, A_4, \_) = \text{bond}(A_3, A_4) + \text{bond}(A_4, \_) - \text{bond}(A_3, \_) = 0.$

In this case, both A_4-A_2-A_3 and A_2-A_3-A_4 are same.

Now,

	A_2	A_3	A_4
A_2	45	45	0
A_3	45	55	0
A_4	0	0	40

and

	A_1	A_2	A_3	A_4
Q_1	0	1	1	0
Q_2	1	1	1	0
Q_3	1	0	0	1
Q_4	0	0	1	0

	S_1	S_2	S_3	Sum
Q_1	10	20	0	30
Q_2	5	0	10	15
Q_3	0	35	5	40
Q_4	0	10	0	10

Hence,

accesses (fragment 1: {A_2}): 0

accesses (fragment 2: {A_3, A_4}): 50

accesses (fragment 1 AND fragment 2): 45

sq = –1,975

accesses (fragment 1: {A_2, A_3}): 55

accesses (fragment 2: {A_4}): 40

accesses (fragment 1 AND fragment 2): 0

sq = 2,200

accesses (fragment 1: {A_2, A_4}): 40

accesses (fragment 2: {A_3}): 10

accesses (fragment 1 AND fragment 2): 45

sq = –1,625

Therefore, two partitions are {A_1, A_4} and {A_1, A_2, A_3}. In the case of vertical fragmentation, the primary key will be repeated in each partition. The same calculation can be done with all attributes.

Mixed fragmentation

Mixed fragmentation is a combination of horizontal and vertical fragmentation. This is also referred to as hybrid or nested fragmentation. A mixed fragment consists of a horizontal fragment that is subsequently vertically fragmented, or a vertical fragment that is then horizontally fragmented. A mixed fragment is defined by using selection and projection operations of relational algebra. For example, a mixed fragment for a given relation R can be defined as follows:

$\sigma_p (\Pi\, a_1, a_2, \ldots, a_n (R))$ or $\Pi\, a_1, a_2, \ldots, a_n (\sigma_p (R))$

where ρ is a predicate based on one or more attributes of the relation R and $a_1, a_2, \ldots, a_n$ are attributes of the relation R.

Example 5.5

Let us consider the same Project relation used in the previous example. The mixed fragments of the above Project relation can be defined as follows:

P11: $\sigma_{\text{project-type} = \text{“inside”}}(\Pi_{\text{Project-id, branch-no, project-leader-id}}(\text{Project}))$

P12: $\sigma_{\text{project-type} = \text{“abroad”}}(\Pi_{\text{Project-id, branch-no, project-leader-id}}(\text{Project}))$

P21: $\sigma_{\text{project-type} = \text{“inside”}}(\Pi_{\text{Project-id, Project-name, Project-type, amount}}(\text{Project}))$

P22: $\sigma_{\text{project-type} = \text{“abroad”}}(\Pi_{\text{Project-id, Project-name, Project-type, amount}}(\text{Project}))$, where

P1: $\Pi_{\text{Project-id, branch-no, project-leader-id}}(\text{Project})$ and

P2: $\Pi_{\text{Project-id, Project-name, Project-type, amount}}(\text{Project})$.

Hence, first the Project relation is partitioned into two vertical fragments P1 and P2 and then each of the vertical fragments is subsequently divided into two horizontal fragments, which are shown in figure 5.3.

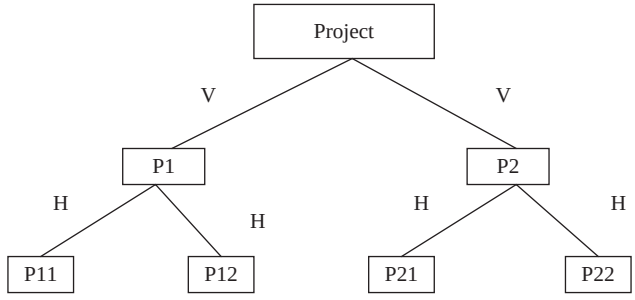


Fig. 5.3 Mixed Fragmentation of the Relation Project

By virtue of fragmentation, mixed fragmentation satisfies all correctness rules for fragmentation as explained below.

» **Completeness** – Each attribute in the Project relation appears in either of the fragments P1 or P2 and each (part) tuple that appears in the fragment P1 appears in either of the fragments P11 or P12. Similarly, each (part) tuple that appears in the fragment P2 appears in either of the fragments P21 or P22.

- **Reconstruction** – The Project relation can be reconstructed from the fragments P11, P12, P21, and P22 by using union and natural join operations of relational algebra. Thus, Project = P1 ⋈ P2, where P1 = P11 ∪ P12 and P2 = P21 ∪ P22, respectively.
- **Disjointness** – The fragments P11, P12, P21, and P22 are disjoint, since there can be no such project whose project type is both "inside" and "abroad". The fragments P1 and P2 are also disjoint, except for the necessary duplication of the primary key project-id.

Derived fragmentation

A **derived fragmentation** is a horizontal fragment that is based on the horizontal fragmentation of a parent relation and it does not depend on the properties of its own attributes. Derived fragmentation is used to facilitate the join between fragments. The term **child** is used to refer to the relation that contains the foreign key, and the term **parent** is used for the relation containing the targeted primary key. Derived fragmentation is defined by using the semi-join operation of relation algebra. For a given child relation *C* and parent relation *P*, the derived fragmentation of *C* can be represented as follows:

$$C_i = C \triangleright P_i, 1 \leq i \leq w$$

where *w* is the number of horizontal fragments defined on *C* and

$$P_i = \sigma_{Fi}(S)$$

where *Fi* is the predicate according to which the primary horizontal fragment *Si* is defined.

If a relation contains more than one foreign key, it will be necessary to select one of the referenced relations as the parent relation. The choice can be based on one of the following two strategies:

(i) The fragmentation that is used in most of the applications.

(ii) The fragmentation with better join characteristics, that is, the join involves smaller fragments or the join that can be performed in parallel to a greater degree.

Example 5.6

Let us consider the following Department and Employee relation together [**Chapter 1, Example 1.1**].

Employee (<u>emp-id</u>, ename, designation, salary, deptno, voter-id)

Department (<u>deptno</u>, dname, location)

Assume that the Department relation is horizontally fragmented according to the deptno so that data relating to a particular department is stored locally. For instance,

P1: $\sigma_{\text{deptno} = 10}$ (Department)

P2: $\sigma_{\text{deptno} = 20}$ (Department)

P3: $\sigma_{\text{deptno} = 30}$ (Department)

Hence, in the Employee relation, each employee belongs to a particular department (deptno) that references to the deptno field in the Department relation. Thus, it should be beneficial to store

Employee relation using the same fragmentation strategy of the Department relation. This can be achieved by derived fragmentation strategy that partitions the Employee relation horizontally according to the deptno as follows:

$Ci = \text{Employee} \triangleright_{\text{deptno}} Pi, 1 \leq i \leq 3$

It can be shown that derived fragmentation also satisfies the correctness rules of fragmentation.

- **Completeness** – Since the predicate of the derived fragmentation involves two relations, it is more difficult to ensure the completeness rule for derived fragmentation. In the above example, each tuple of Department relation must appear either in fragments P1, P2, or P3. Similarly, each tuple of Employee relation must appear in either of the fragments *Ci*. This rule is known as *referential integrity constraint*, which ensures that the tuples of any fragment of the child relation (Employee) are also in the parent relation (Department).
- **Reconstruction** – A global relation can be reconstructed both from its horizontal and derived fragments by using join and union operations of relational algebra. Thus, for a given relation *R* with fragments $R_1, R_2, \ldots, R_n$,

  $R = \cup R_i, 1 \leq i \leq n$
- **Disjointness** – Since derived fragmentation involves a semijoin operation, it adds extra complexity to ensure the disjointness of derived fragments. However, it is not desirable that a tuple of a child relation be joined with two or more tuples of the parent relation when these tuples are in different fragments of the parent relation. For example, an employee of the Employee relation with branch-no 10 should not belong to deptno = 20 or deptno = 30.

No fragmentation

A final strategy of the fragmentation is not to fragment a relation. If a relation contains a smaller number of tuples and not updated frequently, then is better not to fragment the relation. It will be more sensible to leave the relation as a whole and simply replicate the relation at each site of the distributed system.

5.4 The Allocation of Fragments

The allocation of fragments is a critical performance issue in the context of distributed database design. Before allocation of fragments into different sites of a distributed system, it is necessary to identify whether the fragments are replicated or not. The allocation of non-replicated fragments can be handled easily by using "best-fit" approach. In best-fit approach, the best cost-effective allocation strategy is selected among several alternatives of possible allocation strategies. Replication of fragments adds extra complexity to the fragment allocation issue as follows:

- The total number of replicas of fragments may vary.
- It is difficult to design read-only applications, because applications can access several alternative replicas of a fragment from different sites of the distributed system.

For allocating replicated fragments, one of the following methods can be used:

- In the first approach, the set of all sites in the distributed system is determined where the benefit of allocating one replica of the fragment is higher than the cost of allocation. One replica of the fragment is allocated to such beneficial sites.
- In the alternative approach, allocation of fragments is done using best-fit method considering fragments are not replicated, and then progressively replicas are introduced starting from the most beneficial. This process is terminated when addition of replicas is no more beneficial.

Both the above approaches have some limitations. In the first approach, determination of the cost and the benefit of each replica of the fragment is very complicated, whereas in the latter approach, progressive increment of additional replicas is less beneficial. Moreover, the reliability and availability increases if there are two or three copies of the fragment, but further copies give a less than proportional increase.

5.4.1 Measure of Costs and Benefits for Fragment Allocation

To evaluate the cost and benefit of the allocation of fragments, it is necessary to consider the type of fragments. Moreover, the following definitions are assumed:

- The fragment index is represented by i
- The site index is represented by j
- The application index is represented by k
- F_{kj} indicates the frequency of application k at site j
- R_{ki} represents the number of retrieval references (read operations) of application k to fragment i
- U_{ki} indicates the number of update references (write operations) of application k to fragment i
- $N_{ki} = R_{ki} + U_{ki}$

Horizontal fragments

(i) In this case, using best-fit approach for non-replicated fragments, the fragment Ri of relation R is allocated at site j where the number of references to the fragment Ri is maximum. The number of local references of Ri at site j is as follows:

$$B_i j = \sum_k F_{kj} N_{ki}$$

where R_i is allocated to site j^* such that B_{ij^*} is maximum.

(ii) In this case, using the first approach for replicated fragments, the fragment R_i of relation R is allocated at site j, where the cost of retrieval references of applications is larger than the cost of update references to R_i from applications at any other site. Hence, B_{ij} can be evaluated as follows:

$$B_{ij} = \sum_{\mathrm{k}} F_{kj} R_{ki} - C \sum_k \sum_{k\text{-}j} F_{kj} U_{ki}$$

where C is a constant that measures the ratio between the cost of an update and a retrieval access. Typically, update accesses are more expensive and $C \geq 1$. R_i is allocated at all sites j^* where B_{ij^*} is positive. When B_{ij^*} is negative, a single copy of R_i is placed at the site where $B_{ij}\ B_{ij^*}$ is maximum.

(iii) In this case, the benefit of additional replica of a fragment is evaluated in terms of availability and reliability. Let d_i denote the degree of replicas, or R_i and F_i denote the benefit of having R_i fully replicated at each site. The benefit function can be defined as

$$\beta(d_i) = (1 - 2^{1-d_i})F_i$$

Now, the benefit of introducing a new copy of R_i at site j is as follows:

$$B_{ij} = \sum_{k} F_{kj} R_{ki} - C \sum_{k} \sum_{k-j} F_{kj} U_{ki} + \beta(d_i)$$

Vertical fragments

In this case, the benefit is calculated by vertically partitioning a fragment R_i into two vertical fragments R_s and R_t allocated at site s and site t, respectively. The effect of this partition is listed below.

- It is assumed that there are two sets of applications A_s and A_t issued at site s and site j, respectively, which use the attributes of R_s and R_t and become local to sites s and t, respectively.
- There is a set A_1 of applications previously local to r, which uses only attributes of R_s and R_t. An additional remote reference is now required for these applications.
- There is a set A_2 of applications previously local to r, which reference attributes of both R_s and R_t. These applications make two additional remote references.
- There is a set A_3 of applications at sites different than r, s, or t, which reference attributes of both R_s and R_t. These applications make one additional remote reference.

Now the benefit of this partition is as follows:

$$B_{ist} = \sum_{k \varepsilon A_s} F_{ks} N_{ki} + \sum_{k \varepsilon A_t} F_{kt} N_{ki} - \sum_{k \varepsilon A_1} F_{kr} N_{ki} - \sum_{k \varepsilon A_2} 2F_{kr} N_{ki} - \sum_{k \varepsilon A_3} \sum_{j \neq r,s,t} F_{kj} N_{ki}$$

For simplicity, it is sufficient to use $R_{ki} + CU_{ki}$ instead of N_{ki}. This formula can be used within an exhaustive splitting algorithm to determine whether the splitting of R_i at site i into R_s and R_t at site t is convenient or not by trying all possible combinations of sites s and t. Some care must be taken in the case of $r = s$ or $r = t$.

5.5 Transparencies in Distributed Database Design

According to the definition of distributed database, one major objective is to achieve the transparency into the distributed system. **Transparency** refers to the separation of the higher-level semantics of a system from lower-level implementation issues. In a distributed system, transparency hides the implementation details from users of the system. In other words, the user believes that he or she is working with a centralized database system, and that all the complexities of a distributed database are either hidden or transparent to the user. A distributed DBMS may have various levels

of transparency. In a distributed DBMS, the following four main categories of transparency have been identified:

- Distribution transparency
- Transaction transparency
- Performance transparency
- DBMS transparency.

5.5.1 Data Distribution Transparency

Data distribution transparency allows the user to perceive the database as a single, logical entity. In other words, distribution transparency refers to the degree or extent to which details of fragmentation (**fragmentation transparency**), replication (**replication transparency**) and distribution (**location transparency**) are hidden from users. If a user sees all fragmentation, allocation, and replication, the distributed DBMS is said to have no distribution transparency. The user needs to refer to specific fragment copies by appending the site name to the relation. If the user needs to know that the data are fragmented and the location of fragments for retrieving data, then it is called **local mapping transparency**.

If a system supports higher degree of distribution transparency, the user sees a single integrated schema with no details of fragmentation, allocation, or distribution. The distributed DBMS stores all the details in the distribution catalog. All these distribution transparencies are discussed in the following:

(a) Fragmentation transparency – **Fragmentation transparency** hides the fact from users that the data are fragmented. This is the highest level of distribution transparency. If a distributed system has fragmentation transparency, then the user must not be aware regarding the fragmentation of data. As a result, database accesses must be based on global schema and the user need not to specify the particular fragment names or data locations.

Example 5.7

Let us consider the relation **Employee (emp-id, emp-name, designation, salary, emp-branch, project-no)** is fragmented and stored in different sites of a distributed DBMS. Further assume that the distributed DBMS has fragmentation transparency and, thus, users have no idea regarding the fragmentation of the Employee relation. Therefore, to retrieve the names of all employees of branch number 10 from Employee relation, the user will write the following SQL statement:

Select emp-name from Employee where emp-branch = 10.

This SQL statement is same as centralized DBMS.

(b) Location transparency – With **location transparency**, the user is aware how the data are fragmented but still does not have any idea regarding the location of the fragments. Location transparency is the middle level of distribution transparency. To retrieve data from a distributed database with location transparency, the end user or programmer has to specify the database fragment names but need not specify where these fragments are located in the distributed system.

Example 5.8

Let us assume that the tuples of the above **Employee** relation is horizontally partitioned into two fragments **EMP1** and **EMP2** depending on the selection predicates "emp-id ≤ 100" and "emp-id > 100". Hence, the user is aware that the Employee relation is horizontally fragmented into two relations EMP1 and EMP2, but they have no idea in which sites these relations are stored. Thus, the user will write the following SQL statement for the above query "retrieve the names of all employees of branch number 10":

Select emp-name from EMP1 where emp-branch = 10

union

Select emp-name from EMP2 where emp-branch = 10.

(c) Replication transparency – **Replication transparency** means that the user is unaware of the fact that the fragments of relations are replicated and stored in different sites of the distributed DBMS. Replication transparency is closely related to location transparency and it is implied by location transparency. However, it may be possible for a distributed system not to have location transparency but to have replication transparency. In such cases, the user has to mention the location of the fragments of a relation for data access. The replication transparency is illustrated in the example 5.8.

Example 5.9

Let us assume that the horizontal fragments EMP1 and EMP2 of the Employee relation in example 5.7 are replicated and stored in different sites of the distributed system. Further, assume that the distributed DBMS supports replication transparency. In this case, the user will write the following SQL statement for the query "retrieve the names of all employees of branch number 20 whose salary is greater than Rs. 50,000":

Select emp-name from EMP1 where emp-branch = 20 and salary > 50,000

union

Select emp-name from EMP2 where emp-branch = 20 and salary > 50,000.

If the distributed system does not support replication transparency, then the user will write the following SQL statement for the above query considering there are a number of replicas of fragments EMP1 and EMP2 of Employee relation:

Select emp-name from copy1 of EMP1 where emp-branch = 20 and salary > 50,000

union

Select emp-name from copy3 of EMP2 where emp-branch = 20 and salary > 50,000.

Similarly, the above query can be rewritten as follows for a distributed DBMS with replication transparency which does not exhibit location transparency:

Select emp-name from EMP1 at site 1 where emp-branch = 20 and salary > 50,000

union

Select emp-name from EMP2 at site 3 where emp-branch = 20 and salary > 50,000.

(d) Local mapping transparency – Local mapping transparency is the lowest level of distribution transparency. Local transparency refers that users are aware regarding both the fragment names and the location of fragments, taking into account that any replication of the fragments may exist. If a distributed system has local mapping transparency, then the user has to explicitly mention both the fragment name and the location for data access.

Example 5.10

Let us consider the relation **Project (project-id, project-name, project-type, project-leader-id, branch-no, amount)** is horizontally partitioned into two fragments **P1** and **P2** depending on the project-type "inside" and "abroad". Assume that the fragmented relations P1 and P2 are replicated and stored in different sites of the distributed DBMS. With local mapping transparency, the user will write the following SQL statement for the query "retrieve project names and branch numbers of all the projects where the project amount is greater than Rs. 10,000,000":

Select project-name, branch-no from copy1 of P1 at site 1 where amount>10000000

union

Select project-name, branch-no from copy3 of P2 at site 4 where amount>10000000

Hence, it is assumed that replicas of fragments P1 and P2 of the Project relation are allocated to different sites of the distributed system such as site1, site3, and site4.

(e) Naming transparency – Naming transparency means that the users are not aware of the actual name of the database objects in the system. If a system supports naming transparency, the user will specify the alias names of database objects for data accessing. In a distributed database system, each database object must have a unique name. The distributed DBMS must ensure that no two sites create a database object with the same name. To ensure this, one solution is to create a **central name server,** which ensures the uniqueness of names of database objects in the system. However, this approach has several disadvantages as follows:

- Loss of some local autonomy, because during creation of new database objects, each site has to ensure uniqueness of names of database objects from the central name server.
- Performance may be degraded, if the central site becomes a bottleneck.
- Low availability and reliability, because if the central site fails, the remaining sites cannot create any new database object. As availability decreases, reliability also decreases.

A second alternative solution is to prefix a database object with the identifier of the site that created it. This naming method will also be able to identify each fragment along with each of its copies. Therefore, copy2 of fragment 1 of Employee relation created at site 3 can be referred as S3.Empolyee. F1.C2. The only problem with this approach is that it results in loss of distribution transparency.

One solution that can overcome the disadvantages of the above two approaches is the use of **aliases** (sometimes called **synonyms**) for each database object. It is the responsibility of the distributed database system to map an alias to the appropriate database object.

The distributed system R* differentiates between an object's name and its system-wide name (global name). The printname is the name through which the users refer to the database object. The system-wide name or global name is a globally unique internal identifier for the database object that is never changed. The system-wide name contains four components as follows:

- **Creator ID** – This represents a unique site identifier for the user who created the database object.

- **Creator site ID** – It indicates a globally unique identifier for the site from which the database object was created.
- **Local name** – It represents an unqualified name for the database object.
- **Birth-site ID** – This represents a globally unique identifier for the site at which the object was initially stored.

For example, the system-wide name, Project-leader@India.localBranch@kolkata, represents an object with local name localBranch, created by Project Leader in India site and initially stored at the Kolkata site.

Distribution transparency is supported by a **distributed data dictionary** or a **distributed data catalog**. The distributed data catalog contains the description of the entire database as seen by the database administrator. The database description is known as **distributed global schema**.

Example 5.11

In this example, data distribution transparencies for update application have been considered. Assume that the relational schema **Employee (emp-id, emp-name, designation, salary, emp-branch, project-no)** is fragmented as follows:

Emp1: $\sigma_{\text{emp-branch} \leq 10}$ **(**$\Pi_{\text{emp-id, emp-name, emp-branch, project-no}}$**(Employee))**

Emp2: $\sigma_{\text{emp-branch} \leq 10}$**(**$\Pi_{\text{emp-id, salary,design}}$**(Employee))**

Emp3: $\sigma_{\text{emp-branch} > 10}$ **(**$\Pi_{\text{emp-id, emp-name, emp-branch, project-no}}$**(Employee))**

Emp4: $\sigma_{\text{emp-branch} > 10}$**(**$\Pi_{\text{emp-id, salary,design}}$**(Employee))**

Consider an update request is generated in the distributed system that the branch of an Employee with emp-id 55 will be modified from emp-brach 10 to emp-branch 20. The user written queries are illustrated in the following for different levels of transparency such as fragmentation transparency, location transparency, and local mapping transparency.

Fragmentation transparency:

```
Update Employee
set emp-branch = 20
where emp-id = 55.
```

Location transparency:

```
Select emp-name, project-no into $emp-name, $project-no from Emp1
where emp-id = 55
Select salary, design into $salary, $design from Emp2
where emp-id = 55
Insert into Emp3 (emp-id, emp-name, emp-branch, project-no)
values (55, $emp-name, 20, $project-no)
Insert into Emp4 (emp-id, salary, design)
```

values (55, $salary,$design)

Delete from Emp1 where emp-id = 55

Delete from Emp2 where emp-id = 55

Local Mapping transparency:

Select emp-name, project-no into $emp-name, $project-no from Emp1 at site 1

where emp-id = 55

Select salary, design into $salary, $design from Emp2 at site 2

where emp-id = 55

Insert into Emp3 at site 3 (emp-id, emp-name, emp-branch, project-no)

values (55, $emp-name, 20, $project-no)

Insert into Emp3 at site 7 (emp-id, emp-name, emp-branch, project-no)

values (55, $emp-name, 20, $project-no)

Insert into Emp4 at site 4(emp-id, salary, design)

values (55, $salary,$design)

Insert into Emp4 at site 8(emp-id, salary, design)

values (55, $salary,$design)

Delete from Emp1 at site 1 where emp-id = 55

Delete from Emp1 at site 5 where emp-id = 55

Delete from Emp2 at site 2 where emp-id = 55

Delete from Emp2 at site 6 where emp-id = 55

Hence, it is assumed that the fragment Emp1 has two replicas stored at site 1 and site 5, respectively, the fragment Emp2 has two replicas stored at site 2 and site 6, respectively, the fragment Emp3 has two replicas stored at site 3 and site 7, respectively, and the fragment Emp4 has two replicas stored at site 4 and site 8, respectively.

5.5.2 Transaction Transparency

Transaction transparency in a distributed DBMS ensures that all distributed transactions maintain the distributed database integrity and consistency. A distributed transaction can update data stored at many different sites connected by a computer network. Each transaction is divided into several **subtransactions** (represented by an **agent**), one for each site that has to be accessed. Transaction transparency ensures that the distributed transaction will be successfully completed only if all subtransactions executing in different sites associated with the transaction are completed successfully. Thus, a distributed DBMS requires complex mechanism to manage the execution of distributed transactions and to ensure the database consistency and integrity. Moreover, transaction transparency becomes more complex due to fragmentation, allocation, and replication schemas in

distributed DBMS. Two further aspects of transaction transparency are **concurrency transparency** and **failure transparency**, which are discussed in the following:

(a) Concurrency transparency – Concurrency transparency in a distributed DBMS ensures that all concurrent transactions (distributed and non-distributed) execute independently in the system and are logically consistent with the results, that are obtained if transactions are executed one at a time in some arbitrary serial order. The distributed DBMS requires complex mechanism to ensure that both local and global transactions do not interfere with each other. Moreover, the distributed DBMS must ensure the consistency of all subtransactions involved with a global transaction.

Replication adds extra complexity to the issue of concurrency in a distributed DBMS. For example, if a copy of a replicated data item is updated, then the update must be propagated eventually to all copies. One obvious solution to update all copies of a data item is to propagate the changes as part of the original transaction, making it an atomic operation. However, the update process is delayed if one of the sites holding a copy is not reachable during update due to site or communication link failure. If there are many copies of the data item, the probability of succeeding the transaction decreases exponentially.

An alternative solution is to propagate the update to those sites only that are currently available. The remaining sites must be updated when they become available. Therefore, a further strategy is to update the copies of a data item **asynchronously**, sometimes after the original update.

(b) Failure transparency – Failure transparency in a distributed DBMS promises that the system will continue its normal execution in the event of failure and it must maintain the atomicity of the global transaction. The atomicity of global transaction ensures that subtransactions of the global transaction are either all committed or all aborted. Thus, the distributed DBMS must synchronize the global transaction to ensure that all subtransactions have completed successfully before recording a final COMMIT for the global transaction. Failure transparency also ensures the durability of both local and global transactions. In addition to all different types of failures in centralized system, the following additional types of failures can occur in a distributed environment:

- The loss of a message
- The failure of a communication link
- The failure of a site
- Network partitioning.

Functions that are lost due to failures will be picked up by another network node and continued.

5.5.3 Performance Transparency

Performance transparency in a distributed DBMS ensures that it performs its tasks as centralized DBMS. In other words, performance transparency in a distributed environment assures that the system does not suffer from any performance degradation due to the distributed architecture and it will choose the most cost-effective strategy to execute a request. In a distributed environment, the **distributed Query processor** maps a data request into an ordered sequence of operations on local databases. In this context, the added complexity of fragmentation, allocation, and replication

schemas is to be considered. The distributed Query processor has to take decision regarding the following issues:

- To perform a data request as to which fragment to access.
- If the fragment is replicated, which copy of the fragment to use.
- Which data location should be used to perform a data request?

The distributed Query processor determines an execution strategy that would be optimized with respect to some cost function. Typically, the costs associated with a distributed data request include the following:

- The access time (I/O) cost involved in accessing the physical data on disk.
- The CPU time cost incurred when performing operations on data in main memory.
- The communication cost associated with the transmission of data across the network.

A number of query processing and query optimization techniques have been developed for distributed database system: some of them minimize the total cost of query execution time [Sacco and Yao, 1982], and some of them attempt to maximize the parallel execution of operations [Epstein et al., 1978] to minimize the response time of queries.

5.5.4 DBMS Transparency

DBMS transparency in a distributed environment hides the knowledge that the local DBMSs may be different and is, therefore, only applicable to heterogeneous distributed DBMSs. This is also known as **heterogeneity transparency**, which allows the integration of several different local DBMSs (relational, network, and hierarchical) under a common global schema. It is the responsibility of distributed DBMS to translate the data requests from the global schema to local DBMS schemas to provide DBMS transparency.

CHAPTER SUMMARY

- Distributed database design involves the following important issues: fragmentation, replication, and allocation.
  - **Fragmentation** – A global relation may be divided into a number of subrelations, called fragments, which are then distributed among sites. There are two main types of fragmentation: **horizontal** and **vertical**. Horizontal fragments are subsets of tuples and vertical fragments are subsets of attributes. Other two types of fragmentations are mixed and horizontal.
  - **Allocation** – Allocation involves the issue of allocating fragments among sites.
  - **Replication** – The distributed database system may maintain a copy of fragment at several different sites.
- Fragmentation must ensure the correctness rules – completeness, reconstruction, and disjointness.
- Alternative data allocation strategies are centralized, partitioned, selective replication, and complete replication.
- Transparency hides the implementation details of the distributed systems from the users. Different transparencies in distributed systems are distribution transparency, transaction transparency, performance transparency, and DBMS transparency.

EXERCISES

1. Multiple choice questions:

(i) Which of the following statements is true?
   a. Top-down approach is not suitable if the database design starts from the beginning.
   b. Bottom-up approach is suitable for the database design that is designed from the beginning.
   c. Bottom-up approach deals with the integration of existing database systems.
   d. None of these.

(ii) Which of the following is not a data allocation strategy?
   a. Partitioned
   b. Centralized
   c. Replication
   d. None of these.

(iii) Which of the following refers to the fact that the command used to perform a task is independent of the location of the data and the location of the system where the command is used?
   a. Naming transparency
   b. Fragmentation transparency
   c. Location transparency
   d. All of these.

(iv) Which of the following is not a benefit of data fragmentation?
   a. Parallelism
   b. Security
   c. Concurrency
   d. Integrity

(v) Which of the following is an objective of data distribution in the distributed system?
   a. Locality of reference
   b. Availability and reliability
   c. Communication cost
   d. All of these.

(vi) Preservation of functional dependency is ensured by which of the correctness rules of fragmentation?
   a. Disjointness
   b. Reconstruction
   c. Completeness
   d. All of these.

(vii) Which of the following statements is true?
   a. Horizontal fragments are subsets of tuples
   b. Vertical fragments are subsets of attributes
   c. Mixed fragments are subsets of a combination of tuples and attributes.
   d. All of these
   e. None of these.

(viii) Bond Energy algorithm is used for
   a. Horizontal fragmentation
   b. Vertical fragmentation
   c. Mixed fragmentation
   d. Derived fragmentation.

(ix) Which of the following statements is false?
   a. Mixed fragments are vertical fragments of horizontal fragments
   b. Mixed fragments are horizontal fragments of vertical fragments
   c. Derived fragments are horizontal fragments of horizontal fragments
   d. None of these.

(x) Which of the following statements is correct?
   a. A minimal predicate is the conjunction of simple predicates, which is complete and relevant.
   b. A set of predicates is complete if and only if any two tuples in the same fragment are referenced with the same probability by any transaction
   c. A predicate is relevant if there is at least one transaction that accesses the resulting fragments differently
   d. All of these.

(xi) Which of the following techniques is not used for vertical fragmentation?
   a. Grouping
   b. Splitting
   c. Clustering
   d. None of these.

(xii) Which transparency is provided if users are unaware about the names of database objects?
   a. Location transparency
   b. Naming transparency
   c. Local mapping transparency
   d. Replication transparency

(xiii) Clustering represents
   a. Storing common data in one place
   b. Storing different types of data in one place
   c. Storing different types of data in different places
   d. Storing common data in different places.

(xiv) Which of the following statements is correct?
   a. Local mapping transparency is level 1 transparency
   b. Replication transparency is level 3 transparency
   c. Location transparency is level 1 transparency
   d. None of these.

(xv) Which of the following statements is true?
   a. DBMS transparency is provided by homogeneous distributed DBMS
   b. DBMS transparency is provided by heterogeneous distributed DBMS
   c. DBMS transparency is provided by both homogeneous and heterogeneous distributed DBMS
   d. DBMS transparency is not provided by either of homogeneous or heterogeneous DBMS.

Review Questions

1. Describe the objectives of data distribution.
2. "Bottom-up approach of distributed database design is applicable for integrating existing databases"; justify your answer.
3. Discuss the issues that have to be considered during the distributed database design.
4. Define data allocation. What are the factors that affect data allocation? Write down the different strategies for data allocation.
5. Comment on the information required for data allocation. Design an allocation model that will attempt to minimize the total cost of processing and storage while trying to meet response time restrictions.
6. What do you mean by fragmented schema? What are the advantages and disadvantages of replication?
7. What is fragmentation? Discuss the different types of fragmentation with examples.
8. Describe the advantages and disadvantages of fragmentation. What are the effects of replication on fragmentation?
9. What are the advantages of keeping the fragmentation schema independent of the allocation schema?
10. Write down the objectives of data allocation. Explain how no loss of information, no redundancy of information, and functional dependencies are preserved during fragmentation.
11. Define derived fragmentation illustrated with example. Prove that derived fragmentation satisfies all the correctness rules.
12. Explain the notions of transparency and autonomy. Discuss the difference between fragmentation transparency, replication transparency, and location transparency.
13. Discuss the different types of distributed transparency.
14. Explain how distribution transparency is achieved for distributed databases in multiple levels?
15. Define distribution transparency. How significant is this in the context of distributed databases?
16. Define naming transparency. Discuss how naming transparency is achieved?
17. Discuss transaction transparency.

18. Comment on the following statements:
    a. "Derived horizontal fragmentation might lead to violation of disjointness condition in between fragments".
    b. "Horizontal fragments are necessarily disjoint".
    c. "Redundancy control is not at all an objective for designing distributed database systems, since the same fragment may reside in multiple sites".
19. Define midterm predicate. Briefly describe the methodology for identification of primary horizontal fragments towards the design of a distributed database system.
20. Explain the significance of derived fragmentation towards efficient design of a distributed database system using a suitable example.
21. Consider the following schemas:

    Global schema: Employee (emp-id, name, dept, design, salary)

    Fragment schema: EMP1 = $\sigma_{\text{dept = "DEVELOPMENT"}}$(Employee)

    EMP2 = $\sigma_{\text{dept = "PRODUCTION"}}$(Employee)

    Allocation schema: EMP1 at sites 1, 2 and EMP2 at sites 3, 4.

    Assume that "DEVELOPMENT" and "PRODUCTION" are only possible values for the dept.

    (a) Write an application that requires the employee id (emp-id) from the terminal and outputs the name, design, salary, and department at levels fragment, location, and local mapping transparency.

    (b) Write an application that moves the employee having emp-id 56 from the department "DEVELOPMENT" to the department "PRODUCTION" at different levels of data distribution transparency.

    (c) Write an application that moves an employee whose emp-id and dept is given at the terminal to the other department at location transparency.
22. Consider the following schemas:

    Global schema: Student (regno, name, coursename, year_of admission, DOB)

    Fragment schema: STU1 = $\sigma_{\text{coursename = "MBA"}}(\Pi_{\text{regno, name, course-name}}(\text{Student}))$

    STU2 = $\sigma_{\text{coursename = "MBA"}}(\Pi_{\text{regno, year-of-admission, DOB}}(\text{Student}))$

    STU3 = $\sigma_{\text{coursename = "PGDBM"}}(\Pi_{\text{regno, name, course-name}}(\text{Student}))$

    STU4 = $\sigma_{\text{coursename = "PGDBM"}}(\Pi_{\text{regno,year-of-admission, DOB}}(\text{Student}))$

    Allocation schema: STU1 at sites 1, 5

    STU2 at sites 2, 6

    STU3 at sites 3, 7 and

    STU4 at sites 4, 8.

    Assume that "MBA" and "PGDBM" are only possible values for the course name.

    (a) Write an application that requires the student id (regno) from the terminal and outputs the name, course name, DOB, and year-of-admission at levels fragment, location, and local mapping transparency.

(b) Write an application that moves the student no 102 from the course "MBA" to the course "PGDBM" at different levels of data distribution transparency.

(c) Write an application that moves a student whose regno and course-name is given at the terminal to the other course site at location transparency.

23. Give an example of a Bank application, accessing a database that is distributed over the branches of the bank, in which the relevant predicates for data distribution are not in the text of the application program.

24. Give an example of an IT company, accessing a database that is distributed over the regions of the country, in which the relevant predicates for data distribution are not in the text of the application program.

25. Consider the following two relations: EMP and PAY.

EMP

Eno	Ename	Title
E1	J. Doe	Elect. Eng.
E2	M. Smith	Syst. Anal.
E3	A. Lee	Mech. Eng.
E4	J. Miller	Programmer
E5	B. Cassey	Syst. Anal.
E6	L. Chu	Elect. Eng.
E7	R. Davis	Mech. Eng.
E8	J. Jones	Syst. Anal.

PAY

Title	Salary
Elect. Eng.	45,000
Syst. Anal.	32,000
Mech. Eng.	28,000
Programmer	25,000

Assume that P1: Salary ,28,000 and P2: salary $28,000 are two simple predicates. Perform a horizontal fragmentation of PAY with respect to predicates P1 and P2 to obtain two fragments PAY1 and PAY2. Using these fragments, perform derived fragmentation for EMP and prove completeness, reconstruction, and disjointness rules for fragmentation of EMP relation are satisfied.

26. Consider a Course Management System. The following information is to be stored:

a. Course description including course-id, coursename, duration, fees, and intake-no.

b. Student description including regno, name, DOB, and address

c. Admission information contains student regno, course-id, and total number of students enrolled.

Assume that there are two applications as follows:

a. An enquiry about the seat availability for admission, which is to be displayed at the terminal.

b. A request for admission in a course, which includes the checking of availability of the seat and insertion of enrollment details.

Design the distributed database over three sites, which are geographically located at "eastern" region, "northern" region, and "southern" region. Give the statistics for the application that are considered for justifying your design. The design includes the following:

(a) The definition of the global schema.

(b) The definition of the fragmentation schema in case of horizontal fragmentation with a complete and minimal set of predicates.

(c) The definition of the allocation schema.

(d) The reconstruction of global relations from fragments.

27. Consider a Project Management System. The following information is to be stored.

a. Project description including project-id, projectname, project-type, start-date, and total-req.

b. Employee description including emp-id, name, design, specialization, dept, and salary.

c. Employee allocation information contains student emp-id, project-id, and total number of employees allocated in project.

Assume that there are two applications as follows:

a. An enquiry about the manpower allocation for projects, which is to be displayed at the terminal.

b. A request for allocation in a project, which includes the checking of manpower requirement for the project and insertion of allocation details.

Design the distributed database over four sites, which are geographically located at "eastern" region, "northern" region, "western" region, and "southern" region. Give the statistics for the application that are considered for justifying your design. The design includes the following:

(a) The definition of the global schema.

(b) The definition of the fragmentation schema in case of horizontal fragmentation with a complete and minimal set of predicates.

(c) The definition of the allocation schema.

(d) The reconstruction of global relations from fragments.

28. Consider a movie multiplex reservation system. The information to be stored is given below.

1. Show information: show-id, show-time, auditorium number, total number of seats, number of seats reserved, and price of tickets.

2. Viewer information: viewer-id, name, address, phone number.

3. Reservation information: viewer-id, show-id, seat number.

Design a distributed database solution for three booking counters across the city considering the following frequent queries:

Availability of tickets: Booking information of a particular show is displayed on the screen.

Request for booking: Checking and inserting viewer's data for new customers, checking for availability of tickets, inserting booking description.

The design should include the definition of global schema, fragmentation schema, and allocation schema.

29. Consider the following schemas:

    Global schema: Guest (guest-id, name, block-id, room-no)

    Fragment schema: GUEST1 = $\sigma_{\text{block-id = "NORTH"}}$ (Guest)

    GUEST2 = $\sigma_{\text{block-id = "SOUTH"}}$ (Guest)

    Allocation schema: GUEST1 at sites 1, 2 and GUEST2 at sites 3, 4.

    (a) Write an application that accepts the guest-id from the terminal and outputs the name, block-id, and the room number at levels fragment, location, and local mapping transparency.

    (b) Write an application that moves a guest having guest-id 541 from the north block to the south block at different levels of data distribution transparency.

30. A country-wide drug supplier chain operates from five different cities in the country and it maintains the following database:

    Shop (ds-id, ds-city, ds-contactno)

    Medicine (med-id, med-name, manuf-id)

    Manufacturer (manuf-id, manu-name, manu-city)

    Order (med-id, ds-id, qty)

    Suggest a fragmentation and allocation schema considering the following frequent queries:

    a. List the manufacturers' names who belong to the same city in which the drug shop that has placed an order resides.

    b. How many orders are generated from a city, say "C"?

    Justify your design and mention your assumptions clearly.

This page is intentionally left blank

6 Distributed DBMS Architecture

This chapter introduces the architecture of different distributed systems such as client/server system and peer-to-peer distributed system. Owing to the diversity of distributed systems, it is very difficult to generalize the architecture of distributed DBMSs. Different alternative architectures of the distributed database systems and the advantages and disadvantages of each system are discussed in detail. This chapter also introduces the concept of a multi-database system (MDBS), which is used to manage the heterogeneity of different DBMSs in a heterogeneous distributed DBMS environment. The classification of MDBSs and the architecture of such databases are presented in detail.

The outline of this chapter is as follows. Section 6.2 introduces different alternative architectures of client/server systems and pros and cons of these systems. In Section 6.3 alternative architectures for peer-to-peer distributed systems are discussed. Section 6.4 focuses on MDBSs. The classifications of MDBSs and their corresponding architectures are illustrated in this section.

6.1 Introduction

The architecture of a system reflects the structure of the underlying system. It defines the different components of the system, the functions of these components and the overall interactions and relationships between these components. This concept is true for general computer systems as well as for software systems. The software architecture of a program or computing system is the structure or structures of the system, which comprises software elements or modules, the externally visible properties of these elements and the relationships between them. Software architecture can be thought of as the representation of an engineering system and the process(es) and discipline(s) for effectively implementing the design(s) of such a system.

A distributed database system can be considered as a large-scale software system; thus, the architecture of a distributed system can be defined in a manner similar to that of software systems. This chapter introduces the different alternative reference architectures of distributed database systems such as client/server, peer-to-peer and MDBSs.

6.2 Client/Server System

In the late 1970s and early 1980s smaller systems (mini computer) were developed that required less power and air conditioning. The term client/server was first used in the 1980s, and it gained acceptance in referring to personal computers (PCs) on a network. In the late 1970s, Xerox developed the standards and technology that is familiar today as the Ethernet. This provided a standard means

for linking together computers from different manufactures and formed the basis for modern local area networks (LANs) and wide area networks (WANs). Client/server system was developed to cope up with the rapidly changing business environment. The general forces that drive the move to client/server systems are as follows:

- A strong business requirement for decentralized computing horsepower.
- Standard, powerful computers with user-friendly interfaces.
- Mature, shrink-wrapped user applications with widespread acceptance.
- Inexpensive, modular systems designed with enterprise class architecture, such as power and network redundancy and file archiving network protocols, to link them together.
- Growing cost/performance advantages of PC-based platforms.

The client/server system is a versatile, message-based and modular infrastructure that is intended to improve usability, flexibility, interoperability and scalability as compared to centralized, mainframe, time-sharing computing. In the simplest sense, the client and the server can be defined as follows:

- A **Client** is an individual user's computer or a user application that does a certain amount of processing on its own and sends and receives requests to and from one or more servers for other processing and/or data.
- A **Server** consists of one or more computers or an application program that receives and processes requests from one or more client machines. A server is typically designed with some redundancies in power, network, computing, and file storage.

Usually, a client is defined as a requester of services, and a server is defined as the provider of services. A single machine can be both a client and a server depending on the software configuration. Sometimes, the term server or client refers to the software rather than the machines. Generally, server software runs on powerful computers dedicated for exclusive use of business applications. On the other hand, client software runs on common PCs or workstations. The properties of a server are:

- Passive (slave)
- Waiting for requests
- On request serves clients and sends reply.

The properties of a client are:

- Active (Master)
- Sending requests
- Waits until reply arrives.

A server can be stateless or stateful. A stateless server does not keep any information between requests. A stateful server can remember information between requests.

6.2.1 Advantages and Disadvantages of Client/Server System

A client/server system provides a number of advantages over a powerful mainframe centralized system. The major advantage is that it improves usability, flexibility, interoperability and scalability

as compared to centralized, time-sharing, mainframe computing. In addition, a client/server system has the following advantages:

- A client/server system has the ability to distribute the computing workload between client workstations and shared servers.
- A client/server system allows the end user to use a microcomputer's graphical user interfaces, thereby improving functionality and simplicity.
- It provides better performance at a reduced cost for hardware and software than alternative mini or mainframe solutions.

The client/server environment is more difficult to maintain for a variety of reasons, which are as follows:

- The client/server architecture creates a more complex environment in which it is often difficult to manage different platforms (LANs, operating systems, DBMS etc.).
- In a client/server system, the operating system software is distributed over many machines rather than a single system, thereby increasing complexity.
- A client/server system may suffer from security problems as the number of users and processing sites increases.
- The workstations are geographically distributed in a client/server system, and each of these workstations is administrated and controlled by individual departments, which adds extra complexity. Furthermore, communication cost is incurred with each processing.
- The maintenance cost of a client/server system is greater than that of an alternative mini or mainframe solution.

6.2.2 Architecture of Client/Server Distributed Systems

Client/server architecture is a prerequisite to the proper development of client/server systems. The Client/Server architecture is based on hardware and software components that interact to form a distributed system. In a client/server distributed database system, entire data can be viewed as a single logical database while at the physical level data may be distributed. From the data organizational view, the architecture of a client/server distributed database system is mainly concentrated on software components of the system, and this system includes three main components: clients, servers and communications middleware.

(i) A **Client** is an individual computer or process or user's application that requests services from the server. A Client is also known as **front-end application,** as the end user usually interacts with the client process. The software components required in the client machine are the client operating system, client DBMS and client graphical user interface. Client process is run on an operating system that has at least some multi-tasking capabilities. The end users interact with the client process via a graphical user interface. In addition, a client DBMS is required at the client side, which is responsible for managing the data that is cached in the client. In some client/server architectures, communication software is embedded into the client machine, as a substitute for communication middleware, to interact efficiently with other machines in the network.

(ii) A **Server** consists of one or more computers or is a computer process or application that provides services to clients. A Server is also known as **back-end application,** as the server process provides the background services for the client processes. A server provides most of the data management services such as query processing and optimization, transaction management, recovery management, storage management and integrity maintenance services to the clients. In addition, sometimes communication software is embedded into the server machine, instead of communication middleware, to manage communications with clients and other servers in the network.

(iii) **Communication middleware** is any process(es) through which clients and servers communicate with each other. The communication middleware is usually associated with a network that controls data and information transmission between clients and servers. Communication middleware software consists of three main components: application program interface (API), database translator and network translator. The API is public to client applications through which they can communicate with the communication middleware. The middleware API allows the client process to be database server–independent. The database translator translates the SQL requests into the specific database server syntax, thus enabling a DBMS from one vendor to communicate directly with a DBMS from another vendor, without using a gateway. The network translator manages the network communications protocols; thus, it allows clients to be network protocol–independent. To accomplish the connection between the client and the server, the communication middleware software operates at two different levels. The physical level deals with the communications between the client and the server computers (computer to computer) whereas the logical level deals with the communications between the client and the server processes (interprocess). The basic client/server architecture is illustrated in figure 6.1.

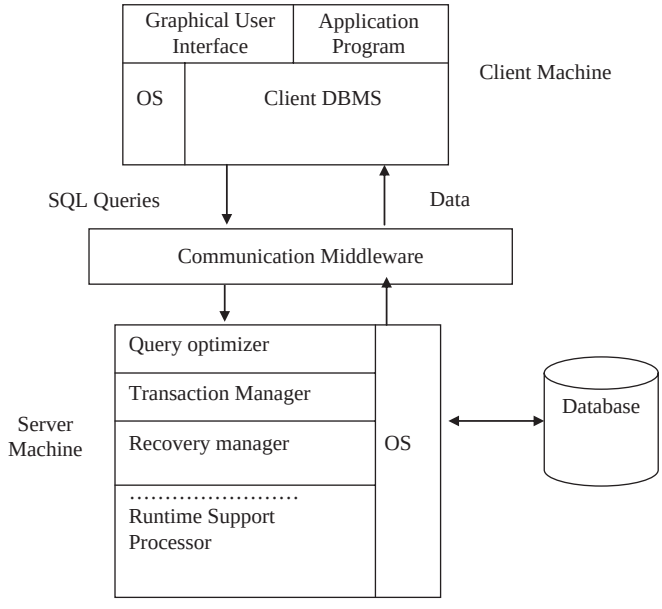


Fig. 6.1 Client/Server Reference Architecture

The client/server architecture is intended to provide a scalable architecture, whereby each computer or process on the network is either a client or a server.

6.2.3 Architectural Alternatives for Client/Server Systems

A client/server system can have several architectural alternatives known as two-tier, three-tier and multi-tier or *n*-tier.

Two-tier architecture – A generic client/server architecture has two types of nodes on the network: clients and servers. As a result, these generic architectures are sometimes referred to as **two-tier** architectures. In two-tier client/server architecture, the user system interface is usually located in the user's desktop environment, and the database management services are usually located on a server that services many clients. Processing management is split between the user system interface environment and the database management server environment. The general two-tier architecture of a Client/Server system is illustrated in figure 6.2.

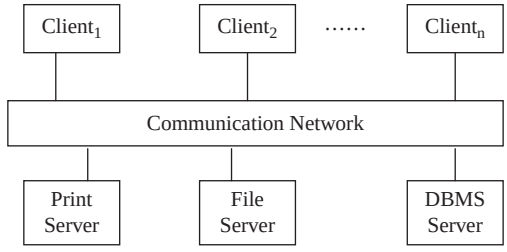


Fig. 6.2 Two-tier Client/Server Architecture

In a two-tier client/server system, it may occur that multiple clients are served by a single server, called **multiple clients–single server approach**. Another alternative is multiple servers providing services to multiple clients, which is called **multiple clients–multiple servers approach**. In the case of multiple clients–multiple servers approach, two alternative management strategies are possible: either each client manages its own connection to the appropriate server or each client communicates with its home server, which further communicates with other servers as required. The former approach simplifies server code but complicates the client code with additional functionalities, which leads to a **heavy (fat) client system**. On the other hand, the latter approach loads the server machine with all data management responsibilities and thus leads to a **light (thin) client system**. Depending on the extent to which the processing is shared between the client and the server, a server can be described as fat or thin. A fat server carries the larger proportion of the processing load, whereas a thin server carries a lesser processing load.

The two-tier client/server architecture is a good solution for distributed computing when work groups of up to 100 people are interacting on a local area network simultaneously. It has a number of limitations also. The major limitation is that performance begins to deteriorate when the number of users exceeds 100. A second limitation of the two-tier architecture is that implementation of processing management services using vendor proprietary database procedures restricts flexibility and choice of DBMS for applications.

Three-tier architecture – Some networks of client/server architecture consist of three different kinds of nodes: clients, application servers, which process data for the clients, and database servers, which store data for the application servers. This arrangement is called **three-tier** architecture. The three-tier architecture (also referred to as **multi-tier** architecture) emerged to overcome the limitations of the two-tier architecture. In the three-tier architecture, a middle tier was added between the user system interface client environment and the database management server environment. The middle tier can perform queuing, application execution, and database staging. There are various ways of implementing the middle tier, such as transaction processing monitors, message servers, web servers, or application servers. The typical three-tier architecture of a client/server system is depicted in figure 6.3.

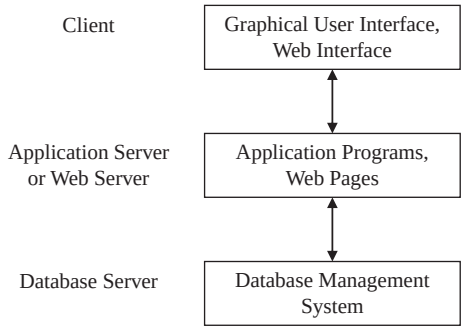


Fig. 6.3 Three-tier Client/Server Architecture

The most basic type of three-tier architecture has a middle layer consisting of transaction processing (TP) monitor technology. The TP monitor technology is a type of message queuing, transaction scheduling and prioritization service where the client connects to the TP monitor (middle tier) instead of the database server. The transaction is accepted by the monitor, which queues it and takes responsibility for managing it to completion, thus, freeing up the client. TP monitor technology also provides a number of services such as updating multiple DBMSs in a single transaction, connectivity to a variety of data sources including flat files, non-relational DBMSs and the mainframe, the ability to attach priorities to transactions and robust security. When all these functionalities are provided by third-party middleware vendors, it complicates the TP monitor code, which is then referred to as **TP heavy**, and it can service thousands of users. On the other hand, if all these functionalities are embedded in the DBMS and can be considered as two-tier architecture, it is referred to as **TP Lite**. A limitation of TP monitor technology is that the implementation code is usually written in a lower level language, and the system is not yet widely available in the popular visual toolsets.

Messaging is another way of implementing three-tier architectures. Messages are prioritized and processed asynchronously. The message server connects to the relational DBMS and other data sources. The message server architecture mainly focuses on intelligent messages. Messaging systems are good solutions for wireless infrastructure.

The three-tier architecture with a middle layer consisting of an application server allocates the main body of an application on a shared host for execution, rather than in the user system interface client environment. The application server hosts business logic, computations, and a data retrieval engine. Thus, major advantages with application server are with less software on the client side there is less security to worry about, applications are more scalable, and installation costs are less on a single server compared to maintaining each on a desktop client.

Currently, developing client/server systems using technologies that support distributed objects is gaining popularity, as these technologies support interoperability across languages and platforms, as well as enhancing maintainability and adaptability of the system. There are currently two prominent distributed object technologies: one is Common Object Request Broker Architecture (CORBA) and the other is COM (Component Object Model)/DCOM.

The major advantage of three-tier client/server architecture is that it provides better performance for groups with a large number of users and improves flexibility compared to two-tier approach. In the case of three-tier architecture, as data processing is separated from different servers, it provides more scalability. The disadvantage of three-tier architecture is that it puts a greater load on the network. Moreover, in the case of three-tier architecture, it is much more difficult to develop and test software than in two-tier architecture, because more devices have to communicate to complete a user's transaction.

In general, a multi-tier (or *n*-tier) architecture may deploy any number of distinct services, including transitive relations between application servers implementing different functions of business logic, each of which may or may not employ a distinct or shared database system.

6.3 Peer-to-Peer Distributed System

The peer-to-peer architecture is a good way to structure a distributed system so that it consists of many identical software processes or modules, each module running on a different computer or node. The different software modules stored at different sites communicate with each other to complete the processing required for the execution of distributed applications. Peer-to-peer architecture provides both client and server functionalities on each computer. Therefore, each node can access services from other nodes as well as providing services to other nodes. In contrast with the client/server architecture, in a peer-to-peer distributed system each node provides user interaction facilities as well as processing capabilities.

Considering the complexity associated with discovering, communicating with and managing the large number of computers involved in a distributed system, the software module at each node in a peer-to-peer distributed system is typically structured in a layered manner. Thus, the software modules of peer-to-peer applications can be divided into the three layers, known as **the base overlay layer, the middleware layer, and the application layer**. The base overlay layer deals with the issue of discovering other participants in the system and creating a mechanism for all nodes to communicate with each other. This layer ensures that all participants in the network are aware of other participants. The middleware layer includes additional software components that can be potentially reused by many different applications. The functionalities provided by this layer include the ability to create a distributed index for information in the system, a publish/subscribe facility and security services. The functions provided by the middleware layer are not necessary for all applications, but they are developed to be reused by more than one application. The application layer provides software packages intended to be used by users and developed so as to exploit the distributed nature of the peer-to-peer infrastructure. There is no standard terminology across different implementations of the peer-to-peer system, and thus, the term "peer-to-peer" is used for general descriptions of the functionalities required for building a generic peer-to-peer system. Most of the peer-to-peer systems are developed as single-application systems.

As a database management system, each node in a peer-to-peer distributed system provides all data management services, and it can execute local queries as well as global queries. Thus, in

this system there is no distinction between client DBMS and server DBMS. As a single-application program, DBMS at each node accepts user requests and manages execution. Like in client/server system, also in a peer-to-peer distributed database system data is viewed as a single logical database although the data is distributed at the physical level. In this context, the identification of the reference architecture for a distributed database system is necessary.

6.3.1 Reference Architecture of Distributed DBMSs

This section introduces the reference architecture of a distributed database system. Owing to diversities of distributed DBMSs, it is much more difficult to represent a common architecture that is generally applicable for all applications. However, it may be useful to represent a possible reference architecture that addresses data distribution. Data in a distributed system are usually fragmented and replicated. Considering this fragmentation and replication issue, the reference architecture of a distributed DBMS consists of the following schemas (as depicted in figure 6.4):

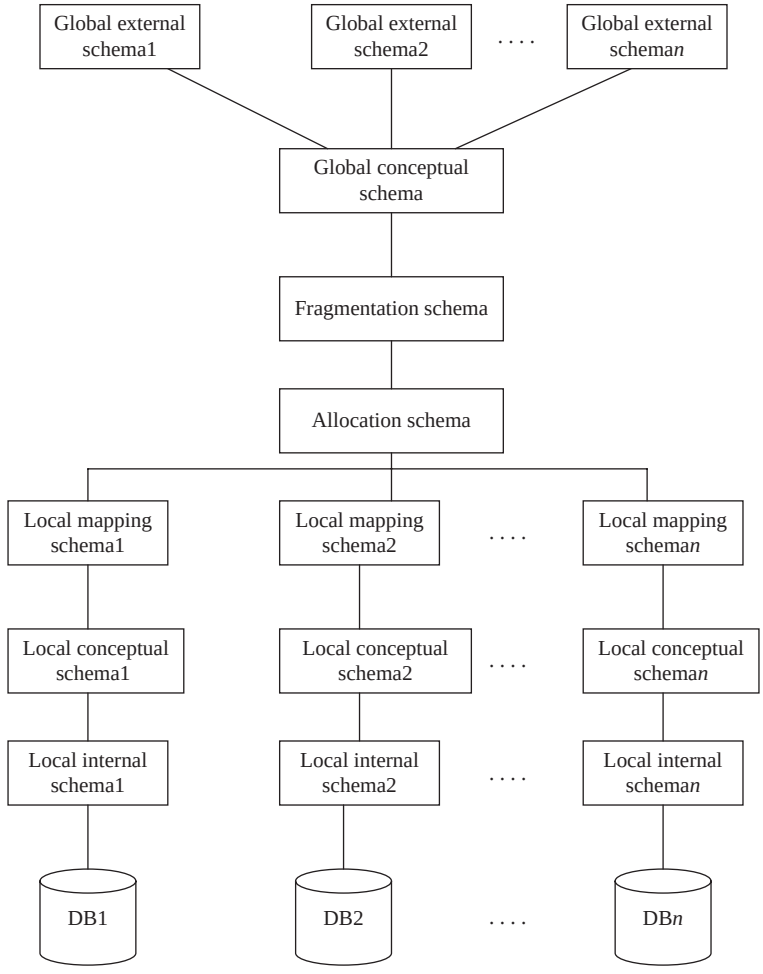


Fig. 6.4 Reference Architecture of Distributed DBMS

- A set of global external schemas
- A global conceptual schema (GCS)
- A fragmentation schema and allocation schema
- A set of schemas for each local DBMS, conforming to the ANSI–SPARC three-level architecture.

Global external schema – In a distributed system, user applications and user accesses to the distributed database are represented by a number of global external schemas. This is the topmost level in the reference architecture of a distributed DBMS. This level describes the part of the distributed database that is relevant to different users.

Global conceptual schema – The GCS represents the logical description of the entire database as if it is not distributed. This level corresponds to the conceptual level of the ANSI–SPARC architecture of centralized DBMS and contains definitions of all entities, relationships among entities and security and integrity information for the whole database stored at all sites in a distributed system.

Fragmentation schema and allocation schema – In a distributed database, the data can be split into a number of non-overlapping portions, called fragments. There are several different ways to perform this fragmentation operation. The fragmentation schema describes how the data is to be logically partitioned in a distributed database. The GCS consists of a set of global relations, and the mapping between the global relations and fragments is defined in the fragmentation schema. This mapping is one-to-many, that is, a number of fragments correspond to one global relation but only one global relation corresponds to one fragment. The allocation schema is a description of where the data (fragments) are to be located, taking account of any replication. The type of mapping defined in the allocation schema determines whether the distributed database is redundant or non-redundant. In the case of redundant data distribution, the mapping is one-to-many, whereas in the case of non-redundant data distribution the mapping is one-to-one.

Local schemas – Each local DBMS in a distributed system has its own set of schemas. The local conceptual and local internal schemas correspond to the equivalent levels of ANSI–SPARC architecture. In a distributed database system, the physical data organization at each machine is probably different, and therefore it requires an individual internal schema definition at each site, called **local internal schema**. To handle fragmentation and replication issues, the logical organization of data at each site is described by a third layer in the architecture, called **local conceptual schema**. The GCS is the union of all local conceptual schemas; thus, the local conceptual schemas are mappings of the global schema onto each site. This mapping is done by **local mapping schemas**. The local mapping schema maps fragments in the allocation schema onto external objects in the local database, and this mapping depends on the type of local DBMS. Therefore, in a heterogeneous distributed DBMS, there may be different types of local mappings at different nodes.

This architecture provides a very general conceptual framework for understanding distributed databases. Furthermore, such databases are typically designed in a top-down manner; therefore, all external-view definitions are made globally.

6.3.2 Component Architecture of Distributed DBMSs

This section introduces the component architecture of a distributed DBMS that is independent of the reference architecture. The four major components of a distributed DBMS that has been identified are as follows:

- » Distributed DBMS (DDBMS) component
- » Data communications (DC) component
- » Global system catalog (GSC)
- » Local DBMS (LDBMS) component

Distributed DBMS (DDBMS) component The DDBMS component is the controlling unit of the entire system. This component provides the different levels of transparencies such as data distribution transparency, transaction transparency, performance transparency and DBMS transparency (in the case of heterogeneous DDBMS). *Ozsu* has identified four major componets of DDBMS as listed below:

(a) The user interface handler – This component is responsible for interpreting user commands as they come into the system and formatting the result data as it is sent to the user.

(b) The semantic data controller – This component is responsible for checking integrity constraints and authorizations that are defined in the GCS, before processing the user requests.

(c) The global query optimizer and decomposer – This component determines an execution strategy to minimize a cost function and translates the global queries into local ones using the global and local conceptual schemas as well as the global system catalog. The global query optimizer is responsible for generating the best strategy to execute distributed join operations.

(d) The distributed execution monitor – It coordinates the distributed execution of the user request. This component is also known as distributed transaction manager. During the execution of distributed queries, the execution monitors at various sites may and usually do communicate with one another.

DC component. The DC component is the software that enables all sites to communicate with each other. The DC component contains all information about the sites and the links.

Global system catalog (GSC) The GSC provides the same functionality as system catalog of a centralized DBMS. In addition to metadata of the entire database, a GSC contains all fragmentation, replication and allocation details considering the distributed nature of a DDBMS. It can itself be managed as a distributed database and thus, it can be fragmented and distributed, fully replicated or centralized like any other relations in the system. [The details of GSC management will be introduced in **Chapter 12, Section 12.2**].

Local DBMS (LDBMS) component The LDBMS component is a standard DBMS, stored at each site that has a database and responsible for controlling local data. Each LDBMS component has its own local system catalog that contains the information about the data stored at that particular site. In a homogeneous DDBMS, the LDBMS component is the same product, replicated at each site, while in a heterogeneous DDBMS, there must be at least two sites with different DBMS products and/or platforms. The major components of an LDBMS are as follows:

(a) The local query optimizer – This component is used as the access path selector and responsible for choosing the best access path to access any data item for the execution of a query (the query may be a local query or part of a global query executed at that site).

(b) The local recovery manager – The local recovery manager ensures the consistency of the local database inspite of failures.

(c) The run-time support processor – This component physically accesses the database according to the commands in the schedule generated by the query optimizer and is responsible for managing main memory buffers. The run-time support processor is the interface to the operating system and contains the database buffer (or cache) manager.

6.3.3 Distributed Data Independence

The reference architecture of a DDBMS is an extension of ANSI–SPARC architecture; therefore, data independence is supported by this model. Distributed data independence means that upper levels are unaffected by changes to lower levels in the distributed database architecture. Like a centralized DBMS, both **distributed logical data independence** and **distributed physical data independence** are supported by this architecture. In a distributed system, the user queries data irrespective of its location, fragmentation or replication. Furthermore, any changes made to the GCS do not affect the user views in the global external schemas. Thus, distributed logical data independence is provided by global external schemas in distributed database architecture. Similarly, the GCS provides distributed physical data independence in the distributed database environment.

6.4 Multi-Database System (MDBS)

In recent years, MDBS has been gaining the attention of many researchers who attempt to logically integrate several different independent DDBMSs while allowing the local DBMSs to maintain complete control of their operations. Complete autonomy means that there can be no software modifications to the local DBMSs in a DDBMS. Hence, an MDBS, which provides the necessary functionality, is introduced as an additional software layer on top of the local DBMSs.

An **MDBS** is a software that can be manipulated and accessed through a single manipulation language with a single common data model (i.e., through a single application) in a heterogeneous environment without interfering with the normal execution of the individual database systems. The MDBS has developed from a requirement to manage and retrieve data from multiple databases within a single application while providing complete autonomy to individual database systems. To support DBMS transparency, MDBS resides on top of existing databases and file systems and presents a single database to its users. An MDBS maintains a global schema against which users issue queries and updates, and this global schema is constructed by integrating the schemas of local databases. To execute a global query, the MDBS first translates it into a number of subqueries, and converts these subqueries into appropriate local queries for running on local DBMSs. After completion of execution, local results are merged and the final global result for the user query is generated. An MDBS controls multiple gateways and manages local databases through these gateways.

MDBSs can be classified into two different categories based on the autonomy of the individual DBMSs. These are **non-federated MDBSs** and **federated MDBSs**. A federated MDBS is again categorized as **loosely coupled federated MDBS** and **tightly coupled federated MDBS** based on who manages the federation and how the components are integrated. Further, a tightly coupled federated MDBS can be classified as **single federation tightly coupled federated MDBS** and **multiple federations tightly coupled federated MDBS**. The complete taxonomy of MDBSs [Sheth and Larson, 1990] is depicted in figure 6.5.

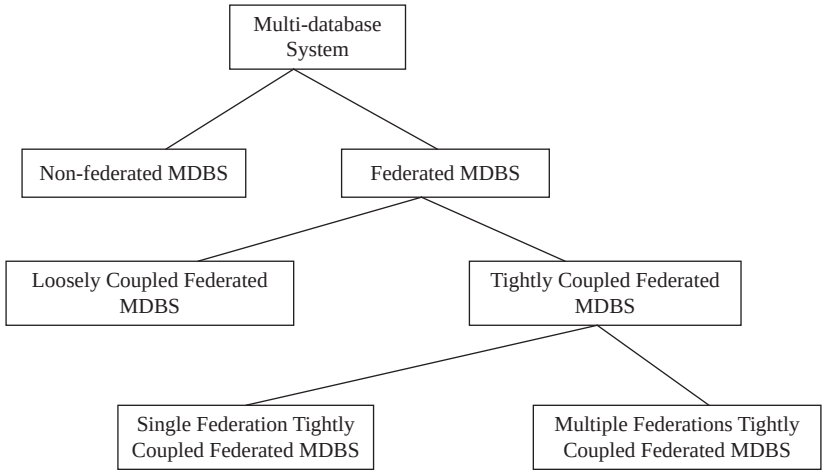


Fig. 6.5 Taxonomy of Multi-database Systems

Federated MDBS (FMDBS) It is a collection of cooperating database management systems that are autonomous but participate in a federation to allow partial and controlled sharing of their data. In a federated MDBS, all component DBMSs cooperate to allow different degrees of integration. There is no centralized control in a federated architecture because the component databases control access to their data. To allow controlled sharing of data while preserving the autonomy of component DBMSs and continued execution of existing applications, a federated MDBS supports two types of operations: local or global (or federation). Local operations are directly submitted to a component DBMS, and they involve data only from that component database. Global operations access data from multiple databases managed by multiple component DBMSs via the FMDBS. Thus, an FMDBS is a cross between a DDBMS and a centralized DBMS. It is a distributed system to global users whereas a centralized DBMS to local users. In a simple way, an MDBS is said to be an FMDBS, if users interface to the MDBS through some integrated views, and there is no connection between any two integrated views. The features of an FMDBS are listed below:

- **Integrated schema exists** – The FMDBS administrator (MDBA) is responsible for the creation of integrated schemas in a heterogeneous environment.
- **Component databases are transparent to users** – Users are not aware of the multiple component DBMSs in an FMDBS; thus, the users only need to understand the integrated schemas to implement the operations on an FMDBS. They cannot change the integrated components when they operate this FMDBS.
- **A common data model (CDM) is required to implement the federation** – The CDM must be very powerful to represent all data models in the different components. The integration of export schemas of component data models is placed on the CDM.
- **Update transactions are a difficult issue in FMDBS** – The component databases are completely independent and join the federation through the integrated schema. It is difficult to decide whether the FMDBS or the local component database systems will control the transactions.

Two types of FMDBSs have been identified, namely, **loosely coupled FMDBS** and **tightly coupled FMDBS** depending on how multiple component databases are integrated. An FMDBS is

loosely coupled if it is the user's responsibility to create and maintain the federation and there is no control enforced by the federated system and its administrators. Similarly, an FMDBS is tightly coupled if the federation and its administrator(s) have the responsibility for creating and maintaining the integration and they actively control the access to component databases. A federation is built by a selective and controlled integration of its components. A tightly coupled FMDBS may have one or more federated schemas. A tightly coupled FMDBS is said to have **single federation** if it allows the creation and management of only one federated schema. On the other hand, a tightly coupled FMDBS is said to have **multiple federations** if it allows the creation and management of multiple federated schemas. A loosely coupled FMDBS always supports multiple federated schemas.

Non-federated MDBS In contrast to a FMDBS, a non-federated MDBS does not distinguish local and global users. In a non-federated MDBS, all component databases are fully integrated to provide a single global schema known as the unified MDBS (sometimes called enterprise or corporate). Thus, in a non-federated MDBS, all applications are global applications (because there is no local user) and data are accessed through a single global schema. It logically appears to its users like a distributed database.

6.4.1 Five-Level Schema Architecture of federated MDBS

The terms federated database system and federated database architecture were introduced by Heimbigner and McLeod (1985) to facilitate interactions and sharing among independendently designed databases. Their main purpose was to build up a loosely coupled federation of different component databases. Sheth and Larson (1990) have identified a five-level schema architecture for a federated MDBS to solve the heterogeneity of FMDBSs, which is depicted in figure 6.6.

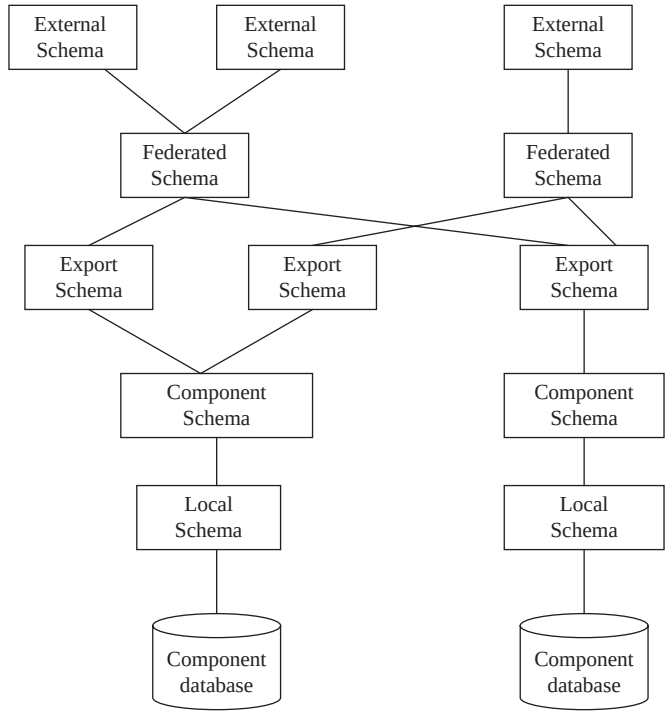


Fig. 6.6 Five-level Schema Architecture of Federated MDBS

(i) Local schema – A local schema is used to represent each component database of a federated MDBS. A local schema is expressed in the native data model of the component DBMS, and hence different local schemas can be expressed in different data models.

