CONTENTS

Syllabus Distributed Database systemSemester I

Unit I: Distributed Database Concepts

Definition of Distributed databases and Distributed Database Management System (DDBMS), Distributed transparent system. DDBMS Architecture: DBMS standardization, Global, Local, External, and Internal Schemas, Architectural models for DDBMS. Distributed database design: Design problem of distributed systems, Design, strategies (top-down, bottom-up), Fragmentation, Allocation and replication of fragments. Query Processing Overview, Query Optimization.

Unit II: Transaction Processing in Distributed databases and Parallel databases

Transaction Management: Definition and examples, formalization of a transaction, ACID properties, classification of transaction. Concurrency Control: definition, execution schedules, examples, locking based algorithms, timestamp ordering algorithms, deadlock management. DBMS reliability: Definitions and Basic Concepts, Local Recovery Management, In-place update, out-of-place update, Distributed Reliability Protocols, Two phase commit protocol, Three phases commit protocol. Parallel Database System: Definition of Parallel Database Systems. Parallel query evaluation: Speed up and scale up, Query Parallelism: I/O Parallelism (Data Partitioning) Intra-query Parallelism, Inter –Query Parallelism, Intra Operation Parallelism, Inter Operation Parallelism.

Unit III: Object Oriented, Temporal and Spatial Databases:

Object Oriented Database: Object Identity, Object structure, Type Constructors, Encapsulation of Operations, Methods, Persistence, Type and Class Hierarchies, Inheritance, Complex Objects, Object-oriented DBMS , Languages and Design: ODMG Model, Object Definition Languages (ODL), Object Query Languages (OQL). Temporal and SpatialDatabase: Introduction to Temporal Database: Time ontology, structure, and granularity, Temporal data models, Temporal relational algebras. Introduction to Spatial Database: Definition, Types of spatial data, Geographical Information Systems (GIS), Conceptual Data Models for spatial databases, Logical data models for spatial databases: rastor and vector model. Physical data models for spatial databases: Clustering methods (space filling curves), Storage methods (R-tree). Query processing.

Unit IV: Deductive, Active, Multimedia and XML Databases

Deductive Database: Introduction to recursive queries, Datalog Notation, Clause Form and Horn Clauses, Interpretation of model: Least Model semantics, The fixed point operator, safe Datalog program, recursive query with negation. Active Database: Languages for rule specification: Events, Conditions, Actions. XML and Database: Structure of XML Data, XML Document Schema, Querying and Transformation, Storage of XML Data. Introduction to multimedia database systems.

Text book:

- Distributed Database; Principles & Systems By Publications, Stefano Ceri and Giuseppo Pelagatti,, McGraw-Hill International Editions (1984)
- Database Management Systems, 3rd edition, Raghu Ramakrishnan and
- Johannes Gehrke, McGraw-Hill (2002).
- Fundamentals of Database Systems, 6thEdition, Elmasri and Navathe, Addison. Wesley (2003).
- Unifying temporal data models via a conceptual model, C.S. Jensen, M.D. Soo, and R.T. Snodgrass: Information Systems, vol. 19, no. 7, pp. 513-547, 1994.
- Spatial Databases: A Tour by Shashi Shekhar and Sanjay Chawla, Prentice Hall, 2003 (ISBN 013-017480-7)
- Principles of Multimedia Database Systems, Subramanian V. S. Elsevier Publishers, 2013.

References:

- Principles of Distributed Database Systems; 2nd Editied By M. Tamer Ozsu and Patrick Valduriez, Person Education Asia.
- Database System Concepts, 5th edition, Avi Silberschatz , Henry F. Korth , S. Sudarshan: McGraw-Hill (2010)
- Database Systems: Concepts, Design and Applications, 2nd edition, Shio Kumar Singh, Pearson Publishing, (2011).
- Multi-dimensional aggregation for temporal data. M. Böhlen, J. Gamper, and C.S. Jensen. In Proc. of EDBT-2006, pp. 257-275, (2006).
- Moving objects databases (chapter 1 and 2), R.H. Güting and M. Schneider: Morgan Kaufmann Publishers, Inc., (2005)
- Advanced Database Systems, (chapter 5, 6, and 7), Zaniolo et al.: Morgan Kaufmann Publishers, Inc., (1997).

❖❖❖❖

Module - I

1

DISTRIBUTED DATABASE CONCEPTS

Unit Structure

1.0 OBJECTIVE:

After going through this unit, you will be able to:

- understand what Distributed database is.
- define what is Distributed Database Management System
- describe features of DDBMS its advantages and disadvantages
- Illustrate Distributed transparent system
- Classify Distributed transparent System.

1.1 INTRODUCTION:

For appropriate working of any business/organisation, there's a requirement for a well-organised database management system. In the past databases used to centralize in nature. But, with the growth of globalization, organisations lean towards expanded crosswise the world.

Because of this reason they have to choose distributed data instead of centralized system. This was the reason concept of Distributed Databases came in picture.

Distributed Database Management System is a software system that manages a distributed database which is partitioned and placed on different location. Its objective is to hide data distribution and appears as one logical database system to the clients.

1.2 DISTRIBUTED DATABASE CONCEPT:

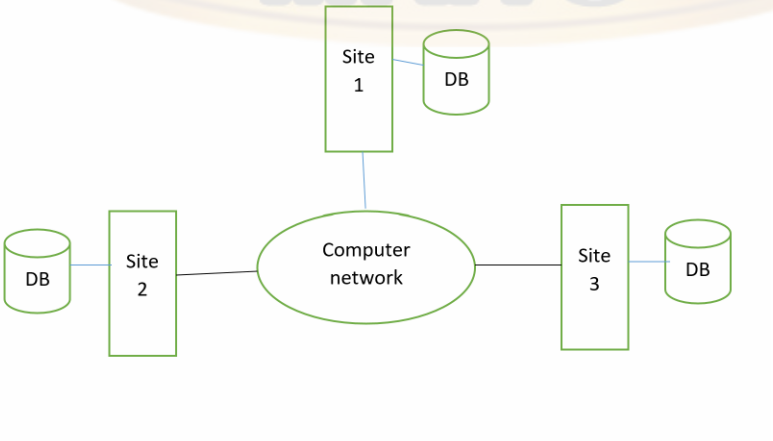
Distributed Database is database which is not restricted to one system only. It is a group of several interconnected databases. These are spread physically across various locations that communicate through a computer network. Distributed Database Management System (DDBMS) manages the distributed database and offers mechanisms so as to make the databases clear to the users. In these systems, data is intentionally distributed among multiple places so that all computing resources of the organization can be optimally used.

1.2.1. Definition of Distributed Databases and Distributed Database Management System (DDBMS)

The concept that is most important to the DDBMS is location clearness, meaning the user should be unaware of the actual location of data.

"A distributed database management system (DDBMS) can be defined as the software system that permits the management of the distributed database and makes the distribution transparent to the users.":- ***M. Tamer Özsu***

A Distributed Database Management System allows end users or application programmers to view a pool of physically detached databases as one logical unit. In another word, we can say distributed database is, where different data stored among multiple locations but connected via network, and for user it represent as a single logical unit.



Distributed Database Management System

1.2.1.1 Features of Distributed Database Management System

Some features of Distributed Database Management system are as follows:

- DDBMS software maintain CRUD (create, retrieve, Update, Delete) functions.
- It covers all application areas where huge volume of data are processed and retrieved simultaneously by n number of users.
- It ensure that data modified at any location update universally.
- It ensures confidentiality and data integrity which is important feature in transaction management.
- It can handle heterogeneous data platforms.

1.2.1.2 Advantages of Distributed Database Management System:

Some of the advantages of DDBMS are as follows:

- **Reliable:**

  Incase of centralized DBMS if database fails entire system comes to a halt whereas in DDBMS when a component fails may be reduce performance but it will not stop fully.

- **Easy Expansion**

  In centralized database system if system needs to be expanded, the implementation require extensive efforts and interruption in the existing functionality. However in DDBMS no disturbance in current functioning.

- **Faster Response**

  In centralized database all queries are passing through central data repository because of that response time is more although in DDBMS data is distributed in well-organized, so it runs faster response on queries.

1.2.1.3 Disadvantages of Distributed Database Management System:

- **Complex and Expensive**

  DDBMS provides data transparency and work on different sites so it may require complex and expensive software for proper working.

- **Overheads**

  Simple and complex operation and queries may require large communication and calculation. Responsiveness is largely dependent upon appropriate data distribution. Improper data distribution often leads to slow response to user requests.

- **Integrity**

  As data is on multiple sites it may create problem in updating data and maintaining data integrity.

1.2.2. Reasons to Boosting DDBMS

The following Reasons inspire moving towards DDBMS −

- **Distributed Nature of Structural Units** – Now a days most organizations are partitioned into several units that are physically scattered over the world. Each unit needs its own set of local data. Thus, the total database of the organization converts into distributed.
- **Data sharing Need** −The several organizational divisions often need to interact with each other and share data and resources. This demands common databases or simulated databases that should be used in a coordinated manner.
- **Provision for OLTP and OLAP**–Online Analytical Processing (OLAP) and Online Transaction Processing (OLTP) works on diversified systems. Distributed database systems supports and both OLAP and OLTP.
- **Database Retrieval** − One of the common methods used in DDBMS is imitation of data across different locations. Replication of data spontaneously helps in data recovery if database in any site is broken. Users can access data from other sites while the damaged site is being rebuilt. Thus, database disaster may convert inconspicuous to users.
- **Useful in Multiple Application Software** − Most organizations use a variant of application software and each is having different database support. DDBMS provides an identical functionality for using the same data among diversified platforms.

1.2.3 Databases Types:

1.2.3.1. Homogeneous Database:

In a homogeneous database, all diverse sites collect data identically. At all the sites same operating system, database management system and the data structures used is being used. Therefore, they are easy to manage.

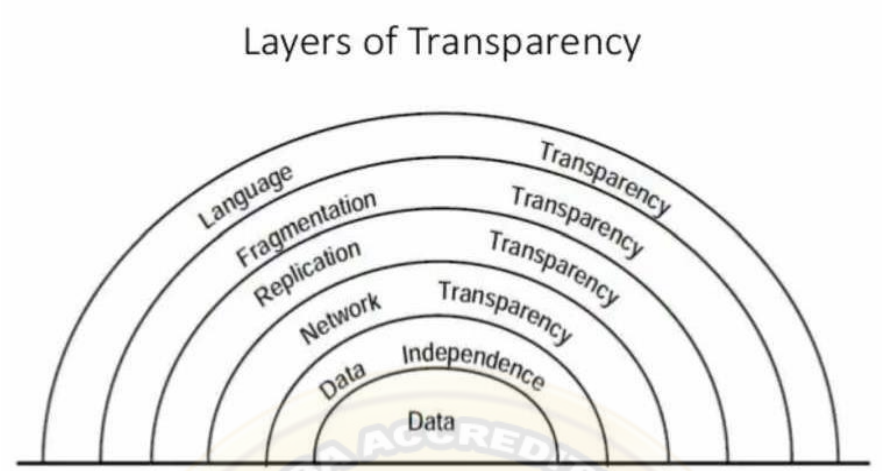
1.2.3.2 Heterogeneous Database:

In a heterogeneous distributed database, different sites can use dissimilar schema and software that can lead to glitches in transactions and query processing. Also, a particular site might be completely uninformed of the other sites. Diverse computers may use a differentoperating system, different database application. They possibly will even use changed data models for the database. Therefore, conversions arecompulsory for different sites to interconnect.

1.3 DISTRIBUTED TRANSPARENT SYSTEM:

One of the property of Distributed Database Management System is Distributed transparent system. Because of this feature internal details of

the distribution is hidden from the users. DDBMS hides all the distributed complexities and allow users to feel that they are working on single and centralized database.



Different Layers of transparencies

1.3.1 Levels of Distributed Transparent System:

DDBMS is supporting transparency at three levels:

1.3.1.1 Fragmentation Transparency

In Fragmentation transparency, fragments are created to store the data in distributed wayand should stay transparent. In this all the data administration work necessarily control by the system, not by the user. In this job, when a user sets a query, the global query is distributed in many sites to get data from fragments and this data is place together at the end to produce the result.

1.3.1.2 Location Transparency

Location transparency confirms that the user can fire query on any relation or fragment of a relation like they are stored locally on user's place. But the table or its fragments are kept at isolated site in the distributed database system, should be completely unaware to the user. The address and access mechanism of the remote site are completely hidden.

In order to integrate location transparency, DDBMS must have access to restructured and perfect data dictionary and DDBMS directory which contains the details of locations of data.

1.3.1.3 Replication Transparency

Replication transparency certifies that duplication of databases are concealed from the users. It permits users to query upon a relation as if only a single copy of the table is in place.

Replication transparency is connected with concurrency transparency and failure transparency. At any time a user updates a data element, the update is replicated in all the replicas of the table. Though,

this process should not be identified to the user. This is known as concurrency transparency.

In case of let-down of a site, the user can still progress with his queries using replicated copies without any information of failure then this is failure transparency.

1.4 SUMMARY

Distributed Database Management System (DDBMS) software which manages number of databases raised at different locations and connected with each other through a computer network. It offers mechanisms so that the delivery remains unaware to the users, who see the database as a single database. Its internal details hidden from users with transparency feature.

1.5 LIST OF REFERENCES AND BIBLIOGRAPHY AND FURTHER READING

- Principles of Distributed Database Systems; 2nd Editied By M. Tamer Ozsu and Patrick Valduriez, Person Education Asia.
- Distributed Database; Principles & Systems By Publications, Stefano Ceri and Giuseppo Pelagatti,, McGraw-Hill International Editions (1984)
- https://cs.uwaterloo.ca/~tozsu/publications/distdb/distdb.pdf
- https://www.tutorialspoint.com/distributed_dbms/index.htm
- https://www.geeksforgeeks.org/distributed-database-system/

1.6 MODEL QUESTIONS:

1. Explain Distributed Database Management System? Where we can use it instead of DBMS?
2. Write and explain problem areas of distributed data base system.
3. Write advantages and disadvantages of DDBMS.
4. What is Distributed Transparent System? Explain its types.
5. Explain reasons for advancement of DDBMS.
6. Write a short note on:
    - Fragmentation Transparency
    - Location Transparency
    - Replication Transparency

❖❖❖❖

2

DDBMS ARCHITECTURE

Unit Structure

2.0 OBJECTIVES

After going through this Chapter, you will be able to:

- understand Distributed database management system architecture
- define what is Global, Local, External, and Internal Schemas
- describe different architectural model for DDBM

2.1 INTRODUCTION

In any system architecture defines its structure. This means that the components of the system are identified, the purpose of each element is specified, and the interrelationships and interactions among these components are defined. The specification of the architecture of a system requires identification of the various units, with their connections and relationships, in terms of the data and control flow over the system.

2.2 DBMS STANDARDIZATION

Data standardization is the acute method of fetching data into a collective layout that allows for combined research, large-scale analytics, and sharing of refined tools and procedures

2.3 DDBMS ARCHITECTURE

Database systems comprise of complex data structures. Thus, to make the system efficient for retrieval of data and reduce the complexity of the users, developers use the method of Data Abstraction.

2.3.1. Factors for DDBMS Architecture:

DDBMS architectures are commonly developed dependent on three factors –

2.3.1.1. Distribution–Itstates the physical dispersal of data crosswise the different sites. Autonomy refers to the distribution of control, the distribution aspect of the classification deals with data. The user sees the data as one logical group. There are a numeralways DBMS have been distributed. We abstract these alternatives into two classes:

- client/server distribution
- peer-to-peer distribution (or full distribution).

2.3.1.2 Autonomy

Autonomy, in this perspective, refers to the distribution of mechanism, not of data. It identifies the distribution of regulator of the database system and the degree to which each component DBMS can work independently. Autonomy is a function of a quantity of factors such as whether the module systems interchange information, whether they can independently accomplish transactions, and whether one is certified to modify them. Requirements of an autonomous structure have been stated as follows:

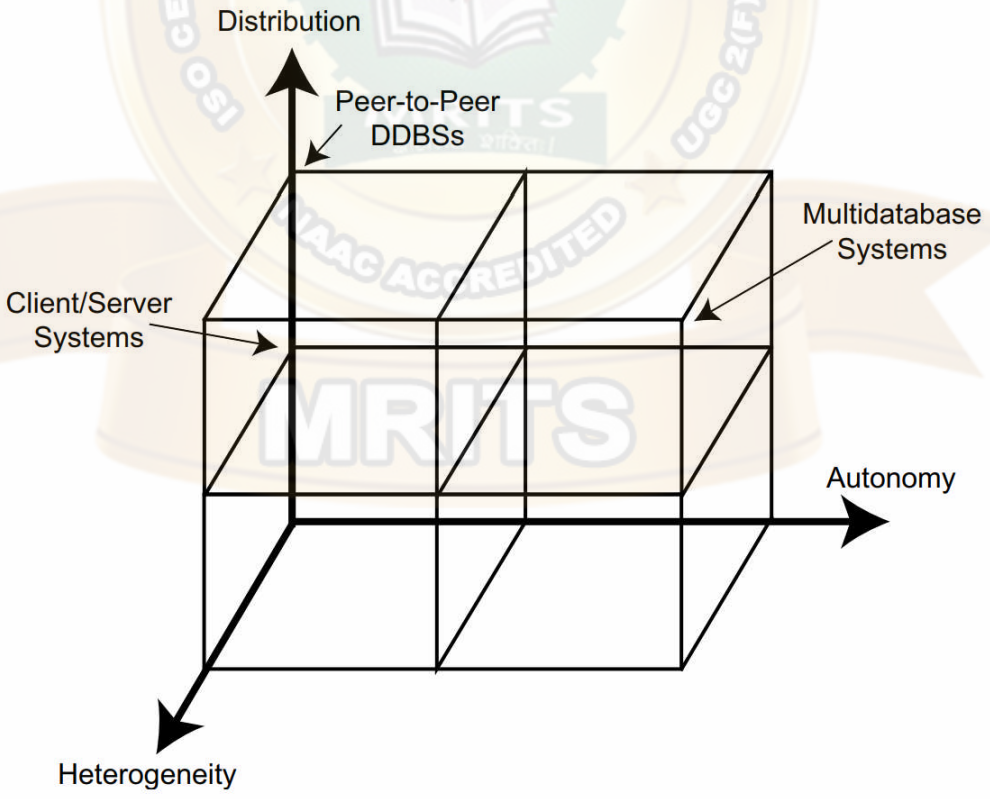
- The local procedures of the individual DBMSs are not affected by their involvement in the distributed system.
- The method in which the individual DBMSs develop queries and optimize them should not be affected by the accomplishment of global queries that access multiple databases.
- System regularity or operation should not be negotiated when individual DBMS join or leave the distributed system.

On the other hand, the proportions of autonomy can be stated as follows:

Design autonomy: Individual DBMS are permitted to use the data models and transaction management systems that they desire.

- **Communication autonomy:** To each of the discrete DBMS is free to make its own decision as to what type of information it wants to offer to the other DBMS or to the software that controls their global execution.
- **Execution autonomy:** Each DBMS can implement the transactions that are submitted to it in any way that it wants to.

2.3.1.3. Heterogeneity– It refers to the uniformity or variation of the data models, system tools and databases. Heterogeneity may happen in various forms in distributed systems, ranging from hardware heterogeneity and dissimilarities in networking protocols to distinctions in data managers. Representing data with different modelling tools creates heterogeneity because of the inherent expressive powers and limitations of individual data models. Heterogeneity in query languages not only involves the useof completely different data access paradigms in different data models (set-at-a-time access in relational systems versus record-at-a-time access in some object-oriented systems), but also covers differences in languages even when the individual systems use the same data model. Although SQLis now the standard relational query language, there are many different implementations and every vendor's language has a slightly different flavour.



DBMS Implementation Alternatives

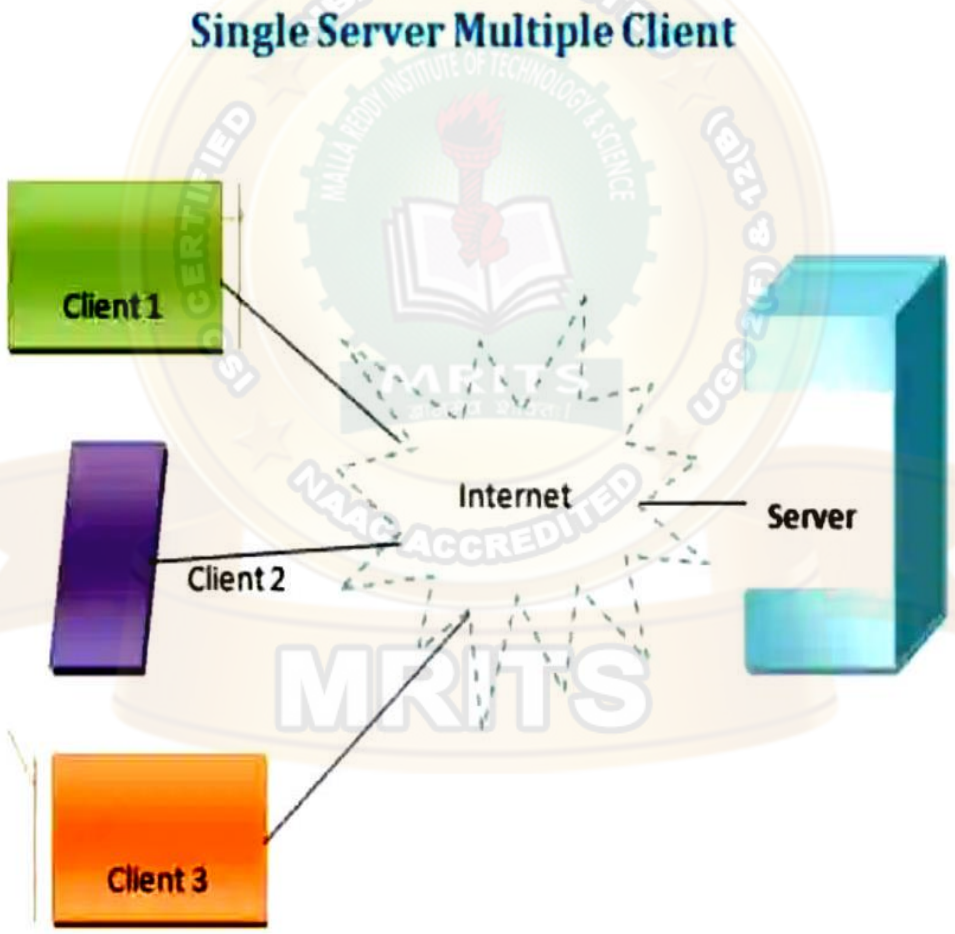
2.3 ARCHITECTURAL MODELS OF DISTRIBUTED DBMS:

2.4.1 Client-Server Architecture:

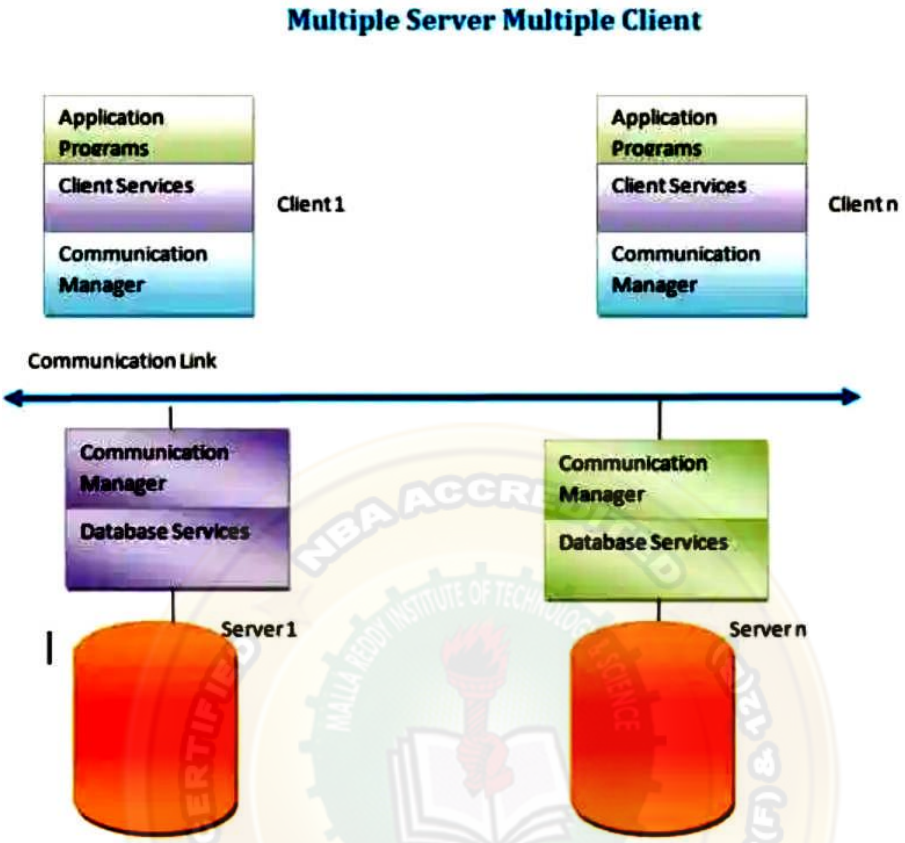
Client-Server architecture is a two-level architecture where the functionality is distributed into servers and clients. The server functions mainly comprise data management, query handling, transaction management and optimization. Client functions contain mainly user interface. Nevertheless, they have some functions resembling consistency checking and transaction management.

The two different types of clients – server architecture are as follows:

- **Single Server Multiple Client**



- **Multiple Server Multiple Client**:



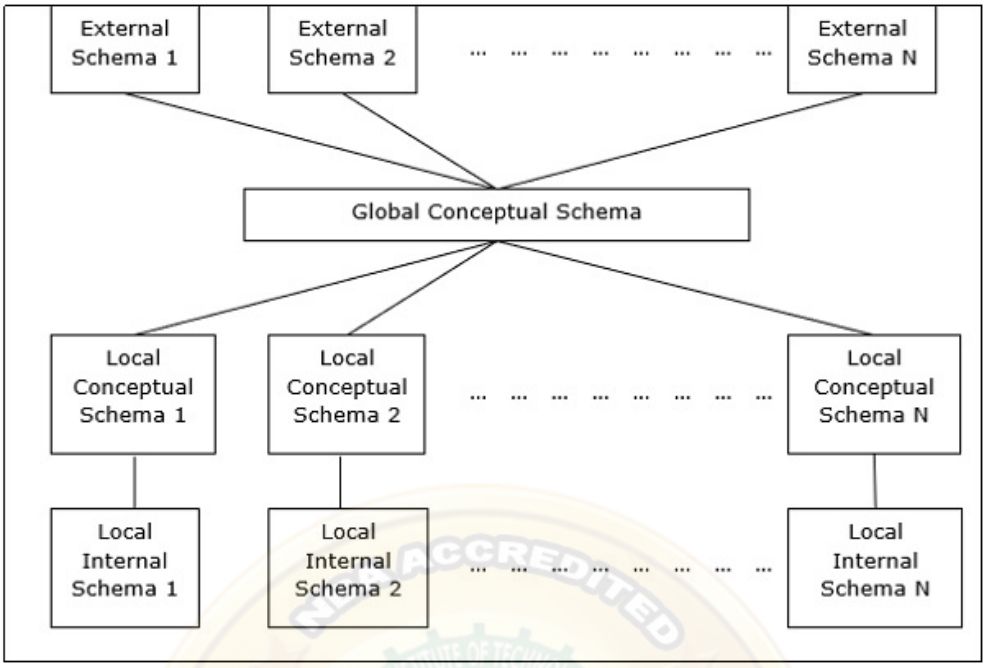
2.4.2 Peer- to-Peer Architecture for Distributed DBMS

In this systems, each peer actions both as a client and a server for instructing database services. The peers share their source with otherpeers and co-ordinate their actions.

This architecture in general has four levels of schemas −

2.4.2.1 Global, Local, External, and Internal Schemas:

- **Global Conceptual Schema** –Global Conceptual Schema represents the global logical view of data. It represents the logical explanation of entire database as if it is not circulated. This level encloses definitions of all units, relationships among entities and security and integrityfacts of whole databases kept at all sites in a distributed system.
- **Local Conceptual Schema** –Local Conceptual Schema Show logical data organization at individual location.
- **Local Internal Schema** –Local Internal Schema represents physical record at each site.
- **External Schema** –External Schema Describes user's vision of facts and figures.



2.4.3 Multi - DBMS Architectures

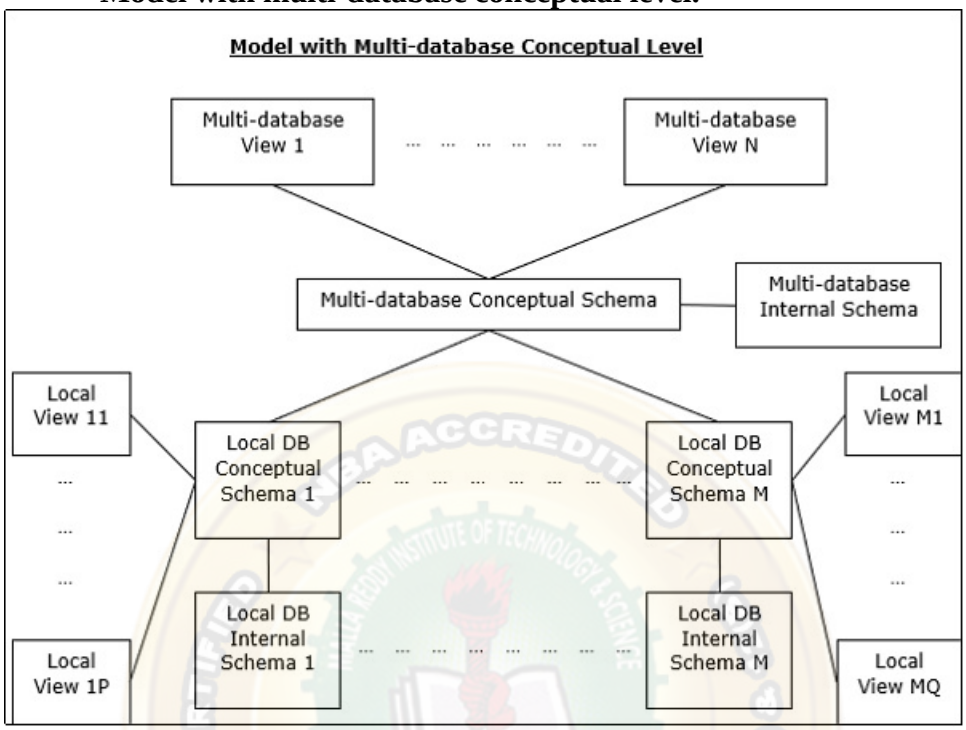
This is an integrated database system formed by a collection of two or more autonomous database systems.

Multi-DBMS can be expressed through six levels of schemas −

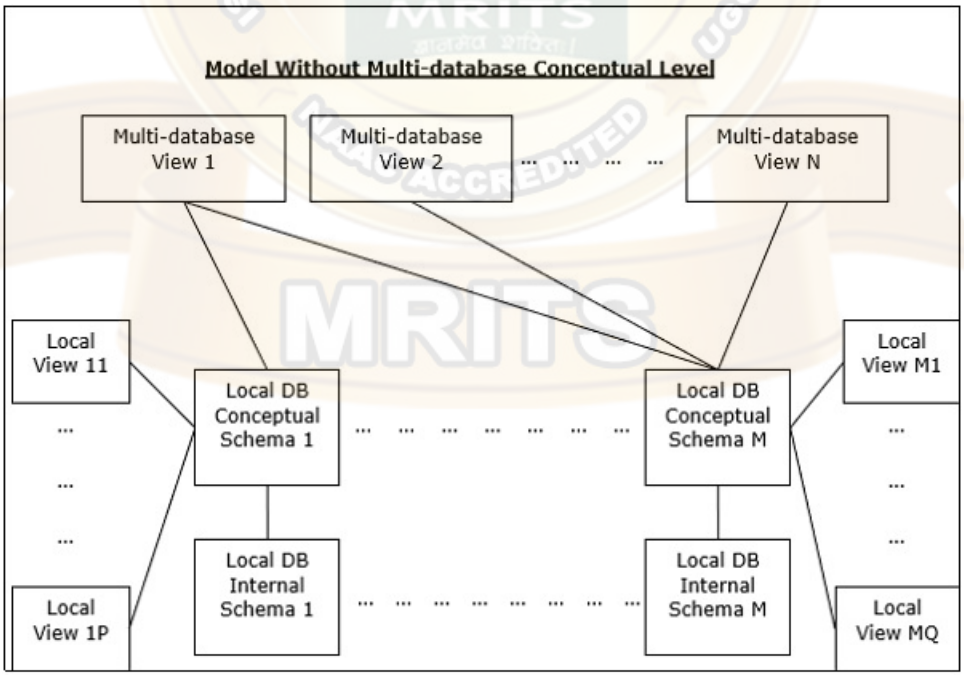
- **Multi-database View Level** − Describes multiple user views including of subsets of the integrated distributed database.
- **Multi-database Conceptual Level** − Shows integrated multi- database that comprises of global logical multi-database structure definitions.
- **Multi-database Internal Level** − Illustrates the data distribution across different sites and multi-database to local data mapping.
- **Local database View Level** − Give a picture of public view of local data.
- **Local database Conceptual Level** − Describes local data organization at each site.
- **Local database Internal Level** − Shows physical data organization at each site.

There are two design alternatives for multi-DBMS −

- **Model with multi-database conceptual level.**



- **Model without multi-database conceptual level.**



–

2.5 SUMMARY

There is different types of distributed databases. Distributed databases can be classified into homogeneous and heterogeneous databases having further divisions. Distributed architecture can beclassified in various types namely client – server, peer – to – peer and multi – DBMS.

2.5 LIST OF REFERENCES AND BIBLIOGRAPHY AND FURTHER READING

- https://www.csitweb.com/distributed-dbms-features-needs-and-architecture/
- https://www.ohdsi.org/data-standardization/
- Principles of Distributed Database Systems; 2nd Editied By M. Tamer Ozsu and Patrick Valduriez, Person Education Asia.

2.7 MODEL QUESTIONS:

1. What is Distributed Database Management System Architecture? Explain
2. Explain different architectural model for DDBMS
3. Explain Peer- to-Peer Architecture for Distributed DBMS. Write Short Notes on the following:

- Global Schema
- Local Schema
- External Schema
- Internal Schemas

❖❖❖❖

3

DISTRIBUTED DATABASE DESIGN

Unit Structure

3.0 OBJECTIVES

After going through this Chapter, you will be able to:

- understand Design of Distributed System
- Know Top-down and Bottom-up Strategies of Database Design
- describe Fragmentation and Allocation and replication of fragments
- gain knowledge about Query processing and Query Optimization

3.1 INTRODUCTION

The design of a distributed computer system contains making conclusions on the placement of data and programs through the sites of a computer network, as well as probably designing the network itself. In Distributed DBMS, the distribution of applications includes two things:

- Distribution of the distributed DBMS software
- Distribution of the application programs that run on it.

3.2 DESIGN PROBLEM OF DISTRIBUTED SYSTEMS

The distributed information system is defined as *"a number of interdependent computers linked by a network for sharing information among them"*. A distributed information system comprises of multiple independent computers that transfer or exchange information via a computer network.

- **Heterogeneity:**

Heterogeneity is functional to the network, computer hardware, operating system and execution of different developers. A crucial component of the heterogeneous distributed structure client-server environment is middleware. Middleware is a set of facilities that permits application and end-user to interrelate with each other across a heterogeneous distributed system.

- **Openness:**

The openness of the distributed system is determined mainly by the point to which new resource-sharing facilities can be made offered to the users. Open systems are considered by the fact that their key interfaces are circulated. It is based on a uniform communication tool and published interface for access to pooled resources. It can be built from varied hardware and software.

- **Scalability:**

Scalability of the system should persist efficient even with a important increase in the number of operators and resources coupled.

- **Security:**

Security of information system has three mechanisms confidentially, integrity and availability. Encryption defends shared resources, preserves delicate information secrets when communicated.

- **Failure Handling:**

When some errorsarise in hardware and the software suite, it may produce incorrect results or they may stop before they have completed the predicted computation so corrective techniques should to implement to handle this case. Failure control is challenging in distributed systems because the let-down is incomplete i.e. some components fail while others come to an end.

- **Concurrency:**

There is a chance that several users will attempt to access a common resource at the similar time. Multiple users create requests for thesame resources, i.e. read, write, and update. Each source must be safe in a parallel environment. Any item that signifies a shared resource a distributed system must confirm that it operates properly in a concurrent setting.

- **Transparency:**

Transparency confirms that the distributed system should be observed as a single object by the users or the application programmers somewhat than the pool of autonomous systems, which is work together. The user should be uninformed of where the services are situated and the transmitting from a local machine to anisolated one should be transparent.

3.3 DESIGN, STRATEGIES (TOP-DOWN, BOTTOM-UP)

It has been recommended that the group of distributed systems can be scrutinized along three scopes

1. Level of Sharing
2. Behaviour of access forms
3. Level of information on access pattern behaviour

To follow all extents some proper method has to be there to grow distributed database design. There are two methods for developing any database, the top-down method and the bottom-up method. Although these approaches appear completely different, they share the mutual goal of employing a system by relating all of the communication between the processes.

3.3.1 Top-down design Strategy

The top-down design structure starts from the common and transfers to the specific. In other words, you start with a universal idea of what is required for the system and then work your method down to the more specific particulars of how the system will work together. This process contains the identification of diverse entity types and the definition of each entity's characteristics.

3.3.2 Bottom – up design Strategy

The bottom-up approach begins with the specific details and moves up to the general. This is complete by first recognizing the data elements and then alliance them collected in data sets. In other words, this technique first identifies the aspects, and then groups them to form objects.

3.4 FRAGMENTATION

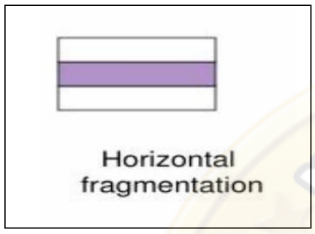
Data fragmentation is a procedure used to break up entities. The item might be a user's database, a system database, or a table. It permits you to breakdown a single object into two or more sectors, or fragments. Each fragment can be put in storage at any site over a computer network. In designing a scattered database, you must decide which portion of the database is to be put in storage where. One method used to break up the database into logical entities called fragments. Facts about data fragmentation is kept in the distributed data catalog(DDC), from which it is retrieved by the TP to process user requests. Fragmentation information is deposited in a distributed data catalogue which the dealing out computer uses to process a user's demand.

3.4.1 Data Fragmentation Strategies:

Data fragmentation strategies, are established at the table level and comprise of dividing a table into logical fragments. There are three forms of data fragmentation strategies: horizontal, vertical, and mixed.

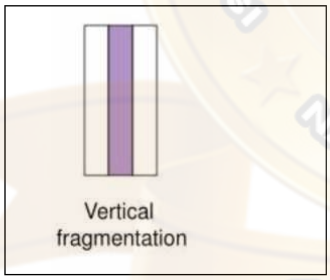
3.4.1.1 Horizontal fragmentation

This kind of fragmentation refers partition of a relation into fragments of rows. Each fragment is kept at a different workstation or node, and each fragment comprises unique rows. Each horizontal fragmentmay have a changed number of rows, but each fragment must have the identical attributes.



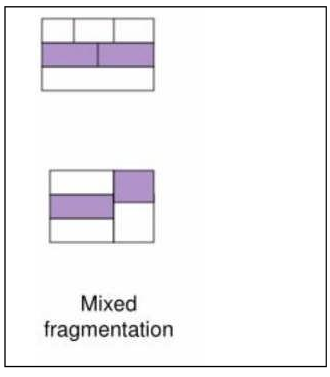
3.4.1.2 Vertical fragmentation

This type of fragmentation refers to the partition of a relation into fragments that contain a collection of attributes. Each vertical fragment must have the same number of rows, but can have dissimilar attributes depending on the key.



3.4.1.3 Mixed fragmentation

This type of fragmentation is a two-step procedure. First,horizontal fragmentation is completed to obtain the essential rows, then vertical fragmentation is done to distribute the attributes between the rows.



3.5 ALLOCATION AND REPLICATION OF FRAGMENTS

3.5.1 Data Allocation

Data allocation is a procedure of deciding where to accumulate the data. It also comprises a decision as to which data is stored at whatlocation. Data provision can be centralised, partitioned or replicated.

3.5.1.1. Centralised

The entire database is stored at one place. No distribution happens.

3.5.1.2 Partitioned

The database is distributed into several fragments that are deposited at numerous sites.

3.5.1.3 Replicated

Copies of one or added database fragments are kept at several sites.

3.5.2 Data Replication

Data replication is the storage of data replicas at numerous sites on the network. Fragment copies can be stored at several site, thus increasing data availability and reply time. Replicated data is subject to a common consistency rule. This rule involves that all replicas of the data fragments must be same and to ensure data consistency among all of the imitations.

Although data replication is favourable in terms of availability and response periods, the maintenance of the replications can turn into complex. For example, if data is simulated over multiple sites, the DDBMS needs to decide which copy to access. For a query process, the nearest copy is all that is necessary to satisfy a transaction. Though, if the operation is an update, at that time all copies must be selected and restructured to satisfy the common consistency rule.

A database can be moreover fully replicated, partially replicated or not replicated.

3.5.2.1 Full replication

Stores multiple copies of each database fragment at various sites. Fully replicated databases can be unlikely because of the amount of overhead forced on the system.

3.5.2.2 Partial replication

Stores multiple copies of some database fragments at multiplesites. Most DDBMS can hold this type of replication precise well.

3.5.2.3 No replication

Stores each database section at a single site. No repetition arises.

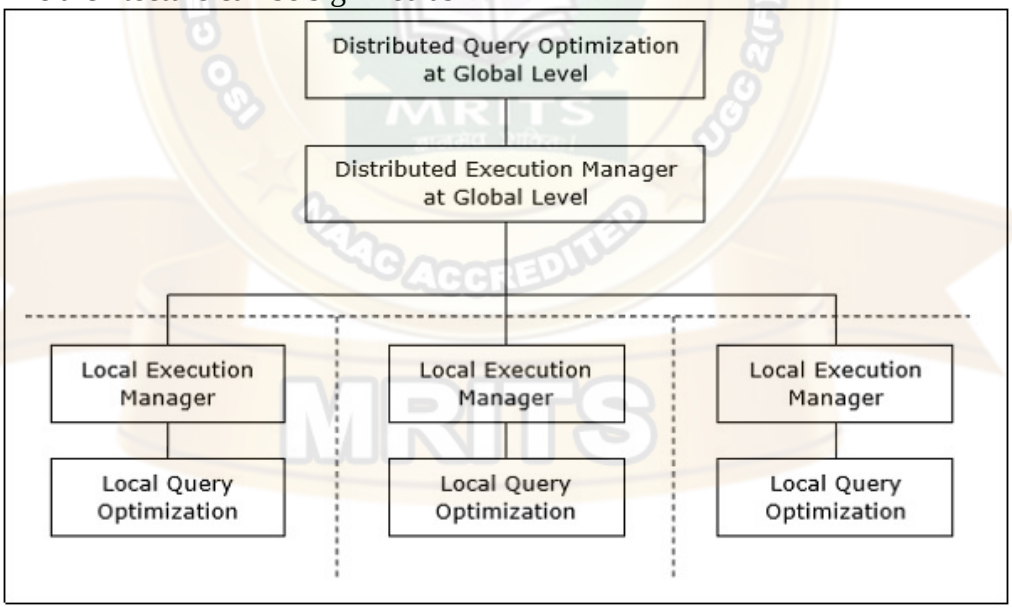
Data replication is mainly useful if usage frequency of remote data is great and the database is fairly huge. Another advantage of data replication is the opportunity of restoring lost data at a specific site.

3.6 QUERY PROCESSING OVERVIEW

A Query processing in a distributed database management system needs the transmission of data among the computers in a network. A distribution approach for a query is the ordering of data diffusions and local data processing in a database system. Usually, a query in Distributed DBMS entails data from multiple sites, and this need for data from different sites is termed the transmission of data that causes communication costs. Query processing in DBMS is unlike from query processing in centralized DBMS due to this communication cost of data transmission over the network. The transmission cost is small when sites are joined through high-speed Networks and is pretty significant in other networks.

In a distributed database system, handling a query comprises of optimization at both the world-wide and the local level. The query move in the database system at the client or supervisory site. Here, the user is legalised, the query is checked, translated, and enhanced at a global level.

The architecture can be signified as −



Mapping Global Queries into Local Queries

The procedure of mapping global queries to local ones can be recognised as follows −

- The tables essential in a global query have fragments distributed crosswise multiple sites. The local databases have data only about limited data. The supervisory site uses the global data dictionary to collect information about the distribution and recreates the global vision from the fragments.

- If there is no duplication, the global optimizer tracks local queries at the sites where the fragments are kept. If there is replication, the global optimizer selects the site based upon communication cost, workload, and server speed.

- The global optimizer produces a distributed execution proposal so that least amount of data allocation occurs across the sites. The plan shapes the location of the fragments, order in which query steps wishes to be executed and the processes involved in transferring transitional results.

- The local queries are optimized by the local database servers. Finally, the local query effects are merged together through blending operation in case of horizontal fragments and join process for vertical fragments.

3.7 QUERY OPTIMIZATION

Distributed query optimization needs evaluation of anenormous number of query trees each of which produce the necessary results of a query. This is primarily due to the occurrence of large volume of replicated and fragmented data. Hence, the goal is to find an optimal solution instead of the finest solution.

The main concerns for distributed query optimization are −

- Optimal consumption of resources in the distributed system.
- Query trading.
- Decrease of solution space of the query.

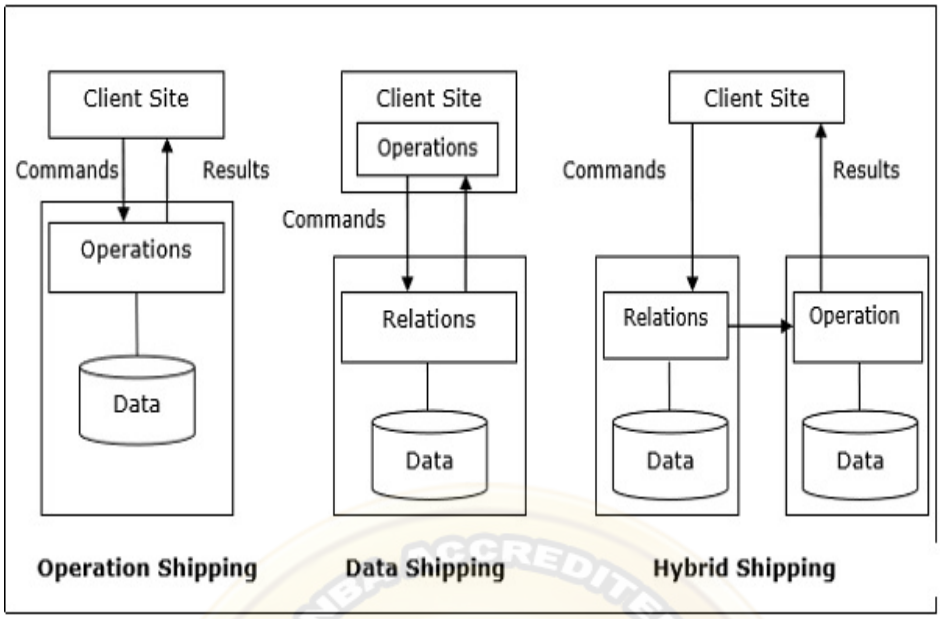
3.7.1 Optimal Utilization of Resources in the Distributed System

A distributed system has a number of database servers in thevarious sites to perform the actionsbelong to a query. Following are the approaches for optimal resource utilization −

- **Operation Shipping** − In operation shipping, the process is run at the location where the data is kept and not at the client site. The results are then transported to the client site. This is applicable for operations where the operands are presented at the same site. i.e. Select and Project operations.

- **Data Shipping** − In data shipping, the facts fragments are transported to the database server, where the processes are executed. This is used in procedures where the operands are distributed at diverse sites. This is also suitable in systems where the communication overheads are low, and local processors are abundant slower than the client server.

- **Hybrid Shipping** − This is a mixture of data and operation shipping. At this point, data fragments are transmitted to the high-speed processors, where the process runs. The results are then lead to the client site.



3.7.2 Query Trading

In query trading system for distributed database systems, the controlling/client site for a dispersed query is called the buyer and the locations where the local queries execute are entitled sellers. The buyer expresses a number of options for choosing sellers and for restructuring the global results. The goal of the buyer is to reach the optimal cost.

The algorithm jumps with the buyer allocating sub-queries to the vender sites. The best plan is created from local improved query plans proposed by the sellers joined with the communication cost for renovating the final result. Once the global optimum plan is framed, the query is performed.

3.7.3 Reduction of Solution Space of the Query

Optimal solution normally involves reduction of clarification space so that the cost of query and data relocation is reduced. This can be attained through a set of experimental rules, just as heuristics in centralized structures.

Some of the rules are as follows:

- Implement selection and projection tasks as early as promising. This eases the data flow over communication web.
- Streamline operations on horizontal fragments by removing selection conditions which are not applicable to a particular site.
- In case of join and union procedures comprising of fragments sited in multiple sites, transfer fragmented data to the site where utmost of the data is present and implement operation there.
- Use semi-join process to qualify tuples that are to be combined. This decreases the amount of data relocation which in turn reduces communication cost.
- Combine the common leaves and sub-trees in a dispersed query tree.

3.5 SUMMARY

The improvement in technology has opened the locks for unlimited volumes of data to transfer into the system. Distributed database technology is certainly one of the key growths in the field of database systems. Though, with the remarkable amount of data driving in from various sources and in many formats, it may become relatively a difficult task for a business to stock, process and manage this data. Choosing the services of a database expansion company that provides tradition database development solutions provider may support to meet the specific experiments of the business by keeping data well-organized, protected andeasily accessible for approved users.

3.6 LIST OF REFERENCES AND BIBLIOGRAPHY AND FURTHER READING

- https://www.dlsweb.rmit.edu.au/Toolbox/knowmang/content/distributed_sys/ddms_design.htm
- http://www.myreadingroom.co.in/notes-and-studymaterial/65-dbms/559-database-design-concepts.html
- https://www.geeksforgeeks.org/design-issues-of-distributed-system/

3.7 MODEL QUESTIONS

1. Explain Design problem of distributed systems.
2. What is Query Optimization? Explain Types.
3. Explain Query Processing. Differentiate Global Queries into Local Queries.
4. Explain Data Fragmentation Procedure.
5. Explain Design Problem of Distributed System.

❖❖❖❖

Module II

4

TRANSACTION PROCESSING IN DISTRIBUTED DATABASES AND PARALLEL DATABASES

Unit Structure

4.1.0 OBJECTIVES

In this chapter you will learn about:

- What four properties of transactions does a DBMS guarantee?
- Why does a DBMS interleave transactions?
- What is the correctness criterion for interleaved execution?
- What kinds of anomalies can interleaving transactions cause?
- How does a DBMS use locks to ensure correct interleaving?
- What is the impact of locking on performance?
- What SQL commands allow programmers to select transaction characteristics and reduce locking overhead?
- How does a DBMS guarantee transaction atomicity and recovery from system crashes?

4.1.1 INTRODUCTION

Often, a collection of several operations on the database appears to be a single unit from the point of view of the database user. For example, a transfer of funds from a checking account to a savings account is a single operation from the customer's standpoint; within the database system, however, it consists of several operations.

Clearly, it is essential that all these operations occur, or that, in case of a failure, none occur. It would be unacceptable if the checking accounts were debited but the savings account not credited. Collections of operations that form a single logical unit of work are called **transactions**. A database system must ensure proper execution of transactions despite failures—either the entire transaction executes, or none of it does. Furthermore, it must manage concurrent execution of transactions in away that avoids the introduction of inconsistency. In our funds-transfer example, a transaction computing the customer's total balance might see the checking-account balance before it is debited by the funds-transfer transaction, but see the savings balance after it is credited. As a result, it would obtain an incorrect result. In this chapter, we cover the concept of

the foundation for concurrent execution and recovery from system failure in a DBMS.

4.1.2 TRANSACTION MANAGEMENT

A transaction is defined as *anyone* of a user program in a DBMS and differs from an execution of a program outside the DBMS (e.g., a C program executing on Unix) in important ways. (Executing the same program several times generates several transactions.) For performance reasons interleave the actions of several transactions. However, to give users a simple way to understand the effect of running their programs, the interleaving is done carefully to ensure that the result of a concurrent execution of transactions is nonetheless equivalent (in its effect on the database) to some serial, or one-at-a-time, execution of the same set of transactions, How the DBMS handles concurrent executions is an important aspect of transaction management and the subject of *concurrency control.* A closely related issue is how the DBMS handles partial transactions, or transactions that are interrupted before they run to normal completion, The DBMS ensures that the changes made by such partial transactions are not seen by other transactions. How this is achieved is the subject of *crash recovery.*

4.1.3 DEFINITION AND EXAMPLES

A **transaction** is a **unit** of program execution that accesses and possibly updates various data items. Usually, a transaction is initiated by a user program written in a high-level data-manipulation language (typically SQL), or programming language (for example, C++, or Java), with embedded database accesses in JDBC or ODBC. A transaction is delimited by statements (or function calls) of the form **begin transaction** and **end transaction**. The transaction consists of all operations executed between the **begin transaction** and **end transaction**.