(ii) Component schema – A componet schema is generated by translating local schemas into a data model called the **canonical or CDM** of the FMDBS. A component schema is used to facilitate negotiation and integration among divergent local schemas to execute global tasks.

(iii) Export schema – An export schema is a subset of a component schema, and it is used to represent only those portions of a local database that are authorized by the local DBMS for accessing of non-local users. The purpose of defining export schemas is to control and manage the autonomy of component databases.

(iv) Federated schema – A federated Schema is an integration of multiple export schemas. It is always connected with a data dictionary that stores the information about the data distribution and the definitions of different schemas in the heterogeneous environment. There may be multiple federated schemas in an FMDBS, one for each class of federation users.

(v) External schema or application schema – An external schema or application schema is derived from the federated schema and is suitable for different users. Application schema can be a subset of a large complicated federated schema or may be changed into a different data model, to fit in a specific user interface for fulfilling the requirements of different users. This allows users to put additional integrity constraints or access-control constraints on the federated schema.

Reference architecture of tightly coupled federated MDBS

The architecture of an FMDBS is primarily determined by which schemas are present, how they are arranged and how they are constructed. The reference architecture is necessary to understand, categorize and compare different architectural options for developing federated database systems. This section describes the reference architecture of a tightly coupled FMDBS. Usually, an FMDBS is designed in a bottom-up manner to integrate a number of existing heterogeneous databases.

In a tightly coupled FMDBS, federated schema takes the form of schema integration. For simplicity, a single (logical) federation is considered for the entire system, and it is represented by a GCS. A number of export schemas are integrated into the GCS, where the export schemas are created through negotiation between the local databases and the GCS. Thus, in an FMDBS, the GCS is a subset of local conceptual schemas and consists of the data that each local DBMS agrees to share. The GCS of a tightly coupled FMDBS involves the integration of either parts of the local conceptual schemas or the local external schemas. Global external schemas are generated through negotiation between global users and the GCS. The reference architecture of a tightly coupled FMDBS is depicted in figure 6.7.

Reference architecture of loosely coupled federated MDBS

In contrast with tightly coupled FMDBS, schema intergration does not take place in loosely coupled FMDBS; therefore, a loosely coupled FMDBS cannot have a GCS. In this case, federated schemas for global users are defined by importing export schemas using a user interface or an application program or by defining a multi-database language query that references export schema objects of local databases. Export schemas are created based on local component databases. Thus, in a loosely coupled FMDBS, a global external schema consists of one or more local conceptual schemas. The reference architecture of a loosely coupled FMDBS is depicted in figure 6.8.

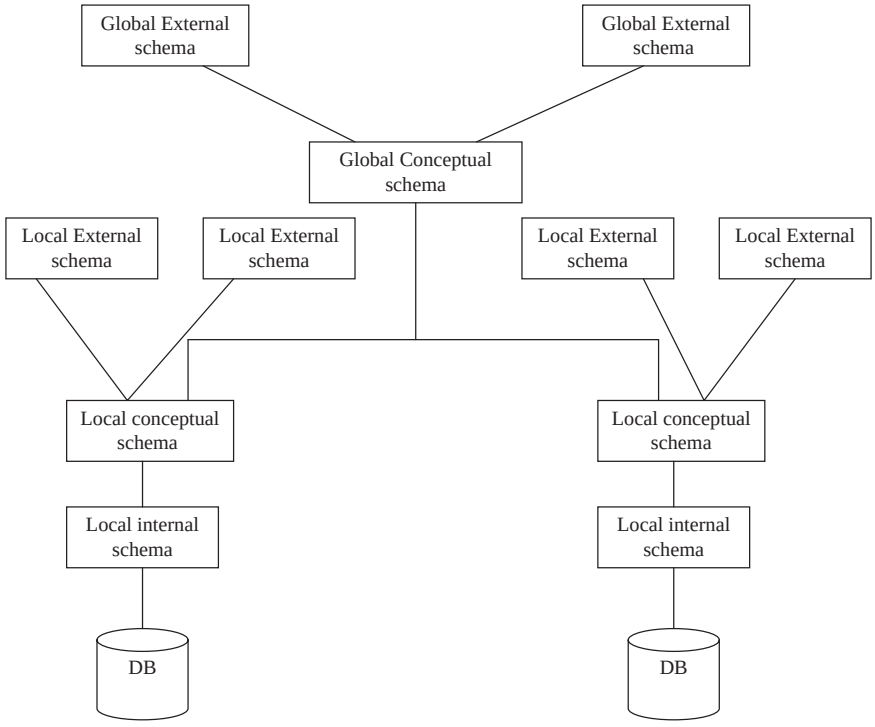


Fig. 6.7 Reference Architecture of Tightly Coupled Federated MDBS

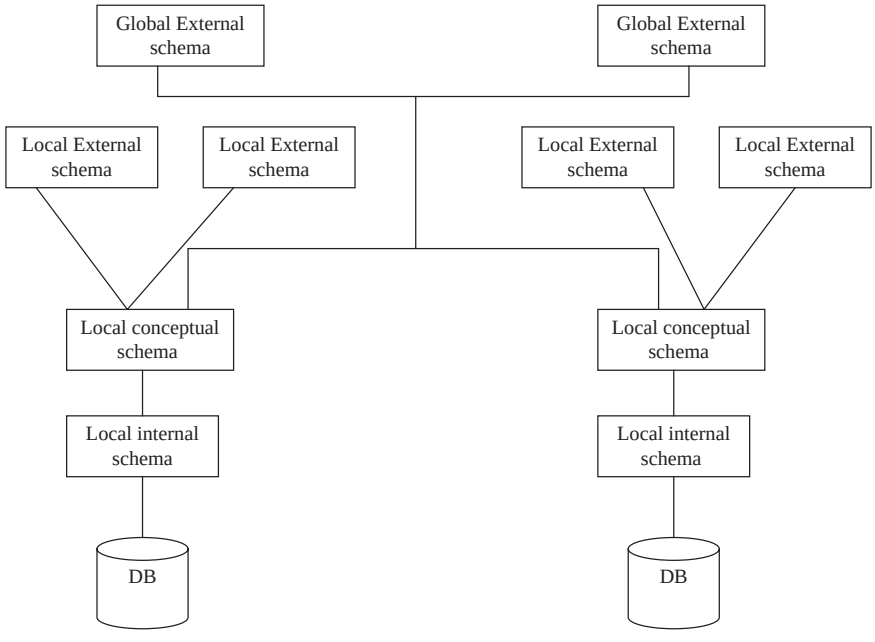


Fig. 6.8 Reference Architecture of Loosely Coupled Federated MDBS

CHAPTER SUMMARY

This chapter introduces several alternative architectures for a distributed database system such as Client/server, peer-to-peer and MDBSs.

» A client/server system is a versatile, message-based and modular infrastructure that is intended to improve usability, flexibility, interoperability and scalability as compared to centralized, time-sharing mainframe computing. In a client/server system, there are two different kinds of nodes: clients and servers. In simplest sense clients request for the services to the server and servers provide services to the clients.

» In peer-to-peer architecture, each node provides user interaction facilities as well as processing capabilities. A peer-to-peer architecture provides both client and server functionalities on each node.

» An MDBS is a software system that attempts to logically integrate several different independent DDBMSs while allowing the local DBMSs to maintain complete control of their operations. MDBSs can be classified into two different categories: Non-federated MDBS and federated MDBS. Federated MDBSs are again categorized as loosely coupled federated MDBSs and tightly coupled federated MDBSs. Further, tightly coupled FMDBSs can be classified as single federation tightly coupled FMDBSs and multiple federations tightly coupled FMDBSs.

EXERCISES

Multiple Choice Questions

(i) Which of the following computing models is used by distributed database systems?
a. Mainframe computing model
b. Disconnect, personal computing model
c. Client/server computing model
d. None of these.

(ii) Which of the following is not a property of a server?
a. Active (Master)
b. Waiting for requests
c. On request serves clients and sends reply
d. None of these.

(iii) Which of the following is not a property of a client?
a. Active (Master)
b. Sending requests
c. Waits until reply arrives
d. None of these.

(iv) Which of the following statements is correct?
a. A heavy client complicates the server code
b. A heavy client simplifies the client code
c. A heavy client simplifies the server code
d. None of these.

(v) A thin client
a. Simplifies the server code
b. Complicates the client code
c. Complicates the server code
d. Simplifies both client and server codes.

(vi) In DDBMSs, distributed physical data independence is provided by
a. Local conceptual schema
b. Local external schema
c. Global conceptual schema
d. Local mapping schema.

(vii) In DDBMSs, distributed logical data independence is provided by
a. Local conceptual schema
b. Global external schema
c. Global conceptual schema
d. Local mapping schema.

(viii) Which of the following statements is correct?
a. There is no local user in a federated MDBS
b. There is no global user in a non-federated MDBS
c. There is no local user in a non-federated MDBS
d. None of these.

(ix) Which of the following statements is incorrect?
a. An MDBS provides DBMS transparency
b. An MDBS provides complete autonomy to individual database systems
c. An MDBS controls multiple gateways
d. All of the above
e. None of the above.

(x) Which of the following statements is true?
   a. A loosely coupled FMDBS has no global conceptual schema
   b. A tightly coupled FMDBS has no global external schema
   c. A loosely coupled FMDBS has no local external schema
   d. A tightly coupled FMDBS has no local conceptual schema.

(xi) In peer-to-peer architecture
   a. Each node has the client functionality
   b. Each node has the same capability
   c. Each node has the client functionality as well as server functionality
   d. Both b. and c.

(xii) Which of the following schema is used in federated MDBS?
   a. Component schema
   b. Federated schema
   c. Export schema
   d. All of these
   e. None of these.

Review Questions

1. Describe the architecture of a client/server system. Compare and contrast client/server and peer-to-peer architectures of DDBMSs.
2. Write down the benefits of client/server System.
3. Compare and contrast two-tier and three-tier architectures of a client/server system.
4. Write a short note on peer-to-peer architecture.
5. Briefly discuss the component architecture of a DDBMS. Draw the reference architecture of distributed databases.
6. Why the top three layers in the reference architecture for distributed database systems are often referred to as site-independent schemas? Define physical image of a global relation at a site.
7. Comment on the different types of dependency for the bottom layers in the reference architecture of DDBMS. Explain the role of local mapping schema towards the integration of heterogeneous multi-site databases in this context.
8. Comment on the statement, "The allocation schema in distributed database architecture is site-independent".
9. What is distributed data independence? Explain how distributed data independence is provided by the architecture of a DDBMS.
10. What is an MDBS? Discuss the utilities of such a database system.
11. Briefly discuss the classification of MDBSs.
12. Differentiate between federated and non-federated MDBSs.
13. Write down the features of a federated MDBS.
14. Compare federated schema and export schema of a federated MDBS.
15. Describe the reference architecture of loosely coupled federated MDBSs.

This page is intentionally left blank

7 Distributed Transaction Management

This chapter introduces the concept of a transaction and its fundamental properties. A transaction consists of a set of operations that represent a single logical unit of work in a database system and moves the database from one consistent state to another. The classification of transactions and the objectives of transaction management in a distributed database environment are clearly explained here. A model for transaction management in a distributed database system is represented in this chapter.

The organization of this chapter is as follows. Section 7.1 introduces the basic concepts of transaction management, and ACID properties of transactions are described in Section 7.2. In Section 7.3, the objectives of distributed transaction management are explained, and a model for transaction management in distributed systems is described in Section 7.4. The classification of transactions is introduced in Section 7.5.

7.1 Basic Concepts of Transaction Management

Transaction management in a database system deals with the problems of maintaining consistency of data in spite of system failures and concurrent accesses to data. A **transaction** consists of a set of operations that perform a single logical unit of work in a database environment. It may be an entire program, or a part of a program, or a single command (SQL command), and it may involve any number of operations on the database. If a transaction is completed successfully, then the database moves from one consistent state to another. This consistency must be ensured irrespective of whether transactions are successfully executed simultaneously or there are failures during execution. Thus, a transaction is a unit of consistency and reliability. **Reliability** refers to both the resilience of a system to various types of failures and its capability to recover from them. A resilient system continues its normal execution even when failures occur. A **recoverable DBMS** always keeps the database in a consistent state either by moving back to a previous consistent state or by moving forward to a new consistent state following various types of failures.

The basic difference between a query and a transaction is the fundamental properties of a transaction: atomicity and durability. A transaction ensures the consistency of the database irrespective of the facts that several transactions are executed concurrently and that failures may occur during their execution. A transaction can be considered to be made up of a sequence of read and write operations on the database, together with some computational steps. In that sense, a transaction may be thought of as a program with embedded database access queries. The difference between a query and a transaction is illustrated with an example in the next section.

Each transaction has to terminate. The outcome of the termination depends on the success or failure of the transaction. Once a transaction starts execution, it may terminate in one of the following possible ways:

- The transaction aborts if a failure occurred during its execution.
- The transaction commits if it was completed successfully.

A transaction may abort due to a variety of reasons. When a transaction is aborted, to maintain the consistency of the data its execution is stopped, and the already executed actions are undone by returning the database to the prior state that existed before the start of the execution. This is known as **rollback.** If a transaction is **committed,** the result will permanently be stored in the database, and this cannot be undone.

Example 7.1

Let us consider the following Employee relation:

Employee = (emp-id, emp-name, designation, salary, emp-branch, project-no)

and the SQL query

"Update employee salary by 20% for the employee whose emp-id is E001 in the Employee relation".

The SQL statement for this query is:

```
UPDATE EMPLOYEE
    SET SALARY = SALARY * 1.2
WHERE EMP-ID = "E001".
```

This query can be represented as a transaction with the name **salary_update** by using the embedded SQL notation as listed below.

```
Begin-transaction salary_update
begin
EXEC SQL
    UPDATE EMPLOYEE SET SALARY = SALARY * 1.2
    WHERE EMP-ID = "E001"
end
```

The **Begin-transaction** and **end** statements represent the beginning and end of a transaction respectively, which is treated as a program that performs a database access.

7.2 ACID Properties of Transactions

A transaction has four properties that lead to the consistency and reliability of a database management system. These are atomicity, consistency, isolation and durability.

Atomicity Atomicity refers to the fact that each transaction is a single logical unit of work in a database environment that may consist of a number of operations. Thus, either all operations of a transaction are to be completed successfully, or none of them are carried out at all. This property is known as **all-or-nothing** property of a transaction. It is the responsibility of the DBMS to maintain atomicity even when failures occur in the system. If the execution of a transaction is interrupted by any sort of failures, the DBMS will determine what actions should be performed with that transaction to recover from the failed state. To maintain atomicity during failures, either the system should complete the remaining operations of the transaction or it should undo all the operations that have already been executed by the transaction before the failure.

The execution of a transaction may be interrupted owing to a variety of reasons such as bad input data, deadlock, system crash, processor failures, power failure, or media and communication links failures (in case of distributed system). The recovery after a transaction failure is divided into two different categories based on the different types of failures that can occur in a database environment. The activity of ensuring transaction atomicity in the case of transaction aborts due to failures caused by bad input data, deadlock and other factors (except system crash and hardware failures) is called **transaction recovery**. On the other hand, the activity of preserving transaction atomicity in the case of system crash or hardware failures is called **crash recovery**.

Consistency Referring to the correctness of data, this property states that a transaction must transform the database from one consistent state to another consistent state. It is the responsibility of both the semantic data controller (integrity manager) and the concurrency control mechanisms to maintain consistency of the data. The semantic data controller can ensure consistency by enforcing all the constraints that have been specified in the database schema, such as integrity and enterprise constraints. Similarly, the responsibility of a concurrency control mechanism is to disallow transactions from reading or updating "dirty data". **Dirty data** refers to the state of the data at which the data have been updated by a transaction, but the transaction has not yet committed. (The details of distributed concurrency control mechanisms are discussed in Chapter 8). A classification of consistency has been defined [Grey et al., 1976] based on dirty data, which groups databases into four levels of consistency, which are listed in the following.

Degree 3: A transaction T1 observes degree 3 consistency if it is restricted by the following rules:

(i) *T*1 does not overwrite dirty data of other transactions.

(ii) *T*1 does not commit any write operation until it completes all its write operations, that is, until the end of the transaction.

(iii) *T*1 does not read dirty data from other transactions.

(iv) Other transactions in the system do not read any dirty data updated by *T*1 before *T*1 completes.

Degree 2: A transaction T1 observes degree 2 consistency if it is restricted by the following rules.

(i) *T*1 does not overwrite dirty data of other transactions.

(ii) *T*1 does not commit any write operation until it completes all its write operations, that is, until the end of the transaction.

(iii) *T*1 does not read dirty data from other transactions.

Degree 1: A transaction T1 observes degree 1 consistency if it is restricted by the following rules.

(i) *T*1 does not overwrite dirty data of other transactions.

(ii) *T*1 does not commit any write operation until it completes all its write operations, that is, until the end of the transaction.

Degree 0: A transaction *T*1 observes degree 0 consistency if it is restricted by the following facts.

(i) T1 does not overwrite dirty data of other transactions.

Thus, it is clear that a higher degree of consistency covers all the lower levels. These different degrees of consistency provide flexibility to application developers to define transactions that can operate at different levels.

Isolation. According to this property, transactions can execute independently in a database environment; that is, they are not dependent on each other. In other words, in a database environment, transactions are isolated from one another and the partial effect of incomplete transactions should not be visible to other transactions. This isolation property of transactions is ensured by the concurrency control mechanism of a DBMS. Consequently, no transaction can read or modify data that is being modified by another transaction. If this property is not maintained, one of the following two events as listed below can take place in the database.

(a) Lost update – This problem occurs when an apparently successful completed update operation made by one transaction (first) is overridden by another transaction (second). The lost update problem is illustrated in example 7.2.

Example 7.2

Assume that there are two concurrent transactions *T*1 and *T*2 respectively occurring in a database environment. The transaction *T*1 is withdrawing $500 from an account with balance *B*, with an initial value of $2,000, while the transaction *T*2 is depositing an amount of $600 into the same account. If these two transactions are executed serially, then the balance *B* of the account will be $2,100. Consider that both the transactions are started nearly at the same time, and both have read the same value of *B*, that is, $2,000. Owing to the concurrent access to the same data item, when the transaction *T*1 is decrementing the value of *B* to $1,500, the transaction *T*2 is incrementing the value of *B* to $2,600. Thus, update to the data item *B* made by the transaction *T*1 will be overridden by the transaction *T*2. This is known as lost update problem.

(b) Cascading aborts – This problem occurs when one transaction allows other transactions to read its uncommitted data and decides to abort; the transactions that have read or modified its uncommitted data will also abort. Consider the previous example 7.2. If transaction T1 permits transaction T2 to read its uncommitted data, then if transaction T1 aborts, transaction T2 will also abort consequently.

The type of isolation that is ensured by not allowing uncommitted data to be seen by other transactions is called **cursor stability.** It is obvious that higher levels of consistency provide more isolation among transactions. If different degrees of consistency are considered from the point of view of the isolation property, then degree 0 prevents lost update problem, degree 2 prevents cascading aborts and degree 3 provides the full isolation by restricting both the above problems.

A set of isolation levels has been defined by ANSI SQL2 [ANSI, 1992] based on the events **dirty read, non-repeatable or fuzzy read and phantom read,** when full isolation is not maintained in a database environment. These are **read uncommitted, read committed, repeatable read** and **anomaly serializable. Dirty read** refers to the situation when a transaction reads the dirty data of another transaction. **Fuzzy read** or **non-repeatable read** occurs when one transaction reads the value of a data item that is modified or deleted by another transaction. If the first transaction attempts to re-read the data item, either that data item may not be found or it may read a different value. Similarly, it is possible that when a transaction is performing a search operation depending on a selection predicate, another transaction may be inserting new tuples into the database satisfying that predicate. This is known as **phantom read.** The different isolation levels based on these events are listed below.

- **Read uncommitted** – At this isolation level, all the three above events may occur.
- **Read committed** – At this isolation level, non-repeatable read and phantom read are possible, but dirty read cannot occur.
- **Repeatable read** – Only phantom read is possible at this isolation level.
- **Anomaly serializable** – At this isolation level, none of the above events – dirty read, phantom read and fuzzy read – is possible.

Example 7.3

Assume that there are two concurrent transactions *T*1 and *T*2 respectively occurring in a database environment. The transaction *T*1 is depositing $500 to an account *X* with an initial value $3,000, while the transaction *T*2 is withdrawing an amount $1,000 from the same account *X*. Further assume that the transaction *T*1 is executed first and allows the transaction *T*2 to read the uncommitted value $3,500 for *X* and then aborts. This problem is known as dirty read problem, which is illustrated in the following:

T1: write(X)

T2: read(X)

T1: abort

Thus, the transaction *T*2 reads the dirty data of the transaction *T*1.

Example 7.4

In a DBMS, consider a situation when one transaction *T*1 reads the value of a data item *X*, while another transaction *T*2 updates the same data item. Now, if the transaction *T*1 reads the same data item *X* again, it will read some different value, because the data item is already manipulated by the transaction *T*2. This problem is known as non-repeatable or fuzzy read problem, which is illustrated in the following:

T1: read(X)

T2: write(X)

T2: end transaction

T1: read(X)

In this case, transaction *T*1 has read two different values for *X*.

Example 7.5

The phantom read problem is illustrated below.

T1: read range [X – Y]

T2: insert Z, X < Z < Y

T2: end transaction

T1: read range [X – Y]

In this case, when transaction *T*1 performs an operation depending on a criterion, transaction *T*2 inserts a new tuple *Z* satisfying that criterion. Now, if the transaction *T*1 performs the same operation again, the result will be different. Hence, *Z* is a "phantom" data item.

Durability. Durability refers to the fact that the effects of a successfully completed (committed) transaction are permanently recorded in the database and will not be affected by a subsequent failure. This property ensures that once a transaction commits, the changes are durable and cannot be erased from the database. It is the responsibility of the recovery subsystem of a DBMS to maintain durability property of transactions. The recovery manager determines how the database is to be recovered to a consistent state (database recovery) in which all the committed actions are reflected.

7.3 Objectives of Distributed Transaction Management

This section introduces the objectives of a transaction management subsystem of a distributed DBMS. The responsibilities of the transaction manager in a distributed DBMS is same as those of a corresponding subsystem in a centralized database system, but its task is complicated owing to fragmentation, replication and the overall distributed nature of the database. The consistency of the database should not be violated because of transaction executions. Therefore, the primary objective of a transaction manger is to execute transactions in an efficient way in order to preserve the ACID properties of transactions (local as well as global transactions) irrespective of concurrent execution of transactions and system failures. At the same time, the efficiency of transaction executions should be improved to meet the performance criteria. In addition, a good transaction management scheme must consider the following issues.

(i) CPU and main memory utilization should be improved Most of the typical database applications spend much of their time waiting for I/O operations rather than on computations. In a large system, the concurrent execution of these I/O bound applications can turn into a bottleneck in main memory or in CPU time utilization. To alleviate this problem, that is, to improve CPU and main memory utilization, a transaction manager should adopt specialized techniques to deal with specific database applications. This aspect is common to both centralized and distributed DBMSs.

(ii) Response time should be minimized To improve the performance of transaction executions, the response time of each individual transaction must be considered and should be minimized. In the case of distributed applications, it is very difficult to achieve an acceptable response time owing to the additional time required to communicate between different sites.

(iii) Availability should be maximized Although the availability in a distributed system is better than that in a centralized system, it must be maximized for transaction recovery and

concurrency control in distributed databases. The algorithms implemented by the distributed transaction manager must not block the execution of those transactions that do not strictly need to access a site that is not operational.

(iv) Communication cost should be minimized In a distributed system an additional communication cost is incurred for each distributed or global application, because a number of message transfers are required between sites to control the execution of a global application. These messages are not only used to transfer data, but are required to control the execution of the application. Preventative measures should be adopted by the transaction manager to minimize the communication cost.

7.4 A Model for Transaction Management in a Distributed System

This section represents an abstract model for transaction management in a distributed database environment. In a distributed system, transactions are classified into two different categories: **local transactions** and **global transactions (or distributed transactions).** If the data requirement of a transaction can be fulfilled from the local site, it is called a **local transaction.** Local transactions access data only from local sites. On the other hand, **global transactions** access data from remote sites or multiple sites. It is obvious that the transaction management in a distributed DBMS is more complicated than in a centralized DBMS, as the distributed DBMS must ensure the atomicity of the global transaction as well as of each component subtransaction executed at the local sites. This complicated task is performed by four high-level interconnected modules of the DBMS. These are transaction manager, concurrency control manager, recovery manager and buffer manager. The **transaction manager** coordinates transactions on behalf of application programs by communicating with the scheduler and implements a particular strategy for concurrency control. The responsibility of the **concurrency control manager** is to maximize concurrency, without allowing concurrently executing transactions to interfere with one another, and thereby maintain the consistency of the database as well as the isolation property of the transactions. The **recovery manager** preserves the database in a consistent state in case of failures. The **buffer manager** manages the efficient transfer of data between disk storage and main memory.

In a distributed DBMS, all these modules exist in each local DBMS. In addition, a global transaction manager or transaction coordinator is required at each site to control the execution of global transactions as well as of local transactions initiated at that site. Therefore, an abstract model of transaction management at each site of a distributed system consists of two different submodules: the transaction manager and the transaction coordinator (which are described below).

Transaction manager – The transaction manager at each site manages the execution of the transactions that access data stored at that local site. Each such transaction may be a local transaction or part of a global transaction. The structure of the transaction manager is similar to that of its counterpart in a centralized system, and it is responsible for the following:

- Maintaining a log for recovery purposes in case of failures.
- Implementing an appropriate concurrency control mechanism to coordinate the concurrent execution of transactions executing at that local site.

Transaction coordinator – The transaction coordinator at each site in a distributed system coordinates the execution of both local and global transactions initiated at that site. This component or module is not required in centralized DBMSs, as there is only one site in a centralized system. The transaction coordinator at each site performs the following tasks to coordinate the execution of transactions initiated at that local site.

- » It starts the execution of the transactions initiated at that site.
- » It breaks the transactions into a number of subtransactions and distributes these subtransactions to the appropriate sites for execution.
- » It coordinates the termination of the transactions, which may result in the transactions being committed at all sites or aborted at all sites.

The structure of this model is depicted in figure 7.1.

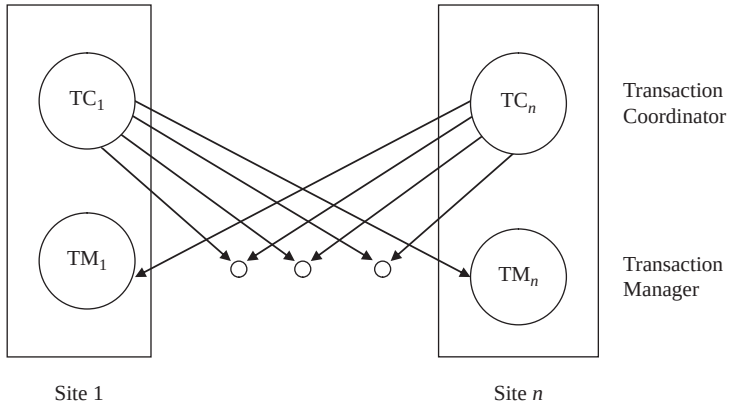


Fig. 7.1 A Model for Transaction Management at Each Site in a DDBMS

A transaction is considered as a part of an application. When a user requests for the execution of an application (may be local or global), the application issues a **begin_transaction** primitive. All actions that are performed by the application after that are to be considered part of the same transaction until a **commit** or an **abort** primitive is issued. In some systems, all these primitives are implicitly associated with the application; thus, it is not necessary to define all these primitives explicitly. To execute a global application generated at site *S*1, the transaction coordinator of site *S*1 initiates the transaction and breaks the transaction into a number of subtransactions. It then involves multiple sites by consulting the global system catalog for parallel execution of these subtransactions at different sites. When these subtransactions are distributed over multiple local sites and perform some tasks on behalf of the application, they are called **agents.** Agents reside at multiple sites and communicate with each other via message passing. The transaction manager at a local site maintains the log when a local transaction or part of a global transaction is executed at that local site. It also controls the concurrent execution of transactions executing at that site. The results retrieved from the parallel execution of these subtransactions at multiple sites are integrated by the transaction coordinator of site *S*1, and finally the transaction is terminated.

7.5 Classification of Transactions

At this point it is useful to discuss the classification of transactions. These classifications have been defined based on various criteria such as the lifetime of the transaction, organization of read and write operations within the transaction and the structure of the transaction. Depending on their lifetime or duration, transactions can be classified into two broad categories: **short-duration transactions** and **long-duration transactions.** A **short-duration transaction** (also called **online transaction**) requires a very short execution/response time and accesses a relatively small portion of the database. On the other hand, a **long-duration transaction** (also called a **batch transaction**) requires a longer execution/response time and generally accesses a larger portion of the database. Most of the statistical applications – report generation, complex queries, image processing – are characterized by long-duration transactions, whereas most of the current normal database applications including railway reservation, banking system, project management require the short-duration category transactions.

Another way to divide transactions into different classes is based on the organization of the read and the write operations within the transaction. In some transactions, read and write operations are organized in a mixed manner; they do not appear in any specific order. These are called **general transactions.** Some transactions restrict the organization of read and write operations in such a way that all the read operations within the transaction are performed before the execution of any write operation. These transactions are called **two-step transactions.** Similarly, some transactions may restrict that a data item has to be read before it can be written (updated) within the transaction operation. These transactions are known as **restricted (or read-before-write) transactions.** If a transaction is characterized by features of both two-step and restricted transactions, then it is called a **restricted two-step transaction.**

Transactions can also be classified into three broad categories based on the structure of transactions, namely, **flat transactions, nested transactions** and **workflow models,** which are discussed in the following:

(a) A **flat transaction** has a single start point (begin_transaction) and a single end point or termination point (end_transaction). Most of the typical transaction management database applications are characterized by flat transactions. Flat transactions are relatively simple and perform well in short activities, but they are not well-suited for longer activities.

(b) A **nested transaction** includes other transactions within its own start point and end point; thus, several transactions, called subtransactions, may be nested into one transaction. A nested transaction is further classified into two categories: closed nested transaction and open nested transaction. In a **closed nested transaction** commit occurs in a bottom-up fashion, that is, parent transactions are committed after the commit of subtransactions or child transactions that are nested within it, although subtransactions are started after the parent transaction. On the other hand, **open nested** transactions provide flexibility by allowing the commit of the parent transaction before the commit of subtransactions. An example of an open nested transaction is "Saga" [Garcia-Molina and Salem, 1987], which is a sequence of transactions that can be interleaved with other transactions. In "Saga", only two levels of nesting are permitted, and it does not support full atomicity at the outer level. Thus, flexibility is provided with respect to ensuring the ACID properties of transactions.

The primary advantage of a nested transaction is that it provides a higher level of concurrency among transactions, as a number of subtransactions may be nested or embedded into one transaction. The nested transaction is also beneficial for recovery, as it is possible

to recover independently each subtransaction from failures, thus, leading to less expensive recovery control mechanisms. In the case of nested transactions it is possible to restart only the operations of a particular subtransaction instead of restarting the entire transaction when failures occur in the system. Finally, the creation of a new transaction is not cumbersome as this can be done by simply inserting the newly created transaction into an existing transaction as a subtransaction.

(c) The name of the third category of transactions based on transaction structure is workflow. A **workflow** is defined as a collection of tasks organized to carry out some business process. Three different types of workflow [Georgakopoulos, et al., 1995] are identified, which are listed in the following:

- **Human-oriented workflows** – These workflows provide support for collaboration and coordination among humans, but humans themselves are ultimately responsible for the consistency of the operations.
- **System-oriented workflows** – These workflows provide support for specialized tasks that can be executed by a computer such as concurrency control and recovery, automatic task execution, and notification.
- **Transactional workflows** – This type of workflows has the characteristics of both human-oriented and system-oriented workflows. These workflows provide supports for coordinating execution of multiple tasks and selective use of transactional properties for individual tasks or entire workflows.

There are two general issues involved in workflows: specification of workflows and execution of workflows. The key issues involved in specifying a workflow are as follows:

- **Task specification –** The execution structure of each task is defined by providing a set of externally observable execution states and a set of transitions between these states.
- **Task coordination requirements –** These are usually represented as inter-task execution dependencies and data flow dependencies, and the termination conditions of the workflow.
- **Execution requirements –** These restrict the execution of the workflow to fulfill application-specific correctness criteria. These include failure/execution atomicity requirements, workflow concurrency control, and recovery requirements.

Example 7.6

Let us consider the transactions *T*1, *T*2, *T*3 and *T*4, where *T*1 represents a general transaction, *T*2 represents a two-step transaction, *T*3 represents a restricted transaction and *T*4 represents a two-step restricted transaction.

T1: {read (a1), read (b1), write(a1), read(b2), write(b2), write(b1), commit}

T2: {read (a1), read (b1), read(b2), write(a1), write(b2), write(b1), commit}

T3: {read (a1), read (b1), read(b2), write(a1), read(c1), write(b2), read(c2), write(b1), write(c1), write(c2), commit}

T4: {read (a1), read (b1), read(b2), read(c1), read(c2), write(a1), write(b2), write(b1), write(c1), write(c2), commit}

Example 7.7

An example of a nested transaction is illustrated below.

Begin_transaction Banking_services
Begin
  Begin_transaction Accounts_maintenance
  ……
  end. {Accounts_maintenance}
  Begin_transaction Loan_maintenance
  ……
  end. {Loan_maintenance}
  Begin_transaction Fixed_deposit
  ……
  end. {Fixed_deposit}
  ……
end. {Banking_services}

Example 7.8

Let us consider the example of patient's health checkup transaction. The entire health checkup activity consists of the following tasks and involves the following data.

- Task1 (T1): A patient requests for health checkup registration and Patient database is accessed.
- Task2 (T2 & T3): Some medical tests have been performed, and the patient's health information is stored in the Health database. In this case, several medical tests can be performed simultaneously. For simplicity, here it is assumed that two medical tests have been performed.
- Task3 (T4): The doctor's advice is taken and hence Doctor database is accessed.
- Task4 (T5): A bill is generated and the billing information is recorded into the database.

In this patient's health checkup transaction, there is a serial dependency of *T*2, *T*3 on *T*1, and *T*4 on *T*2, *T*3. Hence, *T*2 and *T*3 can be performed in parallel and *T*5 waits until their completion. The corresponding workflow is illustrated in figure 7.2.

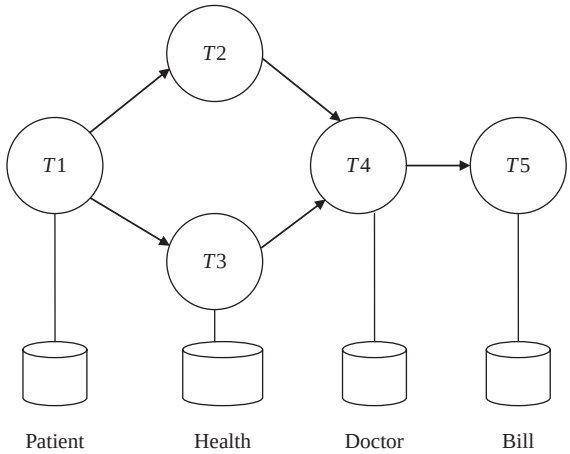


Fig. 7.2 An Example of a Workflow

CHAPTER SUMMARY

- A transaction consists of a set of operations that represents a single logical unit of work in a database environment and moves the database from one consistent state to another. ACID property of a transaction represents atomicity, consistency, isolation and durability properties.
- Atomicity property stipulates that either all the operations of a transaction should be completed successfully or none of them are carried out at all.
- Consistency property ensures that a transaction transforms the database from one consistent state to another consistent state.
- Isolation property indicates that transactions are executed independently of one another in a database environment; that is, they are isolated from one another.
- Durability refers to the fact that the effects of a successfully completed (committed) transaction are permanently recorded in the database and are not affected by a subsequent failure.
- The major components or subsystems of a distributed transaction management system are the transaction manager and the transaction coordinator. The transaction manager at each site manages the execution of those transactions that access data stored at that local site. The transaction coordinator at each site in a distributed system coordinates the execution of both local and global transactions initiated at that site.
- Depending on their lifetime or duration, transactions can be classified into two broad categories: **short-duration transactions** and **long-duration transactions.** Based on the organization of read and write operations within them, transactions are classified as general, restricted and two-step restricted. Transactions can also be classified into three broad categories based on the structure of transactions: **flat transactions, nested transactions** and **workflow models.**

EXERCISES

Multiple Choice Questions

(i) Changes made in a database are called
   a. Transaction
   b. Commit
   c. Replication
   d. Fragmentation.

(ii) "All-or-nothing" property of a transaction is called
   a. Consistency
   b. Durability
   c. Isolation
   d. Atomicity.

(iii) Which of the following statements is correct?
   a. Transaction recovery is the activity of ensuring transaction atomicity in the case of transaction aborts owing to hardware failure
   b. Crash recovery is the activity of ensuring transaction atomicity in the case of transaction aborts owing to bad input or deadlock
   c. Crash recovery is the activity of ensuring transaction atomicity in the case of transaction aborts owing to hardware failure or system crash
   d. None of the above.

(iv) Dirty data refers to the state in which data has been updated by a transaction and
   a. The transaction has committed
   b. The transaction has aborted
   c. The transaction has restarted
   d. The transaction has not yet committed.

(v) From the viewpoint of the isolation property
   a. Degree 0 provides full isolation
   b. Degree 1 prevents lost update problem
   c. Degree 2 prevents cascading aborts
   d. Degree 3 prevents dirty read.

(vi) Which of the following is not an objective of distributed transaction management?
   a. Maximizing availability
   b. Minimizing reliability

c. Reducing communication cost
d. Minimizing response time.

(vii) Which of the following components preserves the database in a consistent state in case of failures?
a. Transaction manager
b. Buffer manager
c. Concurrency control manager
d. Recovery manager.

(viii) Which of the following is a responsibility of the transaction manager?
a. Initiation of transactions
b. Partitioning of transactions
c. Maintaining log records
d. Termination of transactions.

(ix) Which of the following is a responsibility of transaction coordinator?
a. Maintaining log records
b. Breaking the transactions into subtransactions
c. Implementing an appropriate concurrency control mechanism
d. All of these.

(x) Transactions are classified based on
a. Lifetime of transactions
b. Read and write operations within transactions
c. Structure of transactions
d. All of these
e. None of these.

(xi) A short-duration transaction
a. Requires a very short execution/response time
b. Accesses a small portion of the database
c. Both a and b
d. None of the above.

(xii) A long-duration transaction
a. Requires a very long execution/response time
b. Accesses a small portion of the database
c. Accesses a large portion of the database
d. Both a and c.

(xiii) In two-step transactions
a. All write operations are executed before the execution of any read operation
b. All read operations are executed before the execution of any write operation
c. Some of the read operations must be executed before any write operation
d. Some of the write operations must be executed before any read operation.

(xiv) In a restricted transaction
a. All write operations are executed before the execution of any read operation
b. All read operations are executed before execution of any write operation
c. A data item has to be read before it can be written within a transaction
d. None of these.

(xv) A flat transaction has
a. A single start point and a single end point
b. A number of start and end points
c. A single start point, but multiple end points
d. None of these.

(xvi) Which of the following is a key issue in transaction workflows?
a. Task specification
b. Task coordination requirements
c. Execution requirements
d. All of these.

(xvii) Which of the following is not a category of transaction workflows?
a. Human-oriented workflows
b. System-oriented workflows
c. Transactional workflows
d. None of these.

(xviii) Saga is an example of a
a. Flat transaction
b. Open nested transaction
c. Closed nested transaction
d. Transaction workflows.

Review Questions

1. Define a transaction. Discuss the ACID properties of a transaction.
2. Explain with an example the use of termination conditions in transactions.
3. Differentiate between transaction recovery and crash recovery.

4. Describe the different degrees of consistency.
5. What is cursor stability? Briefly explain the different isolation levels.
6. Write down the responsibilities of transaction manager and transaction coordinator.
7. Briefly explain the objectives of distributed transaction management.
8. Differentiate between short-duration and long-duration transactions.
9. Write down the features of two-step and restricted transactions with examples.
10. Compare and contrast flat transactions and nested transactions.
11. Explain nested transaction with an example.
12. Define a workflow. Discuss the classification of workflows. Give an example for a workflow transaction.
13. Design the workflows for example 7.7.
14. Design the workflows for project management system.
15. Design the workflows for courseware management system.

8 Distributed Concurrency Control

This chapter introduces the different concurrency control techniques to handle concurrent accesses of data items in a distributed database environment. In this chapter, the concurrency control anomalies and distributed serializability are discussed in detail. To maintain distributed serializability, both pessimistic as well as optimistic concurrency control mechanisms are employed, which are also briefly discussed in this chapter.

The organization of this chapter is as follows. The objectives of distributed concurrency control are represented in Section 8.1. Section 8.2 introduces different concurrency control anomalies. Distributed serializability is explained in Section 8.3. The classification of distributed concurrency control techniques is discussed in Section 8.4. Section 8.5 describes different locking-based concurrency control techniques while Section 8.6 describes different time-based concurrency control techniques. The optimistic concurrency control techniques are represented in Section 8.7.

8.1 Objectives of Distributed Concurrency Control

Concurrency control is the activity of coordinating concurrent accesses to a database in a multi-user system. Concurrency control allows users to access a database in a multi-programmed fashion while preserving the consistency of the data. The main technical difficulty in achieving this goal is the necessity to prevent database updates performed by one user from interfering with database retrievals and updates performed by other users. The concurrency control problem is exacerbated in a distributed DBMS (DDBMS), because in a distributed system users may access data stored at many different sites, and a concurrency control mechanism at one site cannot be aware of transactions at other sites instantaneously. The replication of data items in a distributed database adds extra complexity to the concurrency control mechanism. In this context, it is necessary to discuss the objectives of concurrency control mechanism in a distributed database environment. Like in centralized DBMSs, **the primary goal** of the concurrency control mechanism in a DDBMS is to ensure that the consistency of the data items is preserved and each atomic action will be completed in a finite time if the system has not failed. In addition, a good concurrency control mechanism for a DDBMS has the following objectives:

- It must be resilient to site and communication failures.
- It should permit parallel execution of transactions to achieve maximum concurrency, thus satisfying performance requirements.
- Its computational methods and storage mechanisms should be modest to minimize overhead.

- It should perform satisfactorily in a network environment taking into consideration that it involves significant communication delay.
- It must impose few constraints on the structure of atomic actions of transactions.

8.2 Concurrency Control Anomalies

The goal of concurrency control is to prevent interference among users who are simultaneously accessing a database. This section introduces the different anomalies that can arise owing to concurrent accesses of data in a multi-user centralized database environment as well as in a distributed database environment. These are **lost update anomaly, uncommitted dependency or dirty read anomaly, inconsistent analysis anomaly, non-repeatable or fuzzy read anomaly** and **phantom read anomaly**. In addition, in a DDBMS an extra problem can arise, namely, **multiple-copy consistency problem**, as data items may be replicated in distributed databases. [Some of these anomalies were already introduced in **Chapter 7**.]

(i) Lost update anomaly – This anomaly can occur when an apparently successful completed update operation made by one user (transaction) is overridden by another user (transaction). This anomaly is illustrated in example 8.1.

Example 8.1

Consider the example of an online electronic funds transfer system accessed via remote automated teller machines (ATMs). Assume that one customer ($T1$) is withdrawing \$500 from an account with balance B, initially \$2,000, while another customer ($T2$) is depositing an amount \$600 into the same account via ATM from a different place. If these two tasks are executed serially, then the balance B will be \$2,100. Further assume that both the tasks are started nearly at the same time and both read the same value \$2,000. Due to concurrent access to the same data item, when customer $T1$ is decrementing the value of B to \$1,500, customer $T2$ is incrementing the value of B to \$2,100. Thus, update to the data item B made by customer $T1$ will be overridden by the customer $T2$. Here, the consistency of the data is violated owing to this anomaly, known as lost update anomaly.

(ii) Uncommitted dependency or dirty read anomaly – This problem occurs when one transaction allows other transactions to read its data before it has committed and then decides to abort. The dirty read anomaly is explained in example 8.2.

Example 8.2

Assume that two transactions $T1$ and $T2$ are concurrently accessing an Employee database. The transaction $T1$ is updating the salary of an employee by 10 percent whose basic salary is sal_x, initially \$4,000, while transaction $T2$ is calculating tax deduction for the same employee depending on his basic salary, sal_x. Transaction $T1$ changes the value of sal_x to \$4,400, allows transaction $T2$ to read this value before transaction $T1$ has committed, and then decides to abort. Since transaction $T1$ is aborted, sal_x should be restored to its original value \$4,000. However, transaction $T2$ has read the new value of sal_x and calculates the tax deduction depending on this new value \$4,400 instead of \$4,000. The value of sal_x read by the transaction $T2$ is called **dirty data**, which leads to uncommitted dependency or **dirty read anomaly**. This problem can be avoided by preventing transaction $T2$ from reading the new value of sal_x updated by transaction $T1$, until the decision has been made to either commit or abort transaction $T1$.

(iii) Inconsistent analysis anomaly – The problem of inconsistent analysis occurs when a transaction reads several values from the database but a second transaction updates some of them during the execution of the first.

Example 8.3

Let us consider the example of a banking system in which two transactions $T3$ and $T4$ are accessing the database concurrently. The transaction $T3$ is calculating the average balance of all customers of a particular branch while the transaction $T4$ is depositing \$1,000 into one customer account with balance bal_x at the same time. Hence, the average balance calculated by the transaction $T3$ is incorrect, because the transaction $T4$ incremented and updated the balance of one customer during the execution of the transaction $T3$. This anomaly, known as inconsistent analysis anomaly, can be avoided by preventing the transaction $T4$ from reading and updating the value of bal_x until the transaction $T3$ completes its execution.

(iv) Non-repeatable or fuzzy read anomaly – **Fuzzy read** or **non-repeatable read anomaly** occurs when one transaction reads the value of a data item that is subsequently modified or deleted by another transaction. If the first transaction attempts to re-read the data item either that data item may not be found or it may read a different value.

Example 8.4

Assume that in the employee database, one transaction $T1$ reads the salary of an employee while another transaction $T2$ is updating the salary of the same employee (or the same employee record is deleted from the employee database by the transaction $T2$) concurrently. Now, if the transaction $T1$ attempts to re-read the value of the salary of the same employee, it will read a different value (or that record will be not found) since employee database is updated by the transaction $T2$. Thus, two read operations within the same transaction $T1$ return different values. This anomaly occurs due to concurrent access of the same data item, known as fuzzy read anomaly.

(v) Phantom read anomaly – This anomaly occurs when a transaction performing some operation on the database based on a selection predicate, another transaction inserts new tuples satisfying that predicate into the same database. This is known as **phantom read**. For example, assume that in the employee database, one transaction $T1$ retrieves all employees belonging to the R&D department while another transaction $T2$ inserts new employees into that department. Hence, if the transaction $T1$ re-executes the same operation, then the retrieved data set will contain additional (phantom) tuples that has been inserted by the transaction $T2$ in the meantime.

Besides all the above anomalies, **multiple-copy consistency problem** can arise in a distributed database system as a result of data distribution. This problem occurs when data items are replicated and stored at different locations in a DDBMS. To maintain the consistency of the global database, when a replicated data item is updated at one site, all other copies of the same data item must be updated. If any copy is not updated, the database becomes inconsistent. The updates of the replicated data items are carried out either synchronously or asynchronously to preserve the consistency of the data.

8.3 Distributed Serializability

A transaction consists of a sequence of read and write operations together with some computation steps that represents a single logical unit of work in a database environment. All the operations of a transaction are units of atomicity; that is, either all the operations should be completed successfully

or none of them are carried out at all. Ozsu has defined a formal notation for the transaction concept. According to this formalization, a transaction *Ti* is a partial ordering of its operations and the termination condition. The partial order $P = \{\Sigma, \alpha\}$ defines an ordering among the elements of Σ (called the domain), where Σ consists of the read and write operations and the termination condition (abort, commit) of the transaction *Ti*, and α indicates the execution order of these operations within *Ti*.

The execution sequence or execution ordering of operations of a transaction is called the **schedule**. Let E denote an execution sequence of transactions $T1, T2, \ldots, Tn,$. E is a **serial execution** or **serial schedule** if no transactions execute concurrently in E; that is, each transaction is executed to completion before the next one begins. Every serial execution or serial schedule is deemed to be correct, because the properties of transactions imply that a serial execution terminates properly and preserves database consistency. On the other hand, an execution is said to be a **non-serial execution** or **non-serial schedule** if the operations from a set of concurrent transactions execute simultaneously. A non-serial schedule (or execution) is **serializable** if it is computationally equivalent to a serial schedule, that is, if it produces the same result and has the same effect on the database as some serial execution. To prevent data inconsistency, it is essential to guarantee serializability of concurrent transactions.

In serializability, the ordering of read and write operations are important. In this context, it is necessary to introduce the concept of conflicting transactions. If two transactions perform read or write operations on different data items, then they are not conflicting, and hence the execution order is not important. If two transactions perform read operation on the same data item, then they are non-conflicting. Two transactions are said to be **conflicting** if they access the same data item concurrently, and at least one transaction performs a write operation. Thus, concurrent read-read operation by two transactions is non-conflicting, but concurrent read-write, write-read and write-write operations by two transactions are conflicting. The execution order is important if there is a conflicting operation caused by two transactions.

If concurrent accesses of data items are allowed in a database environment, then the schedule may be non-serial, which indicates that the consistency of data may be violated. To allow maximum concurrency and to preserve the consistency of data, it is sometimes possible to generate a schedule from a non-serial schedule that is equivalent to a serial schedule or serial execution order by swapping the order of the non-conflicting operations. If schedule *S*1 is derived from a non-serial schedule *S*2 by swapping the execution order of non-conflicting operations in *S*2, and if *S*1 produces the same output as that of a serial schedule *S*3 with the same set of transactions, then *S*1 is called a **serializable** schedule. This type of serializability is known as **conflict serializability** and Schedule *S*1 is said to be conflict equivalent to schedule S2. A conflict serializable schedule orders any conflicting operations in the same way as a serial execution. A directed precedence graph or a serialization graph can be produced to test for conflict serializability.

Serializability theory for centralized databases can be extended in a straightforward manner for distributed non-replicated databases. In a distributed system, each local site maintains a schedule or execution order of transactions or subtransactions (part of a global transaction) running at that site, called **local schedule**. The **global schedule** or **global execution order** is the union of all local schedules in a non-replicated distributed database system. Hence, if each local site maintains serializability, then the global schedule also becomes serializable as the local serialization orders are identical. Replication of data items in a distributed system adds extra complexity in maintaining **global serializability** or **distributed serializability**. In this case also, it is possible that local schedules are serializable, but mutual consistency of the replicated data items may not be preserved owing to the conflicting operations on the same data item at multiple sites. Thus, a **distributed schedule** or **global schedule** (the union of all local schedules) is said to be **distributed serializable** if the execution order at each local site is serializable and the local serialization orders are identical. This requires that all subtransactions appear in the same order in the equivalent serial schedule at all sites.

Formally, distributed serializability can be defined as follows. Consider *S* is the union of all local serializable schedules *S*1, *S*2,…, S*n* respectively in a distributed system. Now, the global schedule *S* is said to be **distributed serializable**, if for each pair of conflicting operations *Oi* and *Oj* from distinct transactions *Ti* and *Tj* respectively from different sites, O*i* precedes O*j* in the total ordering *S*, and if and only if *Ti* precedes *Tj* in all of the local schedules where they appear together. To attain serializability, a DDBMS must incorporate synchronization techniques that control the relative ordering of conflicting operations.

Example 8.5

Let us consider a simple transaction *Ti* that consists of the following operations:

```
Transaction Ti:   Read(a);
                  Read(b);
                  a := a + b;
                  write(a);
                  commit;
```

The partial ordering of the above transaction *Ti* can be formally represented as $P = \{\Sigma, \alpha\}$,
Where $\Sigma = \{R(a), R(b), W(a), C\}$ and

α **= {(R(a), W(a)), (R(b), W(a)), (W(a), (C)), (R(a), (C)), (R(b), (C))}.**

Here, the ordering relation specifies the relative ordering of all operations with respect to the termination condition. The partial ordering of a transaction facilitates to derive the corresponding directed acyclic graph (DAG) for the transaction. The DAG of the above transaction Ti is illustrated in figure 8.1.

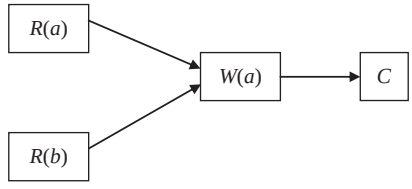


Fig. 8.1 Directed Acyclic Graph Representation of Transaction Ti

8.4 Classification of Concurrency Control Techniques

Concurrency control in a distributed system involves the activity of controlling the relative order of conflicting operations and thereby ensuring database consistency. Currently, a number of concurrency control techniques are being developed for distributed database systems, and these techniques can be classified into different categories based on different issues such as data distribution and network topology. The most obvious classification is done based on the synchronization approaches that are used in concurrency control techniques to ensure distributed serializability. Thus, concurrency control techniques are divided into two broad categories based on the synchronization techniques: **pessimistic concurrency control mechanisms** and **optimistic concurrency control mechanisms**.

Pessimistic algorithms synchronize the concurrent execution of transactions early in their execution life cycle, whereas optimistic algorithms delay the synchronization of transactions until transactions are near to their completion. Optimistic concurrency control algorithms are based

on the assumption that a conflict is rare and it is more efficient to allow transactions to proceed without delays. When a transaction wishes to commit, a check is performed to determine whether a conflict has occurred or not. Therefore, optimistic concurrency control algorithms have the desirable properties of being non-blocking and deadlock-free. In the case of pessimistic concurrency control algorithms, the synchronization of transactions is achieved earlier before the starting of transaction executions. The main drawback of pessimistic concurrency control algorithms is that they are neither deadlock-free nor non-blocking algorithms. The pessimistic concurrency control algorithms are further classified into **locking-based algorithms, timestamp-based algorithms** and **hybrid algorithms**. Similarly, the optimistic concurrency control algorithms are also classified into locking-based algorithms and timestamp-based algorithms. The complete classification of distributed concurrency control techniques is illustrated in figure 8.2.

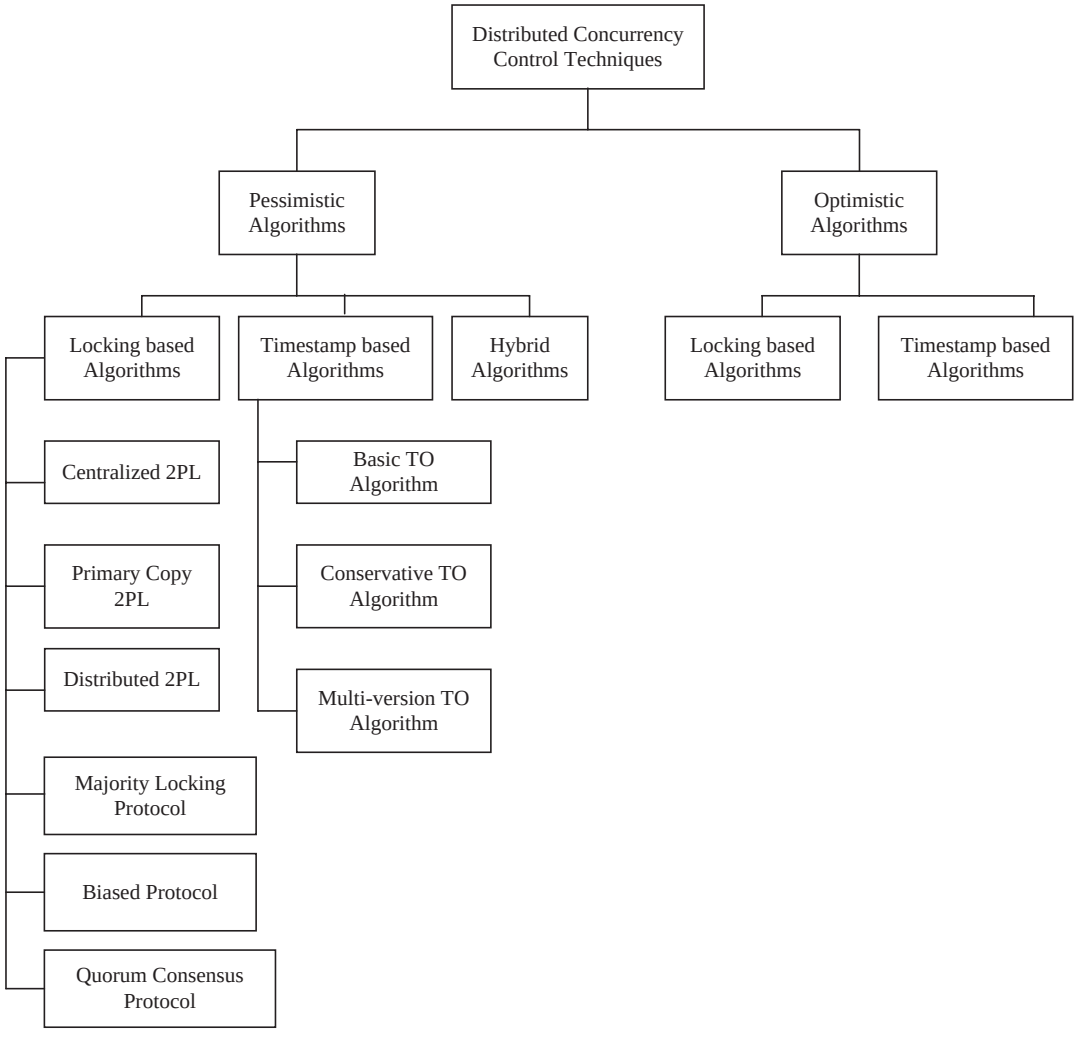


Fig. 8.2 Classification of Distributed Concurrency Control Techniques

In locking-based concurrency control algorithms, the synchronization of transactions is achieved by employing physical or logical locks on some portion of the database or on the entire database. In the case of timestamp-based concurrency control algorithms, the synchronization of transactions is achieved by assigning timestamps both to the transactions, and to the data items that are stored in the database. To allow maximum concurrency and to improve efficiency, some locking-based concurrency control algorithms also involve timestamp ordering; these are called **hybrid concurrency control algorithms**.

8.5 Locking-based Concurrency Control Protocols

This section introduces the details of locking-based concurrency control algorithms. To ensure serializability of concurrent transactions locking methods are the most widely used approach. In this approach, before accessing any data item, a transaction must acquire a lock on that data item. When a transaction acquires a lock on a particular data item, the lock prevents another transaction from modifying that data item. Based on the facts that read-read operation by two different transactions is non-conflicting and the main objective of the locking-based concurrency control techniques is to synchronize the conflicting operations of conflicting transactions, there are two types of locking modes: **read lock** (also called **shared lock**) and **write lock** (also called **exclusive lock**). If a transaction obtains a read lock on a data item, it can read, but cannot update that data item. On the other hand, if a transaction obtains a write lock on a data item, it can read as well as update that data item. When a transaction obtains a read lock on a particular data item, other transactions are allowed to read that data item because read-read operation is non-conflicting. Thus, several transactions can acquire shared lock or read lock on the same data item simultaneously. When a transaction achieves an exclusive lock on a particular data item, no other transactions are allowed to read or update that data item, as read-write and write-write operations are conflicting. A transaction can acquire locks on data items of various sizes, ranging from the entire database down to a data field. The size of the data item determines the fineness or **granularity** of the lock.

In a distributed database system, the **lock manager** or **scheduler** is responsible for managing locks for different transactions that are running on that system. When any transaction requires read or write lock on data items, the transaction manager passes this request to the lock manager. It is the responsibility of the lock manager to check whether that data item is currently locked by another transaction or not. If the data item is locked by another transaction and the existing locking mode is incompatible with the lock requested by the current transaction, the lock manager does not allow the current transaction to obtain the lock; hence, the current transaction is delayed until the existing lock is released. Otherwise, the lock manager permits the current transaction to obtain the desired lock and the information is passed to the transaction manager. In addition to these rules, some systems initially allow the current transaction to acquire a read lock on a data item, if that is compatible with the existing lock, and later the lock is converted into a write lock. This is called **upgradation** of lock. The level of concurrency increases by upgradation of locking. Similarly, to allow maximum concurrency some systems permit the current transaction to acquire a write lock on a data item, and later the lock is converted into a read lock; this is called **downgradation** of lock.

The synchronization of transaction executions in DDBMSs cannot be ensured by implementing locks in transactions only. Consider the following two transactions *Ti* and *Tj*:

Ti:	Read(a);	Tj:	Read(a);
	a := a + 2;		a := a*10;
	Write(a);		Write(a);

Read(b);	Read(b);
b := b * 2;	b := b − 5;
Write(b);	Write(b);
Commit;	Commit;

In the above transactions, assume that transaction *Ti* acquires a lock on data item *a* initially. The transaction *Ti* releases the lock on data item *a* as soon as the associated database operation (read or write) is executed. However, the transaction *Ti* may acquire a lock on another data item, say *b*, after releasing the lock on data item *a*. Although it seems that maximum concurrency is allowed here and it is beneficial for the system, it permits transaction *Ti* to interfere with other transactions, say *Tj*, which results in the loss of total isolation and atomicity. Therefore, to guarantee serializability, the locking and releasing operations on data items by transactions need to be synchronized. The most well-known solution is **two-phase locking** (2PL) protocol.

The 2PL protocol simply states that no transaction should acquire a lock after it releases one of its locks. According to this protocol, the lifetime of each transaction is divided into two phases: **Growing phase** and **Shrinking phase**. In the growing phase, a transaction can obtain locks on data items and can access data items, but it cannot release any locks. On the other hand, in the shrinking phase a transaction can release locks but cannot acquire any new locks after that. Thus, the ending of growing phase of a transaction determines the beginning of the shrinking phase of that transaction. It is not necessary for each transaction to acquire all locks simultaneously and then start processing. Normally, each transaction obtains some locks initially, does some processing, and then requests for additional locks that are required. However, it never releases any lock until it has reached a stage where no more locks are required. If upgradation and downgradation of locks are allowed, then upgradation of locks can take place only in the growing phase, whereas downgradation of locks can occur in the shrinking phase.

When a transaction releases a lock on a data item, the lock manager allows another transaction waiting for that data item to obtain the lock. However, if the former transaction is aborted and rolled back, then the latter transaction is forced to be aborted and rolled back to maintain the consistency of data. Thus, the rollback of a single transaction leads to a series of rollbacks, called **cascading rollback**. One solution to the cascading rollback problem is to implement 2PL in such a way that a transaction can release locks only when it has reached the end of the transaction or when the transaction is committed. This is called **rigorous 2PL** protocol. To avoid cascading rollback, another possible variation in 2PL is **strict 2PL protocol**, in which only write locks are held until the end of the transaction. In a distributed database system, the 2PL protocol is further classified into several categories such as centralized 2PL, primary copy 2PL, distributed 2PL and majority locking depending on how the lock manager activities are managed.

8.5.1 Centralized 2PL

In the centralized 2PL method, the lock manager or scheduler is a central component, and hence a single site is responsible for managing all activities of the lock manager for the entire distributed system. Before accessing any data item at any site, appropriate locks must be obtained from the central lock manager. If a global transaction *Ti* is initiated at site *Si* of the distributed system, then the centralized 2PL for that global transaction should be implemented in the following way.

The transaction manager at the site S*i* (called transaction coordinator) partitions the global transaction into several subtransactions using the information stored in the global system catalog. Thus, the transaction coordinator is responsible for maintaining consistency of data. If the data

items are replicated, then it is also the responsibility of the transaction coordinator to ensure that all replicas of the data item are updated. Therefore, for a write operation, the transaction coordinator requests exclusive locks on all replicas of the data item that are stored at different sites. However, for a read operation, the transaction coordinator can select any one of the replicas of the data item that are available, preferably at its own site. The local transaction managers of the participating sites request and release locks from/to the centralized lock manager following the normal rules of the 2PL protocol. Participating sites are those sites that are involved in the execution of the global transaction (subtransactions). After receiving the lock request, the centralized lock manager checks whether that lock request is compatible with the existing locking status or not. If it is compatible, the lock manager sends a message to the corresponding site that the lock has been granted; otherwise, the lock request is put in a queue until it can be granted.

In some systems, a little variation of this method is followed. Here the transaction coordinator sends lock requests to the centralized lock manager on behalf of participating sites. In this case, a global update operation that involves n sites requires a minimum of $2n + 3$ messages to implement centralized 2PL method. These are 1 lock request from the transaction coordinator, 1 lock grant message from the centralized lock manager, n update messages from the transaction coordinator, n acknowledgement messages from the n participating sites and 1 unlock request from the transaction coordinator as illustrated in figure 8.3.

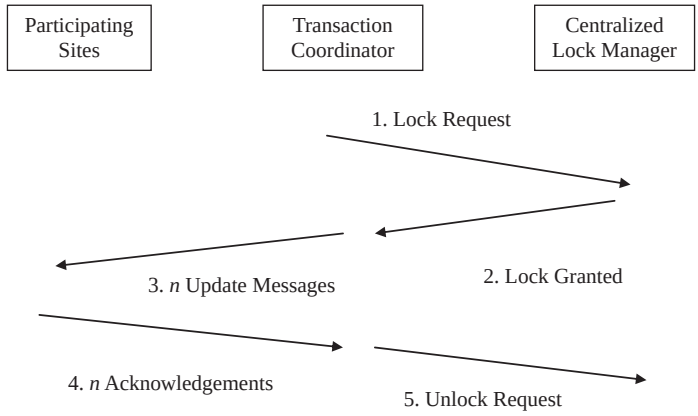


Fig. 8.3 Communicating Messages in Centralized 2PL Method

The main advantage of the centralized 2PL method in DDBMSs is that it is a straightforward extension of the 2PL protocol for centralized DBMSs; thus, it is less complicated and implementation is relatively easy. In this case, the deadlock detection can be handled easily, because the centralized lock manager maintains all the locking information. Hence, the communication cost also is relatively low. In the case of DDBMSs, the disadvantages of implementing the centralized 2PL method are system bottlenecks and lower reliability. For example, as all lock requests go to the centralized lock manager, that central site may become a bottleneck to the system. Similarly, the failure of the central site causes a major failure of the system; thus, it is less reliable. This method also hampers the local autonomy.

8.5.2 Primary Copy 2PL

Primary copy 2PL method is a straightforward extension of the centralized 2PL method. Here the responsibilities of the centralized lock manager are distributed among a number of sites

of the distributed system to overcome the disadvantages of the centralized 2PL method. Thus, in the **primary copy 2PL** method lock managers are implemented at several sites. Each lock manager is responsible for managing locks for a given set of data items. For replicated data items, one copy is chosen as the **primary copy**, and the other copies are called **slave copies**. The choice of the site as the primary site of a data item is flexible. However, it is not necessary that the primary site hold the primary copy of data items to manage the locks [Stonebraker and Neuhold, 1977].

Whenever a transaction is initiated at a site, the transaction coordinator will determine where the primary copy is located, and then it will send lock requests to the appropriate lock manager. If the transaction requires an update operation of a data item, it is necessary to exclusively lock the primary copy of that data item. Once the primary copy of a data item is updated, the change is propagated to the slave copies immediately to prevent other transactions reading the old value of the data item. However, it is not mandatory to carry out the updates on all copies of the data item as an atomic operation. The primary copy 2PL method ensures that the primary copy is always updated.

The main advantage of primary copy 2PL method is that it overcomes the bottlenecks of centralized 2PL approach and also reliability increases. This approach incurs lower communication costs because less amount of remote locking is required. The major disadvantage of this method is that it is only suitable for those systems where the data items are selectively replicated, updates are infrequent and sites do not always require the latest version of the data. In this approach, deadlock handling is more complex, as the locking information is distributed among multiple lock managers. In the primary copy 2PL method there is still a degree of centralization in the system, as the primary copy is only handled by one site (primary site). This latter disadvantage can be partially overcome by nominating backup sites to hold locking information.

8.5.3 Distributed 2PL

Distributed 2PL method implements the lock manager at each site of a distributed system. Thus, the lock manager at each site is responsible for managing locks on data items that are stored locally. If the database is not replicated, then distributed 2PL becomes the same as primary copy 2PL. If the database is replicated, then distributed 2PL implements a **read-one-write-all (ROWA)** replica control protocol. In ROWA replica control protocol, any copy of a replicated data item can be used for a read operation, but for a write operation, all copies of the replicated data item must be exclusively locked before performing the update operation.

In distributed 2PL, when a global transaction is initiated at a particular site, the transaction coordinator (the transaction manager of that site is called transaction coordinator) sends lock requests to lock managers of all participating sites. In response to the lock requests the lock manager at each participating site can send a lock granted message to the transaction coordinator. However, the transaction coordinator does not wait for the lock granted message in some implementations of the distributed 2PL method. The operations that are to be performed at a participating site are passed by the lock manager of that site to the corresponding transaction manager instead of the transaction coordinator. At the end of the operation the transaction manager at each participating site can send the corresponding message to the transaction coordinator. In an alternative approach, the transaction manager at a participating site can also pass the "end of operation" message to its own lock manager, who can then release the locks and inform the transaction coordinator. The communication between the participating sites

and the transaction coordinator when executing a global transaction using distributed 2PL is depicted in figure 8.4.

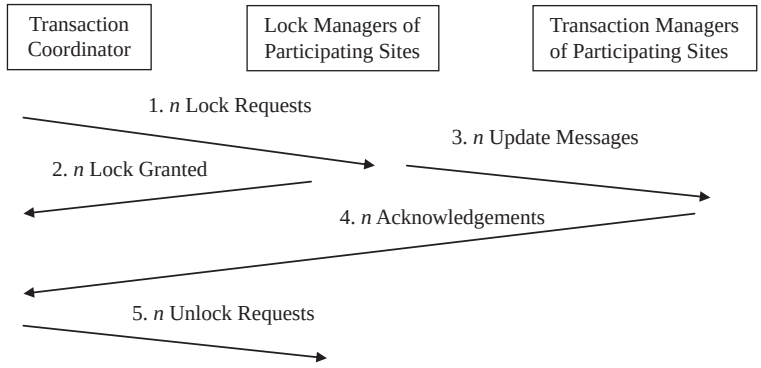


Fig. 8.4 Communicating Messages in Distributed 2PL Method

In distributed 2PL method, locks are handled in a decentralized manner; thus, it overcomes the drawbacks of centralized 2PL method. This protocol ensures better reliability than centralized 2PL and primary copy 2PL methods. However, the main disadvantage of this approach is that deadlock handling is more complex owing to the presence of multiple lock managers. Hence, the communication cost is much higher than in primary copy 2PL, as all data items must be locked before an update operation. In distributed 2PL method, a global update operation that involves n participating sites may require a minimum of $5n$ messages. These are n lock requests, n lock granted messages, n update messages, n acknowledgement messages and n unlock requests.

8.5.4 Majority Locking Protocol

Majority locking protocol was designed to overcome the disadvantage of distributed 2PL method that all replicas of a data item must be locked before an update operation. In this protocol, a lock manager is implemented at each site to manage locks for all data items stored at that site locally. The majority locking protocol in a DDBMS works in the following way.

When a transaction requires to read or write a data item that is replicated at m sites, the transaction coordinator must send a lock request to more than half of the m sites where the data item is stored. Each lock manager determines whether the lock can be granted immediately or not. If the lock request is not compatible with the existing lock status, the response is delayed until the request is granted. The transaction coordinator waits for a certain timeout period to receive lock granted messages from the lock mangers, and if the response is not received within that time period, it cancels its request and informs all sites about the cancellation of the lock request. The transaction cannot proceed until the transaction coordinator obtains locks on a majority of copies of the data item. In majority locking protocol, any number of transactions can simultaneously hold a shared lock on a majority of copies of a data item, but only one transaction can acquire an exclusive lock on a majority of copies. However, the execution of a transaction can proceed in the same manner as in distributed 2PL method after obtaining locks on a majority of copies of the data items that are required by the transaction.