The data items in our simplified model contain a single data value (a number in our examples). Each data item is identified by a name(typically a single letter in our examples, that is, *A*, *B*, *C*, etc.). We shall illustrate the transaction concept using a simple bank applicationconsisting of several accounts and a set of transactions that access and update those accounts. Transactions access data using two operations:

• read(*X*), which transfers the data item *X* from the database to a variable, also called *X*, in a buffer in main memory belonging to the transaction that executed the read operation.

• write(*X*), which transfers the value in the variable *X* in the main-memory buffer of the transaction that executed the write to the data item *X* in the database.

It is important to know if a change to a data item appears only in main memory or if it has been written to the database on disk. In a real database system, the write operation does not necessarily result in the immediate update of the data on the disk; the write operation may be temporarily stored elsewhere and executed on the disk later. For now, however, we shall assume that the write operation updates the database immediately. Let *Ti* be a transaction that transfers $50 from account *A* to account *B*. This transaction can be defined as:

T_i: read(A);
$A := A - 50$;
write(A);
read(B);
$B := B + 50$;
write(B).

4.1.4 FORMALIZATION OF A TRANSACTION

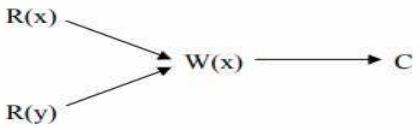
Characterization

- Data items that a given transaction
  - reads: Read Set (RS)
  - writes: Write Set (WS)
  - they are not necessarily mutually exclusive
  - Base Set (BS): BS = RS Ç WS
- Insertion and deletion are omitted, the discussion is restricted to static databases

- $O_{ij}(x)$: some atomic operation O_j of transaction T_i that operates on DB entity x
- $O_j \in$ {read, write}
- $OS_i = \cup_j O_{ij}$, i.e. all operations in T_i
- $N_i \in$ {abort, commit}, the termination condition for T_i
- Transaction T_i is a partial ordering over its operations and the termination condition
- Partial order $P = \{\Sigma, \prec\}$ where
  - Σ is the domain
  - $\prec$ is an irreflexive and transitive relation
- Transition T_i is a partial order $\{\Sigma_i, \prec_i\}$ where
  - $\Sigma_i = OS_i \cup N_i$
  - For any two operations $O_{ij}, O_{ik} \in OS_i$, if O_{ij}=R(x) and O_{ik}=W(x) for any data item x then either $O_{ij} \prec_i O_{ik}$ or $O_{ik} \prec_i O_{ij}$, i.e. ´there must be an order between conflicting operations´
  - $\forall O_{ij} \in OS_i, O_{ij} \prec_i N_i$, i.e. áll operations must precede the termination´
- The ordering relation $\prec_i$ is application dependent

- Example

  Read(x)
  Read(y)
  x = x + y
  Write(x)
  Commit



  - $\Sigma = \{R(x), R(y), W(x), C\}$
  - $\prec = \{(R(x), W(x)), (R(y), W(x)), (W(x), C), (R(x), C), (R(y), C)\}$ where (O_i, O_j) means $O_i \prec O_j$

- Partial order: the ordering is not specified for every pair of operations

2.1.5 ACID PROPERTIES

The concept of database transactions to recapitulate briefly, a transaction is an execution of a user program, seen by the DBMS as a series of read and write operations. A DBMS must ensure four important properties of transactions to maintain data in the face of concurrent access and system failures: Users should be able to regard the execution of each transaction as atomic: Either all actions are carried out or none are. Users should not have to worry about the effect of incomplete transactions (say, when a system crash occurs).

Each transaction, run by itself with no concurrent execution of other transactions, must preserve the consistency of the database The DBMS assumes that consistency holds for each transaction. Ensuring this property of a transaction is the responsibility of the user.

Users should be able to understand a transaction without considering the effect of other concurrently executing transactions, even if the DBMS interleaves the actions of several transactions for performance reasons. This property is sometimes referred to & isolation. Transactions are isolated, or protected, from the effects of concurrently scheduling othertransactions.

Once the DBMS informs the user that a transaction has been successfully completed, its effects should persist even if the system crashes before all its changes are reflected on disk. This property is called durability. The acronym ACID is sometimes used to refer to these four properties of transactions: atomicity, consistency, isolation and durability. We now consider how each of these properties is ensured in a DBMS.

Atomicity: Suppose that, just before the execution of transaction *Ti*, the values of accounts *A* and *B* are $1000 and $2000, respectively. Now suppose that, during the execution of transaction *Ti*, a failure occurs that prevents *Ti* from completing its execution successfully. Further, suppose that the failure happened after the write (*A*) operation but before

the write (*B*) operation. In this case, the values of accounts *A* and *B* reflected in the database are $950 and $2000. The system destroyed $50 asa result of this failure. In particular, we note that the sum $A + B$ is no longer preserved. Thus, because of the failure, the state of the system no longer reflects a real state of the world that the database is supposed to capture. We term such a state an **inconsistent state**. We must ensure that such inconsistencies are not visible in a database system. Note, however, that the system must at some point be in an inconsistent state. Even if transaction *Ti* is executed to completion, there exists a point at which the value of account *A* is $950 and the value of account *B* is $2000, which is clearly an inconsistent state. This state, however, is eventually replaced bythe consistent state where the value of account *A* is $950, and the value of account *B* is $2050. Thus, if the transaction never started or was guaranteed to complete, such an inconsistent state would not be visible except during the execution of the transaction. That is the reason for the atomicity requirement: If the atomicity property is present, all actions of the transaction are reflected in the database, or none are.

The basic idea behind ensuring atomicity is this: The database system keeps track (on disk) of the old values of any data on which a transaction performs a write. This information is written to a file called the *log*. If the transaction does not complete its execution, the database system restores the old values from the log to make it appear as though the transaction never executed. Ensuring atomicity is the responsibility of the database system; specifically, it is handled by a component of the database called the **recovery system**,

Consistency: The consistency requirement here is that the sum of *A* and *B* be unchanged by the execution of the transaction. Without the consistency requirement, money could be created or destroyed by the transaction! It can be verified easily that, if the database is consistent before an execution of the transaction, the database remains consistent after the execution of the transaction.

Ensuring consistency for an individual transaction is the responsibility of the application programmer who codes the transaction. This task may be facilitated by automatic testing of integrity constraints

Isolation: Even if the consistency and atomicity properties are ensured for each transaction, if several transactions are executed concurrently, their operations may interleave in some undesirable way, resulting in an inconsistent state.

For example, as we saw earlier, the database is temporarily inconsistent while the transaction to transfer funds from *A* to *B* is executing, with the deducted total written to *A* and the increased total yet to be written to *B*. If a second concurrently running transaction reads *A*and *B* at this intermediate point and computes *A*+*B*, it will observe an inconsistent value. Furthermore, if this second transaction then performs

updates on *A* and *B* based on the inconsistent values that it read, the database may be left in an inconsistent state even after both transactions have completed. A way to avoid the problem of concurrently executing transactions is to execute transactions serially—that is, one after the other. However, concurrent execution of transactions provides significant performance benefits. Other solutions have therefore been developed; they allow multiple transactions to execute concurrently. The isolation property of a transaction ensures that the concurrent execution of transactions results in a system state that is equivalent to a state that could have been obtained had these transactions executed one at a time in some order. Ensuring the isolation property is the responsibility of a component of the database system called the **concurrency-control system**

Durability: Once the execution of the transaction completes successfully, and the user who initiated the transaction has been notified that the transfer of funds has taken place, it must be the case that no system failure can result in a loss of data corresponding to this transfer of funds. The durability property guarantees that, once a transaction completes successfully, all the updates that it carried out on the database persist, evenif there is a system failure after the transaction completes execution. We assume for now that a failure of the computer system may result in loss of data in main memory, but data written to disk are never lost.

1. The updates carried out by the transaction have been written to disk before the transaction completes.

2. Information about the updates carried out by the transaction and written to disk is sufficient to enable the database to reconstruct the updates when the database system is restarted after the failure.

4.1.6 CLASSIFICATION OF TRANSACTION

A transaction is seen by the DBMS as a series, or list, of actions. The actions that can be executed by a transaction include reads and writes of database objects. To keep our notation simple, we assume that an object 0 is always read into a program variable that is also named O. It can therefore denote the action of a transaction T reading an object 0 asRT(O); similarly, we can denote writing as HTT(O). When the transactionT is clear from the context, we omit the subscript. In addition to reading and writing, each transaction must specify as its final action either commit(i.e., complete successfully) or abort (i.e., terminate and undo all theactions carried out thus far). Abort T denotes the action of T aborting, and Commit T denotes T committing. We make two important assumptions:

1. Transactions interact with each other only via database read and write operations; for example, they are not allowed to exchange messages.

2. A database is a filed collection of independent objects. When objects are added to or deleted from a database or there are relationships between

databases objects that we want to exploit for performance, some additional issues arise. If the first assumption is violated, the DBMS has no way to detect or prevent inconsistencies cause by such external interactions between transactions, and it is up to the writer of the application to ensure that the program is well-behaved. A schedule is a list of actions (reading, writing, aborting, or committing) from a set of transactions, and the order in which two actions of a transaction T appear in a schedule must be the same as the order in which they appear in T. Intuitively, a schedule represents an actual or potential execution sequence. For example, the schedule shows an execution order for actions of two transactions T1 and T2. It moves forward in time as we go down from one row to the next. It emphasize that a schedule describes the actions of transactions as seen by the DBMS. In addition to these actions, a transaction may carry out other actions, such as reading or writing from operating system files, evaluating arithmetic expressions, and so on; however, we assume that these actions do not affect other transactions; that is, the effect of a transaction on another transaction can be understood solely in terms of the common database objects that they read and write.

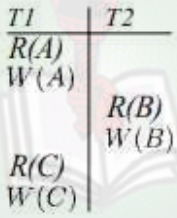


Figure 4.1.1 A Schedule involving Two Transactions

Note that the schedule in Figure 2.1.1 does not contain an abort or commit action for either transaction. A schedule that contains either an abort or a commit for each transaction whose actions are listed in it is called a complete schedule. A complete schedule must contain all the actions of every transaction that appears in it. If the actions of different transactions are not interleaved that is, transactions are executed from startto finish, one by one-we call the schedule a serial schedule.

CONCURRENT EXECUTION OF TRANSACTIONS

Now that we have introduced the concept of a schedule, we have a convenient way to describe interleaved executions of transactions. The DBMS interleaves the actions of different transactions to improve performance, but not all interleaving should be allowed. In this section, we consider what interleaving, or schedules, a DBMS should allow.

Motivation for Concurrent Execution

The schedule shown in Figure 2.1.1 represents an interleaved execution of the two transactions. Ensuring transaction isolation while permitting such concurrent execution is difficult and necessary for

performance reasons. First, while one transaction is waiting for a page to be read in from disk, the CPU can process another transaction. This is because I/O activity can be done in parallel with CPU activity in a computer. Overlapping I/O and CPU activity reduces the amount of time disks and processors are idle and increases system throughput (the averagenumber of transactions completed in a given time). Second, interleaved execution of a short transaction with a long transaction usually allows the short transaction to complete quickly. In serial execution, a short transaction could get stuck behind a long transaction, leading to unpredictable delays in response time, or average time taken to complete atransaction.

Serializability

A serializable schedule over a set S of committed transactions is a schedule whose effect on any consistent database instance is guaranteed to be identical to that of some complete serial schedule over S. That is, the database instance that results from executing the given schedule is identical to the database instance that results from executing the transactions in some serial order.

As an example, the schedule shown in Figure 2.1.2 is serializable. Even though the actions of T1 and T2 are interleaved, the result of this schedule is equivalent to running T1 (in its entirety) and then running T2. Intuitively, T1 's read and write of B is not influenced by T2's actions on A, and the net effect is the same if these actions are 'swapped' to obtain the serial schedule Tl; T2

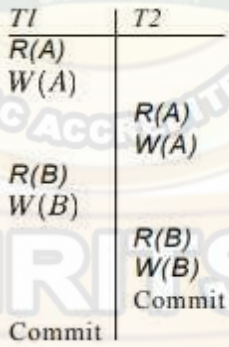


Fig: A Serializable Schedule

Executing transactions serially in different orders may produce different results, but all are presumed to be acceptable: the DBMS makes no guarantees about which of them will be the outcome of an interleaved execution. To see this, note that the two example transactions from Figure 2.1.2 can be interleaved as shown in Figure 2.13. This schedule, also serializable, is equivalent to the serial schedule T2; Tl. If T1 and T2 are submitted concurrently to a DBMS, either of these schedules (amongothers) could be chosen. The preceding definition of a serializable

schedule does not cover the case of schedules containing aborted transactions. We extend the definition of serializable schedules to cover aborted transactions in Section 16.3.4.

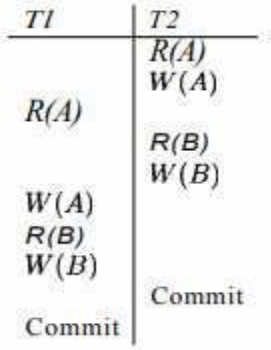


Fig:4.1.3 Another Serializable Schedule

Finally, we note that a DBMS might sometimes execute transactions in a way that is not equivalent to any serial execution; that is, using a schedule that is not serializable. This can happen for two reasons. First, the DBMS might use a concurrency control method that ensures the executed schedule, though not itself serializable, is equivalent to some serializable schedule. Second, SQL gives application programmers theability to instruct the DBMS to choose non-serializable schedules.

Transaction Characteristics in SQL

In order to give programmers control over the locking overhead incurred by their transactions, SQL allows them to specify three characteristics of a transaction: access mode, diagnostics size, and isolation level. The diagnostics size determines the number of error conditions that can be recorded; we will not discuss this feature further. If the access mode is READ ONLY, the transaction is not allowed to modifythe database. Thus, INSERT, DELETE, UPDATE, and CREATE commands cannot be executed. If we have to execute one of these commands, the access mode should be set to READ WRITE. for transactions with READ ONLY access mode only shared locks need to be obtained, thereby increasing concurrency.

The isolation level controls the extent to which a given transaction is exposed to the actions of other transactions executing concurrently. By choosing one of four possible isolation level settings, a user can obtain greater concurrency at the cost of increasing the transaction's exposure to other transactions' uncommitted changes. Isolation level choices are READ UNCOMMITTED, READ COMMITTED, REPEATABLE
READ, and SERIALIZABLE. The effect of these levels is summarized in Figure 2.14. In this context, dirty read and unrepeatable read are defined as usual

Level	Dirty Read	Unrepeatable Read	
READ UNCOMMITTED	Maybe	Maybe	Maybe
READ COMMITTED	No	Maybe	Maybe
REPEATABLE READ	No	No	Maybe
SERIALIZABLE	No	No	No

Fig: 4.1.4 Transaction Isolation Levels in SQL-92

The highest degree of isolation from the effects of othertransactions is achieved by setting the isolation level for a transaction T to SERIALIZABLE. This isolation level ensures that T reads only the changes made by committed transactions, that no value read or written by T is changed by any other transaction until T is complete, and that if T reads a set of values based on some search condition, this set is notchanged by other transactions until T is complete (i.e., T avoids the phantom phenomenon). In terms of a lock-based implementation, a SERIALIZABLE transaction obtains locks before reading or writing objects, including locks on sets of objects that it requires to be unchanged and holds them until the end, according to Strict 2PL. REPEATABLE READ ensures that T reads only the changes made by committed transactions and no value read or written by T is changed by any other transaction until T is complete. However, T could experience the phantom phenomenon; for example, while T examines all Sailors records withrating=1, another transaction might add a new such Sailors record, whichis missed by T. A REPEATABLE READ transaction sets the same locksas a SERIALIZABLE transaction, except that it does not do index locking;that is, it locks only individual objects, not sets of objects. READ COMMITTED ensures that T reads only the changes made by committed transactions, and that no value written by T is changed by any other transaction until T is complete. However, a value read by T may well be modified by another transaction while T is still in progress, and T is exposed to the phantom problem. A READ COMMITTED transaction obtains exclusive locks before writing objects and holds these locks until the end. It also obtains shared locks before reading objects, but these locks are released immediately; their only effect is to guarantee that the transaction that last modified the object is complete. (This guarantee relies on the fact that every SQL transaction obtains exclusive locks beforewriting objects and holds exclusive locks until the end.) A READ UNCOMMITTED transaction T can read changes made to an object by an ongoing transaction; obviously, the object can be changed further while T is in progress, and T is also vulnerable to the phantom problem. A READ UNCOMMITTED transaction does not obtain shared locks before reading objects. This mode represents the greatest exposure to uncommitted changes of other transactions; so much so that SQL prohibits such a transaction from making any changes itself-a READ UNCOMMITTED transaction is required to have an access mode of READ ONLY. Since such a transaction obtains no locks for reading objects and it is not allowed to write objects (and therefore never requests exclusive locks), it never makes any lock requests. The SERIALIZABLE isolation level is generally the safest and is recommended for most transactions. Some

transactions, however, can run with a lower isolation level, and the smaller number of locks requested can contribute to improved system performance. For example, a statistical query that finds the average sailor age can be run at the READ COMMITTED level or even the READ UNCOMMITTED level, because a few incorrect or missing values do not significantly affect the result if the number of sailors is large. The isolationlevel and access mode can be set using the SET TRANSACTION command. For example, the following command declares the current transaction to be SERIALIZABLE and READ ONLY: SET TRANSACTION ISOLATION LEVEL SERIALIZABLE READ ONLY
When a transaction is started, the default is SERIALIZABLE and READ WRITE.

Schedules Involving Aborted Transactions

Intuitively, all actions of aborted transactions are to be undone, and we can therefore imagine that they were never carried out to begin with. Using this intuition, we extend the definition of a serializable schedule as follows: A serializable schedule over a set S of transactions is a schedule whose effect on any consistent database instance is guaranteed to be identical to that of some complete serial schedule over the set of committed transactions in S. This definition of serializability relies on the actions of aborted transactions being undone completely, which may be impossible in some situations. For example, suppose that (1) an account transfer program T1 deducts $100 from account A, then (2) an interest deposit program T2 reads the current values of accounts A and B and adds6% interest to each, then commits, and then (3) T1 is aborted. The corresponding schedule is shown in Figure 2.1.5.

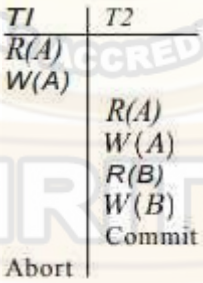


Fig:4.1.5 An Unrecoverable Schedule

Now, T2 has read a value for A that should never have been there. (Recall that aborted transactions' effects are not supposed to be visible to other transactions.) If T2 had not yet committed, we could deal with the situation by cascading the abort of TI and also aborting T2; this process recursively aborts any transaction that read data written by T2, and so on. But T2 has already committed, and so we cannot undo its actions. We say that such a schedule is unrecoverable. In a recoverable schedule, transactions commit only after (and if!) all transactions whose changesthey read commit. If transactions read only the changes of committed transactions, not only is the schedule recoverable, but also aborting a

transaction can be accomplished without cascading the abort to other transactions. Such a schedule is said to avoid cascading aborts. There is another potential problem in undoing the actions of a transaction. Suppose that a transaction T2 overwrites the value of an object A that has been modified by a transaction TI, while TI is still in progress, and Tl subsequently aborts. All of Tl's changes to database objects are undone by restoring the value of any object that it modified to the value of the object before Tl's changes. When Tl is aborted and its changes are undone in this manner, T2's changes are lost as well, even if T2 decides to commit. So, for example, if A originally had the value 5, then WetS changed by T1 to 6, and by T2 to 7, if T1 now aborts, the value of A becomes 5 again. Even if T2 commits, its change to A is inadvertently lost. A concurrency control technique called Strict 2PL.

4.2.1 CONCURRENCY CONTROL

The DBMS interleaves the actions of different transactions to improve performance, but not all interleaving should be allowed. In this section, we consider what interleaving, or schedules, a DBMS should allow.

4.2.2 DEFINITION

The database system must control the interaction among the concurrent transactions to prevent them from destroying the consistency of the database. It does so through a variety of mechanisms called concurrency-control schemes

4.2.3 EXECUTION SCHEDULES

Consider again the simplified banking system which has several account, and a set of transactions that access and update those accounts. Let *T*1 and *T*2 be two transactions that transfer funds from one account to another. Transaction *T*1 transfers $50 from account *A* to account *B*. It is defined as:

T_1: read(A);
$A := A - 50$;
write(A);
read(B);
$B := B + 50$;
write(B).

Transaction *T*2 transfers 10 percent of the balance from account *A* to account *B*. It is defined as:

```
T2: read(A);
    temp := A * 0.1;
    A := A − temp;
    write(A);
    read(B);
    B := B + temp;
    write(B).
```

Suppose the current values of accounts *A* and *B* are $1000 and $2000, respectively. Suppose also that the two transactions are executed one at a time in the order *T*1 followed by *T*2. This execution sequence appears in above Figure. In the figure, the sequence of instruction steps is in chronological order from top to bottom, with instructions of *T*1 appearing in the left column and instructions of *T*2 appearing in the right column. The final values of accounts *A* and *B*, after the execution in Figure 2.2.1 takes place, are $855 and $2145, respectively. Thus, the total amount of money in accounts *A* and *B*—that is, the sum *A* + *B*—is preserved after the execution of both transactions.

T_1	T_2
read(A)	
$A := A - 50$	
write(A)	
read(B)	
$B := B + 50$	
write(B)	
commit	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
	$B := B + temp$
	write(B)
	commit

Fig: 4.2.1 Schedule 1-a serial schedule in which T1 is followed by T2

Similarly, if the transactions are executed one at a time in the order *T*2 followed by *T*1, then the corresponding execution sequence is that of Figure 2.1.2. Again, as expected, the sum *A* + *B* is preserved, and the final values of accounts *A* and *B* are $850 and $2150, respectively.

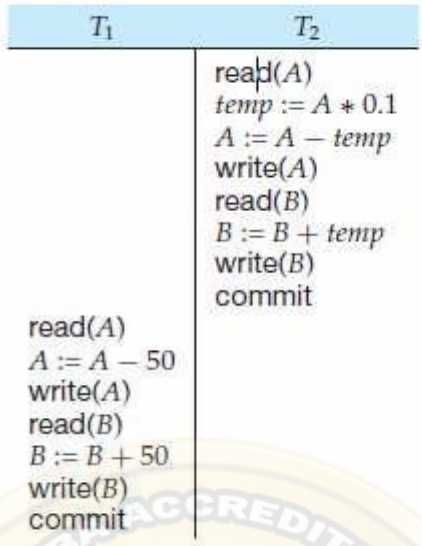


Fig: 4.1.2 Schedule 2-a serial schedule in which T2 is followed by T1

The execution sequences just described are called **schedules**. They represent the chronological order in which instructions are executed in the system. Clearly, a schedule for a set of transactions must consist of all instructions of those transactions, and must preserve the order in which the instructions appear in each individual transaction. For example, in transaction *T*1, the instruction write(*A*) must appear before the instruction read(*B*), in any valid schedule. Note that we include in our schedules the **commit** operation to indicate that the transaction has entered the committed state. In the following discussion, we shall refer to the first execution sequence (*T*1 followed by *T*2) as schedule 1, and to the second execution sequence (*T*2 followed by *T*1) as schedule 2. These schedulesare **serial**: Each serial schedule consists of a sequence of instructions fromvarious transactions, where the instructions belonging to one single transaction appear together in that schedule. Recalling a well-known formula from combinations, we note that, for a set of *n* transactions, there exist *n* factorial (*n*!) different valid serial schedules.

When the database system executes several transactions concurrently, the corresponding schedule no longer needs to be serial. If two transactions are running concurrently, the operating system may execute one transaction for a little while, then perform a context switch, execute the second transaction for some time, and then switch back to the first transaction for some time, and so on.

With multiple transactions, the CPU time is shared among all the transactions. Several execution sequences are possible, since the various instructions from both transactions may now be interleaved. In general, it is not possible to predict exactly how many instructions of a transaction will be executed before the CPU switches to another transaction.

Returning to our previous example, suppose that the two transactions are executed concurrently. One possible schedule appears in Figure 2.2.3. After this execution takes place, we arrive at the same state as the one in which the transactions are executed serially in the order *T*1 followed by *T*2. The sum *A* + *B* is indeed preserved. Not all concurrent executions result in a correct state. To illustrate, consider the schedule of Figure 2.2.4. After the execution of this schedule, we arrive at a state where the final values of accounts *A* and *B* are $950 and $2100, respectively. This final state is an *inconsistent state*, since we have gained
$50 in the process of the concurrent execution. Indeed, the sum *A* + *B* is not preserved by the execution of the two transactions.

If control of concurrent execution is left entirely to the operating system, many possible schedules, including ones that leave the database in an inconsistent state, such as the one just described, are possible. It is the job of the database system to ensure that any schedule that is executed will leave the database in a consistent state. The **concurrency-control** component of the database system carries out this task.

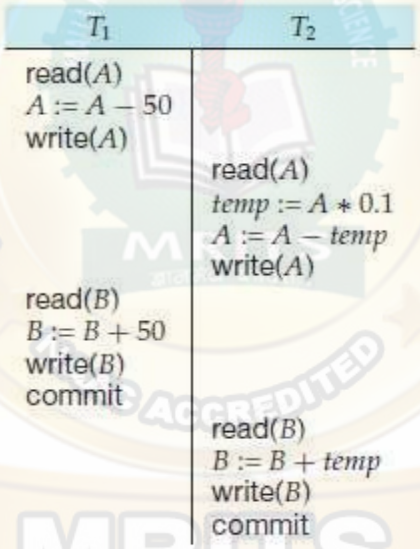


Fig: 4.2.3 Schedule 3-a concurrent schedule equivalent to schedule 1 We

can ensure consistency of the database under concurrent execution by making sure that any schedule that is executed has the same effect as a schedule that could have occurred without any concurrent execution. That is, the schedule should, in some sense, be equivalent to a serial schedule. Such schedules are called **serializable** schedules.