This protocol is an extension of the distributed 2PL method, and hence it avoids the bottlenecks of centralized 2PL method. The main disadvantage of this protocol is that it is more complicated and hence the deadlock handling also is more complex. In this method, locking a data item that has m copies requires at least $(m+1)/2$ messages for lock requests and $(m+1)/2$ messages for unlock requests; thus, communication cost is higher than that of centralized 2PL and primary copy 2PL methods.

8.5.5 Biased Protocol

The biased protocol is another replica control approach. This approach assigns more preference to requests for shared locks on data items than to requests for exclusive locks. Like in majority locking protocol, in biased protocol a lock manager is implemented at each site to manage locks for all data items stored at that site. The implementation of the biased protocol in a distributed environment is described below.

In this protocol, when a transaction requires to read a data item that is replicated at m sites, the transaction coordinator simply sends a lock request to the lock manager of one site that contains a replica of that data item. Similarly, when a transaction needs to lock a data item in exclusive mode, the transaction coordinator sends lock requests to lock managers at all sites that contain a replica of that data item. As before, if the lock request is not compatible with the existing lock status, the response is delayed until the request is granted.

The main advantage of the biased protocol is that it imposes less overhead on read operations than the majority locking protocol. This scheme is beneficial for common transactions where the frequency of read operations is much higher than the frequency of write operations. However, this approach shares the additional overhead on write operations as in distributed 2PL protocol. In this case, the deadlock handling is also complicated as in majority locking protocol.

8.5.6 Quorum Consensus Protocol

The quorum consensus protocol is a generalization of the majority locking protocol. To implement this protocol, a non-negative weight is assigned to each site of the distributed system. In addition, this approach also assigns integer values to read operations and write operations on each data item, called the **read quorum** and **write quorum** of the data item. In quorum consensus protocol, the read quorum and write quorum of a data item P, denoted as P_r and P_w, are assigned in such a way that they satisfy the following conditions:

$$P_r + P_w > T$$

and $2 * P_w > T$

where T is the total weight of all sites of the distributed system where the data item P resides. To perform a read operation on the data item P, enough number of replicas of the data item P stored at different sites must be read, so that their total weight is greater than or equal to P_r. On the other hand, to perform a write operation on the data item P enough number of copies of the data item P stored at different sites must be written, so that their weight is greater than or equal to P_w. In both cases, the protocol ensures that locks are achieved on a majority of the replicas of the data item P stored at different sites before executing a read or a write operation on P. Hence, the transaction coordinator would send lock requests to different lock managers accordingly.

The major benefit of the quorum consensus protocol is that it allows to reduce the cost of read and write operations selectively by appropriately assigning the read and write quorums to data items. For example, a read operation on a data item with a small read quorum requires reading of fewer replicas of the data item. But the write quorum of that data item may be higher, and hence, to succeed, the write operation needs to obtain locks on more replicas of the data item that are available. Furthermore, if higher weights are assigned to some sites, fewer sites need to be accessed for acquiring locks on a data item. The quorum consensus protocol can simulate the majority locking protocol and the biased protocol if appropriate weights and quorums are assigned. The disadvantage of majority locking protocol such as complexity in deadlock handling is shared by this protocol also.

Comparison of Lock-based Concurrency Control Strategies

	Implementation Complexity	**Reliability**	**Deadlock Handling**	**Communication Cost**
Centralized 2PL	Very Simple	Lowest reliability	Easy	Lowest
Primary Copy 2PL	Relatively simple	Lower reliability	Relatively complicated	Relatively low
Distributed 2PL	Complicated	Highest reliability	Highly complicated	Highest
Majority Locking Protocol	Complicated	High	Complicated	High
Biased Protocol	Complicated	High	Complicated	High
Quorum Consensus Protocol	Highly complicated	High	Complicated	High

8.6 Timestamp-Based Concurrency Control Protocols

The principal idea behind the timestamp-based concurrency control techniques is that a unique timestamp is assigned to each transaction to determine the serialization order of transactions in a distributed system. The serialization order is determined depending on the timestamp values of transactions prior to the execution of transactions. A timestamp value is a simple identifier that is used to identify each transaction uniquely and to permit ordering. In a centralized DBMS, generation of unique timestamp values for transactions can be handled in a simpler way while in a DDBMS timestamp values must be derived from a totally ordered domain.

There are two primary methods for generating unique timestamp values in a distributed environment: **centralized** and **distributed**. In the **centralized approach**, a single site is responsible for assigning unique timestamp values to all transactions that are generated at different sites of the distributed system. The central site can use a logical counter or its own system clock for assigning timestamp values. The centralized approach is very simple, but the major disadvantage of this approach is that it uses a central component for generating timestamp values, which is vulnerable to system failures. In the **distributed approach**, each site generates unique local timestamp values by using either a logical counter or its own system clock. To generate unique

global timestamp values, unique local timestamp values are concatenated with site identifiers and the pair <local timestamp value, site identifier> represents a unique global timestamp value across the whole system. The site identifier is appended in the least significant position to ensure that the global timestamp value generated in one site is not always greater than those generated in another site. This approach still has a problem: the system clocks at different sites would not be synchronized and one site can generate local timestamp values at a rate faster than that of other sites. If logical counters are used at each site to generate local timestamp values, there is the possibility of different sites generating the same value on the counter. The general approach that is used to generate unique global timestamp values in a distributed system is discussed in the following.

In this approach, each site generates unique local timestamp values based on its own local logical counter. To maintain uniqueness, each site appends its own identifier to the local counter value, in the least significant position. Thus, the global timestamp value is a two-tuple of the form <local counter value, site identifier>. To ensure the synchronization of all logical counters in the distributed system, each site includes its timestamp value in inter-site messages. After receiving a message, each site compares its timestamp value with the timestamp value in the message and if its timestamp value is smaller, then the site changes its own timestamp to a value that is greater than the message timestamp value. For instance, if site1 sends a message to site2 with its current timestamp value <8, 1>, where the current timestamp value of site2 is <10, 2>, then site2 would not change its timestamp value. On the other hand, if the current timestamp value of site2 is <6, 2>, then site2 would change its current timestamp value to <11, 2>.

8.6.1 Basic Timestamp Ordering (TO) Algorithm

The **basic TO algorithm** is a straightforward extension of the TO protocol for centralized DBMSs. Timestamp ordering is a technique by which a serialization order is selected a priori, and transaction execution is forced to obey this order. For two given conflicting operations Oi and Oj belonging to transactions $T1$ and $T2$ respectively, Oi is executed before Oj if and only if ts($T1$) < ts($T2$), where ts($T1$) and ts($T2$) are the corresponding timestamp values of transactions $T1$ and $T2$. Here $T1$ is the older transaction and $T2$ is the younger transaction, and according to the timestamp ordering, operations of transaction $T1$ will be executed before those of transaction $T2$.

In the basic TO algorithm, each transaction is assigned a unique timestamp value by its transaction manager. A timestamp ordering scheduler (which is a software module) is implemented to check the timestamp of each new operation against those of conflicting operations that have already been scheduled. The basic TO implementation distributes the schedulers along with the database. At each site, if the timestamp value of a new operation is found greater than the timestamp values of all conflicting operations that have already been scheduled, then it is accepted; otherwise, the corresponding transaction manager assigns a new timestamp value to the new operation, and the entire transaction is restarted. A timestamp ordering scheduler guarantees that transactions are conflict serializable and that the results are equivalent to a serial schedule in which the transactions are executed chronologically, that is, in the order of the timestamp values. The comparison between the timestamp values of different transactions can be done only when the timestamp ordering scheduler has received all the operations to be scheduled. In a distributed database environment, it may happen that operations come to the scheduler one at a time, and an operation may reach out of sequence. To overcome this, each data item is assigned two timestamp values: one is a **read_timestamp value**, which is the timestamp value of the last

transaction that had read the data item, and the other is a **write_timestamp value**, which is the timestamp value of the last transaction that had updated the data item. The implementation of the basic TO algorithm for a transaction T in a DDBMS is described below.

The coordinating transaction manager assigns a timestamp value to the transaction T, say ts(T), determines the sites where each data item is stored, and sends the relevant operations to these sites. Let us consider that the transaction T requires to perform a read operation on a data item x located at a remote site. The data item x has a read_timestamp value and a write_timestamp value, which are Rx and Wx respectively. The read operation on the data item x proceeds only if ts(T) > Wx, and a new value, ts(T), is assigned to the read_timestamp of data item x; otherwise, the operation is rejected and the corresponding message is sent to the coordinating transaction manager. In other cases, for instance, if ts(T) < Wx, it is indicated that the older transaction T is trying to read a value of the data item x that has been updated by a younger transaction. The older transaction T is too late to read the previous outdated value of the data item x, and the transaction T is aborted here. Similarly, the transaction T can perform a write operation on the data item x, only when the conditions ts(T) > Rx and ts(T) > Wx are satisfied. When an operation (read or write) of a transaction is rejected by the timestamp ordering scheduler, the transaction should be restarted with a new timestamp value, and it is assigned by the corresponding transaction manager.

To maintain the consistency of data, the data processor must execute the accepted operations in the order in which the scheduler passes them. The timestamp ordering scheduler maintains a queue for each data item to enforce the ordering, and delays the transfer of the next accepted operation on the same data item until an acknowledgement is received from the data processor regarding the completion of the previous operation.

The major advantage of the basic TO algorithm is that deadlock never occurs here, because a transaction that is rejected by the timestamp ordering scheduler can restart with a new timestamp value. However, in this approach a transaction may have to restart numerous times to facilitate deadlock freedom.

8.6.2 Conservative TO Algorithm

The **Conservative TO algorithm** is a technique for eliminating transaction restarts during timestamp ordering scheduling and thereby reducing the system overhead. The conservative TO algorithm differs from the basic TO algorithm in the way the operations are executed in each method. In basic TO algorithm, when an operation is accepted by the timestamp ordering scheduler, it passes the operation to the data processor for execution as soon as possible. In the case of conservative TO algorithm, when an operation is received by the timestamp ordering scheduler, the scheduler delays the execution of the operation until it is sure that no future restarts are possible. Thus, the execution of operations is delayed to ensure that no operation with a smaller timestamp value can arrive at the timestamp ordering scheduler. The scheduler will never reject an operation if the satisfaction of the above condition is assured. Thus, the operations of each transaction are buffered here until an ordering can be established, so that rejections are not possible, and they are executed in that order. The conservative TO algorithm is implemented in the following way.

In conservative TO algorithm, each scheduler maintains one queue for each transaction manager in the distributed system. Whenever an operation is received by a timestamp ordering scheduler, it buffers the operation according to the increasing order of timestamp values in the appropriate queue for the corresponding transaction manager. For instance, the timestamp ordering scheduler at site p stores all the operations that it has received from the transaction manager of site q in a queue Q_{pq}. The scheduler at each site passes a read or write operation with

the smallest timestamp value from these queues to the data processor for execution. The execution of the operation can proceed after establishing an ordering of the buffered operations so that rejections cannot take place.

The above implementation of the conservative TO algorithm suffers from some major problems. First, if some transaction managers never send any operations to a scheduler, the scheduler will get stuck and stop outputting. To overcome this problem, it is necessary that each transaction manager communicate regularly with every scheduler in the system, which is infeasible in large networks. Another problem is that although conservative TO reduces the number of restarts of the transactions, it is not guaranteed that they will be eliminated completely. For instance, consider that the timestamp scheduler at site $S1$ has chosen an operation with the smallest timestamp value, say $t(x)$, and passed it to the data processor for execution. It may happen that the site $S2$ has sent an operation to the scheduler of site $S1$ with a timestamp value less than $t(x)$ that may still be in transit in the network. When this operation reaches the site $S1$, it will be rejected, because it violates the TO rule. In this case, the operation wants to access a data item that is already accessed by another operation with a higher timestamp value.

One solution to the above problems is to ensure that each scheduler chooses an operation for execution and passes it to the data processor only when there is at least one operation in each queue. In the absence of real traffic, when a transaction manager does not have any operations to execute, it will send a **null operation** to each timestamp scheduler in the system periodically. Thus, the null operations are sent periodically to ensure that the operations that will be sent to schedulers in future will have higher timestamp values than the null operations. This approach is more restrictive and is called **extremely conservative TO algorithm**.

Another solution to the above problem is **transaction classes**, which is less restrictive and reduces communications compared to extremely conservative TO algorithm. In this technique, transaction classes are defined based on readset and writeset of transactions, which are known in advance. A transaction T is a member of a class C if readset(T) is a subset of readset(C) and writeset(T) is a subset of writeset(C). However, it is not necessary that transaction classes should be disjoint. The conservative TO algorithm has to maintain one queue for each class instead of maintaining one queue for the transaction manager at each site in the distributed system. With this modification, the timestamp scheduler will choose an operation of a transaction for execution with smallest timestamp value until there is at least one operation in each class to which the transaction belongs.

Example 8.6

Let us consider the following transaction classes and transactions:

Class C1: readset = {a1}, writeset = {b1, b2}

Class C2: readset = {a1, b2}, writeset = {b1, b2, c2, c3}

Class C3: readset = {b2, c3}, writeset = {a1, c2, c3}

Transaction T1: readset = {a1}, writeset = {b1, b2}

Transaction T2: readset = {b2}, writeset = {c2, c3}

Transaction T3: readset = {c3}, writeset = {a1}

Hence, $T1$ is a member of classes $C1$ and $C2$, $T2$ is a member of classes $C2$ and $C3$, and $T3$ is a member of class $C3$. Now, the conservative TO algorithm has to maintain one queue for each class

*C*1, *C*2, and *C*3 instead of maintaining one queue for each transaction manager at each site in the distributed system. The timestamp scheduler will choose an operation of a transaction, with the smallest timestamp value, for execution until there is at least one operation in each class *C*1, *C*2 or *C*3 to which the transaction belongs.

8.6.3 Multi-version TO Algorithm

Multi-version TO algorithm is another timestamp-based protocol that reduces restart overhead of transactions. In multi-version TO algorithm, the updates of data items do not modify the database, but each write operation creates a new version of a data item while retaining the old version. When a transaction requires reading a data item, the system selects one of the versions that ensures serializability. Multi-version TO algorithm is implemented in the following way.

In this method, for each data item *P* in the system, the database holds a number of versions such as *P*1, *P*2, *P*3,. . . , *Pn* respectively. Furthermore, for each version of the data item *P*, the system stores three values: the value of version *Pi*, read_timestamp(*Pi*), which is the largest timestamp value of all transactions that have successfully read the version *Pi*, and write_timestamp(*Pi*), which is the timestamp value of the transaction that created the version *Pi*. The existence of versions of data items is transparent to users. The transaction manager assigns a timestamp value to each transaction, which is used to keep track of the timestamp values of each version. When a transaction *T* with timestamp value ts(*T*) attempts to perform a read or write operation on the data item *P*, the multi-version TO algorithm uses the following two rules to ensure serializability.

Rule 1: If the transaction *T* wants to read the data item *P*, a version *Pj* is chosen that has the largest write_timestamp value that is less than ts(*T*), that is, write_timestamp(*Pj*) < ts(*T*). In this case, the read operation is sent to the data processor for execution and the value of read_timestamp(*Pj*) is reset as ts(*T*).

Rule 2: If the transaction *T* wishes to perform a write operation on the data item *P*, it must be ensured that the data item *P* has not already been read by some other transaction whose timestamp value is greater than ts(*T*). Thus, the write operation is permitted and sent to the data processor if the version *Pj* that has the largest write_timestamp of data item *P* that is less than ts(*T*) satisfies the condition read_timestamp(*Pj*) < ts(*T*). If the operation is permitted, a new version of the data item *P*, say *Pk*, is created where read_timestamp(*Pk*) = write_timestamp(*Pk*) = ts(*T*). Otherwise, the transaction *T* is aborted and restarted with a new timestamp value.

The multi-version TO algorithm requires more storage space to store the values of different versions of data items with respect to time. To save storage space, older versions of data items can be deleted if they are no longer required. The multi-version TO algorithm is a suitable concurrency control mechanism for DBMSs that are designed to support applications that inherently have a notion of versions of database objects such as engineering databases and document databases.

8.7 Optimistic Concurrency Control Technique

Optimistic concurrency control technique is a non-blocking and deadlock free approach for concurrency control. In this technique, it is assumed that the conflicts between transactions are rare, and the transactions are allowed to proceed for execution without imposing delays. When a transaction

wants to commit, a validation checking is made to determine whether a conflict has occurred or not. To ensure serializability, the transaction is aborted and restarted if any conflict has occurred. In optimistic concurrency control technique, the execution of any operation of a transaction follows a sequence of phases depending on whether it is a read operation or an update operation. These are read phase, validate phase and write phase as in figure 8.5

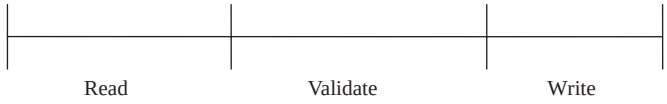


Fig. 8.5 Phases of Optimistic Concurrency Control Technique

The read and write operations of each transaction are processed without delay, but no update is made to the actual database. Initially, each transaction makes its updates on local copies of data items and to ensure the consistency of the database, a checking is done on these updates in the validate phase. The changes are made global and written into the actual database if local updates caused no inconsistencies of the database.

The optimistic concurrency control techniques can be implemented either based on locking or based on timestamp ordering. The optimistic concurrency control technique for DDBMS based on timestamp ordering [proposed by Ceri and Owicki] is implemented in the following way.

Whenever a global transaction is generated at a site, the transaction manager of that site (called the transaction coordinator) divides the transaction into several subtransactions, each of which can execute at many sites. Each local execution can then proceed without delay and follows the sequence of phases of optimistic concurrency control technique until it reaches the validate phase. The transaction manager assigns a timestamp value to each transaction at the beginning of the validate phase that is copied to all its subtransactions. Hence, no timestamp values are assigned to data items and a timestamp value is assigned to each transaction only at the beginning of the validate phase, because its early assignment may cause unnecessary transaction restarts. Let us consider that a transaction T_p is divided into a number of subtransactions, and a subtransaction of T_p that executes at site q is denoted by T_{pq}. To perform the local validation in the validate phase, the following three rules are followed:

Rule 1: For each transaction T_r, if $\text{ts}(T_r) < \text{ts}(T_{pq})$ (that is, the timestamp value of each transaction T_r is less than the timestamp value of the subtransaction T_{pq}) and if each transaction T_r has finished its write phase before T_{pq} has started its read phase, the validation succeeds.

Rule 2: If there is any transaction T_r such that $\text{ts}(T_r) < \text{ts}(T_{pq})$ that completes its write phase while T_{pq} is in its read phase, then the validation succeeds, if none of the data items updated by the transaction T_r are read by the subtransaction T_{pq} and the transaction T_r completes writing its updates into the database before the subtransaction T_{pq} starts writing. Thus, $WS(T_r) \cap RS(T_{pq}) = \Phi$, where $WS(T_r)$ represents the write_timestamp value of the data item S after the transaction T_r completes a write operation on it, and $RS(T_{pq})$ represents the read_timestamp value of the data item S after the subtransaction T_{pq} has performed a read operation on S.

Rule 3: If there is any transaction T_r such that $\text{ts}(T_r) < \text{ts}(T_{pq})$ that completes its read phase before T_{pq} finishes its read phase, then the validation succeeds, if the updates made by the transaction T_r do not affect the read phase or the write phase of the transaction $T_{pq.}$ In this case, $WS(T_r) \cap RS(T_{pq}) = \Phi$, and $WS(T_r) \cap WS(T_{pq}) = \Phi$.

In this algorithm Rule1 is used to ensure that the transactions are executed serially according to their timestamp ordering. Rule2 indicates that the set of data items written by the older transaction are not read by the younger transaction, and the older transaction completes its write phase before the younger transaction enters its validate phase. Rule3 state that the updates on data items made by the older transaction do not affect the read phase or write phase of the younger transaction, but as in Rule2 it does not require that the older transaction finish writing on data items before the younger transaction starts writing.

The major advantage of optimistic concurrency control algorithm is that a higher level of concurrency is allowed here, since the transactions can proceed without imposed delays. This approach is very efficient when there are few conflicts, but conflicts can result in the rollback of individual transactions. However, there are no cascading rollbacks, because the rollback involves only a local copy of the data. A disadvantage of optimistic concurrency control algorithm is that it incurs higher storage cost. To validate a transaction, the optimistic approach has to store the read and the write sets of several other terminated transactions. Another disadvantage with this approach is starvation. It may happen that a long transaction fails repeatedly in the validate phase during successive trials. This problem can be overcome by allowing the transaction exclusive access to the data items after a specified number of trials, but this reduces the level of concurrency to a single transaction.

CHAPTER SUMMARY

- Concurrency control is the activity of coordinating concurrent accesses to a database in a multi-user system, while preserving the consistency of the data. A number of anomalies can occur owing to concurrent accesses on data items in a DDBMS, which can lead to the inconsistency of the database. These are **lost update anomaly, uncommitted dependency or dirty read anomaly, inconsistent analysis anomaly, non-repeatable or fuzzy read anomaly, phantom read anomaly** and **multiple-copy consistency problem**. To prevent data inconsistency, it is essential to guarantee distributed serializability of concurrent transactions in a DDBMS.
- A **distributed schedule** or **global schedule** (the union of all local schedules) is said to be **distributed serializable** if the execution orders at each local site is serializable and local serialization orders are identical.
- The concurrency control techniques are classified into two broad categories: **Pessimistic concurrency control mechanisms** and **Optimistic concurrency control mechanisms**.
- **Pessimistic algorithms** synchronize the concurrent execution of transactions early in their execution life cycle.
- **Optimistic algorithms** delay the synchronization of transactions until the transactions are near to their completion. The pessimistic concurrency control algorithms are further classified into **locking-based algorithms, timestamp-based algorithms** and **hybrid algorithms**.
- In locking-based concurrency control algorithms, the synchronization of transactions is achieved by employing physical or logical locks on some portion of the database or on the entire database.
- In the case of timestamp-based concurrency control algorithms, the synchronization of transactions is achieved by assigning timestamps both to the transactions and to the data items that are stored in the database.
- To allow maximum concurrency and to improve efficiency, some locking-based concurrency control algorithms also involve timestamps ordering; these are called **hybrid concurrency control algorithms**.

EXERCISES

Multiple Choice Questions

(i) Which of the following problems occurs due to concurrent access of data?
   a. Lost update
   b. Fuzzy read
   c. Phantom read
   d. All of the above.

(ii) Which of the following problems occurs due to concurrent access of data in distributed database systems only?
   a. Lost update
   b. Fuzzy read
   c. Phantom read
   d. Multiple copy consistency.

(iii) Which of the following operations is conflicting?
   a. Read-write
   b. Write-read
   c. Write-write
   d. All of the above
   e. None of these.

(iv) A schedule is said to be serializable
   a. If the result is the same as in serial execution of the same set of transactions
   b. If the result is the same as in non-serial execution of the same set of transactions
   c. If the result is the same as in serial execution of a different set of transactions
   d. None of these.

(v) A global schedule of a distributed system is serializable if
   a. It is the union of local schedules
   b. The execution orders at each local site is serializable
   c. The execution orders at each local site is serializable and local serialization orders are identical
   d. None of these.

(vi) Which of the following statements is true?
   a. Optimistic algorithms synchronize the concurrent execution of transactions early in their execution life cycle
   b. Pessimistic algorithms synchronize the concurrent execution of transactions early in their execution life cycle
   c. In both cases the synchronization is delayed until transactions are nearer to their completion
   d. None of these.

(vii) Which of the following concurrency control algorithms is deadlock free?
   a. Pessimistic algorithms
   b. Optimistic algorithms
   c. Both optimistic and pessimistic algorithms
   d. Hybrid algorithms.

(viii) Which of the following is not a classification of optimistic concurrency control algorithms?
   a. Locking-based algorithms
   b. Timestamp-based algorithms
   c. Hybrid algorithms
   d. None of the above.

(ix) Which of the following concurrency control algorithms is used in DDBMSs?
   a. Lock-based algorithms
   b. Timestamp-based algorithms
   c. Hybrid algorithms
   d. All of these.

(x) The granularity of lock represents
   a. The size of the data item locked
   b. The total number of sharable locks obtained by a transaction
   c. The total number of exclusive locks obtained by a transaction
   d. The sum of b. and c.

(xi) Which of the following statements is correct?
   a. Downgradation of lock is the transformation of a sharable lock into an exclusive lock
   b. Upgradation of lock is the transformation of a sharable lock into an exclusive lock
   c. Upgradation of lock is the transformation of an exclusive lock into a sharable lock
   d. All of these.

(xii) Cascading rollback is
   a. The occurrence of a series of rollbacks owing to the rollback of a single transaction
   b. The occurrence of the rollback of a single transaction owing to a series of rollbacks
   c. Both a. and b.
   d. None of the above.

(xiii) Which of the following methods is used to avoid cascading rollback?
   a. Rigorous 2PL
   b. Strict 2PL

c. All of the above
d. Linear 2PL.

(xiv) If a distributed system has n sites, the total number of message transfers in centralized 2PL is
a. $2n$
b. $3n+2$
c. $2n+3$
d. $5n$.

(xv) Which of the following is a ROWA (read-one-write-all) protocol?
a. Primary copy 2PL
b. Majority locking
c. Distributed 2PL
d. Centralized 2PL.

(xvi) If a distributed system has n sites, the total number of message transfers in distributed 2PL is
a. $2n+3$
b. $5n$
c. $n * n$
d. $n * (n+1)/2$.

(xvii) Which of the following is a replica control protocol?
a. Distributed 2PL
b. Biased protocol
c. All of the above
d. Majority locking.

(xviii) If P_r and P_w denote the read quorum and write quorum of a data item P, and T denotes the total weight of the distributed system, which of the following conditions must be satisfied by quorum consensus protocol?
a. $P_r + P_w > T$
b. $2 * P_w > T$
c. $P_r + P_w > T$ and $2 * P_w > T$
d. None of these.

(xix) In Basic TO algorithm
a. Younger transactions are executed first
b. Older transaction are restarted
c. Older transactions are executed first
d. Both older and younger transactions are restarted.

(xx) In the absence of real traffic, null operations are sent by transaction managers to timestamp scheduler in
a. Conservative TO algorithm
b. Extremely conservative TO algorithm
c. Multi-version TO algorithm
d. Basic TO algorithm.

(xxi) Optimistic concurrency control approaches are
a. Non-blocking
b. Deadlock free
c. Both non-blocking and deadlock free
d. None of the above.

(xxii) Which of the following phases does not belong to an optimistic concurrency control technique?
a. Read
b. Write
c. Execute
d. Validate.

(xxiii) In optimistic concurrency control approach
a. Read phase makes all updates in the actual database
b. Write phase makes all updates in the actual database
c. Both read and write phases do not make any updates in the actual database
d. Write phase makes updates in the actual database but read phase does not.

(xxiv) Transaction restart overhead is removed in
a. Basic TO algorithm
b. Conservative TO algorithm
c. Multi-version TO algorithm
d. Both b. and c.
e. None of the above.

Review Questions

1. Explain the objectives of distributed concurrency control.
2. Discuss the phantom read anomaly and multi-copy consistency problem.
3. Define distributed serializability. How is it ensured in a DDBMS?
4. Compare and contrast Pessimistic and Optimistic Concurrency Control techniques.

5. What is cascading rollback? Differentiate between rigorous 2PL protocol and strict 2PL protocol.
6. How are the disadvantages of 2PL overcome in strict 2PL?
7. Compare locking-based protocols and timestamp-based protocols.
8. Define granularity of a lock. Explain upgradation and downgradation of locking.
9. Compare and contrast different lock-based protocols.
10. Describe quorum-based protocol for distributed concurrency control.
11. Describe distributed 2PL protocol.
12. Discuss how a unique global timestamp is generated in a distributed system.
13. "Unique timestamp generation is difficult in a DDBMS than in a centralized DBMS". Justify.
14. Differentiate between basic TO algorithm and conservative TO algorithm.
15. Explain multi-version TO algorithm. Write down its advantages and disadvantages.
16. Describe optimistic concurrency control algorithm.
17. Write the difference between a complete schedule and a schedule.
18. Explain conflict operations and conflict equivalence.
19. Consider the following transactions:

    T1: {Read(X), Write(X), Commit}

    T2: {Write(X), Write(Y), Read(Z), Commit}

    T3: {Read(X), Read(Y), Read(Z), Commit}

    (a) Draw complete schedule DAG and schedule DAG.

    (b) Write a schedule including all the above transactions, which is serial.

20. Consider a distributed database system in which data items *p* and *q* are stored at Site 1 and *r* and *t* are stored at Site 2. There are two transactions *Ti* and *Tj* that perform read and write operations on these data items at two different sites as mentioned in the following. *Ri*(*p*) denotes that the transaction *Ti* reads the data item *p*, and *Wi*(*p*) denotes that the transaction *Ti* performs a write operation on data item *p*. *S*1 and *S*2 denote the schedules of Site1 and Site2 respectively. For each of the following executions, determine whether global schedules are serializable or not. If serializable, determine all possible total orders of transactions.

    Execution 1: S1: Ri(p), Rj(q), Wj(q), Wi(q)

    S2: Ri(r), Wj(t), Wi(r), Wi(t)

    Execution 2: S1: Ri(p), Rj(p), Wj(p), Wi(p)

    S2: Wi(r)

    Execution 3: S1: Ri(q), Rj(p), Wj(p)

    S2: Ri(t), Rj(r), Wj(t), Wi(t)

21. Consider the following transactions:

    T1: {Read(X), Write(X), Commit}

    T2: {Read(X), Write(Y), Read(Z), Commit}

Draw complete schedule DAG and schedule DAG. Determine whether the following two schedules are serializable or not.

S1: Ri(X), Wi(X), Rj(X), Wj(X)

S2: Rj(X), Wj(X), Ri(X), Wi(X).

22. Consider the example 8.2. Determine all possible executions of the two transactions if 2PL is used for concurrency control.

23. Determine possible executions and transaction restarts in each of the following cases for two transactions *Ti* and *Tj*, assuming read_timestamp(x) = 10, write_timestamp(x) = 10, read_timestamp(y) = 15, and write_timestamp(y) = 15.

    (i) Timestamp(Ti) = 25, timestamp(Tj) = 30

    (ii) Timestamp(Ti) = 30, timestamp(Tj) = 25

    (iii) Timestamp(Ti) = 8, timestamp(Tj) = 25

    (iv) Timestamp(Ti) = 35, timestamp(Tj) = 8

24. Consider read_timestamp(x) = 20, write_timestamp(x) = 15 and {$Ri(x)$, n} denotes a read request of transaction *Ti* on data item x with a timestamp value n. Similarly, {$Wi(x)$, m} denotes a write request of transaction *Ti* on data item x with a timestamp value m. Using basic TO algorithm, determine the behavior of the following sequence of requests.

    (i) {Ri(x), 10} {Rj(x), 20} {Wk(x), 10}, {Wl(x), 25}

    (ii) {Wp(x), 30} {Rq(x), 35} {Wr(x), 32}.

This page is intentionally left blank

9 Distributed Deadlock Management

This chapter introduces different deadlock management techniques to handle deadlock situations in a distributed database system. Distributed deadlock prevention, distributed deadlock avoidance, and distributed deadlock detection and recovery methods are briefly discussed in this chapter.

The outline of this chapter is as follows. Section 9.1 addresses the deadlock problem, and the deadlock prevention methods for distributed systems are discussed in Section 9.2. In Section 9.3, distributed deadlock avoidance is represented. The techniques for distributed deadlock detection and recovery are focused on in Section 9.4.

9.1 Introduction to Deadlock

In a database environment, **a deadlock** is a situation when transactions are endlessly waiting for one another. Any lock-based concurrency control algorithm and some timestamp-based concurrency control algorithms may result in deadlocks, as these algorithms require transactions to wait for one another. In lock-based concurrency control algorithms, locks on data items are acquired in a mutually exclusive manner; thus, it may cause a deadlock situation. Whenever a deadlock situation arises in a database environment, outside interference is required to continue with the normal execution of the system. Therefore, the database systems require special procedures to resolve the deadlock situation.

Deadlock situations can be characterized by **wait-for graphs**, directed graphs that indicate which transactions are waiting for which other transactions. In a wait-for graph, nodes of the graph represent transactions and edges of the graph represent the waiting-for relationships among transactions. An edge is drawn in the wait-for graph from transaction T_i to transaction T_j, if the transaction T_i is waiting for a lock on a data item that is currently held by the transaction T_j. Using wait-for graphs, it is very easy to detect whether a deadlock situation has occurred in a database environment or not. There is a deadlock in the system if and only if the corresponding wait-for graph contains a cycle.

The resolution of a deadlock situation is much easier in a centralized DBMS than in a distributed DBMS. In a centralized DBMS, only one **local wait-for graph** (LWFG) is drawn to detect the deadlock situation. The detection of deadlocks in a distributed DBMS is more complicated, because the circular waiting situation, which determines a deadlock, may involve several different sites. Thus, in a distributed DBMS it is not sufficient to draw a LWFG for each local DBMS only, but it is also necessary to draw a **global wait-for graph** (**GWFG**) for the entire system to detect a deadlock situation. In a distributed database, an LWFG is a portion of the GWFG, which consists of only those nodes and edges that are completely contained at a single site. Three general techniques are available for deadlock resolution in a distributed database system: **distributed**

deadlock prevention, distributed deadlock avoidance and **distributed deadlock detection and recovery from deadlock**. These are described in the following.

Example 9.1

Let us assume that in a database environment there are two transactions T_1 and T_2 respectively. Further assume that currently the transaction T_1 is holding an exclusive lock on data item P, and the transaction T_2 is holding an exclusive lock on data item Q. Now, if the transaction T_1 requires a write operation on data item Q, the transaction T_1 has to wait until the transaction T_2 releases the lock on the data item Q. However, in the meantime if the transaction T_2 requires a read or a write operation on the data item P, the transaction T_2 also has to wait for the transaction T_1. In this situation, both the transactions T_1 and T_2 have to wait for each other indefinitely to release their respective locks, and no transaction can proceed for execution; thus a deadlock situation arises.

9.2 Distributed Deadlock Prevention

Distributed Deadlock prevention is a cautious scheme in which a transaction is restarted when the system suspects that a deadlock might occur. Deadlock prevention is an alternative method to resolve deadlock situations in which a system is designed in such a way that deadlocks are impossible. In this scheme, the transaction manager checks a transaction when it is first initiated and does not permit to proceed if there is a risk that it may cause a deadlock. In the case of lock-based concurrency control, deadlock prevention in a distributed system is implemented in the following way.

Let us consider that a transaction T_i is initiated at a particular site in a distributed database system and that it requires a lock on a data item that is currently owned by another transaction T_j. Here, a deadlock prevention test is done to check whether there is any possibility of a deadlock occurring in the system. The transaction T_i is not permitted to enter into a wait state for the transaction T_j, if there is the risk of a deadlock situation. In this case, one of the two transactions is aborted to prevent a deadlock. The deadlock prevention algorithm is called **non-preemptive** if the transaction T_i is aborted and restarted. On the other hand, if the transaction T_j is aborted and restarted, then the deadlock prevention algorithm is called **preemptive**. The transaction T_i is permitted to wait for the transaction T_j as usual, if they pass the prevention test. The prevention test must guarantee that if T_i is allowed to wait for T_j, a deadlock can never occur. Here, one simple approach to prevent deadlock situations is to ensure that the transaction T_i never waits for the transaction T_j, but this forces a number of restarts.

A better approach to implement deadlock prevention test is by assigning priorities to transactions and checking priorities to determine whether one transaction would wait for the other transaction or not. These priorities can be assigned by using a unique identifier for each transaction in a distributed system. For instance, consider that i and j are the priorities of two transactions T_i and T_j respectively. The transaction T_i would wait for the transaction T_j, if T_i has a lower priority than T_j, that is, if $i<j$. This approach prevents deadlock, but one problem with this approach is that cyclic restart is possible. Thus, some transactions could be restarted repeatedly without ever finishing.

One solution to the above problem is using the unique timestamp value of each transaction as the priority of that transaction. One way to obtain a global timestamp value for every transaction in a distributed system is assigning a unique local timestamp value to each transaction by its local transaction manager and then appending the site identifier to the low-order bits of this value. Thus,

timestamp values are unique throughout the distributed system and do not require that clocks at different sites are synchronized precisely. Based on timestamp values, there are two different techniques for deadlock prevention: **Wait-die** and **Wound-Wait**.

Wait-die is a non-preemptive deadlock prevention technique based on timestamp values of transactions and is implemented in the following way. In this technique, when one transaction is about to block and is waiting for a lock on a data item that is already locked by another transaction, timestamp values of both the transactions are checked to give priority to the older transaction. If a younger transaction is holding the lock on the data item then the older transaction is allowed to wait, but if an older transaction is holding the lock, the younger transaction is aborted and restarted with the same timestamp value. This forces the wait-for graph to be directed from the older to the younger transactions, making cyclic restarts impossible. For example, if the transaction T_i requests a lock on a data item that is already locked by the transaction T_j, then T_i is permitted to wait only if T_i has a lower timestamp value than T_j. On the other hand, if T_i is younger than T_j, then T_i is aborted and restarted with the same timestamp value.

Wound-Wait is an alternative preemptive deadlock prevention technique by which cyclic restarts can be avoided. In this method, if a younger transaction requests for a lock on a data item that is already held by an older transaction, the younger transaction is allowed to wait until the older transaction releases the corresponding lock. In this case, the wait-for graph flows from the younger to the older transactions, and cyclic restart is again avoided. For instance, if the transaction T_i requests a lock on a data item that is already locked by the transaction T_j, then T_i is permitted to wait only if T_i has a higher timestamp value than T_j; otherwise, the transaction T_j is aborted and the lock is granted to the transaction T_i.

Deadlock situation cannot arise and the younger transaction is restarted in both the above methods. The main difference between the two techniques is that whether they preempt the active transaction or not. In the wait-die method a transaction can only be restarted when it requires accessing a new data item for the first time. A transaction that has acquired locks on all the required data items will never be aborted. In wound-wait method, it is possible that a transaction that is already holding a lock on a data item is aborted, because another older transaction may request the same data item. Wait-die method gives preference to younger transactions and aborts older transactions, as older transactions wait for younger ones and they tend to wait longer as they get older. On the other hand, wound-wait method prefers older transactions, as an older transaction never waits for a younger transaction. In wait-die method, it is possible that a younger transaction is aborted and restarted a number of times if the older transaction holds the corresponding lock for a long time, while in wound-wait method the older transaction is aborted and restarted only once.

The main disadvantage of a deadlock prevention scheme is that it may result in unnecessary waiting and rollback. Furthermore, some deadlock prevention schemes may require more sites in a distributed system to be involved in the execution of a transaction.

9.3 Distributed Deadlock Avoidance

Distributed deadlock avoidance is another technique to ensure that deadlock situations will not occur in a distributed system. **Preordering of resources** is a deadlock avoidance technique in which each data item in the database system is numbered, and each transaction requests locks on these data items in that numeric order. This technique requires that each transaction obtain all its locks before execution. Numbering of data items can be done either globally or locally. In distributed

database systems, it is necessary that all sites are numbered, and transactions that require to access data items from multiple sites must request the corresponding locks by visiting the sites according to the predefined numbers. The priority of a transaction is the highest number among the locks that are already owned by the transaction. In this method, as a transaction can only wait for those transactions that have higher priorities, no deadlock situations can occur.

Deadlock avoidance methods are more suitable than deadlock prevention schemes for database environments, but they require runtime support for deadlock management, which adds runtime overheads in transaction execution. Further, in addition to requiring predeclaration of locks, a principal disadvantage of this technique is that it forces locks to be obtained sequentially, which tends to increase response time.

9.4 Distributed Deadlock Detection and Recovery

Deadlock detection and recovery is the most popular and most suitable technique for deadlock management in a database environment. In deadlock detection and recovery method, first it is checked whether any deadlock has occurred in the system. After detection of a deadlock situation in the system, one of the involved transactions is chosen as the victim transaction and is aborted to resolve the deadlock situation. Deadlock situations are detected by explicitly constructing a wait-for graph and searching it for cycles. A cycle in the wait-for graph indicates that a deadlock has occurred, and one transaction in the cycle is chosen as the victim, which is aborted and restarted. To minimize the cost of restarting, the victim selection is usually based on the number of data items used by each transaction in the cycle.

The principal difficulty in implementing deadlock detection in a distributed database environment is constructing the **GWFG** efficiently. In a distributed DBMS, a **LWFG** for each local DBMS can be drawn easily; however, these LWFGs are not sufficient to represent all deadlock situations in the distributed system. For example, consider that there are three different sites in a distributed system, and each site has constructed a LWFG as shown in figure 9.1. The LWFG for each site is constructed in the usual manner using local transactions and data items stored at that particular site. A cycle in a LWFG indicates that a deadlock has occurred locally. The LWFGs in figure 9.1 illustrate that no deadlock has occurred locally in the three different sites, as there are no cycles in the LWFGs, but this does not guarantee that no deadlock has occurred globally. To detect a deadlock situation in the distributed system, it is necessary to construct a GWFG from these different LWFGs and to search it for cycles.

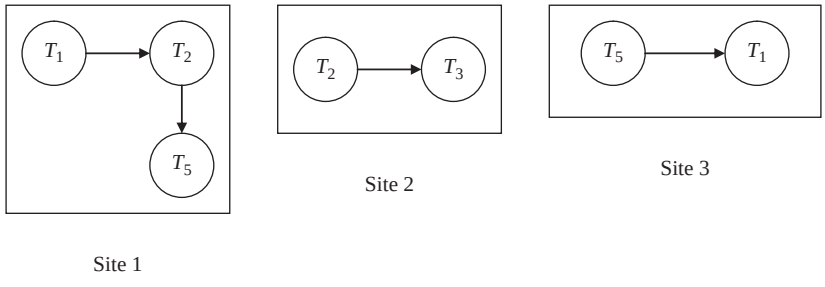


Fig. 9.1 Local Wait-for Graphs at Different Sites

The corresponding GWFG in figure 9.2 illustrate that a deadlock has occurred in the distributed system, although no deadlock has occurred locally.

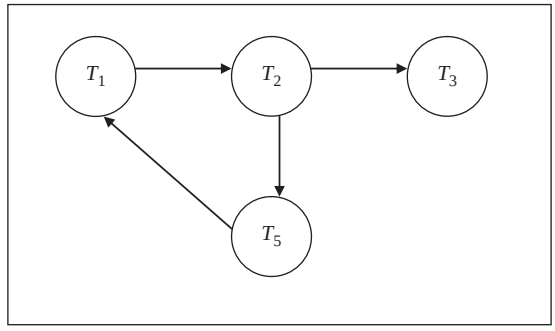


Fig. 9.2 Global Wait-for Graph for Figure 9.1

There are three different techniques for detecting deadlock situations in a distributed system: **Centralized deadlock detection, Hierarchical deadlock detection** and **Distributed deadlock detection**, which are described in the following.

9.4.1 Centralized Deadlock Detection

In **Centralized Deadlock Detection** method, a single site is chosen as **Deadlock Detection Coordinator (DDC)** for the entire distributed system. The DDC is responsible for constructing the GWFG for the system. Each lock manager in the distributed database system transmits its LWFG to the DDC periodically. The DDC constructs the GWFG from these LWFGs and checks for cycles in it. The occurrence of a global deadlock situation is detected if there are one or more cycles in the GWFG. The DDC must break each cycle in the GWFG by selecting the transactions to be rolled back and restarted to recover from a deadlock situation. The information regarding the transactions that are to be rolled back and restarted must be transmitted to the corresponding lock managers by the deadlock detection coordinator.

To minimize communication cost, the DDC should only send the changes that have to be made in the LWFGs to the lock managers. These changes represent the addition or removal of edges in the LWFGs. The actual length of a period for global deadlock detection is a system design decision, and it is a trade-off between the communication cost of the deadlock detection process and the cost of detecting deadlocks late. The communication cost increases if the length of the period is larger, whereas some deadlocks in the system go undetected if the length of the deadlock detection period is smaller.

The centralized deadlock detection approach is very simple, but it has several drawbacks. This method is less reliable, as the failure of the central site makes the deadlock detection impossible. The communication cost is very high in this case, as other sites in the distributed system send their LWFGs to the central site. Another disadvantage of centralized deadlock detection technique is that false detection of deadlocks can occur, for which the deadlock recovery procedure may be initiated, although no deadlock has occurred. In this method, unnecessary rollbacks and restarts of transactions may also result owing to phantom deadlocks. [False deadlocks and phantom deadlocks are discussed in detail in **Section 9.4.4**.]

9.4.2 Hierarchical Deadlock Detection

Hierarchical deadlock detection method reduces the communication overhead of centralized deadlock detection method. With this approach, all sites in a distributed database system are organized into a hierarchy, and a complete tree of deadlock detectors is constructed instead of a single centralized deadlock detector. Each site in the distributed system sends its LWFG to the deadlock detection site above it (adjacent parent node) in the hierarchy. Thus, local deadlock detection is performed in the leaf nodes of the tree, whereas the non-leaf nodes are responsible for detecting any deadlock situation involving all its child nodes. The hierarchical deadlock detection method is illustrated with an example in figure 9.3.

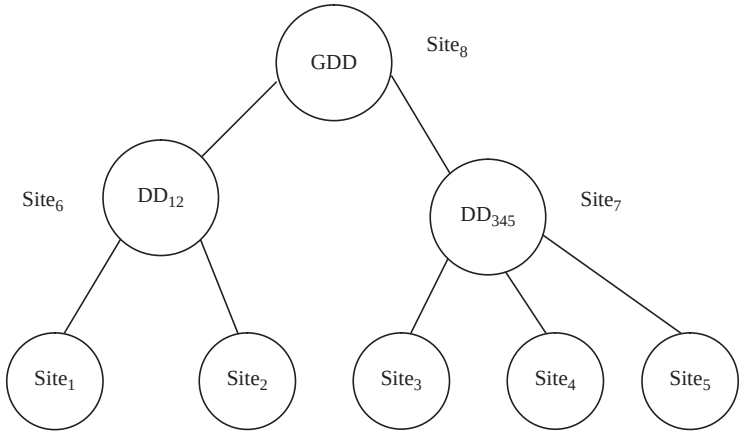


Fig. 9.3 Hierarchical Deadlock Detection

In figure 9.3, local deadlock detection is performed at leaf nodes $site_1$, $site_2$, $site_3$, $site_4$ and $site_5$. The deadlock detector at $site_6$, namely DD_{12}, is responsible for detecting any deadlock situation involving its child nodes $site_1$ and $site_2$. Similarly, $site_3$, $site_4$ and $site_5$ send their LWFGs to $site_7$ and the deadlock detector at $site_7$ searches for any deadlocks involving its adjacent child nodes. A global deadlock detector exists at the root of the tree that would detect the occurrence of global deadlock situations in the entire distributed system. Here, the global deadlock detector resides at $site_8$ and would detect the deadlocks involving $site_6$ and $site_7$.

The performance of the hierarchical deadlock detection approach depends on the hierarchical organization of nodes in the system. This organization should reflect the network topology and the pattern of access requests to different sites of the network. This approach is more reliable than centralized deadlock detection as it reduces the dependence on a central site; also it reduces the communication cost. However, its implementation is considerably more complicated, particularly with the possibility of site and communication link failures. In this approach, detection of false deadlocks can also occur.

9.4.3 Distributed Deadlock Detection

In **distributed deadlock detection** method, a deadlock detector exists at each site of the distributed system. In this method, each site has the same amount of responsibility, and there is no such distinction as local or global deadlock detector. A variety of approaches have been proposed for

distributed deadlock detection algorithms, but the most well-known and simplified version, which is presented here, was developed by R. Obermarck in 1982.

In this approach, a LWFG is constructed for each site by the respective local deadlock detectors. An additional external node is added to the LWFGs, as each site in the distributed system receives the potential deadlock cycles from other sites. In the distributed deadlock detection algorithm, the external node T_{ex} is added to the LWFGs to indicate whether any transaction from any remote site is waiting for a data item that is being held by a transaction at the local site or whether any transaction from the local site is waiting for a data item that is currently being used by any transaction at any remote site. For instance, an edge from the node T_i to T_{ex} exists in the LWFG, if the transaction T_i is waiting for a data item that is already held by any transaction at any remote site. Similarly, an edge from the external node T_{ex} to T_i exists in the graph, if a transaction from a remote site is waiting to acquire a data item that is currently being held by the transaction T_i at the local site. Thus, the local detector checks for two things to determine a deadlock situation. If a LWFG contains a cycle that does not involve the external node T_{ex}, then it indicates that a deadlock has occurred locally and it can be handled locally. On the other hand, a global deadlock potentially exists if the LWFG contains a cycle involving the external node T_{ex}. However, the existence of such a cycle does not necessarily imply that there is a global deadlock, as the external node T_{ex} represents different agents.

The LWFGs are merged so as to determine global deadlock situations. To avoid sites transmitting their LWFGs to each other, a simple strategy is followed here. According to this strategy, one timestamp value is allocated to each transaction and a rule is imposed such that one site S_i transmits its LWFG to the site S_k, if a transaction, say T_k, at site S_k is waiting for a data item that is currently being held by a transaction T_i at site S_i and $ts(T_i) < ts(T_k)$. If $ts(T_i) < ts(T_k)$, the site S_i transmits its LWFG to the site S_k, and the site S_k adds this information to its LWFG and checks for cycles not involving the external node T_{ex} in the extended graph. If there is no cycle in the extended graph, the process continues until a cycle appears and it may happen that the entire GWFG is constructed and no cycle is detected. In this case, it is decided that there is no deadlock in the entire distributed system. On the other hand, if the GWFG contains a cycle not involving the external node T_{ex}, it is concluded that a deadlock has occurred in the system. The distributed deadlock detection method is illustrated in figure 9.4.

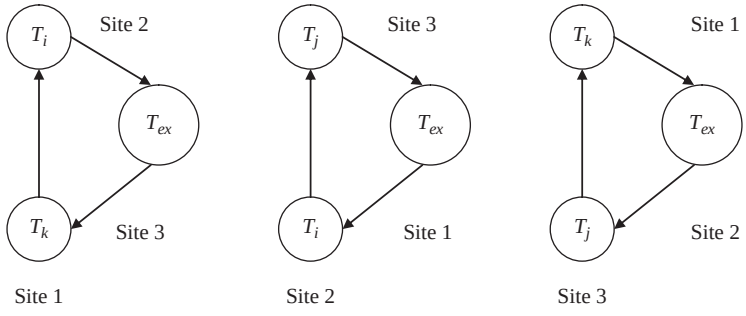


Fig. 9.4 Distributed Deadlock Detection

After detecting a deadlock in the system, an appropriate recovery protocol is invoked to resolve the deadlock situation. One or more transactions are chosen as victims, and these transactions, together with all their subtransactions, are rolled back and restarted.

The major benefit of distributed deadlock detection algorithm is that it is potentially more robust than the centralized or hierarchical methods. Deadlock detection is more complicated in this method, because no single site contains all the information that is necessary to detect a global deadlock situation in the system, and therefore substantial inter-site communication is required, which increases the communication overhead. Another disadvantage of distributed deadlock detection approach is that it is more vulnerable to the occurrence of false deadlocks than centralized or hierarchical methods.

9.4.4 False Deadlocks

To handle the deadlock situation in distributed database systems, a number of messages are transmitted among the sites. On the other hand, both in centralized and in hierarchical deadlock detection methods, LWFGs are propagated periodically to one or more deadlock detector sites in the system. The periodic nature of this transmission process causes two problems. First, the delay that is associated with the transmission of messages that is necessary for deadlock detection can cause the detection of **false deadlocks**. For instance, consider that at a particular time the deadlock detector has received the information that the transaction T_i is waiting for the transaction T_j. Further assume that after some time the transaction T_j releases the data item requested by the transaction T_i and requests for data item that is being currently held by the transaction T_i. If the deadlock detector receives the information that the transaction T_j has requested for a data item that is held by the transaction T_i before receiving the information that the transaction T_i is not blocked by the transaction T_j any more, a false deadlock situation is detected.

Another problem is that a transaction T_i that blocks another transaction may be restarted for reasons that are not related to the deadlock detection. In this case, until the restart message of the transaction T_i is transmitted to the deadlock detector, the deadlock detector can find a cycle in the wait-for graph that includes the transaction T_i. Hence, a deadlock situation is detected by the deadlock detector, and this is called a **phantom deadlock**. When the deadlock detector detects a phantom deadlock, it may unnecessarily restart a transaction other than T_i. To avoid unnecessary restarts for phantom deadlocks, special safety measures are required.

CHAPTER SUMMARY

» In a database environment, **a deadlock** is a situation where transactions are waiting for one another indefinitely. Deadlock situations can be represented by **wait-for graphs,** directed graphs that indicate which transactions are waiting for which other transactions. The handling of deadlock situations is more complicated in a distributed DBMS than in a centralized DBMS, as it involves a number of sites. Three general techniques are available for deadlock resolution in a distributed database system: **distributed deadlock prevention, distributed deadlock avoidance and distributed deadlock detection and recovery from deadlock.**

» Distributed Deadlock Prevention – Distributed deadlock prevention is a cautious scheme in which a transaction is restarted when the distributed database system suspects that a deadlock might occur. Deadlock prevention approach may be preemptive or non-preemptive. Two different techniques are available for deadlock prevention based on transaction timestamp values: **Wait-die** and **Wound-Wait.**

» Distributed Deadlock Avoidance – Distributed deadlock avoidance is an alternative method in which each data item in the database system is numbered and each transaction requests locks on

these data items in that numeric order to avoid deadlock situations.

» Distributed Deadlock Detection and Recovery – In distributed deadlock detection and recovery method, first it is checked whether any deadlock situation has occurred in the distributed system. After the detection of a deadlock situation in the system, a victim transaction is chosen and aborted to resolve the deadlock situation. There are three different techniques for detecting deadlock situations in the distributed system: **Centralized deadlock detection, Hierarchical deadlock detection and Distributed deadlock detection.**

» To handle the deadlock situations in distributed database systems, a number of messages are transmitted periodically among the sites, which may cause two problems: **false deadlock** and **phantom deadlock**.

EXERCISES

Multiple Choice Questions

(i) Deadlock situation can be represented by
  a. Resource allocation graphs
  b. Weighted resource allocation graphs
  c. Wait-for graphs
  d. None of these.

(ii) Which of the following methods is best suited for deadlock handling in a distributed system?
  a. Deadlock prevention
  b. Deadlock avoidance
  c. Deadlock detection and recovery
  d. All of these.

(iii) Wait-die is a
  a. Non-preemptive deadlock avoidance algorithm
  b. Preemptive deadlock prevention algorithm
  c. Non-preemptive deadlock prevention algorithm
  d. Preemptive deadlock avoidance algorithm.

(iv) Wound-wait is a
  a. Non-preemptive deadlock avoidance algorithm
  b. Preemptive deadlock prevention algorithm
  c. Non-preemptive deadlock prevention algorithm
  d. Preemptive deadlock avoidance algorithm.

(v) In wait-die technique
  a. An older transaction is allowed to wait for a younger transaction
  b. A younger transaction is allowed to wait for an older transaction
  c. Both a. and b.
  d. None of the above.

(vi) In wound-wait technique
  a. An older transaction is allowed to wait for a younger transaction
  b. A Younger transaction is allowed to wait for an older transaction
  c. Both a. and b.
  d. None of the above.

(vii) Which of the following statements is true?
  a. In wait-die technique, the older transaction is restarted
  b. In wound-wait technique, the older transaction is restarted
  c. In both cases, younger transaction is restarted
  d. None of the above.

(viii) Which of the following is a deadlock avoidance technique?
  a. Wound-wait
  b. Wait-die
  c. Both a. and b.
  d. Preordering of resources.

(ix) A cycle in a LWFG indicates that
  a. There is the possibility of a local deadlock
  b. There is the possibility of a global deadlock
  c. A deadlock has occurred locally
  d. A deadlock has occurred globally.

(x) A cycle in a GWFG not involving an external node indicates
  a. There is the possibility of a local deadlock
  b. There is the possibility of a global deadlock
  c. A deadlock has occurred locally
  d. A deadlock has occurred globally.

(xi) A cycle in a GWFG involving an external node indicates
   a. There is the possibility of a local deadlock
   b. There is the possibility of a global deadlock
   c. A deadlock has occurred locally
   d. A deadlock has occurred globally.

(xii) A deadlock detector exists at each site of a distributed system
   a. In centralized deadlock detection
   b. In hierarchical deadlock detection
   c. In distributed deadlock detection
   d. Both in hierarchical and distributed deadlock detection.

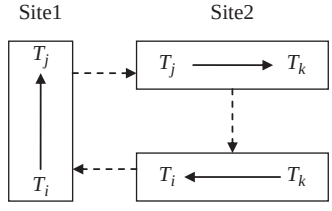
(xiii) More than one deadlock detector exists
   a. In centralized deadlock detection
   b. In hierarchical deadlock detection
   c. In distributed deadlock detection
   d. Both in hierarchical and distributed deadlock detection.

(xiv) Which of the following statements is correct?
   a. False deadlocks occur due to communication delay
   b. Phantom deadlocks occur due to false deadlocks
   c. All of the above
   d. None of these.

(xv) Which of the following deadlock detection methods is more vulnerable to the occurrence of false deadlocks?
   a. Centralized deadlock detection
   b. Hierarchical deadlock detection
   c. Distributed deadlock detection
   d. All of the above.

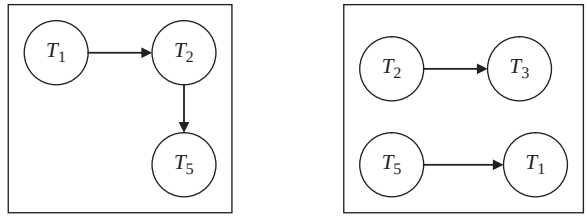
Review Questions

1. Discuss how you perform deadlock management in a distributed DBMS.
2. Explain the working principles of Wait-Die and Wound-Wait algorithms for deadlock prevention.
3. Differentiate between preemptive and non-preemptive methods for distributed deadlock prevention.
4. Explain the algorithm for distributed deadlock avoidance.
5. What is the additional threat involved in handling deadlock situations in a distributed DBMS compared to a centralized DBMS? Discuss the effect of replication in creating deadlocks.
6. Explain the phantom deadlock with an example. Describe and compare the centralized and distributed deadlock detection techniques for distributed DBMSs.
7. Compare and contrast centralized and hierarchical deadlock detection methods.
8. What is a false deadlock? Discuss with an example.
9. Which deadlock handling method is more suitable for deadlock management in a distributed system? Justify your answer.
10. Consider the global wait-for graph in the following figure. Explain whether any deadlock situation has occurred.



11. Consider the same transactions of exercise 1, which are executed with a deadlock prevention method. Prove that the deadlock is prevented using both the preemptive and the non-preemptive methods. Assume arbitrary transaction timestamp values and determine which transactions are restarted in each case.

12. Consider the LWFGs of two different sites in the following figure. Explain whether any deadlock has occurred globally.



Site 1 Site 2

13. Consider that the same transactions of exercise 3 are executed with a deadlock prevention method. Prove that the deadlock is prevented using both the preemptive and the non-preemptive methods. Assume arbitrary transaction timestamp values and determine which transactions are restarted in each case.

This page is intentionally left blank

10 Distributed Recovery Management

This chapter focuses on distributed recovery management. In the context of distributed recovery management, two terms are closely related to each other: reliability and availability. The concept of reliability and availability is clearly explained here. The chapter also introduces different types of failures that may occur in a distributed DBMS. Different distributed recovery protocols are also briefly represented, as well as the concepts of checkpointing and cold restart.

The organization of this chapter is as follows. Section 10.1 introduces the concept of availability and reliability. In Section 10.2 different types of failures that may occur in a distributed database environment are discussed. Section 10.3 represents the different steps that should be followed after a failure has occurred. In Section 10.4, local recovery protocols are described, and distributed recovery protocols are represented in Section 10.5. Network partitioning is focused on in Section 10.6.

10.1 Introduction to Recovery Management

Like in a centralized DBMS or in any other database systems, failures can occur in a distributed database system owing to a variety of reasons. To maintain the consistency of data, each transaction in a database environment must preserve the ACID property. To do this, it is the sole responsibility of a database management system to adopt some appropriate techniques. This complicated task is performed by a DBMS module, called **recovery manager**. The major objective of a recovery manager is to deploy an appropriate technique for recovering the system from failures and to preserve the database in the consistent state that existed prior to the failure.

In the context of distributed recovery management, two terms are often used: reliability and availability. Hence, it is necessary to differentiate between these two terms. **Reliability** refers to the probability that the system under consideration does not experience any failures in a given time period. In general, reliability is applicable for non-repairable (or non-recoverable) systems. However, the reliability of a system can be measured in several ways that are based on failures. To apply reliability to a distributed database, it is convenient to split the reliability problem into two separate parts, as it is very much associated with the specification of the desired behavior of the database: application-dependent part and application-independent part. The application-independent specification of reliability consists of requiring that transactions preserve their ACID property. The application-dependent part consists of requiring that transactions fulfill the system's general specifications, including the consistency constraints. The recovery manager only deals with the application-independent specification of reliability.

Availability refers to the probability that the system can continue its normal execution according to the specification at a given point in time in spite of failures. Failures may occur prior to the execution of a transaction and when they have all been repaired, the system must be available for

continuing its normal operations at that point of time. Thus, availability refers to both the resilience of the DBMS to various types of failures and its capability to recover from them. In contrast with reliability, availability is applicable for repairable systems. In distributed recovery management, reliability and availability are two related aspects. It is accepted that it is easier to develop highly available systems, but it is very difficult to develop highly reliable systems.

10.2 Failures in a Distributed Database System

To design a reliable system that can recover itself from failures, it is necessary to identify the types of failures the system has to deal with. There are various types of failures that may occur in a system, each of which needs to be dealt with in a different manner. A distributed database management system may suffer from all types of failures that may occur in a centralized DBMS. The following is the list of failures that may occur in a centralized DBMS.

- **Transaction failure** – There are two types of errors that may cause one or more transactions to fail: **application software errors** and **system errors**. A transaction can no longer continue owing to logical errors in the application software that is used for database accessing, such as bad input, data not found, overflow or resource limit exceeded. The system enters into an undesirable state (e.g., deadlock) resulting in the failure of one or more transactions, known as a system error.
- **System crash** – A hardware malfunction, or a bug in the software (either in the database software or in the operating system) may result in the loss of main memory, which forces the processing of transactions to halt.
- **Disk failure** – A disk block loses its contents as a result of either a head crash or an unreadable media or a failure during a data transfer operation. This also leads the processing of transactions to halt.

In a distributed DBMS, several different types of failures can occur in addition to the failures listed above for a centralized DBMS. These are site failure, communication link failure, loss of messages and network partition as listed in the following.

- **Site failure** – In a distributed DBMS, system failures are typically referred to as site failures, and it can occur owing to a variety of reasons as mentioned above. In this context, it is necessary to mention that system failures that result in the loss of the contents of the main memory are only considered here. The system failure may be total or partial. A **partial failure** indicates the failure of some sites in the distributed system while the other sites remain operational. A **total failure** indicates the simultaneous failure of all sites in a distributed system.
- **Loss of messages** – The loss of messages, or improperly ordered messages, is the responsibility of the underlying computer network protocol. It is assumed that these types of failures are handled by the data communication component of the distributed DBMS, so these are not considered here.
- **Communication link failure** – The performance of a distributed DBMS is highly dependent on the ability of all sites in the network to communicate reliably with one another. Although the communication technology has improved significantly, link failure can still occur in a distributed DBMS.

» **Network partition** – Owing to communication link failure, it may happen that the entire network gets split into two or more partitions, known as **network partition**. In this case, all sites in the same partition can communicate with each other, but cannot communicate with sites in the other partitions.

In some cases, it is very difficult to distinguish the type of failure that has occurred in the system. Generally, the system identifies that a failure has occurred and initiates the appropriate technique for recovery from the failure.

10.3 Steps Followed after a Failure

A distributed recovery manager must ensure the atomicity of global transactions; that is, it must ensure that all subtransactions of a transaction are either committed successfully or none of them are carried out at all. If a distributed DBMS detects that a site failure has occurred, then the following steps are to be followed for recovery.

» All transactions that are affected by the failure must be aborted.

» A message must be broadcast about the failure of the site so that no other site will try to access the failed site.

» A checking must be done periodically to see whether the failed site has recovered or not or, alternatively, must wait for the message from the failed site that it has recovered.

» On restart, the failed site must initiate a recovery procedure to abort all partial transactions that were active at the time of failure.

» After local recovery, the failed site must update its data to preserve the data consistency.

10.4 Local Recovery Protocols

In this section, the local recovery techniques that are used to handle system failures at each site are introduced. The other types of failures are not possible in this case; thus, the recovery protocols in a centralized DBMS are used here. The database is permanently stored on secondary storage, which is considered as stable storage in this context. These recovery protocols only deal with the failures that result in the loss of volatile storage. A page is the unit of data storage and access. As the size of the secondary storage is much greater than that of the main memory, pages are transferred from secondary storage to main memory on demand, although some pages are kept in the main memory to enhance access performance. The part of the database that is stored in main memory buffer is called **volatile database**. Typically, the buffer pages are allocated dynamically. The buffer (main memory space) is efficiently managed by using the page replacement algorithm. The **least-recently-used (LRU) algorithm** is the most commonly used page replacement algorithm.

The recovery protocols are classified into two different categories based on the method of executing the update operations: **in-place updating** and **out-place updating**. In in-place updating method, the update operation changes the values of data items in the stable storage, thereby losing the old values of data items. In the case of out-place updating method, the update operation does not change the values of data items in the stable storage, but stores the new values

separately. These updated values are integrated into the stable storage periodically. The **immediate modification technique** is a log-based in-place updating recovery protocol. **Shadow paging** is a non-log-based out-place updating recovery protocol. One in-place updating recovery protocol and one out-place updating recovery protocol are described in the next sections.

10.4.1 Immediate Modification Technique

In immediate modification recovery technique, an update operation changes the value of a data item in the stable storage resulting in the loss of the old value of the data. Therefore, if a failure occurs before the transaction committed successfully, the database will be in an inconsistent state. To facilitate the recovery of the database to a consistent state, it is necessary to store enough information about the database state changes. This information is stored in a **database log**. The contents of the database log may differ depending on the implementation, but the typical information that are kept in the database log for each transaction are **transaction identifier, type of log record** (transaction start, insert, update, delete, abort, commit), **identifier of the data item** (affected by the database action), **updated value of the data item** (called **after image**) and **old value of the data item** (called **before image**). Similar to the volatile database, the log information is also maintained in the main memory buffer (called log buffers) and written back to the stable storage. There are two different ways for writing log buffers into stable storage: synchronously (often known as forcing a log) and asynchronously. In the first method, the addition of a log record forces the database log to move from the volatile storage to the stable storage. In the latter method, the log is moved from the main memory to the stable storage at periodic intervals or when the main memory buffer is full.

The database log is used to protect against system failures in the following way.

- When a transaction starts, a **transaction start** record is written in the log.
- When a write operation is performed, a record is written into the log containing the necessary information.
- Once the log record is written, the update is performed in the stable storage.
- When a transaction commits, a **transaction commit** record is written into the log.

To facilitate the recovery procedure, it is essential that the log buffers are written into the stable storage before the corresponding write to the permanent database. This is known as **write-ahead log (WAL) protocol**. If updates were made to the database before the log record was written and if a failure occurred, the recovery manager would have no way of undoing the operations.

Under WAL protocol, using immediate modification technique, the recovery manager preserves the consistency of data in the following way, when failures occur.

- Any transaction that has both a transaction start and a transaction commit information in the log must be redone. The redo procedure performs all the writes to the database using the updated values of data items in the log records for the transaction, in the order in which they were written to the log. It is to be noted that the updated values that have already been written into the database have no effect, even though rewriting is unnecessary.
- Any transaction that has a transaction start record in the log but has no transaction commit record has to be undone. Using the old values of data items, the recovery manager preserves the database in the consistent state that existed before the start of the transaction. The undo operation is performed in the reverse order in which they were written to the log.

Example 10.1

Let us consider a banking transaction *T*1 that transfers $1,000 from account *A* to account *B*, where the balance of account *A* is $5,000 and the balance of account *B* is $2,500 before the transaction starts. Using immediate modification recovery technique, the following information will be written into the database log and into the permanent database to recover from a failure.

Main Memory	Database Log	Permanent Database
read(A, ai)	<T1, start>	A = 5000, B = 2500
ai = ai – 1000		
write(A, ai)	<T1, update, A, 5000, 4000>	A = 4000, B = 2500
read(B, bi)		
bi = bi + 1000		
write(B, bi)	<T1, update, B, 2500, 3500>	A = 4000, B = 3500
	<T1, commit>	

If the transaction *T*1 fails in between the log records <T1, start> and <T1, commit>, then it should be undone. On the other hand, if the database log contains both the records <T1, start> and <T1, commit> it should be redone.

10.4.2 Shadow Paging

Shadow paging is a non-log-based out-place updating recovery protocol in which duplicate stable-storage pages are maintained during the lifetime of a transaction. These are known as **current page** and **shadow page**. When the transaction starts, the two pages are identical. Whenever an update is made, the old stable-storage page, called the shadow page, is left intact for recovery purpose and a new page with the updated data item values is written into the stable-storage database, called the current page. During the transaction processing, the current page is used to record all updates to the database. When the transaction completes, the current page becomes the shadow page. If a system failure occurred before the successful completion of the transaction, the current page is considered as garbage.

Shadow-paging technique has several advantages over log-based immediate modification technique. It is significantly faster as there is no need for redo and undo operations. Shadow-paging technique reduces the overhead of maintaining and searching log records. The main disadvantage of shadow-paging technique is data fragmentation. Further, periodical garbage collection is required to make inaccessible pages accessible for further use.

10.4.3 Checkpointing and Cold Restart

Checkpointing technique is used to reduce the searching overhead of log-based recovery protocols. In a log-based recovery protocol, the entire log is searched to find out all the transactions that are need to be redone or undone. The searching overhead is reduced significantly by pointing a location in the database log that indicates that the database is up-to-date and consistent at that point. This process is known as **checkpointing**. In this case, the redo operation has to start from that point and undo operation has to go back to that point only. Generally, checkpoints are scheduled at predetermined intervals and involve the following operations.

- All log records in the main memory are written into secondary storage.
- All modified pages in the database buffer are written into secondary storage.
- A checkpoint record is written to the log. This record indicates the identifiers of all transactions that are active at the time of checkpoint.

To ensure the atomicity of the checkpointing operations, checkpointing is achieved in three steps [Gray, 1979] as described below.

- A begin_checkpoint record is written into the log.
- The checkpoint data are collected into the stable storage.
- An end_checkpoint record is written into the log.

If a system failure occurs during checkpointing, the recovery process will not find an end_checkpoint record and will conclude that checkpointing is not completed. When a system failure occurs, the database log is checked and all transactions that have committed since the end_checkpoint are redone, and all transactions that were active at the time of the crash are undone. In this case, the redo operation only needs to start from the end_checkpoint record in the log. Similarly, the undo operation can go in the reverse direction, starting from the end of the log and stopping at the end_checkpoint record.

The checkpointing technique described above is called **transaction-consistent checkpointing**. This technique is not the most efficient, as a significant delay is experienced by all transactions. There are many alternative checkpointing mechanisms such as action-consistent checkpointing, fuzzy checkpointing and others [Gray, 1979].

Some catastrophic failure may cause the loss of log information on stable storage, or loss of the stable database. In this case, it is not possible to reconstruct the current consistent state of the database. **Cold restart** technique is required to restore the database to a previous consistent state to maintain the consistency of data in case of loss of information on stable storage. In a distributed database system, the cold restart technique is much more complicated than in a centralized DBMS, because the state of the distributed database as a whole is to be restored to the previous consistent state. The previous consistent state is marked in a database log by a checkpoint. Hence, when the log information on stable storage is lost, it is not possible to restore the database to the previous consistent state using checkpointing.

Several techniques have been proposed for performing cold restarts in a distributed database, and most of them produce a high overhead for the system. One solution to handle this situation is to store an **archive copy** of both the database and the log on a different storage medium. Thus, the DBMS deals with three different levels of memory hierarchy. When loss-of-log failures occur, the database is recovered from the archive copy by redoing and undoing the transactions as stored in the archive log. The overhead of writing the entire database and the log information into a different storage is a significant one. The archiving activity can be performed concurrently with normal processing whenever any changes are made to the database. Each archive version contains only the changes that have occurred since the previous archiving.

Another technique for performing cold restart in a distributed database is to use local logs, local dumps and global checkpoint. A **global checkpoint** is a set of local checkpoints that are performed at all sites of the network and are synchronized by the following criterion:

- If a subtransaction of a transaction is contained in the local checkpoint at some site, then all other subtransactions of that transaction must be contained in the corresponding local checkpoints at other sites.

The reconstruction is relatively easier if global checkpoints are available. First, at the failed site the latest local checkpoint that can be consider as safe is determined, and this determines which earlier global state has to be reconstructed. Then all other sites are requested to re-establish the local states of the corresponding local checkpoints. The main difficulty with this approach is the recording of global checkpoints.

10.5 Distributed Recovery Protocols

The recovery in a distributed DBMS is more complicated than in a centralized DBMS, as it requires ensuring the atomicity of global transactions as well as of local transactions. Therefore, it is necessary to modify the commit and abort processing in a distributed DBMS, so that a global transaction does not commit or abort until all subtransactions of it have successfully committed or aborted. In addition, the distributed recovery protocols must have the capability to deal with different types of failures such as site failures, communication link failures and network partitions. Termination protocols are unique to distributed database systems. In a distributed DBMS, the execution of a global transaction may involve several sites, and if one of the participating sites fails during the execution of the global transaction, termination protocols are used to terminate the transaction at the other participating sites. Similarly, in the case of network partition, termination protocols are used to terminate the active transactions that are being executed at different partitions.

One desirable property for distributed recovery protocols is **independency**. An independent recovery protocol decides how to terminate a transaction that was executing at the time of a failure without consulting any other site. Moreover, the distributed recovery protocols should cater for different types of failures in a distributed system to ensure that the failure of one site does not affect processing at another site. In other words, operational sites should not be left blocked. Protocols that obey this property are known as **non-blocking protocols**. It is preferable that the termination protocols are non-blocking. A non-blocking termination protocol allows a transaction to terminate at the operational sites without waiting for recovery of the failed site.

The following section introduces two distributed recovery protocols: **two-phase commit (2PC) protocol** and **three-phase commit (3PC) protocol**, a non-blocking protocol. To discuss these protocols, the transaction model described in Chapter 7, Section 7.4 is considered.

10.5.1 Two-Phase Commit Protocol (2PC)

In 2PC protocol, there are two phases known as voting phase and decision phase. In the voting phase, the transaction coordinator asks all participants whether they agree to commit the transaction. If one participant votes to abort or fails to vote within a specified time period, the coordinator instructs all participants to abort the transaction. On the other hand, if all participants vote to commit within the time limit, the coordinator instructs all participants to commit the transaction. In this case, any site is free to abort a transaction independently at any time until it votes to commit. This type of abort is called **unilateral abort**. In the decision phase, a decision is made as to whether the transaction will be aborted or committed. A brief description of the 2PC protocol that does not consider failures as follows:

Phase 1 – The coordinator writes a begin_commit record in its log and sends a "prepare" message to all the participating sites. The coordinator waits for the participants to respond for a certain time interval (timeout period).

Phase 2 – When a participant receives a "prepare" message, it may return an "abort" vote or a "ready_commit" vote to the coordinator. If the participant returns an "Abort" vote, it writes an abort record into the local database log and waits for the coordinator for a specified time period. On the other hand, if the participant returns a "Ready_commit" vote, it writes a commit record into the corresponding database log and waits for the coordinator for a specified time interval. After receiving votes from participants, the coordinator decides whether the transaction will be committed or aborted. The transaction is aborted, if even one participant votes to abort or fails to vote within the specified time period. In this case, the coordinator writes an "abort" record in its log and sends a "global_abort" message to all participants and then waits for the acknowledgements for a specified time period. Similarly, the transaction is committed if the coordinator receives commit votes from all participants. Here, the coordinator writes a "commit" record in its log and sends a "global_commit" message to all participants. After sending the "global_commit" message to all participants, the coordinator waits for acknowledgements for a specified time interval. If a participating site fails to send the acknowledgement within that time limit, the coordinator resends the global decision until the acknowledgement is received. Once all acknowledgements have been received by the coordinator, it writes an end_transaction record in its log.

When a participating site receives a "global_abort" message from the coordinator, it writes an "abort" record it its log and after aborting the transaction sends an acknowledgement to the coordinator.

Similarly, when a participating site receives a "global_commit" message from the coordinator, it writes a "commit" record it its log and on completion sends an acknowledgement to the coordinator.

If a participant fails to receive a vote instruction from the coordinator, it simply times out and aborts. Therefore, a participant can abort a transaction before the voting. In 2PC, each participant has to wait for either the "global_commit" or the "global_abort" message from the coordinator. If the participant fails to receive the vote instruction from the coordinator, or the coordinator fails to receive the response from a participant, then it is assumed that a site failure has occurred and the termination protocol is invoked. The termination protocol is followed by the operational sites only. The failed sites follow the recovery protocol on restart. The 2PC protocol is illustrated in figure 10.1.

Termination protocols for 2PC

The termination protocol is invoked when the coordinator or a participant fails to receive a message within the time limit. The action that has to be taken in this situation depends on whether the coordinator or the participant has failed and when the timeout has occurred.

Coordinator The coordinator may timeout in three different states, namely, **wait, abort** and **commit.** Timeouts during the abort and the commit states are handled in the same manner; thus, only two cases are considered in the following.

(i) Timeout in the wait state – In this state, the coordinator is waiting for participants to vote whether they want to commit or abort the transaction and timeout occurs. In this situation, the coordinator cannot commit the transaction, as it has not received all votes. However, it can decide to globally abort the transaction. Hence, the coordinator writes an "abort" record in the log and sends a "global_abort" message to all participants.

(ii) Timeout in the commit or abort state – In this state, the coordinator is waiting for all participants to acknowledge whether they have successfully committed or aborted and timeout occurs. In this case, the coordinator resends the "global_commit" or "global_abort" message to participants that have not acknowledged.

Participant A participant may timeout in two different states. These are initial and ready as shown in figure 10.1.

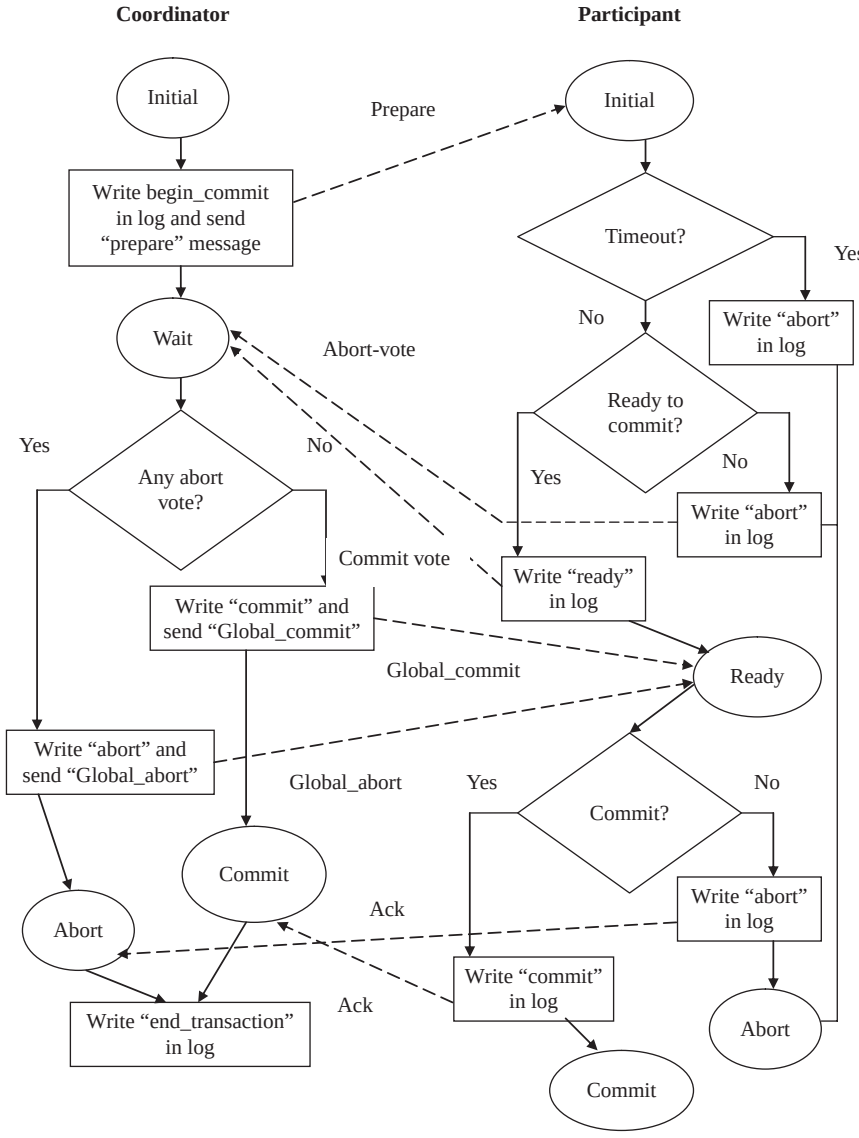
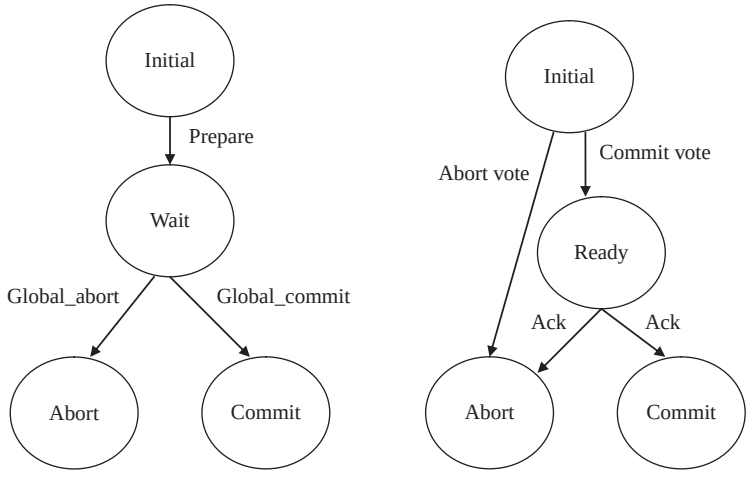


Fig. 10.1 Two-phase Commit Protocol

(i) Timeout in the initial state – In this state, the participant is waiting for a "prepare" message from the coordinator and timeout occurs, which indicates that the coordinator must have failed while in the initial state. Hence, the participant can unilaterally abort the transaction. If the participant subsequently receives a "prepare" message, either it can send an "abort" vote to the coordinator or it can simply ignore it.

(ii) Timeout in the ready state – In this state, the participant has voted to commit and is waiting for the "global_abort" or "global_commit" decision from the coordinator, and timeout occurs. In this case, the participant cannot unilaterally abort the transaction, as it had voted to commit. The participant also cannot commit the transaction, because it does not know the global decision. Hence, the participant is blocked until it can learn from someone the fate of the transaction. The participant could contact any of the other participants to know the fate of the transaction. This is known as **cooperative termination protocol**.

The cooperative termination protocol reduces the likelihood of blocking, but still blocking is possible. If the coordinator fails and all participants detect this as a result of executing the termination protocol, then they can elect a new coordinator and thereby resolve the blocking. The state transition diagram for 2PC is depicted in figure 10.2.



(a) Coordinator **(b) Participant**

Fig. 10.2 State Transition Diagram for 2PC

Recovery protocols for 2PC

Recovery protocols determine the actions to be taken by a failed site on recovery, to maintain the atomicity and the consistency property of transactions. The actions depend on the state of the coordinator or the participant at the time of failure.

Coordinator failure The following are the three possible cases of failure of the coordinator.

(i) Failure in initial state – Here, the coordinator has not yet started the commit procedure. Therefore, it will start the commit procedure on recovery.

(ii) Failure in wait state – In this case, the coordinator has sent the "prepare" message to participants. On recovery, the coordinator will again start the commit procedure from the beginning; thus, it will send the "prepare" message to all participants once more.

(iii) Failure in commit or abort state – In this case, the coordinator has sent the global decision to all participants. On restart, if the coordinator has received all acknowledgements, it can complete successfully. Otherwise, the coordinator will initiate the termination protocol.

Participant failure The objective of the recovery procedure for a participant is to ensure that it performs the same action as all other participants on restart, and this restart can be performed independently. The following are the three possible cases of failure of the participant.

(i) Failure in initial state – Here the participant has not yet voted to commit the transaction. Hence, the participant can unilaterally abort the transaction, because the participant has failed before sending the vote. In this case, the coordinator cannot make a global commit decision without this participant's vote.

(ii) Failure in ready state – In this case, the participant has sent its vote to the coordinator. On recovery, the participant will invoke the termination protocol.

(iii) Failure in commit or abort state – In this case, the participant has completed the transaction; thus, on restart no further action is necessary.

Communication schemes for 2PC

There are several communication schemes that can be employed for implementing 2PC protocol: **centralized 2PC, linear 2PC** and **distributed 2PC**. In centralized 2PC, all communications take place between the coordinator and the participant. The 2PC protocol described above is centralized 2PC. The communication scheme for centralized 2PC is illustrated in figure 10.3.

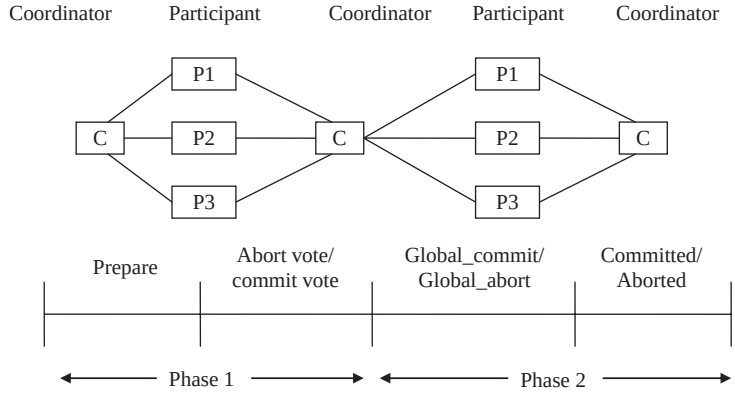


Fig. 10.3 Centralized 2PC Communication Scheme

In linear 2PC, participants can communicate with one another. In this communication scheme, an ordering is maintained among the sites in the distributed system. Let us consider that the sites are numbered as 1, 2, 3, … , n such that site number 1 is the coordinator and the others are participants. The 2PC is implemented by a forward chain of communication from the coordinator to the participant n in the voting phase and a backward chain of communication from participant

n to the coordinator in the decision phase. In the voting phase, the coordinator passes the voting instruction to site 2, site 2 votes and passes it to site 3, site 3 combines its vote and passes it to site 4 and so on. When the *n*th participant adds its vote, the global decision is obtained and it is passed backward to the participants, and eventually back to the coordinator. The linear 2PC reduces the number of messages compared to centralized 2PC, but does not provide any parallelism, thus, suffers from low response time performance. It is suitable for networks that do not have broadcasting capability [Bernstein et al., 1987]. The linear 2PC communication scheme is illustrated in figure 10.4.

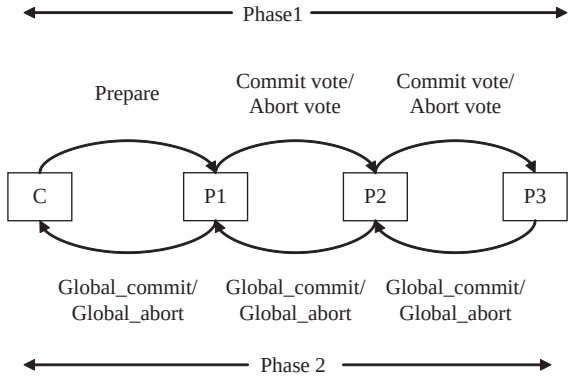


Fig. 10.4 Linear 2PC Communication Scheme

Another alternative popular communication scheme for 2PC protocol is distributed 2PC, in which all participants can communicate with each other directly. In this communication scheme, the coordinator sends "prepare" message to all participants in the voting phase. In the decision phase, each participant sends its decision to all other participants and waits for messages from all other participants. As the participants can reach a decision on their own, the distributed 2PC eliminates the requirement for the decision phase of the 2PC protocol. The distributed 2PC communication scheme is illustrated in figure 10.5.

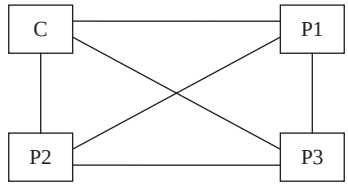


Fig. 10.5 Distributed 2PC Communication Scheme

10.5.2 Three-Phase Commit Protocol

The 3PC protocol [Skeen, 1981] is designed as a non-blocking protocol. In this context, it is necessary to mention that 2PC protocol is not a non-blocking protocol, as in certain circumstances it is possible for sites to become blocked. The coordinator can be blocked in the wait state, whereas

the participants can be blocked in the ready state. The requirements for the 3PC protocol are as follows:

- Network partition should not occur.
- All sites should not fail simultaneously, that is, at least one site must be available always.
- At the most *k* sites can fail simultaneously, where k is less than total number of sites in the distributed system.

In 3PC, a new phase is introduced, known as **precommit phase**, in between the voting phase and the global decision phase, for eliminating the uncertainty period for participants that have voted commit and are waiting for the global decision from the coordinator. In 3PC, if all participants vote for commit, the coordinator sends a "global precommit" message to all participants. A participant who has received a "precommit" message from the coordinator will definitely commit by itself, if it has not failed. Each participant acknowledges the receipt of "precommit" message to the coordinator and after receiving all acknowledgements the coordinator sends a "global commit" message to all participants. The 3PC protocol is illustrated in the figure 10.6.

Termination protocols for 3PC

Like in 2PC, the termination protocol is used in 3PC to handle timeouts.

Coordinator In 3PC, the coordinator may timeout in four different states: **wait, precommit, abort** and **commit**. Timeouts during the abort and the commit states are handled in the same manner as in 2PC, therefore, only three cases are considered here.

(i) Timeout in the wait state – The action taken here is identical to that in the coordinator timeout in the wait state for the 2PC protocol. In this state, the coordinator can decide to globally abort the transaction. Therefore, the coordinator writes an "abort" record in the log and sends a "global_abort" message to all participants.

(ii) Timeout in the precommit state – In this case, the coordinator does not know whether the non-responding participants have already moved to the precommit state or not, but the coordinator can decide to commit the transaction globally as all participants have voted to commit. Hence, the coordinator sends a "prepare-to-commit" message to all participants to move them into the commit state, and then globally commits the transaction by writing a commit record in the log and sending "global_commit" message to all participants.

(iii) Timeout in the commit or abort state – In this state, the coordinator is waiting for all participants to acknowledge whether they have successfully committed or aborted and timeout occurs. Hence, the participants are at least in the precommit state and can invoke the termination protocol as listed in case (ii) and case (iii) in the following section. Therefore, the coordinator is not required to take any special action in this case.

Participant A participant may timeout in three different states: initial, ready and precommit.

(i) Timeout in the initial state – In this case, the action taken is identical to that in the termination protocol of 2PC.

(ii) Timeout in the ready state – In this state, the participant has voted to commit and is waiting for the global decision from the coordinator. As the communication with the coordinator is lost, the termination protocol continues by electing a new coordinator (the election protocol

is discussed later). The new coordinator terminates the transaction by invoking a termination protocol.

(iii) Timeout in the precommit state – In this case, the participant has received the "prepare-to-commit" message and is waiting for the final "global_commit" message from the coordinator. This case is handled in the same way as described in case (ii) above.

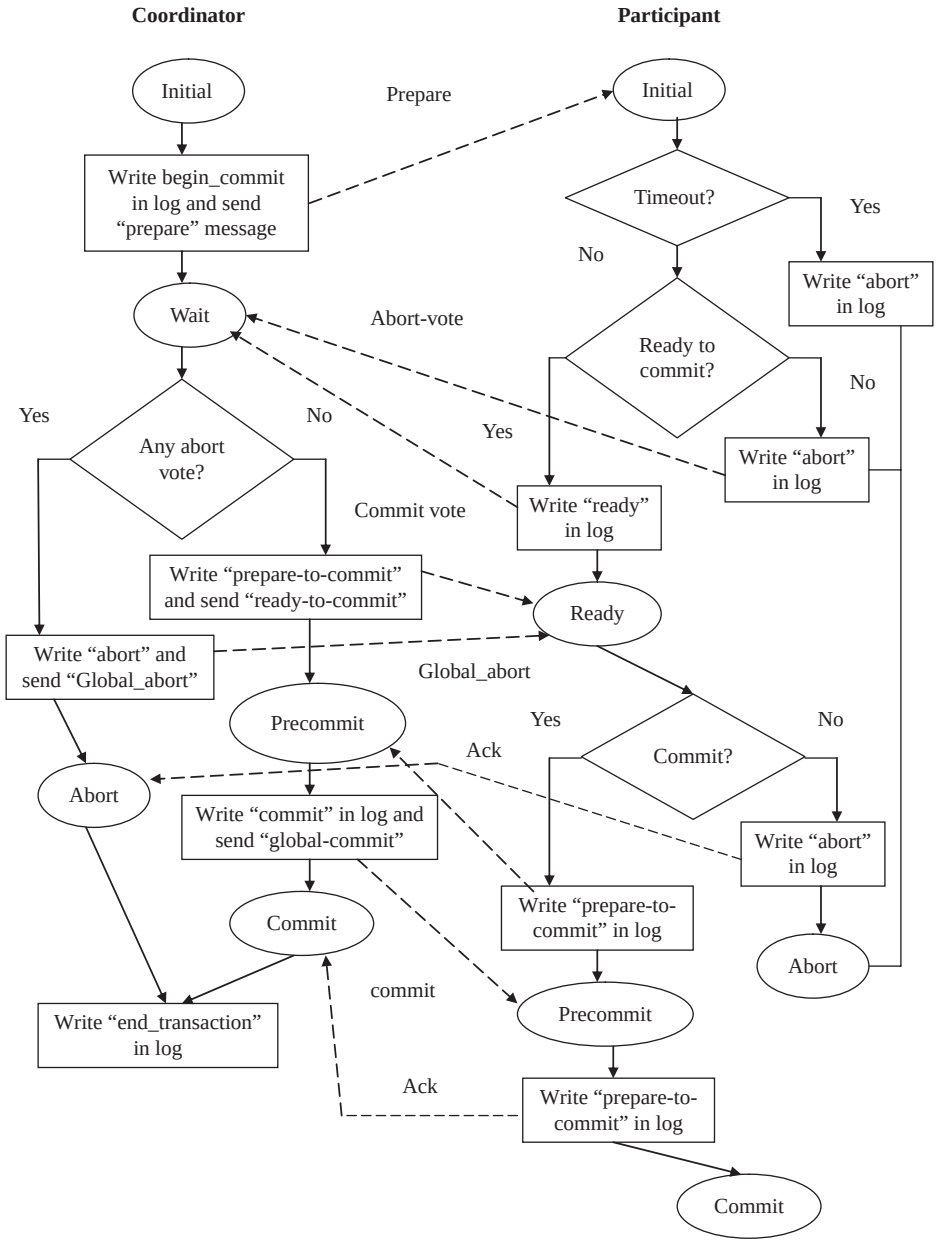


Fig. 10.6 Three-phase Commit Protocol

In the above two cases, the new coordinator terminates the transaction by invoking the termination protocol. The new coordinator does not keep track of participant failures during termination, but it simply guides operational sites for the termination of the transaction. The termination protocol for the new coordinator is described below.

- If the new coordinator is in the wait state, it will globally abort the transaction.
- If the new coordinator is in the precommit state, it will globally commit the transaction and send a "global_commit" message to all participants, as no participant is in the abort state.
- If the new coordinator is in the abort state, it will move all participants to the abort state.

The 3PC protocol is a non-blocking protocol, as the operational sites can properly terminate all ongoing transactions. The state transition diagram for 3PC is shown in figure 10.7.

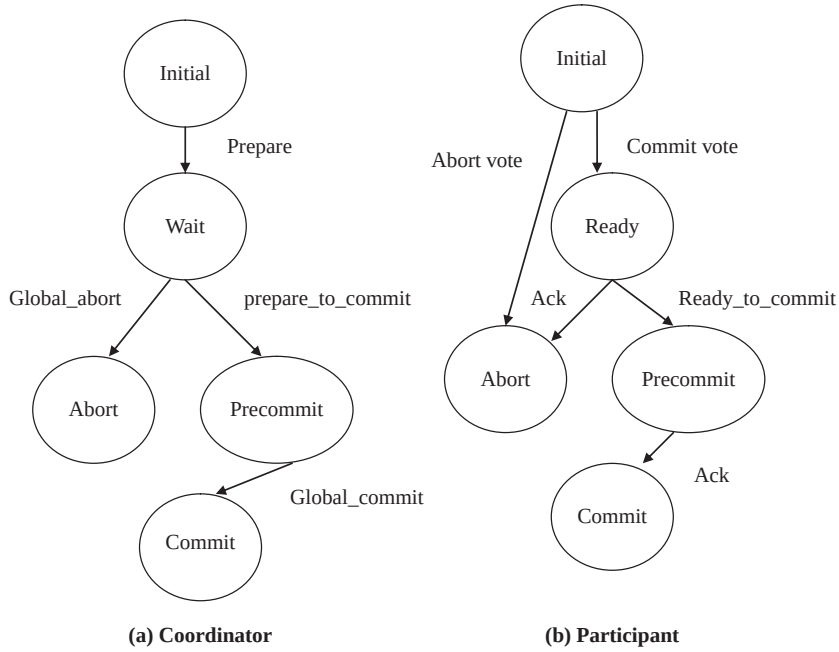


Fig. 10.7 State Transition Diagram for 3PC

Recovery protocols for 3PC

There are some minor differences between the recovery protocol for 3PC and that for 2PC. The differences only are mentioned here.

(i) Coordinator failure in wait state – In this case, the participants have already terminated the transaction during the coordinator failure. On recovery, the coordinator has to learn from other sites regarding the fate of the transaction.

(ii) Coordinator failure in precommit state – The participants have already terminated the transaction. As it is possible to move into the abort state from the precommit state during coordinator failure, on restart, the coordinator has to learn from other sites regarding the fate of the transaction.

(iii) Participant failure in precommit state – On recovery, the participant has to learn from other participants how they terminated the transaction.

Election protocol

Whenever the participants detect the failure of the coordinator, the election protocol is invoked to elect a new coordinator. Thus, one of the participating sites is elected as the new coordinator for terminating the ongoing transaction properly. Using linear ordering, the election protocol can be implemented in the following way. It is assumed here that each site *Si* has an order *i* in the sequence, the lowest being the coordinator, and each site knows the ordering and identification of other sites in the system. For electing a new coordinator, all operational participating sites are asked to send a message to other participants that have higher identification numbers. In this case, the site *Si* sends message to the sites S_{i+1}, S_{i+2}, S_{i+3} and so on. Whenever a participant receives the message from a lower-numbered participant, that participant stops sending messages. Eventually, each participant knows whether there is an operational site with a lower number or not. If not, then it becomes the new coordinator. If the elected new coordinator also times out during this process, then election protocol is again invoked.

This protocol is relatively efficient and most participants stop sending messages quite quickly. When a failed site with a lower identification number recovers it forces all higher-numbered sites to elect it as the new coordinator, regardless of whether there is a new coordinator or not.

10.6 Network Partition

Owing to communication link failures, network partition can occur in a distributed system. If the network is split into only two partitions, then it is called **simple partitioning**. On the other hand, if the network is split into more than two partitions, then it is called **multiple partitioning**. It is very difficult to maintain the consistency of the database in the case of network partition, as messages are lost. In the case of non-replicated databases, a transaction is allowed to proceed during network partitioning, if it does not require any data from a site outside the partition. Otherwise, the transaction must wait until the sites from which it requires data are available. In the case of replicated data, the procedure becomes much more complicated.

During network partitioning, processing involves a trade-off between availability and consistency [Davidson et al., 1984]. Data consistency is easily provided if no processing is allowed for the replicated data during network partitioning. On the other hand, availability is maximized if there are no restrictions on the processing of replicated data during network partitioning. It is possible to terminate a transaction properly during network partitioning with the help of a non-blocking termination protocol, but unfortunately, there is no such non-blocking termination protocol that can handle network partition. The recovery technique that is used to handle network partitioning depends on the particular concurrency control strategy being used, as recovery and concurrency control are closely related to each other. Thus, the recovery techniques that are used to handle network partition can be classified into two categories: **pessimistic** and **optimistic** as discussed in the next section.

10.6.1 Pessimistic Protocols

Pessimistic protocols emphasize on the consistency of the database, and therefore do not allow the processing of transactions during network partitioning if there is no guarantee that consistency can

be maintained. The termination protocols that deal with network partitioning in the case of non-replicated databases are pessimistic. To minimize blocking, it is convenient to allow the termination of the transaction by at least one group, possibly the largest group, during a network partition. However, during network partition, it is impossible for a group to determine whether it is the largest one or not. In this case, the termination protocol can use a pessimistic concurrency control algorithm such as **primary copy 2PL** or **majority locking** as described in Chapter 8, Section 8.5.

In the case of **primary copy 2PL**, for any transaction, only the partition that contains the primary copies of the data items can execute the transaction. Hence, the recovery of the network involves simply propagating all the updates to every other site. However, this method is vulnerable to the failure of the site that contains the primary copy. In many subnets, a primary site failure is more likely to occur than a partition; thus, this approach can increase blocking, instead of reducing it.

The above problem can be solved if **majority locking** technique is used. The basic idea of majority locking protocol is that before committing (or aborting) a transaction, a majority of sites must agree to commit (or abort) the transaction. The majority locking protocol cannot be applied with 2PC protocol, as it requires a specialized commit protocol.

Both of the above approaches are simple, and recovery using these approaches is very much straightforward. However, they require that each site is capable of differentiating network partitioning from site failures. Another straightforward generalization of the majority locking protocol, known as **quorum-based protocol** (or **weighted majority protocol**), can be used as a replica control method for replicated databases as well as a commit method to ensure transaction atomicity in the case of network partitioning. In the case of non-replicated databases the integration of the quorum-based protocols with commit protocol is necessary.

In quorum-based protocol, a weight is assigned to each site usually known as a vote. The basic rules for the quorum-based protocol are as follows:

- In the network, each site i is associated a weight or a vote Vi, which is a positive integer. Assume that V is the sum of the votes of all sites in the network.
- A transaction must collect a commit quorum Vc before committing the transaction.
- A transaction must collect an abort quorum Va before aborting the transaction.
- $Vc + Va > V$.

The last rule ensures that one transaction cannot be aborted and committed at the same time. The rules 2 and 3 ensure that the transaction has to obtain votes before termination.

The integration of 3PC protocol with quorum-based protocols requires a minor modification to the phase 3. To move from precommit state to commit state, it is necessary for the coordinator to obtain a commit quorum from the participants. It is not necessary to implement the rule 3 explicitly, because a transaction that is in the wait or ready state is always willing to abort the transaction; thus, an abort quorum already exists. When network partition occurs, the sites in each partition elect a new coordinator to terminate the transaction. If the newly elected coordinator fails, it is not possible to know whether a commit quorum or an abort quorum was reached. Therefore, it is necessary for the participants to take an explicit decision before joining either the commit quorum or the abort quorum; the votes cannot be changed afterwards. Since the wait and the ready states do not satisfy these requirements, one new state "preabort" is added between ready and abort states. The state transition diagram for the modified 3PC is illustrated in figure 10.8.

During network partition, a transaction can be terminated in the following way using quorum 3PC protocol.

- A new coordinator is elected in each partition.
- If at least one participant is in the commit state, the coordinator decides to commit the transaction and sends a "global_commit" message to all participants.
- If at least one participant is in the abort state, the coordinator decides to abort the transaction and sends a "global_abort" message to all participants.
- If a commit quorum is achieved by the votes of the participants in the precommit state, the coordinator decides to commit the transaction and sends a "global_commit" message to all participants.
- If an abort quorum is achieved by the votes of the participants in the precommit state, the coordinator decides to abort the transaction and sends a "global_abort" message to all participants.
- If case 3 is not satisfied but the sum of the votes of the participants in the precommit and ready states are enough to form a commit quorum, the coordinator sends a "prepare-to-commit" message to all participants to move them into precommit state and waits for case 3 to occur.
- If case 4 is not satisfied but the sum of the votes of the participants in the precommit and ready states are enough to form an abort quorum, the coordinator sends a "prepare-to-abort" message to all participants to move them into "preabort" state. The coordinator then waits for case 4 to occur.

The state transition diagram for quorum 3PC protocol is illustrated in figure 10.8. The quorum 3PC protocol is a blocking protocol, but it is capable of handling site failures as well as network partitioning.

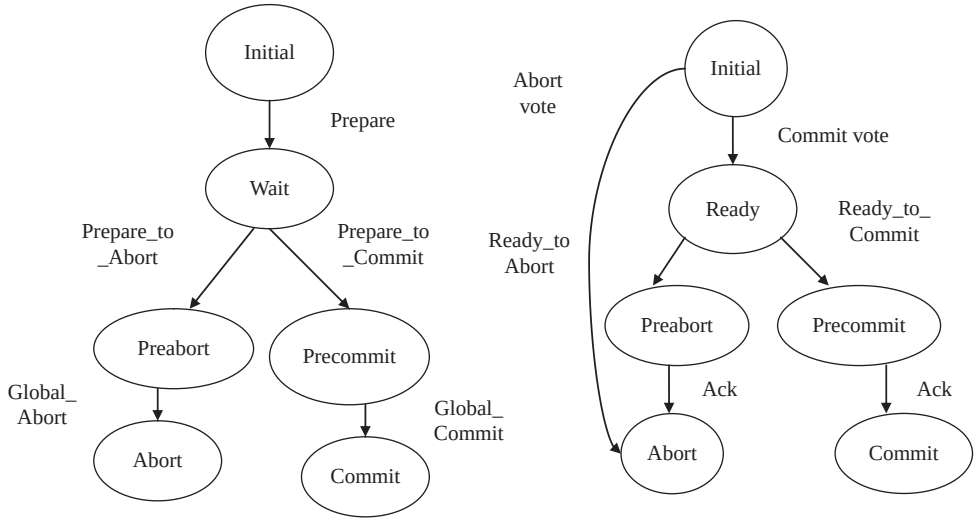


Fig. 10.8 State Transition Diagram for Quorum 3PC

10.6.2 Optimistic Protocols

Optimistic protocols emphasize on availability and use an optimistic concurrency control approach in which updates are allowed to proceed independently in the various partitions to terminate the transaction. Replication of data items improves availability. In this case, inconsistencies are likely when sites recover from network partitioning, but these inconsistencies can be eliminated later. The resolution of inconsistencies depends upon the semantics of the transactions.

One protocol for enforcing consistency of replicated data items is **ROWA (read-one-write-all) protocol** [described in **Chapter 8**]. There are various versions of ROWA protocol. In ROWA-A protocol, update is performed only on all available copies of the replicated data item, and unavailable copies are updated later. In this approach, the coordinator sends an update message to all sites where replicas of the data item reside and waits for update confirmation. If the coordinator times out before receiving acknowledgements from all sites, it is assumed that the non-responding sites are unavailable and the coordinator continues the update with available sites to terminate the transaction. There are two problems with this approach. It this approach, it may so happen that one site containing the replica of a data item has already performed the update and sent the acknowledgement, but as the coordinator has not received it within the specified time interval, the coordinator has considered the site as unavailable. Another problem is that some sites might have been unavailable when the transaction started, but might have recovered since then and might have started executing transactions. Therefore, the coordinator must perform a validation before committing. To perform this validation, the coordinator sends an "inquiry" message to all sites. If the coordinator gets a reply from a site that was unavailable previously (that is, the coordinator had not received any acknowledgement from this site previously), the coordinator terminates the transaction by aborting it; otherwise, the coordinator proceed to commit the transaction.

Another replica control protocol based on voting is **quorum-consensus protocol**, which is described in Chapter 8, Section 8.5.6. The basic idea of this protocol is that each read and write operation has to obtain a sufficient number of votes to commit the transaction. This protocol ensures that a data item cannot be read and written by two transactions concurrently, thereby preserving consistency. Similarly, it also ensures that two transactions cannot perform write operation on a data item concurrently, thereby maintaining serializability. During network partitioning, the quorum-consensus protocol determines which transactions are going to terminate, depending on the votes they obtained. This protocol also ensures that two different transactions that are initiated at two different partitions and access the same data item cannot terminate simultaneously.

The difficulty with the quorum-consensus protocol is that transactions are required to obtain a quorum even to read data. This significantly and unnecessarily slows read access to the database.

CHAPTER SUMMARY

- The major objective of a recovery manager is to employ an appropriate technique for recovering the system from a failure, and to preserve the database in the consistent state that existed prior to the failure.
- **Reliability refers** to the probability that the system under consideration does not experience any failures in a given time period.
- **Availability** refers to the probability that the system can continue its normal execution according to the specification at a given point in time in spite of failures.
- There are different types of failures that may occur in a distributed database environment such as site failure, link failure, loss of message and network partition.

» Two distributed recovery protocols are 2PC protocol and 3PC protocol. 2PC is a blocking protocol whereas 3PC is a non-blocking protocol.

» The termination protocols that are used to handle network partition can be classified into two categories: pessimistic protocols and optimistic protocols. In the case of non-replicated databases, the termination protocols that are used to deal with network partition are pessimistic. In the case of replicated databases, both types of termination protocols can be used.

EXERCISES

Multiple Choice Questions

(i) Which of the following is the probability that the system is up and running at a certain point in time?
   a. Reliability
   b. Availability
   c. Maintainability
   d. None of these.

(ii) Which of the following failures does not occur in a distributed system?
   a. Disk failure
   b. Link failure
   c. Message loss
   d. None of these.

(iii) Immediate modification is a
   a. In-place updating recovery technique
   b. Out-place updating recovery technique
   c. Both in-place and out-place updating recovery technique
   d. None of these.

(iv) Shadow paging is a
   a. Out-place updating recovery technique
   b. In-place updating recovery technique
   c. Both in-place and out-place updating recovery technique
   d. None of these.

(v) In immediate modification technique, if the transaction fails in between the log records <T1, start> and <T1, commit>, then it is
   a. Redone
   b. Undone
   c. Committed
   d. Precommitted.

(vi) Which of the following statements is correct?
   a. In WAL protocol, log buffers are written into the stable storage before the corresponding write to the permanent database.
   b. In WAL protocol, log buffers are written into the stable storage after the corresponding write to the permanent database.
   c. WAL protocol is used in both log-based and non-log-based recovery protocols.
   d. None of these.

(vii) Which of the following recovery technique requires undo and redo operations?
   a. Log-based recovery
   b. Non-log-based recovery
   c. Both log-based and non-log-based recovery
   d. None of these.

(viii) Data fragmentation can occur in
   a. Immediate modification technique
   b. Deferred modification technique
   c. Shadow paging technique
   d. Checkpointing technique.

(ix) Which of the following techniques is used when the information on stable storage is lost?
   a. Shadow paging
   b. Checkpointing
   c. Cold restart
   d. None of these.

(x) Which of the following is the recovery management technique used in distributed databases?
   a. Deferred update
   b. Immediate update
   c. Two-phase commit
   d. All of these.

(xi) Which of the following strategies is used to ensure that either all physical databases in a distributed system are updated or none at all?
   a. Two-phase commit
   b. Two-phase locking

c. Two-phase update
d. Update propagation

(xii) Which of the following is not a part of buffer replacement strategy?
a. Stable blocks
b. Pinned blocks
c. Output blocks
d. Forced output of blocks.

(xiii) Unilateral abort is
a. The participant aborting a transaction after sending a commit vote
b. Aborting a transaction immediately at any time until a participant votes to commit
c. Aborting a transaction at any time by the coordinator
d. None of these.

(xiv) Which of the following is a desirable property for distributed recovery protocols?
a. Blocking
b. Independency
c. Both of these
d. None of these.

(xv) Termination protocol is invoked
a. By the participant if it fails to receive the instruction from the coordinator
b. By the coordinator if it fails to receive the response from a participant
c. Both in the above cases
d. In none of the above cases.

(xvi) The recovery protocol is invoked
a. By all participants after a failure has occurred
b. By the coordinator after a failure has occurred
c. By a failed site on restart
d. None of these.

(xvii) Which of the following phases does not belong to 3PC protocol?
a. Precommit phase
b. Voting phase
c. Decision phase
d. None of the above.

(xviii) Which of the following statements is incorrect?
a. Pessimistic protocols emphasize on the consistency of the database
b. Optimistic protocols emphasis on the availability of the database
c. Primary copy 2PL and majority locking are pessimistic protocols
d. None of these.

(xix) In quorum 3PC, which of the following additional state is added?
a. Preabort
b. Precommit
c. Both preabort and precommit
d. Preinitial.

(xx) Quorum 3PC is used to handle
a. Site failure
b. Network partition
c. Both site failure and network partition
d. None of these.

Review Questions

1. Differentiate between availability and reliability.
2. Write down the possible types of failures in a distributed system. Explain how 2PC protocol ensures transaction atomicity for each of these possible failures.
3. What is checkpointing? Explain how checkpointing reduces the overhead of log-based recovery.
4. Describe shadow paging technique. How is it different from immediate modification technique?
5. Compare blocking and non-blocking protocols.
6. Describe two-phase commit protocol. What are the demerits of this protocol?
7. Discuss the different communication structures for 2PC.
8. Describe three-phase commit protocol.
9. "Three-phase is a non-blocking protocol". Justify the statement.
10. What are the pros and cons of 3PC protocol relative to 2PC protocol.

11. Comment on the following:

    "Cold restart is very hard in distributed databases".
    "3PC protocol overcomes the limitations of 2PC protocol".

12. Describe election protocol. When and why is it used?
13. Differentiate between pessimistic and optimistic termination protocols.
14. Explain how network partitioning is handled in replicated databases.
15. Design linear three-phase commit (3PC) protocol.
16. Draw the state transition diagram for quorum 3PC protocol.
17. Consider a distributed system that uses 2PC protocol. Define the log records that will be maintained by each participant for executing a global transaction.
18. Repeat the exercise 3 assuming that the distributed system uses 3PC protocol.
19. Repeat the exercise 3 assuming that the distributed system uses quorum 3PC protocol.
20. Consider that a data item p is replicated at all sites of a four-site network. Assign votes in two different ways to the sites and assign in each case a commit and an abort quorum. For each possible network partition and for each one of the two vote assignments, construct a table in which group of sites participating in a transaction that reads and writes p can be terminated and indicate whether the transaction is aborted or committed.

11 Distributed Query Processing

This chapter introduces the basic concepts of distributed query processing. A query involves the retrieval of data from the database. In this chapter, different phases and sub phases of distributed query processing are briefly discussed with suitable examples. An important phase in distributed query processing is distributed query optimization. This chapter focuses on different query optimization strategies, distributed cost model and cardinalities of intermediate result relations. Finally, efficient algorithms for centralized and distributed query optimization are also presented in this chapter.

The organization of this chapter is as follows. Section 11.1 introduces the fundamentals of query processing, and the objectives of distributed query processing are presented in Section 11.2. The different steps for distributed query processing such as query transformation, query fragmentation, global query optimization and local query optimization are briefly discussed in Section 11.3. In Section 11.4, join strategies in fragmented relations are illustrated. The algorithms for global query optimization are represented in Section 11.5.

11.1 Concepts of Query Processing

The amount of data handled by a database management system increases continuously, and it is no longer unusual for a DBMS to manage data sizes of several hundred gigabytes to terabytes. In this context, a critical challenge in a database environment is to develop an efficient query processing technique, which involves the retrieval of data from the database. A significant amount of research has been dedicated to develop highly efficient algorithms for processing queries in different databases, but here the attention is restricted on relational databases. There are many ways in which a complex query can be executed, but the main objective of query processing is to determine which one is the most cost-effective. This complicated task is performed by a DBMS module, called **query processor**. In query processing, the database users generally specify what data is required rather than the procedure to follow to retrieve the required data. Thus, an important aspect of query processing is query optimization. A query typically has many possible execution strategies, and the process of choosing a suitable one for processing a query is known as **query optimization**.

A query is expressed by using a high-level language such as SQL (Structured Query Language) in a relational data model. The main function of a relational query processor is to transform a high-level query into an equivalent lower-level query (relational algebra), and the transformation must achieve both correctness and efficiency. The lower-level query actually implements the execution strategy for the given query. In a centralized DBMS, query processing can be divided into four main steps: **query decomposition** (consisting of scanning, parsing and validation), **query optimization, code generation** and **query execution**. In the query decomposition step, the query parser checks

the validity of the query and then translates it into an internal form, usually a relational algebraic expression or something equivalent. The query optimizer examines all relational algebraic expressions that are equivalent to the given query and chooses the optimum one that is estimated to be the cheapest. The code generator generates the code for the access plan selected by the query optimizer and the query processor actually executes the query. The steps of query processing in centralized DBMS are illustrated in figure 11.1.

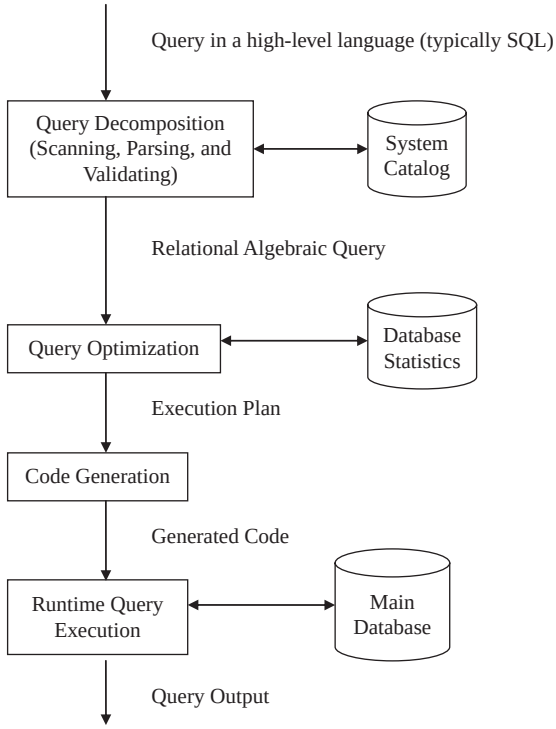


Fig. 11.1 Steps for Query Processing in Centralized DBMS

Query optimization is a critical performance issue in a centralized database management system, and the main challenge is to choose one execution strategy that minimizes the consumption of computing resources. Another objective of query optimization is to reduce the total execution time of the query, which is the sum of the execution times of all individual operations that make up the query. There are two main techniques for query optimization in a centralized DBMS, although the two methods are usually combined in practice. The first approach uses **heuristic rules** that order the operations in a query, whereas the second approach compares different strategies based on their relative costs and selects the one that requires minimum computing resources. In a distributed system, several additional factors further complicate the process of query execution. In general, the relations are fragmented in a distributed database; therefore, the distributed query processor transforms a high-level query on a logically distributed database (entire relation) into a sequence of database operations (relational algebra) on relation fragments. The data accessed by the query in a distributed database must be localized so that the database operations can be performed on local data or relation fragments. Finally, the query on fragments must be extended with communication

operations, and optimization should be done with respect to a cost function that minimizes the use of computing resources such as disk I/Os, CPUs and communication networks.

Example 11.1

Let us consider the following two relations which are stored in a centralized DBMS

Employee (empid, ename, salary, designation, deptno)
Department (deptno, dname, location)

and the following query:

"Retrieve the names of all employees whose department location is 'inside' the campus"

where, **empid** and **deptno** are the primary keys for the relations Employee and Department respectively and **deptno** is a foreign key of the relation Employee.
Using SQL, the above query can be expressed as:

Select ename from Employee, Department where Employee.deptno = Department.deptno and location = "inside".

Two equivalent relational algebraic expressions that correspond to the above SQL statement are as follows:

(i) $\prod_{\text{ename}} (\sigma_{\text{(location = "inside")} \wedge \text{(Employee.deptno = Department.deptno)}} (\text{Employee} \times \text{Department}))$

(ii) $\prod_{\text{ename}} (\text{Employee}_{\text{Employee.deptno = Department.deptno}} (\sigma_{\text{location = "inside"}}(\text{Department})))$

In the first relational algebraic expression the projection and the selection operations have been performed after calculating the Cartesian product of two relations Employee and Department, whereas in the second expression the Cartesian Product has been performed after performing the selection and the projection operation from individual relations. Obviously, the use of computing resources is lesser in the second expression. Thus, in a centralized DBMS, it is easier to choose the optimum execution strategy based on a number of relational algebraic expressions that are equivalent to the same query. In a distributed context, the query processing is significantly more difficult because the choice of optimum execution strategy depends on some other factors such as data transfer among sites and the selection of the best site for query execution. The problem of distributed query processing is discussed below with the help of an example.

Example 11.2

Let us consider the same example 11.1 in a distributed database environment where the Employee and Department relations are fragmented and stored at different sites. For simplicity, let us assume that the Employee relation is horizontally fragmented into two partitions EMP_1 and EMP_2, which are stored at site1 and site2 respectively, and the Department relation is horizontally fragmented into two relations $DEPT_1$ and $DEPT_2$, which are stored in site3 and site4 respectively as listed below.

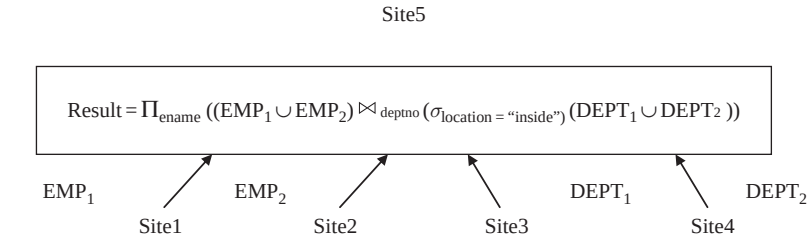
$\mathbf{EMP_1 = \sigma_{deptno <= 10}}$ **(Employee)**

$\mathbf{EMP_2 = \sigma_{deptno > 10}}$ **(Employee)**

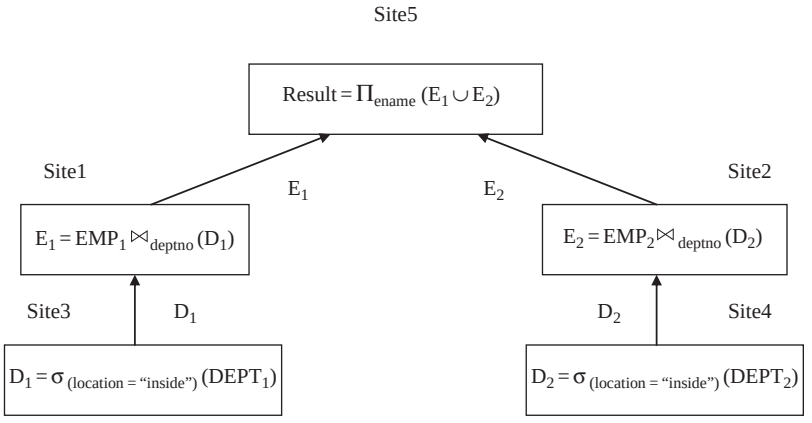
$\mathbf{DEPT_1 = \sigma_{deptno <= 10}}$ **(Department)**

$\mathbf{DEPT_2 = \sigma_{deptno > 10}}$ **(Department)**

Further assume that the above query is generated at Site5 and the result is required at that site. Two different strategies to execute the query in the distributed environment are depicted in figure 11.2.



(i) Strategy A



(ii) Strategy B

Fig. 11.2 Two Equivalent Distributed Query Execution Strategies

In the first strategy, all data are transferred to Site5 before processing the query. In the second strategy, selection operations are performed individually on the fragmented relations $DEPT_1$ and $DEPT_2$ at Site3 and Site4 respectively, and then the resultant data D_1 and D_2 are transferred to Site1 and Site2 respectively. After evaluating the join operations D_1 with EMP_1 at Site1 and D_2 with EMP_2 at Site2, the resultant data are transmitted to Site5 and the final projection operation is performed.

To calculate the costs of the above two different execution strategies, let us assume that the cost of accessing a tuple from any relation is 1 unit, and the cost of transferring a tuple between any two sites is 10 units. Further consider that the number of tuples in Employee and Department relations are 1,000 and 20 respectively, among which the location of 8 departments are inside the campus. For the sake of simplicity, assume that the tuples are uniformly distributed among sites, and the relations Employee and Department are locally clustered on attributes "deptno" and "location" respectively.

The total cost for the strategy *A* is as follows:

(i) The cost of transferring 10 tuples of $DEPT_1$ from Site3 and 10 tuples of $DEPT_2$ from Site4 to Site5 = (10 + 10) * 10 = 200

(ii) The cost of transferring 500 tuples of EMP_1 from Site1 and 500 tuples of EMP_2 from Site2 to Site5 = (500 + 500) * 10 = 10,000

(iii) The cost of producing selection operations from $DEPT_1$ and $DEPT_2$ = 20 * 1 = 20

(iv) The cost of performing join operation of EMP_1 and EMP_2 with resultant selected data from Department relation = 1,000 * 8 * 1 = 8,000

(v) The cost of performing projection operation at Site5 to retrieve the employee names = 8 * 1 = 8

In this case, the total cost is = 18,228.
The total cost for the strategy *B* is listed in the following:

(i) The cost of producing D_1 from $DEPT_1$ and D_2 from $DEPT_2$ = (4 + 4) * 1 = 8

(ii) The cost for transferring 4 tuples of D_1 from Site3 to Site1 and 4 tuples of D_2 from Site4 to Site2 = (4 + 4) * 10 = 80

(iii) The cost of producing E_1 at Site1 and E_2 at Site2 = (4 + 4) * 1 * 2 = 16

(iv) The cost for transferring 4 tuples of E_1 from Site1 to Site5 and 4 tuples of E_2 from Site2 to Site5 = (4 + 4) * 10 = 80

(v) The cost for evaluating projection operation at Site5 to retrieve the employee names = 8 * 1 = 8

Hence, the total cost is = 192.

The cost of performing the projection operation and the selection operation on a tuple is same because in both cases it is equal to the cost of accessing a tuple. Obviously, the execution strategy *B* is much more cost-beneficial than the strategy *A*. Furthermore, the slower communication between the sites and the higher degree of fragmentation may increase the cost difference between the alternative query processing/execution strategies.

11.2 Objectives of Distributed Query Processing

Distributed query processing involves the retrieving of data from physically distributed databases that provide the view of a single logical database to users. It has a number of objectives as listed below.

- The major objective of distributed query processing is to translate a high-level language query on a single logically distributed database (as seen by the users) into low-level language queries on physically distributed local databases.
- There are a number of alternative execution strategies for processing a distributed query. Thus, another important objective of distributed query processing is to select an efficient execution strategy for the execution of a distributed query that minimizes the consumption of computing resources.
- In distributed query processing, the **total cost** for executing a distributed query should be minimized. If no relation is fragmented in a distributed DBMS, the query execution involves only a local processing cost. On the other hand, if relations are fragmented, a communication cost is incurred in addition to the local processing cost. In this case, the aim of distributed query processing is to minimize the total execution cost of the query,

which includes the total processing cost (sum of all local processing costs incurred at the participating sites) of the query and the communication cost. The local processing cost of a distributed query is evaluated in terms of I/O cost (the number of disk accesses) and CPU cost. The CPU cost is incurred when performing data operations in the main memory of the participating sites. The I/O cost can be minimized by using efficient buffer management techniques. In distributed query execution, the communication cost is incurred for the exchange of data between participating sites. Hence, the communication cost depends on several factors such as the amount of data transfer between participating sites, the selection of best site for query execution, the number of message transfers between the participating sites, and the communication network. In the case of high-speed wide area networks (with a bandwidth of a few kilobytes per second), the communication cost is the dominant factor, and the optimization of CPU cost and I/O cost can be ignored in such cases. The optimization of local processing cost is of greater significance in the case of local networks.

» In distributed query processing, one approach to query optimization is to minimize **the total cost of time** required to execute a query. Another alternative approach for query optimization is to reduce **the response time** of a distributed query, which is the time elapsed between the initiation of the query and producing the answer of the query. Therefore, another important objective of distributed query processing is to maximize the parallel execution of operations of a given distributed query so that the response time is significantly less than the total cost time.

11.3 Phases in Distributed Query Processing

In a distributed database system, data distribution details are hidden from the users; thus, it provides data distribution transparency. The distributed query processor is responsible for transforming a query on global relations into a number of subqueries on fragmented relations at local databases. Therefore, distributed query processing is more complicated than query processing on centralized DBMS, and it requires some extra steps in addition to all the steps of a centralized query processing technique. Whenever a distributed query is generated at any site of a distributed system, it follows a sequence of phases to retrieve the answer of the query. These are **query decomposition, query fragmentation, global query optimization** and **local query optimization** as illustrated in figure 11.3. The first three phases in distributed query processing are controlled by the site where the query is generated. The local query optimization is done at the participating local sites, and all these optimized local queries are executed on local databases. Finally, the results of local queries are sent back to the site of the distributed system where the query is generated. The details of all these phases are described in the next section.

11.3.1 Query Decomposition

Query decomposition is the first phase in distributed query processing. The objective of this phase is to transform a query in a high-level language on global relations into a relational algebraic query on global relations. The information required for the transformation is available in the global conceptual schema describing the global relations. In this phase, the syntactical and semantic correctness of the query is also checked. In the context of both centralized DBMS and distributed DBMS, the query decomposition phase is the same. The four successive steps of query decomposition are

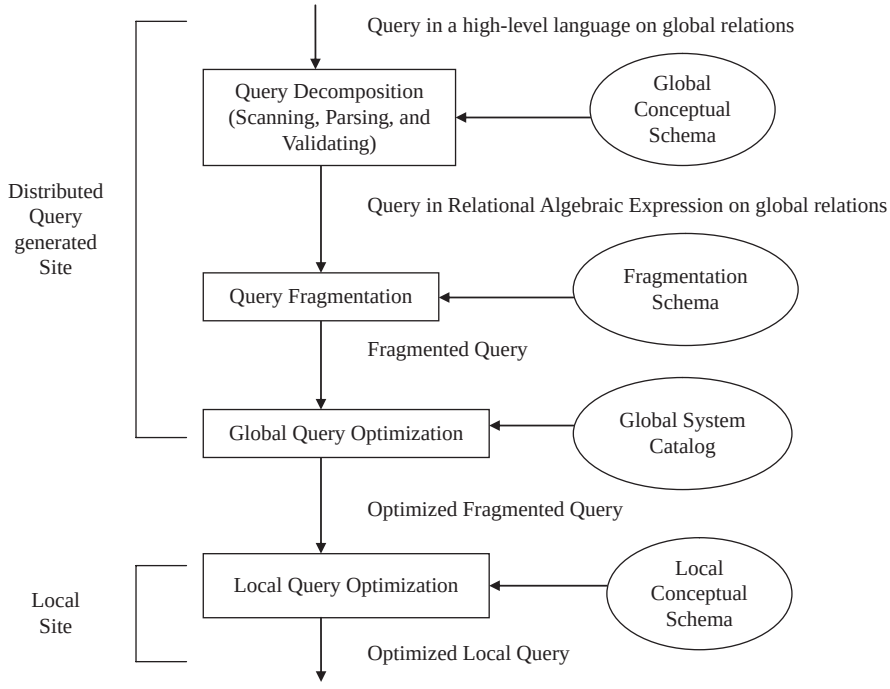


Fig. 11.3 Phases in Distributed Query Processing

normalization, analysis, simplification and **query restructuring,** which are briefly discussed in the following sections.

Normalization

In normalization step, the query is converted into a normalized form to facilitate further processing in an easier way. A given query can be arbitrarily complex depending on the predicates (WHERE clause in SQL) specified in the query. In this step, the complex query is generally converted into one of the two possible normal forms by using a few transformation rules, which are listed below.

- $P_1 \wedge P_2 \Leftrightarrow P_2 \wedge P_1$
- $P_1 \vee P_2 \Leftrightarrow P_2 \vee P_1$
- $P_1 \wedge (P_2 \wedge P_3) \Leftrightarrow (P_1 \wedge P_2) \wedge P_3$
- $P_1 \vee (P_2 \vee P_3) \Leftrightarrow (P_1 \vee P_2) \vee P_3$
- $P_1 \wedge (P_2 \vee P_3) \Leftrightarrow (P_1 \wedge P_2) \vee (P_1 \wedge P_3)$
- $P_1 \vee (P_2 \wedge P_3) \Leftrightarrow (P_1 \vee P_2) \wedge (P_1 \vee P_3)$
- $\neg(P_1 \wedge P_2) \Leftrightarrow \neg P_1 \vee \neg P_2$
- $\neg(P_1 \vee P_2) \Leftrightarrow \neg P_1 \wedge \neg P_2$
- $\neg(\neg P_1) \Leftrightarrow P_1$

where P_i represents a simple predicate specified in the query.

The two possible normal forms are **conjunctive normal form** and **disjunctive normal form**.

Conjunctive Normal Form: In conjunctive normal form, preference is given to the AND ($\wedge$ predicate) operator, and it is a sequence of conjunctions that are connected by the $\wedge$ (AND) operator. Each conjunction contains one or more terms connected by the $\vee$ (OR) operator. For instance,

$$(P_1 \vee P_2 \vee P_3) \wedge (P_4 \vee P_5 \vee P_6) \wedge \ldots \wedge (P_{n-2} \vee P_{n-1} \vee P_n),$$

where P_i represents a simple predicate, is in conjunctive normal form.

Disjunctive Normal Form: In disjunctive normal form, preference is given to the OR ($\vee$ predicate) operator, and it is a sequence of disjunctions that are connected with the $\vee$ (OR) operator. Each disjunction contains one or more terms connected by the $\wedge$ (AND) operator. For example,

$$(P_1 \wedge P_2 \wedge P_3) \vee (P_4 \wedge P_5 \wedge P_6) \vee \ldots, \vee (P_{n-2} \wedge P_{n-1} \wedge P_n),$$

where P_i represents a simple predicate, is in disjunctive normal form. In this normal form, a query can be processed as independent conjunctive subqueries connected by union operations.

Example 11.3

Let us consider the following two relations stored in a distributed database:

Employee (<u>empid</u>, ename, salary, designation, deptno)
Department (<u>deptno</u>, dname, location)

and the following query:

"Retrieve the names of all employees whose designation is 'Manager' and department name is 'Production' or 'Printing'".

In SQL, the above query can be represented as

Select ename from Employee, Department where designation = "Manager" and Employee. deptno = Department.deptno and dname = "Production" or dname = "Printing".

The conjunctive normal form of this query is as follows:

designation = "Manager"∧Employee.deptno = Department.deptno∧(dname = "Production" ∕dname = "Printing")

The disjunctive normal form of the same query is

(designation = "Manager"∧ Employee.deptno = Department.deptno ∧ dname = "Production") ∕(designation = "Manager" ∧ Employee.deptno = Department.deptno ∧ dname = "Printing")

Here, in the above disjunctive normal form, each disjunctive connected by $\vee$ (OR) operator can be processed as independent conjunctive subqueries.

Analysis

The objective of the analysis step is to reject normalized queries that are incorrectly formulated or contradictory. The query is lexically and syntactically analyzed in this step by using the compiler of the high-level query language in which the query is expressed. In addition, this step verifies whether the relations and attributes specified in the query are defined in the global conceptual schema or not. It is also checked in the analysis step whether the operations on database objects specified in the

given query are correct for the object type. When incorrectness is detected in any of the above aspects, the query is returned to the user with an explanation; otherwise, the high-level query is transformed into an internal form for further processing. The incorrectness in the query is detected based on the corresponding **query graph** or **relation connection graph**, which can be constructed as follows:

- A node is created in the query graph for the result and for each base relation specified in the query.
- An edge between two nodes is drawn in the query graph for each join operation and for each project operation in the query. An edge between two nodes that are not result nodes represents a join operation, whereas an edge whose destination node is the result node represents a project operation.
- A node in the query graph that is not the result node is labelled by a select operation or a self-join operation specified in the query.

The relation connection graph is used to check the semantic correctness of the subset of queries that do not contain disjunction or negation. Such a query is semantically incorrect, if its relation connection graph is not connected. A **join graph** for a query is a subgraph of the relation connection graph which represents only join operations specified in the query, and it can be derived from the corresponding query graph. The join graph is useful in the query optimization phase. A query is contradictory if its **normalized attribute connection graph** [Rosenkrantz & Hunt, 1980] contains a cycle for which the valuation sum is negative. The construction of **normalized attribute connection graph** for a query is described in the following:

- A node is created for each attribute referenced in the query and an additional node is created for a constant 0.
- A directed edge between two attribute nodes is drawn to represent each join operation in the query. Similarly, a directed edge between an attribute node and a constant 0 node is drawn for each select operation specified in the query.
- A weight is assigned to each edge depending on the inequality condition mentioned in the query. A weight v is assigned to the directed edge $a1 \rightarrow a2$, if there is an inequality condition in the query that satisfies $a1 \leq a2 + v$. Similarly, a weight $-v$ is assigned to the directed edge $0 \rightarrow a1$, if there is an inequality condition in the query that satisfies $a1 \geq v$.

Example 11.4

Let us consider the following two relations:

Student (s-id, sname, address, course-id, year) and
Course (course-id, course-name, duration, course-fee, intake-no, coordinator)

and the query:

"Retrieve the names, addresses and course names of all those students whose year of admission is 2008 and course duration is 4 years".

Using SQL, the above query can be represented as:

Select sname, address, course-name from Student, Course where year = 2008 and duration = 4 and Student.course-id = Course.course-id.

Here, course-id is a foreign key of the relation Student. The above SQL query can be syntactically and semantically incorrect for several reasons. For instance, the attributes sname, address, course-name, year and duration may not be declared in the corresponding schema or the relations Student, Course may not be defined in the global conceptual schema. Furthermore, if the operations "= 2008" and "= 4" are incompatible with data types of the attributes **year** and **duration** respectively, then the above SQL query is incorrect.

The query graph and the join graph for the above SQL query are depicted in figures 11.4(a) and 11.4(b) respectively.

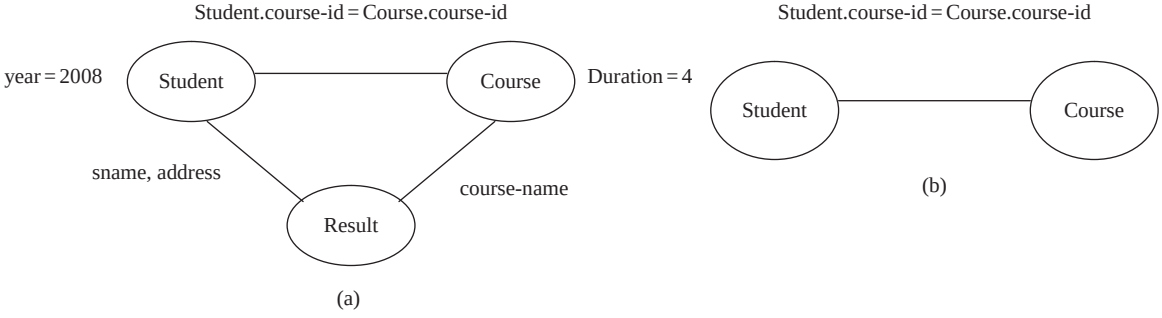


Fig. 11.4 (a) Query Graph; (b) Join Graph

In the above SQL query, if the join condition between two relations (that is, Student.course-id = Course.course-id) is missing, then there would be no line between the nodes representing the relations Student and Course in the corresponding query graph [**figure 11.4(a)**]. Therefore, the SQL query is deemed semantically incorrect as the relation connection graph is disconnected. In this case, either the query is rejected or an implicit Cartesian product between the relations is assumed.

Example 11.5

Let us consider the query "Retrieve all those student names who are admitted into courses where the course duration is greater than 3 years and less than 2 years", that involves the relations Student and Course.

In SQL, the query can be expressed as:

Select sname from Student, Course where duration > 3 and Student.course-id = Course. course-id and duration ⩽ 2.

The normalized attribute connection graph for the above query is illustrated in figure 11.5.

In the above normalized attribute connection graph, there is a cycle between the nodes duration and 0 with a negative valuation sum, which indicates that the query is contradictory.

Simplification

In this step, all redundant predicates in the query are detected and the common subexpressions are eliminated to transform the query into a simpler and efficiently computable form. This transformation must achieve the semantic correctness. Typically, view definitions, access restrictions and integrity constraints are considered in this step, some of which may introduce redundancy in the

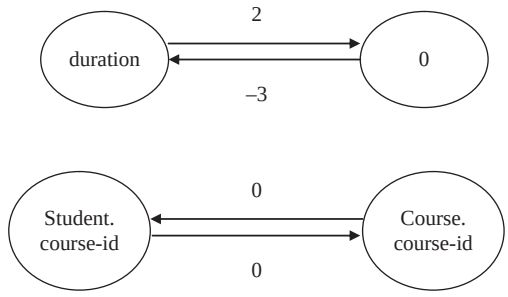


Fig. 11.5 Normalized Attribute Connection Graph

query. The well-known idempotency rules of Boolean algebra are used to eliminate redundancies from the given query, which are listed below.

- $P \wedge P \Leftrightarrow P$
- $P \vee P \Leftrightarrow P$
- $P \wedge \text{true} \Leftrightarrow P$
- $P \wedge \text{false} \Leftrightarrow \text{false}$
- $P \vee \text{true} \Leftrightarrow \text{true}$
- $P \vee \text{false} \Leftrightarrow P$
- $P \wedge (\sim P) \Leftrightarrow \text{false}$
- $P \vee (\sim P) \Leftrightarrow \text{true}$
- $P \wedge (P \vee Q) \Leftrightarrow P$
- $P \vee (P \wedge Q) \Leftrightarrow P$

Example 11.6

Let us consider the following view definition and a query on the view that involves the relation **Employee (empid, ename, salary, designation, deptno)**.

Create view V1 as select empid, ename, salary from Employee where deptno = 10;
Select * from V1 where deptno = 10 and salary > 10000;

During query resolution, the query will be:

Select empid, ename, salary from Employee where (deptno = 10 and salary > 10000) and deptno = 10;

Here, the predicates are redundant and the WHERE condition reduces to "deptno = 10 and salary > 10,000".

Query restructuring

In this step, the query in the high-level language is rewritten into an equivalent relational algebraic form. This step involves two substeps. Initially, the query is converted into an equivalent relational

algebraic form and then the relational algebraic query is restructured to improve performance. The relational algebraic query is represented by a **query tree** or **operator tree** which can be constructed as follows:

- A leaf node is created for each relation specified in the query.
- A non-leaf node is created for each intermediate relation in the query that can be produced by a relational algebraic operation.
- The root of the query tree represents the result of the query and the sequence of operations is directed from the leaves to the root.