T_1	T_2
read(A)	
$A := A - 50$	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
write(A)	
read(B)	
$B := B + 50$	
write(B)	
commit	
	$B := B + temp$
	write(B)
	commit

Fig: 4.2.4 Schedule 4-a concurrent schedule resulting in an inconsistent state

4.2.4 LOCKING BASED ALGORITHMS

Lock-Based Protocols

One way to ensure isolation is to require that data items be accessed in a mutually exclusive manner; that is, while one transaction is accessing a data item, no other transaction can modify that data item. The most common method used to implement this requirement is to allow a transaction to access a data item only if it is currently holding a lock on that item.

Locks

There are various modes in which a data item may be locked. In this section, we restrict our attention to two modes:

1. Shared:- If a transaction *Ti* has obtained a shared-mode lock (denoted by S) on item *Q*, then *Ti* can read, but cannot write, *Q*.

2. Exclusive:- If a transaction *Ti* has obtained an exclusive-mode lock (denoted by X) on item *Q*, then *Ti* can both read and write *Q*.

	S	X
S	true	false
X	false	false

Fig: 2.2.5 Lock-compatibility matix comp

We require that every transaction request a lock in an appropriate mode on data item *Q*, depending on the types of operations that it will perform on *Q*. The transaction makes the request to the concurrency- control manager. The transaction can proceed with the operation only after the concurrency-control manager grants the lock to the transaction. Theuse of these two lock modes allows multiple transactions to read a data item but limits write access to just one transaction at a time.

To state this more generally, given a set of lock modes, we can define a compatibility function on them as follows: Let *A* and *B* represent arbitrary lock modes. Suppose that a transaction *Ti* requests a lock of mode *A* on item *Q* on which transaction *Tj* (*Ti* _= *Tj*) currently holds a lock of mode *B.* If transaction *Ti* can be granted a lock on *Q* immediately, in spite of the presence of the mode *B* lock, then we say mode *A* is compatible with mode *B.* Such a function can be represented conveniently by a matrix. The compatibility relation between the two modes of locking discussed in this section appears in the matrix comp of Figure 2.2.5. An element comp(*A, B*) of the matrix has the value *true* if and only if mode *A* is compatible with mode *B.*

Note that shared mode is compatible with shared mode, but not with exclusive mode. At any time, several shared-mode locks can be held simultaneously (by different transactions) on a particular data item. A subsequent exclusive-mode lock request has to wait until the currently held shared-mode locks are released.

A transaction requests a shared lock on data item *Q* by executing the lock- S(*Q*) instruction. Similarly, a transaction requests an exclusive lock through the lock-X(*Q*) instruction. A transaction can unlock a data item *Q* by the unlock(*Q*) instruction.

To access a data item, transaction *Ti* must first lock that item. If the data item is already locked by another transaction in an incompatible mode, the concurrency control manager will not grant the lock until all incompatible locks held by other transactions have been released. Thus, *Ti* is made to wait until all incompatible locks held by other transactions have been released.

Transaction *Ti* may unlock a data item that it had locked at some earlier point. Note that a transaction must hold a lock on a data item as long as it accesses that item. Moreover, it is not necessarily desirable for a transaction to unlock a data item immediately after its final access of that data item, since serializability may not be ensured.

As an illustration, consider again the banking example. Let *A* and *B* be two accounts that are accessed by transactions *T*1 and *T*2. Transaction *T*1 transfers $50 from account *B* to account *A* (Figure 2.2.6).

```
T1: lock-X(B);
    read(B);
    B := B − 50;
    write(B);
    unlock(B);
    lock-X(A);
    read(A);
    A := A + 50;
    write(A);
    unlock(A).
```

Fig: 4.2.6 Transaction T1

Transaction *T*2 displays the total amount of money in accounts *A* and *B*—that is, the sum *A* + *B* (Figure 2.2.7). Suppose that the values of accounts *A* and *B* are $100 and $200, respectively.

If these two transactions are executed serially, either in the order *T*1, *T*2 or the order *T*2, *T*1, then transaction *T*2 will display the value $300.If, however, these transactions are executed concurrently, then schedule 1, in Figure 2.2.8, is possible. In this case, transaction *T*2 displays $250, which is incorrect. The reason for this mistake is that the transaction *T*1 unlocked data item *B* too early, as a result of which *T*2 saw an inconsistentstate. The schedule shows the actions executed by the transactions, as wellas the points at which the concurrency-control manager grants the locks. The transaction making a lock request cannot execute its next action until the concurrency control manager grants the lock. Hence, the lock must be granted in the interval of time between the lock-request operation and the following action of the transaction.

Exactly when within this interval the lock is granted is not important; we can safely assume that the lock is granted just before the following action of the transaction. We let you infer when locks aregranted.

```
T2: lock-S(A);
    read(A);
    unlock(A);
    lock-S(B);
    read(B);
    unlock(B);
    display(A + B).
```

Fig:4.2.7 Transaction T2

T_1	T_2	concurreny-control manager
lock-X(B)		
		grant-X(B, T_1)
read(B)		
$B := B - 50$		
write(B)		
unlock(B)		
	lock-S(A)	
		grant-S(A, T_2)
	read(A)	
	unlock(A)	
	lock-S(B)	
		grant-S(B, T_2)
	read(B)	
	unlock(B)	
	display($A + B$)	
lock-X(A)		
		grant-X(A, T_1)
read(A)		
$A := A - 50$		
write(A)		
unlock(A)		

Fig: 4.2.8 Schedule 1

Suppose now that unlocking is delayed to the end of the transaction. Transaction *T*3 corresponds to *T*1 with unlocking delayed(Figure 2.2.9). Transaction *T*4 corresponds to *T*2 with unlocking delayed (Figure 2.2.10). You should verify that the sequence of reads and writes inschedule 1, which lead to an incorrect total of $250 being displayed, is no longer possible with *T*3 and *T*4.

```
T3: lock-X(B);
    read(B);
    B := B − 50;
    write(B);
    lock-X(A);
    read(A);
    A := A + 50;
    write(A);
    unlock(B);
    unlock(A).
```

Fig: 4.2.9 Transaction T3(transaction T1 with unlocking delayed)

```
T4: lock-S(A);
    read(A);
    lock-S(B);
    read(B);
    display(A + B);
    unlock(A);
    unlock(B).
```

Fig: 4.2.10 Transaction T4(transaction T2 with unlocking delayed)

Other schedules are possible. *T*4 will not print out an inconsistent result in any of them; Unfortunately, locking can lead to an undesirable situation. Consider the partial schedule of Figure 2.2.11 for *T*3 and *T*4. Since *T*3 is holding an exclusive emode lock on *B* and *T*4 is requesting a shared-mode lock on *B*, *T*4 is waiting for *T*3 to unlock *B*. Similarly, since *T*4 is holding a shared-mode lock on *A* and *T*3 is requesting an exclusive- mode lock on *A*, *T*3 is waiting for *T*4 to unlock *A*. Thus,we have arrived at a state where neither of these transactions can ever proceed with its normal execution. This situation is called deadlock. When deadlock occurs, the system must roll back one of the two transactions. Once a transaction has been rolled back, the data items that were locked by that transaction are unlocked. These data items are then available to the other transaction, which can continue with its execution.

If we do not use locking, or if we unlock data items too soon after reading or writing them, we may get inconsistent states. On the other hand, if we do not unlock a data item before requesting a lock on another data item, deadlocks may occur. There are ways to avoid deadlock in some situations. However, in general, deadlocks are a necessary evil associated with locking, if we want to avoid inconsistent states. Deadlocks are definitely whereas inconsistent states may lead to real-world problems that cannot be handled by the database system.

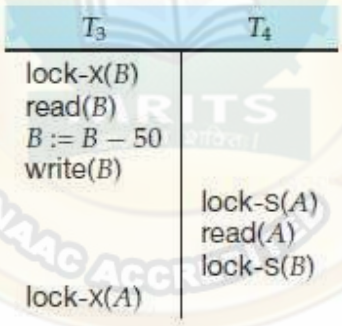


Fig: 4.2.11 Schedule 2

Preferable to inconsistent states, since they can be handled byrolling back transactions. We shall require that each transaction in the system follow a set of rules, called a locking protocol, indicating when a transaction may lock and unlock each of the data items. Locking protocolsrestrict the number of possible schedules. The set of all such schedules is aproper subset of all possible serializable schedules. We shall present several locking protocols that allow only conflict-serializable schedules, and thereby ensure isolation. Before doing so, we introduce some terminology.

Let *{T*0, *T*1, . . . , *Tn}* be a set of transactions participating in a schedule *S*. We say that *Ti* precedes *Tj* in *S*, written *Ti* → *Tj*, if there exists a data item *Q* such that *Ti* has held lock mode *A* on *Q*, and *Tj* has held lock mode *B* on *Q* later, and comp(*A*,*B*) = false. If *Ti* →*Tj* , then that precedence implies that in any equivalent serial schedule, *Ti* must appear

before *Tj* . Observe that this graph is similar to the precedence graph to test for conflict serializability.

Conflicts between instructions correspond to noncompatibility of lock modes. We say that a schedule *S* is legal under a given locking protocol if *S* is a possible schedule for a set of transactions that follows therules of the locking protocol. We say that a locking protocol ensures conflict serializability if and only if all legal schedules are conflict serializable; in other words, for all legal schedules the associated→relationis acyclic.

4.2.5 TIMESTAMP ORDERING ALGORITHMS

The locking protocols that we have described thus far determine the order between every pair of conflicting transactions at execution time by the first lock that both members of the pair request that involves incompatible modes. Another method for determining the serializability order is to select an ordering among transactions in advance. The most common method for doing so is to use a *timestamp-ordering* scheme.

Timestamps

With each transaction *Ti* in the system, we associate a unique fixed timestamp, denoted by TS(*Ti*). This timestamp is assigned by the database system before the transaction *Ti* starts execution. If a transaction *Ti* has been assigned timestamp TS(*Ti*), and a new transaction *Tj* enters the system, then TS(*Ti*) < TS(*Tj*). There are two simple methods for implementing this scheme:

1. Use the value of the system clock as the timestamp; that is, a transaction's timestamp is equal to the value of the clock when the transaction enters the system.

2. Use a logical counter that is incremented after a new timestamp has been assigned; that is, a transaction's timestamp is equal to the value ofthe counter when the transaction enters the system.

The timestamps of the transactions determine the serializability order. Thus, if TS(*Ti*) < TS(*Tj*), then the system must ensure that the produced schedule is equivalent to a serial schedule in which transaction *Ti* appears before transaction *Tj* .

To implement this scheme, we associate with each data item *Q* two timestamp values:

• W-timestamp(*Q*) denotes the largest timestamp of any transaction that executed write(*Q*) successfully.

• R-timestamp(Q) denotes the largest timestamp of any transaction that executed read(Q) successfully.

These timestamps are updated whenever a new read(Q) or write(Q) instruction is executed.

The Timestamp-Ordering Protocol

The timestamp-ordering protocol ensures that any conflicting read and write operations are executed in timestamp order. This protocol operates as follows:

1. Suppose that transaction Ti issues read(Q).

a. If TS(Ti) < W-timestamp(Q), then Ti needs to read a value of Q that was already overwritten. Hence, the read operation is rejected, and Ti is rolled back.

b. If TS(Ti) ≥ W-timestamp(Q), then the read operation is executed, and R-timestamp(Q) is set to the maximum of R-timestamp(Q) and TS(Ti).

2. Suppose that transaction Ti issues write(Q).

a. If TS(Ti) < R-timestamp(Q), then the value of Q that Ti is producing was needed previously, and the system assumed that that value would never be produced. Hence, the system rejects the write operation and rolls Ti back.

b. If TS(Ti) < W-timestamp(Q), then Ti is attempting to write an obsolete value of Q. Hence, the system rejects this write operation and rolls Ti back.

c. Otherwise, the system executes the write operation and sets W-timestamp(Q) to TS(Ti).

If transactions Ti is rolled back by the concurrency-control scheme as result of issuance of either a read or writes operation, the system assigns it a new timestamp and restarts it.

To illustrate this protocol, we consider transactions $T25$ and $T26$. Transaction $T25$ displays the contents of accounts A and B:

T_{25}: read(B);
read(A);
display($A + B$).

```
T26: read(B);
     B := B − 50;
     write(B);
     read(A);
     A := A + 50;
     write(A);
     display(A + B).
```

In presenting schedules under the timestamp protocol, we shall assume that a transaction is assigned a timestamp immediately before its first instruction. Thus, in schedule 3 of Figure 2.2.12, TS(*T*25)< TS(*T*26), and the schedule is possible under the timestamp protocol. We note thatthe preceding execution can also be produced by the two-phase locking protocol.

There are, however, schedules that are possible under the twophase locking protocol, but are not possible under the timestamp protocol, and vice versa.

The timestamp-ordering protocol ensures conflict serializability. This is because conflicting operations are processed in timestamp order. The protocol ensures freedom from deadlock, since no transaction everwaits.

However, there is a possibility of starvation of long transactions if a sequence of conflicting short transactions causes repeated restarting of the long transaction. If a transaction is suffering from repeated restarts, conflicting transactions need to be temporarily blocked to enable the transaction to finish.

T_{25}	T_{26}
read(B)	
	read(B)
	$B := B - 50$
	write(B)
read(A)	
	read(A)
display($A + B$)	
	$A := A + 50$
	write(A)
	display($A + B$)

Fig:2.2.12 Schedule 3

The protocol can generate schedules that are not recoverable. However, it can be extended to make the schedules recoverable, in one of several ways:

• Recoverability and cascadelessness can be ensured by performing all writes together at the end of the transaction. The writes must be atomic in the following sense: While the writes are in progress, no transaction is permitted to access any of the data items that have been written.

• Recoverability and cascadelessness can also be guaranteed by using a limited form of locking, whereby reads of uncommitted items are postponed until the transaction that updated the item commits.

• Recoverability alone can be ensured by tracking uncommitted writes, and allowing a transaction *Ti* to commit only after the commit of any transaction that wrote a value that *Ti* read. Commit dependencies, can be used for this purpose.

Thomas' Write Rule

We now present a modification to the timestamp-ordering protocol that allows greater potential concurrency than does the protocol. Let us consider schedule 4 of Figure 2.2.13, and apply the timestamp-ordering protocol. Since *T*27 starts before *T*28, we shall assume that TS(*T*27) < TS(*T*28). The read(*Q*) operation of *T*27 succeeds, as does the write(*Q*) operation of *T*28. When *T*27 attempts its write(*Q*) operation, we find that TS(*T*27) < W-timestamp(*Q*), since W timestamp(*Q*) = TS(*T*28). Thus, the write(*Q*) by *T*27 is rejected and transaction *T*27
must be rolled back.

Although the rollback of *T*27 is required by the timestamp- ordering protocol, it is unnecessary. Since *T*28 has already written *Q*, the value that *T*27 is attempting to write is one that will never need to be read. Any transaction *Ti* with TS(*Ti*) < TS(*T*28) that attempts a read(*Q*)will be rolled back, since TS(*Ti*)<W-timestamp(*Q*). Any transaction *Tj* with TS(*Tj*) > TS(*T*28) must read the value of *Q* written by *T*28, rather thanthe value that *T*27 is attempting to write.

This observation leads to modified version of the timestamp-ordering protocol in which obsolete write operations can be ignored under certain circumstances. The protocol rules for read operations remain unchanged..

T_{27}	T_{28}
read(Q)	
	write(Q)
write(Q)	

Fig: 2.2.13 Schedule 4

The modification to the timestamp-ordering protocol, called Thomas' write rule, is this: Suppose that transaction *Ti* issues write (*Q*).

1. If TS(*Ti*) < R-timestamp(*Q*), then the value of *Q* that *Ti* is producing was previously needed, and it had been assumed that the value would never be produced. Hence, the system rejects the write operation and rolls *Ti* back.

2. If TS(*Ti*) < W-timestamp(*Q*), then *Ti* is attempting to write an obsolete value of *Q*. Hence, this write operation can be ignored.

3. Otherwise, the system executes the write operation and sets W-timestamp (*Q*) to TS(*Ti*).

The difference between these rules and those of Section lies in the second rule. The timestamp-ordering protocol requires that *Ti* be rolled back if *Ti* issues write (*Q*) and TS(*Ti*) < W-timestamp(*Q*). However, here,in those cases where TS(*Ti*) ≥ R-timestamp(*Q*), we ignore the obsolete write. By ignoring the write, Thomas' write rule allows schedules that are not conflict serializable but are nevertheless correct. Those non-conflict-serializable schedules allowed satisfy the definition of *view serializable* schedules (see example box).

Thomas' write rule makes use of view serializability by, in effect, deleting obsolete write operations from the transactions that issue them. This modification of transactions makes it possible to generate serializable schedules that would not be possible under the other protocols presented in this chapter. For example, schedule 4 of above Figure is not conflict serializable and, thus, is not possible under the two-phase locking protocol, the tree protocol, or the timestamp-ordering protocol. Under Thomas' write rule, the write (*Q*) operation of *T*27 would be ignored. The result is a schedule that is *view equivalent* to the serial schedule <*T*27, *T*28>.

4.2.6 DEADLOCK MANAGEMENT

Deadlocks tend to be rare and typically involve very few transactions. In practice, therefore, database systems periodically check for deadlocks. When a transaction Ti is suspended because a lock that it requests cannot be granted, it must wait until all transactions Tj that currently hold conflicting locks release them. The lock manager maintains a structure called a waits-for graph to detect deadlock cycles. The nodes correspond to active transactions, and there is an arc from Ti to 'Tj if (and only if)Ti is waiting for 1) to release a lode The lock manager adds edges to this graph when it queues lock requests and removes edges when it grants lock requests. Consider the schedule shown in Figure 2.2.14. The last step, shown below the line, creates a cycle in the waits-for graph. Figure 2.2.15 shows the waits-for graph before and after this step.

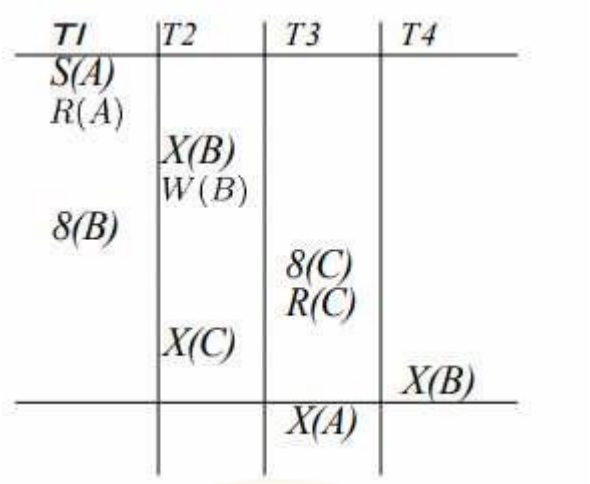


Fig: 2.2.14 Schedule Illustrating Deadlock

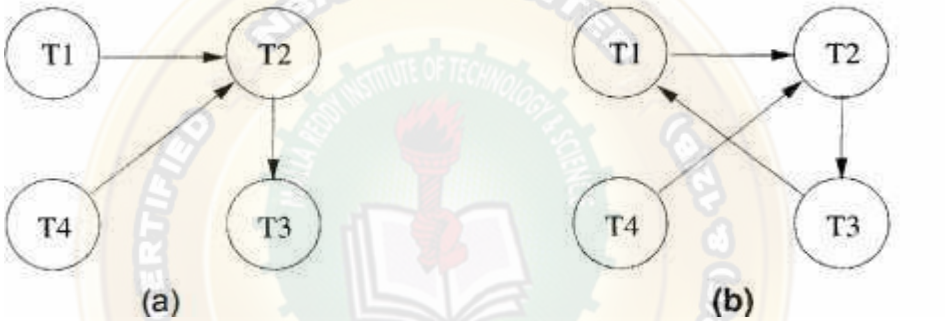


Fig: 2.2.15 Waits-for Graph Before and After Deadlock

Observe that the waits-for graph describes all active transactions, some of which eventually abort. If there is an edge from Ti to T'j in the waits-for graph, and both Ti and Tj eventually commit, there is an edge in the opposite direction (from T'j to Ti) in the precedence graph (which involves only transactions). The waits-for graph is periodically checked for cycles, which indicate deadlock. A deadlock is resolved by aborting a transaction that is on a cycle and releasing its locks; this action allows Some of the waiting transactions to proceed. The choice of which transaction to abort can be made using several criteria: the one with the fewest locks, the one that has done the least work, the one that is farthest from completion, and so all. Further, a transaction might have been repeatedly restarted; if so, it should eventually be favored during deadlock detection and allowed to complete. A simple alternative to maintaining a waits-for graph is to identify deadlocks through a timeout mechanism.

Deadlock Prevention

Empirical results indicate that deadlocks are relatively infrequent, and detection based schemes work well in practice. However, if there is a high level of contention for locks and therefore an increased likelihood of deadlocks, prevention based schedules could perform better. We can prevent deadlocks by giving each transaction a priority and ensuring that

lower-priority transactions are not allowed to wait for higher-priority transactions (or vice versa). One way to assign priorities is to give each transaction a timestamp when it starts up.

The lower the timestamp, the higher is the transaction's priority; that is, the oldest transaction has the highest priority.

If a transaction Ti requests a lock and transaction Tj holds a conflicting lock, the lock manager can use one of the following twopolicies:

II Wait-die: If Ti has higher priority, it is allowed to wait; otherwise, it is aborted.
II Wound-wait: If Ti has higher priority, abort 7); otherwise, l"1i waits

In the wait-die scheme, lower-priority transactions can never wait for higher priority transactions. In the wound-wait scheme, higher-priority transactions never wait for lower-priority transactions. In either ease, no deadlock cycle develops A subtle point is that we must also ensure that no transaction is perennially aborted because it never has a sufficiently high priority. (Note that, in both schemes, the higher-priority transaction is never aborted.) When a transaction is aborted and restarted, it should be given the same timestamp it had originally. Reissuing timestamps in this way ensures that each transaction will eventually become the oldest transaction, and therefore the one with the highest priority, and will get all the locks it requires. The wait-die scheme is non preemptive; only a transaction requesting a lock can be aborted. As a transaction grows older (and its priority increases), it tends to wait for more and more young transactions. A younger transaction that conflicts with an older transaction may be repeatedly aborted (a disadvantage with respect to wound-wait), but on the other hand, a transaction that has all the locks it needs is never aborted for deadlock reasons (an advantage with respect to wound-wait, which is preemptive).

A variant of 2PL, called Conservative 2PL, can also prevent deadlocks. Under Conservative 2PL, a transaction obtains all the locks it will ever need when it begins, or blocks waiting for these locks to become available. This scheme ensures that there will be no deadlocks, and, perhaps more important, that a transaction that already holds some locks will not block waiting for other locks. If lock contention is heavy, Conservative 2PL can reduce the time that locks are held on average, because transactions that hold locks are never blocked. The trade-off isthat a transaction acquires locks earlier, and if lock contention is low, locks are held longer under Conservative 2PL. From a practicalperspective, it is hard to know exactly what locks are needed ahead oftime, and this approach leads to setting more locks than necessary. It also has higher overhead for setting locks because a transaction has to release all locks and try to obtain them all over if it fails to obtain even one lock that it needs.

The Two-Phase Locking Protocol

One protocol that ensures serializability is the two-phase locking protocol. This protocol requires that each transaction issue lock and unlock requests in two phases:

1. Growing phase. A transaction may obtain locks, but may not release any lock.
2. Shrinking phase. A transaction may release locks, but may not obtain any new locks.

Initially, a transaction is in the growing phase. The transaction acquires locks as needed. Once the transaction releases a lock, it enters the shrinking phase, and it can issue no more lock requests.

For example, transactions *T*3 and *T*4 are two phase. On the other hand, transactions *T*1 and *T*2 are not two phase. Note that the unlock instructions do not need to appear at the end of the transaction. For example, in the case of transaction *T*3, we could move the unlock(*B*) instruction to just after the lock-X(*A*) instruction, and still retain the two- phase locking property. We can show that the two-phase locking protocol ensures conflict serializability. Consider any transaction. The point in the schedule where the transaction has obtained its final lock (the end of its growing phase) is called the lock point of the transaction. Now, transactions can be ordered according to their lock point this ordering is, in fact, a serializability ordering for the transactions. Two-phase locking does *not* ensure freedom from deadlock. Observe that transactions *T*3 and *T*4 are two phase, but, in schedule 2, they are deadlocked. Recall from Section that, in addition to being serializable, schedules should be cascadeless. Cascading rollback may occur under two-phase locking. Asan illustration, consider the partial schedule of Figure2.2.16. Each transaction observes the two-phase locking protocol, but the failure of *T*5 after the read (A) step of *T*7 leads to cascading rollback of *T*6 and *T*7.

Cascading rollbacks can be avoided by a modification of two- phase locking called the strict two-phase locking protocol. This protocol requires not only that locking be two phase, but also that all exclusive- mode locks taken by a transaction be held until that transaction commits. This requirement ensures that any data written by an uncommitted transaction are locked in exclusive mode until the transaction commits, preventing any other transaction from reading the data. Another variant of two-phase locking is the rigorous two-phase locking protocol, which requires that all locks be held until the transaction commits.

T_5	T_6	T_7
lock-X(A)		
read(A)		
lock-S(B)		
read(B)		
write(A)		
unlock(A)		
	lock-X(A)	
	read(A)	
	write(A)	
	unlock(A)	
		lock-S(A)
		read(A)

Fig: 2.2.16 Partial Schedule under two-phase locking

We can easily verify that, with rigorous two-phase locking, transactions can be serialized in the order in which they commit. Consider the following two transactions, for which we have shown only some of the significant read and write operations:

T_8: read(a_1);
read(a_2);
...
read(a_n);
write(a_1).

T_9: read(a_1);
read(a_2);
display($a_1 + a_2$).

If we employ the two-phase locking protocol, then *T*8 must lock *a*1 in exclusive mode. Therefore, any concurrent execution of both transactions amounts to a serial execution. Notice, however, that *T*8 needs an exclusive lock on *a*1 only at the end of its execution, when it writes *a*1. Thus, if *T*8 could initially lock *a*1 in shared mode, and then could later change the lock to exclusive mode, we could get more concurrency, since *T*8 and *T*9 could access *a*1 and *a*2 simultaneously. This observation leads us to a refinement of the basic two-phase locking protocol, in which lock conversions are allowed. We shall provide a mechanism for upgrading a shared lock to an exclusive lock, and downgrading an exclusive lock to a shared lock. We denote conversion from shared to exclusive modes by upgrade, and from exclusive to shared by downgrade. Lock conversion cannot be allowed arbitrarily. Rather, upgrading can take place in only the growing phase, whereas downgrading can take place in only the shrinking phase.