In relational data model, the conversion from SQL query to relational algebraic form can be done in an easier way. The leaf nodes in the query tree are created from the FROM clause of the SQL query. The root node is created as a project operation involving the result attributes from the SELECT clause specified in SQL query. The sequence of relational algebraic operations, which depends on the WHERE clause of SQL query, is directed from the leaves to the root of the query tree. After generating the equivalent relational algebraic form from the SQL query, the relational algebraic query is restructured by using **transformation rules** from relational algebra, which are listed below. In listing these rules, three different relations *R, S* and *T* are considered, where the relation *R* is defined over the attributes $A = \{A_1, A_2, \ldots, A_n\}$ and the relation *S* is defined over the attributes $B = \{B_1, B_2, \ldots\ldots\ldots, B_n\}$.

(i) Commutativity of binary operators – Binary operators of relational algebra such as Cartesian product, join, union and intersection are commutative:

$R \times S \Leftrightarrow S \times R$ and $R \bowtie S \Leftrightarrow S \bowtie R$

Similarly, $R \cup S \Leftrightarrow S \cup R$ and $R \cap S \Leftrightarrow S \cap R$

This rule is not applicable to set difference and semijoin operations of relational algebra.

(ii) Associativity of binary operators – Cartesian product and natural join operation are always associative:

$(R \times S) \times T \Leftrightarrow R \times (S \times T)$ and

$(R \bowtie S) \bowtie T \Leftrightarrow R \bowtie (S \bowtie T)$

(iii) Idempotence of unary operators – Several successive unary operations such as selection and projection on the same relation may be grouped together. Conversely, a single unary operation on several attributes can be separated into several successive unary operations as shown below:

$\sigma_{p(A1)}(\sigma_{q(A2)}(R)) = \sigma_{p(A1) \wedge q(A2)}(R)$, where *p* and *q* denote predicates.

Similarly, $\prod_{A'}(\prod_{A''}(R)) \Leftrightarrow \prod_{A'}(R)$,

Where *A, A′* and *A″* are sets of attributes defined on relation *R* and *A′* and *A″* are subsets of *A*, and *A″* is a subset of *A′*.

(iv) Commutativity of selection and projection – Selection and projection operations on the same relation can be commuted:

$\prod_{A1,\ldots,An}(\sigma_{p(Ap)}(R)) \Leftrightarrow \prod_{A1,\ldots,An}(\sigma_{p(Ap)}(\prod_{A1,\ldots,An,Ap}(R)))$,

where *Ap* is not a member of $\{A1, \ldots, An\}$.

(v) **Commutativity of selection with binary operators** – Binary operations of relational algebra such as Cartesian Product and join operations can be commuted with selection operation as follows:

$\sigma_{p(Ai)}(R \times S) \Leftrightarrow (\sigma_{p(Ai)}(R)) \times S$ and

$\sigma_{p(Ai)}(R \bowtie_{p(Aj, Bk)} S) \Leftrightarrow (\sigma_{p(Ai)}(R) \bowtie_{p(Aj, Bk)} S)$, Where $Ai \in R$ and $Bk \in S$.

Similarly, selection operation can be commuted with union and set difference operations if both relations are defined over the same schema:

$\sigma_{p(Ai)}(R \cup S) \Leftrightarrow \sigma_{p(Ai)}(R) \cup \sigma_{p(Ai)}(S)$

(vi) **Commutativity of projection with binary operators** – Binary operations of relational algebra such as Cartesian product and join operation can be commuted with projection operation as shown in the following:

$\Pi_C(R \times S) \Leftrightarrow \Pi_{A'}(R) \times \Pi_{B'}(S)$

$\Pi_C(R \bowtie_{p(Aj, Bk)} S) \Leftrightarrow \Pi_{A'}(R) \bowtie_{p(Aj, Bk)} \Pi_{B'}(S)$,

where $C = A' \cup B'$, $Ai \in A'$, $Bk \in B'$ and $A' \subseteq A$ and $B' \subseteq B$.
Similarly, projection operation can be commuted with union and set difference operations:

$\Pi_C(R \cup S) \Leftrightarrow \Pi_C(R) \cup_C (S)$.

Proofs of the above transformation rules are available in [Aho et al., 1979]. By applying these rules, a large number of equivalent query trees can be constructed for a given query among which the most cost-effective one is chosen at the optimization phase. However, in this approach, generation of excessively large number of operator trees for the same query is not realistic. Therefore, the above transformation rules are used in a methodical way to construct the query tree for a given query. The transformation rules are used in the following sequence.

- Unary operations in the query are separated first to simplify the query.
- Unary operations on the same relation are grouped so that common expressions are computed only once [**rule no. (iii)**].
- Unary operations are commuted with binary operations [**rule no. (v) & (vi)**].
- Binary operations are ordered.

Example 11.7

Let us consider the following SQL query which involves the relations **Student (s-id, sname, address, course-id, year)** and **Course (course-id, course-name, duration, course-fee, intake-no, coordinator)**:

Select sname, course-name from Student, Course where s-id > 0 and year = 2007 and Student. course-id = Course.course-id and duration = 4 and intake-no = 60.

The query tree for the above SQL query is depicted in figure 11.6.

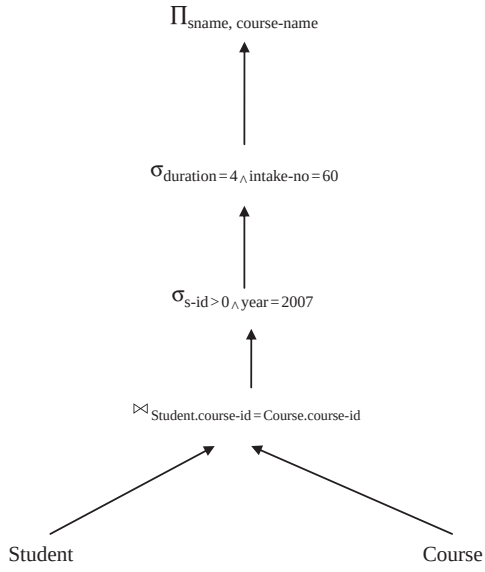


Fig. 11.6 Query tree

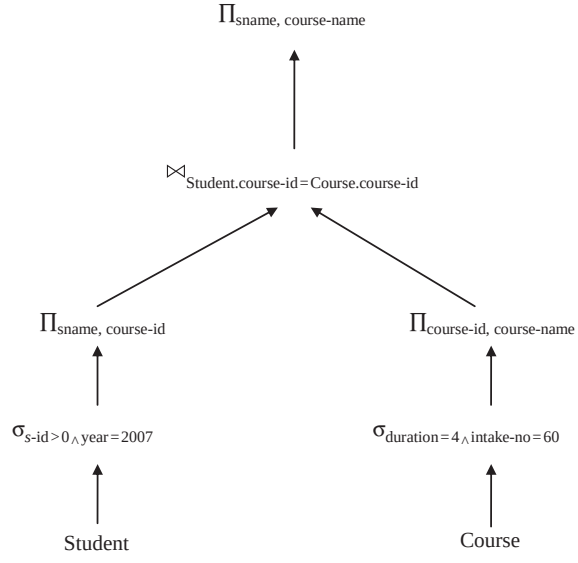


Fig. 11.7 Equivalent Query Tree of Figure 11.6

An equivalent query tree is constructed by applying the above transformation rules as shown in figure 11.7.

Here, unary operations specified in the query are separated first, and all unary operations on the same relation are grouped together to reconstruct the query tree. However, the query tree in the figure 11.7 need not necessarily represent the optimal tree.

11.3.2 Query Fragmentation

In **query fragmentation phase** a relational algebraic query on global relations is converted into an algebraic query expressed on physical fragments, called a **fragment query**, considering data distribution in the distributed database. A global relation can be reconstructed from its fragments using the **reconstruction rules** of fragmentation [as described in **Chapter 5, Section 5.3.2**]. For horizontal fragmentation, the reconstruction rule is the union operation of relational algebra, and for vertical fragmentation the reconstruction rule is the join operation of relational algebra. Thus, query fragmentation is defined through fragmentation rules, and it uses the information stored in the fragment schema of the distributed database. For the sake of simplicity, the replication issue is not considered here.

An easier way to generate a fragment query is to replace the global relations at the leaves of the query tree or the operator tree of the distributed query with their reconstruction rules. The relational algebraic query tree generated by applying the reconstruction rules is known as **generic tree**. This approach is not too efficient because more simplifications and reconstructions are possible in generic trees. Therefore, to generate a simpler and optimized query from a generic query, **reduction techniques** are used where the reduction techniques are dependent on the types of fragmentation. Reduction techniques for different types of fragmentation are illustrated with examples in the next section.

Reduction for horizontal fragmentation

In a distributed database system, horizontal fragmentation is done based on selection predicates. Here, two different reduction techniques are considered for horizontal fragmentation: **reduction with selection operation** and **reduction with join operation**. In the first case, if the selection predicate contradicts the definition of the fragment, then an empty intermediate result relation is produced; thus, this operation can be eliminated. In the second case, the join operation can be commuted with union operation to detect useless join operations in the query, which can be eliminated from the result. A useless join operation exists in a query, if the fragment predicates do not overlap.

Example 11.8

Let us consider the following SQL query that involves the relation **Employee (empid, ename, salary, designation, deptno)**:

Select * from Employee where salary > 10000.

Assume that the Employee relation is partitioned into two horizontal fragments EMP_1 and EMP_2 depending on the selection predicates as mentioned below:

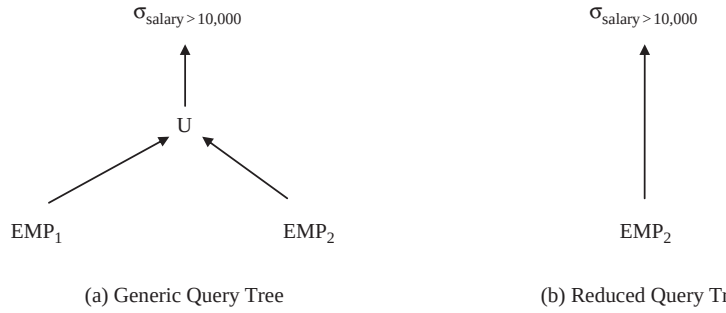
$$\mathbf{EMP_1 = \sigma_{salary <= 10000}\ (Employee)}$$

$$\mathbf{EMP_2 = \sigma_{salary > 10000}\ (Employee)}$$

Now, the relation Employee can be reconstructed from its horizontal fragments EMP_1 and EMP_2 by using the following reconstruction rule.

$$\mathbf{Employee = EMP_1 \cup EMP_2}$$

Therefore, in the generic tree of the above SQL query, the leaf node corresponding to the Employee relation can be replaced by the reconstruction rule $EMP_1 \cup EMP_2$. Here, the selection predicate contradicts the definition of horizontal fragment EMP_1, thereby producing an empty relation. This operation can be eliminated from the generic tree as shown in figure 11.8.



(a) Generic Query Tree (b) Reduced Query Tree

Fig. 11.8 Reduction for Horizontal Fragmentation with Selection

Example 11.9

Let us assume that the relation **Employee (empid, ename, salary, designation, deptno)** is horizontally fragmented into two partitions EMP_1 and EMP_2 and the relation **Department (deptno, dname, location)** is horizontally fragmented into two relations $DEPT_1$ and $DEPT_2$ respectively. These horizontally fragmented relations are defined in the following:

$\mathbf{EMP_1 = \sigma_{deptno \leq 10}}$ **(Employee)**

$\mathbf{EMP_2 = \sigma_{deptno > 10}}$ **(Employee)**

$\mathbf{DEPT_1 = \sigma_{deptno \leq 10}}$ **(Department)**

$\mathbf{DEPT_2 = \sigma_{deptno > 10}}$ **(Department)**

The reconstruction rules for the above horizontal fragments are as follows:

$\mathbf{Employee = EMP_1 \cup EMP_2}$

$\mathbf{Department = DEPT_1 \cup DEPT_2}$

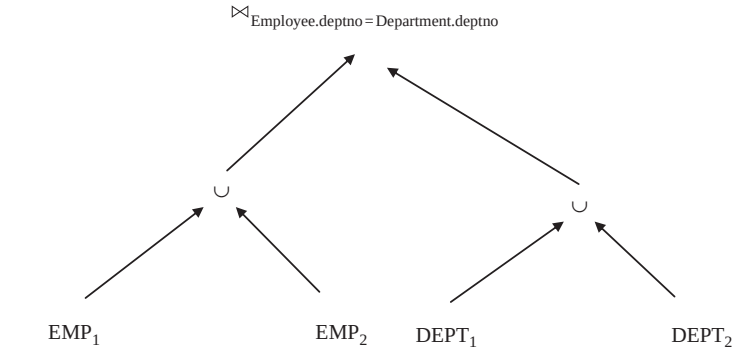
Let us consider the following SQL query in a distributed environment, which involves the relations Employee and Department.

Select * from Employee, Department where Employee.deptno = Department.deptno.

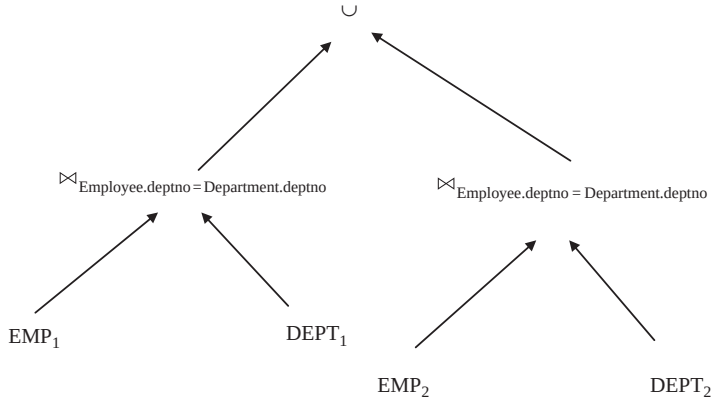
The generic query tree and the reduced query tree for the above query are depicted in figure 11.9. Thus, the commutativity of join operation with union operation is very important in distributed DBMSs, because it allows a join operation of two relations to be implemented as a union operation of partial join operations, where each part of the union operation can be executed in parallel.

Reduction for vertical fragmentation

The vertical fragmentation partitions a relation based on projection operation; thus, the reconstruction rule for vertical fragmentation is join operation. The reduction technique for vertical fragmentation involves the removal of vertical fragments that have no attributes in common except the key attribute.



(a) Generic Query Tree



(b) Reduced Query Tree

Fig. 11.9 Reduction for Horizontal Fragmentation with Join

Example 11.10

Let us assume that the Employee relation is vertically fragmented into two relations EMP_1 and EMP_2 as defined below:

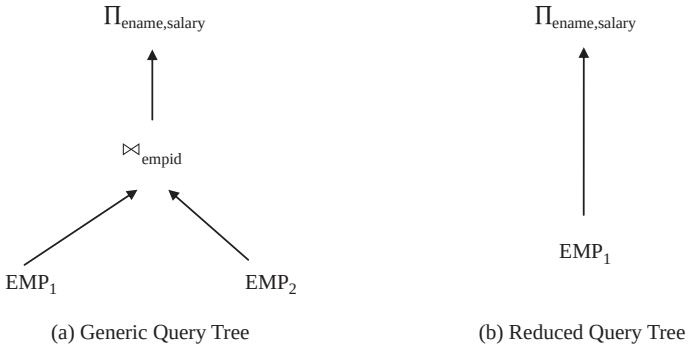
$$\mathbf{EMP_1 = \prod_{empid,ename,salary}\ (Employee)}$$

$$\mathbf{EMP_2 = \prod_{empid,designation,deptno}\ (Employee)}$$

Let us consider the following SQL query:

Select ename, salary from Employee.

In this query, the projection operation on relation EMP_2 is redundant, since the attributes ename and salary are not part of EMP_2. The generic query tree of the above query is illustrated in figure 11.10(a). By commutating the projection operation with join operation and removing the vertical fragment EMP_2 the reduced query tree is produced as shown in figure 11.10(b).



(a) Generic Query Tree (b) Reduced Query Tree

Fig. 11.10 Reduction for Vertical Fragmentation

Reduction for derived fragmentation

The derived fragmentation partitions a child relation horizontally based on the same selection predicate of its parent relation. Derived fragmentation is used to facilitate the join between fragments. The reconstruction rule for derived fragmentation is join operation over union operation. The reduction technique for derived fragmentation involves the commutativity of join operation with union operation and removal of useless join operations and selection operations.

Example 11.11

Let us assume that the relation **Department (deptno, dname, location)** is horizontally fragmented into two relations $DEPT_1$ and $DEPT_2$ respectively. These horizontally fragmented relations are defined in the following:

$$\mathbf{DEPT_1 = \sigma_{deptno \leq 10}\ (Department)}$$

$$\mathbf{DEPT_2 = \sigma_{deptno > 10}\ (Department)}$$

Further assume that the fragmentation of Employee relation is derived from Department relation. Thus, EMP_i = Employee $\triangleright_{deptno}$ $DEPT_i$ $i = 1,2$
Let us consider the following SQL query in a distributed environment, which involves the relations Employee and Department.

Select * from Employee, Department where deptno > 10 and Employee.deptno = Department. deptno.

The generic query tree for the derived fragmentation is depicted in figure 11.11(a).
In the above generic query tree, the selection operation on fragment $DEPT_1$ is redundant and can be eliminated. Similarly, as the relation $DEPT_2$ is defined depending on the selection predicate "deptno>10", the entire selection operation can be eliminated from the above generic tree. By eliminating selection operation and commutating join operation with union operation, the reduced query tree is produced which is shown in figure 11.11(b).

Reduction for mixed fragmentation

A mixed fragment is defined as a combination of horizontal, vertical and derived fragmentations. The main objective of the mixed fragmentation is to support queries that involve selection,

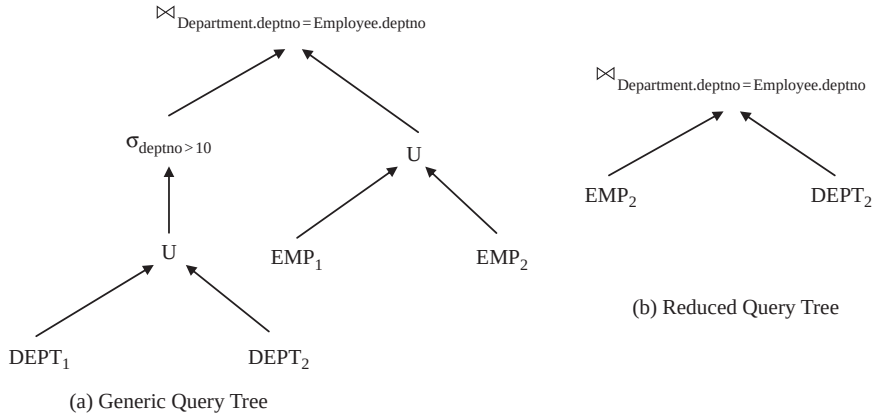


Fig. 11.11 Reduction for Derived Fragmentation

projection and join operations. The reconstruction rule for mixed fragmentation uses union operations and join operations of fragments. All reduction techniques that are used for horizontal, vertical and derived fragmentation can be used for mixed fragmentation. These reduction techniques are summarized below:

- Removal of empty relations that can be produced by contradicting selection predicates on horizontal fragments.
- Removal of useless relations that can be produced by applying projection operations on vertical fragments.
- Removal of useless join operations that can be produced by distributing join operations over union operations on derived fragments.

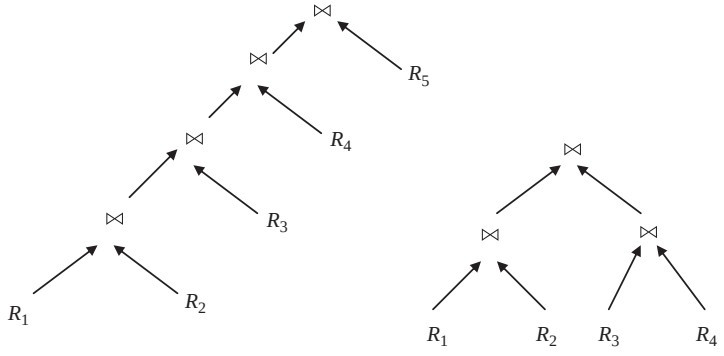
11.3.3 Global Query Optimization

Query optimization is the activity of choosing an efficient execution strategy among several alternatives for processing a query. This is the same both in the contexts of both centralized DBMSs and distributed DBMSs. The selected query execution strategy must minimize the cost of query processing. In a distributed DBMS, the query execution cost is expressed as the total of local processing cost (I/O cost and CPU cost) and communication cost. To simplify the global query optimization, often the local processing cost is ignored, because the communication cost is dominant in this case. The optimal query execution strategy is selected by a software module, known as **query optimizer**, and it can be represented by three components: **search space, cost model** and **optimization strategy**. The search space is obtained by applying the transformation rules of relational algebra as described in Section 11.3.1 under the subheading *Query restructuring*. The cost model determines the cost of a query execution plan. The optimization strategy explores the search space, and it is used to determine the optimal plan using the cost model.

Search space

The **search space** is defined as the set of equivalent query trees for a given query that can be generated by using transformation rules. In query optimization, join trees are particularly important,

because they determine the join order of relations involved in a given query, which affects the performance of query processing. If a given query involves many operators and many relations, then the search space is very large, because it contains a large number of equivalent query trees for the given query. Hence, the query optimization process becomes more expensive than the actual execution, and therefore query optimizers typically impose some restrictions on the size of the search space to be considered. Most query optimizers use heuristics rules that order the relational algebraic operations (selection, projection and Cartesian product) in the query. Another restriction is one that is imposed on the shape of the join tree. There are two different kinds of join trees, known as **linear join trees** and **bushy join trees**. In a linear join tree, at least one operand of each operator node is a base relation. On the other hand, a bushy join tree is more general and may have operators with no base relations as operands. Linear join trees reduce the size of the search space, whereas bushy join trees facilitate parallelism. The examples of a linear join tree and a bushy join tree are illustrated in figures 11.12(a) and 11.2(b) respectively.



(a) An Example of Linear Join Tree (b) An Example of Bushy Join Tree

Fig. 11.12 Two Different Types of Join Trees

Optimization strategy

There are three different kinds of optimization strategies, known as **static optimization strategy, dynamic optimization strategy** and **randomized optimization strategy**, which are described in the following.

(a) Static optimization strategy – In static optimization strategy, the given query is parsed, validated and optimized once. This strategy is similar to the approach adopted by a compiler for a programming language. The advantages of static optimization strategy are that the run-time overhead is removed, and there may be more time available for evaluating a large number of execution strategies, which increases the chances of finding a more optimum strategy. The disadvantage with the static optimization strategy is that the optimal execution strategy that is selected at compile-time may no longer be optimal at run-time.

(b) Dynamic optimization strategy – The most popular optimization strategy is dynamic optimization strategy, which is deterministic. In dynamic optimization strategy, the execution plan starts from base relations and proceeds by joining one more relation at each step until the complete plan is obtained. The advantage of dynamic optimization strategy is that it is almost comprehensive and ensures the best plan. One disadvantage is that the performance of the

query is affected, because the query has to be parsed, validated and optimized each time before execution. Moreover, it is necessary to reduce the number of execution strategies to be analysed to achieve an acceptable overhead, which may have the effect of selecting a less-than-optimum strategy. This strategy is suitable when the number of relations involved in the given query is small.

(c) Randomized optimization strategy – A recent optimization strategy is randomized optimization strategy, which reduces the optimization complexity, but does not guarantee the best execution plan. The randomized optimization strategy allows the query optimization to trade optimization tune for execution time [Lanzelotte et al., 1993]. In this strategy, one or more start plans are built by a greedy strategy, and then it tries to improve the start plan by visiting its neighbours. A neighbour is identified by applying a random transformation to a plan. One typical example of random transformation is exchanging two randomly chosen operand relations of the plan. The randomized optimization strategy provides better performance than other optimization strategies when the number of relations involved in the query increases.

Distributed cost model

The distributed cost model includes cost functions to predict the cost of operators, database statistics, base data and formulas to calculate the sizes of intermediate results.

Cost functions In a distributed system, the cost of processing a query can be expressed in terms of the total cost measures or the response time measures [Yu and Chang, 1984]. The total cost measure is the sum of all cost components. If no relation is fragmented in the distributed system and the given query includes selection and projection operations, then the total cost measure involves the local processing cost only. However, when join and semijoin operations are executed, communication costs between different sites may be incurred in addition to the local processing cost. Local processing costs are usually evaluated in terms of the number of disk accesses and the CPU processing time, whereas communication costs are expressed in terms of the total amount of data transmitted. For geographically dispersed computer networks, communication cost is normally the dominant consideration, but local processing cost is of greater significance for local networks. Thus, most early distributed DBMSs designed for wide area networks have ignored the local processing cost and concentrated on minimizing the communication cost. Therefore, the total cost measure can be represented by using the following formula.

$$\textbf{Total cost measure} = T_{CPU} * \textbf{insts} + T_{I/O} * \textbf{ops} + C_0 + C_1 * X$$

where T_{CPU} is the CPU processing cost per instruction, insts represents the total number of CPU instructions, $T_{I/O}$ is the I/O processing cost per I/O operation, ops represents the total number of I/O operations, C_0 is the start-up cost of initiating transmission, C_1 is a proportionality constant, and X is the amount of data to be transmitted. For wide area networks, the above formula is simplified as follows.

$$\textbf{Total cost measure} = C_0 + C_1 * X.$$

The response time measure is the time from the initiation of the query to the time when the answer is produced. The response time measure must consider the parallel local processing costs and the parallel communication costs. A general formula for response time can be expressed as follows.

$$\textbf{Response time measure} = T_{CPU} * \textbf{Seq\_insts} + T_{I/O} * \textbf{seq\_ops} + C_0 + C_1 * \textbf{seq\_}X$$

where seq_insts represents the maximum number of CPU instructions that can be performed sequentially, seq_ops represents the maximum number of I/O operations that can be performed sequentially, and seq_*X* indicates the amount of data that can be transmitted sequentially. If the local processing cost is ignored, then the above formula is simplified into

Response time measure = $C_0 + C_1$ * seq_*X*.

In this case, any processing or communication that is done in parallel is ignored. The following example illustrates the difference between total cost measure and response time measure.

Example 11.12

Let us consider that the amount of data to be transmitted from site1 to site2 is *P*, and the amount of data to be transmitted from site3 to site2 is *Q* for the execution of a query as shown in the figure 11.13.

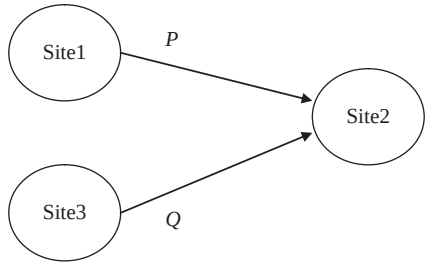


Fig. 11.13 Data Transmission for a Query

Here, the **total cost measure** = $C_0 + C_1 * P + C_0 + C_1 * Q = 2C_0 + C_1(P + Q)$.

Similarly, the **response time measure** = **max**$\{C_0 + C_1 * P, C_0 + C_1 * Q\}$ because the data transmission is done in parallel. In this case, the local processing cost is ignored. The response time measure can be minimized by increasing the degree of parallel execution, whereas the total cost measure can be minimized by improving the utilization of resources.

Database statistics The cost of an execution strategy depends on the size of the intermediate result relations that are produced during the execution of a query. In distributed query optimization, it is very important to estimate the size of the intermediate results of relational algebraic operations to minimize the data transmission cost, because the intermediate result relations are transmitted over the network. This estimation is done based on the database statistics of base relations stored in the system catalog and the formulas used to predict the cardinalities of the results of the relational algebraic operations. Typically, it is expected that a distributed DBMS will hold the following information in its system catalog to predict the size of intermediate result relations.

- The length of each attribute *Ai* in terms of number of bytes denoted by length(*Ai*).
- The number of distinct values of each attribute *Ai* in each fragment *Rj* with the cardinality of the projection of fragment *Rj* on *Ai*, denoted by card($\prod_{Ai}$ (*Rj*)).
- The maximum and minimum possible values in the domain of each attribute *Ai*, denoted by Max (*Ai*) and Min (*Ai*).

- The cardinality of the domain of each attribute Ai, denoted by card(dom[Ai]), which represents the number of unique values in the dom[Ai].
- The number of tuples in each fragment Rj, denoted by card(Rj).

In addition to the above information, sometimes the system catalog also holds the join selectivity factor for some pairs of relations. The **join selectivity factor** of two relations R and S is a real value between 0 and 1, and can be defined as follows.

$$\mathbf{SFj}(R, S) = \mathbf{card}\ (R \bowtie S)/\mathbf{card}(R) * \mathbf{card}(S).$$

Cardinalities of intermediate results To simplify the evaluation of the cardinalities of intermediate results, two assumptions have been made here: first, the distribution of attribute values in a base relation is uniform, and second, all attributes are independent, that is, the value of one attribute does not affect the values of others. The formulas to evaluate the cardinalities of the results of basic relational algebraic operations are listed in the following.

1. Selection Operation.
The cardinality of selection operation for the relation R is

$$\mathbf{card}(\sigma_F (R)) = \mathbf{SF}_S (F) * \mathbf{card}(R)$$

where SF_S (F) is dependent on the selection formula. The $SF_S(F)$s can be calculated as follows.

$$\mathbf{SF}_S (A = \mathbf{value}) = 1/\mathbf{card}(\textstyle\prod_A(R))$$

$$\mathbf{SF}_S (A > \mathbf{value}) = \mathbf{max}(A) - \mathbf{value}/\mathbf{max}(A) - \mathbf{min}(A)$$

$$\mathbf{SF}_S (A < \mathbf{value}) = \mathbf{value} - \mathbf{min}(A)/\mathbf{max}(A) - \mathbf{min}(A)$$

$$\mathbf{SF}_S (p(Ai) \wedge p(Aj)) = \mathbf{SF}_S (p(Ai) * \mathbf{SF}_S (p(Aj)))$$

$$\mathbf{SF}_S (p(Ai) \wedge p(Aj)) = \mathbf{SF}_S (p(Ai)) + \mathbf{SF}_S (p(Aj)) - (\mathbf{SF}_S (p(Ai)) * \mathbf{SF}_S (p(Aj)))$$

$$\mathbf{SF}_S (A \in \{\mathbf{values}\}) = \mathbf{SF}_S (A + \mathbf{value}) * \mathbf{card}(\{\mathbf{values}\})$$

In this case, Ai and Aj are two different attributes, and $p(Ai)$ and $p(Aj)$ denotes the selection predicates.

2. Projection Operation
It is very difficult to evaluate the cardinality of an arbitrary projection operation. However, it is trivial in some cases. If the projection of relation R is done based on a single attribute, the cardinality is the number of tuples when the projection is performed. If one of the projected attributes is a key of the relation R, then

$$\mathbf{card}(\textstyle\prod_A(R)) = \mathbf{card}(R).$$

3. Cartesian Product Operation
The cardinality of the Cartesian product of two relations R and S is denoted as

$$\mathbf{card}(R \times S) = \mathbf{card}(R) * \mathbf{card}(S).$$

4. Join Operation
There is no general way to evaluate the cardinality of the join operation without additional information about the join operation. Typically, the upper bound of the cardinality of the join operation is the cardinality of the Cartesian product. However, there is a common case in which the evaluation

is simple. If the relation R with its attribute A is joined with the relation S with its attribute B via an equijoin operation, where A is a key of the relation R, and B is a foreign key of relation S, then the cardinality of the join operation can be evaluated as follows.

$$\mathbf{card}(R \bowtie_{A=B} S) = \mathbf{card}(S).$$

In the above evaluation, it is assumed that the each tuple of relation R participates in the join operation; therefore, the above estimation is an upper bound. In other cases, it can be calculated as

$$\mathbf{card}(R \bowtie S) = \mathbf{SF}_{\mathbf{J}} * \mathbf{card}(R) * \mathbf{card}(S)$$

5. Semijoin Operation
The selectivity factor of the semijoin operation of the relation R by the relation S selects the subset of tuples of R that join with tuples of S. An approximation for the semijoin selectivity factor is represented as follows [Hevner and Yao, 1979].

$$\mathbf{SF}_{\mathbf{SJ}}(R \ltimes_A S) = \mathbf{card}(\Pi_A(S))/\mathbf{card}(\mathbf{dom}[A])$$

The above formula depends on only the attribute A of the relation S. Thus, it is often called the selectivity factor of attribute A of S and is denoted by $\mathbf{SF}_{\mathbf{SJ}}(S.A)$. Now, the cardinality of the semijoin operation can be expressed by the following formula.

$$\mathbf{card}(R \ltimes_A S) = \mathbf{SF}_{\mathbf{SJ}}(S.A) * \mathbf{card}(R)$$

This formula is applicable for the common case where the attribute $R.A$ is a foreign key of the relation S. In this case, the semijoin selectivity factor is 1, because $\prod_A(S) = \text{card}(\text{dom}[A])$.

6. Union Operation
It is very difficult to evaluate the cardinality of union operation. The simple formulas for evaluating the upper bound and lower bound cardinalities of union operation are listed in the following.

$$\mathbf{card}(R \cup S)\ \mathbf{(upper\ bound)} = \mathbf{card}(R) + \mathbf{card}(S)$$

$$\mathbf{card}(R \cup S)\ \mathbf{(lower\ bound)} = \mathbf{max}\{\mathbf{card}(R), \mathbf{card}(S)\}$$

7. Set Difference Operation
Like union operation, the formulas for evaluating the upper bound and lower bound cardinalities of set difference operation, denoted by $\text{card}(R - S)$, are $\text{card}(R)$ and 0 respectively.

11.3.4 Local Query Optimization

In the context of distributed query processing, local query optimization is the utmost important, because the distributed query is fragmented into several subqueries each of which is processed at local site in a centralized way. Moreover, the query optimization techniques for centralized DBMSs are often extended for use in distributed DBMSs. There are several query optimization techniques for centralized DBMSs. One such centralized query optimization technique, used by the popular relational database system INGRES [Stonebraker et al., 1976], is introduced in the following section.

INGRES algorithm INGRES uses a dynamic query optimization strategy that partitions the high-level query into smaller queries recursively. In this approach, a query is first decomposed into a sequence of queries having a unique relation in common. Each of these single relation queries are

then processed by a one-variable query processor (OVQP). The OVQP optimizes the access to a single relation by selecting, based on the selection predicates, the best access method to that relation. For instance, if there is a selection predicate in the query of the form <A = value>, an index available on attribute A will be used. This algorithm first executes the unary operations and tries to minimize the size of the intermediate results before performing binary operations.

To perform decomposition, two basic techniques are used in the INGRES algorithm: **detachment** and **substitution**. The query processor uses the detachment technique to partition a given query into several smaller queries based on a common relation. For example, using detachment technique, the SQL query q1 can be decomposed into two smaller queries q11 and q12 as follows.

q1: Select $R_2.A_2, R_3.A_3, \ldots, R_n.A_n$

from $R_1, R_2, \ldots, R_n$

where $P_1(R_1.A_1)$ **and** $P_2(R_1.A_1, R_2.A_2, \ldots, R_n.A_n)$

q11: Select $P_1.A_1$ **into** $R_{1'}$

from R_1

where $P_1(R_1.A_{1'})$

q12: Select $R_2.A_2, R_3.A_3, \ldots, R_n.A_n$

from $R_{1'}, R_2, \ldots, R_n$

where $P_2(V_1.A_1, V_2.A_2, \ldots, V_n.A_n)$

where A_i and $A_{i'}$ represent lists of attributes of relation R_i, P_1 is a selection predicate involving the attribute$A_{1'}$ from the relation R_1, and P_2 is a selection predicate involving attributes of relations R_1, $R_2, \ldots, R_n$, $V_1, \ldots, V_n$ represent new names of relation $R_1, \ldots, R_n$ after decomposition. This step is necessary to reduce the size of relations before performing binary operations. Detachment technique extracts the selection operations, which are usually the most selective ones.

Multi-relation queries that cannot be further detached are called **irreducible**. A query is said to be irreducible if and only if the corresponding query graph is a chain with two nodes. Irreducible queries are converted into single relation queries by tuple substitution. In tuple substitution, for a given n-relation query, the tuples of one relation are substituted by their values, thereby generating a set of $(n-1)$-relation queries. Tuple substitution can be implemented in the following way. Assume that the relation R is chosen for tuple substitution in an n-relation query q1. For each tuple in R, the attributes referred to in $q1$ are replaced by their actual values in tuples thereby producing a query q1' with $n-1$ relations. Therefore, the total number of queries produced by tuple substitution is card(R). An example of the INGRES algorithm is illustrated in the following.

Example 11.13

Let us consider the following SQL query that involves three different relations **Student(sreg-no, sname, street, city, course-id), Course(course-id, cname, duration, fees)**, and **Teacher(T-id, name, designation, salary, course-id).**

Select sname, name from Student, Course, Teacher where Student.course-id = Course. course-id and Course.couse-id = Teacher.course-id and duration = 4.

Using detachment technique, the above query can be replaced by the following queries, q1 and q2, where Course1 is an intermediate relation.

q1: Select Course.course-id into Course1 from Course where duration = 4.

q2: Select sname, name from Student, Course1, Teacher where Student.course-id = Course1. course-id and Course1.course-id = Teacher.course-id.

Similarly, the successive detachment of q2 may generate the queries q21 and q22 as follows.

q21: Select name, Teacher.course-id into Teacher1 from Teacher, Course1 where Teacher. course-id = Course1.course-id.

q22: Select sname from Student, Teacher1 where Student.course-id = Teacher1.course-id.

Assume that in query q22, three tuples are selected where course-id are C01, C03 and C06 respectively. The tuple substitution of Teacher1 relation produces three one-relation subqueries which are listed below.

q221: Select sname from Student where Student.course-id = "C01"

q222: Select sname from Student where Student.course-id = "C03"

q223: Select sname from Student where Student.course-id = "C06"

11.4 Join Strategies in Fragmented Relations

The ordering of join operations is very important in the contexts of both centralized and distributed query optimization. The algorithms represented in this section do not consider explicitly the fragmentation of relations. Thus, the generic term "relation" is used here to denote either fragmented or non-fragmented relations. There are two basic approaches for join strategies in fragmented relations. One is **simple join strategy** which tries to optimize the ordering of join operations directly. Another alternative approach is **semijoin strategy**, which replaces join operations by semijoin operations to minimize communication costs.

11.4.1 Simple Join Strategy

The objective of simple join strategy is to optimize the ordering of join operations of relations to minimize the query processing cost. For the sake of simplicity, only specific join queries are considered here whose operand relations are stored at different sites, and it is assumed that relation transfers among different sites of the distributed system are to be done in a set-at-a-time basis rather than in a tuple-at-a-time basis. Finally, the time for generating the data at a result site is ignored here.

Let us consider a simple query that involves the joining of two relations R and S that are stored at different sites. In performing $R \bowtie S$, the smaller relation should be transferred to the site of the larger relation. Therefore, it is necessary to calculate the size of the relations R and S. If the query involves one more relation T, then the obvious choice is to transfer the intermediate result $R \bowtie S$ or the relation T, whichever is smaller.

The difficulty with the simple join strategy is that the join operation may reduce or increase the size of the intermediate results. Hence, it is necessary to estimate the size of the results of join

operations. One solution is to estimate the communication costs of all alternative strategies and select the best one for which the communication cost is the minimum. However, the number of alternative strategies will increase as the number of relations increases.

11.4.2 Semijoin Strategy

The main drawback of simple join strategy is that the entire operand relation has to be transmitted between the sites. The objective of the semijoin strategy is to minimize the communication cost by replacing join operations of a query with semijoin operations. The join operation between two relations R and S over the attribute A, which are stored at different sites of the distributed system, can be replaced by semijoin operations as follows.

$$R \bowtie_A S \Leftrightarrow (R \ltimes_A S) \bowtie_A S \Leftrightarrow R \ltimes_A S \bowtie_A (S \ltimes_A R) \Leftrightarrow R \ltimes_A S \bowtie_A S \ltimes_A R.$$

It is necessary to estimate the cost of the above semijoin operations to understand the benefits of semijoin operations over a join operation. The local processing costs are not considered here for simplicity. The join operation $R \bowtie_A S$ and the semijoin operation $(R \ltimes_A S) \bowtie_A S$ can be implemented in the following way assuming the relations R and S are stored at site1 and site2 respectively, and size(R)<size(S).

R $\bowtie_A$***S:***

1. The relation R is to be transferred to site2.
2. The join operation is performed at site2.

(R $\bowtie_A$***S)*** $\bowtie_A$***S:***

1. The projection operation $\prod_A(S)$ is performed at site2 and result is sent to site1.
2. The site1 computes $R \ltimes_A S$, and the result is, say, T.
3. The result T is transferred to site2.
4. The join operation $T \bowtie_A S$ is computed at site2.

The communication cost for the above join operation is $C_0 + C_1$ * size(R), whereas the communication cost for the semijoin operation is $2C_0 + C_1$*(size($\prod_A(S)$) + size($R \ltimes_A S$)). Thus, the second operation is beneficial if size($\prod_A(S)$) + size($R \ltimes_A S$) is less than size(R), because C_0 is negligible compared to the cost of data transfer.

In general, the semijoin operation is useful to reduce the size of operand relations involved in multiple join queries. The optimal semijoin program is called a **full reducer,** which reduces a relation more than others [Chiu and Ho, 1980]. The determination of full reducers is a difficult task. One solution is to evaluate the size of reduction for all possible semijoin strategies and select the best one. Full reducers cannot be found in the group of queries that have cycles in their join graph, known as **cyclic queries.** For other groups of queries, called **tree queries,** full reducers exist, but the number of alternative semijoin strategies increases as the number of relations increases, which complicates the above solution.

The semijoin strategy is beneficial if only a few number of tuples participate in the join operation, whereas the simple join strategy is beneficial if most of the tuples participate in the join operation, because semijoin strategy involves an additional data transfer cost.

Example 11.14

Let us consider the following join operation that involves three different relations **Student, Course**, and **Teacher,** and over the attribute course-id.

$$\text{Student} \bowtie_{\text{course-id}} \text{Course} \bowtie_{\text{course-id}} \text{Teacher}$$

Further assume that the relations Student, Course and Teacher are stored at site 1, site 2 and site 3 respectively. The corresponding join graph is depicted in figure 11.14.

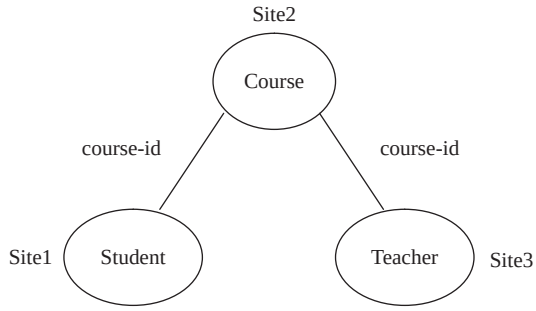


Fig. 11.14 Join Graph for a Distributed Query

There are various ways to perform the above join operation, but to select the best one, some more information must be known: size(Student), size(Teacher), size(Course), size(Student ⋈ Course) and size(Course ⋈ Teacher). Moreover, all data transfer can be done in parallel.

If the above join operation is replaced by semijoin operations, then the number of operations will be increased but possibly on smaller operands. In this case, instead of sending the entire Student relation to site2, only the values for the join attribute course-id of Student relation will be sent to site2. If the length of the join attribute is significantly less than the length of the entire tuple, then the semijoin has a good selectivity in this case, and it reduces the communication cost significantly. This is also applicable for performing join operation between the relations Course and Teacher. However, the semijoin operation may increase the local processing time, because one of the two operand relations must be accessed twice.

11.5 Global Query Optimization Algorithms

This section introduces three global query optimization algorithms, namely, distributed INGRES algorithm, R* algorithm and SDD-1 algorithm. All these algorithms are different from one another in a major way depending on several factors such as objective function, optimization strategy and network topology. The comparison between these three algorithms is shown in table 11.1.

11.5.1 Distributed INGRES Algorithm

Distributed INGRES algorithm is an extension of centralized INGRES algorithm and, thus, uses the dynamic optimization strategy. The main objective of this algorithm is to reduce total cost measure

Table 11.1 Comparison of Algorithms for Global Query Optimization

Algorithms	Objective function	Optim. Strategy	Comm. Schemes	Optim. Factor	Semijoins	Fragments
Distributed INGRES	Total cost measure or response time measure	Dynamic	General or broadcast	Msg. transfer cost, processing cost	No	Horizontal
SDD-1	Total cost measure	Static	General	Msg. transfer cost	Yes	No
R*	Total cost measure	Static	General or local	Msg. transfer and amount of data transfer cost, I/O cost, CPU cost	No	No

as well as response time measure, although these may be conflicting. For instance, increasing communication time by means of parallelism may decrease response time significantly. This algorithm supports horizontal fragmentation, and both general and broadcast networks can be used here. The particular site in the distributed system where the algorithm is executed is called **master site**, and it is the site where the query is initiated. The distributed INGRES algorithm works in the following way.

For a given query, all monorelation queries (unary operations such as selection and projection) can be detached first and these are processed locally. Then the reduction algorithm [Wong and Youssefi, 1976] is applied to the original query, and it produces two different kinds of subqueries: irreducible subqueries and monorelation subqueries. Reduction is a technique that separates all irreducible subqueries and monorelation subqueries from the original query by detachment technique. Since monorelation subqueries are already processed locally, no action is taken with respect to them. Assume that the reduction technique produces a sequence of irreducible subqueries $q1 \rightarrow q2 \rightarrow \ldots \rightarrow qn$, where at least one relation is common between two consecutive subqueries. It is mentioned in [Wong and Youssefi, 1976] that such a sequence is unique.

Based on the sequence of irreducible subqueries and the size of each fragment, one subquery is chosen from the list, say q*i*, that has at least two variables. For processing the subquery q*i*, initially the best strategy is determined for the subquery. This strategy is expressed by a list of pairs (*F, S*), where *F* denotes a fragment that is to be transferred to the processing site *S*. After transferring all fragments to the corresponding processing sites, finally the subquery q*i* is executed. This procedure is repeated for all irreducible subqueries in the sequence $q1 \rightarrow q2 \rightarrow \ldots \rightarrow qn$, and the algorithm terminates. This algorithm is represented in the following:

Distributed INGRES Algorithm:
Input: Given multi-relation query, *q*.
Output: Result of the last subquery *qn*.
Begin
Step 1: Detach all monorelation queries from the given query q and execute by OVQPs (one-variable query processors) at local sites the same way as in centralized INGRES algorithm.

Step 2: Apply reduction technique to q, which will produce a list of monorelation subqueries and a sequence of irreducible subqueries q1 → q2 →…→ q*n*.
Step 3: Ignore all monorelation subqueries (since they are already processed by OVQPs at local sites).
Step 4: for $I = 1$ to n, repeat Step5 to Step8 [n is the total number of irreducible subqueries].
Step 5: Choose the irreducible subquery q*i* involving the smallest fragments.
Step 6: Determine the best strategy, pairs of (*F, S*), for q*i*. [*F* represents a fragment and *S* represents the processing sites.]
Step 7: For each pair (*F, S*), transfer the fragment *F* to the corresponding processing site *S*.
Step 8: Execute the query q*i*. Check for the termination of *for* loop.
Step 9: Output the result.
End.

The query optimization is basically done in step5 and step6 in distributed INGRES algorithm. In this algorithm, subqueries are produced depending on several components and their dependency order. Since the relation involved in a subquery may be fragmented and stored at different sites, the subquery cannot be further subdivided. The main difficulty in step6 is to determine how to execute the subquery by selecting the fragments that will be transferred to sites where the processing will take place. For an n-relation subquery, fragments from $n - 1$ relations must be transferred to the site(s) of fragments of the remaining relation, and then replicated there. Further, the non-transferred relations may be divided into several equal fragments to increase parallelism. This approach is called **fragment-and-replicate**, which performs a substitution of fragments rather than of tuples as in centralized INGRES algorithm. The selection of non-transferred relations and the number of processing sites depends on the objective function and the communication scheme. The choice of the number of processing sites is a trade-off between the total time measure and the response time measure. If the number of sites increases, the response time decreases by parallelism, but the total time increases which leads to higher communication cost. The cost of producing the result is ignored here.

The distributed INGRES algorithm is characterized by a limited search of the solution space, an optimization decision taken for each step without considering the consequences of that decision on global optimization. The exhaustive search approach in which all possible strategies are evaluated to determine the best one is an alternative to the limited search approach. However, dynamic optimization strategy is beneficial, because the exact sizes of the intermediate result relations are known.

11.5.2 Distributed R* Algorithm

The objective of distributed R* algorithm [Selinger and Adiba, 1980] is to reduce the total cost measure which includes the local processing costs and communication costs. Distributed R* algorithm uses an exhaustive search approach to select the best strategy. Although exhaustive search incurs an overhead, it can be worthwhile, if the query is executed frequently. The implemented version of distributed R* algorithm does not support fragmentation or replication; thus, it involves relations as its basic units. This algorithm chooses one site as the **master site** where the query is initiated. The query optimizer at the master site can take all inter-site decisions, such as the selection of the execution sites, the fragments which will be used, and the method for transferring data. The other participating sites that store the relations involved in the query, called **apprentice sites**, can make

the remaining local decisions and generate local access plans for the query execution. Distributed R* algorithm is implemented in the following way.

Distributed R* Algorithm:
Input: Query tree for the given query
Output: Optimum execution strategy (cost is minimum)
Begin
Step 1: For each base relation *Ri* in the query tree, repeat Step2 and Step3.
Step 2: Find each access path of *Ri* and determine the cost of each access path.
Step 3: Determine the access path with the minimum cost.
Step 4: For $I = 1$ to n, repeat Step5.
Step 5: For each order ($Ri1, Ri2, \ldots, Rin$) of the relation *Ri*, build the best strategy ((($APi1, \ldots, \bowtie Ri2$) $\bowtie Ri3$) $\bowtie \ldots\ldots\ldots \bowtie Rin$) and compute the cost of the strategy.
Step 6: Select the strategy with minimum cost.
Step 7: For $J = 1$ to M, repeat step8. [M is the total number of sites in the distributed system storing a relation involved in the query]
Step 8: Determine the local optimization strategy at site *J* and send.
End

In this algorithm, based on database statistics and formulas used to estimate the size of intermediate results, and access path information, the query optimizer determines the join ordering, the join algorithms and the access path for each fragment. In addition, it also determines the sites of join results and the method of data transfer between sites. For performing the join between two relations, there may be three candidate sites. These are the site of the first relation, the site of the second relation, and the site of the result relation. Two inter-site data transfer techniques are supported by distributed R* algorithm: **ship-whole** and **fetch-as-needed**. In ship-whole technique, the entire relation is moved to the join site and stored in a temporary relation before performing the join operation. If the join algorithm is merge join, the incoming tuples are processed in a pipeline mode as they arrive at the join site. In this case, the relation need not be stored in the temporary relation. The fetch-as-needed technique is the same as semijoin operation of an internal relation with an external node. In this method, the external relation is sequentially scanned, and for each tuple the join value is sent to the site of the internal relation. The internal tuples that match with the value of the external tuple are selected there, and then the selected tuples are sent to the site of the external relation.

Ship-whole technique requires larger data transfer but fewer message transfers than fetch-as-needed technique. Obviously, ship-whole technique is beneficial for smaller relations. On the other hand, the fetch-as-needed technique is beneficial if the relation is larger and the join operation has good selectivity. In fetch-as-needed technique, there are four possible join strategies for joining of the external relation with the internal relation over an attribute value. These join strategies and their corresponding costs are calculated in the following, where LC denotes the local processing cost, which involves CPU cost and I/O cost, CC represents the communication cost, and A denotes the average number of tuples of internal relation S that matches with the value of one tuple of external relation R.

Strategy 1: Ship the entire external relation to the site of the internal relation.
In this case, the tuples of external relation R can be joined with the tuples of internal relation S, as they arrive. Therefore,

Total cost measure = LC (retrieving card(R) tuples from R) + CC (size(R)) + LC (retrieving A tuples from internal relation S) * card(R).

Strategy 2: Ship the entire internal relation to the site of the external relation.
In this case, the tuples of internal relation S cannot be joined with the tuples of external relation R, as they arrive. The entire internal relation S has to be stored in a temporary relation. Hence,

Total cost measure = LC (retrieving card(S) tuples from S) + CC (size(S)) + LC (storing card(S) tuples in T) + LC (retrieving card(R) tuples from R) + LC (retrieving A tuples from T) * card(R).

Strategy 3: Fetch tuples of the internal relation as needed for each tuple of the external relation.
In this case, for each tuple in R, the join attribute value is sent to the site of the internal relation S. Then the A number of tuples from S that *matches* with this value are retrieved and sent to the site of R to be joined as they arrive. Therefore,

Total cost measure = LC (retrieving card(R) tuples from R) + CC (length(join attribute value)) * card(R) + LC (retrieving A tuples from S) * card(R) + CC (A * length(S)) * card(R).

Strategy 4: Move both relations to a third site and compute the join there.
In this case, the internal relation is first moved to the third site and stored in a temporary relation. Then, the external relation is moved to the third site and the join is performed as the tuples arrive. Hence,

Total cost measure = LC (retrieving card(S) tuples from S) + CC (size(S)) + LC (storing card(S) tuples in T) + LC (retrieving card(R) tuples from R) + CC (size(R)) + LC (retrieving A tuples from T) * card(R).

The cost of producing the final result is not considered here. In the case of distributed R* algorithm, the complexity increases as the number of relations increases, as it uses the exhaustive search approach based on several factors such as join order, join algorithm, access path, data transfer mode and result site.

11.5.3 SDD-1 Algorithm

SDD-1 query optimization algorithm [Bernstein et al., 1981] is derived from the first distributed query processing algorithm "**hill-climbing**". In hill-climbing algorithm, initially a feasible query optimization strategy is determined, and this strategy is refined recursively until no more cost improvements are possible. The objective of this algorithm is to minimize an arbitrary function which includes the total time measure as well as response time measure. This algorithm does not support fragmentation and replication, and does not use semijoin operation. The "hill-climbing" algorithm works in the following way, where the inputs to the algorithm are the query graph, location of relations involved in the query, and relation statistics.

First, this algorithm selects an initial feasible solution, which is a global execution schedule that includes all inter-site communication. It is obtained by computing the cost of all the execution strategies that transfer all the required relations to a single candidate result site and selecting the minimum-cost strategy. Assume that this initial solution is S. The query optimizer splits S into two strategies, $S1$ followed by $S2$, where $S1$ consists of sending one of the relations involved in the join

operation to the site of the other relation. There the two relations are joined locally, and the resulting relation is sent to the chosen result site. If the sum of the costs of the execution strategies $S1$ and $S2$ and the cost of local join processing is less than the cost of S, then S is replaced by the schedule of $S1$ and $S2$. This process is continued recursively until no more beneficial strategy is obtained. It is to be noted that if an n-way join operation is involved in the given query, then S will be divided into n subschedules instead of two.

The main disadvantage of this algorithm is that it involves higher initial cost, which may not be justified by the strategies produced. Moreover, the algorithm gets stuck at a local minimum-cost solution and fails to achieve the global minimum-cost solution.

In SDD-1 algorithm, lots of modifications to the hill-climbing algorithm have been done. In SDD-1 algorithm, semijoin operation is introduced to improve join operations. The objective of SDD-1 algorithm is to minimize the total communication time and the local processing time, and response time is ignored here. This algorithm uses the database statistics in the form of **database profiles**, where each database profile is associated with a relation. SDD-1 algorithm can be implemented in the following way.

In SDD-1 algorithm also, one initial solution is selected, and it is refined recursively. One post-optimization phase is added here to improve the total time of the selected solution. The main step of this algorithm consists of determining and ordering semijoin operations to minimize cost. There are four phases in SDD-1 algorithm, known as **initialization, selection of beneficial semijoin operations, result site selection** and **post-optimization**. In the initialization phase, an execution strategy is selected that includes only local processing. In this phase, a set of beneficial semijoins BS are also produced for use in the next phase. The second phase selects the beneficial semijoin operations from BS recursively and modifies the database statistics and BS accordingly. This step is terminated when all semijoin operations in BS are appended to the execution strategy. The execution order of semijoin operations is determined by the order in which the semijoins are appended to the execution strategy. In the third phase, the result site is decided based on the cost of data transfer to each candidate site. Finally, in the post-optimization phase, the semijoin operations are removed from the execution strategy, which then involves only the relations stored at the result site. This is necessary because the result site is chosen after all semijoin operations have been ordered.

CHAPTER SUMMARY

- Query processing involves the retrieval of data from the database, in the contexts of both centralized and distributed DBMSs. A Query processor is a software module that performs processing of queries. Distributed query processing involves four phases: query decomposition, query fragmentation, global query optimization and local query optimization.
- The objective of query decomposition phase is to transform a query in a high-level language on global relations into a relational algebraic query on global relations. The four successive steps of query decomposition are normalization, analysis, simplification and query restructuring.
- In the normalization step, the query is converted into a normalized form to facilitate further processing in an easier way. The two possible normal forms are conjunctive normal form and disjunctive normal form.
- The objective of the analysis step is to reject normalized queries that are incorrectly formulated or contradictory.

» In the simplification step, all redundant predicates in the query are detected and common subexpressions are eliminated to transform the query into a simpler and efficiently computable form.

» In the query restructuring step, the query in the high-level language is rewritten into an equivalent relational algebraic form.

» In query fragmentation phase a relational algebraic query on global relations is converted into an algebraic query expressed on physical fragments, called fragment query, considering data distribution in distributed databases.

» A query typically has many possible execution strategies, and the process of choosing a suitable one for processing a query is known as query optimization. Both centralized query optimization and distributed query optimization are very important in the context of distributed query processing.

» The optimal query execution strategy is selected by a software module, known as query optimizer, which can be represented by three components: search space, cost model and optimization strategy.

» The search space is defined as the set of equivalent query trees for a given query, which can be generated by using transformation rules.

» There are three different kinds of optimization strategies, known as static optimization strategy, dynamic optimization strategy and randomized optimization strategy.

» In a distributed system, the cost of processing a query can be expressed in terms of the total cost measure or the response time measure.

EXERCISES

Multiple Choice Questions

(i) Which of the following is not a step of centralized query execution?
   a. Query decomposition
   b. Code generation
   c. Query optimization
   d. Access plan
   e. None of these.

(ii) Which of the following statements is correct?
   a. Code generator produces the equivalent relational algebraic query from a high-level language (SQL) query
   b. Query processor checks the validity of the high-level language query
   c. Query optimizer generates the optimum execution strategy
   d. All of these.

(iii) Which of the following is an objective of distributed query processing?
   a. To reduce the total cost of time
   b. To reduce the response time
   c. All of these
   d. None of these.

(iv) Which of the following is not a phase of distributed query processing?
   a. Query decomposition
   b. Query fragmentation
   c. Local query optimization
   d. Access plan generation.

(v) Which of the following is not a step of query decomposition?
   a. Normalization
   b. Analysis
   c. Fragmentation
   d. Query restructuring.

(vi) Which of the following statements is incorrect?
   a. The objective of normalization is to transform the query into a normalized form
   b. The objective of the analysis step is to reject normalized queries that are incorrectly formulated or contradictory.
   c. In the simplification step, the query is converted into a simpler form
   d. All of these
   e. None of these.

(vii) The normalized attribute connection graph is used in
   a. Normalization step
   b. Simplification step
   c. Query restructuring step
   d. Analysis step.

(viii) Redundant predicates are eliminated in
  a. Normalization step
  b. Simplification step
  c. Query restructuring step
  d. Analysis step.

(ix) Reduction techniques are used in
  a. Query decomposition
  b. Query fragmentation
  c. Global query optimization
  d. Local query optimization.

(x) Which of the following techniques is used in reduction of horizontal fragmentation?
  a. Reduction with selection operation
  b. Reduction with join operation
  c. None of these
  d. All of these.

(xi) Which of the following is done in reduction of mixed fragmentation?
  a. Removal of empty relations
  b. Removal of useless relations
  c. Removal of useless join operations
  d. None of these
  e. All of these.

(xii) Which of the following is not included in the query execution cost of a distributed DBMS?
  a. CPU cost
  b. I/O cost
  c. Storage cost
  d. Communication cost.

(xiii) Which of the following components is used to represent a query optimizer?
  a. Cost model
  b. Search space
  c. Optimization strategy
  d. All of these.

(xiv) Which of the following statements is correct?
  a. In a linear join tree, at least one operand of each node is a base relation
  b. In a bushy join tree, all operands must be base relations.
  c. Bushy join trees reduce the size of the search space
  d. Linear join trees facilitate parallelism.

(xv) Which of the following is not an optimization strategy?
  a. Static
  b. Dynamic
  c. Randomized
  d. All of these
  e. None of these.

(xvi) Which of the following is included in a distributed cost model?
  a. Cost functions
  b. Database statistics
  c. Base data
  d. All of these
  e. None of these.

(xvii) The join selectivity factor between two relation is
  a. Greater than 1
  b. Less than 0
  c. Is a real number between 0 and 1
  d. None of these.

(xviii) Which of the following techniques is not used by INGRES algorithm?
  a. Reconstruction
  b. Detachment
  c. Substitution
  d. None of these.

(xix) In simple join strategy,
  a. The ordering of join operations are optimized
  b. Join operations are replaced by semijoin operations
  c. Semijoin operations are replaced by join operations
  d. All of these.

(xx) The full reducer exists for
  a. Cyclic queries
  b. Tree queries
  c. Both tree and cyclic queries
  d. Neither cyclic nor tree queries.

(xxi) The objective of distributed INGRES algorithm is to reduce
  a. Total cost measure
  b. Response time measure
  c. Both total cost and response time measures
  d. None of these.

(xxii) Which of the following data transfer techniques is supported by distributed R* algorithm?
  a. Ship-whole
  b. Fetch-as-needed
  c. Both ship-whole and fetch-as-needed
  d. None of these.

(xxiii) The objective of R* algorithm is to reduce
  a. Total cost measure
  b. Response time measure
  c. Both total cost and response time measures
  d. None of these.

(xxiv) Which of the following is supported by distributed SDD-1 algorithm?
   a. Fragmentation of relations
   b. Replication of relations
   c. Both fragmentation and replication of relations
   d. None of these.

(xxv) Which of the following is supported by implementation version of distributed R* algorithm?
   a. Fragmentation of relations
   b. Replication of relations
   c. Both fragmentation and replication of relations
   d. None of these.

(xxvi) Semijoin is required to
   a. Reduce network traffic
   b. Reduce memory usage
   c. Increase speed
   d. None of these.

Review Questions

1. Discuss the objectives of distributed query processing.
2. Explain the phases of centralized query processing.
3. What is query decomposition? Is it the same for centralized and distributed databases? Write down the steps of query decomposition.
4. Discuss the different normal forms with examples in the context of query processing.
5. Explain with a suitable example how a given query is analysed in the analysis step.
6. Describe what actions are performed in query restructuring phase and why?
7. What are the objectives of query fragmentation phase? Explain with example how these are achieved in the case of horizontal fragmentation.
8. Discuss, giving an example, the reduction of mixed fragmentation.
9. Differentiate between linear join tree and bushy join tree.
10. Compare and contrast different optimization strategies.
11. Describe the centralized INGRES algorithm.
12. Comment on query optimization in distributed databases.
13. Why global query optimization is difficult in distributed DBMS?
14. In distributed DBMS context, compare and contrast simple join strategy and semijoin strategy.
15. Describe the distributed R* query optimization algorithm.
16. Explain the distributed SDD-1 query optimization algorithm.
17. How do you define cardinality, size and distinct values for a fragment? What is the effect of these parameters on a semijoin operation?
18. Explain with an example the rationale of semijoin reduction for query optimization in distributed databases.
19. What are the different matrices for estimating the cost of a relational algebraic operation?
20. Comment on the following:

    "Semijoin can be used to reduce the cost of a join operation in a distributed environment".
    "Join operation should be done after selection, projection and union operations for distributed database systems".

21. Consider the following schemas:

    EMP = (ENO, ENAME, TITLE)
    ASG = (ENO, PNO, RESP, DUR)
    PROJ = (PNO, PNAME, BUDGET, LOC)

    Further consider the following query:

    Select ENAME, PNAME from EMP, ASG, PROJ where EMP.ENO = ASG.ENO and ASG.PNO = PROJ.PNO and (TITLE = "ELECT.ENG" or ASG.PNO < "P3").

    Draw the generic query tree (canonical tree). Transform the generic query tree to an optimized reduced query tree.

22. Simplify the following query using the idempotency rules:

    SELECT ENO FROM ASG WHERE (NOT (TITLE = "PROGRAMMER") AND (TITLE = "PROGRAMMER" OR TITLE = "ELECT.ENG") AND NOT (TITLE = "ELECT.ENG")) OR ENAME = "J. Das".

23. Simplify the following query using the idempotency rules:

    SELECT ENO FROM ASG WHERE RESP = "Analyst" AND NOT (PNO = "P2" OR DUR = 12) AND PNO ≠ "P2" AND DUR = 12, considering
    ASG (ENO, PNO, RESP, DUR).

24. Consider the following relations EMP (eno, ename, title) and ASG (eno, pno, resp, dur). Write down suitable queries in SQL-like syntax and in relational algebra for finding the names of employees who are managers of any project. Is the query optimized? If not optimize it.

25. Consider the following schemas:
    EMP (ENO, ENAME, TITLE)
    PROJ (PNO, PNAME, BUDGET)
    ASG (ENO, PNO, RESP, DUR)
    The relation PROJ is horizontally fragmented as
    PROJ1 = $\sigma_{PNO \leq 'P3'}$ (PROJ)
    PROJ2 = $\sigma_{PNO > 'P3'}$ (PROJ)

    Transform the following query into a reduced query on fragments.

    SELECT BUDGET FROM PROJ, ASG WHERE PROJ.PNO = ASG.PNO AND ASG.PNO = "P4".

26. Consider the following schemas:

    EMP (ECODE, ENAME, DESIGN, SALARY, PNO)
    PROJECT (PNO, PNAME, BUDGET, PSTATUS)

    The relations PROJECT and EMP are horizontally fragmented as follows:

    PROJ1 = $\sigma_{PNO \leq 'P3'}$ (PROJECT)
    PROJ2 = $\sigma_{PNO > 'P3'}$ (PROJECT)
    EMP1 = $\sigma_{PNO \leq 'P3'}$ (EMP)
    EMP2 = $\sigma_{PNO > 'P3'}$ (EMP)

    Draw the generic query tree for the following query and convert it into a reduced query on fragments.

    SELECT PNAME FROM PROJECT, EMP WHERE PROJECT.PNO = EMP.PNO AND PSTATUS = "Outside".

27. Consider the following schemas:

    EMP (ENO, ENAME, TITLE, SALARY)
    ASG (ENO, PNO, RESP, DUR)

    The relations EMP and ASG are horizontally fragmented as follows:

    $ASG1 = {}_{ENO \leq 'E4'}(ASG)$
    $ASG2 = {}_{ENO > 'E4'}(ASG)$
    $EMP1 = {}_{ENO \leq 'E4'}(\prod_{ENO, ENAME, SALARY}(EMP))$
    $EMP2 = {}_{ENO > 'E4'}(\prod_{ENO, ENAME, SALARY}(EMP))$
    $EMP3 = \prod_{ENO, TITLE}(EMP)$

    Draw the generic query tree for the following query and convert it into a reduced query on fragments.

    SELECT * FROM ASG, EMP WHERE ASG.ENO = EMP.ENO AND TITLE = "Developer".

28. Consider the following schemas:

    EMP (ENO, ENAME, TITLE)
    PROJ (PNO, PNAME, BUDGET)
    ASG (ENO, PNO, RESP, DUR)

    Using INGRES algorithm, detach and substitute for the query "Retrieve names of employees working on the ORACLE project", considering there are four different values for ENO, which are E1, E2, E3 and E4.

12 Distributed Database Security and Catalog Management

This chapter focuses on distributed database security. Database security is an integral part of any database system, and it refers to the preventive measures to ensure data consistency. In this chapter, view management in distributed DBMSs is discussed in detail. The chapter also introduces authorization control and data protection both in centralized and in distributed contexts. In a DBMS, all security constraints and semantic integrity constraints are stored in the system catalog. In this context, catalog management in distributed DBMS also is represented in this chapter.

The outline of this chapter is as follows. Section 12.1 introduces the concept of database security. View management in distributed database context is described in Section 12.2. In Section 12.3, authorization control and data protection are discussed, and Section 12.4 represents semantic integrity constraints. Catalog management in distributed database systems is focused on in Section 12.5.

12.1 Distributed Database Security

Database security is a major concern in the context of both centralized DBMS and distributed DBMS in preserving the consistency of the database. It is an integral part of any database system. Maintaining data security in a distributed DBMS is more complicated than in a centralized DBMS, because the underlying network is to be made secure. Database security is closely related to database integrity. Data integrity refers to the correctness of data and is represented in terms of integrity constraints. Database security refers to the preventive measures for maintaining data integrity. In a centralized DBMS, database security management typically includes view management, data protection and authorization control, and semantic integrity constraints. All of these can be defined as database rules that the system will automatically enforce. Data that do not satisfy these rules are considered as invalid data, and they are not permitted to be stored in the database. Similarly, any violations of these database rules are automatically detected, and the violating data are rejected by the database system. In distributed DBMS context, database security also involves network security, as information are exchanged between different nodes in the system via a computer network.

The rules that are used to control data manipulation is a part of the database administration, and generally these are defined by the database administrator (DBA). The cost of enforcing semantic integrity constraints in a centralized DBMS is very high in terms of resource utilization, and it can be prohibitive in a distributed environment. The semantic integrity constraints are stored in the system catalog. In a distributed DBMS, the global system catalog contains data distribution details in addition to all information that are stored in a centralized system catalog. Maintaining system catalog in a distributed database environment is a very complicated task.

12.2 View Management

A **view** can be defined as a virtual relation or table that is derived from one or more base relations. A view does not necessarily exist in the database as a stored set of data values; only view definitions are stored in the database. When a DBMS encounters a reference to a view, it can be resolved in two different ways. One approach is to look up the definition of the view and translate the definition into an equivalent request against the source tables of the view, and then perform that request. This process is known as **view resolution.** Another alternative approach stores the view as a temporary table in the database and maintains the changes of the view as the underlying base tables are updated. This process is called **view materialization.**

In relational data model, a view is derived as the result of a relational query on one or more base relations. It is defined by associating the name of the view with the relational query that specifies it. Views provide several advantages, which are listed in the following:

- **Improved Security** – The major advantage of a view is that it provides data security. It is possible to grant each user the privileges to access the database only through a small set of views that contain the appropriate data for the user, thereby restricting and controlling each user's access to the database.
- **Reduced Complexity** – A view simplifies queries by deriving data from more than one table into a single table and thus converts multi-table queries into single-table queries.
- **Data sharing** – The same underlying base tables can be shared by different users in different ways through views. Therefore, it provides the facility for data sharing and for customizing the appearance of the database.
- **Convenience** – Using views only that portion of the data is presented to the users, in which they are interested. Thus, it reduces the complexity from the user's point of view and provides greater convenience.

In some cases, a view defined by a complex, multi-table query may take a long time to process, because view resolution must join the tables together every time the view is accessed.

Example 12.1

Let us consider the relational schema Employee which is defined in Chapter 1, example1.1. Using the SQL query "Retrieve the name, designation and department number of all employees whose designation is Manager from Employee relation", a view with the name V1 can be created as follows:

Create view V1 (Name, Designation, Deptno) as select ename, designation, deptno from Employee where designation = "Manager"

The execution of the above SQL statement will store a view definition into the system catalog with the name V1. The result generated against the user request "**Select * from V1**" is shown in figure 12.1.

In this case, the view V1 can be manipulated as a base relation.