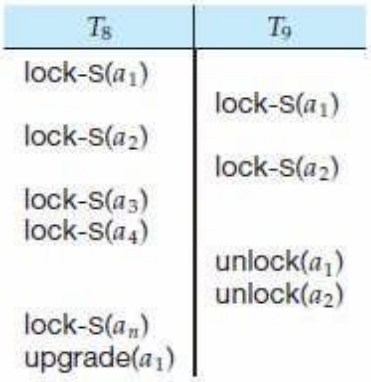


Fig: 2.2.17 Incomplete schedule with a lock conversion

Returning to our example, transactions *T*8 and *T*9 can run concurrently under the refined two-phase locking protocol, as shown in the incomplete schedule of Figure 2.2.17, where only some of the locking instructions are shown. Note that a transaction attempting to upgrade a lock on an item *Q* may be forced to wait. This enforced wait occurs if *Q* is currently locked by *another* transaction in shared mode.

Just like the basic two-phase locking protocol, two-phase locking with lock conversion generates only conflict-serializable schedules, and transactions can be serialized by their lock points. Further, if exclusive locks are held until the end of the transaction, the schedules are cascadeless. For a set of transactions, there may be conflict-serializable schedules that cannot be obtained through the two-phase locking protocol. However, to obtain conflict-serializable schedules through non-two-phase locking protocols, we need either to have additional information about the transactions or to impose some structure or ordering on the set of data items in the database. Strict two-phase locking and rigorous two-phase locking (with lock conversions) are used extensively in commercial database systems. A simple but widely used scheme automatically generates the appropriate lock and unlock instructions for a transaction, onthe basis of read and write requests from the transaction:

• When a transaction *Ti* issues a read(*Q*) operation, the system issues a lock-S(*Q*) instruction followed by the read(*Q*) instruction.

• When *Ti* issues a write(*Q*) operation, the system checks to see whether *Ti* already holds a shared lock on *Q*. If it does, then the system issues an upgrade(*Q*) instruction, followed by the write(*Q*) instruction. Otherwise, the system issues a lock-X(*Q*) instruction, followed by the write(*Q*) instruction.

• All locks obtained by a transaction are unlocked after that transaction commits or aborts.

Implementation of Locking

A lock manager can be implemented as a process that receives messages from transactions and sends messages in reply. The lock- manager process replies to lock-request messages with lock-grant messages, or with messages requesting rollback of the transaction (in case of deadlocks). Unlock messages require only an acknowledgment in response, but may result in a grant message to another waiting transaction.

The lock manager uses this data structure: For each data item that is currently locked, it maintains a linked list of records, one for each request, in the order in which the requests arrived. It uses a hash table, indexed on the name of a data item, to find the linked list (if any) for a data item; this table is called the lock table. Each record of the linked list for a data item notes which transaction made the request, and what lock mode it requested. The record also notes if the request has currently been granted. Figure 2.2.18 shows an example of a lock table. The table contains locks for five different data items, I4, I7, I23, I44, and I912. The lock table uses overflow chaining, so there is a linked list of data items for each entry in the lock table.

There is also a list of transactions that have been granted locks, or are waiting for locks, for each of the data items. Granted locks are the rectangles filled in a darker shade, while waiting requests are the rectangles filled in a lighter shade. We have omitted the lock mode to keepthe figure simple. It can be seen, for example, that T23 has been granted locks on I912 and I7, and is waiting for a lock on I4. Although the figure does not show it, the lock table should also maintain an index on transaction identifiers, so that it is possible to determine efficiently the set of locks held by a given transaction.

The lock manager processes requests this way: When a lock request message arrives, it adds a record to the end of the linked list for thedata item, if the linked list is present. Otherwise it creates a new linkedlist, containing only the record for the request. It always grants a lock request on a data item that is not currently locked. But if the transaction requests a lock on an item on which a lock is currently held, the lock manager grants the request only if it is compatible with the locks that are currently held, and all earlier requests have been granted already. Otherwise the request has to wait.

• When the lock manager receives an unlock message from a transaction, it deletes the record for that data item in the linked list corresponding to that transaction. It tests the record that follows, if any, as described in the previous paragraph, to see if that request can now be granted. If it can, the lock manager grants that request, and processes the record following it, if any, similarly, and so on.

• If a transaction aborts, the lock manager deletes any waiting request made by the transaction. Once the database system has taken appropriate actions to undo the transaction, it releases all locks held by the aborted transaction.

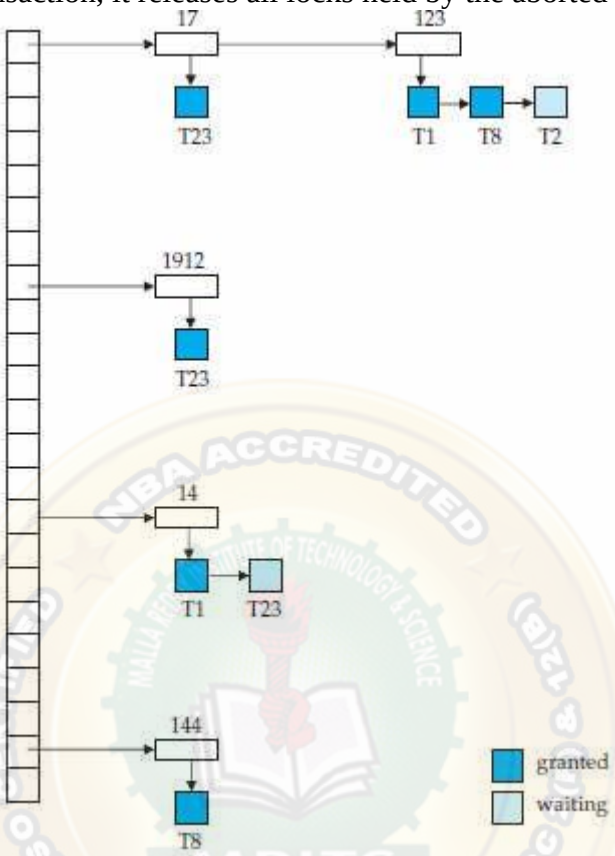


Fig: 2.2.18 Lock table

This algorithm guarantees freedom from starvation for lock requests, since a request can never be granted while a request received earlier is waiting to be granted.

4.3.1 DBMS RELIABILITY

We have referred to "reliability" and "availability" of the database a number of times so far without defining these terms precisely. Specifically, we mentioned these terms in conjunction with data replication, because the principle method of building a reliable system isto provide redundancy in system components. However, the distribution of the database or the replication of data items is not sufficient to make the distributed DBMS reliable. A number of protocols need to be implemented within the DBMS to exploit this distribution and replication in order to make operations more reliable. A reliable distributed database management system is one that can continue to process user requests even when the underlying system is unreliable. In other words, even whencomponents of the distributed computing environment fail, a reliable

distributed DBMS should be able to continue executing user requests without violating database consistency.

Too often, the terms reliability and availability are used loosely in literature. Even among the researchers in the area of reliable computer systems, the definitions of these terms sometimes vary. In this section, we give precise definitions of a number of concepts that are fundamental to an understanding and study of reliable systems.

4.3.2 DEFINITIONS AND BASIC CONCEPTS

Reliability refers to a system that consists of a set of components. The system has a state, which changes as the system operates. The behavior of the system in providing response to all the possible external stimuli is laid out in an authoritative specification of its behavior. The specification indicates the valid behavior of each system state. Any deviation of a system from the behavior described in the specification is considered a failure. For example, in a distributed transaction manager the specification may state that only serializable schedules for the execution ofconcurrent transactions should be generated. If the transaction manager generates a non-serializable schedule, we say that it has failed.

Each failure obviously needs to be traced back to its cause. Failures in a system can be attributed to deficiencies either in the components that make it up, or in the design, that is, how these components are put together. Each state that a reliable system goes through is valid in the sense that the state fully meets its specification. However, in an unreliable system, it is possible that the system may get to an internal state that may not obey its specification. Further transitions from this state would eventually cause a system failure. Such internalstates are called erroneous states; the part of the state that is incorrect is called an error in the system. Any error in the internal states of thecomponents of a system or in the design of a system is called a fault in the system. Thus, a fault causes an error that results in a system failure (Figure12.1).

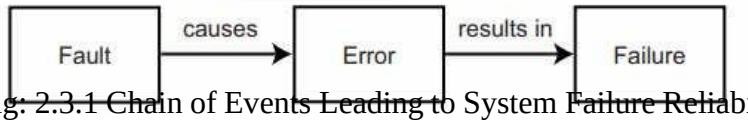


Fig: 2.3.1 Chain of Events Leading to System Failure Reliability

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

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

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

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

Under the same assumptions, it is possible to derive that

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

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

The expected (mean) number of failures in time $[0, t]$ can then be computed as

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

and the variance as

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

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

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

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

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

4.3.3 LOCAL RECOVERY MANAGEMENT

In this section we discuss the functions performed by the local recovery manager (LRM) that exists at each site. These functions maintain the atomicity and durability properties of local transactions. They relate to the execution of the commands that are passed to the LRM, which are begin transaction, read, write, commit, and abort. Later in this section we introduce a new command into the LRM's repertoire that initiates recovery actions after a failure. Note that in this section we discuss the execution of these commands in a centralized environment. The complications introduced in distributed databases are addressed in theupcoming sections.

Architectural Considerations

It is again time to use our architectural model and discuss the specific interface between the LRM and the database buffer manager(BM). The simple DP implementation that was given earlier will be enhanced with the reliability protocols discussed in this section. Also remember that all accesses to the database are via the database buffer

manager. The detailed discussion of the algorithms that the buffer manager implements is beyond the scope of this book; we provide asummary later in this subsection. Even without these details, we can still specify the interface and its function, as depicted in Figure 2.3.2 In this discussion we assume that the database is stored permanently on secondary storage, which in this context is called the stable storage [Lampson and Sturgis, 1976]. The stability of this storage medium is due to its robustness to failures. A stable storage device would experience considerably less-frequent failures than would a non-stable storage device.In today's technology, stable storage is typically implemented by means ofduplexed magnetic disks which store duplicate copies of data that are always kept mutually consistent (i.e., the copies are identical). We call the version of the database that is kept on stable storage the stable database. The unit of storage and access of the stable database is typically a page. The database buffer manager keeps some of the recently accessed data in main memory buffers. This is done to enhance access performance. Typically, the buffer is divided into pages that are of the same size as the stable database pages. The part of the database that is in the databasebuffer is called the volatile database. It is important to note that the LRM executes the operations on behalf of a transaction only on the volatile database, which, at a later time, is written back to the stable database.

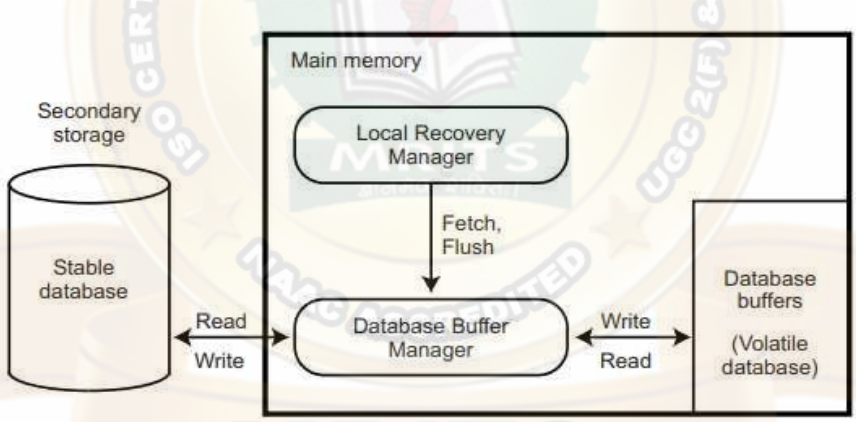


Fig: 2.3.2 Interface between the Local Recovery Manager and the Buffer Manager

When the LRM wants to read a page of data4 on behalf of a transaction—strictly speaking, on behalf of some operation of a transaction—it issues a fetch command, indicating the page that it wants to read. The buffer manager checks to see if that page is already in the buffer (due to a previous fetch command from another transaction) and if so, makes it available for that transaction; if not, it reads the page from the stable database into an empty database buffer. If no empty buffers exist, it selects one of the buffer pages to write back to stable storage and reads the requested stable database page into that buffer. There are a number of different algorithms by which the buffer manager may choose the buffer page to be replaced; these are discussed in standard database textbooks. The buffer manager also provides the interface by which the LRM can

actually force it to write back some of the buffer pages. This can be accomplished by means of the flush command, which specifies the buffer pages that the LRM wants to be written back. We should indicate that different LRM implementations may or may not use this forced writing. This issue is discussed further in subsequent sections. As its interface suggests, the buffer manager acts as a conduit for all access to the database via the buffers that it manages. It provides this function by fulfilling three tasks:

1. Searching the buffer pool for a given page;

2. If it is not found in the buffer, allocating a free buffer page and loading the buffer page with a data page that is brought in from secondary storage;

3. If no free buffer pages are available, choosing a buffer page for replacement.

Recovery Information

In this section we assume that only system failures occur. We defer the discussion of techniques for recovering from media failures until later. Since we are dealing with centralized database recovery, communication failures are not applicable. When a system failure occurs, the volatile database is lost. Therefore, the DBMS has to maintain some information about its state at the time of the failure in order to be able to bring the database to the state that it was in when the failure occurred. We call this information the recovery information. The recovery information that the system maintains is dependent on the method of executing updates. Two possibilities are in-place updating and out-of-place updating. In-place updating physically changes the value of the data item in the stable database. As a result, the previous values are lost. Out-of-place updating, on the other hand, does not change the value of the data item in the stable database but maintains the new value separately. Of course, periodically, these updated values have to be integrated into the stable database. We should note that the reliability issues are somewhat simpler if in-place updating is not used. However, most DBMS use it due to its improved performance.

4.3.4 IN-PLACE UPDATE

Since in-place updates cause previous values of the affected data items to be lost, it is necessary to keep enough information about the database state changes to facilitate the recovery of the database to a consistent state following a failure. This information is typically maintained in a database log. Thus each update transaction not only changes the database but the change is also recorded in the database log (Figure 2.3.3). The log contains information necessary to recover the database state following a failure.

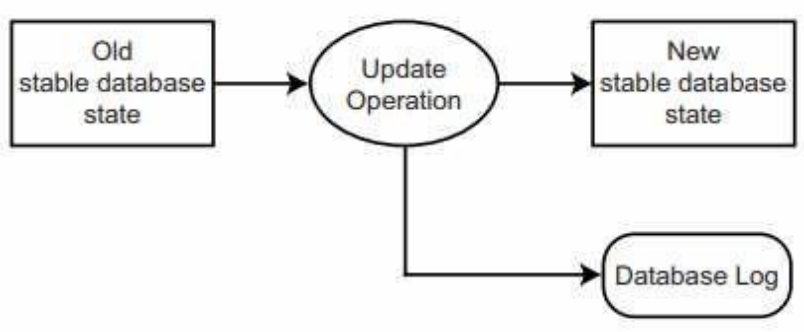


Fig: 2.3.2 Update Operation Execution

For the following discussion assume that the LRM and buffer manager algorithms are such that the buffer pages are written back to the stable database only when the buffer manager needs new buffer space. In other words, the flush command is not used by the LRM and the decision to write back the pages into the stable database is taken at the discretion ofthe buffer manager. Now consider that a transaction T1 had completed (i.e., committed) before the failure occurred. The durability property of transactions would require that the effect os T1 be reflected in the database. However, it is possible that the volatile database pages that have been updated by T1 may not have been written back to the stable database at the time of the failure. Therefore, upon recovery, it is important to be able to redo the operations of T1. This requires some information to be stored in the database log about the effects of T1. Given this information,it is possible to recover the database from its "old" state to the "new" state that reflects the effects of T1 (Figure 2.3.3).

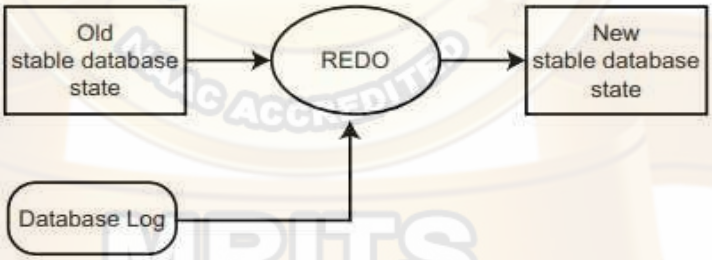


Fig: 2.3.3 REDO Action

Now consider another transaction, T2, that was still running when the failure occurred. The atomicity property would dictate that the stable database not contain any effects of T2. It is possible that the buffer manager may have had to write into the stable database some of thevolatile database pages that have been updated by T2. Upon recovery fromfailures it is necessary to undo the operations of T2. 5 Thus the recovery information should include sufficient data to permit the undo by taking the "new" database state that reflects partial effects of T2 and recovers the "old" state that existed at the start of T2 (Figure 2.3.4). We should indicatethat the undo and redo actions are assumed to be idempotent. In other words, their repeated application to a transaction would be equivalent to performing them once. Furthermore, the undo/redo actions form the basis

of different methods of executing the commit commands. The contents of the log may differ according to the implementation. However, the following minimal information for each transaction is contained in almost all database logs: a begin transaction record, the value of the data item before the update (called the before image), the updated value of the data item (called the after image), and a termination record indicating the transaction termination condition (commit, abort). The granularity of the before and after images may be different, as it is possible to log entire pages or some smaller unit. As an alternative to this form of state logging, operational logging, as in ARIES [Haderle et al., 1992], may be supported where the operations that cause changes to the database are logged rather than the before and after images.

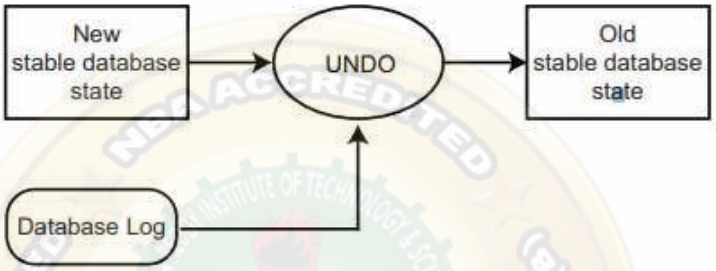


Fig: 2.3.4 UNDO Action

The log is also maintained in main memory buffers (called log buffers) and written back to stable storage (called stable log) similar to the database buffer pages (Figure 2.3.5). The log pages can be written to stable storage in one of two ways. They can be written synchronously (more commonly known as forcing a log) where the addition of each log record requires that the log be moved from main memory to stable storage.They can also be written asynchronously, where the log is moved to stablestorage either at periodic intervals or when the buffer fills up. When the log is written synchronously, the execution of the transaction is suspended until the write is complete. This adds some delay to the response-time performance of the transaction. On the other hand, if a failure occurs immediately after a forced write, it is relatively easy to recover to a consistent database state.

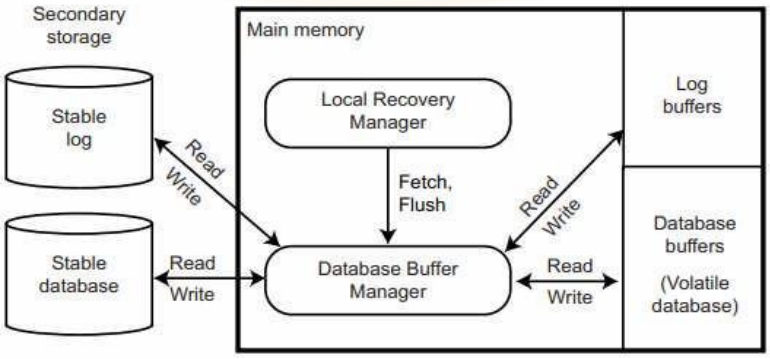


Fig:4.3.5 Logging Interface

Whether the log is written synchronously or asynchronously, one very important protocol has to be observed in maintaining logs. Consider a case where the updates to the database are written into the stable storage before the log is modified in stable storage to reflect the update. If a failure occurs before the log is written, the database will remain in updatedform, but the log will not indicate the update that makes it impossible to recover the database to a consistent and up-to-date state. Therefore, the stable log is always updated prior to the updating of the stable database. This is known as the write-ahead logging (WAL) protocol [Gray, 1979] and can be precisely specified as follows:

1. Before a stable database is updated (perhaps due to actions of a yet uncommitted transaction), the before images should be stored in the stable log. This facilitates undo.

2. When a transaction commits, the after images have to be stored in the stable log prior to the updating of the stable database. This facilitates redo.

4.3.5 OUT-OF-PLACE UPDATE

As we mentioned above, the most common update technique is in-place updating. Therefore, we provide only a brief overview of the other updating techniques and their recovery information. Details can be found in [Verhofstadt, 1978] and the other references given earlier. Typical techniques for out-of-place updating are shadowing ([Astrahan et al., 1976; Gray, 1979]) and differential files [Severence and Lohman, 1976]. Shadowing uses duplicate stable storage pages in executing updates. Thus every time an update is made, the old stable storage page, called the shadow page, is left intact and a new page with the updated data item values is written into the stable database. The access path data structures are updated to point to the new page, which contains the current data so that subsequent accesses are to this page. The old stable storage page is retained for recovery purposes (to perform undo).

Recovery based on shadow paging is implemented in System R's recovery manager [Gray et al., 1981]. This implementation uses shadowing together with logging. In general, the method maintains each stable database file as a read-only file. In addition, it maintains a corresponding read-write differential file that stores the changes to thatfile. Given a logical database file F, let us denote its read-only part as FR and its corresponding differential file as DF. DF consists of two parts: an insertions part, which stores the insertions to F, denoted DF+, and acorresponding deletions part, denoted DF−. All updates are treated as the deletion of the old value and the insertion of a new one. Thus each logical file F is considered to be a view defined as $F = (FR \cup DF+) - DF-$. Periodically, the differential file needs to be merged with the read-only base file. Recovery schemes based on this method simply use private differential files for each transaction, which are then merged with the

differential files of each file at commit time. Thus recovery from failures can simply be achieved by discarding the private differential files of non-committed transactions. There are studies that indicate that the shadowing and differential files approaches may be advantageous in certain environments. One study by Agrawal and DeWitt [1985] investigates the performance of recovery mechanisms based on logging, differential files, and shadow paging, integrated with locking and optimistic (using timestamps) concurrency control algorithms. The results indicate that shadowing, together with locking, can be a feasible alternative to the more common log-based recovery integrated with locking if there are only large (in terms of the base-set size) transactions with sequential access patterns. Similarly, differential files integrated with locking can be a feasible alternative if there are medium-sized and large transactions.

4.3.6 DISTRIBUTED RELIABILITY PROTOCOLS

As with local reliability protocols, the distributed versions aim to maintain the atomicity and durability of distributed transactions that execute over a number of databases. The protocols address the distributed execution of the begin transaction, read, write, abort, commit, and recover commands. At the outset we should indicate that the execution of the begin transaction, read, and write commands does not cause any significant problems. Begin transaction is executed in exactly the same manner as in the centralized case by the transaction manager at the originating site of the transaction. At each site, the commands are executed in the manner described in Section. Similarly, abort is executed by undoing its effects. The implementation of distributed reliability protocols within the architectural model we have adopted in this book raises a number of interesting and difficult issues. For the time being, we adopt a common abstraction: we assume that at the originating site of a transaction there is a coordinator process and at each site where the transaction executes there are participant processes. Thus, the distributed reliability protocols are implemented between the coordinator and the participants.

Components of Distributed Reliability Protocols

The reliability techniques in distributed database systems consist of commit, termination, and recovery protocols. Recall from the preceding section that the commit and recovery protocols specify how the commit and the recover commands are executed. Both of these commands need to be executed differently in a distributed DBMS than in a centralized DBMS. Termination protocols are unique to distributed systems. Assume that during the execution of a distributed transaction, one of the sites involved in the execution fails; we would like the other sites to terminate the transaction somehow. The techniques for dealing with this situationare called termination protocols. Termination and recovery protocols are two opposite faces of the recovery problem: given a site failure, termination protocols address how the operational sites deal with the

failure, whereas recovery protocols deal with the procedure that the process (coordinator or participant) at the failed site has to go through to recover its state once the site is restarted. In the case of network partitioning, the termination protocols take the necessary measures to terminate the active transactions that execute at different partitions, while the recovery protocols address the establishment of mutual consistency of replicated databases following reconnection of the partitions of the network. The primary requirement of commit protocols is that they maintain the atomicity of distributed transactions. This means that even though the execution of the distributed transaction involves multiple sites, some of which might fail while executing, the effects of the transaction on the distributed database is all-or-nothing. This is called atomic commitment. We would prefer the termination protocols to be non- blocking. A protocol is non-blocking if it permits a transaction to terminate at the operational sites without waiting for recovery of the failed site. This would significantly improve the response-time performance of transactions. We would also like the distributed recovery protocols to be independent. Independent recovery protocols determine how to terminatea transaction that was executing at the time of a failure without having to consult any other site. Existence of such protocols would reduce the number of messages that need to be exchanged during recovery. Note that the existence of independent recovery protocols would imply the existenceof non-blocking termination protocols, but the reverse is not true.

4.3.7 TWO PHASE COMMIT PROTOCOL

Two-phase commit (2PC) is a very simple and elegant protocol that ensures the atomic commitment of distributed transactions. It extends the effects of local atomic commit actions to distributed transactions by insisting that all sites involved in the execution of a distributed transaction agree to commit the transaction before its effects are made permanent. There are a number of reasons why such synchronization among sites is necessary. First, depending on the type of concurrency control algorithm that is used, some schedulers may not be ready to terminate a transaction. For example, if a transaction has read a value of a data item that is updated by another transaction that has not yet committed, the associated scheduler may not want to commit the former. Of course, strict concurrency control algorithms that avoid cascading aborts would not permit the updated value of a data item to be read by any other transaction until the updating transaction terminates. This is sometimes called the recoverability condition.

Another possible reason why a participant may not agree to commit is due to deadlocks that require a participant to abort the transaction. Note that, in this case, the participant should be permitted to abort the transaction without being told to do so. This capability is quite important and is called unilateral abort. A brief description of the 2PC protocol that does not consider failures is as follows. Initially, the coordinator writes a begin commit record in its log, sends a “prepare”

message to all participant sites, and enters the WAIT state. When a participant receives a "prepare" message, it checks if it could commit the transaction. If so, the participant writes a ready record in the log, sends a "vote-commit" message to the coordinator, and enters READY state; otherwise, the participant writes an abort record and sends a "vote-abort" message to the coordinator. If the decision of the site is to abort, it can forget about that transaction, since an abort decision serves as a veto (i.e., unilateral abort). After the coordinator has received a reply from every participant, it decides whether to commit or to abort the transaction. If even one participant has registered a negative vote, the coordinator has to abort the transaction globally. So it writes an abort record, sends a "global-abort" message to all participant sites, and enters the ABORT state; otherwise, it writes a commit record, sends a "global-commit" message to all participants, and enters the COMMIT state. The participants either commit or abort the transaction according to the coordinator's instructions and send back an acknowledgment, at which point the coordinator terminates the transaction by writing an end of transaction record in the log.