Example 12.2

Let us consider the query "Retrieve the name, designation and department name of all employees whose designation is Manager". The query involves the base relation Department and the view V1. Using SQL, the query can be expressed as follows:

V1

Name	Designation	Deptno
J. Lee	Manager	10
D. Davis	Manager	30
A. Sasmal	Manager	20

Fig. 12.1 Result of Query Involving View V1

Select Name, Designation, dname from V1, Department where Designation = "Manager" and V1.Deptno = Department.deptno

Assume that a view V2 is created based on the above SQL query. Now any request against the view V2 will be converted into a data request against the base relations upon which the view V2 is created. Thus, the above query will be modified as listed below, and it will produce the output shown in figure 12.2.

V2

Name	Designation	Dname
J. Lee	Manager	Development
D. Davis	Manager	Maintenance
A. Sasmal	Manager	Production

Fig. 12.2 Result of Query Involving View V2

Select ename, designation, dname from Employee, Department where designation = "Manager" and Employee.deptno = Department.deptno

12.2.1 View Updatability

All updates made to a base relation must be reflected in all views that encompass that base relation. Similarly, it is expected that if a view is updated, then the changes will be reflected in the underlying base relations, but all views cannot be manipulated in such a way. Views are classified into two categories, known as **updatable views** and **non-updatable views**. The updatability of a view depends on the query expression based on which the view is created. A view is updatable only if the updates to the view can be propagated correctly to the base relations without ambiguity. A view is non-updatable if the corresponding query based on which the view is generated has the following properties.

- The query expression contains DISTINCT, GROUP BY or HAVING clause.
- The query expression includes aggregate functions such as SUM, MAX or MIN.
- The query expression involves more than one base relation or a view which is produced using join, union, intersection or set difference operations.
- If the query contains subqueries.

In addition, no tuple that is added through a view must violate the integrity constraints of the base table.

Example 12.3

The view V1 in example 12.1 is updatable, whereas the view V2 in example 12.2 is non-updatable because the query expression for creating view V2 involves one base relation and one view. The insertion of a new tuple <D. Jones, Manager, 40> into V1 can be propagated to the base relation Employee without any ambiguity. The following view is a non-updatable view.

Create view V3 as select max(salary) from Employee

The above view V3 contains an aggregate function MAX, thus, it is non-updatable. Similarly, the following view V4 is also non-updatable.

Create view V4 as select * from V2

The above view is non-updatable because it is derived from another view V2 which is non-updatable. It must be noted that views derived by join are updatable if they include the keys of the base relations.

12.2.2 Views in Distributed DBMS

Views are the same in the context of both distributed DBMS and centralized DBMS, but in distributed DBMS views can be derived from fragmented relations stored at different sites. When a view is created, its name and the corresponding definition are stored into the system catalog as base relation descriptions. In a distributed database system, view definitions can be stored centrally at one site or partially or fully replicated depending on the degree of site autonomy offered by the system [Williams et al., 1982]. In the centralized approach, all view definitions in the distributed system are stored in a single central site. In the partially replicated approach, some of the view definitions are replicated and stored at different sites of the distributed system. All view definitions are available at all sites of a distributed system in the fully replicated approach.

If a query is generated against a view and the view definition is not available at that site, then remote accessing to the view definition is necessary. The processing of requests is costly against a view that is derived from distributed relations. Moreover, it is likely that several different queries are generated almost at the same time based on the same view. To handle this situation, an approach called **view materialization** is used. In this approach, the view is stored as a temporary table in the database, called **snapshot** or **materialized view,** when the view is first referenced. A snapshot represents a particular state of the database and is therefore static. Snapshots do not reflect updates to base relations and these are only useful for the applications that do not require the most updated version of the database. Snapshots are very useful in many new applications such as data warehousing, replication server, data visualization and mobile systems.

The difficulty with view materialization is in maintaining the changes of the view while the base tables are being updated. The process of updating a snapshot in response to changes to the underlying data is called **view maintenance**. The objective of view maintenance is to apply only those changes that are necessary to keep the view current. However, this can be done when the system is idle.

12.3 Authorization and Protection

Authorization and data protection are two very important aspects of database security, and they are closely related to each other. **Protection** is required to prevent unauthorized disclosure, alteration or destruction of data. Data protection is generally provided by file systems in both centralized and

distributed operating systems. A well-known method for data protection is data encryption. Data encryption is suitable for both storing the information on the disk as well as data exchange through the network. In data encryption method, the data is encoded into ciphertext, and it can be decoded into original data as required by using an algorithm. The algorithms used for encoding and decoding data are called key. There are various standard data encryption techniques such as symmetric key encryption and asymmetric key encryption. A complete presentation of protection schemes is available in [Fernandez et al., 1981].

The granting of rights or privileges that enable users to have legitimate access to a system or a system's objects is called **authorization.** Authorization ensures that only the authorized users are accessing the data. An authorization control must have the ability to identify authorized users and thereby to restrict unauthorized accessing of data. Authorization control was being provided by operating systems for a long time, and recently by distributed operating systems as a service of the file system. Generally, a centralized approach is used for authorization control. In this approach, the centralized control creates database objects and provides permission to other users to access these objects. Database objects are identified by their external names. Another aspect of authorization is that different users can have different privileges on the same database objects in a database environment.

In a relational DBMS, the DBA uniformly controls authorizations using high-level constructs. A DBA is usually responsible for allowing users to have access to database objects by creating individual user accounts. In a distributed DBMS, a decentralized authorization control approach is more suitable than a centralized authorization control approach. Authorization control in distributed DBMSs is more complicated because database objects and users may be distributed among different sites.

12.3.1 Centralized Authorization Control

In authorization control, three main entities are involved: **users, database objects** and **operations** that can be performed on database objects. Therefore, an authorization can be defined as a triple (user, operation, object), which indicates that the specified user has the right to perform an operation of the specified type on the specified object. Authorization control checks whether a given triple can be permitted to process or not. A user is introduced into the system by the pair (username, password). The **username** is used to identify each user uniquely in the system, and the **password** is used to authenticate the user. Both of these information are required to login into the database environment, thereby preventing unauthorized users from entering into the database system.

A database is a collection of database objects. In relational data model, a database object can be defined by its type, which is expressed as (view, relation, tuple, attribute), as well as by its content using selection predicates. A **right or privilege** represents a relationship between a user and a database object for a particular set of operations. In SQL, an operation is defined by a high-level statement such as INSERT, DELETE, UPDATE, ALTER, SELECT, GRANT, REFERENCES or ALL and privileges are defined using GRANT and REVOKE statements. The keyword **public** is used to mean all users in the system. In centralized authorization control, the DBA has all privileges on all database objects, and he/she is allowed to grant (or take away) permissions to (from) other users.

In decentralized authorization control, the creator of a database object is the owner of that object. The owner has the right to grant permission to other users to perform certain operations on the database object. In this case, the authorization control is distributed among the owners of database objects. If the owner grants the GRANT permission on a database object to some other user, then that specified user can subsequently grant permissions to other users on this specified database

object. The revoking process must be recursive, and to perform revoking the system must maintain a hierarchy of grants per database object where the owner of the database object is the root.

The privileges on database objects are stored in the system catalog as authorization rules. There are various ways to store authorization rules in the system catalog; authorization matrix is one of them. In the **authorization matrix,** each row represents a user (or a subject), each column represents a database object and each cell, which is a pair (user, database object), indicates the authorized operations for a particular user on a particular database object. The authorized operations are specified by their operation type, and in the case of SELECT operation type, sometimes the selection predicates are also mentioned to further restrict the access to database objects. The authorization matrix can be stored in three different ways: by row, by column and by element. When the authorization matrix is stored by row, each user is associated with the list of objects that can be accessed with the corresponding access rights. Similarly, when the authorization matrix is stored by column, each object is associated with the list of users who can access it with the corresponding access rights. In both of the above two approaches, authorizations can be enforced efficiently, but the manipulation of access rights per database object is not efficient as all user profiles must be accessed. This disadvantage can be overcome if the authorization matrix is stored by element, that is, by a relation (user, database object, right).

This approach provides faster access in right manipulation per user per database object. A sample authorization matrix is shown in figure 12.3.

	Employee	**Department**	**Project**
User 1	SELECT	INSERT	ALTER
User 2	INSERT	DELETE	SELECT WHERE project-type = "inside"
User 3	SELECT	INSERT	UPDATE
User 4	UPDATE	SELECT	DELETE

Fig. 12.3 A Sample Authorization Matrix

Example 12.4

The following SQL statement grants SELECT, UPDATE and DELETE permissions on the database object Employee to all users.

GRANT SELECT, UPDATE, DELETE ON Employee TO PUBLIC.

Similarly, the SQL statement

GRANT ALL PRIVILEGES ON Employee TO user2

grants all permission on the database object Employee to the user user2.

GRANT SELECT ON Employee TO user1 WITH GRANT OPTION.

The above SQL statement allows user1 to grant SELECT permission on Employee database object to other users. The following SQL statement takes away SELECT and INSERT rights on Employee database object from the user user4.

REVOKE SELECT, INSERT ON Employee FROM user4.

12.3.2 Distributed Authorization Control

Authorization control in distributed DBMSs involves remote user authentication, management of distributed authorization rules, view management and control of user groups. Management of views in distributed database systems was already discussed in Section 12.2.2. In a distributed database system, users may identify themselves from any sites; therefore, remote user authentication is necessary to prevent unauthorized accessing.

There are two different ways for implementing remote user authentication as described in the following:

- In the first approach, the information that are required for authenticating users (i.e., username and password) is replicated and stored at all sites in the system catalog.
- In the second case, each user in the distributed system is identified by its home site. Whenever a user wants to login from a remote site a message is sent to its home site for authentication, and then the user is identified by the home site.

The first approach is costly in terms of catalog management. The second approach is more reasonable, because it restricts each user to identifying him(her)self at the home site. However, site autonomy is not preserved in the second approach.

Authorization rules are used to restrict the actions performed by users on database objects after authentication. In a distributed DBMS, the allocation of authorization rules is a part of system catalog, and enforcement of these rules can be implemented in two different ways. These are full replication and partial replication as listed in the following:

- In full replication, authorization rules are replicated and stored at all sites of the distributed system.
- In the case of partial replication, the authorization rules are replicated at the sites where the referenced database objects are distributed.

The full replication approach allows authorization to be checked at the beginning of the compile time, but this approach is costly in terms of data replication. The latter approach is better if localization of reference is high, but it does not allow distributed authorization to be controlled at compile time.

Views can be considered as composite database objects; thus, granting access to a view translates into granting access to the underlying objects. The authorization granted on a view depends on the access rights of the view creator on the underlying objects. If the view definitions and authorization rules are fully replicated, then this translation becomes simpler, and it can be done locally. If the view definitions and their underlying objects are distributed, then the translation is difficult. In this case, the association information can be stored at the site of each underlying object.

To simplify the authorization control and to reduce the amount of data stored, individual users are typically divided into different classes known as groups, which are all granted the same privileges. Like centralized DBMS, in a distributed DBMS all users as a group can be referred to as public, and all users at a particular site may be referred to as public@site*i*. The management of groups in distributed systems is very difficult, because the users belonging to the same group may be distributed at different sites, and the access to a database object can be granted to several groups, where the groups can also be distributed. There are several alternative solutions as listed below.

- One solution is full replication of all authorization rules although maintaining replication is quite expensive in this case.
- Another solution enforces access privileges by performing a remote query to the sites that holds the group definitions.
- Another alternative solution is to replicate a group definition at each site containing a database object that can be accessed by users of that group.

The last two solutions decrease the degree of autonomy. It is obvious that full replication of authorization information simplifies authorization control, as it can be done at compile time. However, the overhead cost for maintaining replicas is significant, if there are many sites in the distributed system.

12.4 Semantic Integrity Constraints

Database consistency is ensured by a set of restrictions or constraints, known as **semantic integrity constraints.** A database state is said to be consistent if the database satisfies a set of constraints. Maintaining data consistency involves several DBMS modules such as concurrency control, recovery management, protection and authorization control, and semantic integrity control. The responsibility of a semantic integrity control module is to verify whether the updates performed by application programs violate the consistency of database states. If any such violation is detected, the semantic integrity control module rejects the updates or performs some actions to compensate the effects of updates. Semantic integrity constraints are closely connected to integrity constraints in the sense that more semantic information can be captured about applications by using integrity rules or constraints.

There are two different types of semantic integrity constraints: **structural constraints** and **behavioral constraints.** Structural constraints represent basic semantic properties inherent to a model, whereas behavioural constraints control application behaviour. It has been suggested [Florentin, 1974] that integrity constraints can be expressed using assertions of predicate calculus. In this approach, one can easily declare and modify complex integrity assertions. The main difficulty in supporting automatic semantic integrity control is the cost of checking assertions. Moreover, enforcing integrity assertions generally requires access to a large amount of data that is not involved in the database updates. Therefore, implementation of automatic semantic integrity control can be prohibitive in distributed database systems. The management of semantic integrity constraints is not discussed in this context, as we are mainly concerned about the distributed DBMSs.

In a distributed DBMS context, it is very difficult to design a semantic integrity control subsystem. The two major problems are the definition and storage of assertions, and enforcement of these assertions. An assertion is a predicate expressing an integrity constraint that must be satisfied by the database. Domain constraints and referential integrity constraints are special type of assertions. In a distributed system, as assertions can involve data stored at different sites, their storage must be decided to minimize the cost of integrity checking. Assertions can be classified into three different categories as listed below.

- **Individual assertions** – These assertions involve a single variable and a single relation, and refer only to tuples that can be updated independently. Domain constraints are individual assertions.

» **Set-oriented assertions** – This type of assertions may involve a single relation and multiple variables or multiple relations and multiple variables. Referential integrity constraints are examples of set-oriented assertions.

» **Assertions involving aggregates** – This type of assertions contain aggregate functions and therefore testing of these assertions is too costly.

The enforcement of assertions in a distributed system depends on the category of the assertion, the type of update and the nature of the site where the update is issued. The site where the update is issued may or may not store the updated relation or some of the relations involved in the integrity assertions.

» **Enforcement of individual assertions** – There are two methods for the enforcement of individual assertions in a distributed DBMS. If the update is an insert operation, all individual assertions can be enforced at the site where the update is issued. If the update is a delete or a modify operation, it is sent to the sites where the relation resides and update should be performed there. The query processor performs update operation for each fragment. The resulting tuples at each site are combined into one temporary relation in the case of a delete statement, whereas in the case of a modify statement, the resulting tuples are combined into two temporary relations. Each site involved in the distributed update enforces the assertions relevant to that site.

» **Enforcement of set-oriented assertions** – Two cases are considered here. In the case of multiple-variable single-relation set-oriented assertions, the update (may be insert, delete or modify) is sent to the sites where the relation resides, and they return one or two temporary relations after performing the update, as in case of individual assertions. These temporary relations are then sent to all sites storing the specified relation, and each site enforces the assertions locally. Each site storing the specified relation then sends a message to the site where the update is issued indicating whether the assertions are satisfied or not. If any assertion is not satisfied at any site, then it is the responsibility of the semantic integrity control subsystem to reject the entire update program. In the case of multiple-variable multiple-relation set-oriented assertions, the enforcement of assertions is done at the site where the update is issued. Hence, after performing update at all fragments of the involved relations, all results are centralized at the site where the update is issued, called **query master site**. The query master enforces all assertions on the centralized result, and if any inconsistency is found the update is rejected. On the other hand, if no inconsistencies are found, the resultant tuples are sent to the corresponding sites where the fragments of the relation reside.

» **Enforcement of assertions involving aggregates** – The testing of these assertions is the costliest, because it requires the calculation of aggregate functions. To efficiently enforce these assertions, it is possible to produce compiled assertions that isolate redundant data. These redundant data, known as **concrete views,** can then be stored at each site where the associated relation resides [Bernstein and Blaustein, 1982].

Example 12.5

The following is an example of an individual assertion that involves a single attribute salary and a single relation Employee.

Create assertion salary_constraint check (not exists (select * from Employee where salary < 20000)).

The above assertion imposes a restriction that salary should be greater than 20,000 and this condition must hold true in every database state for the assertion to be satisfied.

Example 12.6

The following is an example of a set-oriented assertion that involves two variables salary and deptno and two relations Employee and Department.

Create assertion salarydept_constraint check (not exists (select * from Employee, Department where salary < 20000 and Employee.deptno = Department.deptno)).

The above assertion also represents a referential integrity constraint involving two relations Employee and Department.

12.5 Global System Catalog

In a distributed database system, the global system catalog stores all the information that are required to access data correctly and efficiently, and to control authorization. The global system catalog is used for the following purposes.

- **Processing applications** – Catalog information are used in processing application to check whether access plans are valid and the users have the appropriate access privileges.
- **Mapping applications** – Using catalog information data referenced by application programs at different levels of transparency are mapped into physical data.
- **Optimizing applications** – Catalog information are used to produce an access plan for query execution. Typically, the information required for producing access plans include data allocation, statistical information and access methods available at each site.

The updates of data definition (e.g., modification of database schemas, changes in authorization rules) modify the catalog information.

12.5.1 Contents of Global System Catalog

A global system catalog contains the following information depending on the architecture of the distributed system.

(i) The names of all global relations and their corresponding attributes.

(ii) The details of fragmentation schema, that is, the selection predicates for horizontal fragments, the names of attributes for vertical fragments and both of these for mixed fragments.

(iii) The details of data allocation, that is, the mapping between fragments and physical databases at each site of the distributed system, and the names of local data stored at each site.

(iv) The access method descriptions (e.g., number and types of index, hash function) which are locally available at each site.

(v) The statistical information about the data.

(vi) Consistency information, which include data protection and authorization rules, integrity constraints, security constraints etc.

12.5.2 Catalog Management in Distributed Systems

Catalog management in a distributed system is closely related with the degree of local autonomy at each site. There are three alternative approaches for data allocation and catalog management in distributed systems, known as centralized, fully replicated, and fragmented and distributed.

Centralized – In this approach, the global system catalog is stored at a single site. All other sites in the distributed system access catalog information from this central site. This approach is very simple, but it has several limitations. The system is vulnerable to the failure of the central site. The availability and reliability are very low in this case. The major drawback of this approach is that it decreases site autonomy.

Fully replicated – In this case, catalog information are replicated at each site of the distributed system. The availability and reliability are very high in this approach. The site autonomy increases in this case. The disadvantage of this approach is the information update overhead of the global system catalog.

Fragmented and distributed – This approach is adopted in the distributed system R* to overcome the drawbacks of the above two approaches. In this approach, there is a local catalog at each site that contains the metadata related to the data stored at that site. Thus, the global system catalog is fragmented and distributed at the sites of a distributed system. For database objects created at any site (the birth-site), it is the responsibility of that site's local catalog to store the definition, replication details and allocation details of each fragment or replica of that object. Whenever a fragment or a replica is moved to a different location, the local catalog at the corresponding site must be updated. The birth-site of each global database object is recorded in each local system catalog. Thus, local references to data can be performed locally. On the other hand, if remote reference is required, the system-wide name [system-wide name is discussed in **Chapter 5, Section 5.5.1**] reveals the birth-site of the database object, and then the catalog information are accessed from that site.

CHAPTER SUMMARY

» **Database security** refers to the preventive measures for maintaining data integrity. Database security typically includes view management, data protection and authorization control, and semantic integrity constraints, all of .which can be defined as database rules that the system will automatically enforce.

» A **view** can be defined as a virtual relation or table that is derived from one or more base relations. A data request or an update request against a view can be resolved in two different ways: **view resolution** and **view materialization.** Views can be classified as **updatable views** and **non-updatable views.**

» **Protection** is required to prevent unauthorized disclosure, alteration or destruction of data.

» The granting of rights or privileges that enable users to have legitimate access to a system or a system's objects is called **authorization**. The authorization control in distributed DBMSs involves remote user authentication, management of distributed authorization rules, view management and control of user groups.

» Database consistency is ensured by a set of restrictions or constraints, known as **semantic integrity constraints**. There are two different types of

semantic integrity constraints: **structural constraints** and **behavioural constraints.** Semantic integrity constraints can be expressed using assertions.

» An **assertion** is a predicate expressing an integrity constraint that must be satisfied by the database. Assertions can be classified into three different categories: **individual assertions, set-oriented assertions** and **assertions involving aggregates.**

» The global system catalog stores all the information that are required to access data correctly and efficiently and to control authorization. There are three alternative approaches for catalog management in a distributed system: centralized, fully replicated, and fragmented and distributed.

EXERCISES

Multiple Choice Questions

(i) Data integrity refers to
  a. Correctness of data
  b. Preventive measure for maintaining data consistency
  c. Both consistent and inconsistent data
  d. None of these.

(ii) In a distributed system, database security involves
  a. Data protection and authorization control
  b. Semantic integrity constraints
  c. View management
  d. All of these
  e. None of these.

(iii) What is data about data called?
  a. System catalog
  b. Data dictionary
  c. Metadata
  d. None of these.

(iv) Which of the following is a disadvantage of a view?
  a. Data security
  b. Data sharing
  c. Convenience
  d. None of these.

(v) Data request against a view is resolved by
  a. View resolution
  b. View materialization
  c. None of these
  d. All of these.

(vi) A view is updatable if
  a. It is derived from more than one base relation, but view definition contains the keys of the base relations
  b. View definition contains an aggregate function
  c. View definition contains subqueries
  d. None of these.

(vii) Which of the following statements is correct?
  a. A snapshot represents a particular state of the database
  b. A snapshot is static
  c. A snapshot does not reflect updates to base relations
  d. None of these
  e. All of these.

(viii) The granting of privileges that enable users to have legitimate access to a system or a system's objects is called
  a. Protection
  b. Authorization
  c. Authentication
  d. None of these.

(ix) An authorization can be represented as
  a. A triple of user, operation and object
  b. A pair of username and password
  c. A combination of view, relation, tuple and attributes
  d. All of these
  e. None of these.

(x) In an authorization matrix, the authorized operations of a user on a database object is represented by a
  a. Row
  b. Column
  c. Cell
  d. All of these.

(xi) The manipulation of access rights per database object is efficient if the authorization matrix is stored by
  a. Row
  b. Column
  c. Cell
  d. None of these.

(xii) Which of the following statements is true?
  a. An assertion is a predicate expressing an integrity constraint
  b. A domain constraint is an assertion
  c. A referential integrity constraint is an assertion
  d. None of these
  e. All of these.

(xiii) Referential integrity constraints are
  a. Individual assertions
  b. Set-oriented assertions
  c. Assertions involving aggregates
  d. All of the above.

(xiv) Domain constraints are
  a. Assertions involving aggregates
  b. Individual assertions
  c. Set-oriented assertions
  d. None of these.

(xv) Which of the following catalog management techniques is used in an R* distributed database system?
  a. Centralized
  b. Fully replicated
  c. Fragmented and distributed
  d. None of these.

Review Questions

1. Explain briefly how database integrity is related to database security.
2. What are the objectives of using views instead of base relations?
3. Differentiate between view resolution and view materialization.
4. In what ways are views managed in distributed databases?
5. How is data protection ensured in a distributed DBMS?
6. Explain how authorization rules are managed in a distributed DBMS.
7. Discuss how user groups are handled in a distributed DBMS.
8. Define semantic integrity constraints. Explain how semantic integrity constraints are enforced in a distributed DBMS.
9. Describe the different types of assertions with examples.
10. Discuss how you will manage the storage and allocation of global system catalog in a distributed system.
11. Consider a banking application in which there are three different kinds of users: Manager, Employee, and Customer. Assume that three different relations Account, Empinfo and Branch are maintained by the bank. Managers have all privileges on all relations, customers have the rights to read the information from all relations, and employees have all privileges on Account relation, SELECT right on Branch relation for that particular branch and SELECT right on Empinfo relation. Design an authorization matrix.
12. Assume that in the above exercise, there are several branches of the bank, which are physically distributed in many locations. Explain how you will enforce the above authorization rules.
13. Consider example 12.5. Assume that the Employee relation is distributed in three sites S1, S2 and S3 respectively, and an insert operation is issued at site S1 with the tuple (E12, J. Miller, Programmer, 10,000, 10, V12). Explain how assertions will be enforced in this case.
14. Repeat the above exercise for the update operation (E09, S.Sen, Analyst, 75,000, 40, V10) generated at site S4.
15. Consider example 12.6. Assume that the Employee relation is distributed in three sites S1, S2 and S3 respectively, and the Department relation is distributed in two sites S4 and S5 respectively. An insert operation is issued at site S1 with the tuple (E12, J. Miller, Programmer, 10,000, 10, V12). Explain how assertions will be enforced in this case.

This page is intentionally left blank

13 Mobile Databases and Object-Oriented DBMS

The fundamentals of mobile databases and object-oriented database management systems (OODBMSs) are introduced in this chapter. Portable computing devices coupled with wireless communications allow clients to access data in databases from virtually anywhere and at any time. However, communications are still restricted owing to security reasons, cost, limited battery power and other factors. Mobile databases overcome some of these limitations of mobile computing. OODBMSs provide an environment where users can avail the benefits of both object-orientation as well as database management systems. The features of an OODBMS proposed by the Object-Oriented Database Manifesto are described here. The benefits and problems of OODBMSs over conventional DBMSs are also discussed in this chapter.

This chapter is organized as follows. Section 13.1 introduces the basic concepts of mobile databases. In Section 13.2, basic object-oriented concepts are discussed. The details of OODBMSs are represented in Section 13.3.

13.1 Mobile Databases

Mobile computing has been gaining an increased amount of attention owing to its usefulness in many applications such as medical emergencies, disaster management and other emergency services. Recent advancements in portable and wireless technology have led to mobile computing becoming a new dimension in data communication and processing. Wireless computing creates a situation where machines no longer have fixed locations and network addresses. The rapid expansion of cellular, wireless and satellite communications makes it possible to access any data from anywhere, at any time. This feature is especially useful to geographically dispersed organizations. However, business etiquette, practical situations, security and costs may still limit communications. Furthermore, energy (battery power) is a scarce resource for most of the mobile devices. These limitations are listed in the following:

- The wireless networks have restricted bandwidth. The cellular networks have bandwidths of the order of 10 Kbps, whereas wireless local area networks have bandwidths of the order of 10 Mbps.
- The power supplies (i.e., batteries) in mobile stations have limited lifetimes. Even with the new advances in battery technology, the typical lifetime of a battery is only a few hours, which is reduced further with increased computation and disk operations.
- Mobile stations are not available as widely as stationary ones, because of power restrictions. Owing to the same reason, the amount of computation that can be performed on mobile stations is restricted.

» As mobile stations can move, additional system functionality is required to track them. Moreover, managing mobility is a complicated task, because it requires the management of the heterogeneity of the base stations where the topology of the underlying network changes continuously.

Therefore, it is not possible to establish online connections for as long as the users want and whenever they want. In this context, mobile databases offer a solution for some of these limitations.

A **mobile database** is a database that is portable and physically separate from a centralized database server, but capable of communicating with that server from remote sites allowing the sharing of corporate data. Mobile databases help users by facilitating the sharing of corporate data on their laptops, PDA (Personal Digital Assistant) or other internet access devices from remote sites. A mobile database environment has the following components:

» **Corporate database server and DBMS** – The corporate database server stores the corporate data, and the DBMS is used to manage it. This component provides corporate applications.

» **Remote database server and DBMS** – This server stores the mobile data, and the DBMS is used to control mobile data. This component provides mobile applications.

» **Mobile database platform** – This component involves laptops, PDAs or any other internet access devices.

» **Both-way communication link** – To establish a connection between the corporate and mobile DBMSs, a both-way communication link is required.

The general architecture of a mobile database environment is illustrated in figure 13.1.

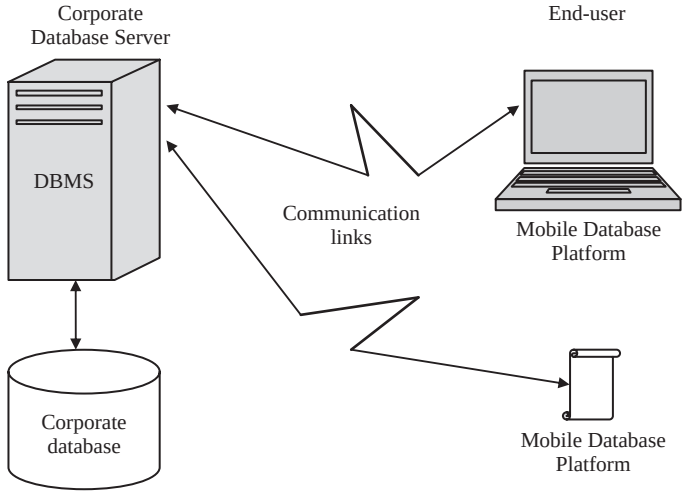


Fig. 13.1 General Architecture of a Mobile Database Environment

In some mobile applications, the mobile user may logon to a corporate database server from his/her mobile device and can work with data there, whereas in other applications the user may download (or upload) data from (to) the corporate database server at the remote site, and can work with it on a mobile device. The connection between the corporate and the mobile databases

is established for short periods of time at regular intervals. The main issues associated with mobile databases are management of mobile databases and the communication between the mobile and corporate databases.

13.1.1 Mobile DBMS

A mobile DBMS provides database services for mobile devices, and now most of the major DBMS vendors offer mobile DBMSs. A mobile DBMS must have the capability to communicate with a range of major relational DBMSs, and it must work properly with limited computing resources, which is required to match with the environment of mobile devices. The additional functionalities that are required for a mobile DBMS are listed in the following:

(i) The major function of a mobile DBMS is to manage mobile databases and to create customized mobile applications.

(ii) It must be able to communicate with the centralized database server through wireless or internet access.

(iii) A mobile DBMS can capture data from various sources such as the internet.

(iv) It has the ability to analyse data on a mobile device.

(v) A mobile DBMS must be able to replicate and synchronize data on the centralized database server and on the mobile device.

Currently, most of the mobile DBMSs provide only prepackaged SQL functions for mobile applications, rather than supporting any extensive database querying or data analysis.

Mobile computing can be considered as a variation of distributed computing from a data management viewpoint. Mobile databases can be distributed under two possible scenarios.

(i) The entire database is distributed among the wired (fixed) components, possibly with full or partial replication. A base station (or fixed host managers or corporate database server) manages its own database with additional functionality for locating mobile units, and additional query and transaction management features to meet the requirements of mobile environments.

(ii) The entire database is distributed among the wired and wireless components. Data management responsibility is shared among the base stations or fixed hosts and the mobile units.

13.2 Introduction to Object-Oriented Databases

Object-oriented database systems (OODBs) evolved from a requirement to satisfy the demand for a more appropriate representation and modeling of real-world entities. OODBs provide a much richer data model than conventional relational databases. The object-oriented database paradigm is based on several basic concepts, namely object, object identifier, attributes, methods, class, class hierarchy, inheritance, overriding, and late binding (dynamic binding). In the object-oriented data model, any real-world entity is represented by one modeling concept, the **object**. An object has a **state** and a **behaviour** associated with it. The state of an object is defined by the value of its properties (attributes). Properties can have primitive values (such as strings and integers) or non-primitive objects. A **non-primitive object** would in turn consist of a set of properties. Therefore, objects can

be recursively defined in terms of other objects; such objects are known as **complex objects**. Each **attribute** has an **access specifier**, namely, public, private or protected, that limits the visibility of the attribute. The behaviour of an object is specified by a set of methods, which manipulates the state of the object. The state and behaviour encapsulated in an object are invoked from the external world only through explicit message passing. Each object is identified by a unique system-defined identifier, known as **object identifier (OID)**, that distinguishes each object from all other objects in an object-oriented database system.

Objects with the same properties and behaviours are grouped into classes. An object can be an instance of only one class or an instance of several classes. Classes are organized in class hierarchies. The process of forming a superclass is referred to as **generalization**, and the process of forming a subclass is called **specialization**. A **subclass** inherits properties and methods from a **superclass**, and in addition, a subclass may have specific properties and methods. This is the principle of substitutability, which helps to take advantage of code reusability. In some systems, a class may have more than one superclass, called **multiple inheritance**, whereas in others it is restricted to only one superclass, called **single inheritance**. Most models allow overriding of inherited properties and methods. **Overriding** is the substitution of a property domain with a new domain or the substitution of a method implementation with a different one. The subclass can override the implementation of an inherited method or instance variable by providing an alternative definition or implementation of the base class. The ability to apply the same methods to different classes, or rather the ability to apply different methods with the same name to different classes, is referred to as **polymorphism**. The process of selecting the appropriate method based on an object's type is called **binding**. If the determination of the object's type is carried out at run-time, then it is called **dynamic binding** or **late binding**. On the other hand, if the object's type is decided before run-time (i.e., at compile time), then it is called **static binding** or **early binding**.

Database technology focuses on the static aspects of information storage, whereas software engineering concentrates on modeling the dynamic aspects of software. Currently, it is possible to combine both of these technologies that allow the concurrent modeling of both data and the processes acting upon the data. Two such technologies are **Object-Oriented Database Management Systems (OODBMSs)** and **Object–Relational Database Management Systems (ORDBMSs)**. The basic concepts of OODBMS are described in the following section.

13.3 Object-Oriented Database Management Systems

An **object-oriented database** is a persistent and sharable collection of objects based on an object-oriented data model. The **OODBMS** is used to manage object-oriented databases. In other words, OODBMS is a combination of object orientations and database handling capabilities. In 1990, [Zdonik and Maier] had presented a threshold model for OODBMS, which stipulated the following minimum criteria that should be satisfied by an OODBMS.

- An OODBMS must provide database functionality.
- It must support object identity.
- An OODBMS must provide encapsulation.
- It must support objects with complex states.

An OODBMS provides the flavors of object-oriented concepts as well as database management services.

13.3.1 Features of OODBMS

The Object-Oriented Database System Manifesto proposes 13 mandatory features for OODBMS based on two criteria [Atkinson et al., 1989]. The first criterion is that OODBMS should be an object-oriented system, and the second criterion is that it should be a DBMS. The first eight rules are applicable for object-oriented features and the last five rules are applicable for DBMS characteristics. These features are listed in the following:

(i) **Support for complex objects** – In an OODBMS, it should be possible to build complex objects by applying constructors to basic objects. The minimal set of constructors consists of SET, TUPLE and LIST (or ARRAY). Furthermore, object constructors must be orthogonal.

(ii) **Support for object identity** – All objects must have a unique identity that is independent of its attribute values.

(iii) **Support for Encapsulation** – In an OODBMS, proper encapsulation is achieved by ensuring that programmers have access only to the interface specification of methods, and the data and implementation of these methods are hidden in the objects. However, there may be cases where the enforcement of encapsulation is not required.

(iv) **Types or classes must be supported** – An OODBMS must support either the concept of types or the concept of classes. The database schema in an object-oriented system comprises of a set of classes or a set of types. However, it is not a requirement that the system automatically maintain the extent of a type.

(v) **Types or classes must be able to inherit from their ancestors** – A subtype or subclass should inherit attributes and methods from its supertype or superclass respectively.

(vi) **Support for dynamic binding** – Methods should apply to objects of different types (overloading). The implementation of a method will depend on the type of the object it is applied to (overriding). To provide this functionality, the system should bind method names dynamically (during run-time).

(vii) **The data manipulation language (DML) must be computationally complete** – The DML for the OODBMS should be a general-purpose programming language.

(viii) **The set of data types must be extensible** – The user must be able to create new user-defined types from the set of pre-defined system types. Moreover, there must be no distinction in the usage between system-defined and user-defined types.

(ix) **Data persistence must be provided** – Like in conventional DBMSs, data must persist in OODBMS after the application that created it has terminated. The user should not have to explicitly move or copy data to make it persistent.

(x) **The OODBMS must be capable of managing very large databases** – An OODBMS must have data distribution mechanisms that are invisible to the user, thereby providing a clear independence between the logical and physical levels of the system.

(xi) **The OODBMS must support concurrent users** – An OODBMS should provide concurrency control mechanisms similar to those in conventional database management systems.

(xii) **The OODBMS must be capable of recovery from hardware and software failures** – An OODBMS should provide recovery mechanisms similar to the conventional database management systems.

(xiii) The OODBMS must provide a simple way of querying data – An OODBMS must provide ad hoc queries facility that is high-level, efficient, and application independent. It is not mandatory for the system to provide a query language, but it can provide a graphical browser instead.

In addition, the manifesto also proposes some optional features such as multiple inheritance, type checking and type inferencing, distribution across a network, design transactions and versions. However, the manifesto does not mention any direct rules for the support of integrity, security, or views.

13.3.2 Benefits of OODBMS

An OODBMS provides several benefits, which are listed in the following:

- **Enriched modeling capabilities** – In an OODBMS, real-world entities can be modeled more appropriately and more realistically. As an OODBMS uses object-oriented concepts, it provides a richer set of semantic constructs than the conventional relational databases.
- **Extensibility** – OODBMS allows the building of new data types using existing data types. The concept of generalization and specialization greatly reduces the redundancy within the system, because it allows the sharing of properties and methods among several classes. Overriding feature allows special cases to be handled easily with minimal impact on the rest of the system. Moreover, the reusability of classes promotes faster development and easier maintenance of the database and its applications.
- **Query language is more expressive** – In an OODBMS, navigational access from one object to the next object is very common, and it is more suitable for handling recursive queries, parts explosion etc. The Object Data Management Group (ODMG) standard specifies a declarative language based on an object-oriented form of SQL, which is convenient for both programmers and end-users. A single-language interface between the DML and the programming language overcomes the impedance mismatch.
- **Support for schema evolution** – In OODBMS, the tight coupling of data and applications makes schema evolution more feasible. Generalization and inheritance permit the schema to be more intuitive and better-structured.
- **Support for long-duration transactions** – Some OODBMSs use a different protocol to handle the types of long-duration transactions that are common in many advanced database applications.
- **Improved performance** – There are several benchmarks that suggest that OODBMSs provide significant performance improvements over relational DBMSs.

13.3.3 Disadvantages of OODBMS

Although an OODBMS provides several benefits over relational data model, it has a number of disadvantages also, as listed below.

- **Complexity** – The increased functionalities provided by an OODBMS such as schema evolution, long-duration transactions and version management make it extremely complex. The complexity of an OODBMS increases the cost of the product and makes it difficult to understand.

- **Lack of universal data model** – There is no universal data model for an OODBMS, and most models lack the theoretical foundation. There is no standard object-oriented query language, although ODMG has specified an object query language (OQL), which has become a de facto standard.
- **Lack of experience** – The use of OODBMSs is very limited in comparison with relational DBMSs.
- **No support for security and views** – Most OODBMSs do not provide a view mechanism. In conventional DBMS, views provide several benefits, such as security and data independence, that are not possible in an OODBMS environment. Moreover, most OODBMSs do not provide appropriate security mechanisms.
- **Query optimization compromises encapsulation** – In OODBMSs, to access the database efficiently, query optimization requires an understanding of the underlying implementation of the system, thereby compromising the concept of encapsulation.
- **Locking at object level may impact performance** – Many OODBMSs use lock-based concurrency control techniques. If locking is done at the object level, locking of inheritance hierarchy may be problematic, which may affect performance.

CHAPTER SUMMARY

- A **mobile database** is a database that is portable and physically separate from a centralized database server but capable of communicating with that server from remote sites allowing the sharing of corporate data. A mobile database environment has four components. These are corporate database server and DBMS, remote database and DBMS, mobile database platform, and both-way communication link.
- A mobile DBMS provides database services for mobile devices.
- An object-oriented database is a persistent and sharable collection of objects based on object-oriented data model.
- An **object-oriented DBMS (OODBMS)** is used to manage object-oriented databases, and provides both the features of object-orientation and database management systems.

EXERCISES

Multiple Choice Questions

(i) Which of the following statements is false?
   a. Mobile databases are portable
   b. Mobile databases are physically separated from the central database server
   c. All of the above
   d. None of the above.

(ii) Which of the following statements is true?
   a. An object is a real-world entity
   b. An object is an instance of a class
   c. All of the above
   d. None of the above.

(iii) The behaviour of an object is specified by
   a. A set of methods
   b. A set of attributes
   c. A set of methods and attributes
   d. None of the above.

(iv) Which of the following statements is false?
   a. OID is used to uniquely identify each object in the object-oriented data model
   b. OID is used to uniquely identify each object, and it is user-defined

c. OID is used to uniquely identify each object, and it is system-defined
d. All of the above.

(v) In late binding, an object's type is decided
a. Before run-time
b. At compile-time
c. During run-time
d. None of the above.

(vi) Which of the following statements is true?
a. The attributes of a complex object must be primitive
b. The attributes of a complex object may be non-primitive
c. A complex object can contain other objects.
d. Both b. and c.

(vii) Which of the following statements is false?
a. A class can have more than one superclass
b. A class can have only one superclass
c. A class can have more than one subclass
d. All of the above.

(viii) Which of the following mechanisms is not supported by an OODBMS?
a. Complex object
b. View
c. Dynamic Binding
d. Object identity.

(ix) Which of the following is supported by an OODBMS?
a. Long-duration transactions
b. Concurrency control
c. Version management
d. All of the above.

(x) Data in an OODBMS is
a. Persistent
b. Non-persistent
c. Both a. and b.
d. None of the above.

(xi) The process of forming a superclass is known as
a. Specialization
b. Generalization
c. Inheritance
d. Aggregation.

(xii) Mobile computing can be considered as a variation of
a. Parallel computing
b. Centralized computing
c. Distributed computing
d. None of these.

(xiii) If a class has more than one superclass, it is called
a. Polymorphism
b. Overriding
c. Single inheritance
d. Multiple inheritance

(xiv) Which of the following statements is correct?
a. A subclass can have extra attributes and methods in addition to its superclass's
b. A subclass can have only extra attributes in addition to its superclass's
c. A subclass can have only extra methods in addition to its superclass's
d. A superclass can have extra attributes and methods in addition to its subclass's.

(xv) Polymorphism is the ability to apply
a. Same methods to different classes
b. Different methods with the same name to different classes
c. All of these
d. None of these.

Review Questions

1. Define a mobile database. Describe the architecture of a mobile database environment.
2. Write a short note on mobile DBMS.
3. Discuss the basic object-oriented concepts.
4. Differentiate between simple objects and complex objects.
5. Differentiate between early binding and late binding.
6. Define OODBMSs. Describe the features of OODBMSs.
7. Explain the advantages of OODBMS.

14 Distributed Database Systems

This chapter focuses on two popular distributed database systems SDD-1 and R*. Architectures, transaction management, concurrency control techniques and query processing of these distributed database systems are discussed in this chapter.

The outline of this chapter is as follows. Section 14.1 introduces the basic concepts of SDD-1 distributed database system. The architecture of SDD-1 is introduced in Section 14.2. Section 14.2 also describes the concurrency control, reliability, catalog management and query processing techniques of SDD-1 distributed database system. In Section 14.3, another distributed database system R* is presented. The architecture of R* system is described in Section 14.4, and the query processing of R* is discussed in Section 14.5. The transaction management of R* distributed database system is illustrated in Section 14.6.

14.1 SDD-1 Distributed Database System

SDD-1 is a distributed database management system developed by Computer Corporation of America between 1976 and 1978. Like in any other distributed database management system, the data in SDD-1 database is physically distributed among a number of nodes connected by a computer network, but it provides a single logical interrelated data view to the users. It provides all the facilities of any modern database management system. Owing to decentralized processing, SDD-1 distributed database management system is more reliable and flexible than centralized database management systems. Parallel processing capability is also an added advantage of the SDD-1 distributed system, but sometimes it is very difficult to develop algorithms that can exploit parallelism. Therefore, the techniques used to implement centralized DBMS are extended in the context of SDD-1 distributed DBMS. The principal disadvantages of SDD-1 distributed database system are communication cost and the maintenance overhead.

SDD-1 distributed database system supports the relational data model. An SDD-1 database consists of a number of (logical) relations, where each relation is partitioned into a number of subrelations called **logical fragments,** which are the units of data distribution. A logical fragment is derived from a relation by using two different steps. Initially, the relation is horizontally divided into a number of subrelations by using a simple predicate or selection operation of relational algebra, and then it is again vertically partitioned into logical fragments by using a projection operation. To facilitate the reconstruction of a relation from its logical fragments, a unique tuple identifier is attached to each logical fragment. A logical fragment can be stored in one or several sites in an SDD-1 distributed database system, and it may be replicated. The allocation of logical fragments is done during database design. A stored copy of a logical fragment at a site is called a **stored fragment.** SDD-1 distributed database system provides data distribution transparency to the users; thus, users are unaware of the details of fragmentation, replication and distribution of data.

14.2 General Architecture of SDD-1 Database System

The general architecture of SDD-1 is a collection of three independent virtual machines: **Transaction Modules or Transaction Managers (TMs), Data Modules or Data Managers (DMs)** and a **Reliable Network (RelNet)** [Bernstein et al., 1980]. DMs are responsible for managing all data that are stored in an SDD-1 database system. The execution of any transaction in the SDD-1 database system is controlled by the TMs. The RelNet facilitates the necessary communications among different sites that are required to maintain an SDD-1 distributed database system. The architecture of SDD-1 distributed database system is illustrated in figure 14.1.

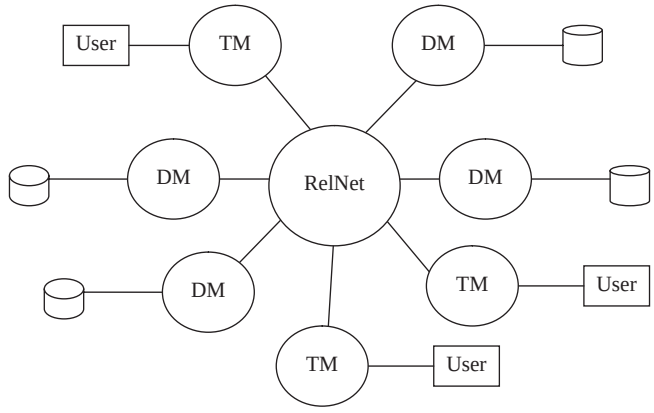


Fig. 14.1 Architecture of SDD-1 Distributed Database System

Data Module (DM) – In an SDD-1 distributed database system, each DM controls local data that are stored at that local site. Hence, each DM is a back-end DBMS that responds to commands from TMs. Generally, each DM responds to four types of commands from TMs, which are listed in the following:

(i) Reading data from the local database into the local workspace at that DM assigned to each transaction.

(ii) Moving data between workspaces at different DMs.

(iii) Manipulate data in a local workspace at the DM.

(iv) Writing data from the local workspace into the permanent local database stored at that DM.

Transaction Module (TM) – Each TM in an SDD-1 database system plans and manages the execution of transactions in the system. A TM performs the following tasks:

(i) Fragmentation – The TM translates queries on relations into queries on local fragments and determines which instances of stored fragments should be accessed to execute database transactions.

(ii) Concurrency Control – To control the concurrent access of data items, the TM synchronizes a transaction with all other active transactions in the system.

(iii) Access Planning – The TM converts the transaction into a parallel program that can be executed cooperatively by several DMs.

(iv) Distributed Query Execution – The TM coordinates the execution of the compiled access plan of the distributed query exploiting parallelism whenever possible.

The Reliable Network (RelNet) – This module interconnects TMs and DMs and provides the following services:

(i) Guaranteed delivery – RelNet guarantees that the message will be delivered even if the recipient is down at the time of the message sent, and even if the sender and the receiver are never up simultaneously.

(ii) Transaction Control – This module also ensures that either updates at all the multiple DMs are posted or none at all, thus, maintaining atomicity of transactions.

(iii) Site Monitoring – RelNet keeps information regarding the site failures in the system and informs other sites about it.

(iv) A Global Network Clock – It provides a virtual clock which is roughly synchronized at all the sites in the system.

This architecture divides the SDD-1 distributed database into three subsystems, namely, database management, management of distributed transactions and distributed DBMS reliability with limited interactions.

The basic unit of user computation in SDD-1 is the transaction. A transaction essentially corresponds to a program in a high-level host language with several DML statements sprinkled within it. In SDD-1 database system, the execution of each transaction is supervised by a TM (called transaction coordinator also) and proceeds in three phases, known as read, execute and write. Each of these phases deals with individual problems in the system. The read phase deals with concurrency control, the execute phase deals with distributed query execution, and the write phase deals with the execution of updates at all replicas of modified data. In **read phase,** the transaction coordinator (TC) (transaction manager of the site where the query is initiated) determines which portion of the (logical) database is to be read by the transaction, called the **read set** of the transaction. In addition, the TC decides which stored fragments (physical data) are to be accessed to obtain the data and then issues read commands to the corresponding DMs. To maintain the consistency of the data, the TC instructs each DM to set aside a private copy of the fragment for use during subsequent processing phases. The private copies obtained by the read phase are guaranteed to be consistent even though the copies reside at distributed sites. As the data are consistent and the copies are private, any operation on the data can be performed freely without any interference from other transactions. Each DM sets aside the specified data in a private workspace, and for each DM the private workspace is implemented by using a differential file mechanism so that data are not actually copied. A **page map** is a function that associates a physical storage location with each page and it behaves like a private copy, because pages are never updated in original. Whenever a transaction wants to update a page, a new block of secondary storage is allocated, and the modified page is written there. These pages cannot be modified by other transactions, because if another transaction wants to update them, it has to allocate a new physical storage location and write the updated page on it.

The second phase, called the **Execute phase,** implements distributed query processing. In this phase, a distributed program that takes as input the distributed workspace created by the read phase is compiled and an access plan for the execution is generated and executed. The execution of the program is supervised by the TC, and it involves several DMs at the different sites. During the execute phase, all actions are performed in the local workspaces of the transaction. The output of the program is a list of data items to be written into the database (in the case of update transactions) or displayed to the user (in the case of retrievals). This output list is produced in a workspace, not in the permanent database.

The final phase, called the **write phase,** either writes the modified data into permanent database and/or displays retrieved data to the user. In the write phase, the output list produced by the transaction is broadcast to the "relevant" DMs as write messages. A DM is relevant if it contains a physical copy of some logical data item that is referenced in the output list. The updated data are written in SDD-1 database using a write-all approach. The atomicity of transactions is preserved by using a specialized commit protocol that allows the commitment of transactions in case of involved site failures.

The three-phase processing of transactions in SDD-1 neatly partitions the key technical challenges of distributed database management such as distributed concurrency control, distributed query processing and distributed reliability.

14.2.1 Distributed Concurrency Control in SDD-1

SDD-1 distributed database system adopts the concept of serializability to maintain the consistency of concurrent data. To ensure serializability, SDD-1 uses two synchronization mechanisms that are distinctly different from locking. The first mechanism, known as **conflict graph analysis,** is a technique for analyzing "classes" of transactions to detect those transactions that require little or no synchronization. The second mechanism consists of a set of synchronization protocols based on "**timestamp**" that synchronize those transactions that require synchronization. When transactions are designed, a static set of transaction classes is established by assigning a read set and a write set to transactions.

Conflict graph analysis

In conflict graph analysis, it is checked whether conflict transactions can be safely run concurrently or whether a given conflict is troublesome. Two transactions are said to be conflicting if the read set or the write set of one transaction intersects the write set of the other transaction. The read set of a transaction is the portion of the database it reads, and the write set of the transaction is the portion of the database it updates. For example, consider that there are four transaction classes *T*1, *T*2, *T*3 and *T*4 with their corresponding read sets and write sets as follows.

T1: read set R1={x, y, z} write set W1={y}

T2: read set R2={p} write set W2={x}

T3: read set R3={a, b, c} write set W3={b, c}

T4: read set R4={a, b, c} write set W4={a, b}

Figure 14.2 illustrates the conflict graphs for transactions *T*1 and *T*2, and *T*3 and *T*4 respectively.

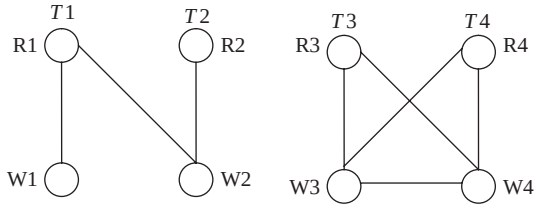


Fig. 14.2 Conflict Graphs

The nodes of a conflict graph represent the read sets and write sets of transactions, and edges represent conflicts among these sets. There is also an edge between the read set and write set of each transaction. Here, the important point is that different kinds of edges require different levels of synchronization, and strong synchronization such as locking is required only for edges that participate in cycles. In figure 14.2, transactions *T*l and *T*2 do not require synchronization as strong as locking, but transactions *T*3 and *T*4 do require it.

It is not appropriate to do conflict graph analysis at run-time because too much inter-site communication would be required to exchange information about conflicts. The conflict graph analysis is done offline in the following way.

The database administrator defines transaction classes, which are named groups of commonly executed transactions, according to their names, read sets, write sets, and the TM at which they run. A transaction is a member of a class if the transaction's read set and write set are contained in the class's read set and write set, respectively. Conflict graph analysis is actually performed on these transaction classes instead of individual transactions. Two transactions are said to be conflicting if their classes are conflicting. The output of the conflict analysis is a table that indicates for each class which other classes are conflicting and how much synchronization is required for each of such conflict to ensure serializability.

In SDD-1, each TC is only permitted to supervise one class of transactions. Each TC synchronizes a transaction with other transactions in its class using a local timestamp mechanism similar to locking. The TC uses the synchronization method(s) specified by the conflict graph analysis and these methods, known as "protocols", are described in the following section.

Timestamp-based protocols

To synchronize two transactions that conflict dangerously, SDD-1 uses timestamp values of transactions. In SDD-1, the order is determined by total ordering of the transaction's timestamps. Each transaction submitted to SDD-1 is assigned a globally unique timestamp value by its own TM, and these timestamp values are generated by concatenating a TM identifier to the right of the network clock time, so that timestamp values from different TMs always differ in their lower-order bits. The timestamp of a transaction is also attached to all read and write commands sent to the DMs. In addition, each read command contains a list of classes that conflict dangerously with the transaction issuing the read, and this list is generated using conflict graph analysis. The DM delays the execution of a read command until it has processed all earlier write commands but not the later write commands for the specified class. The DM–TM communication discipline is called **piping,** and it requires that each TM send its write commands to DMs in their timestamp order. Moreover, the RelNet guarantees that messages are received in the order they are sent.

The idle TMs may send null (empty) timestamp write commands to avoid excessive delays in waiting for write commands. This synchronization protocol corresponds to locking and is designed to avoid "race conditions". However, there are several variations of this protocol depending on the type of timestamp attached to read commands and the interpretation of the timestamps by DMs. For example, read-only transactions can use a less expensive protocol in which the DM selects the timestamp, thereby avoiding the possibility of rejection and reducing delays. An important feature of concurrency control of SDD-1 database is the availability of a variety of synchronization protocols. When all read commands have been processed, consistent private copies of the read set have been set aside at all necessary DMs, and the read phase is complete at this point.

14.2.2 Distributed Query Processing in SDD·1

In SDD-1, the query processing involves two steps, namely, the compilation of the transaction that converts the transaction into a parallel program, and the execution of it. The key part of the compilation is access planning in which an optimization procedure is determined that minimizes the object program's inter-site communication needs while maximizing its parallelism.

Access planning

The simplest way to execute a distributed transaction is to move all of its read set to a single DM and then execute the transaction at that DM. This approach is very simple, but it has two drawbacks. The first one is that the read set of the distributed transaction may be very large, and moving it between sites can be exorbitantly expensive. The second drawback is that parallel processing is put to very little use. To overcome these drawbacks, the access planner generates object programs in two phases, known as **reduction** and **final processing.**

In the reduction phase, data is eliminated from the read set of the transaction as much as is economically feasible without changing the output of the transaction. In the final processing phase, the reduced read set is moved to the designated final DM where the transaction is executed. This technique improves the performance of the above simple approach by decreasing communication cost and increasing parallelism. To reduce the volume of the read sets of transaction, the reduction technique employs selection, projection and semijoin operations of relational algebra. The cost-effectiveness of a semijoin operation depends on the database state. The challenge of access planning is to construct a program of cost-beneficial semijoin operations for a given transaction and a database state. The hill-climbing algorithm is used here to construct a program of cost-beneficial semijoin operations. In hill-climbing algorithm, the optimization starts with an initial feasible solution and recursively improves it. This process terminates when no further cost-beneficial semijoin operations can be found [The details of hill-climbing procedure is described in **Chapter 11, Section 11.5.3.**] A final stage reorders the semijoin operations for execution to take maximal advantage of their reductive power.

Distributed execution

The programs produced by the access planner are non-looping parallel programs and can be represented as data flow graphs. To execute the programs, the TC issues commands to the DMs involved in each operation as soon as all predecessors of the operation are ready to produce output. The output of this phase is either stored as a temporary file at the final DM that is to be written into the permanent database (if it is an update operation) or displayed to the user (if it is a data retrieval request). The execute phase is complete at this point.

14.2.3 Distributed Reliability and Transaction Commitment in SDD·1

In this phase, the output generated in the execute phase is written into the permanent database from the temporary file of the final DM and/or displayed to the user. As the database is distributed, the temporary file is first split into a set of temporary files that list the updates to be performed at each DM. Each of these temporary files is transmitted to the appropriate DM as a write command. In this case, it is necessary to ensure that failures cannot cause some DMs not to perform the updates while the others perform them. The failure of a receiving DM or of the sender must not occur during updates. In SDD-1, this is handled by reliable delivery and transaction control. In addition, it

is necessary to ensure that updates from different transactions are performed in the same effective order at all DMs. This problem is addressed in the subsection *The write rule*.

Guaranteed delivery

In the communication field, there are well-known techniques for reliable message delivery as long as both sender and receiver are up. In ARPANET, errors due to duplicate messages, missing messages, and damaged messages are detected and corrected by the network software. In SDD-l, guaranteed delivery is ensured by an extended communication facility called **reliable network or RelNet,** even when the sender and the receiver are not up simultaneously to exchange messages. The RelNet employs a mechanism called **spooler** for guaranteed message delivery.

A spooler is a process, with access to secondary storage, that serves as a first-in-first-out message queue for a failed site. Any message sent to a failed DM is delivered to its spooler instead. When a message is marked as "guaranteed delivery" and the receiver site is down, it is the responsibility of the reliable delivery mechanism to deliver the message to the spooler of that site. When the message is safely stored at the spooler, the transmission is acknowledged to the sending process, even if the message has not yet reached its destination. To ensure the integrity of the messages, each spooler manages its secondary storage using conventional DBMS reliability techniques. In addition, protection against spooler site failures is achieved by employing multiple spoolers. Messages can be reliably stored as long as one spooler is up and running correctly. Write messages can be sent to failed DMs using reliable delivery. When a failed DM recovers, it can receive its (spooled) write messages to bring its database up-to-date.

Transaction control

Transaction control handles failures of the final DM during the write phase. If the final DM fails after sending some files but not all, the database becomes inconsistent as it stores the partial effects of the transaction. Transaction control rectifies this type of inconsistencies in a timely fashion. The basic technique employed here is a variant of the two-phase commit protocol. During phase1, the final DM transmits the files, but the receiving DMs do not perform the updates. During phase 2, the final DM sends commit messages to involved DMs, and each receiving DM updates its data upon receiving this message. If some DM, say DM*i*, has received files but not a commit message and the final DM fails, then the data manager DM*i* can consult with other DMs. If any DM has received a commit message, the data manager DM*i* performs the updates. If none of the DMs have received a commit message, none of them performs the update, thereby aborting the transaction. This technique offers complete protection against failures of the final DM but is susceptible to multi-site failures.

The write rule

If transactions complete execution at approximately the same time and have intersecting write sets, a mechanism is needed to ensure that their updates are installed in the same order at all DMs. One way to do this is to attach the transaction's timestamp to each write command and make the DMs process write commands according to timestamp order. However, this technique introduces unnecessary delays. A betterment of the above approach can be done by attaching timestamps to the data items as well as to the write commands. Every physical data item in the database is assigned a timestamp value of the time of the most recent transaction that updated it. In addition, each write command carries the timestamp value of the transaction that generated it. When an update is committed at a DM, the following write rule is applied.

For each data item *X* in the write command, the value of *X* is modified at the DM if and only if the timestamp value of *X* is less than the timestamp value of the write command. Thus, recent updates are never overwritten by write commands with the older timestamp value. This has the same effect as processing write commands according to the timestamp order. A major disadvantage of this approach is the apparent high cost of storing timestamps for every data item in the database. However, this cost can be reduced to an acceptable level by caching the timestamps. The write phase is completed when updates are made at all DMs, and at this point the transaction execution is complete.

14.2.4 Catalog Management in SDD-1

In SDD-1 distributed database system, system catalog contains relations and fragment definitions, fragment locations, and usage statistics. It is necessary to maintain system catalog in an efficient and flexible way, because to process any transaction the TM must access system catalog information. The main issues in catalog management are whether to store catalogs redundantly and whether catalog updates should be centralized or decentralized.

One simple approach to maintain the system catalog is to treat catalog information as user data and manage it in the same way as database information. This approach allows catalogs to be fragmented, distributed with arbitrary replication, and updated from arbitrary TMs. This approach has some limitations. First, the performance may degrade because every catalog access incurs general transaction overhead and every access to remotely stored catalogs incurs communication delays. These limitations can be overcome by caching recently referenced catalog fragments at each TM, discarding them if rendered obsolete by catalog updates.

This solution is more appropriate, because the catalogs are relatively static. However, it requires a directory that tells where each catalog fragment is stored. This directory is called the **catalog locator,** and a copy of it is stored at every DM. This solution is appropriate because catalog locators are relatively small and quite static.

14.3 R* Distributed Database System

R* is an experimental distributed database system developed and operational at the IBM San Jose Research Laboratory at California. The star of R* represents an arbitrary number of Rs and it comes from the Kleene star operator (R*=R, RR, RRR,…). The objective of R* is to develop a cooperative distributed database system, where each site is a relational database system with site autonomy. R* distributed database system provides a high-level relational data interface, and it provides data independence by isolating the end-users as much as possible from the underlying storage structure.

In R* distributed database system, data are stored as relations or tables. The R* distributed database system currently does not support fragmentation and replication of data. Moreover, in R* it is not necessary that sites are geographically separated; different sites can be on the same computer. The most important feature of R* distributed database system is that it provides site autonomy. Site autonomy requires that the system be able to expand incrementally and operate continuously. It is possible to add new sites without requiring the existing sites to agree with the joining sites on global data structures or definitions. Another important feature of R* distributed system is that it provides location transparency. The major achievement of R* is that it provides most of the functionalities of a relational database management system in a distributed environment.

14.4 Architecture of R*

The general architecture of R* distributed database system consists of three major components [Astrahan et al., 1976]. These are a **local database management system, a data communication system** that facilitates message transmission, and a **transaction manager** that controls the implementation of multi-site transactions. The local DBMS component is further subdivided into two components: **storage system** and **database language processor.** The storage system of R* is responsible for retrieval and storage of data, and this is known as **Relational Storage System** (**RSS*).** The database language processor translates high-level SQL statements into operations on the storage system of R*. The architecture of R* distributed database system is illustrated in figure 14.3.

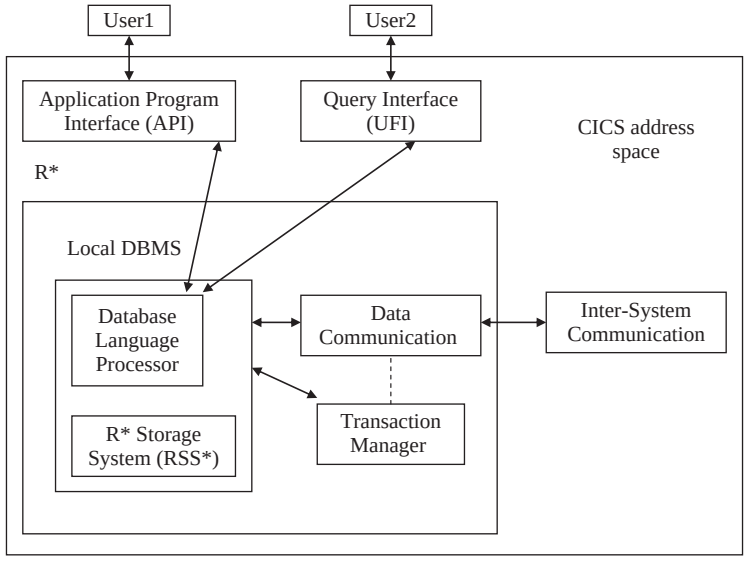


Fig. 14.3 Architecture of R* Distributed Database System

The **relational storage interface** (RSI) is an internal interface that handles access to single tuples of base relations. This interface and its supporting system, the RSS*, is actually a complete storage subsystem that manages devices, space allocation, storage buffers, transaction consistency and locking, deadlock detection, transaction recovery and system recovery. Furthermore, it maintains indexes on selected fields of base relations and pointer chains across relations. The **relational data interface (RDI)** is the external interface that can be called directly from a programming language or used to support various emulators and other interfaces. The **relational data system (RDS),** which supports the RDI, provides authorization, integrity enforcement, and support for alternative views of data. The high-level SQL language is embedded within the RDI, and is used as the basis for all data definition and manipulation. In addition, the RDS maintains the catalogs of external names, as the RSS* uses only system-generated internal names. The RDS contains an optimizer which chooses an appropriate access path for any given request among the paths supported by the RSS*.

In R* distributed system, sites can communicate with each other using **inter-system communication (ISC)** facility of customer information control system (CICS). Each site in R* runs in a CICS address space, and CICS handles I/Os and message communications. Unlike in RelNet, here the

communication is not assumed to be reliable. However, the delivered messages are correct, not replicated, and received in the same order in which they are sent. All database access requests are made through an application program at each local site of the R* system. All inter-site communications are between R* systems at different sites, thus, remote application programs are not required in R* distributed database environment.

Any transaction generated at a particular site is initiated and controlled by the transaction manager at that site (known as transaction coordinator). The transaction manager at that site implicitly performs a begin_transaction, and the implicit end_transaction is assumed when the user completes a session. An R* database system can also be invoked using a user-friendly interface (UFI), and in this case, SQL statements are individually submitted from the UFI to R* storage system. Here each individual SQL statement is considered as an individual transaction. The transaction manager at each site assigns a unique identifier to each transaction originating at that site; this is generated by the concatenation of a local counter value and the site identifier.

In R* database system, for processing any request the "process" model of computation is used. According to this computation model, a process is created for each user on the first request for a remote site and maintained until the end of the application, instead of allocating a different process to each individual data request. Thus, a limited number of processes are assigned to applications, thereby reducing the cost of process creation. User identification is verified only once – at the time of creation of the process. A process activated at one site can request the activation of another process at another site, in turn. Therefore, a computation in R* may require the generation of a tree of processes, all of which belong to the same application. In R* system, communication between processes can be done using sessions, and these sessions are established when the remote process is created and are retained throughout the life of the process. Thus, processes and sessions constitute a stable computational structure for an application in R* distributed database system.

14.5 Query Processing in R*

In R* distributed database system, there are two main steps for query processing known as **compilation** and **execution** [Chamberlin et al., 1981]. These are described in the following.

Compilation – The query can be compiled either statically or dynamically in R* environment. Static compilation is used for repetitive queries, whereas dynamic compilation is used for single-execution queries. In both cases, the high-level SQL statement is converted into an access plan that indicates the order in which the relations are to be accessed, the sites at which the access is to be made, the method to be used for performing each operation, and the access path to be used for retrieving or manipulating tuples in RSS*. In R* environment, more importance is given to repetitive queries rather than single-execution queries.

Query compilation includes the generation and distribution of low-level programs for accessing the RSS* for each high-level SQL query. These programs can be executed repeatedly, and recompilation is required only when definitions of data used by the SQL statement change. To determine whether recompilation is required or not, it is necessary to store the information about the dependencies of the compiled query on data definitions. If the query is interpreted, the determination of an access plan is done for every execution of the query.

In R* distributed database system, there are several options for compilation of a query. In a centralized approach, a single site is responsible for the compilation of all queries generated at any site. This centralized approach is unacceptable, because it compromises the site autonomy.

The compilation of a query can be done at the site where it is generated; this approach is also not acceptable because it does not preserve the local autonomy of other sites. The access plan for a query execution determined by the originating site may be rejected by other participating sites. In recursive compilation approach, each site performs part of the query compilation and asks some other site to compile the remaining subquery. This recursive approach also has the drawback in that it requires negotiation among sites for compilation. The approach that is used for compilation of queries in R* distributed database environment distributes the query compilation responsibilities among a **master site** and **apprentices sites.** The site, where the query is generated, is called the master, and is responsible for the global compilation of the query. The global plan specifies the order of operations and the method that should be used to execute them. The other participating sites, called apprentices, are responsible for selecting the best plan to access their local data for the portion of the global plan that pertains to them. These local access plans must be consistent with the global access plan.

The query compilation at the master site includes the following steps.

» **Parsing of the query** – This step involves syntax verification of SQL query and the resolution of database objects' names, which corresponds to the transformation from print names to system-wide names.

» **Accessing of catalog information** – If the query involves only local data, then the local catalog is accessed; otherwise, catalog information can be retrieved either from local caches of remote data or from remote catalogs at the sites where the data are stored.

» **Authorization checking** – Authorization checking is done to determine whether the user compiling the query has proper access privileges or not. It is performed on local relations, initially at the master site and later at the apprentice sites.

» **Query optimization** – In this step, an optimized global access plan is determined based on I/O cost, CPU cost and communication cost.

» **Plan distribution** – In this step, the full global plan is distributed to all apprentices that specifies the order of operations and the methods to be used for them, together with some guidelines for performing local operations, which may or may not be followed by the apprentices. The global plan also includes the original SQL query, catalog information, and the specification of parameters to be exchanged between sites when the query is executed.

» **Local binding** – The names used in the part of the global plan that is executed at the master site are bound to internal names of database objects at that local site.

» **Storage of plan and dependencies** – The local plan at the master site is converted into a local access module that will be executed at the local instance of the RSS*. This plan is stored together with the related dependencies that indicate the relations and access methods used by the plan.

The master distributes to the apprentices portions of the global plan that contain the original SQL statement, the global plan from which each apprentice can isolate its portion, and the catalog information used by the master. It is to be noted that passing of high-level SQL statement by the master provides more independence and portability, so that the apprentice can generate a different local version of the optimized plan. The steps that are performed by apprentice sites are parsing, local catalog accessing, authorization checking and local access optimization. Finally, local names of database objects are bound, and plans and dependencies are stored.

When all SQL statements of the same application program have been compiled by all apprentice sites, the master requests the transaction manager to commit the compilation considering it as an atomic unit. Here the two-phase commit protocol is used to ensure that the compilation is successful at all sites or at none at all.

Execution – In this step, plans sent to all apprentices in the compilation step are retrieved and executed. The master sends execute requests to all apprentices, which in turn may send the same request to their subordinates, thereby activating a tree of processes. Each intermediate-level process receives result data, which is packed in blocks, from its subordinates, and starts execution as soon as the first results are received. Then the intermediate-level process sends the result to the higher-level processes, also packed in blocks, thus, pipelining is achieved. A superior in the tree may send signals to its subordinates to stop the transmission of results.

Recompilation – In R* system, recompilation is required only when definitions of data used by the SQL statement change. At execution time, a flag in the catalogs is tested to check whether the access module is valid or not. This flag is set to valid after each successful compilation and it becomes invalid when data definitions change. Any change in data definition causes a search through the catalogs that record dependencies to find access methods that involve the object. If the flag is invalid, recompilation is required. First, a local recompilation is attempted, even if it may not be optimal in the global sense, because it is less expensive. However, in some cases global recompilation is necessary, in this case the site where the application was presented acting as the master.

14.6 Transaction Management in R*

A transaction in R* system is a sequence of one or more SQL statements enclosed within begin_ transaction and end_transaction. It also serves as a unit of consistency and recovery [Mohan et al., 1986]. Each transaction is assigned a unique transaction identifier, which is the concatenation of a sequence number and the site identifier of the site at which the transaction is initiated. Each process of a transaction is able to provisionally perform the operations of the transaction in such a way that they can be undone if the transaction needs to be aborted. Also, each database of the distributed database system has a log that is used for recovery. The R* distributed database system, which is an evolution of the centralized DBMS System R, like its predecessor, supports transaction serializability and uses the two-phase locking (2PL) protocol as the concurrency control mechanism. In R*, concurrency control is provided by 2PL protocol where transactions hold their locks until the commit or abort, to provide isolation. The use of 2PL introduces the possibility of deadlocks in R*. The R* distributed database system, instead of preventing deadlocks, allows them to occur and then resolves them by deadlock detection and victim transaction abort.