Note the manner in which the coordinator reaches a global termination decision regarding a transaction. Two rules govern this decision, which, together, are called the global commit rule:

1. If even one participant votes to abort the transaction, the coordinator has to reach a global abort decision.

2. If all the participants vote to commit the transaction, the coordinator has to reach a global commit decision. The operation of the 2PC protocol between a coordinator and one participant in the absence of failures is depicted in Figure 2.3.6, where the circles indicate the states and the dashed lines indicate messages between the coordinator and the participants. The labels on the dashed lines specify the nature of themessage.

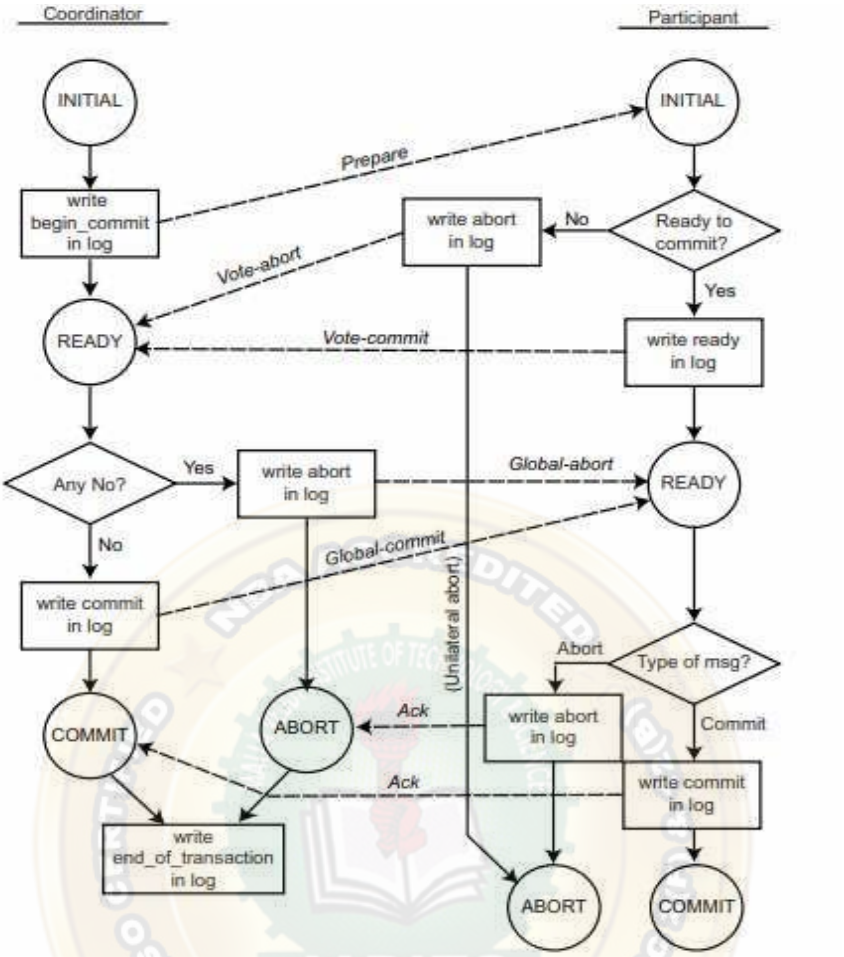


Fig:4.3.6 2PC Protocol Actions

A few important points about the 2PC protocol that can be observed from Figure 2.3.6 are as follows. First, 2PC permits a participant to unilaterally abort a transaction until it has decided to register an affirmative vote. Second, once a participant votes to commit or abort a transaction, it cannot change its vote. Third, while a participant is in the READY state, it can move either to abort the transaction or to commit it, depending on the nature of the message from the coordinator. Fourth, the global termination decision is taken by the coordinator according to the global commit rule. Finally, note that the coordinator and participant processes enter certain states where they have to wait for messages from one another. To guarantee that they can exit from these states and terminate, timers are used. Each process sets its timer when it enters a state, and if the expected message is not received before the timer runsout, the process times out and invokes its timeout protocol (which will be discussed later). There are a number of different communication paradigms that can be employed in implementing a 2PC protocol. The onediscussed above and depicted in Figure 2.3.6 is called a centralized 2PC since the communication is only between the coordinator and the participants; the participants do not communicate among themselves. This communication structure, which is the basis of our subsequent discussions in this chapter, is depicted more clearly in Figure 2.3.7.

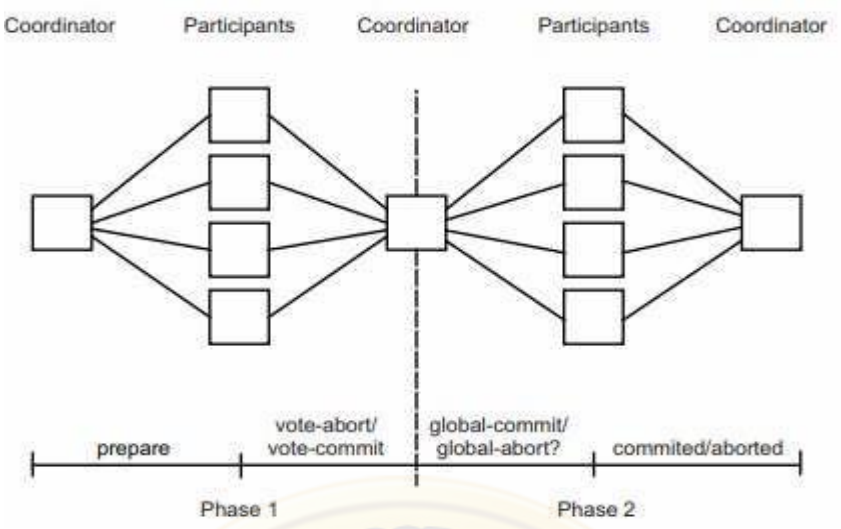


Fig: 4.3.6 Centralized 2PC Communication Structure

Another alternative is linear 2PC (also called nested 2PC [Gray, 1979]) where participants can communicate with one another. There is an ordering between the sites in the system for the purposes of communication. Let us assume that the ordering among the sites that participate in the execution of a transaction are 1, ..., N, where the coordinator is the first one in the order. The 2PC protocol is implemented by a forward communication from the coordinator (number 1) to N, during which the first phase is completed, and by a backward communication from N to the coordinator, during which the second phase is completed. Thus linear 2PC operates in the following manner. The coordinator sends the "prepare" message to participant 2. If participant 2 is not ready to commit the transaction, it sends a "vote-abort" message (VA) to participant 3 and the transaction is aborted at this point (unilateral abort by2). If, on the other hand, participant 2 agrees to commit the transaction, it sends a "votecommit" message (VC) to participant 3 and enters the READY state. This process continues until a "vote-commit" vote reaches participant N. This is the end of the first phase. If N decides to commit, it sends back to N −1 "global-commit" (GC); otherwise, it sends a "global- abort" message (GA). Accordingly, the participants enter the appropriate state (COMMIT or ABORT) and propagate the message back to the coordinator. Linear 2PC, whose communication structure is depicted in Figure 2.3.7, incurs fewer messages but does not provide any parallelism. Therefore, it suffers from low response-time performance.

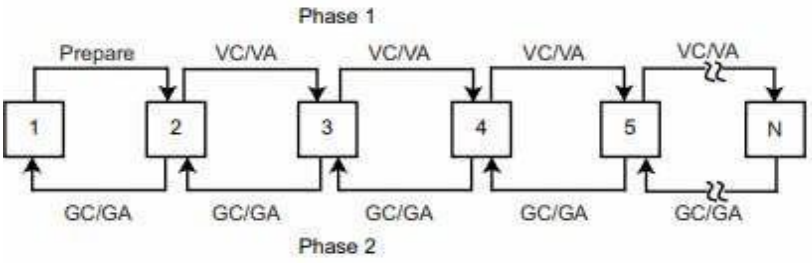


Fig: 4.3.7 Linear 2PC Communication Structure,VC vote.commit; VA vote.abort; GC global.commit; GA global.abort

Another popular communication structure for implementation of the 2PC protocol involves communication among all the participants during the first phase of the protocol so that they all independently reach their termination decisions with respect to the specific transaction. This version, called distributed 2PC, eliminates the need for the second phaseof the protocol since the participants can reach a decision on their own. It operates as follows. The coordinator sends the prepare message to all participants. Each participant then sends its decision to all the other participants (and to the coordinator) by means of either a "vote-commit"or a "vote-abort" message. Each participant waits for messages from allthe other participants and makes its termination decision according to the global commit rule. Obviously, there is no need for the second phase ofthe protocol (someone sending the global abort or global commit decision to the others), since each participant has independently reached that decision at the end of the first phase. The communication structure of distributed commit is depicted in Figure 2.3.8. One point that needs to be addressed with respect to the last two versions of 2PC implementation is the following. A participant has to know the identity of either the next participant in the linear ordering (in case of linear 2PC) or of all the participants (in case of distributed 2PC). This problem can be solved by attaching the list of participants to the prepare message that is sent by the coordinator. Such an issue does not arise in the case of centralized 2PC since the coordinator clearly knows who the participants are. The algorithm for the centralized execution of the 2PC protocol by the coordinator is given in Algorithm 12.1, and the algorithm for participantsis given in Algorithm 12.2.

Algorithm 12.1: 2PC Coordinator Algorithm (2PC-C)

```
begin
    repeat
        wait for an event ;
        switch event do
            case Msg Arrival
                Let the arrived message be msg ;
                switch msg do
                    case Commit          {commit command from scheduler}
                        write begin_commit record in the log ;
                        send "Prepared" message to all the involved
                        participants ;
                        set timer
                    case Vote-abort {one participant has voted to abort;
                    unilateral abort}
                        write abort record in the log ;
                        send "Global-abort" message to the other involved
                        participants ;
                        set timer
                    case Vote-commit
                        update the list of participants who have answered ;
                        if all the participants have answered then {all must
                        have voted to commit}
                            write commit record in the log ;
                            send "Global-commit" to all the involved
                            participants ;
                            set timer
                    case Ack
                        update the list of participants who have acknowledged ;
                        if all the participants have acknowledged then
                            write end_of_transaction record in the log
                        else
                            send global decision to the unanswering participants
            case Timeout
                execute the termination protocol
    until forever ;
end
```

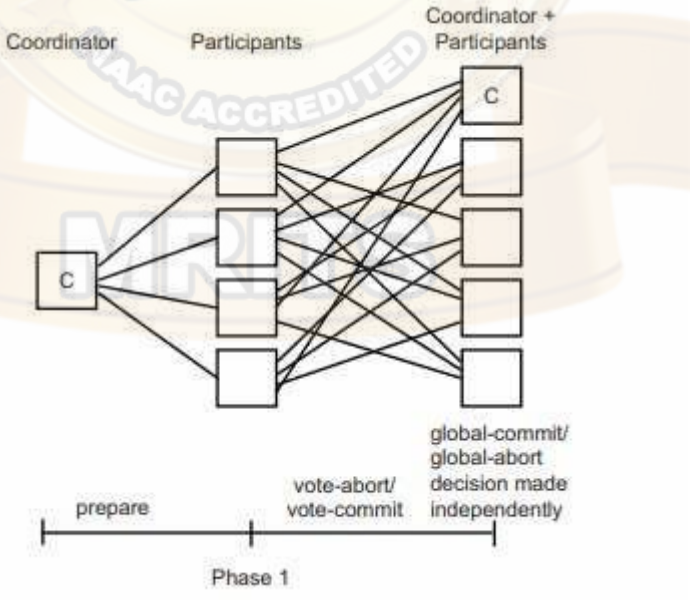


Fig: 4.3.8 Distributed 2PC Communication Structure

4.3.8 THREE PHASES COMMIT PROTOCOL

The three-phase commit protocol (3PC) [Skeen, 1981] is designed as a non-blocking protocol. We will see in this section that it is indeed non-blocking when failures are restricted to site failures. Let us first consider the necessary and sufficient conditions for designing non- blocking atomic commitment protocols. A commit protocol that is synchronous within one state transition is non-blocking if and only if its state transition diagram contains neither of the following:

1. No state that is "adjacent" to both a commit and an abort state.

2. No non-committable state that is "adjacent" to a commit state ([Skeen, 1981; Skeen and Stonebraker, 1983]). The term adjacent here means that it is possible to go from one state to the other with a single state transition

Consider the COMMIT state in the 2PC protocol. If any process is in this state, we know that all the sites have voted to commit the transaction. Such states are called committable. There are other states in the 2PC protocol that are non-committable. The one we are interested in is the READY state, which is non-committable since the existence of aprocess in this state does not imply that all the processes have voted to commit the transaction. It is obvious that the WAIT state in the coordinator and the READY state in the participant 2PC protocol violate the non-blocking conditions we have stated above. Therefore, one might be able to make the following modification to the 2PC protocol to satisfy the conditions and turn it into a non-blocking protocol. We can add another state between the WAIT (and READY) and COMMIT stateswhich serves as a buffer state where the process is ready to commit (if thatis the final decision) but has not yet committed. The state transition diagrams for the coordinator and the participant in this protocol are depicted in Figure 2.3.9. This is called the three-phase commit protocol (3PC) because there are three state transitions from the INITIAL state to a COMMIT state. The execution of the protocol between the coordinator and one participant is depicted in Figure. Note that this is identical to Figure except for the addition of the PRECOMMIT state. Observe that 3PC is also a protocol where all the states are synchronous within onestate transition. Therefore, the foregoing conditions for non-blocking 2PC apply to 3PC.

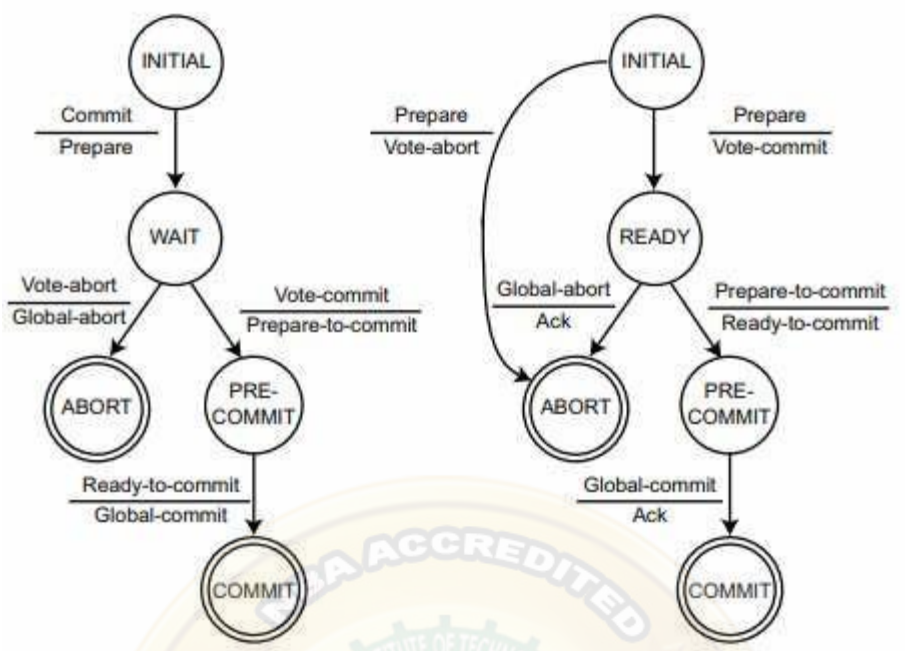


Fig: 4.3.9 State Transitions in 3PC Protocol

It is possible to design different 3PC algorithms depending on the communication topology. The one given in Figure is centralized. It is also straightforward to design a distributed 3PC protocol. A linear 3PC protocol is somewhat more involved, so we leave it as an exercise.

4.4.1 PARALLEL DATABASE SYSTEM

A parallel computer, or multiprocessor, is a special kind of distributed system made of a number of nodes (processors, memories and disks) connected by a very fast network within one or more cabinets in the same room. The main idea is to build a very powerful computer out of many small computers, each with a very good cost/performance ratio, at a much lower cost than equivalent mainframe computers. As discussed in Chapter 1, data distribution can be exploited to increase performance (through parallelism) and availability (through replication). This principle can be used to implement parallel database systems, i.e., database systems on parallel computers [DeWitt and Gray, 1992; Valduriez, 1993]. Parallel database systems can exploit the parallelism in data management in order to deliver high-performance and high-availability database servers. Thus, they can support very large databases with very high loads. Most of the research on parallel database systems has been done in the context of the relational model that provides a good basis for data-based parallelism. In this chapter, we present the parallel database system approach as a solution to high performance and high-availability data management. We discuss the advantages and disadvantages of the various parallel system architectures and we present the generic implementation techniques.

Implementation of parallel database systems naturally relies on distributed database techniques. However, the critical issues are data placement, parallel query processing, and load balancing because the number of nodes may be much higher than in a distributed DBMS. Furthermore, a parallel computer typically provides reliable, fast communication that can be exploited to efficiently implement distributed transaction management and replication. Therefore, although the basic principles are the same as in distributed DBMS, the techniques for parallel database systems are fairly different.

4.4.2 DEFINITION OF PARALLEL DATABASE SYSTEMS

Parallel processing exploits multiprocessor computers to run application programs by using several processors cooperatively, in order to improve performance. Its prominent use has long been in scientific computing by improving the response time of numerical applications [Kowalik, 1985; Sharp, 1987]. The developments in both general-purpose parallel computers using standard microprocessors and parallel programming techniques [Osterhaug, 1989] have enabled parallel processing to break into the data processing field. Parallel database systems combine database management and parallel processing to increase performance and availability. Note that performance was also the objective of database machines in the 70s and 80s [Hsiao, 1983]. The problem faced by conventional database management has long been known as "I/O bottleneck" [Boral and DeWitt, 1983], induced by high disk access time with respect to main memory access time (typicallyhundreds of thousands times faster).

A parallel database system can be loosely defined as a DBMS implemented on a parallel computer. This definition includes many alternatives ranging from the straightforward porting of an existing DBMS, which may require only rewriting the operating system interface routines, to a sophisticated combination of parallel processing and database system functions into a new hardware/software architecture. As always, we have the traditional trade-off between portability (to several platforms) and efficiency. The sophisticated approach is better able tofully exploit the opportunities offered by a multiprocessor at the expense of portability. Interestingly, this gives different advantages to computer manufacturers and software vendors. It is therefore important to characterize the main points in the space of alternative parallel system architectures. In order to do so, we will make precise the parallel database system solution and the necessary functions. This will be useful in comparing the parallel database system architectures. The objectives of parallel database systems are covered by those of distributed DBMS (performance, availability, extensibility). Ideally, a parallel database system should provide the following advantages.

1. **High-performance.** This can be obtained through several complementary solutions: database-oriented operating system support, parallel data management, query optimization, and load balancing. Having the operating system constrained and "aware" of the specific database requirements (e.g., buffer management) simplifies the implementation of low-level database functions and therefore decreases their cost. For instance, the cost of a message can be significantly reduced to a few hundred instructions by specializing the communication protocol. Parallelism can increase throughput, using inter-query parallelism, and decrease transaction response times, using intra-query parallelism. However, decreasing the response time of a complex query through large- scale parallelism may well increase its total time (by additional communication) and hurt throughput as a side-effect. Therefore, it is crucial to optimize and parallelize queries in order to minimize the overhead of parallelism, e.g., by constraining the degree of parallelism for the query. Load balancing is the ability of the system to divide a given workload equally among all processors. Depending on the parallel system architecture, it can be achieved statically by appropriate physical database design or dynamically at run-time.

2. **High-availability.** Because a parallel database system consists of many redundant components, it can well increase data availability and fault-tolerance. In a highly-parallel system with many nodes, the probability of a node failure at any time can be relatively high. Replicating data at several nodes is useful to support failover, a fault-tolerance technique that enables automatic redirection of transactions from a failed node to another node that stores a copy of the data. This provides un-interrupted service to users. However, it is essential that a node failure does not crate load imbalance, e.g., by doubling the load on the available copy. Solutions to this problem require partitioning copies in such a way that they can also beaccessed in parallel.

3. Extensibility. In a parallel system, accommodating increasing database sizes or increasing performance demands (e.g., throughput) should beeasier. Extensibility is the ability to expand the system smoothly by adding processing and storage power to the system. Ideally, the parallel database system should demonstrate two extensibility advantages [DeWitt and Gray, 1992]: linear speedup and linear scale up see Figure 2.4.1. Linear speedup refers to a linear increase in performance for a constant database size while the number of nodes (i.e., processing and storage power) are increased linearly. Linear scaleup refers to a sustained performance for a linear increase in both database size and number of nodes. Furthermore, extending the system should require minimal reorganization of the existing database.

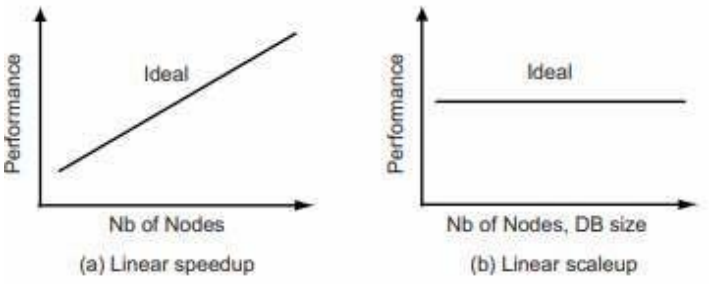


Fig: 4.4.1 Extensibility Metrics

Functional Architecture

Assuming client/server architecture, the functions supported by a parallel database system can be divided into three subsystems much like in a typical DBMS. The differences, though, have to do with implementation of these functions, which must now deal with parallelism, data partitioning and replication, and distributed transactions. Depending on the architecture, a processor node can support all (or a subset) of these subsystems. Figure 2.4.2 shows the architecture using these subsystems due to Bergsten et al. [1991].

1. Session Manager. It plays the role of a transaction monitor, providing support for client interactions with the server. In particular, it performs the connections and disconnections between the client processes and the two other subsystems. Therefore, it initiates and closes user sessions (which may contain multiple transactions). In case of OLTP sessions, the session manager is able to trigger the execution of pre-loaded transaction code within data manager modules.

2. Transaction Manager. It receives client transactions related to query compilation and execution. It can access the database directory that holds all meta-information about data and programs. The directory itself should be managed as a database in the server. Depending on the transaction, it activates the various compilation phases, triggers query execution, and returns the results as well as error codes to the client application. Because it supervises transaction execution and commit, it may trigger the recovery procedure in case of transaction failure. To speed up query execution, it may optimize and parallelize the query at compile-time.

3. Data Manager. It provides all the low-level functions needed to run compiled queries in parallel, i.e., database operator execution, parallel transaction support, cache management, etc. If the transaction manager is able to compile dataflow control, then synchronization and communication among data manager modules is possible. Otherwise, transaction control and synchronization must be done by a transaction manager module.

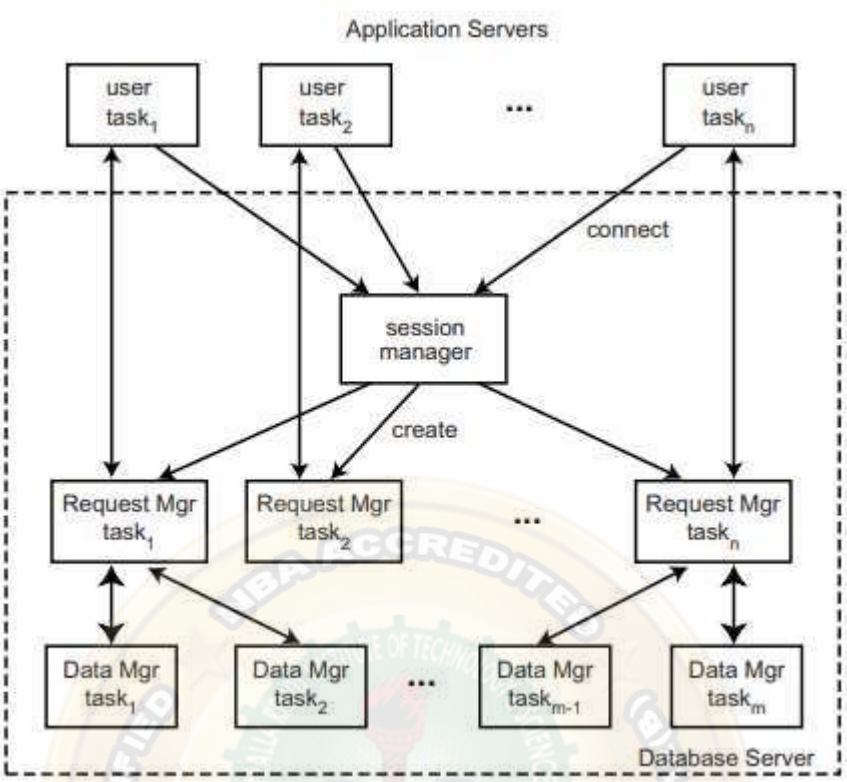


Fig: 4.4.2 General Architecture of a Parallel Database System

Parallel DBMS Architectures

As any system, a parallel database system represents a compromise in design choices in order to provide the aforementioned advantages with a good cost/performance. One guiding design decision is the way the main hardware elements, i.e., processors, main memory, and disks, are connected through some fast interconnection network. There are three basic parallel computer architectures depending on how main memory or disk is shared: shared-memory, shared-disk and shared-nothing. Hybrid architectures such as NUMA or cluster try to combine the benefits of the basic architectures. In the rest of this section, when describing parallel architectures, we focus on the four main hardware elements: interconnect, processors (P), main memory (M) and disks. For simplicity, we ignore other elements such as processor cache and I/O bus.

Shared-Memory

In the shared-memory approach (see Figure 2.4.3), any processor has access to any memory module or disk unit through a fast interconnect (e.g., a high-speed bus or a cross-bar switch). All the processors are under the control of a single operating system Current mainframe designs and symmetric multiprocessors (SMP) follow this approach. Examples of shared-memory parallel database systems include XPRS [Hong, 1992], DBS3 [Bergsten et al., 1991], and Volcano [Graefe, 1990], as well as

portings of major commercial DBMSs on SMP. In a sense, the implementation of DB2 on an IBM3090 with 6 processors [Cheng et al., 1984] was the first example. All shared-memory parallel database products today can exploit inter-query parallelism to provide high transaction throughput and intra-query parallelism to reduce response timeof decision-support queries.

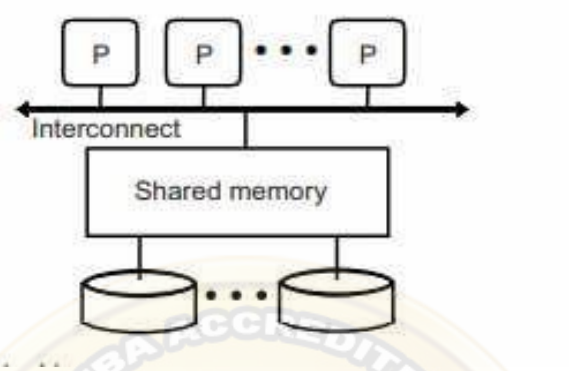


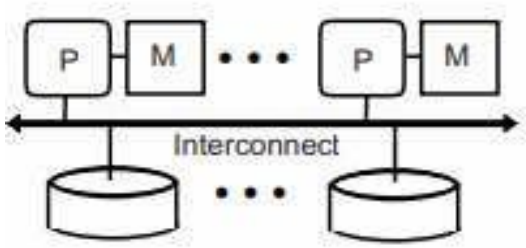
Fig: 4.4.3 Shared-Memory Architecture

Shared-memory has two strong advantages: simplicity and load balancing. Since meta-information (directory) and control information (e.g., lock tables) can be shared by all processors, writing database software is not very different than for single processor computers. In particular, inter-query parallelism comes for free. Intra-query parallelism requires some parallelization but remains rather simple. Load balancing is easy to achieve since it can be achieved at run-time using the shared- memory by allocating each new task to the least busy processor. Shared- memory has three problems: high cost, limited extensibility and low availability. High cost is incurred by the interconnect that requires fairly complex hardware because of the need to link each processor to each memory module or disk. With faster processors (even with larger caches), conflicting accesses to the shared-memory increase rapidly and degrade performance [Thakkar and Sweiger, 1990]. Therefore, extensibility is limited to a few tens of processors, typically up to 16 for the best cost/performance using 4-processor boards. Finally, since the memory space is shared by all processors, a memory fault may affect most processors thereby hurting availability. The solution is to use duplex memory with a redundant interconnect.

Shared-Disk

In the shared-disk approach (see Figure 2.4.4), any processor has access to any disk unit through the interconnect but exclusive (non-shared) access to its main memory. Each processor-memory node is under the control of its own copy of the operating system. Then, each processor can access database pages on the shared disk and cache them into its own memory. Since different processors can access the same page in conflicting update modes, global cache consistency is needed. This is typically achieved using a distributed lock manager that can be implemented using the techniques. The first parallel DBMS that used

shared-disk is Oracle with an efficient implementation of a distributed lock manager for cache consistency. Other major DBMS vendors such as IBM, Microsoft and Sybase provide shared-disk implementations



Fig; 4.4.4 Shared-Disk Architecture

Shared-disk has a number of advantages: lower cost, high extensibility, load balancing, availability, and easy migration from centralized systems. The cost of the interconnect is significantly less than with shared-memory since standard bus technology may be used. Given that each processor has enough main memory, interference on the shared disk can be minimized. Thus, extensibility can be better, typically up to a hundred processors. Since memory faults can be isolated from other nodes, availability can be higher. Finally, migrating from a centralized system to shared-disk is relatively straightforward since the data on disk need not be reorganized. Shared-disk suffers from higher complexity and potential performance problems. It requires distributed database system protocols, such as distributed locking and two-phase commit. As we have discussed in previous chapters, these can be complex. Furthermore, maintaining cache consistency can incur high communication overhead among the nodes. Finally, access to the shared-disk is a potential bottleneck.

Shared-Nothing

In the shared-nothing approach (see Figure 2.4.5), each processor has exclusive access to its main memory and disk unit(s). Similar to shared-disk, each processor memory-disk node is under the control of its own copy of the operating system. Then, each node can be viewed as a local site (with its own database and software) in a distributed database system. Therefore, most solutions designed for distributed databases such as database fragmentation, distributed transaction management and distributed query processing may be reused. Using a fast interconnect, it is possible to accommodate large numbers of nodes. As opposed to SMP, this architecture is often called Massively Parallel Processor (MPP). Many research prototypes have adopted the shared-nothing architecture, e.g., BUBBA [Boral et al., 1990], EDS [Group, 1990], GAMMA [DeWitt et al., 1986], GRACE [Fushimi et al., 1986], and PRISMA [Apers et al., 1992], because it can scale. The first major parallel DBMS product was Teradata's Database Computer that could accommodate a thousand processors in its early version. Other major DBMS vendors such as IBM, Microsoft and Sybase provide shared-nothing implementations.

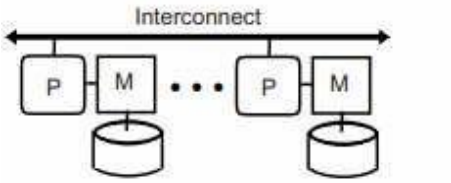


Fig: 4.4.5 Shared-Nothing Architecture

As demonstrated by the existing products, shared-nothing has three main virtues: lower cost, high extensibility, and high availability. The cost advantage is better than that of shared-disk that requires a special interconnect for the disks. By implementing a distributed database design that favors the smooth incremental growth of the system by the addition of new nodes, extensibility can be better (in the thousands of nodes). With careful partitioning of the data on multiple disks, almost linear speedup and linear scale-up could be achieved for simple workloads. Finally, by replicating data on multiple nodes, high availability can also be achieved. Shared-nothing is much more complex to manage than either shared-memory or shared-disk. Higher complexity is due to the necessary implementation of distributed database functions assuming large numbers of nodes. In addition, load balancing is more difficult to achieve because it relies on the effectiveness of database partitioning for the queryworkloads. Unlike shared-memory and shared-disk, load balancing is decided based on data location and not the actual load of the system. Furthermore, the addition of new nodes in the system presumably requires reorganizing the database to deal with the load balancing issues.

Hybrid Architectures

Various possible combinations of the three basic architectures are possible to obtain different trade-offs between cost, performance, extensibility, availability, etc. Hybrid architectures try to obtain the advantages of different architectures: typically the efficiency and simplicity of shared-memory and the extensibility and cost of either shareddisk or shared nothing. In this section, we discuss two popular hybrid architectures: NUMA and cluster.

NUMA. With shared-memory, each processor has uniform memory access (UMA), with constant access time, since both the virtual memory and the physical memory are shared. One major advantage is that the programming model based on shared virtual memory is simple. With either shared-disk or shared-nothing, both virtual and shared memory are distributed, which yields scalability to large numbers of processors. The objective of NUMA is to provide a shared-memory programming model and all its benefits, in a scalable architecture with distributed memory. The term NUMA reflects the fact that an access to the (virtually) shared memory may have a different cost depending on whether the physical memory is local or remote to the processor. The most successful class of NUMA multiprocessors is Cache Coherent NUMA (CC-NUMA)

[Goodman and Woest, 1988; Lenoski et al., 1992]. With CC-NUMA, the main memory is physically distributed among the nodes as with shared-nothing or shared-disk. However, any processor has access to all other processors' memories (see Figure 2.4.6). Each node can itself be an SMP. Similar to shared-disk, different processors can access the same data in a conflicting update mode, so global cache consistency protocols are needed. In order to make remote memory access efficient, the only viable solution is to have cache consistency done in hardware through a special consistent cache interconnect [Lenoski et al., 1992]. Because shared- memory and cache consistency are supported by hardware, remote memory access is very efficient, only several times (typically between 2 and 3 times) the cost of local access

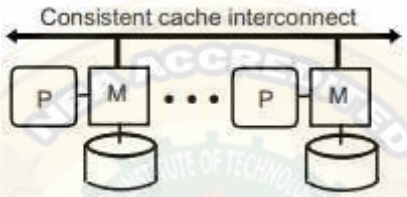


Fig: 4.4.6 Cache coherent NUMA (CC-NUMA)

Most SMP manufacturers are now offering NUMA systems that can scale up to a hundred processors. The strong argument for NUMA is that it does not require any rewriting of the application software. However some rewriting is still necessary in the database engine (and the operating system) to take full advantage of access locality [Bouganim et al., 1999]

Cluster

A cluster is a set of independent server nodes interconnected to share resources and form a single system. The shared resources, called clustered resources, can be hardware such as disk or software such as data management services. The server nodes are made of off-the-shelf components ranging from simple PC components to more powerful SMP. Using many off-the-shelf components is essential to obtain the best cost/performance ratio while exploiting continuing progress in hardware components. In its cheapest form, the interconnect can be a local network. However, there are now fast standard interconnects for clusters (e.g., Myrinet and Infiniband) that provide high bandwidth (Gigabits/sec) with low latency for message traffic. Compared to a distributed system, acluster is geographically concentrated (at a single site) and made of homogeneous nodes. Its architecture can be either shared nothing or shared-disk. Shared-nothing clusters have been widely used because they can provide the best cost/performance ratio and scale up to very large configurations (thousands of nodes). However, because each disk is directly connected to a computer via a bus, adding or replacing cluster nodes requires disk and data reorganization. Shared-disk avoids such reorganization but requires disks to be globally accessible by the cluster nodes. There are two main technologies to share disks in a cluster:

network-attached storage (NAS) and storage-area network (SAN). A NAS is a dedicated device to shared disks over a network (usually TCP/IP) using a distributed file system protocol such as Network File System (NFS). NAS is well suited for low throughput applications such as data backup and archiving from PC's hard disks. However, it is relatively slow and not appropriate for database management as it quickly becomes a bottleneck with many nodes. A storage area network (SAN) provides similar functionality but with a lower level interface. For efficiency, it uses a block-based protocol thus making it easier to manage cache consistency (at the block level). In fact, disks in a SAN are attached to the network instead to the bus as happens in Directly Attached Storage (DAS),but otherwise they are handled as sharable local disks. Existing protocols for SANs extend their local disk counterparts to run over a network (e.g., i-SCSI extends SCSI, and ATA-over-Ethernet extends ATA). As a result, SAN provides high data throughput and can scale up to large numbers of nodes. Its only limitation with respect to shared-nothing is its higher cost of ownership.

A cluster architecture has important advantages. It combines the flexibility and performance of shared-memory at each node with the extensibility and availability of shared-nothing or shared-disk. Furthermore, using off-the-shelf shared-memory nodes with a standard cluster interconnect makes it a cost-effective alternative to proprietary high-end multiprocessors such as NUMA or MPP. Finally, using SAN eases disk management and data placement.

4.4.3 PARALLEL QUERY EVALUATION

The objective of parallel query processing is to transform queries into execution plans that can be efficiently executed in parallel. This is achieved by exploiting parallel data placement and the various forms of parallelism offered by high-level queries. In this section, we first introduce the various forms of query parallelism. Then we derive basic parallel algorithms for data processing. Finally, we discuss parallel query optimization.

4.4.4 QUERY PARALLELISM

Parallel query execution can exploit two forms of parallelism: inter- and intra-query. Inter-query parallelism enables the parallel execution of multiple queries generated by concurrent transactions, inorder to increase the transactional throughput. Within a query (intra-query parallelism), inter-operator and intra-operator parallelism are used to decrease response time. Inter-operator parallelism is obtained by executing in parallel several operators of the query tree on several processors while with intra-operator parallelism, the same operator is executed by many processors, each one working on a subset of the data. Note that these two forms of parallelism also exist in distributed query processing

4.4.5 I/O PARALLELISM (DATA PARTITIONING)

A shared-nothing architecture because it is the most general case and its implementation techniques also apply, sometimes in a simplified form, to other architectures. Data placement in a parallel database system exhibits similarities with data fragmentation in distributed databasee. An obvious similarity is that fragmentation can be used to increase parallelism. In what follows, we use the terms partitioning and partition instead of horizontal fragmentation and horizontal fragment, respectively, to contrast with the alternative strategy, which consists of clustering a relation at a single node. The term declustering is sometimes used to meanpartitioning [Livny et al., 1987]. Vertical fragmentation can also be usedto increase parallelism and load balancing much as in distributed databases. Another similarity is that since data are much larger than programs, execution should occur, as much as possible, where the data reside. However, there are two important differences with the distributed database approach. First, there is no need to maximize local processing (at each node) since users are not associated with particular nodes. Second, load balancing is much more difficult to achieve in the presence of a large number of nodes. The main problem is to avoid resource contention,which may result in the entire system thrashing (e.g., one node ends up doing all the work while the others remain idle). Since programs are executed where the data reside, data placement is a critical performance issue. Data placement must be done so as to maximize system performance, which can be measured by combining the total amount of work done by the system and the response time of individual queries. In Chapter 8, we have seen that maximizing response time (through intra-query parallelism) results in increased total work due to communication overhead. For the same reason, inter-query parallelism results in increased total work. On the other hand, clustering all the data necessary to a program minimizes communication and thus the total work done by the system in executing that program. In terms of data placement, we have the following trade-off: maximizing response time or inter-query parallelism leads to partitioning, whereas minimizing the total amount of work leads to clustering. As we have seen in Chapter 3, this problem is addressed in distributed databases in a rather static manner. The database administrator is in charge of periodically examining fragment access frequencies, and when necessary, moving and reorganizing fragments. An alternative solution to data placement is full partitioning, whereby each relation is horizontally fragmented across all the nodes in the system. There are three basic strategies for data partitioning: round-robin, hash, and range partitioning (Figure 2.4.7).

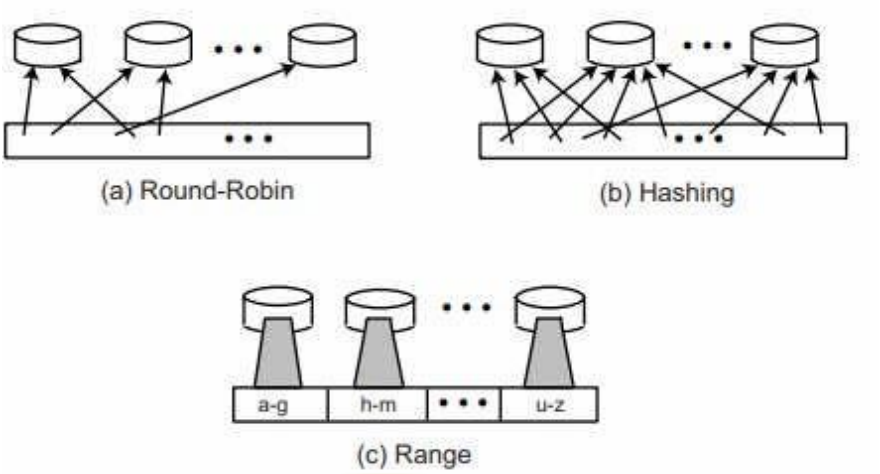


Fig: 4.4.7 Different Partitioning Schemes

1. Round-robin partitioning is the simplest strategy, it ensures uniform data distribution. With n partitions, the ith tuple in insertion order isassigned to partition (i mod n). This strategy enables the sequential access to a relation to be done in parallel. However, the direct access to individual tuples, based on a predicate, requires accessing the entire relation.

2. Hash partitioning applies a hash function to some attribute that yields the partition number. This strategy allows exact-match queries on the selection attribute to be processed by exactly one node and all otherqueries to be processed by all the nodes in parallel.

3. Range partitioning distributes tuples based on the value intervals (ranges) of some attribute. In addition to supporting exact-match queries (as in hashing), it is well-suited for range queries. For instance, a query with a predicate "A between A1 and A2" may be processed by the only node(s) containing tuples whose A value is in range [A1,A2].

However, range partitioning can result in high variation in partition size. Compared to clustering relations on a single (possibly very large) disk, full partitioning yields better performance [Livny et al., 1987]. Although full partitioning has obvious performance advantages, highly parallel execution might cause a serious performance overhead for complex queries involving joins. Furthermore, full partitioning is not appropriate for small relations that span a few disk blocks. These drawbacks suggest that a compromise between clustering and full partitioning (i.e., variable partitioning), needs to be found. A solution is to do data placement by variable partitioning [Copeland et al., 1988]. The degree of partitioning, i.e., the number of nodes over which a relation is fragmented, is a function of the size and access frequency of the relation. This strategy is much more involved than either clustering or full partitioning because changes in data distribution may result in reorganization. For example, a relation initially placed across eight nodes

may have its cardinality doubled by subsequent insertions, in which case it should be placed across 16 nodes. In a highly parallel system with variable partitioning, periodic reorganizations for load balancing are essential and should be frequent unless the workload is fairly static and experiences only a few updates. Such reorganizations should remain transparent to compiled programs that run on the database server. In particular, programs should not be recompiled because of reorganization. Therefore, the compiled programs should remain independent of data location, which may change rapidly. Such independence can be achieved if the run-time system supports associative access to distributed data. This is different from a distributed DBMS, where associative access is achieved at compiletime by the query processor using the data directory.

4.4.6 INTRA-QUERY PARALLELISM

Parallel query execution can exploit two forms of parallelism: inter- and intra-query. Within a query (intra-query parallelism), inter-operator and intra-operator parallelism are used to decrease response time.

4.4.7 INTER –QUERY PARALLELISM

Inter-query parallelism enables the parallel execution of multiple queries generated by concurrent transactions, in order to increase the transactional throughput. Inter-operator parallelism is obtained by executing in parallel several operators of the query tree on several processors while with intra-operator parallelism, the same operator is executed by many processors, each one working on a subset of the data. Note that these two forms of parallelism also exist in distributed query processing.

4.4.8 INTRA OPERATION PARALLELISM

Intra-operator parallelism is based on the decomposition of one operator in a set of independent sub-operators, called operator instances. This decomposition is done using static and/or dynamic partitioning of relations. Each operator instance will then process one relation partition, also called a bucket. The operator decomposition frequently benefits from the initial partitioning of the data (e.g., the data are partitioned on the join attribute). To illustrate intra-operator parallelism, let us consider a simple select-join query. The select operator can be directly decomposed into several select operators, each on a different partition, and no redistribution is required (Figure 2.4.8). Note that if the relation is partitioned on the select attribute, partitioning properties can be used to eliminate some select instances. For example, in an exact-match select, only one select instance will be executed if the relation was partitioned by hashing (or range) on the select attribute. It is more complex to decompose the join operator. In order to have independent joins, each bucket of the firstrelation R may be joined to the entire relation S. Such a join will be very

inefficient (unless S is very small) because it will imply a broadcast of S on each participating processor. A more efficient way is to use partitioning properties. For example, if R and S are partitioned by hashing on the join attribute and if the join is an equijoin, then we can partition the join into independent joins. This is the ideal case that cannot be always used,because it depends on the initial partitioning of R and S. In the other cases,one or two operands may be repartitioned [Valduriez and Gardarin, 1984].Finally, we may notice that the partitioning function (hash, range, round robin) is independent of the local algorithm (e.g., nested loop, hash, sort merge) used to process the join operator (i.e., on each processor). For instance, a hash join using a hash partitioning needs two hash functions. The first one, h1, is used to partition the two base relations on the join attribute. The second one, h2, which can be different for each processor, isused to process the join on each processor.

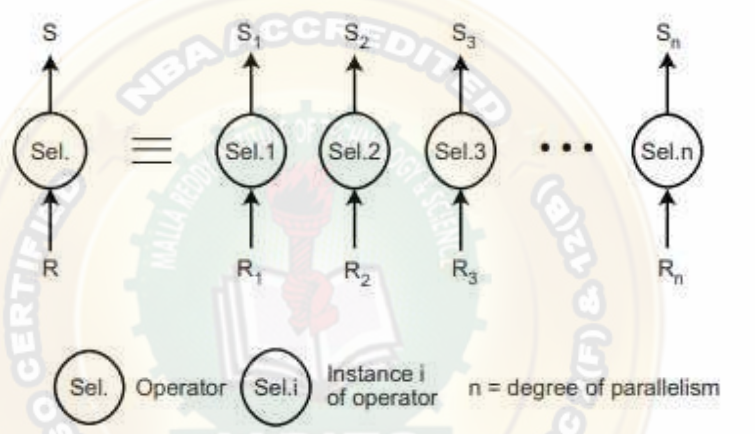


Fig: 4.4.8 Intra-operator Parallism

4.4.9 INTER OPERATION PARALLELISM

Two forms of inter-operator parallelism can be exploited. With pipeline parallelism, several operators with a producer-consumer link are executed in parallel. For instance, the select operator in Figure 2.4.9 will be executed in parallel with the join operator. The advantage of such execution is that the intermediate result is not materialized, thus saving memory and disk accesses. In the example of Figure 2.4.9, only S may fit in memory. Independent parallelism is achieved when there is no dependency between the operators that are executed in parallel. For instance, the two select operators of Figure 2.4.9 can be executed inparallel. This form of parallelism is very attractive because there is no interference between the processors.

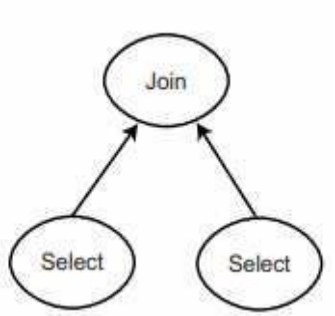


Fig:4.5.3 Inter-operator Parallelism

4.4.10 LET US SUM UP

Thus, we have studied basic concepts of database transaction, ACID properties, concurrency control, timestamp ordering algorithm, deadlock management. With this local recovery management, 2PC and 3PC protocol. Also the major aspect parallel database system as well.

4.4.11 LIST OF REFERENCES

- Principles of Distributed Database Systems; 2nd Editied By M. Tamer Ozsu and Patrick Valduriez, Person Education Asia.
- Database System Concepts, 5th edition, Avi Silberschatz , Henry F. Korth , S. Sudarshan: McGraw-Hill (2010)
- Distributed Database; Principles & Systems By Publications, Stefano Ceri and Giuseppo Pelagatti,, McGraw-Hill International Editions (1984)
- Database Management Systems, 3rd edition, Raghu Ramakrishnan and Johannes Gehrke, McGraw-Hill (2002).
- Fundamentals of Database Systems, 6thEdition, Elmasri and Navathe, Addison. Wesley (2003).
- Unifying temporal data models via a conceptual model, C.S. Jensen, M.D. Soo, and R.T. Snodgrass: Information Systems, vol. 19, no. 7, pp. 513-547, 1994.
- Spatial Databases: A Tour by Shashi Shekhar and Sanjay Chawla, Prentice Hall, 2003 (ISBN 013-017480-7)
- Principles of Multimedia Database Systems, Subramanian V. S. Elsevier Publishers, 2013.

4.4.12 UNIT END EXERCISES

1) Explain with Example:
    a. Intra-query parallelism
    b. Intra-operation parallelism
    c. Inter-operation parallelism
2) Explain ACID properties.
3) Explain two and three phase commit protocol.
4) Explain timestamp ordering algorithm.

❖❖❖❖

5

OBJECT ORIENTED, TEMPORAL AND SPATIAL DATABASES

Unit Structure

5.1.0 OBJECTIVES

In this chapter you will learn about:

- Basic of object, constructor, methods, inheritance etc.
- Object-oriented database-language and design ODMG model, ODL,OQL
- Temporal Database-Time series, structure, granularity, data model etc.
- Spatial Database-conceptual, logical data model
- Geographical Information Systems (GIS)s

5.1.1 INTRODUCTION

This chapter introduces database concepts for some of the common features that are needed by advanced applications and are being used widely. We will *temporal concepts* that are used in temporal database applications, and, briefly, some of the issues involving *spatial database.*

In this chapter, we discuss the features of object oriented data models and show how some of these features have been incorporated in relational database systems. Object-oriented databases are now referred to as **object databases** (**ODB**) (previously called OODB), and the database systems are referred to as **object data management systems** (**ODMS**) (formerly referred to as ODBMS or OODBMS). Traditional data models and systems, such as relational, network, and hierarchical, have been quite successful in developing the database technologies required for many traditional business database applications.

We introduce the concepts of **temporal databases**, which permit the database system to store a history of changes, and allow users to query both current and past states of the database. Some temporal databasemodels also allow users to store future expected information, such as planned schedules. It is important to note that many database applications are temporal, but they are often implemented without having much temporal support from the DBMS package—that is, the temporal concepts are implemented in the application programs that access the database.

We discuss types of spatial data, different kinds of spatial analyses, operations on spatial data, types of spatial queries, spatial data indexing, spatial data mining, and applications of spatial databases.

5.1.2 OBJECT ORIENTED DATABASE

The features of object oriented data models and show how some of these features have been incorporated in relational database systems. Object-oriented databases are now referred to as **object databases** (**ODB**) (previously called OODB), and the database systems are referred to as **object data management systems** (**ODMS**) (formerly referred to as ODBMS or OODBMS). Traditional data models and systems, such as relational, network, and hierarchical, have been quite successful in developing the database technologies required for many traditional business database applications. However, they have certain shortcomings when more complex database applications must be designed and implemented—for example, databases for engineering design and manufacturing (CAD/CAM and CIM1), scientific experiments, telecommunications, geographic information systems, and multimedia. These newer applications have requirements and characteristics that differ from those of traditional business applications, such as more complex structures for stored objects; the need for new data types for storing images, videos, or large textual items; longer-duration transactions; and the need to define nonstandard application-specific operations. Object databases were proposed to meet some of the needs of these more complex applications. A key feature of object databases is the power they give the designer to specify both the *structure* of complex objects and the *operations* that can be applied to these objects. Another reason for the creation of object-oriented databases is the vast increase in the use of object-oriented programming languages for developing software applications. Databases are fundamental components in many software systems, and traditional databases are sometimes difficult to use with software applications that are developed in an object-oriented programming language such as C++ or Java. Object databases are designed so they can be directly—or *seamlessly*—integrated with software that is developed using object-oriented programming languages. Relational DBMS (RDBMS) vendors have also recognized the need for incorporating features that were proposed for object databases, and newer versions of relational systems have incorporated many of these features. This has led to database systems that are characterized as *object-relational* or ORDBMSs. The latest version of the SQL standard (2008) for RDBMSs includes many of these features, which were originally known as SQL/Object and they have now been merged into the main SQL specification, known as SQL/Foundation. Although many experimental prototypes and commercial object-oriented database systems have been created, they have not found widespread use because of the popularity of relational and object-relational systems. The experimental prototypes included the Orion system developed at MCC, Open OODB at Texas Instruments, the Iris system at Hewlett-Packard laboratories, the Ode system at AT&T Bell Labs, and the ENCORE/ObServer project at Brown University. Commercially available systems included GemStone Object Server of GemStone Systems, ONTOS DB of Ontos, Objectivity/DB of Objectivity Inc.,Versant Object Database and Fast Objects by Versant

Corporation (and Poet), ObjectStore of Object Design, and Ardent Database of Ardent. These represent only a partial list of the experimental prototypes and commercial object-oriented database systems that were created. As commercial object DBMSs became available, the need for a standard model and language was recognized. Because the formal procedure for approval of standards normally takes a number of years, a consortium of object DBMS vendors and users, called ODMG, proposed a standard whose current specification is known as the ODMG 3.0 standard. Object-oriented databases have adopted many of the concepts that were developed originally for object-oriented programming languages. We describe the key concepts utilized in many object database systems andthat were later incorporated into object-relational systems and the SQL standard. These include *object identity, object structure* and *typeconstructors*, *encapsulation of operations* and the definition of *methods* as part of class declarations, mechanisms for storing objects in a database by making them *persistent,* and *type and class hierarchies* and *inheritance*. Then, in we see how these concepts have been incorporated into the latest SQL standards, leading to object-relational databases. Object features were originally introduced in SQL:1999, and then updated in the latest version (SQL:2008) of the standard. We turn our attention to "pure" objectdatabase standards by presenting features of the object database standard ODMG 3.0 and the object definition language ODL. An overview of the database design process for object databases. The object query language (OQL), which is part of the ODMG 3.0 standard. We discuss programming language bindings, which specify how to extend object oriented programming languages to include the features of the object database standard.