The log records are carefully written sequentially in a file that is kept in non-volatile storage. When a log record is written, the write can be done synchronously or asynchronously. In synchronous write, called forcing a log record, the forced log record and all preceding ones are immediately moved from the virtual memory buffers to stable storage. The transaction writing the log record is not allowed to continue execution until this operation is completed. On the other hand, in asynchronous write, the record gets written to the virtual memory buffer storage and is allowed to migrate to the stable storage later on. The transaction writing the record is allowed to continue execution before the migration takes place. It is to be noted that a synchronous write increases the response time of the transaction compared to an asynchronous write. The former case is called force-write, and the latter is called, simply, write.

In R* distributed database system, transactions are committed using two variations of the standard two-phase commit protocol. These variations differ in terms of the number of messages

sent, the time for completion of the commit processing, the level of parallelism permitted during the commit processing, the number of state transitions that the protocols go through, the time required for recovery once a site becomes operational after a failure, the number of log records written, and the number of those log records that are force-written to stable storage. The conventional two-phase commit protocol is extended for transaction commitment in R* system, which supports a tree of processes. The **presumed abort (PA)** and the **presumed commit (PC)** protocols are defined to improve the performance of distributed transaction commit.

14.6.1 The Presumed Abort Protocol

In the PA protocol, in the absence of any information about a transaction, the recovery process orders an inquiring subordinate to abort, assuming that the transaction has been aborted. If this deduction is made consistently by the recovery procedure, it is possible to design a two-phase commit protocol in which the coordinator can forget an aborted transaction earlier than in standard two-phase commit protocol. A careful examination of this scenario reveals the fact that it is safe for a coordinator to forget a transaction and to write an abort record immediately after it makes the decision to abort the transaction. This means that the abort record need not be forced, and no acknowledgements need to be sent for aborts. Furthermore, the coordinator need not record the names of the subordinates in the abort record or write an end record after an abort record. Also, if the coordinator notices the failure of a subordinate while attempting to send an abort message to it, the coordinator need not handle the transaction up to the recovery process. The subordinate will find out about the abort when the recovery process at the subordinate's site sends an inquiry message.

The PA protocol is also beneficial for **complete or partial read-only** transactions. A transaction is said to be partially read-only if some processes of the transaction do not perform any updates to the database whereas the others do. A transaction is said to be (completely) read-only if no process of the transaction performs any updates to the database. It is not necessary to know whether a transaction is partially read-only or completely read-only before the transaction starts. If a leaf process receives a prepare message and it finds that it has not done any updates, it simply sends a read vote, releases its locks, and forgets the transaction. The subordinate writes no log records, and it need not know whether the transaction ultimately gets aborted or committed. If all the subordinates send read votes, the coordinator need not enter the second phase of the commit protocol. In this case, the coordinator simply forces a commit record, releases its locks, and forgets about the transaction. When some of the subordinates send read votes and others require updates, then the coordinator enters the second phase, and it is sufficient to commit only the subordinates that enter the ready state.

The PA protocol does not change the performance in terms of log writes and message sending of the two-phase commit protocol with respect to committing transactions.

14.6.2 The Presumed Commit Protocol

In the PC protocol, if no information is available from the master site, the recovery mechanism presumes it to be a commit. The PC protocol is dual to the PA protocol, and this duality leads to the design of an approach in which aborts are to be acknowledged whereas the commits are not. However, there is a problem with this approach regarding this point. Consider the situation when the coordinator fails after sending the prepare message for a transaction and recovers before writing the global commit record for that transaction. In this case, coordinator recovery would undo the transaction while the subordinates would find no information and commit, thereby causing an unacceptable inconsistency. This can be avoided by forcing the prepare record in the log of the coordinator.

Each coordinator must record the names of its subordinates safely before any of the latter could get into the prepared state. Then, when the coordinator site aborts on recovery from a crash that occurred after the sending of the prepare message, the restart process will know whom to inform about the abort message and from which subordinates acknowledgements should be received. These modifications complete the PC protocol. The name arises from the fact that in the no-information case the transaction is presumed to have committed, and hence the response to an inquiry is a commit.

The PC protocol is more efficient than the PA protocol in cases where subordinates perform update transactions successfully, as it does not require acknowledgements and forcing of commit records. On the other hand, the PA protocol is beneficial for read-only transactions, because the additional record with the participant information need not be forced in the log of the coordinator. R* distributed database system provides facilities to select the protocol for each individual transaction. The PA protocol is suggested for read-only transactions, whereas the PC protocol is suggested for all other transactions.

CHAPTER SUMMARY

- SDD-1 is a distributed database management system developed by Computer Corporation of America between 1976 and 1978. It supports relational data model.
- The general architecture of SDD-1 is a collection of three independent virtual machines, namely, the Transaction Modules (TMs), Data Modules (DMs) and a Reliable Network (RelNet).
- In SDD-1 database system, the execution of each transaction is processed in three phases known as read, execute, and write. The read phase deals with concurrency control, the execute phase deals with distributed query execution, and the write phase deals with the execution of updates at all replicas of the modified data.
- R* is a distributed database system developed by and operational at the IBM San Jose Research Laboratory at California. The objective of R* is to develop a cooperative distributed database system where each site is a relational database system with site autonomy.
- The general architecture of R* distributed database system consists of three major components. These are a local database management system, a data communication system, and a transaction manager. The local DBMS component is further subdivided into two components, known as R* storage system (RSS*) and database language processor.
- There are two steps involved in the processing of a query in R* environment, known as compilation and execution. Sometimes, recompilation is also required.
- In R* system, two-phase commit protocol is used with variations for transaction commitment. These are presumed abort protocol and presumed commit protocol.

EXERCISES

Multiple Choice Questions

(i) SDD-1 distributed database system supports
   a. Object-oriented data model
   b. Relational data model
   c. None of these
   d. All of these.

(ii) In SDD-1, guaranteed delivery is provided by
   a. Transaction module
   b. Data module
   c. Reliable network
   d. All of these.

(iii) SDD-1 supports
   a. Fragmentation but not replication
   b. Replication but not fragmentation
   c. Both fragmentation and replication
   d. Neither fragmentation nor replication.

(iv) In SDD-1, read phase of a transaction deals with
   a. Concurrency control
   b. Deadlock handling
   c. Query processing
   d. All of these.

(v) In SDD-1, execute phase of a transaction handles
  a. Distributed recovery
  b. Security and integrity
  c. Concurrency control
  d. Query processing.

(vi) Which of the following mechanisms is used to ensure distributed serializability in SDD-1?
  a. Timestamp-based protocols
  b. Conflict graph analysis
  c. All of these
  d. None of these.

(vii) In SDD-1, reduction and final processing are two phases of
  a. Query execution
  b. Access planning
  c. Guaranteed delivery
  d. Transaction control.

(viii) Inter-system communication facility of CICS is used in
  a. SDD-1 database system
  b. INGRES system
  c. R* system
  d. All of these.

(ix) Which of the following techniques is used for concurrency control in R* system?
  a. Two-phase commit protocol
  b. Timestamp-based protocol
  c. Two-phase locking protocol
  d. Hybrid protocol.

(x) Presumed abort protocol is beneficial for
  a. Partially read-only transactions
  b. Completely read-only transactions
  c. Update transactions
  d. All of the above.

(xi) Which of the following deadlock handling mechanisms is used in R*?
  a. Deadlock avoidance
  b. Deadlock prevention
  c. Deadlock detection and recovery from deadlock
  d. None of these.

(xii) In R* distributed database system, query recompilation is required if
  a. The given SQL statement involves more than one site
  b. The given SQL statement involves only the local site
  c. The definitions of data in the given SQL statement change
  d. None of the above.

(xiii) Tree of processes is created in
  a. INGRES system
  b. SDD-1 system
  c. R* system
  d. All of these.

(xiv) Which of the following statements is correct?
  a. In R*, when log is written synchronously, it is called "write"
  b. In R*, when log is written asynchronously, it is called "force-write"
  c. In R*, when log is written synchronously, it is called "force-write"
  d. None of these.

Review Questions

1. Describe the architecture of SDD-1 distributed database system.
2. Write down the functions of transaction module in SDD-1 database system.
3. What are the different services provided by RelNet in SDD-1 distributed system?
4. Explain how read phase of transactions deals with concurrency control in SDD-1.
5. Discuss the query processing technique in SDD-1 database system.
6. Explain how write phase of transactions handles distributed reliability in SDD-1.
7. Write down the architecture of R* database system.
8. Describe the steps of query compilation at the master site in R* database system.
9. Describe the presumed abort protocol and explain how it is beneficial for read-only transactions.
10. Discuss the presumed commit protocol.

15 Data Warehousing and Data Mining

This chapter focuses on the basic concepts of data warehousing, data mining and online analytical processing (OLAP). A data warehouse is a subject-oriented, time-variant, integrated, non-volatile repository of information for strategic decision-making. Data in warehouses may be multi-dimensional. OLAP is the dynamic synthesis, analysis and consolidation of large volumes of multi-dimensional data. Data mining is the process of extracting valid, previously unknown, comprehensible and actionable information from large volumes of multi-dimensional data, and it is used for making crucial business decisions. Therefore, all the above three concepts are related to each other. Benefits and problems of data warehouses and architecture of a data warehouse are briefly explained in this chapter. The concepts of data mart and data warehouse schemas, and different data mining techniques are also discussed.

The organization of this chapter is as follows. Section 15.1 introduces the concepts of data warehousing. The architecture and the different components of a data warehouse are described in Section 15.2. Database schemas for data warehouses are illustrated in Section 15.3, and the concept of a data mart is introduced in Section 15.4. OLAP technique and OLAP tools are discussed in Section 15.5. In Section 15.6, the basic concept of data mining is represented, and the different data mining techniques are focused on in Section 15.7.

15.1 Concepts of Data Warehousing

The advancement in computing technology has significantly influenced modern business scenario and scientific computing. Business is becoming inherently competitive, and in this competitive world people have to take appropriate strategic decisions very quickly for survival. Operational database systems have traditionally been designed to meet critical requirements of online transaction processing and batch processing. In contrast, information that are required for strategic decision-making are characterized by online and ad hoc query processing or batch intelligence-gathering functions for decision support. In the domain of scientific computing, the major challenge is to infer some valuable information from the observed data. In this context, data warehousing has emerged as a new paradigm that is specifically intended to provide strategic information to its users.

In the 1990s, organizations began moving from traditional database systems to data warehouse systems to achieve competitive advantage. A data warehouse integrates data from multiple, heterogeneous sources and transforms them into meaningful information, thereby allowing business managers to perform more substantive, accurate and consistent analysis. Data warehousing improves the productivity of corporate decision-makers through consolidation, conversion, transformation and integration of operational data, and provides a consistent view of an enterprise. Data warehousing is an environment, not a product. It is an architectural construct of information systems

that provides current and historical decision support information to users. In 1993, W.H. Inmon offered the following formal definition for the data warehouse.

A **data warehouse** is a **subject-oriented, integrated, time-variant, non-volatile** collection of data in support of management's decision-making process.

Subject-oriented – A data warehouse is organized around the major subjects of an enterprise such as customer, products, and sales. Data are organized according to subject instead of application. The data organized by subject obtains only the information necessary for the decision support processing.

Non-volatile – A data warehouse is always physically separated from enterprise-wide application systems. Owing to this separation, data warehouses do not require transaction processing, recovery management, concurrency control etc. The data in the data warehouse is not updated or changed in real time but is loaded and refreshed from operational systems on a regular basis, and accessed by queries. New data is always added as a supplement to the database, rather than as a replacement.

Time-variant – In a data warehouse, data are stored to provide a historical perspective. Every key structure in the data warehouse contains an element of time either implicitly or explicitly. The data in the data warehouse is only accurate and valid at some point in time or over some time interval. The data warehouse contains a place for sorting data that are 5 to 10 years old, or older, to be used for comparisons, trends and forecasting.

Integrated – A data warehouse is usually constructed by integrating multiple heterogeneous sources such as flat files, relational databases and online transaction processing (OLTP) files. The data sources are often inconsistent and use different data formats. When data are moved from an operational environment to a data warehouse, the integrated data must be consistent to provide a unified view of the data to users. Data cleaning and data transformation techniques are used to maintain consistency in naming conventions, measures of variables, encoding structures and physical attributes.

There are numerous definitions of data warehouse, but the earlier definition focuses on the characteristics of the data that are stored in the data warehouse. The data in the warehouse is used for analysis purpose, thus, it involves complex queries on data. In recent years, a new concept is associated with the data warehouse, known as data web house. A **data web house** is a distributed data warehouse that is implemented on the Web with no central data repository.

15.1.1 Benefits of Data warehousing

The successful implementation of data warehousing provides several tangible benefits as well as several intangible benefits, which are listed in the following:

(i) A huge amount of resources are required to ensure the successful implementation of a data warehouse, and the cost can vary enormously, because of the availability of a variety of technical solutions. However, a study by the International Data Corporation (IDC) in 1996 reported that the average three-year returns on investment in data warehousing reached 401 percent for more than 90 percent of the companies. Therefore, returns on investment in data warehousing is very high.

(ii) A substantial competitive advantage is gained from data warehousing, which allows decision-makers access to data that were previously unavailable, unknown or not captured. The huge returns on investment for those companies that have successfully implemented a data warehouse are evidence of the enormous competitive advantage.

(iii) Data warehousing improves the productivity of corporate decision-makers by creating an integrated database of consistent, subject-oriented, historical data. A data warehouse helps business managers to perform more substantive, accurate and consistent analysis.

(iv) Data warehousing reduces the redundant processing and provides software to support overlapping decision support applications. It enables more cost-effective decision-making by separating query processing from running against operational databases.

(v) A data warehouse facilitates better business intelligence by increased quality and flexibility of market analysis made available through multi-level data structures, which may range from detailed to highly summarized.

(vi) It also facilitates business process re-engineering. Data warehousing can provide useful insights into the work processes themselves, which results in developing breakthrough ideas for the re-engineering of those processes.

15.1.2 Problems in Data Warehousing

Although data warehousing provides several benefits, there are some problems associated with the development and the maintenance of data warehouses. These are listed in the following.

- **Complexity of data integration** – The major challenge in developing and maintaining a data warehouse is ensuring the integration capability, as data are collected from multiple heterogeneous sources. An organization spends a significant amount of time in developing an efficient tool for integrating these heterogeneous resources. Moreover, there are a number of tools for every operation in the data warehouse; thus, it is a very complicated task to integrate them well for the organization's benefit.
- **Hidden problems with data sources** – Some hidden problems in data sources may be identified a long period (may be one or more years) after loading the data into the data warehouse. In such cases, it is very difficult to decide whether the problem should be fixed in the data warehouse or in the data sources.
- **Required data not captured** – It may happen that some required data for a data warehouse were not captured by the existing data sources. In this situation, it is very difficult to determine whether to modify the online transaction processing systems or to create a system dedicated to capturing missing data.
- **Underestimation of time for data loading** – Developers may underestimate the time required to extract, clean and load the data into the warehouse, which may take a significant proportion of the total development time.
- **Increased storage space** – Typically, the data warehouse stores huge amounts of data, which requires large amount of storage space. Many relational databases that are used for decision support may require the creation of very large fact tables. If there are many dimensions to the factual data, the combination of aggregate tables and indexes to the fact tables can use more storage space than the raw data.
- **High maintenance** – Any re-arrangement of the business processes or the data sources may affect the data warehouse, therefore, high maintenance is required to maintain the consistency of the data in the data warehouse.

» **Long-duration projects** – A data warehouse represents a single data resource for the organization, but the building of a data warehouse generally requires a long period. This is the reason why some organizations build data marts instead of data warehouse. A data mart supports the requirements of a particular department or functional area, and therefore it can be developed more quickly.

» **Data homogenization** – Large-scale data warehousing can become an exercise in data homogenization, which reduces the value of the data. To produce a consolidated and integrated view of the organization's data, the designers emphasize on searching the similarities of data used by different application areas rather than the dissimilarities.

15.1.3 Data Warehouses and OLTP Systems

OLTP systems are not suitable for data warehouse, because the warehouse data is mainly used for analysis purpose that helps business developers in important strategic decision-making. Therefore, data warehouses are generally called **on-line analytical processing (OLAP)** systems. In contrast to an OLTP system, warehouse data do not require transaction management, concurrency control, recovery management, etc. In data warehouse, most of the operations are complex queries. Moreover, a data warehouse contains historical, detailed and summarized data at various levels and rarely subject to change. The comparison between a data warehouse and an OLTP system is presented in figure 15.1.

OLTP Systems	Data Warehouses
Holds current data and detailed data	Holds detailed data, historical data and summarized data
Data is dynamic	Data is largely static
Processing is repetitive	Processing is ad hoc, unstructured and heuristic
Transaction throughput is very high	Transaction throughput is low to medium.
Usage pattern is predictable	Usage pattern is unpredictable
Data is application-oriented	Data is subject-oriented
Supports day-to-day decisions	Supports strategic decisions
Serves large number of operational users	Serves relatively low number of managerial users

Fig. 15.1 Comparison between Data Warehouses and OLTP Systems

15.2 Data Warehousing Architecture

This section introduces the popular three-tier architecture and the components of a data warehouse at different layers. Figure 15.2 illustrates the typical architecture of a data warehouse, where tier 1 is essentially the warehouse server or warehouse manager, tier 2 is the OLAP engine for analytical processing and tier 3 is the client containing several tools such as reporting tools, visualization

tools, data mining tools and querying tools. Moreover, in the data warehouse architecture, operational data and processing are completely separated from data warehouse processing. Typically, the source data for the warehouse is coming from operational applications or the operational data store (ODS), and these are transformed into an integrated structure and format. The transformation process involves several steps such as extraction, cleansing, conversion and summarizing. The components of a data warehouse are described in the following.

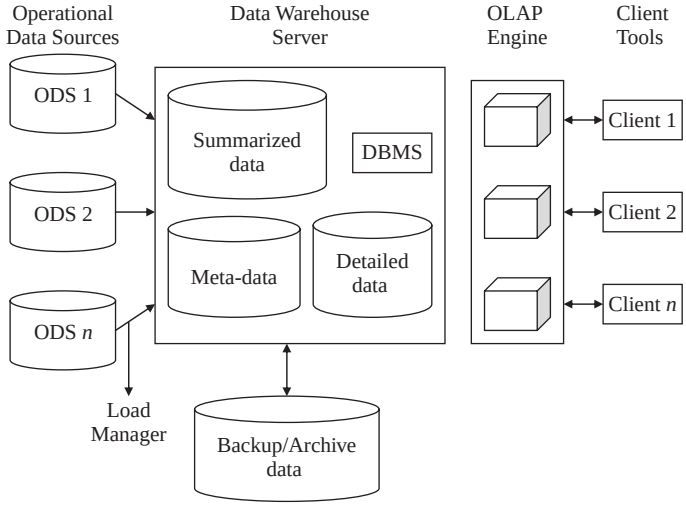


Fig. 15.2 Typical Architecture of a Data Warehouse

15.2.1 Operational Data Source

In a data warehouse, data comes from different sources as listed below.

- Mainframe operational data stored in hierarchical and network databases.
- Data stored in proprietary file systems and relational database management systems.
- Private data held on workstations and servers.
- External systems such as in Internet or commercial databases.
- Operational data stores.

An ODS is a repository of current and integrated operational data that are used for analysis. The data that are coming from an ODS are often structured and already extracted from data sources and cleaned; therefore, the work of the data warehouse is simplified. The ODS is often created when legacy operational systems are found to be incapable of achieving reporting requirements.

15.2.2 Load Manager

The load manager (also known as front-end component) performs all the operations associated with the extraction and loading of data from different data sources into the data warehouse. The data may be extracted from data sources as well as ODSs. The load manager is constructed by

combining vendor data loading tools and custom-built programs. The size and the complexity of this component may vary for different data warehouses.

15.2.3 Query Manager

The query manager (also known as back-end component) is responsible for managing user queries. This component performs various operations such as directing queries to the appropriate tables, scheduling the execution of queries, and generating query profiles to allow the warehouse manager to determine which indexes and aggregations are appropriate. The query profile is generated based on information that describes the characteristics of the query such as frequency, target tables and size of the result sets. The query manager is constructed by combining vendor end-user access tools, data warehouse monitoring tools, database facilities and custom-built programs. The complexity of this component depends on the functionalities provided by the end-user tools and the database.

15.2.4 Warehouse Manager

The warehouse manager is responsible for the overall management of the data in the data warehouse. This component is constructed by using data management tools and custom-build programs. The operations that are performed by this component are as follows:

- To maintain the consistency of data in the warehouse, the warehouse manager performs the analysis of data.
- The warehouse manager transforms and merges source data from temporary storage into data warehouse tables.
- It creates indexes and views on data warehouse tables.
- If necessary, generation of aggregations and de-normalizations are done by the warehouse manager.
- The warehouse manager is also responsible for data backup and archiving of data.

In some cases, the query manager is also responsible for generating query profiles.

15.2.5 Detailed Data

The data warehouse stores all detailed data in the database schema. In most cases, the detailed data is not stored online, but is made available by aggregating the data to the next detailed level. On a regular basis, detailed data is added to the warehouse to supplement the aggregated data.

15.2.6 Summarized Data

The data warehouse stores all summarized data generated by the warehouse manager. Summarized data may be lightly summarized or highly summarized. The objective of summarized data is to speed up query processing. Although summarizing of data involves increased operational costs initially, this is compensated, as it removes the requirement of repeatedly performing summary operations (such as sorting, grouping and aggregating) while answering user queries. The summary data is updated continuously as new data are loaded into the data warehouse.

15.2.7 Archive/Backup Data

The data warehouse also stores the detailed data and the summarized data for archiving and backup. Although the summarized data can be generated from the detailed data, it is necessary to backup the summarized data online. The data is transferred to storage archives such as magnetic tapes or optical disks.

15.2.8 Metadata

Data about data are called metadata. The data warehouse stores the metadata information used by all processes in the warehouse. A metadata repository should contain the following information.

- A description of the structure of the data warehouse that includes the warehouse schema, views, dimensions, hierarchies and derived data definitions, data marts locations and contents etc.
- Operational metadata that involves data linkage, currency of data, warehouse usage statistics, and error reports and their trails.
- The summarization processes that involve dimension definitions and data on granularity, partitions, summary measures, aggregation, and summarization.
- Details of data sources that include source databases and their contents, gateway descriptions, data partitions, rules for data extractions, cleaning and transformation, and defaults.
- Data related to system performance such as indexes, query profiles and rules for timing and scheduling of refresh, update and replication cycles.
- Business metadata that includes business terms and definitions, data ownership information and changing policies.

Metadata are used for a variety of purposes, which are listed below.

- In data extraction and loading processes, to map data sources to a common integrated format of the data within the data warehouse.
- In warehouse management process, to automate the generation of summary tables.
- In query management process, to direct a query to the most appropriate data sources.
- In end-user access tools, to understand how to build a query.

The structure of metadata differs between processes, because the purpose is different. Based on the variety of metadata, they are classified into three different categories: **build-time metadata, usage metadata** and **control metadata**. The metadata generated at the time of building a data warehouse is termed as build-time metadata. The usage metadata is derived from the build-time metadata when the warehouse is in production and these metadata are used by user queries and for data administration. The control metadata is used by the databases and other tools to manage their own operations.

15.2.9 End-User Access Tools

The main objective of a data warehouse is to provide information to business users for strategic decision-making. Business users can interact with the warehouse through end-user tools. A data

warehouse must efficiently support ad hoc queries and routine analyses. The end-user tools can be classified into five categories, namely, reporting and query tools, application development tools, executive information system (EIS) tools, OLAP tools and data mining tools.

(i) Reporting and query tools – Reporting tools include production reporting tools and report writers. Production reporting tools are used to generate regular operational reports such as employee pay cheques, customer orders and invoices. Report writers are inexpensive desktop tools used by end-users to generate different types of reports. For relational databases, query tools are also designed to accept SQL or generate SQL statements to query the data stored in the warehouse.

(ii) Application development tools – In addition to reporting and query tools, user access may require the development of in-house applications using graphical data access tools designed primarily for client/server environments. Some of these applications tools are also integrated with OLAP tools to access all major database systems.

(iii) EIS tools – These tools were originally developed to support high-level strategic decision-making. EIS tools are associated with mainframes and facilitate users for building customized graphical decision support applications that provide an overview of the organization's data and access to external data sources.

(iv) OLAP tools – These are very important tools based on multi-dimensional data models that assists a sophisticated user by facilitating the analysis of the data using complex, multi-dimensional views. Typical business applications that use these tools include assessing the effectiveness of a marketing campaign, product sales forecasting and capacity planning.

(v) Data mining tools – Data mining tools are used to discover new meaningful correlations, patterns and trends from large amounts of data. Generally, data mining tools use statistical, mathematical and artificial intelligence techniques for mining knowledge from large volumes of data.

15.2.10 Data Warehouse Background Processes

To get populated and to refresh the data, data warehouses use some backend tools and utilities. These tools and utilities provide several facilities such as data extraction, data cleaning, data transformation, loading and refresh. All these are described in the following.

- **Data extraction** – Data extraction is the process of extracting data for the data warehouse from various data sources such as production data, legacy data, metadata and data from external systems.
- **Data cleaning** – To ensure validity and relevance of data in the warehouse, data cleaning is essential while building a data warehouse. The data cleaning process is very complicated, and it involves several techniques such as transformation rules, domain-specific knowledge, parsing and fuzzy matching, and auditing. The transformation rules are used to translate attributes (e.g., age to date of birth). The parsing and fuzzy matching technique is used to assign a preferred source as a matching standard among multiple data sources. The auditing technique is used to discover facts that flag unusual patterns.
- **Data transformation** – In a data warehouse, data generally come from multiple heterogeneous sources. Data transformation is the process of transforming different heterogeneous data into a uniform structure so that the data can be combined and integrated.

- **Loading** – A huge amount of data is loaded into the data warehouse, as it integrates time-varying data from multiple sources. Further, the time interval when the data warehouse can be taken offline for loading data is usually insufficient. Also, while loading data, indexes and summary tables are required to be rebuilt. The loading process allows system administrators to monitor the status, cancel, suspend, resume loading or change the loading rate, and restart loading after failures without any loss of data integrity. There are different data loading strategies such as batch loading, sequential loading and incremental loading.
- **Refresh** – It is required to update the warehouse when the source data is updated. This process is known as refreshing. It is very difficult to determine how frequently the refreshing of the warehouse should be performed. One solution is to refresh on every update, but it is very expensive. Another solution is to perform refreshing periodically. Refresh policies can be set by the data administrator depending on user needs and data traffic.

15.3 Data Warehouse Schema

The designing of a database for a data warehouse is highly complicated. Data warehouses typically have schemas that are designed for data analysis using tools such as OLAP tools. Thus, the data in a warehouse are usually multi-dimensional with dimension attributes and measure attributes. The database component of a data warehouse is represented by using a technique called **dimensionality modeling**. The concept of entity–relationship modeling with some restrictions is used in dimensionality modeling. In this modeling, every dimensional model is composed of one **fact table** with a composite primary key, and a set of **dimension tables**. Tables containing multi-dimensional data are called fact tables and attributes of fact tables are called **dimension attributes**. Each dimension attribute is represented by a table, known as a dimension table. Each dimension table has a primary key that corresponds to exactly one component of the composite primary key of the fact table. In other words, the primary key of the fact table is made up two or more foreign keys. This star-like structure is called **star schema**. There are several variations of star schema that are used in warehouse database design. The three different types of data warehouse schemas are star schema, snowflake schema and fact constellation schema, which are described in the following.

15.3.1 Star Schema

Star Schema is represented by a starlike structure that has a fact table containing multi-dimensional data in the centre surrounded by dimension tables containing reference data that can be de-normalized. Since the bulk of data in a data warehouse is represented as facts, fact tables are usually very large. The dimension tables are relatively smaller in size. The fact table contains the details of summary data where each dimension is a single, highly de-normalized data. In star schema, each tuple in the fact table corresponds to one and only one tuple in each dimension table, but one tuple in a dimension table may correspond to more than one tuple in the fact table. Therefore, there is an N:1 relationship between the fact table and the dimension table. The structure of a star schema is illustrated in example 15.1.

The advantage of a star schema is that it is easy to understand, and hierarchies can be defined easily. Moreover, it reduces the number of physical joins and requires low maintenance.

Example 15.1

A table recording project completion information for an IT company is a typical example of a fact table, where each tuple represents a project that is successfully completed. The dimensions of the project table includes the details of each project, completion date of each project, the branch details where the project was executed, client details for each project, and the project leader details for each project. The resultant star schema is depicted in figure 15.3.

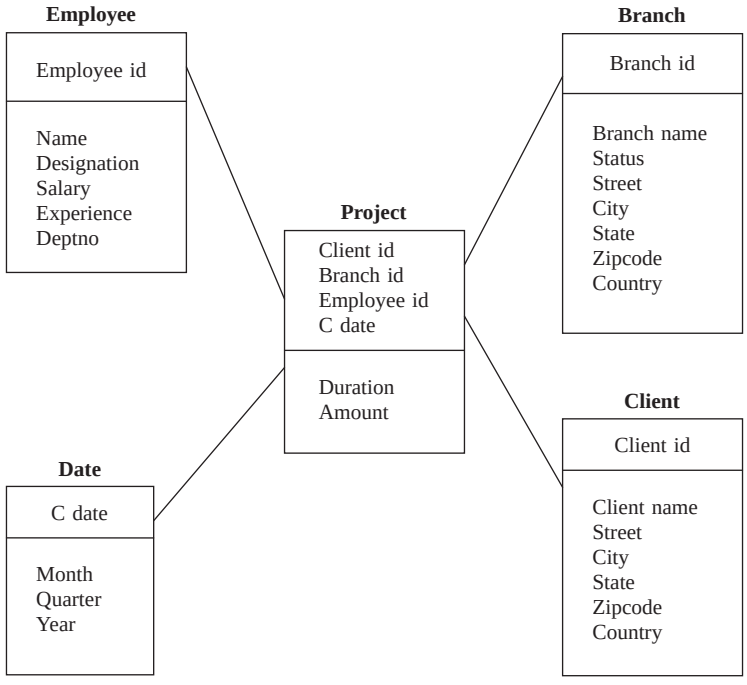


Fig. 15.3 An Example of Star Schema

15.3.2 Snowflake Schema

A variation of the star schema where the dimension tables do not contain de-normalized data is called snowflake schema. In snowflake schema, as the dimension tables contain normalized data, it allows dimension tables to have dimensions. To support attribute hierarchies, the dimension tables can be normalized to create a snowflake schema.

In a snowflake schema, it is easier to maintain normalized dimension tables, and owing to normalization, storage space is also saved. However, the snowflake schema may reduce the effectiveness of navigating across the tables owing to the large number of join operations.

Example 15.2

In example 15.1, a new dimension table Department of Employee has been created to convert it into a snowflake schema. The resultant snowflake schema is illustrated in figure 15.4.

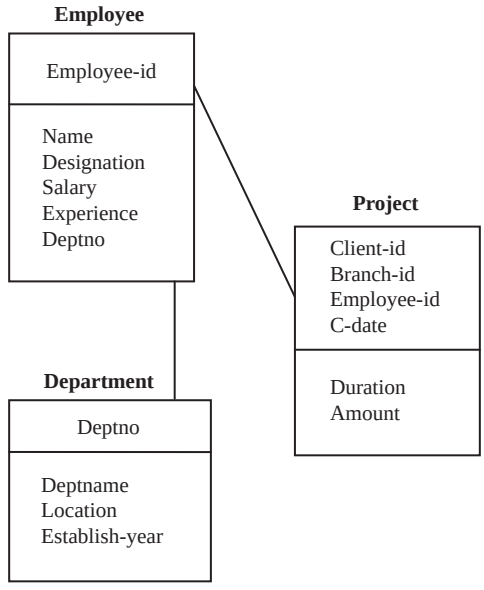


Fig. 15.4 An Example of Snowflake Schema

15.3.3 Fact Constellation Schema

The fact constellation schema is a hybrid structure, which is a combination of star and snowflake schemas. In fact constellation schema, more than one fact table can be shared among several dimension tables. It is also known as **galaxy schema**.

Example 15.3

In example 15.1, assume that the IT Company is also a manufacturer of some products. In this case, the product information and the project information can be represented by using a fact constellation schema. The corresponding fact constellation schema is illustrated in figure 15.5.

15.4 Data Marts

A subset of a data warehouse that supports the requirements of a particular department or business function is called a data mart. The data mart can be standalone or linked centrally to the corporate data warehouse. Data marts may contain some overlapping data. The characteristics that differentiate a data mart from a data warehouse include the following.

- A data mart focuses on only the requirements of users associated with one department or business function.
- Generally, a data mart does not contain detailed operational data.
- A data mart contains less data compared to a data warehouse; thus, it is easier to understand and navigate.

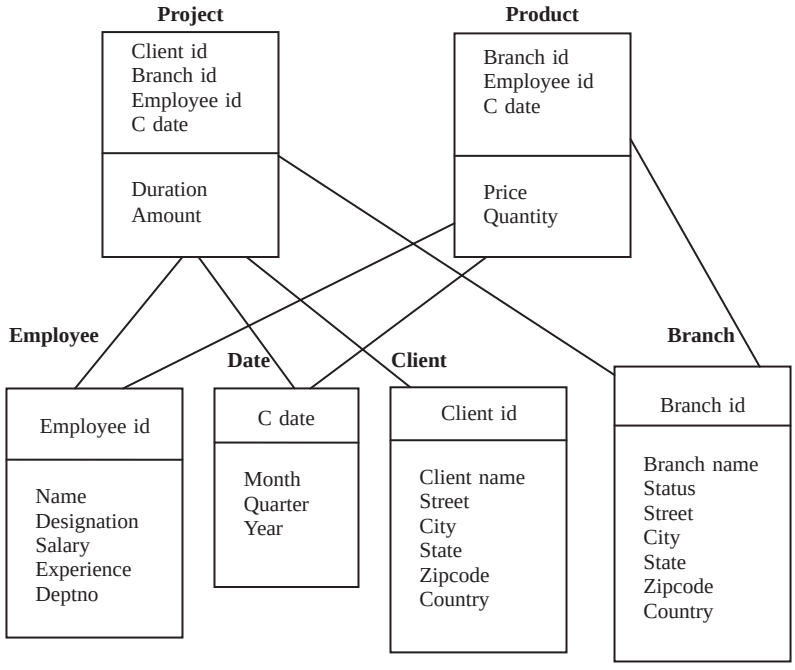


Fig. 15.5 An Example of Fact Constellation Schema

There are two different approaches for building a data mart. In the first approach, several data marts are created, and these are eventually integrated into a data warehouse. In the second approach, to build a corporate data warehouse, one or more data marts are created to fulfill the immediate business requirements. The architecture of a data mart may be two-tier or three-tier depending on database applications. In the three-tier architecture of data marts, tier 1 represents the data warehouse, tier 2 represents the data mart and tier 3 represents the end-user tools.

A data mart is often preferable over a data warehouse. The reasons are listed in the following.

» A data mart satisfies the requirements of a group of users in a particular department or business function; therefore, it is easier to understand and analyse.

» A data mart improves the response time of end-user queries, as it stores less volume of data than a data warehouse.

» The cost of implementing a data mart is much lesser than that of a data warehouse, and the implementation is less time-consuming.

» A data mart normally stores less volume of data; thus, data cleaning, loading, transformation and integration are much easier in this case. The implementation and establishment of a data mart is less complicated than that of a data warehouse.

» The potential users of a data mart can be defined easily, and they can be targeted easily to obtain support for a data mart project rather than for a corporate data warehouse project.

15.5 Online Analytical Processing

OLAP represents a technology that uses a multi-dimensional view of aggregated data to provide quick access to strategic information for performing advanced analysis [Codd et al., 1995]. OLAP facilitates users to gain a deeper knowledge and understanding about various aspects of corporate data through fast, consistent, interactive access to a wide variety of possible views of the data. In 1995, E.F. Codd had introduced 12 rules as the basis for selecting OLAP tools. These are listed in the following.

(i) **Multi-dimensional conceptual view** – OLAP tools should provide users with a multi-dimensional model that corresponds to the business problems, and is intuitively analytical and easy to use.

(ii) **Transparency** – The OLAP system's technology, the underlying database and computing architecture (client/server, mainframe gateways, etc.), and the heterogeneity of input data sources should be transparent to the users to preserve their productivity and proficiency with familiar front-end environments and tools.

(iii) **Accessibility** – The OLAP tools must be able to access the data required for the analysis from all heterogeneous enterprise data sources such as relational, object-oriented, and legacy systems.

(iv) **Consistent reporting performance** – As the number of dimensions, levels of aggregations, and the size of the database increases, users should not perceive any significant degradation in performance.

(v) **Client/Server architecture** – The OLAP system must be capable of operating efficiently in a client/server environment to provide optimal performance, flexibility, adaptability, scalability and interoperability.

(vi) **Generic dimensionality** – Every data dimension must be equivalent in both structure and operational capabilities.

(vii) **Dynamic sparse matrix handling** – The OLAP system must be able to adapt its physical schema to the specific analytical model that optimizes sparse matrix handling to achieve and maintain the required level of performance.

(viii) **Multi-user support** – The OLAP system must be able to support a group of users working concurrently on the same or different models of the enterprise's data.

(ix) **Unrestricted cross-dimensional operations** – The OLAP system must be able to recognize dimensional hierarchies and automatically perform associated roll-up calculations within and across dimensions.

(x) **Intuitive data manipulation** – Slicing and dicing (pivoting), drill-down and consolidation, and other manipulations should be accomplished via direct "point-and-click" and "drag-and-drop" actions on the cells of the cube.

(xi) **Flexible reporting** – The ability to arrange rows, columns and cells in a fashion that facilitates analysis by intuitive visual presentation of analytical reports must exist.

(xii) **Unlimited dimensions and aggregation levels** – Depending on business requirements, an analytical model may have numerous dimensions, each having multiple hierarchies. The

OLAP system should not impose any artificial restrictions on the number of dimensions or aggregation levels.

In addition to these 12 rules, a robust production-quality OLAP system should also support **comprehensive database management tools, the ability to drill down to detail level, incremental database refresh** and **SQL interface**.

15.5.1 OLAP Tools

OLAP tools are based on the concepts of multi-dimensional databases and allow a sophisticated user to analyse the data using elaborate, multi-dimensional, complex views. There are three main categories of OLAP tools, known as **multi-dimensional OLAP** (**MOLAP** or **MD-OLAP**), **relational OLAP** (**ROLAP**) and **hybrid OLAP** (**HOLAP**) [Berson and Smith, 2004]. These are described in the following.

Multi-dimensional OLAP– MOLAP tools utilize specialized data structures and multi-dimensional database management systems to organize, navigate and analyse data. Data structures in MOLAP tools use array technology and provide improved storage techniques to minimize the disk space requirements through sparse data management in most cases. The advantage of using a data cube is that it allows fast indexing to pre-compute summarized data. This architecture offers excellent performance when the data is used as designed, and the focus is on data for a specific decision support application. In addition, some OLAP tools treat time as a special dimension for enhancing their ability to perform time series analysis. The architecture of MOLAP tools is depicted in figure 15.6.

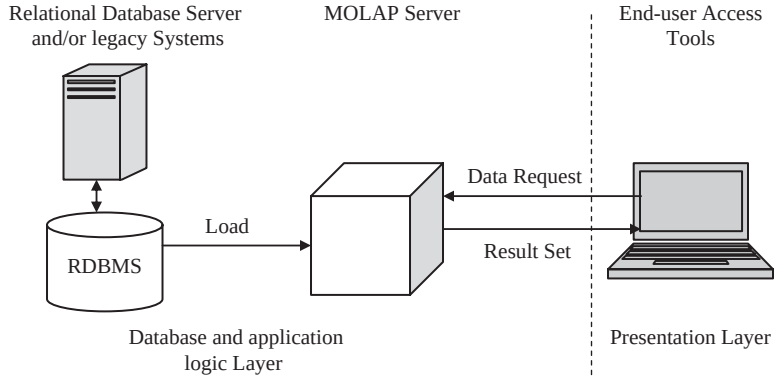


Fig. 15.6 MOLAP Architecture

Applications that require iterative and comprehensive time series analysis of trends are well suited for MOLAP technology. The underlying data structures in MOLAP tools are limited in their ability to support multiple subject areas and to provide access to detailed data. In this case, navigation and analysis of data are also limited, because the data is designed according to previously determined requirements. To support new requirements, data may need physical reorganization. Moreover, MOLAP tools require a different set of skills and tools to build and maintain the database,

thereby increasing the cost and complexity of support. Examples of MOLAP tools include Arbor Software's Essbase, Oracle's Express Server, Pilot Software's Lightship Server and Sinper's TM/1.

Relational OLAP – ROLAP, also called multi-relational OLAP, is the fastest growing style of OLAP technology. These tools support RDBMS products directly through a dictionary layer of metadata, thereby avoiding the requirement to create a static multi-dimensional data structure. ROLAP tools facilitate the creation of multiple multi-dimensional views of the two-dimensional relations. To improve performance, some ROLAP tools have developed enhanced SQL engines to support the complexity of multi-dimensional analysis. To provide flexibility, some ROLAP tools recommend or require the use of highly de-normalized database designs such as the star schema. The architecture of ROLAP tools is illustrated in figure 15.7.

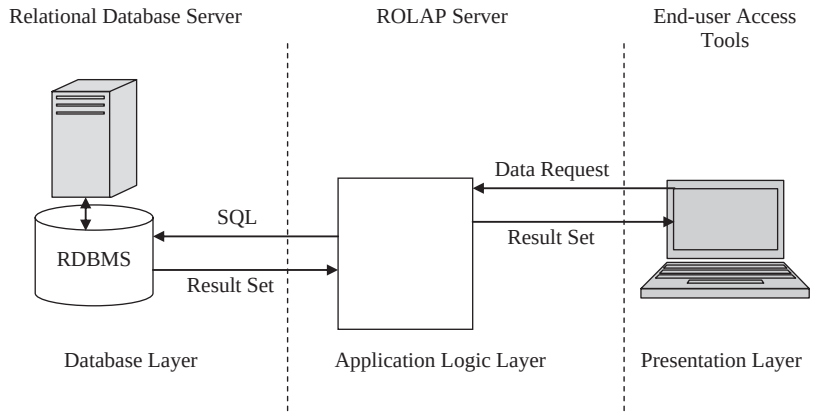


Fig. 15.7 ROLAP Architecture

ROLAP tools provide the benefit of full analytical functionality while maintaining the advantages of relational data. These tools depend on a specialized schema design, and their technology is limited by their non-integrated, disparate tier architecture. As the data is physically separated from the analytical processing, sometimes the scope of analysis is limited. In ROLAP tools, any change in the dimensional structure requires a physical reorganization of the database, which is very time-consuming. Examples of ROLAP tools are Information Advantage (Axsys), MicroStrategy (DSS Agent/DSS Server), Platinum/Prodea Software (Beacon) and Sybase (HighGate Project).

Hybrid OLAP – HOLAP tools, also called Managed Query Environment (MQE) tools, are relatively a new development. These tools provide limited analysis capability, either directly against RDBMS products, or by leveraging an intermediate MOLAP server. HOLAP tools deliver selected data directly from the DBMS or via a MOLAP server to the desktop in the form of a data cube where it is stored, analyzed and maintained locally. This reduces the overhead of creating the structure each time the query is executed. Once the data is in the data cube, users can perform multi-dimensional analysis against it. The architecture for HOLAP tools is depicted in figure 15.8.

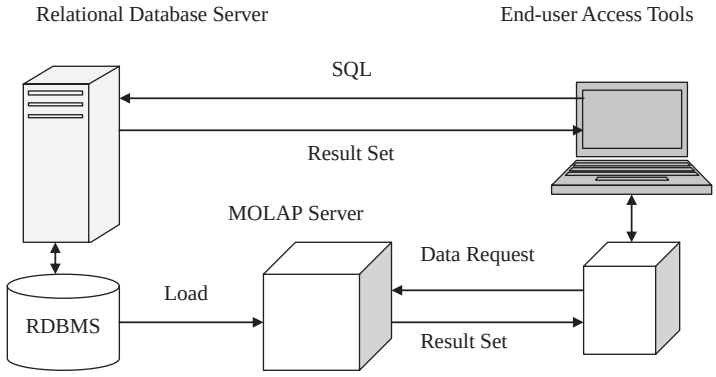


Fig. 15.8 HOLAP Architecture

The simplicity of installation and administration of HOLAP tools attracts vendors to promote this technology with significantly reduced cost and maintenance. However, this architecture results in significant data redundancy and may cause problems for networks that support many users. In this technology, only a limited amount of data can be maintained efficiently. Examples of HOLAP tools are Cognos Software's PowerPlay, Andyne Software's Pablo, Dimensional Insight's CrossTarget, and Speedware's Media.

15.6 Introduction to Data Mining

Owing to the widespread use of databases and the excessive growth in their volumes, organizations are facing the problem of information overloading. Simply storing information in a data warehouse does not provide the benefits that an organization is seeking. To realize the value of a data warehouse, it is necessary to extract the knowledge hidden within the warehouse. Data mining is the non-trivial process of identifying valid, novel, potentially useful and ultimately understandable pattern in data. Data mining is concerned with the analysis of data and the use of software techniques for searching hidden and unexpected patterns and relationships in sets of data. There is increasing desire to use this new technology in the application domain, and a growing perception is that these large passive databases can be made into useful actionable information. There are numerous definitions for data mining. One such focused definition is presented in the following.

Data mining is the process of extracting valid, previously unknown, comprehensible and actionable information from large databases and using it to make crucial business decisions.

Data mining is essentially a system that learns from existing data by scanning it and discovers patterns and correlations between attributes.

15.6.1 Knowledge Discovery in Database (KDD) Vs. Data Mining

In 1989, **KDD** was formalized for seeking knowledge from data with reference to the general concept of being broad and high-level. The term data mining was then coined, and this high-level technique is used to present and analyze data for business decision-makers. Actually, data mining

is a step involved in knowledge discovery process. There are various steps in knowledge discovery process, known as **selection, preprocessing, transformation, data mining**, and **interpretation and evaluation**. These steps are described below.

- In the selection step, the data that are relevant to some criteria are selected or segmented.
- Preprocessing is the data cleaning step where unnecessary information are removed.
- In the transformation step, data is converted to a form that is suitable for data mining. In this step, the data is made usable and navigable.
- Data mining step is concerned with the extraction of patterns from the data.
- In the interpretation and evaluation step, the patterns obtained in the data mining stage are converted into knowledge, which is used to support decision-making.

KDD and data mining can be distinguished by using the following definitions [Fayyad et al., 1996].

KDD is the process of identifying a valid, potentially useful and ultimately understandable structure in data. KDD involves several steps as mentioned above.

Data mining is a step in the KDD process concerned with the algorithmic means by which patterns or structures are enumerated from the data under acceptable computational efficiency limitations.

The structures that are the outcomes of the data mining process must meet certain criteria so that these can be considered as knowledge. These criteria are **validity, understandability, utility, novelty** and **interestingness**.

15.7 Data Mining Techniques

The fundamental objectives of data mining that are identified by researchers are **prediction** and **description**. To predict unknown or future values of interest, prediction makes use of existing variables in the database. Description focuses on finding patterns describing the data and the subsequent presentation for user interpretation. The relative emphasis of both prediction and description differ with respect to the underlying application and the technique used. To fulfill the above-mentioned objectives, several data mining techniques have been developed. These are predictive modeling, database segmentation or clustering, link analysis and deviation detection. All these data mining techniques are described in the following subsections.

15.7.1 Predictive Modeling

Predictive modeling is similar to the human learning experience in using observations to form a model of the important characteristics of some phenomenon. Predictive modeling is used to analyse an existing database to determine some essential features (model) about the data set. The model is developed using a **supervised learning** approach that has two different phases. These are **training** and **testing**. In the training phase, a model is developed using a large sample of historical data, known as the **training set**. In the testing phase, the model is tested on new, previously unseen, data to determine its accuracy and physical performance characteristics. Two methods are used in predictive modeling. These are **classification** and **value prediction**.

- **Classification** – Classification involves finding rules that partition the data into disjoint groups. The input for the classification is the training data set, whose class labels are already known. Classification analyses the training data set and constructs a model based on the class label, and assigns a class label to the future unlabelled records. There are two techniques for classification, known as **tree induction** and **neural induction**. In tree induction technique, the classification is achieved by using a decision tree, while in neural induction the classification is achieved by using a neural network. A neural network contains a collection of connected nodes with input, output and processing at each node. There may be several hidden processing layers between the visible input and output layers. The decision tree is the most popular technique for classification, and it presents the analysis in an intuitive way.
- **Value prediction** – Value prediction is used to estimate a continuous numeric value that is associated with a database record. There are two different techniques for value prediction that use the traditional statistical methods, known as **linear regression** and **non-linear regression**. Both these techniques are well-known methods and are easy to understand. In linear regression, an attempted is made to fit a straight line by plotting the data such that the line is the best representation of the average of all observations at that point in the plot. The limitation of linear regression is that the technique only works well with linear data and is sensitive to the presence of outliers. The non-linear regression also is not flexible enough to handle all possible shapes of the data plot. Applications of value prediction are credit card fraud detection, target mailing list identification etc.

15.7.2 Clustering

Clustering is a method of grouping data into an unknown number of different groups, known as **clusters or segments** so that the data in a group represents similar trends and patterns. This approach uses unsupervised learning to discover homogeneous subpopulations in a database to improve the accuracy of the profiles. Clustering constitutes a major class of data mining algorithms. The algorithm attempts to automatically partition the data space into a set of clusters, either deterministically or probabilitywise. Clustering is less precise than other operations and is therefore less sensitive to redundant and irrelevant features.

Two different methods are used for clustering based on allowable input data, the methods used to calculate the distance between records, and the presentation of the resulting clusters for analysis. These are **demographic** and **neural clustering**.

15.7.3 Link Analysis

The objective of link analysis is to establish links, called **associations**, between the individual records, or sets of records in a database. There are numerous applications of data mining that fit into this framework, and one such popular application is market-basket analysis. Three different techniques are used for link analysis, known as **associations discovery, sequential pattern discovery** and **similar time sequence discovery**. Associations discovery technique finds items that indicate the presence of other items in the same event. These affinities between items are represented by association rules. In sequential pattern discovery, patterns are searched between events such that the

presence of one set of items is followed by another set of items in a database of events over a period of time. Similarly time sequence discovery is used to discover the links between two sets of data that are time-dependent and is based on the degree of similarity between the patterns that both time series demonstrate.

15.7.4 Deviation Detection

Deviation detection identifies outlying points in a particular data set and explains whether they are due to noise or other impurities being present in the data or due to trivial reasons. Applications of deviation detection involve forecasting, fraud detection, customer retention etc. Deviation detection is often a source of true discovery because it identifies outliers that express deviation from some previously known expectation or norm. Deviation detection can be performed by using **statistical** and **visualization** techniques or as a by-product of data mining.

CHAPTER SUMMARY

- A data warehouse is a subject-oriented, time-variant, integrated, non-volatile repository of information for strategic decision-making.
- A **data web house** is a distributed data warehouse that is implemented on the Web with no central data repository.
- The components of a data warehouse are operational data source, load manager, query manager, warehouse manager, detailed data, summarized data, archive/backup data, metadata, end-user access tools and data warehouse background processes.
- The three different types of database schemas that are used to represent data in warehouses are star schema, snowflake schema and fact constellation schema.
- A subset of a data warehouse that supports the requirements of a particular department or business function is called a data mart.
- OLAP is the dynamic synthesis, analysis and consolidation of large volumes of multi-dimensional data. There are three different categories of OLAP tools, which are MOLAP tools, ROLAP tools and HOLAP tools.
- Data mining is the process of extracting valid, previously unknown, comprehensible and actionable information from large volumes of multi-dimensional data and it is used to make crucial business decisions. The different data mining techniques are predictive modeling, database segmentation or clustering, link analysis and deviation detection.

EXERCISES

Multiple Choice Questions

(i) A data warehouse is a collection of
   a. Time-variant data
   b. Integrated data
   c. Non-volatile data
   d. All of these.

(ii) A data warehouse is said be time-variant collection of data, because
   a. It contains multi-dimensional data
   b. It contains historical data
   c. Its contents change with time
   d. It contains summarized data.

(iii) A data warehouse is said to be non-volatile, because
   a. The contents remain same after a system failure
   b. It disappears when the system is switched off
   c. It stores read-only data
   d. None of the above.

(iv) Which of the following techniques is required for a data warehouse?
   a. Concurrency control
   b. Recovery Management

c. Loading & refreshing
d. All of the above.

(v) A ROLAP tool is an OLAP tools that supports
a. Multi-dimensional data model
b. Relational data model
c. Both a. and b.
d. None of the above.

(vi) A star schema generally contains
a. Normalized data
b. Data that are not normalized
c. Both a. and b.
d. None of the above.

(vii) A fact constellation schema can have
a. Multiple fact tables
b. Multiple dimensional tables
c. Multiple fact tables and multiple dimensional tables
d. All of these.

(viii) Which of the following statements is true?
a. Data mining and KDD are the same
b. Data mining is a step in KDD
c. Data mining is not related with KDD.
d. All of the above statements are true.

(ix) Training and testing are two phases of
a. Supervised learning method
b. Unsupervised learning method
c. Both supervised and unsupervised learning methods
d. None of the above.

(x) Which of the following statements is false?
a. The elements in a cluster represent same trends and patterns
b. The elements in a cluster represent different trends and patterns
c. The elements in different clusters represent different trends and patterns
d. All of the above statements.

(xi) In a data warehouse, which of the following is a data loading strategy?
a. Batch loading
b. Sequential loading
c. Incremental loading
d. All of these
e. None of these.

(xii) Which of the following processes updates data in the data warehouse?
a. Loading
b. Refreshing
c. Cleaning
d. Transformation.

(xiii) Which of the following does not belong to Codd's 12 rules for OLAP?
a. Transparency
b. Accessibility
c. Client/Server architecture
d. All of these
e. None of these.

(xiv) Which of the following statements is correct?
a. OLAP tools support multiple users
b. OLAP tools support dynamic sparse matrix handling
c. OLAP tools support unlimited dimensions and aggregation levels
d. All of these
e. None of these.

(xv) Which of the following is a MOLAP tool?
a. Arbor Software's Essbase
b. Sybase (HighGate Project)
c. Cognos Software's PowerPlay
d. All of these.

(xvi) Decision tree technique is used in
a. Clustering
b. Classification
c. Link analysis
d. Deviation detection.

(xvii) Which of the following techniques is used for clustering?
a. Regression
b. Decision tree
c. Demographic
d. Association discovery.

(xviii) Which of the following is used in deviation detection?
a. Similar time sequence discovery
b. Demographic
c. Statistics and visualization
d. All of these.

(xix) Which of the following is incorrect?
a. Classification involves finding rules that partition the data into disjoint groups
b. Clustering is a method of grouping data into an unknown number of different groups
c. Predictive modeling is used to analyse an existing database to determine some essential features about the data set.
d. None of these
e. All of these.

(xx) Which of the following data mining techniques is associated with link analysis?
a. Value predication
b. Sequential pattern discovery
c. Neural clustering
d. Demographic.

Review Questions

1. Explain the features of data in a data warehouse.
2. Describe the architecture of a data warehouse.
3. Write down the problems in data warehousing.
4. How is a data warehouse different from a database? How are they similar?
5. Discuss the utility of different back ground processes in a data warehouse.
6. Describe the characteristics and main functions of the following components of a data warehouse.
   (i) Load manager
   (ii) Warehouse Manager
   (iii) Query manager
   (iv) Metadata
7. "Data marts are often preferable over a data warehouse". Justify.
8. Explain how a data mart is different from a data warehouse.
9. Compare different data warehouse schemas with suitable examples.
10. Differentiate between OLAP and OLTP.
11. Describe Codd's 12 rules for OLAP tools.
12. Compare the architecture, characteristics and issues associated with different categories of OLAP tools, namely, MOLAP, ROLAP and HOLAP tools.
13. Define data mining. Discuss the role of data mining in a data warehouse.
14. How is data mining different from knowledge discovery in databases (KDD).
15. What do you understand by deviation detection? Is it similar to association rule discovery? Give a justification.
16. Differentiate between classification and clustering.
17. Comment on the following:

    "Data mining systems are not necessarily built on data warehouses."

    "Background knowledge adds intelligence in data mining process."
18. Assume that a particular course is offered by several educational institutes, where each institute can have a number of departments. Each department can have a number of employees and multiple students can enroll for a particular course. Consider the major entities and design an appropriate data warehouse schema for course information.
19. Assume that a bank has several branches throughout the country and each branch maintains saving bank account information and loan account information separately. Customer services at each branch are provided by different employees of that branch. Consider the major entities involved with the banking system and design an appropriate data warehouse schema for customer information.

20. Consider a data warehouse that consists of three dimensions time, doctor and patient and two measures count and charge, where charge is the fee that a doctor charges a patient for a visit. Draw a fact constellation schema for the data warehouse.
21. Discuss the applications of data mining in the banking industry.
22. Discuss the applications of data mining in sales management.

Appendix

Solved Examples of B.Tech, M.Tech and M.C.A Examinations

1. Consider the following PROJECT relation. This relation is fragmented in two fragments namely, P1 and P2 as given below.

PNO	ENO	PTYPE	PDESC
P001	E001	DEVELOPMENT	Proj1
P002	E005	DEVELOPMENT	Proj2
P003	E0014	DEVELOPMENT	Proj3
P004	E002	MAINTENANCE	Proj4

P1: $\sigma_{\text{PTYPE = "DEVELOPMENT"}}$ (PROJECT), P2: $\sigma_{\text{PTYPE = "MAINTENANCE"}}$ (PROJECT)

Show the correctness of the Fragmentation.

[WBUT (MCA)-2008]

Answer:

The correctness of the fragmentation is ensured by three different rules, namely, completeness, reconstruction and disjointness. According to completeness rule, if a relation is fragmented, there should be no loss of information after fragmentation. Horizontal fragments are subsets of tuples of the original relation and hence the relation PROJECT is horizontally fragmented. After fragmentation, the relation PROJECT is partitioned into two different relations P1 and P2 as follows:

P1

PNO	ENO	PTYPE	PDESC
P001	E001	DEVELOPMENT	Proj1
P002	E005	DEVELOPMENT	Proj2
P003	E0014	DEVELOPMENT	Proj3

P2

PNO	ENO	PTYPE	PDESC
P004	E002	MAINTENANCE	Proj4

Since any tuple of the PROJECT relation is appeared either in the relation P1 or P2, so there is no loss of information and the completeness rule of fragmentation is ensured here.

According to reconstruction rule, it should be possible to reconstruct the original relation PROJECT from the fragmented relations P1 and P2 by using relational algebraic operations. Hence, PROJECT = P1 $\cup$ P2. In this case, there is no loss of functional dependencies and the reconstruction rule of fragmentation is ensured.

The disjointness rule of fragmentation ensures no repetition of information in fragmented relations. In this case, any tuple of the PROJECT relation is appeared either in the relation P1 or in the relation P2, but not in both. Hence, P1 $\cap$ P2 = Φ, and therefore disjointness rule is ensured.

The above fragmentation is correct since all the correctness rules of fragmentation are guaranteed.

2. Consider the following global, fragmentation and allocation schema:

Global Schema: STUDENT(STUD-ID, DEPT, NAME)

Fragmentation Schema: $STUDENT_1 = SL_{DEPT = "EE"}$ STUDENT
$STUDENT_2 = SL_{DEPT = "CS"}$ STUDENT

(Assume that EE and CS are only possible values for DEPT).

Allocation Schema: $STUDENT_1$ at sites 1, 2
$STUDENT_2$ at sites 3, 4.

(i) Write a query that accepts the STUD-ID from terminal and outputs the name and the department at levels 1, 2, and 3 of transparency.

(ii) Write a query that moves the student having number 232 from department "EE" to department "CS", at levels 1, 2, and 3 of transparency.

[WBUT (B.TECH) – 2003]

Answer:

(i)

Fragmentation Transparency (Level 1):

```
read (SStud-id);
Select DEPT, NAME into $DEPT, $NAME from STUDENT
where STUD-ID = SStu-id;
write($DEPT, $NAME); [in output]
```

Location Transparency (Level 2):

```
read (SStud-id);
Select DEPT, NAME into $DEPT, $NAME from STUDENT1
where STUD-ID = SStu-id;
if not FOUND then
Select DEPT, NAME into $DEPT, $NAME from STUDENT2
where STUD-ID = SStu-id;
write($DEPT, $NAME); [in output]
```

Local Mapping Transparency (Level 3):

```
read (SStud-id);
Select DEPT, NAME into $DEPT, $NAME from STUDENT1 AT SITE 1
where STUD-ID = SStu-id;
if not FOUND then
Select DEPT, NAME into $DEPT, $NAME from STUDENT2 AT SITE 3
where STUD-ID = SStu-id;
write($DEPT, $NAME); [in output]
```

(ii)

Fragmentation Transparency (Level 1):

```
Update STUDENT
set DEPT = "CS"
where STUD-ID = 232;
```

Location Transparency (Level 2):

```
Select NAME into $NAME from STUDENT1
where STUD-ID = 232;
insert into STUDENT2 (STUD-ID, DEPT, NAME)
values (232, 'CS', '$NAME');
Delete from STUDENT1 where STUD-ID = 232;
```

Local Mapping Transparency (Level 3):

Select NAME into $NAME from $STUDENT_1$ AT SITE 1
where STUD-ID = 232;
insert into $STUDENT_2$ AT SITE 3 (STUD-ID, DEPT, NAME)
values (232, 'CS', '$NAME');
insert into $STUDENT_2$ AT SITE 4 (STUD-ID, DEPT, NAME)
values (232, 'CS', '$NAME');
Delete from $STUDENT_1$ AT SITE 1 where STUD-ID = 232;
Delete from $STUDENT_1$ AT SITE 2 where STUD-ID = 232;

3. Consider the following relations:

EMP (EMPNUM, NAME, SAL, TAX, MGRNUM, DEPTNUM)

Fragments horizontal EMP1, EMP2, EMP3 based on DEPTNUM

DEPT (DEPTNUM, NAME, AREA, MGRNUM)

Fragments horizontal DEPT1, DEPT2, DEPT3 based on DEPTNUM.

SUPPLIER (SNUM, NAME, CITY)

SUPPLY (SNUM, PNUM, DEPTNUM, QUAN)

Fragments horizontal SUPPLIER1, SUPPLIER2 based on CITY. Fragments derived horizontal SUPPLY1, SUPPLY2 based on SNUM of corresponding fragments of SUPPLIER relation.

[WBUT B.TECH (CSE) – 2008]

Answer:

The horizontal fragments of EMP relation based on DEPTNUM are as follows.

$EMP1 = SL_{DEPTNUM = 10} (EMP)$

$EMP2 = SL_{DEPTNUM = 20} (EMP)$

$EMP3 = SL_{DEPTNUM = 30} (EMP)$

It has been assumed that there are three possible values for DEPTNUM attribute.
The horizontal fragments of DEPT relation based on DEPTNUM are as follows.

$DEPT1 = SL_{DEPTNUM = 10} (DEPT)$
$DEPT2 = SL_{DEPTNUM = 20} (DEPT)$
$DEPT3 = SL_{DEPTNUM = 30} (DEPT)$

The horizontal fragments of SUPPLIER relation based on CITY are:

$SUPPLIER1 = SL_{CITY = "KOLKATA"} (SUPPLIER)$
$SUPPLIER2 = SL_{CITY = "DELHI"} (SUPPLIER)$

It has been assumed that 'KOLKATA' and 'DELHI' are only two possible values for the CITY attribute in SUPPLIER relation. The derived horizontal fragments of SUPPLY relation based on SNUM of corresponding fragments of SUPPLIER relation are as follows.

$SUPPLY1 = SUPPLY\ SJ_{SNUM}\ SUPPLIER1$
$SUPPLY2 = SUPPLY\ SJ_{SNUM}\ SUPPLIER2.$

4. A country-wide drug supplier chain operates from five different cities in the country has the following tables in its database:

DrugShopMstr (DS-id, DS-city, DS-contact-no)
MedicineMstr (med-id, med-name, manu-id)
ManufacturerMstr (manu-id, manu-name, manu-city)
Order (med-id, DS-id, Qty)

Suggest a fragmentation and allocation schema keeping in mind the following frequent queries:

(a) List the manufacturer's names who belong to the same city in which the drug shop that has placed an order resides.

(b) How many orders are generated from a city, say "X"?

Justify your design and state your assumption clearly.

[C.U. M.Tech (CSE) – 2005]

Answer:

The global schema, fragmentation schema and allocation schema for the country-wide drug supplier chain management system is as follows:

Global Schema:

DrugShopMstr (DS-id, DS-city, DS-contact-no)

MedicineMstr (med-id, med-name, manu-id)

ManufacturerMstr (manu-id, manu-name, manu-city)

Order (med-id, DS-id, Qty)

Fragmentation Schema:

$\mathbf{DrugShopMstr_1 = SL_{DS\text{-}city = "X"}\ (DrugShopMstr)}$

$\mathbf{DrugShopMstr_2 = SL_{DS\text{-}city = "Y"}\ (DrugShopMstr)}$

$\mathbf{DrugShopMstr_3 = SL_{DS\text{-}city = "Z"}\ (DrugShopMstr)}$

$\mathbf{DrugShopMstr_4 = SL_{DS\text{-}city = "P"}\ (DrugShopMstr)}$

$\mathbf{DrugShopMstr_5 = SL_{DS\text{-}city = "Q"}\ (DrugShopMstr)}$

$\mathbf{Order_i = Order\ SJ_{DS\text{-}id}\ DrugShopMstr_i,\ where\ i = 1, 2, 3, 4, 5.}$

$\mathbf{MedicineMstr_i = MedicineMstr\ SJ_{mem\text{-}id}\ Order_i,\ where\ i = 1, 2, 3, 4, 5.}$

$\mathbf{ManufacturerMstr_i = ManufacturerMstr\ SJ_{manu\text{-}id}\ MedicineMstr_i,\ where\ i = 1, 2, 3, 4, 5.}$

Allocation Schema:

$DrugshopMstr_1$, $ManufacturerMstr_1$, $MedicineMstr_1$, and $Order_1$ at city X site (say site no. 1).

$DrugshopMstr_2$, $ManufacturerMstr_2$, $MedicineMstr_2$, and $Order_2$ at city Y site (say site no. 2).

$DrugshopMstr_3$, $ManufacturerMstr_3$, $MedicineMstr_3$, and $Order_3$ at city Z site (say site no. 3).

$DrugshopMstr_4$, $ManufacturerMstr_4$, $MedicineMstr_4$, and $Order_4$ at city P site (say site no. 4).

$DrugshopMstr_5$, $ManufacturerMstr_5$, $MedicineMstr_5$, and $Order_5$ at city Q site (say site no. 5).

In the above distributed database design, only five cities, that is, 'X', 'Y', 'Z', 'P', and 'Q' have been considered. Further, it has been assumed that the drug shops of a particular city will place the order for medicine to the manufacturer of the same city. Thus, the drug shops of city 'X' will place the order to the manufacturer of city X. Similarly, the drug shops at city "Y" will place order to the manufacturer of city 'Y', the drug shops at city 'Z' will place order to the manufacturer of city 'Z' and so on. Based on these assumptions, local data are stored locally here, which is beneficial for any distributed database system. Therefore, all information regarding the country-wide drug supplier chain management system such as drug shop details, manufacturer details, medicine details and order details have been stored at city X site (site no. 1) locally. Similarly, the detailed information for city 'Y' has been stored at city 'Y' site (site no. 2). In the above distributed database design, the following predicates are complete and minimal.

Application for listing the names of manufacturer who belong to same city in which the drug shop that has placed an order resides:

Fragmentation Transparency (Level 1):

read ($city);
Select DS-id into $DS-id from DrugShopMstr where DS-city = $city;
Select mem-id into $mem-id from Order where DS-id = $DS-id;
Select manu-id into $manu-id from MedicineMstr where mem-id = $mem-id;
Select manu-name into $manu-name from ManufacturerMstr where manu-id = $manu-id;
write ($manu-name);
[in output]

Location Transparency (Level 2):

read ($city);
Select DS-id into $DS-id from $DrugShopMstr_1$ where DS-city = $city;
Select mem-id into $mem-id from $Order_1$ where DS-id = $DS-id;
Select manu-id into $manu-id from $MedicineMstr_1$ where mem-id = $mem-id;
Select manu-name into $manu-name from $ManufacturerMstr_1$ where manu-id = $manu-id;

If not FOUND then
Select DS-id into $DS-id from $DrugShopMstr_2$ where DS-city = $city;
Select mem-id into $mem-id from $Order_2$ where DS-id = $DS-id;
Select manu-id into $manu-id from $MedicineMstr_2$ where mem-id = $mem-id;
Select manu-name into $manu-name from $ManufacturerMstr_2$ where manu-id = $manu-id;

If not FOUND then
Select DS-id into $DS-id from $DrugShopMstr_3$ where DS-city = $city;
Select mem-id into $mem-id from $Order_3$ where DS-id = $DS-id;
Select manu-id into $manu-id from $MedicineMstr_3$ where mem-id = $mem-id;
Select manu-name into $manu-name from $ManufacturerMstr_3$ where manu-id = $manu-id;

If not FOUND then
Select DS-id into $DS-id from $DrugShopMstr_4$ where DS-city = $city;
Select mem-id into $mem-id from $Order_4$ where DS-id = $DS-id;
Select manu-id into $manu-id from $MedicineMstr_4$ where mem-id = $mem-id;
Select manu-name into $manu-name from $ManufacturerMstr_4$ where manu-id = $manu-id;

If not FOUND then
Select DS-id into $DS-id from $DrugShopMstr_5$ where DS-city = $city;
Select mem-id into $mem-id from $Order_5$ where DS-id = $DS-id;
Select manu-id into $manu-id from $MedicineMstr_5$ where mem-id = $mem-id;
Select manu-name into $manu-name from $ManufacturerMstr_5$ where manu-id = $manu-id;

write ($manu-name);
[in output]

Local Mapping Transparency (Level 3):

read ($city);
Select DS-id into $DS-id from $DrugShopMstr_1$ AT SITE 1 where DS-city = $city;
Select mem-id into $mem-id from $Order_1$ AT SITE 1 where DS-id = $DS-id;
Select manu-id into $manu-id from $MedicineMstr_1$ AT SITE 1 where mem-id = $mem-id;
Select manu-name into $manu-name from $ManufacturerMstr_1$ AT SITE 1 where manu-id = $manu-id;

If not FOUND then
Select DS-id into $DS-id from $DrugShopMstr_2$ AT SITE 2 where DS-city = $city;
Select mem-id into $mem-id from $Order_2$ AT SITE 2 where DS-id = $DS-id;
Select manu-id into $manu-id from $MedicineMstr_2$ AT SITE 2 where mem-id = $mem-id;
Select manu-name into $manu-name from $ManufacturerMstr_2$ AT SITE 2 where manu-id = $manu-id;

If not FOUND then
Select DS-id into $DS-id from $DrugShopMstr_3$ AT SITE 3 where DS-city = $city;
Select mem-id into $mem-id from $Order_3$ AT SITE 3 where DS-id = $DS-id;
Select manu-id into $manu-id from $MedicineMstr_3$ AT SITE 3 where mem-id = $mem-id;
Select manu-name into $manu-name from $ManufacturerMstr_3$ AT SITE 3 where manu-id = $manu-id;

If not FOUND then
Select DS-id into $DS-id from $DrugShopMstr_4$ AT SITE 4 where DS-city = $city;
Select mem-id into $mem-id from $Order_4$ AT SITE 4 where DS-id = $DS-id;
Select manu-id into $manu-id from $MedicineMstr_4$ AT SITE 4 where mem-id = $mem-id;
Select manu-name into $manu-name from $ManufacturerMstr_4$ AT SITE 4 where manu-id = $manu-id;

If not FOUND then
Select DS-id into $DS-id from $DrugShopMstr_5$ AT SITE 5 where DS-city = $city;
Select mem-id into $mem-id from $Order_5$ AT SITE 5 where DS-id = $DS-id;
Select manu-id into $manu-id from $MedicineMstr_5$ AT SITE 5 where mem-id = $mem-id;
Select manu-name into $manu-name from $ManufacturerMstr_5$ AT SITE 5 where manu-id = $manu-id;

write ($manu-name);
[in output]

Application for finding how many orders are generated from a city, say "X":

Fragmentation Transparency (Level 1):

Select DS-id into $DS-id from DrugShopMstr where DS-city = 'X';
Select sum (Qty) into $sum from Order where DS-id = $DS-id;
write ($sum);
[in output]

Location Transparency (Level 2):

Select DS-id into $DS-id from $DrugShopMstr_1$ where DS-city = 'X';
Select sum (Qty) into $sum from $Order_1$ where DS-id = $DS-id;

If not FOUND then
Select DS-id into $DS-id from $DrugShopMstr_2$ where DS-city = 'X';
Select sum (Qty) into $sum from $Order_2$ where DS-id = $DS-id;

If not FOUND then
Select DS-id into $DS-id from $DrugShopMstr_3$ where DS-city = 'X';
Select sum (Qty) into $sum from $Order_3$ where DS-id = $DS-id;

If not FOUND then
Select DS-id into $DS-id from $DrugShopMstr_4$ where DS-city = 'X';
Select sum (Qty) into $sum from $Order_4$ where DS-id = $DS-id;

If not FOUND then
Select DS-id into $DS-id from $DrugShopMstr_5$ where DS-city = 'X';
Select sum (Qty) into $sum from $Order_5$ where DS-id = $DS-id;

write ($sum);
[in output]

Local Mapping Transparency (Level 3):

Select DS-id into $DS-id from $DrugShopMstr_1$ AT SITE 1 where DS-city = 'X';
Select sum (Qty) into $sum from $Order_1$ AT SITE 1 where DS-id = $DS-id;

If not FOUND then
Select DS-id into $DS-id from $DrugShopMstr_2$ AT SITE 2 where DS-city = 'X';
Select sum (Qty) into $sum from $Order_2$ AT SITE 2 where DS-id = $DS-id;

If not FOUND then
Select DS-id into $DS-id from $DrugShopMstr_3$ AT SITE 3 where DS-city = 'X';
Select sum (Qty) into $sum from $Order_3$ AT SITE 3 where DS-id = $DS-id;

If not FOUND then
Select DS-id into $DS-id from $DrugShopMstr_4$ AT SITE 4 where DS-city = 'X';
Select sum (Qty) into $sum from $Order_4$ AT SITE 4 where DS-id = $DS-id;

If not FOUND then
Select DS-id into $DS-id from $DrugShopMstr_5$ AT SITE 5 where DS-city = 'X';
Select sum (Qty) into $sum from $Order_5$ AT SITE 5 where DS-id = $DS-id;

write ($sum);
[in output]

5. Consider the following global schema:

Supplier (S#, Sname, City)
Supply (S#, P#, Qty)
Parts (P#, Pname, Pcolor)

The S# value is in the range 1-465, inclusive of both. Design a horizontal fragmentation schema for the database assuming that most of the queries will be based on supplier number.

Comment on the disjointness properties in the context of the fragmentation schema suggested by you.

[C.U. M.Tech (CSE) – 2005]

Answer:

The horizontal fragmentation of the Supplier relation assuming most of the queries will be based on supplier number is as follows.

$Supplier_1 = SL_{City = 'Kolkata'}$ (Supplier)

$Supplier_2 = SL_{City = 'Delhi'}$ (Supplier)

$Supplier_3 = SL_{City = 'Chennai'}$ (Supplier)

$Supplier_4 = SL_{City = 'Mumbai'}$ (Supplier)

It has been assumed that there are only four possible values for the City attribute. The predicates which are relevant for describing this fragmentation cannot be deduced by analyzing the code of an application. In this example, most of the queries are based on supplier number and the horizontal fragmentation is based on the City attribute, therefore, the values of City generate the predicates and there is no reference to them in the query.

The horizontal fragmentation of the Supply relation is derived from the horizontal fragmentation of the Supplier relation. In this case,

$Supply_i = Supply\ SJ_{S\#}\ Supplier_i$, i = 1, 2, 3, 4.

The horizontal fragmentation of the Parts relation is derived from the fragmentation of the Supply relation. Therefore,

$Parts_i = Parts\ SJ_{P\#}\ Supply_i$, i = 1, 2, 3, 4.

In the above fragments, disjointness property of fragmentation has been ensured both in case of horizontal and derived fragmentation. In Supplier relation, a supplier does not belong to more than one cites, thus, after fragmentation each tuple of Supplier relation belongs to a single horizontal fragment. Hence, each tuple of Supplier relation belongs to either in $Supplier_1$ or in $Supplier_2$, or in $Supplier_3$, or in $Supplier_4$, thereby, the disjointness property is ensured. Since, the horizontal fragments of Supply relation are derived from the Supplier relation based on the same predicates, the disjointness property of fragmentation can be guaranteed in the same way. Similarly, the horizontal fragments of the Parts relation are derived from the Supply relations based on the same predicates. In this case, the disjointness property of fragmentation is also ensured.

6. Consider a Movie Multiplex reservation System. The information to be stored is given below:

- Show information: show id, show time, auditorium number, total number of seats, number of seats reserved and price of tickets.
- Viewer information: Viewer id, name, address, phone number.
- Reservation description: Viewer id, show id, seat number.

Design a distributed database solution for three booking counters across the city considering the following frequent queries/operations:

- Availability of tickets: Booking information of a particular show is displayed on the screen.
- Request for reservation: Checking for and inserting viewer's data for new customers; checking for availability of seats, inserting reservation description.

Your design should include the definition of global schema, fragmentation schema and allocation schema.

[C. U. M.Tech (CSE) – 2006 & 2007]

Answer:

The global schema, fragmentation schema and allocation schema for Movie Multiplex Reservation System is as follows:

Global Schema: SHOW (show-id, show-time, auditorium-no, totalseat, rseat, price)

VIEWER (viewer-id, name, address, phone-number)

RESERVATION (viewer-id, show-id, seat-no)

Fragmentation Schema: $SHOW_1 = SL_{\text{show-time = "MATINEE"}}$ (SHOW)

$SHOW_2 = SL_{\text{show-time = "EVENING"}}$ (SHOW)

$SHOW_3 = SL_{\text{show-time = "NIGHT"}}$ (SHOW)

$RESERVATION_i$ = RESERVATION $SJ_{\text{show-id}}$ SHOWi, where i = 1, 2, 3.

VIEWERi = VIEWER $SJ_{\text{viewer-id}}$ RESERVATIONi, where i = 1, 2, 3.

Allocation Schema: $SHOW_1$, $RESERVATION_1$, $VIEWER_1$ at Booking Counter No.1 site (say site no. 1).

$SHOW_2$, $RESERVATION_2$, $VIEWER_2$ at Booking Counter No. 2 site (say site no. 2).

$SHOW_3$, $RESERVATION_3$, $VIEWER_3$ at Booking Counter No. 3 site (say site no. 3).

In the above distributed database design, only three show times, that is, 'MATINEE', 'EVENING', and 'NIGHT' have been considered. Further, it has been assumed that most of the tickets for MATINEE show will be issued from the booking counter no. 1. Similarly, most of the tickets for EVENING show will be issued from the booking counter no. 2 and most of the tickets for NIGHT show will be issued from the booking counter no.3. Based on these assumptions, local data are stored locally here, which is beneficial for any distributed database system. Therefore, all information regarding the MATINEE show such as show details, viewer's details and reservation details have been stored at booking counter no. 1 locally. Similarly, the detailed information for EVENING show has been stored at booking counter no. 2 and the detailed information for NIGHT show has been stored at booking counter no. 3. In the above distributed database design, the following predicates are complete and minimal for SHOW relation.

p1: show-time = "MATINEE"
p2: show-time = "EVENING"-
p3: show-time = "NIGHT".

Application for availability of tickets:

Fragmentation Transparency (Level 1):

read ($show-id);
Select show-time, auditorium-no, totalseat, rseat, price into $show-time, $auditorium-no, $totalseat, $rseat, $price from SHOW where show-id = $show-id;

write ($show-time, $auditorium-no, $totalseat, $rseat, $price);
[in output]

Location Transparency (Level 2):

read ($show-id);
Select show-time, auditorium-no, totalseat, rseat, price into $show-time, $auditorium-no, $totalseat, $rseat, $price from $SHOW_1$ where show-id = $show-id;

if not FOUND then
Select show-time, auditorium-no, totalseat, rseat, price into $show-time, $auditorium-no, $totalseat, $rseat, $price from $SHOW_2$ where show-id = $show-id;

if not FOUND then
Select show-time, auditorium-no, totalseat, rseat, price into $show-time, $auditorium-no, $totalseat, $rseat, $price from $SHOW_3$ where show-id = $show-id;

write ($show-time, $auditorium-no, $totalseat, $rseat, $price);
[in output]

Local Mapping Transparency (Level 3):

read ($show-id);
Select show-time, auditorium-no, totalseat, rseat, price into $show-time, $auditorium-no, $totalseat, $rseat, $price from $SHOW_1$ AT SITE 1 where show-id = $show-id;

if not FOUND then
Select show-time, auditorium-no, totalseat, rseat, price into $show-time, $auditorium-no, $totalseat, $rseat, $price from $SHOW_2$ AT SITE 2 where show-id = $show-id;

if not FOUND then
Select show-time, auditorium-no, totalseat, rseat, price into $show-time, $auditorium-no, $totalseat, $rseat, $price from $SHOW_3$ AT SITE 3 where show-id = $show-id;

write ($show-time, $auditorium-no, $totalseat, $rseat, $price);
[in output]

Application for Request for reservation:

Fragmentation Transparency (Level 1):

read ($show-id, $viewer-id);
Select totalseat, rseat into $totalseat, $rseat from SHOW where show-id = $show-id;
If ($totalseat - $rseat) > 0 then [if seats are available]
    Select name, address, phone-number into $name, $address, $phone-number from VIEWER where viewer-id = $viewer-id;
    insert into RESERVATION (viewer-id, show-id, seat-no) values ($viewer-id, $show-id, $seat-no);

    if not FOUND then
    read (nviewer-id, nname, naddress, nphone-number);
    insert into VIEWER (viewer-id, name, address, phone-number) values (nviewer-id, 'nname', 'naddress', 'nphone-number');
    insert into RESERVATION (viewer-id, show-id, seat-no) values (nviewer-id, $show-id, $seat-no);

Location Transparency (Level 2):

read ($show-id, $viewer-id);
Select totalseat, rseat into $totalseat, $rseat from $SHOW_1$ where show-id = $show-id;
If ($totalseat - $rseat) > 0 then [if seats are available]
    Select name, address, phone-number into $name, $address, $phone-number from $VIEWER_1$ where viewer-id = $viewer-id;
    insert into $RESERVATION_1$ (viewer-id, show-id, seat-no) values ($viewer-id, $show-id, $seat-no);

```
        if not FOUND then
        read (nviewer-id, nname, naddress, nphone-number);
        insert into VIEWER1 (viewer-id, name, address, phone-number) values
        (nviewer-id, 'nname', 'naddress', 'nphone-number');
        insert into RESERVATION1 (viewer-id, show-id, seat-no) values (nviewer-id, $show-id,
        $seat-no);
if not FOUND then
Select totalseat, rseat into $totalseat, $rseat from SHOW2 where show-id = $show-id;
If ($totalseat - $rseat) > 0 then [if seats are available]
        Select name, address, phone-number into $name, $address, $phone-number from VIEWER2 where viewer-id =
        $viewer-id;
        insert into RESERVATION2 (viewer-id, show-id, seat-no) values ($viewer-id, $show-id, $seat-no);

        if not FOUND then
        read (nviewer-id, nname, naddress, nphone-number);
        insert into VIEWER2 (viewer-id, name, address, phone-number) values
        (nviewer-id, 'nname', 'naddress', 'nphone-number');
        insert into RESERVATION2 (viewer-id, show-id, seat-no) values (nviewer-id, $show-id, $seat-no);

if not FOUND then
Select totalseat, rseat into $totalseat, $rseat from SHOW3 where show-id = $show-id;
If ($totalseat - $rseat) > 0 then [if seats are available]
        Select name, address, phone-number into $name, $address, $phone-number from VIEWER3 where viewer-id =
        $viewer-id;
        insert into RESERVATION3 (viewer-id, show-id, seat-no) values ($viewer-id, $show-id,
        $seat-no);

        if not FOUND then
        read (nviewer-id, nname, naddress, nphone-number);
        insert into VIEWER3 (viewer-id, name, address, phone-number) values
        (nviewer-id, 'nname', 'naddress', 'nphone-number');
        insert into RESERVATION3 (viewer-id, show-id, seat-no) values (nviewer-id, $show-id, $seat-no);
```

Local Mapping Transparency (Level 3):

```
read ($show-id, $viewer-id);
Select totalseat, rseat into $totalseat, $rseat from SHOW1 AT SITE 1 where show-id = $show-id;
If ($totalseat - $rseat) > 0 then [if seats are available]
        Select name, address, phone-number into $name, $address, $phone-number from VIEWER1 AT SITE 1 where
        viewer-id = $viewer-id;
        insert into RESERVATION1 AT SITE 1 (viewer-id, show-id, seat-no) values
        ($viewer-id, $show-id, $seat-no);

        if not FOUND then
        read (nviewer-id, nname, naddress, nphone-number);
        insert into VIEWER1 AT SITE 1 (viewer-id, name, address, phone-number) values
        (nviewer-id, 'nname', 'naddress', 'nphone-number');
        insert into RESERVATION1 AT SITE 1 (viewer-id, show-id, seat-no) values
        (nviewer-id, $show-id, $seat-no);

if not FOUND then
Select totalseat, rseat into $totalseat, $rseat from SHOW2 AT SITE 2 where show-id = $show-id;
If ($totalseat - $rseat) > 0 then [if seats are available]
        Select name, address, phone-number into $name, $address, $phone-number from VIEWER2 AT SITE 2 where
        viewer-id = $viewer-id;
        insert into RESERVATION2 AT SITE 2 (viewer-id, show-id, seat-no) values
        ($viewer-id, $show-id, $seat-no);
```

if not FOUND then
read (nviewer-id, nname, naddress, nphone-number);
insert into $VIEWER_2$ AT SITE 2 (viewer-id, name, address, phone-number) values
(nviewer-id, 'nname', 'naddress', 'nphone-number');
insert into $RESERVATION_2$ AT SITE 2 (viewer-id, show-id, seat-no) values
(nviewer-id, $show-id, $seat-no);

if not FOUND then
Select totalseat, rseat into $totalseat, $rseat from $SHOW_3$ AT SITE 3 where show-id = $show-id;
If ($totalseat - $rseat) > 0 then [if seats are available]
Select name, address, phone-number into $name, $address, $phone-number from $VIEWER_3$ AT SITE 3 where viewer-id = $viewer-id;
insert into $RESERVATION_3$ AT SITE 3 (viewer-id, show-id, seat-no) values
($viewer-id, $show-id, $seat-no);

if not FOUND then
read (nviewer-id, nname, naddress, nphone-number);
insert into $VIEWER_3$ AT SITE 3 (viewer-id, name, address, phone-number) values
(nviewer-id, 'nname', 'naddress', 'nphone-number');
insert into $RESERVATION_3$ AT SITE 3 (viewer-id, show-id, seat-no) values
(nviewer-id, $show-id, $seat-no);

7. Consider the following global, fragmentation and allocation schema:

Global Schema: Guest (guest-ID, block-ID, room-no, room-type, name, city, phone-no)

Fragmentation Schema:
$Guest_1 = PJ_{\text{guest-ID, block-ID, room-no, room-type}} (SL_{\text{block-ID = "NORTH"}} (Guest))$
$Guest_2 = PJ_{\text{guest-ID, block-ID, room-no, room-type}} (SL_{\text{block-ID = "SOUTH"}} (Guest))$
$Guest_3 = PJ_{\text{guest-ID, name, city, phone-no}} (SL_{\text{block-ID = "NORTH"}} (Guest))$
$Guest_4 = PJ_{\text{guest-ID, name, city, phone-no}} (SL_{\text{block-ID = "SOUTH"}} (Guest))$

Allocation Schema:
$Guest_1$ at site 1, 2
$Guest_2$ at site 3, 4
$Guest_3$ at site 5, 6
$Guest_4$ at site 7, 8

(i) Write a query that accepts the guest-ID from terminal and outputs the name, block-ID and the room number at levels 1, 2, and 3 of transparency.

(ii) Write a query that moves a guest having guest-ID = 41 from the NORTH block to the SOUTH block at levels 1, 2, and 3 of transparency.

[C. U. M.Tech (CSE) – 2006 & 2007]

Answer:

(i) Query for displaying guest details:

Fragmentation Transparency (Level 1):

read ($guest-ID);
Select name, block-ID, room-no into $name, $block-ID, $room-no from GUEST
where guest-ID = $guest-ID;
write ($name, $block-ID, $room-no); [in output]

Location Transparency (Level 2):

read ($guest-ID);
Select block-ID, room-no into $block-ID, $room-no from $GUEST_1$

where guest-ID = $guest-ID;
Select name into $name from $GUEST_3$
where guest-ID = $guest-ID;

if not FOUND then
Select block-ID, room-no into $block-ID, $room-no from $GUEST_2$
where guest-ID = $guest-ID;
Select name into $name from $GUEST_4$
where guest-ID = $guest-ID;

write ($name, $block-ID, $room-no); [in output]

Local Mapping Transparency (Level 3):

read ($guest-ID);
Select block-ID, room-no into $block-ID, $room-no from $GUEST_1$ AT SITE 1
where guest-ID = $guest-ID;
Select name into $name from $GUEST_3$ AT SITE 5
where guest-ID = $guest-ID;

if not FOUND then
Select block-ID, room-no into $block-ID, $room-no from $GUEST_2$ AT SITE 3
where guest-ID = $guest-ID;
Select name into $name from $GUEST_4$ AT SITE 7
where guest-ID = $guest-ID;

write ($name, $block-ID, $room-no); [in output]

(ii) Query for moving the guest with guest-ID 41 from NORTH block to SOUTH block:

Fragmentation Transparency (Level 1):

Update GUEST
set block-ID = "SOUTH"
where guest-ID = 41;

Location Transparency (Level 2):

Select room-no, room-type into $room-no, $room-type from $GUEST_1$
where guest-ID = 41;
Select name, city, phone-no into $name, $city, $phone-no from $GUEST_3$
where guest-ID = 41;

insert into $GUEST_2$ (guest-ID, block-Id, room-no, room-type)
values (41, 'SOUTH', $room-no, '$room-type');
insert into $GUEST_4$ (guest-ID, name, city, phone-no)
values (41, '$name', '$city', '$phone-no');

Delete from $GUEST_1$ where guest-ID = 41;
Delete from $GUEST_3$ where guest-ID = 41;

Local Mapping Transparency (Level 3):

Select room-no, room-type into $room-no, $room-type from $GUEST_1$ AT SITE 1
where guest-ID = 41;
Select name, city, phone-no into $name, $city, $phone-no from $GUEST_3$ AT SITE 5
where guest-ID = 41;

insert into $GUEST_2$ AT SITE 3 (guest-ID, block-Id, room-no, room-type)
values (41, 'SOUTH', $room-no, '$room-type');
insert into $GUEST_2$ AT SITE 4 (guest-ID, block-Id, room-no, room-type)
values (41, 'SOUTH', $room-no, '$room-type');

insert into $GUEST_4$ AT SITE 7 (guest-ID, name, city, phone-no)
values (41, '$name', '$city', '$phone-no');

insert into $GUEST_4$ AT SITE 8 (guest-ID, name, city, phone-no)
values (41, '$name', '$city', '$phone-no');

Delete from $GUEST_1$ AT SITE 1where guest-ID = 41;
Delete from $GUEST_1$ AT SITE 2where guest-ID = 41;
Delete from $GUEST_3$ AT SITE 5 where guest-ID = 41;
Delete from $GUEST_3$ AT SITE 6 where guest-ID = 41;

8. Consider the relation EMP and PAY as follows:

EMP

ENO	ENAME	TITLE
E1	J. Doe	Elect. Eng.
E2	M. Smith	Syst. Anal.
E3	A. Lee	Mech. Eng.
E4	J. Miller	Programmer
E5	B. Cassey	Syst. Anal.
E6	L. Chu	Elect. Eng.
E7	R. Davis	Mech. Eng
E8	J. Jones	Syst. Anal.

PAY

TITLE	SALARY
Elect. Eng.	45,000
Syst. Anal.	32,000
Mech. Eng.	28,000
Programmer	25,000

Assume, p1: SAL < 28000 and p2: SAL ≥ 28000 are two simple predicates. Perform a horizontal fragmentation of PAY with respect to predicates p1 and p2 to obtain two fragments PAY1 and PAY2. Using these fragments perform derived horizontal fragmentation for EMP and show completeness, reconstruction and disjointness of the fragmentation of EMP.
[WBUT B.TECH (CSE) – 2005]

Answer:

Using the predicates p1 and p2, the horizontal fragmentation PAY1 and PAY2 of relation PAY are as follows:

PAY1

TITLE	SALARY
Programmer	25,000

PAY2

TITLE	SALARY
Elect. Eng.	45,000
Syst. Anal.	32,000
Mech. Eng.	28,000

Based on the horizontal fragments of relation PAY, the derived horizontal fragments of relation EMP are as follows:

EMP1

ENO	ENAME	TITLE
E1	J. Doe	Elect. Eng.
E2	M. Smith	Syst. Anal.
E3	A. Lee	Mech. Eng.
E5	B. Cassey	Syst. Anal.
E6	L. Chu	Elect. Eng.
E7	R. Davis	Mech. Eng.
E8	J. Jones	Syst. Anal.

EMP2

ENO	ENAME	TITLE
E4	J. Miller	Programmer

Hence, EMPi = EMP SJ_{TITLE} PAYi, where i = 1,2.

Since any tuple of EMP relation is appeared either in EMP1 or EMP2, there is no loss of information after fragmentation, thereby the completeness rule is ensured. Moreover, origin relations can be reconstructed here from fragmented relations by using union and semijoin operations of relational algebra. Thus, the reconstruction rule of fragmentation is guaranteed. Further, any tuple of EMP relation is appeared either in EMP1 or EMP2, but not is both. Therefore, the disjointness rule of fragmentation is also ensured here.

9. Consider an airline reservation database. The information to be stored is given below.
   - » Flight description: flight number, departure and arrival place and time, total number of seats, number of seats reserved, fare.
   - » Passenger description: passenger code, name, address, phone number.
   - » Reservation description: passenger code, flight number, reserved seat number.

   Design a distributed database for the above case over four sites at Kolkata, Delhi, Mumbai and Chennai considering the following major applications:
   - » Request about flight availability: all information about flight is displayed on the screen.
   - » Request for reservation: checking for and inserting passenger's data for new passengers, checking for availability of seats, inserting reservation description.

   Your design should include the definition of global schema, fragmentation schema and allocation schema.

   [C.U. M.Tech (CSE) – 2003]

Answer:

The global schema, fragmentation schema and allocation schema for distributed airline reservation database is as follows:

Global Schema:
FLIGHT (FNO, DPLACE, DTIME, APLACE, ATIME, TOTALSEAT, FARE)
PASSENGER (PCODE, NAME, ADDRESS, PHONENO)
RESERVATION (PCODE, FNO, SEATNO)

Fragmentation Schema:
$FLIGHT_1 = SL_{DPLACE = "KOLKATA"}$ (FLIGHT)
$FLIGHT_2 = SL_{DPLACE = "DELHI"}$ (FLIGHT)
$FLIGHT_3 = SL_{DPLACE = "MUMBAI"}$ (FLIGHT)
$FLIGHT_4 = SL_{DPLACE = "CHENNAI"}$ (FLIGHT)

$RESERVATION_i$ = RESERVATION SJ_{FNO} FLIGHTi, where i = 1,2,3,4.

PASSENGERi = PASSENGER SJ_{PCODE} RESERVATIONi, where i = 1,2,3,4.

Allocation Schema:
$FLIGHT_1$, $RESERVATION_1$, $PASSENGER_1$ at KOLKATA site (say site no. 1).
$FLIGHT_2$, $RESERVATION_2$, $PASSENGER_2$ at DELHI site (say site no. 2).
$FLIGHT_3$, $RESERVATION_3$, $PASSENGER_3$ at MUMBAI site (say site no. 3).
$FLIGHT_4$, $RESERVATION_4$, $PASSENGER_4$ at CHENNAI site (say site no. 4).

In the above distributed database design, only those flight information have been considered whose departure place are either KOLKATA, or DELHI, or MUMBAI, or CHENNEI. In case of distributed database design, it should be beneficial if local data are stored locally. Thus, all flight information regarding the flights whose departure place is KOLKATA are stored at KOLKATA site, all flight information regarding the flights whose departure

place is DELHI are stored at DELHI site and so on. Similarly, the corresponding PASSENGER and RESERVATION information are also stored locally. In this case FLIGHT relation, the following predicates are complete and minimal.

p1: DPLACE = "KOLKATA"
p2: DPLACE = "DELHI"
p3: DPLACE = "MUMBAI"
p4: DPLACE = "CHENNAI".

Application for Request about flight availability:

Fragmentation Transparency (Level 1):

read (QFNO);
Select DPLACE, DTIME, APLACE, ATIME, TOTALSEAT, FARE into $DPLACE, $DTIME, $APLACE, $ATIME, $TOTALSEAT, $FARE from FLIGHT where FNO = QFNO;
write ($DPLACE, $DTIME, $APLACE, $ATIME, $TOTALSEAT, $FARE);
[in output]

Location Transparency (Level 2):

read (QFNO);
Select DPLACE, DTIME, APLACE, ATIME, TOTALSEAT, FARE into $DPLACE, $DTIME, $APLACE, $ATIME, $TOTALSEAT, $FARE from $FLIGHT_1$ where FNO = QFNO;

if not FOUND then
Select DPLACE, DTIME, APLACE, ATIME, TOTALSEAT, FARE into $DPLACE, $DTIME, $APLACE, $ATIME, $TOTALSEAT, $FARE from $FLIGHT_2$ where FNO = QFNO;

if not FOUND then
Select DPLACE, DTIME, APLACE, ATIME, TOTALSEAT, FARE into $DPLACE, $DTIME, $APLACE, $ATIME, $TOTALSEAT, $FARE from $FLIGHT_3$ where FNO = QFNO;

if not FOUND then
Select DPLACE, DTIME, APLACE, ATIME, TOTALSEAT, FARE into $DPLACE, $DTIME, $APLACE, $ATIME, $TOTALSEAT, $FARE from $FLIGHT_4$ where FNO = QFNO;

write ($DPLACE, $DTIME, $APLACE, $ATIME, $TOTALSEAT, $FARE);
[in output]

Local Mapping Transparency (Level 3):

read (QFNO);
Select DPLACE, DTIME, APLACE, ATIME, TOTALSEAT, FARE into $DPLACE, $DTIME, $APLACE, $ATIME, $TOTALSEAT, $FARE from $FLIGHT_1$ AT SITE 1 where FNO = QFNO;

if not FOUND then
Select DPLACE, DTIME, APLACE, ATIME, TOTALSEAT, FARE into $DPLACE, $DTIME, $APLACE, $ATIME, $TOTALSEAT, $FARE from $FLIGHT_2$ AT SITE 2 where FNO = QFNO;

if not FOUND then
Select DPLACE, DTIME, APLACE, ATIME, TOTALSEAT, FARE into $DPLACE, $DTIME, $APLACE, $ATIME, $TOTALSEAT, $FARE from $FLIGHT_3$ AT SITE 3 where FNO = QFNO;

if not FOUND then
Select DPLACE, DTIME, APLACE, ATIME, TOTALSEAT, FARE into $DPLACE, $DTIME, $APLACE, $ATIME, $TOTALSEAT, $FARE from $FLIGHT_4$ AT SITE 4 where FNO = QFNO;

write ($DPLACE, $DTIME, $APLACE, $ATIME, $TOTALSEAT, $FARE);
[in output]

Application for Request about reservation:

Fragmentation Transparency (Level 1):

```
read (PNO, RFNO);
Select TOTALSEAT into $TOTALSEAT from FLIGHT where FNO = RFNO;
If $TOTALSEAT > 0 then [if seats are available]
        Select NAME, ADDRESS, PHONENO into $NAME, $ADDRESS, $PHONENO from PASSENGER where
        PCODE = PNO;
        insert into RESERVATION (PCODE, FNO, SEATNO) values (PCODE, FNO, $SEATNO);

        if not FOUND then
        read (NPCODE, NNAME, NADDRESS, NPHONENO);
        insert into PASSENGER(PCODE, NAME, ADDRESS, PHONENO) values
        (NPCODE, 'NNAME', 'NADDRESS', 'NPHONENO');
        insert into RESERVATION (PCODE, FNO, SEATNO) values (NPCODE, RFNO, $SEATNO);
```

Location Transparency (Level 2):

```
read (PNO, RFNO);
Select TOTALSEAT into $TOTALSEAT from FLIGHT1 where FNO = RFNO;
If $TOTALSEAT > 0 then [if seats are available]
        Select NAME, ADDRESS, PHONENO into $NAME, $ADDRESS, $PHONENO from PASSENGER1 where
        PCODE = PNO;
        insert into RESERVATION1 (PCODE, FNO, SEATNO) values (PCODE, FNO, $SEATNO);

        if not FOUND then
        read (NPCODE, NNAME, NADDRESS, NPHONENO);
        insert into PASSENGER1 (PCODE, NAME, ADDRESS, PHONENO) values
        (NPCODE, 'NNAME', 'NADDRESS', 'NPHONENO');
        insert into RESERVATION1 (PCODE, FNO, SEATNO) values (NPCODE, RFNO, $SEATNO);

if not FOUND then
Select TOTALSEAT into $TOTALSEAT from FLIGHT2 where FNO = RFNO;
If $TOTALSEAT > 0 then [if seats are available]
        Select NAME, ADDRESS, PHONENO into $NAME, $ADDRESS, $PHONENO from PASSENGER2 where
        PCODE = PNO;
        insert into RESERVATION2 (PCODE, FNO, SEATNO) values (PCODE, FNO, $SEATNO);

        if not FOUND then
        read (NPCODE, NNAME, NADDRESS, NPHONENO);
        insert into PASSENGER2 (PCODE, NAME, ADDRESS, PHONENO) values
        (NPCODE, 'NNAME', 'NADDRESS', 'NPHONENO');
        insert into RESERVATION2 (PCODE, FNO, SEATNO) values (NPCODE, RFNO, $SEATNO);

if not FOUND then
Select TOTALSEAT into $TOTALSEAT from FLIGHT3 where FNO = RFNO;
If $TOTALSEAT > 0 then [if seats are available]
        Select NAME, ADDRESS, PHONENO into $NAME, $ADDRESS, $PHONENO from PASSENGER3 where
        PCODE = PNO;
        insert into RESERVATION3 (PCODE, FNO, SEATNO) values (PCODE, FNO, $SEATNO);

        if not FOUND then
        read (NPCODE, NNAME, NADDRESS, NPHONENO);
        insert into PASSENGER3 (PCODE, NAME, ADDRESS, PHONENO) values
        (NPCODE, 'NNAME', 'NADDRESS', 'NPHONENO');
        insert into RESERVATION3 (PCODE, FNO, SEATNO) values (NPCODE, RFNO, $SEATNO);

if not FOUND then
Select TOTALSEAT into $TOTALSEAT from FLIGHT4 where FNO = RFNO;
```

If $TOTALSEAT > 0 then [if seats are available]
Select NAME, ADDRESS, PHONENO into $NAME, $ADDRESS, $PHONENO from $PASSENGER_4$ where PCODE = PNO;
insert into $RESERVATION_4$ (PCODE, FNO, SEATNO) values (PCODE, FNO, $SEATNO);

if not FOUND then
read (NPCODE, NNAME, NADDRESS, NPHONENO);
insert into $PASSENGER_4$ (PCODE, NAME, ADDRESS, PHONENO) values
(NPCODE, 'NNAME', 'NADDRESS', 'NPHONENO');
insert into $RESERVATION_4$ (PCODE, FNO, SEATNO) values (NPCODE, RFNO, $SEATNO);

The above application at the level of local mapping transparency can be written in the same way.

10. Simplify the following query using the idempotency rules:

SELECT ENO FROM ASG WHERE (NOT (TITLE = 'PROGRAMMER') AND (TITLE = 'PROGRAMMER' OR TITLE = 'ELECT.ENG') AND NOT (TITLE = 'ELECT.ENG')) OR ENAME = 'J.DAS'.

[WBUT B.TECH (CSE) – 2002]

Answer:

Let p1 = TITLE = 'PROGRAMMER', p2 = TITLE = 'ELECT. ENG', and p3 = ENAME = 'J.DAS'. Thus, the above query qualification can be rewrite as follows:

$(\neg p1 \wedge (p1 \vee p2) \wedge \neg p2) \vee p3$

By applying the idempotency rule, $[p1 \wedge (p2 \vee p3) \Leftrightarrow (p1 \wedge p2) \vee (p1 \wedge p3)]$, the disjunctive normal form for the above qualification can be rewritten as

$(\neg p1 \wedge ((p1 \wedge \neg p2) \vee (p2 \wedge \neg p2))) \vee p3$

$\Leftrightarrow (\neg p1 \wedge p1 \wedge \neg p2) \vee (\neg p1 \wedge p2 \wedge \neg p2) \vee p3$ [By applying the same idempotency rule]

$\Leftrightarrow (false \wedge \neg p2) \vee (\neg p1 \wedge false) \vee p3$ [By applying idempotency rule]

$\Leftrightarrow false \vee false \vee p3$

$\Leftrightarrow p3$

Thus, the above SQL query can be simplified as

SELECT ENO FROM ASG
WHERE ENAME = 'J.DAS'.

11. Consider the following schemas:

EMP = (ENO, ENAME, TITLE)

ASG = (ENO, PNO, RESP, DUR)

PROJ = (PNO, PNAME, BUDGET, LOC)

Consider the following query:

SELECT ENAME, PNAME FROM EMP, ASG, PROJ WHERE EMP.ENO = ASG.ENO AND ASG.PNO = PROJ.PNO AND (TITLE = 'ELECT.ENG' OR ASG.PNO < 'P3')

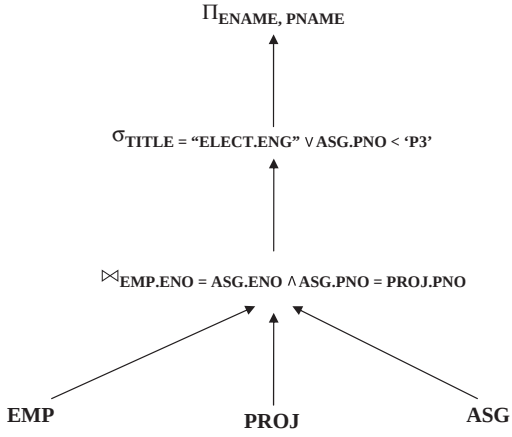
(a) Draw the canonical tree.

(b) Transform the canonical tree obtained in (a) to an optimized tree.

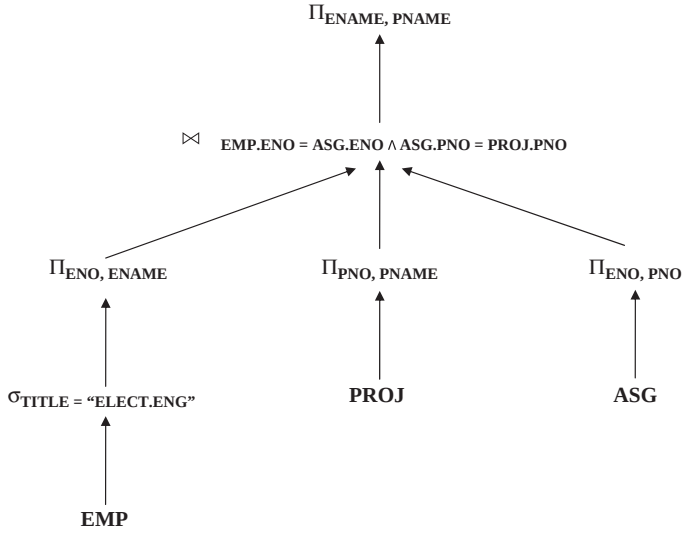
[WBUT B.TECH (CSE) – 2002]

Answer:

(a) The canonical tree for the above query is as follows.



(b) The optimized query tree for the above query is listed below.



12. Consider a relation that is fragmented horizontally by plant-number:

EMP (NAME, ADDRESS, SALARY, PLANT-NO)

Assume that each fragment has two replicas:
One stored at Durgapur City site and one stored locally at the plant site. Describe a good processing strategy for the following queries entered at Kolkata.

(i) Find all employees at the Jamshedpur plant.

(ii) Find the average salary of all employees.

(iii) Find the highest paid employee at each of the following site Bokaro, Asansol, Dhanbad and Haldia.

(iv) Find the lowest-paid employee in the company.

[WBUT (MCA) – 2003]

Answer:

The relation EMP is horizontally fragmented by PLANT-NO, thus, the horizontal fragments are as follows:

$EMP_1 = SL_{PLANT\text{-}NO = 'Jamshedpur'}$ (EMP)
$EMP_2 = SL_{PLANT\text{-}NO = 'Bokaro'}$ (EMP)
$EMP_3 = SL_{PLANT\text{-}NO = 'Asansol'}$ (EMP)
$EMP_4 = SL_{PLANT\text{-}NO = 'Dhanbad'}$ (EMP)
$EMP_5 = SL_{PLANT\text{-}NO = 'Haldia'}$ (EMP)

……
……
……

Further, each fragment has two replicas, one replica is stored at the plant site locally and the other replica is stored at Durgapur city site. In this case, Durgapur city site contains the replicas of all horizontal fragments of the EMP relation and each local site contains only one replica of the corresponding horizontal fragment of local plant. A good query processing strategy for each of the above query is listed below.

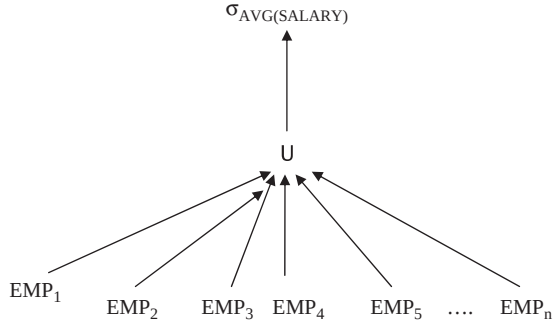
» **Find all employees at the Jamshedpur plant.**

Select * from EMP_1

» **Find the average salary of all employees.**

Select AVG (SALARY) from EMP

This query should be processed at Durgapur city site, since one replica of all horizontal fragments of EMP relation is stored here. By using union operation of relational algebra, the original EMP relation can be reconstructed from its horizontal fragments. Thus, the query processing strategy for the above query is as follows:



» **Find the highest paid employee at each of the following site Bokaro, Asansol, Dhanbad and Haldia.**

Select NAME from EMP_2 where SALARY = (Select MAX (SALARY) from EMP_2)
UNION
Select NAME from EMP_3 where SALARY = (Select MAX (SALARY) from EMP_3)
UNION
Select NAME from EMP_4 where SALARY = (Select MAX (SALARY) from EMP_4)
UNION
Select NAME from EMP_5 where SALARY = (Select MAX (SALARY) from EMP_5)

» **Find the lowest-paid employee in the company.**

Select NAME from EMP where SALARY = (Select MIN (SALARY) from EMP)

This query should be also processed at Durgapur city site like query number (ii). Initially, the original relation EMP will be reconstructed from its horizontal fragments by using the UNION operation of relational algebra and then the query will be processed to find the lowest-paid employee in the company.

13. Simplify the following query using the idempotency rules:

Select ENO from ASG where RESP = "Analyst" AND NOT (PNO = "P2" OR DUR = 12) AND PNO ≠ "P2" AND DUR = 12

Considering ASG (ENO, PNO, RESP, DUR).

[WBUT (MCA) – 2003 & 2004]

Answer:

Let p1 = RESP = "Analyst", p2 = PNO = "P2", and p3 = DUR = 12. Thus, the above query qualification can be rewrite as follows:

$p1 \wedge \neg (p2 \vee p3) \wedge \neg p2 \wedge p3$

By applying the idempotency rule, $[\neg (p1 \vee p2) \Leftrightarrow (\neg p1 \wedge \neg p2)]$, the disjunctive normal form for the above qualification can be rewritten as

$p1 \wedge (\neg p2 \wedge \neg p3) \wedge \neg p2 \wedge p3$

$\Leftrightarrow p1 \wedge \neg p2 \wedge \neg p3 \wedge \neg p2 \wedge p3$

$\Leftrightarrow p1 \wedge (\neg p2 \wedge \neg p2) \wedge (\neg p3 \wedge p3)$ [By applying the same idempotency rule]

$\Leftrightarrow p1 \wedge (\neg p2) \wedge (false)$ [By applying idempotency rule]

$\Leftrightarrow p1 \wedge \neg p2 \wedge false$

$\Leftrightarrow p1 \wedge (\neg p2 \wedge false)$

$\Leftrightarrow p1 \wedge false$

$\Leftrightarrow false$

Thus, the above SQL query can be simplified as

Select ENO from ASG.

14. Optimize the following query"

List the flats that are for rent along with the corresponding branch details.

Relations: BRANCH (BranchNo, Street, City, PostCode)

PROPFORRENT (PropNo, Addr, Type, RentAmount, OwnerNo, BranchNo)

Partitions:

P1: $\sigma_{\text{BranchNo = 'B003'} \wedge \text{Type = 'HOUSE'}}$ (PROPFORRENT)

P2: $\sigma_{\text{BranchNo = 'B003'} \wedge \text{Type = 'FLAT'}}$ (PROPFORRENT)

P3: $\sigma_{\text{BranchNo} \neq \text{'B003'}}$ (PROPFORRENT)

B1: $\sigma_{\text{BranchNo = 'B003'}}$ (BRANCH)

B2: $\sigma_{\text{BranchNo} \neq \text{'B003'}}$ (BRANCH)

Write the SQL. Write the corresponding relational algebraic expression. Then optimize the query.

[WBUT (MCA) – 2008]

Answer:

By using SQL, the above query can be represented as:

SELECT PropNo, BranchNo, Street, City, PostCode FROM PROPFORRENT, BRANCH WHERE Type = 'FLAT' AND PROPFORRENT.BranchNo = BRANCH.BranchNo

In relational algebraic expression, the above SQL query can be represented as follows.

$\Pi_{\text{PropNo, BranchNo, Street, City, PostCode}}$ ($\sigma_{\text{Type = 'FLAT'} \wedge \text{PROPFORRENT.BranchNo = BRANCH.BranchNo}}$ (PROPFORRENT $\bowtie$ BRANCH))

The horizontal fragments of PROPFORRENT relation are listed below.

$\text{PROPFORRENT}_1 = \sigma_{\text{BranchNo = 'B003'} \wedge \text{Type = 'HOUSE'}}$ (PROPFORRENT)

$\text{PROPFORRENT}_2 = \sigma_{\text{BranchNo = 'B003'} \wedge \text{Type = 'FLAT'}}$ (PROPFORRENT)

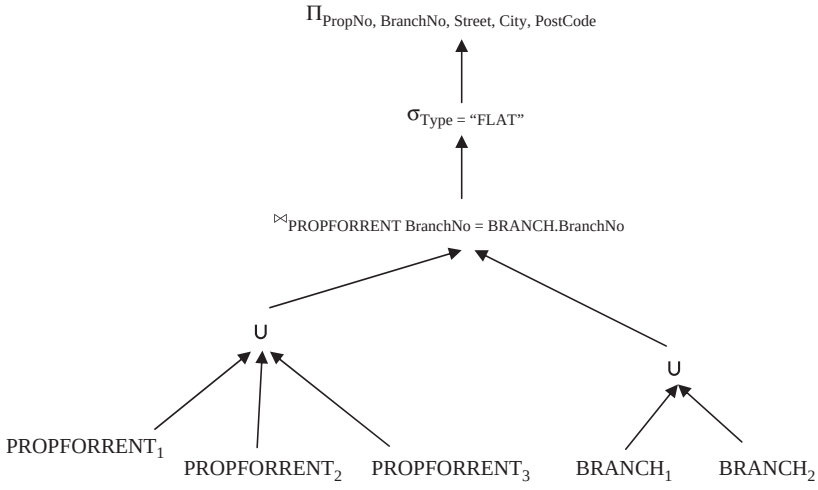
$\text{PROPFORRENT}_3 = \sigma_{\text{BranchNo} \neq \text{'B003'}}$ (PROPFORRENT)

Similarly, the horizontal fragments of BRANCH relation are as follows.

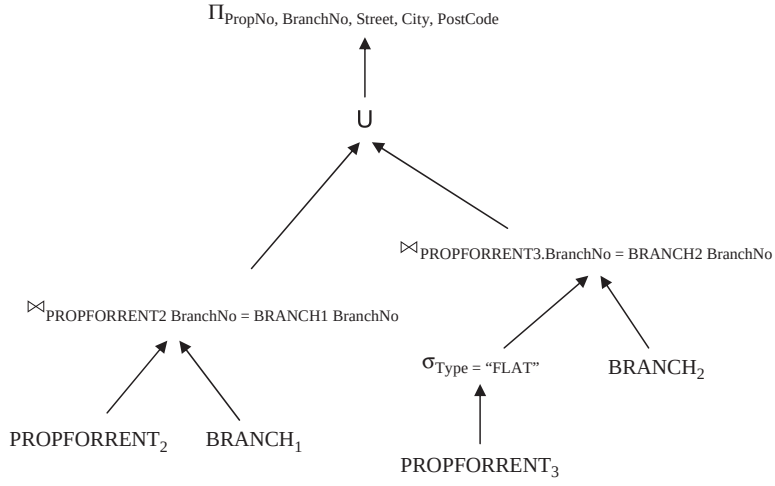
$\text{BRANCH}_1 = \sigma_{\text{BranchNo = 'B003'}}$ (BRANCH)

$\text{BRANCH}_2 = \sigma_{\text{BranchNo} \neq \text{'B003'}}$ (BRANCH)

The canonical tree for the above query is



The optimized query tree for the above query is as follows.



15. Consider the following Schema:

EMP = (ENO, ENAME, TITLE)
ASG = (ENO, PNO, RESP, DUR)
PROJ = (PNO, PNAME, BUDGET)

The relation PROJ is horizontally fragmented in

$PROJ_1 = \sigma_{PNO \leq 'P3'}(PROJ)$
$PROJ_2 = \sigma_{PNO > 'P3'}(PROJ)$

Transform the following query into a reduced query on fragments:

SELECT BUDGET FROM PROJ, ASG
WHERE PROJ.PNO = ASG.PNO AND PNO = "P4".

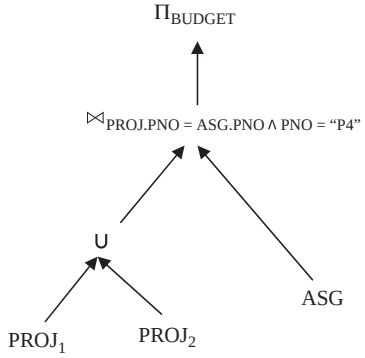
[WBUT (MCA) – 2004)

Answer:

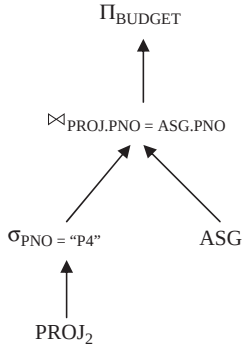
In relational algebraic format, the SQL query is represented as follows:

$\Pi_{BUDGET}(\sigma_{PROJ.PNO = ASG.PNO \wedge PNO = "P4"}((PROJ_1 \cup PROJ_2) \bowtie BRANCH))$

The canonical tree of the above SQL query is



The reduced query tree for the above query is listed in the following.



It has been assumed that the relation $PROJ_2$ contains more than one project numbers (PNO) which are greater than P3.

16. Consider the following relations:

EMP = (ENO, ENAME, TITLE)
ASG = (ENO, PNO, RESP, DUR)

(i) Find the names of employees who are managers of any project in SQL-like syntax and relational algebra.

(ii) 3Is the query optimized? If not then optimized it.

[WBUT (MCA) – 2003]

Answer:

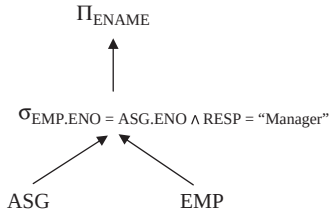
(i) By using SQL, the above query can be represented as:

SELECT ENAME FROM EMP, ASG WHERE RESP = "Manager"
AND EMP.ENO = ASG.ENO

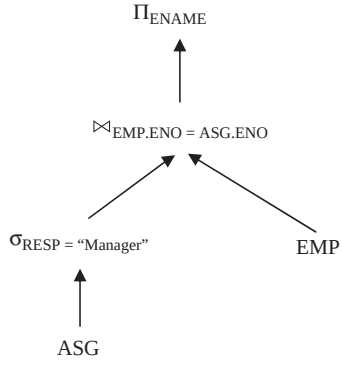
In relational algebraic format, the above query can be written as

$\Pi_{ENAME}(\sigma_{EMP.ENO = ASG.ENO \wedge RESP = "Manager"}(EMP \bowtie PROJ))$

The canonical tree for the query is as follows:



(ii) The optimized query tree is listed in the following.



17. Given three transactions below.

T1: Read (X)	T2: Write (X)	T3: Read (X)
Write (X)	Write(Y)	Read(Y)
Commit	Read (Z)	Read (Z)
	Commit	Commit

Draw complete schedule DAG and schedule DAG.

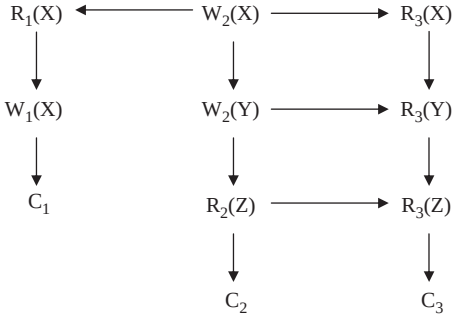
[WBUT (MCA) – 2004]

Answer:

A complete schedule for the above set of transactions can be written as the partial order of the transactions. Hence,

$\Sigma_1 = \{R_1(X), W_1(X), C_1\}$
$\Sigma_2 = \{W_2(X), W_2(Y), R_2(Z), C_2\}$
$\Sigma_3 = \{R_3(X), R_3(Y), R_3(Z), C_3\}$ and

The complete schedule DAG (Directed Acyclic Graph) for the transactions is as follows:



The schedule for the above set of transactions is

$S = \{W_2(X), W_2(Y), R_2(Z), C_2, R_1(X), W_1(X), C_1, R_3(X), R_3(Y), R_3(Z), C_3\}$

The above schedule is also serial since all the operations of T_2 are executed before all the operations of T_1 and all operations of T_1 are executed before all operations of T_3. Thus, the precedence relationship between transaction executions (schedule DAG) can be represented as

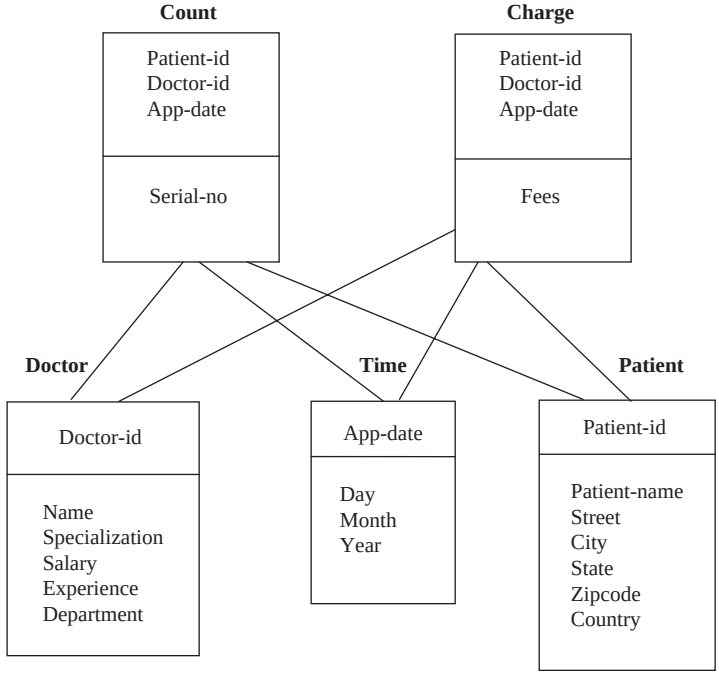
$T_2 \rightarrow T_1 \rightarrow T_3$

18. Suppose that a data warehouse consists of the three dimensions time, doctor and patient and two measures count and charge, where is the fee that a doctor charges a patient for a visit.
Draw a fact constellation schema for the above warehouse.

[C. U. M.Tech (CSE) – 2003]

Answer:

The fact constellation schema for the above data warehouse is as follows:



Bibliography

Aho A., Y. Sagiv, and J.D. Ullman, "Equivalence among relational Expressions", *SIAM Journal of computing*, Vol. 8, Issue 2, pp 218–246, 1979.

Alonso R. and H.F. Korth, "Database System Issues in Nomadic Computing", *Proceedings of ACM SIGMOD International Conference on Management of Data*, pp 388–392, May 1993.

American National Standard for Information System (ANSI), *Database Language SQL*, ANSI x3. 135, November 1992.

Apers P.M.G., A.R. Hevner, and S.B. Yao, "Optimization Algorithms for Distributed Queries", *IEEE Transactions on Software Engineering*, Vol. 9, Issue 1, pp 57–68, 1983.

Astrahan M.M., M.W. Blasgen, D.D. Chamberlin, K.P. Eswaran, J.N. Gray, P.P. Griffiths, W.F. King, R.A. Lorie, P.R. McJones, J.W. Mehl, G.R. Putzolu, I.L. Traiger, B.W. Wade, and V. Watson, "System R: A Relational Database Management System", *ACM Transactions on Database System*, Vol. 1, Issue 2, pp 97–137, June 1976.

Atkinson M., F. Bancilhon, D.J. Dewitt, K. Dittrich, D. Maier, and S. Zdonik, "The Object-Oriented Database System Manifesto", *Proceedings of 1st International Conference on Deductive and Object-Oriented Databases*, pp 40–57, December 1989.

Beeri C., P.A. Bernstein, and N. Goodman, "A Model for Concurrency in Nested Transaction Systems", *Journal of the ACM*, Vol. 36, Issue 2, pp 230–269, April 1989.

Bergsten B., M. Couprie, and P. Valduriez, "Prototyping DBS3, A Shared-Memory Parallel Database System", *Proceedings of International Conference on Parallel and Distributed Information Systems*, pp 226–234, December 1991.

Bernstein P. and B. Blaustein, "Fast Methods for Testing Quantified Relational Calculus Assertions", *Proceedings of ACM SIGMOD International Conference on Management of Data*, pp 39–50, June 1982.

Bernstein P.A. and N. Goodman, "Concurrency Control in Distributed Database Systems", *ACM Computing Surveys*, Vol. 13, Issue 2, pp 185–221, June 1981.

Bernstein P.A., V. Hadzilacos, and N. Goodman, "*Concurrency Control and Recovery in Database Systems*", Addision-Wesley, 1987.

Bernstein P.A., N. Goodman, E. Wong, C.L. Reeve, and J.B. Rothnie, "Query Processing in a System for Distributed Databases (SDD-1)", *ACM Transactions on Database Systems*, Vol. 6, Issue 4, pp 602–625, December 1981.

Bernstein P.A., David. W. Shipman, and J.B. Rothnie, "Concurrency Control in a System for Distributed Databases (SDD-1)", *ACM Transactions on Database Systems*, Vol. 5, Issue 1, pp 19–51, March 1980.

Berson A. and S.J. Smith, *Data Warehousing, Data Mining & OLAP*, New Delhi, Tata McGraw-Hill edition, ISBN 0–07–006272–2, 2004.

Bouganim L., D. Florescu, and P. Valduriez, "Dynamic Load Balancing in Hierarchical Parallel Database Systems", *Proceedings of 22nd International Conference on Very Large Databases*, pp 436–447, September 1996.

Bouganim L., D. Florescu, and P. Valduriez, "Multi-Join Query Execution with Skew in NUMA Multiprocessors", *Transaction on Distributed and Parallel Databases*, Vol. 7, Issue 1, 1999.

Bratbergsengen K., "Hashing Methods and Relational Algebra Operations", *Proceedings of 10th International Conference on Very Large Data Bases*, pp 323–333, August 1984.

Bright M., and A. Hurson, and S.H. Pakzad, "A Taxonomy and Current Issues in Multidatabase Systems", *Transactions on Computers*, Vol. 25, Issue 3, pp 50–60, March 1992.

Burkhardt J., H. Henn, S. Hepper, K. Rintdorff, and T. Schack, *Pervasive Computing – Technology and Architecture of Mobile Internet Applications*, New Delhi, Pearson Education, ISBN 81–297–0759–4, 2005.

Ceri S. and G. Pelagatti, *Distributed Databases – Principles & Systems*, New York, McGraw-Hill International Edition, Computer Science Series, 1985.

Ceri S., M. Negri, and G. Pelagatti, "Horizontal Data Parttioning in Database Design", *Proceedings of ACM SIGMOD International Conference on Management of Data*, pp 128–136, June 1982.

Ceri S. and S. Owicki, "On the Use of Optimistic Methods for Concurrency Control in Distributed Databases", *Proceedings of 6th Berkeley Workshop on Distributed Data Management and Computer Networks*, pp 117–130, February 1982.

Chamberlin D. et. al., "support for Repetitive Transaction and Adhoc Queries in System R", *ACM transaction of Distributed system*, Vol. 6, Issue 1, 1981.

Chiu D.M. and Y.C. Ho, "A Methodology for Interpreting Tree Queries into Optimal Semi-join Expressions", *Proceedings of ACM SIGMOD International Conference on Management of Data*, pp 169–178, May 1980.

Codd E., "Twelve Rules for On-line Analytical Processing", *Computer World*, April 1995.

ConNolly T. and C. Begg, *Database Systems – A Practical Approach to Design, Implementation and Management*, New Delhi, Pearson Education, 2003.

Copeland G., W. Alexander, E. Bougherty, and T. Keller, "Data Placement in Bubba", *Proceedings of ACM SIGMOD International Conference on Management of Data*, pp 99–108, May 1988.

Date C.J., *An Introduction to Database Systems*, Reading, MA, Addison-Wesley, Vol. 2, 1983.

Date C.J., *An Introduction to Database Systems,* 7th edition, Reading MA, Addison-werley, 2000.

Davidson S.B., H. Garcia-Molina, and D. Skeen, "Consistency in Partitioned Networks", *ACM Computing Surveys*, Vol. 17, Issue 3, pp 341–370, September 1984.

Dewitt D.J. and R. Gerber, "Multi Processor Hash-Based Join Algorithms", *Proceedings of 11th International Conference on Very Large Data Bases*, pp 151–164, August 1985.

Dewitt D.J. and J. Gray, "Parallel Database Systems: The Future of High Performance Database Systems", *ACM Communications*, Vol. 35, Issue 6, pp 85–98, June 1992.

Elmarsi R. and S.B. Navathe, *Fundamentals of Database Systems*, 2nd edition, Menlo Park, CA, Benjamin-Cummings, 1994.

Epstein R., M. Stonebraker, and E. Wong, "Query Processing in a Distributed Relational Database System", *Proceedings of ACM SIGMOD International Conference on Management of Data*, pp 169–180, May 1978.

Fayyad U., G. Piatetsky-Shapiro, P. Smyth, "From Data mining to knowledge Discovery in Data bases", *AI Magazine, American Association for Artificial Intelligence,* 0738-4602, pp 37–54, 1996.

Fernandez E.B., R.C. Summers, and C. Wood, *Database Security and Integrity*, Addision-Wesley, 1981.

Florentin J.J., "Consistency Auditing of Databases", *The Computer Journal*, Vol. 17, Issue 1, pp 52–58, 1974.

Forouzan B.A., *Data Communications and Net-working*, New Delhi, Tata McGraw-Hill Edition, ISBN 0–07–043563–4, 2000.

Garcia-Molina H. and K. Salem, "Sagas", *Proceedings of ACM SIGMOD International Conference on Management of Data*, pp 249–259, 1987.

Georgakopoulos D., M. Hornick, and A sheth, "An Overview of Workflow Management: From Process Modeling to Workflow Automation Infrastructure", *Journal of Distributed and Parallel Databases*, Vol. 3, pp 119–153, 1995.

Graefe G., "Encapsulation of Parallelism in the VolcaNo Query Processing Systems", *Proceedings of ACM SIGMOD International Conference on Management of Data*, pp 102–111, May 1990.

Gray J.N., "Notes on Database Operating Systems" in R. Bayer, R.M. Graham, and G. Seegmiiler (eds.), *Operating Systems: An Advanced Course*, Springer-Verlag, New York, pp 393–481, 1979.

Gray J.N., R.A. Lorie, G.R. Putzolu, and I.L. Traiger, "Granularity of Locks and Degrees of consistency in a shared Database", in G.N. Nijssen (ed.), Modelling in Database Management System, Amsterdam, North Holland, pp 365–394, 1976.

Gray J.N., P. McJones, M. Blasgen, B. Lindsay, R. Lorie, T. Price, F. Putzolu, and I. Traiger, "The Recovery Manager of the System R Database Manager", *ACM Computing Serveys*, Vol. 13, Issue 2, pp 223–242, June 1981.

Hammer M. and B. Niamir, "A Heuristic Approach to Attribute Partitioning", *Proceedings of ACM SIGMOD International Conference on Management of Data*, pp 93–101, May 1979.

Hammer M. and D.W. Shipman, "Reliability Mechanisms for SDD-1: A system for Distributed Databases", *ACM Transactions on Database Systems*, Vol. 5, Issue 4, pp 431–466, December 1980.

Heimbigner D. and D. McLeod, "A Federated Architecture for Information Management", *ACM Transactions on Information Systems*, Vol. 3, Issue 3, pp 253–278, July 1985.

Hevner A.R. and S.B. Yao, "Query Processing in Distributed Database Systems", *IEEE Transactions on Software Engineering*, Vol. 5, Issue 3, pp 177–182, March 1979.

Hoffer H.A. and D.G. Severance, "The Use of Cluster Analysis in Physical Database Design", *Proceedings of 1st International Conference on Very Large Data Bases*, pp 69–86, September 1975.

Hong W., "Exploiting Inter-Operation Parallelism in XPRS", *Proceedings of ACM SIGMOD International Conference of Management of Data*, pp 19–28, June 1992.

Inmon W.H., *Building the Data Warehouse*, New York, John Wiley and Sons, 1993.

Kitsuregawa M. and Y. Ogawa, "Bucket Spreading Parallel Hash: A New, Robust, Parallel Hash Join Method for Data Skew in the Super Database Computer", *Proceedings of 16th International Conference on Very Large Data Bases*, pp 210–221, August 1990.

Lanzelotte R., P. Valduriez, and M. Zait, "On the Effectiveness of Optimization Search Strategies for Parallel Execution Spaces", *Proceedings of 19th International Conference on Very Large Data Bases*, pp 493–504, September 1993.

Li V.O.K., "Performance Models of Timestamp-Ordering Concurrency Control Algorithms in Distributed Databases", *IEEE Transactions on Computers*, Vol. 36, Issue 9, pp 1041–1051, September 1987.

Lin W.K. and J. Notle, "Basic Timestamp, Multiple Version Timestamp, and Two-Phase Locking", *Proceedings of 9th International Conference on Very Large Data Bases*, pp 109–119, October–November 1983.

Lohman G. and L.F. Mackert, "R* Optimizer Validation and Performance Evaluation for Distributed Queries", *Proceedings of 11th International Conference on Very Large Data Bases*, pp 149–159, August 1986.

Mclean G. (Jr.), "Comments on SDD-1 Concurrency Control Mechanisms", *ACM Transactions on Database Systems*, Vol. 6, Issue 2 pp 347–350, June 1981.

Mohan C., B. Lindsay, and R. Obermarck, "Transaction Management in the R* Distributed Database Management System", *ACM Transactions on Database Systems*, Vol. 11, Issue 4, pp 378–396, December 1986.

Navathe S., S. Ceri, G. Wiederhold, and J. Dou, "Vertical Partitioning of Algorithms for Database Design", *ACM Transactions on Database Systems*, Vol. 9, Issue 4, pp 680–710, December 1984.

Obermarck R., "Distributed Deadlock Detection Algorithm", *ACM Transactions on Database Systems*, Vol. 7, Issue 2, pp 187–208, 1982.

Omiecinski E., "Performance Analysis of a Load Balancing Hash-Join Algorithm for a Shared-Memory Multiprocessor", *Proceedings of 17th International Conference on Very Large Data Bases*, pp 375–385, September 1991.

Pitoura E. and G. Samaras, *Data Management for Mobile Computing*, Norwell, Kluwer Academic, 1998.

Pujari A.K., *Data Mining Techniques*, Hyderabad India, Universities Press, ISBN 81–7371–380–4, 2001.

Ray C., S. Chattopadhyay, and S. Bhattacharya, "Modeling Constraints as Methods in Object Oriented Data Model", *International workshop on Engineering Automation for Software Intensive System integration*, Monterey, California, USA, June18–22, 2001 pp 220–227.

Ray C., S. Chattopadhyay, and S. Bhattacharya, "Representing Constraint Models in Object Oriented Data Using UML and ODL", *International Workshop on High Performance Computing in Asia Pacific Region*, Bangalore, India, December 16–19, 2002.

Ray C., S. Chattopadhyay, and M. Chattopadhyay, "Expressing Constraint Models in Object Oriented Data Using UML and ODL", *International Workshop on Distributed Computing*, Calcutta, India, December 28–31, 2002 Springer-Verlag LNCS#2571, pp 329–338.

Ray C., S. Chattopadhyay, and S. Bhattacharya, "An XML-Based Representation of Constraints in Object Oriented Information System", *International Workshop on Software Engineering Research & Practice*, Las Vegas, Nevada, USA, June 23–26, 2003.

Ray C., S. Chattopadhyay, and S. Bhattacharya, " Representation of Constraints in Object Oriented Data Using XML", *International Workshop on Advanced Computing (ADCOM 2003)*, Coimbatore, India, December 17–23, 2003.

Ray C., S. Chattopadhyay, and S. Bhattacharya, "Binding XML Data with Constraint Information into Java Object Model", *International Workshop on CCCT*, 2004, University of Texus, Austin, USA, August, 2004.

Ray C., S. Chattopadhyay, and S. Bhattacharya, "Interoperability of XML Data with Constraints into Java Object Model", *IASTED International Workshop on Advances in Computer Science and Technology*, ACST, 2004, St. Thomas US Virgin Islands, USA, November 22–24, 2004.

Ray C., S. Chattopadhyay, and S. Bhattacharya, "Transforming XML Data with Constraint Information into Java Object Model", *International Workshop on Advanced Computing (ADCOM 2004)*, Ahmedabad, India, December 2004, pp 751–758.

Ray C., "Modeling of Constraints in Distributed Object-Oriented Environment", *Accepted Doctoral Symposium of 1st International Conference on Interoperability of Enterprise Software and Applications (INTEROP-ESA, 2005)*, Geneva, Switzerland, February 23–25, 2005.

Ray C., S. Tripathi, A. Chatterjee, and A. Das, "An efficient Bi-directional String Matching Algorithm for Statistical Estimation", *International Symposium on Data, Information & Knowledge Spectrum (ISDIKS07)*, Kochi, India, December 2007, pp 73–79.

Rosenkrantz D.J. and H.B. Hunt, "Processing Conjunctive Predicates and Queries", *Proceedings of 6th International Conference on Very Large Data Bases*, pp 64–72, October, 1980.

Rothnie J.B., Jr. P.A. Bernstein, S. Fox, N. Goodman, M. Hammer, T.A. Landers, C. Reeve, D.W. Shipman, and E. Wong, "Introduction to a System for Distributed Databases (SDD-1)", *ACM Transactions on Database Systems*, Vol. 5, Issue 1, pp 1–17, March 1980.

Sacca D. and W. Wiederhold, "Database Partitioning in a Cluster of Processor", *ACM Transactions on Database Systems*, Vol. 10, Issue 1, pp 29–56, October 1985.

Sacco M.S. and S.B. Yao, "Query Optimization in Distributed Database Management Systems", in M.C. Yovits (ed.), *Advances in Computers*, New York Academic Press, Vol. 21, pp 225–273, 1982.

Selinger P.G. and M. Adiba, "Access Path Selection in Distributed Database Management Systems", *Proceedings of 1st International conference on Databases*, pp 204–215, 1980.

Sheth A.P. and J.A. Larson, "Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases", *ACM Computing Surveys*, Vol. 22, Issue 3, pp 183–236, September 1990.

Sheth A.P. and J.A. Larson, "Federated Databases: Architectures and Integration", *ACM Computing Surveys*, Special Issue on Heterogeneous Databases, September 1990.

Silberschatz A., H.F. Korth, and S. Sudarshan, *Database System Concepts*, New York, McGraw-Hill Publication, 1986.

Skeen D., "Non-blocking Commit Protocols", *Proceedings of ACM SIGMOD International Conference on Management of Data*, pp 133–142, April–May 1981.

Smith J.M., P.A. Bernstein, U. Dayal, N. Goodman, T. Landers, K.Lin, and E. Wong, "Multibase: Integrating Heterogeneous Distributed Database Systems", *Proceedings of National Computer Conference*, pp 487–499, May 1981.

Stonebraker M. and E. Neuhold, "A Distributed Database Version of INGRES", *Proceedings of 2nd Berkeley Workshop on Distributed Data Management and Computer Networks*, pp 9–36, May 1977.

Stonebraker M., P. Kreps, W. Wong, and G. Held, "The Design and Implementation of INGRES", *ACM Transactions on Database Systems*, Vol. 1 Issue 3, pp 198–222, September 1976.

Tanenbaum A.S., *Computer Networks*, 3rd edition, Engle wood cliffs NJ Prentice-Hall, 1997.

Tamer Ozsu M., P. Valduriez, and S. Sridhar, *Principles of Distributed Database Systems*, New Delhi, Pearson Education, 2006.

Williams R., D. Daniels, L. Haas, G. Lapis, B. Lindsay, P. Ng, R. Obermarck, P. Selinger, A. Walker, P. Wilms, and R. Yost, "R* : An Overview of the Architecture", *Proceedings of 2nd International Conference on Databases*, pp 1–28, June 1982.

Wong E. and K. Youssefi, "Decomposition: A Strategy for Query Processing", *ACM Transactions on Database Systems*, Vol. 1, Issue 3, pp 223–241, September 1976.

Yu C.T. and C.C. Chang, "Distributed Query Processing", *ACM Computing Surveys*, Vol. 16, Issue 4, pp 399–433, December 1984.

Zdonik S. and D. Maier (eds.), *Object-Oriented Database Systems*, San Mateo, Morgan Kaufmann, 1990.

Index