5.1.3 OBJECT IDENTITY

One goal of an ODMS (Object Data Management System) is to maintain a direct correspondence between real-world and database objects so that objects do not lose their integrity and identity and can easily be identified and operated upon. Hence, an ODMS provides a **unique identity** to each independent object stored in the database. This unique identity is typically implemented via a unique, system-generated **object identifier (OID)**. The value of an OID is not visible to the external user, but is used internally by the system to identify each object uniquely and to create and manage inter-object references. The OID can be assigned to program variables of the appropriate type when needed.

The main property required of an OID is that it be **immutable**; that is, the OID value of a particular object should not change. This preserves the identity of the real-world object being represented. Hence, an ODMS must have some mechanism for generating OIDs and preserving the immutability property. It is also desirable that each OID be used onlyonce; that is, even if an object is removed from the database, its OID should not be assigned to another object. These two properties imply that the OID should not depend on any attribute values of the object, since the

value of an attribute may be changed or corrected. We can compare this with the relational model, where each relation must have a primary key attribute whose value identifies each tuple uniquely. In the relational model, if the value of the primary key is changed, the tuple will have a new identity, even though it may still represent the same real-world object.Alternatively, a real-world object may have different names for key attributes in different relations, making it difficult to ascertain that the keys represent the same real-world object (for example, the object identifier may be represented as Emp_id in one relation and as Ssn in another). It is inappropriate to base the OID on the physical address of the object in storage, since the physical address can change after a physical reorganization of the database.

However, some early ODMSs have used the physical address asthe OID to increase the efficiency of object retrieval. If the physicaladdress of the object changes, an *indirect pointer* can be placed at the former address, which gives the new physical location of the object. It is more common to use long integers as OIDs and then to use some form of hash table to map the OID value to the current physical address of the object in storage. Some early OO data models required that everything— from a simple value to a complex object—was represented as an object; hence, every basic value, such as an integer, string, or Boolean value, has an OID. This allows two identical basic values to have different OIDs, which can be useful in some cases. For example, the integer value 50 can sometimes be used to mean a weight in kilograms and at other times to mean the age of a person. Then, two basic objects with distinct OIDs could be created, but both objects would represent the integer value 50. Although useful as a theoretical model, this is not very practical, since it leads to the generation of too many OIDs. Hence, most OO database systems allow for the representation of both objects and **literals** (or values). Every object must have an immutable OID, whereas a literal value has no OID and its value just stands for itself. Thus, a literal value istypically stored within an object and *cannot be referenced* from other objects. In many systems, complex structured literal values can also be created without having a corresponding OID if needed.

5.1.4 OBJECT STRUCTURE

The term *object-oriented*—abbreviated *OO* or *O-O*—has itsorigins in OO programming languages, or OOPLs. Today OO concepts areapplied in the areas of databases, software engineering, knowledge bases, artificial intelligence, and computer systems in general. OOPLs have their roots in the SIMULA language, which was proposed in the late 1960s. The programming language Smalltalk, developed at Xerox PARC8 in the 1970s, was one of the first languages to explicitly incorporate additional OO concepts, such as message passing and inheritance. It is known as a *pure* OO programming language, meaning that it was explicitly designedto be object-oriented. This contrasts with *hybrid* OO programming languages, which incorporate OO concepts into an already existing

language. An example of the latter is C++, which incorporates OO concepts into the popular C programming language.

An **object** typically has two components: state (value) and behavior (operations). It can have a *complex data structure* as well as *specific operations* defined by the programmer. Objects in an OOPL exist only during program execution; therefore, they are called *transient objects.* An OO database can extend the existence of objects so that they are stored permanently in a database, and hence the objects become *persistent objects* that exist beyond program termination and can be retrieved later and shared by other programs. In other words, OO databases store persistent objects permanently in secondary storage, and allow the sharing of these objects among multiple programs and applications. This requires the incorporation of other well-known features of database management systems, such as indexing mechanisms to efficiently locate the objects, concurrency control to allow object sharing among concurrent programs, and recovery from failures. An OO database system will typically interface with one or more OO programming languages to provide persistent and shared object capabilities.

The internal structure of an object in OOPLs includes the specification of **instance variables**, which hold the values that define the internal state of the object. An instance variable is similar to the concept of an *attribute* in the relational model, except that instance variables may be encapsulated within the object and thus are not necessarily visible to external users. Instance variables may also be of arbitrarily complex data types. Object-oriented systems allow definition of the operations or functions (behavior) that can be applied to objects of a particular type. In fact, some OO models insist that all operations a user can apply to an object must be predefined.

This forces a *complete encapsulation* of objects. This rigid approach has been relaxed in most OO data models for two reasons. First, database users often need to know the attribute names so they can specify selection conditions on the attributes to retrieve specific objects. Second, complete encapsulation implies that any simple retrieval requires a predefined operation, thus making ad hoc queries difficult to specify onthe fly.

To encourage encapsulation, an operation is defined in two parts. The first part, called the *signature* or *interface* of the operation, specifies the operation name and arguments (or parameters). The second part, called the *method* or *body*, specifies the *implementation* of the operation, usually written in some general-purpose programming language. Operations can be invoked by passing a *message* to an object, which includes the operation name and the parameters. The object then executes the method for that operation. This encapsulation permits modification of the internal structure of an object, as well as the implementation of its operations,

without the need to disturb the external programs that invoke these operations.

Hence, encapsulation provides a form of data and operation independence. Another key concept in OO systems is that of type and class hierarchies and *inheritance.* This permits specification of new types or classes that inherit much of their structure and/or operations from previously defined types or classes. This makes it easier to develop the data types of a system incrementally, and to *reuse* existing type definitions when creating new types of objects. One problem in early OO database systems involved representing *relationships* among objects. The insistence on complete encapsulation in early OO data models led to the argument that relationships should not be explicitly represented, but should instead be described by defining appropriate methods that locate related objects. However, this approach does not work very well for complex databases with many relationships because it is useful to identify these relationships and make them visible to users. The ODMG object database standard has recognized this need and it explicitly represents binary relationships via a pair of *inverse references*.

Another OO concept is *operator overloading,* which refers to an operation's ability to be applied to different types of objects; in such a situation, an *operation name* may refer to several distinct *implementations,* depending on the type of object it is applied to. This feature is also called *operator polymorphism.* For example, an operation to calculate the area of a geometric object may differ in its method (implementation), depending on whether the object is of type triangle, circle, or rectangle. This may require the use of *late binding* of the operation name to the appropriate method at runtime, when the type of object to which the operation is applied becomes known. In the next several sections, we discuss in some detail the main characteristics of object databases. The types for complex- structured objects are specified via type constructors; encapsulation and persistence; and presents inheritance concepts. Some additional OO concepts, and gives a summary of all the OO concepts that we introduced. In we show how some of these concepts have been incorporated into the SQL:2008 standard for relational databases. Then we show how these concepts are realized in the ODMG 3.0 object database standard.

5.1.5 TYPE CONSTRUCTORS

Another feature of an ODMS (and ODBs in general) is that objects and literals may have a *type structure* of *arbitrary complexity* in order to contain all of the necessary information that describes the object or literal. In contrast, in traditional database systems, information about a complex object is often *scattered* over many relations or records, leading to loss of direct correspondence between a real-world object and its database representation. In ODBs, a complex type may be constructed from other types by *nesting* of **type constructors**. The three most basic constructors are atom, struct (or tuple), and collection.

One type constructor has been called the **atom** constructor, although this term is not used in the latest object standard. This includes the basic built-in data types of the object model, which are similar to the basic types in many programming languages: integers, strings, floating point numbers, enumerated types, Booleans, and so on. They are called **single-valued** or **atomic** types, since each value of the type is considered an atomic (indivisible) single value.

A second type constructor is referred to as the **struct** (or **tuple**) constructor. This can create standard structured types, such as the tuples (record types) in the basic relational model. A structured type is made up of several components, and is also sometimes referred to as a *compound* or *composite* type. More accurately, the struct constructor is not considered to be a type, but rather a **type generator**, because many different structured types can be created. For example, two different structuredtypes that can be created are: struct Name<FirstName: string, MiddleInitial: char, LastName: string>, and struct CollegeDegree<Major: string, Degree: string, Year: date>. To create complex nested type structures in the object model, the *collection* type constructors are needed, which we discuss next. Notice that the type constructors *atom* and *struct* are the only ones available in the original (basic) relational model.

Collection (or *multivalued*) type constructors include the **set(T)**, **list(T)**, **bag(T)**, **array(T)**, and **dictionary(K,T)** type constructors. These allow part of an object or literal value to include a collection of other objects or values when needed. These constructors are also considered to be **type generators** because many different types can be created. For example, set(*string*), set(*integer*), and set(*Employee*) are three different types that can be created from the *set* type constructor. All the elements in a particular collection value must be of the same type. For example, all values in a collection of type set(*string*) must be string values. The *atom constructor* is used to represent all basic atomic values, such as integers, real numbers, character strings, Booleans, and any other basic data types that the system supports directly. The *tuple constructor* can create structured values and objects of the form <*a*1:*i*1, *a*2:*i*2, ..., *an*:*in*>, where each *aj* is an attribute name10 and each *ij* is a value or an OID. The other commonly used constructors are collectively referred to as collection types, but have individual differences among them. The **set constructor** will create objects or literals that are a set of *distinct* elements {*i*1, *i*2, ..., *in*}, all of the same type. The **bag constructor** (sometimes called a *multiset*) is similar to a set except that the elements in a bag *need not be distinct*. The **list constructor** will create an *ordered list* [*i*1, *i*2, ..., *in*] of OIDs or values of the same type. A list is similar to a **bag** except that the elements in a list are *ordered*, and hence we can refer to the first, second, or *j*th element. The **array constructor** creates a single-dimensional array of elements of the same type. The main difference between array and list is that a list can have an arbitrary number of elements whereas an array typically has a maximum size.

Finally, the **dictionary constructor** creates a collection of two tuples (*K*, *V*), where the value of a key *K* can be used to retrieve the corresponding value *V*. The main characteristic of a collection type is that its objects or values will be a *collection of objects or values of the same type* that may be unordered (such as a set or a bag) or ordered (such as a list or an array). The **tuple** type constructor is often called a **structured type**, since it corresponds to the **struct** construct in the C and C++ programming languages.

An **object definition language** (**ODL**) that incorporates the preceding type constructors can be used to define the object types for a particular database application. In this we will describe the standard ODL of ODMG, but first we introduce the concepts gradually in this section using a simpler notation. The type constructors can be used to define the *data structures* for an OO *database schema.* Figure shows how we may declare EMPLOYEE and DEPARTMENT types.

In Figure, the attributes that refer to other objects—such as Dept of EMPLOYEE or Projects of DEPARTMENT—are basically OIDs thatserve as **references** to other objects to represent *relationships* among the objects. For example, the attribute Dept of EMPLOYEE is of type DEPARTMENT, and hence is used to refer to a specific DEPARTMENT object (the DEPARTMENT object where the employee works). The value of such an attribute would be an OID for a specific DEPARTMENTobject. A binary relationship can be represented in one direction, or it can have an *inverse reference.* The latter representation makes it easy to traverse the relationship in both directions. For example, in Figure the attribute Employees of DEPARTMENT has as its value a *set of references*(that is, a set of OIDs) to objects of type EMPLOYEE; these are the employees who work for the DEPARTMENT. The inverse is the referenceattribute Dept of EMPLOYEE.

```
define type EMPLOYEE
    tuple (  Fname:       string;
             Minit:       char;
             Lname:       string;
             Ssn:         string;
             Birth_date:  DATE;
             Address:     string;
             Sex:         char;
             Salary:      float;
             Supervisor:  EMPLOYEE;
             Dept:        DEPARTMENT;

define type DATE
    tuple (  Year:        integer;
             Month:       integer;
             Day:         integer; );

define type DEPARTMENT
    tuple (  Dname:       string;
             Dnumber:     integer;
             Mgr:         tuple (  Manager:     EMPLOYEE;
                                   Start_date:  DATE; );
             Locations:   set(string);
             Employees:   set(EMPLOYEE);
             Projects:    set(PROJECT); );
```

Figure5.1.1 Specifying the object type EMPLOYEE, DATE and DEPARTMENT using type constructors.

5.1.6 ENCAPSULATION OF OPERATIONS

Encapsulation of Operations The concept of *encapsulation* is one of the main characteristics of OO languages and systems. It is also related to the concepts of *abstract data types* and *information hiding* in programming languages. In traditional database models and systems this concept was not applied, since it is customary to make the structure of database objects visible to users and external programs. In these traditional models, a number of generic database operations are applicable to objects *of all types*. For example, in the relational model, the operations for selecting, inserting, deleting, and modifying tuples are generic and may be applied to *any relation* in the database. The relation and its attributes are visible to users and to external programs that access the relation by using these operations. The concepts of encapsulation is applied to database objects in ODBs by defining the **behavior** of a type of object based on the **operations** that can be externally applied to objects of that type. Some operations may be used to create (insert) or destroy (delete) objects; other operations may update the object state; and others may be used to retrieve parts of the object state or to apply some calculations. Still other operations may perform a combination of retrieval, calculation, and update. In general, the **implementation** of an operation can be specified ina *general-purpose programming language* that provides flexibility and power in defining the operations. The external users of the object are only made aware of the **interface** of the operations, which defines the name and arguments (parameters) of each operation. The implementation is hidden from the external users; it includes the definition of any hidden internal data structures of the object and the implementation of the operations that access these structures. The interface part of an operation issometimes called the **signature**, and the operation implementation is sometimes called the **method**.

For database applications, the requirement that all objects be completely encapsulated is too stringent. One way to relax this requirement is to divide the structure of an object into **visible** and **hidden** attributes (instance variables). Visible attributes can be seen by and are directly accessible to the database users and programmers via the query language. The hidden attributes of an object are completely encapsulated and can be accessed only through predefined operations. Most ODMSs employ high-level query languages for accessing visible attributes. In this we will describe the OQL query language that is proposed as a standard query language for ODBs.

The term **class** is often used to refer to a type definition, along with the definitions of the operations for that type. Figure 3.1.2 shows how the type definitions in Figure 3.1.1 can be extended with operations to define classes. A number of operations are declared for each class, and the signature (interface) of each operation is included in the class definition.

```
define class EMPLOYEE
    type tuple (  Fname:          string;
                  Minit:          char;
                  Lname:          string;
                  Ssn:            string;
                  Birth_date:     DATE;
                  Address:        string;
                  Sex:            char;
                  Salary:         float;
                  Supervisor:     EMPLOYEE;
                  Dept:           DEPARTMENT;  );
    operations    age:            integer;
                  create_emp:     EMPLOYEE;
                  destroy_emp:    boolean;
end EMPLOYEE;

define class DEPARTMENT
    type tuple (  Dname:          string;
                  Dnumber:        integer;
                  Mgr:            tuple ( Manager:      EMPLOYEE;
                                          Start_date:   DATE;  );
                  Locations:      set (string);
                  Employees:      set (EMPLOYEE);
                  Projects        set(PROJECT);  );
    operations    no_of_emps:     integer;
                  create_dept:    DEPARTMENT;
                  destroy_dept:   boolean;
                  assign_emp(e:   EMPLOYEE): boolean;
                  (* adds an employee to the department *)
                  remove_emp(e: EMPLOYEE): boolean;
                  (* removes an employee from the department *)
end DEPARTMENT;
```

Figure 5.1.2 Adding operations to the definitions of EMPLOYEE and DEPARTMENT

A method (implementation) for each operation must be defined elsewhere using a programming language. Typical operations include the **object constructor** operation (often called *new*), which is used to create a new object, and the **destructor** operation, which is used to destroy (delete) an object. A number of **object modifier** operations can also be declared to modify the states (values) of various attributes of an object. Additional operations can **retrieve** information about the object.

An operation is typically applied to an object by using the **dot notation**. For example, if d is a reference to a DEPARTMENT object, we can invoke an operation such as no_of_emps by writing d.no_of_emps. Similarly, by writing d.destroy_dept, the object referenced by d is destroyed (deleted). The only exception is the constructor operation, which returns a reference to a new DEPARTMENT object. Hence, it is customary in some OO models to have a default name for the constructor operation that is the name of the class itself, although this was not used in Figure. The dot notation is also used to refer to attributes of an object—forexample, by writing d.Dnumber or d.Mgr_Start_date.

5.1.7 METHODS AND PERSISTENCE

Specifying Object Persistence via Naming and Reachability An ODBS is often closely coupled with an object-oriented programming language (OOPL). The OOPL is used to specify the method (operation)

implementations as well as other application code. Not all objects are meant to be stored permanently in the database.

Transient objects exist in the executing program and disappear once the program terminates. **Persistent objects** are stored in the database and persist after program termination. The typical mechanisms for making an object persistent are *naming* and *reachability.*

The **naming mechanism** involves giving an object a unique persistent name within a particular database. This persistent **object name** can be given via a specific statement or operation in the program, as shown in Figure. The named persistent objects are used as **entry points** to the database through which users and applications can start their database access. Obviously, it is not practical to give names to all objects in a large database that includes thousands of objects, so most objects are made persistent by using the second mechanism, called **reachability**. The reachability mechanism works by making the object reachable from some other persistent object. An object *B* is said to be **reachable** from an object *A* if a sequence of references in the database lead from object *A* to object *B.*

If we first create a named persistent object *N*, whose state is a *set* (or possibly a *bag*) of objects of some class *C*, we can make objects of *C* persistent by *adding them* to the set, thus making them reachable from *N*. Hence, *N* is a named object that defines a **persistent collection** of objects of class *C*. In the object model standard, *N* is called the **extent** of *C*.

For example, we can define a class DEPARTMENT_SET (see Figure) whose objects are of type set(DEPARTMENT). We can create an object of type DEPARTMENT_SET, and give it a persistent name ALL_DEPARTMENTS, as shown in Figure. Any DEPARTMENT object that is added to the set of ALL_DEPARTMENTS by using the add_dept operation becomes persistent by virtue of its being reachable from ALL_DEPARTMENTS. As we will see in Section, the ODMG ODL standard gives the schema designer the option of naming an extent as part of class definition. Notice the difference between traditional database models and ODBs in this respect.

```
define class DEPARTMENT_SET
    type set (DEPARTMENT);
    operations   add_dept(d: DEPARTMENT):    boolean;
                 (* adds a department to the DEPARTMENT_SET object *)
                    remove_dept(d: DEPARTMENT): boolean;
                 (* removes a department from the DEPARTMENT_SET object *)
                    create_dept_set:        DEPARTMENT_SET;
                    destroy_dept_set:       boolean;
end DEPARTMENT_SET;
...
persistent name ALL_DEPARTMENTS: DEPARTMENT_SET;
(* ALL_DEPARTMENTS is a persistent named object of type DEPARTMENT_SET *)
...
d:= create_dept;
(* create a new DEPARTMENT object in the variable d *)
...
b:= ALL_DEPARTMENTS.add_dept(d);
(* make d persistent by adding it to the persistent set ALL_DEPARTMENTS *)
```

Figure 5.1.3 creating persistent objects by naming and reachability.

In traditional database models, such as the relational model, *all* objects are assumed to be persistent. Hence, when a table such as EMPLOYEE is created in a relational database, it represents both the *type declaration* for EMPLOYEE and a *persistent set* of *all* EMPLOYEE records (tuples). In the OO approach, a class declaration of EMPLOYEE specifies only the type and operations for a class of objects. The user must separately define a persistent object of type set(EMPLOYEE) or bag(EMPLOYEE) whose value is the *collection of references* (OIDs) to all persistent EMPLOYEE objects, if this is desired, as shown in Figure. This allows transient and persistent objects to follow the same type and class declarations of the ODL and the OOPL. In general, it is possible to define several persistent collections for the same class definition, if desired.

5.1.8 TYPE AND CLASS HIERARCHIES

Simplified Model for Inheritance Another main characteristic of ODBs is that they allow type hierarchies and inheritance. We use a simple OO model in this section a model in which attributes and operations are treated uniformly—since both attributes and operations can be inherited. In this section, we will discuss the inheritance model of the ODMG standard, which differs from the model discussed here because it distinguishes between *two types of inheritance*. Inheritance allows the definition of new types based on other predefined types, leading to a **type** (or **class**) **hierarchy**.

Type is defined by assigning it a type name, and then defining a number of attributes (instance variables) and operations (methods) for the type. In the simplified model we use in this section, the attributes and operations are together called *functions*, since attributes resemble functions with zero arguments. A function name can be used to refer to thevalue of an attribute or to refer to the resulting value of an operation (method).We use the term **function** to refer to both attributes *and*

operations, since they are treated similarly in a basic introduction to inheritance.

A type in its simplest form has a **type name** and a list of visible (*public*) **functions**. When specifying a type in this section, we use the following format, which does not specify arguments of functions, to simplify the discussion: TYPE_NAME: function, function,..., function For example, a type that describes characteristics of a PERSON may be defined as follows: PERSON: Name, Address, Birth_date, Age, Ssn In thePERSON type, the Name, Address, Ssn, and Birth_date functions can be implemented as stored attributes, whereas the Age function can be implemented as an operation that calculates the Age from the value of the Birth_date attribute and the current date.

The concept of **subtype** is useful when the designer or user must create a new type that is similar but not identical to an already defined type. The subtype then inherits all the functions of the predefined type, which is referred to as the **supertype**. For example, suppose that we want to define two new types EMPLOYEE and STUDENT as follows:

EMPLOYEE: Name, Address, Birth_date, Age, Ssn, Salary, Hire_date, Seniority STUDENT: Name, Address, Birth_date, Age, Ssn, Major, Gpa Since both STUDENT and EMPLOYEE include all the functions defined for PERSON plus some additional functions of their own, we can declare them to be **subtypes** of PERSON. Each will inherit the previously defined functions of PERSON—namely, Name, Address, Birth_date, Age, and Ssn. For STUDENT, it is only necessary to define the new (local) functions Major and Gpa, which are not inherited. Presumably, Major can be defined as a stored attribute, whereas Gpa may be implemented as an operation that calculates the student's grade point average by accessing the Grade values that are internally stored (hidden) within each STUDENT object as *hidden attributes.*
For EMPLOYEE, the Salary and Hire_date functions may be stored attributes, whereas
Seniority may be an operation that calculates Seniority from the value of Hire_date.

Therefore, we can declare EMPLOYEE and STUDENT as follows:

EMPLOYEE **subtype-of** PERSON: Salary, Hire_date, Seniority
STUDENT **subtype-of** PERSON: Major, Gpa

In general, a subtype includes *all* of the functions that are defined for its super type plus some additional functions that are *specific* only to the sub type. Hence, it is possible to generate a **type hierarchy** to showthe supertype/subtype relationships among all the types declared in the system.

As another example, consider a type that describes objects in plane geometry, which may be defined as follows: GEOMETRY_OBJECT:

Shape, Area, Reference_point For the GEOMETRY_OBJECT type, Shape is implemented as an attribute (its domain can be an enumerated type with values ‘triangle’, ‘rectangle’, ‘circle’, and so on), and Area is a method that is applied to calculate the area. Reference_point specifies the coordinates of a point that determines the object location. Now suppose that we want to define a number of subtypes for the GEOMETRY_OBJECT type, as follows:

RECTANGLE **subtype-of** GEOMETRY_OBJECT: Width, Height
TRIANGLE S **subtype-of** GEOMETRY_OBJECT: Side1, Side2, Angle
CIRCLE **subtype-of** GEOMETRY_OBJECT: Radius

Notice that the Area operation may be implemented by a different method for each subtype, since the procedure for area calculation is different for rectangles, triangles, and circles. Similarly, the attribute Reference_point may have a different meaning for each subtype; it might be the center point for RECTANGLE and CIRCLE objects, and the vertex point between the two given sides for a TRIANGLE object. Notice that type definitions describe objects but *do not* generate objects on their own. When an object is created, typically it belongs to one or more of these types that have been declared. For example, a circle object is of type CIRCLE and GEOMETRY_OBJECT (by inheritance). Each object also becomes a member of one or more persistent collections of objects (or extents), which are used to group together collections of objects that are persistently stored in the database. **Constraints on Extents Corresponding to a Type Hierarchy** in most ODBs, an **extent** is definedto store the collection of persistent objects for each type or subtype.

In this case, the constraint is that every object in an extent that corresponds to a subtype must also be a member of the *extent* that corresponds to its supertype. Some OO database systems have a predefined system type (called the ROOT class or the OBJECT class) whose extent contains all the objects in the system. Classification then proceeds by assigning objects into additional subtypes that are meaningful to the application, creating a **type hierarchy** (or **class hierarchy**) for the system. All extents for system- and user-defined classes are subsets of the extent corresponding to the class OBJECT, directly or indirectly. In the ODMG model, the user may or may not specify an extent for each class (type), depending on the application.

An extent is a named persistent object whose value is a **persistent collection** that holds a collection of objects of the same type that are stored permanently in the database. The objects can be accessed andshared by multiple programs. It is also possible to create a **transient collection**, which exists temporarily during the execution of a program butis not kept when the program terminates. For example, a transient collection may be created in a program to hold the result of a query that selects some objects from a persistent collection and copies those objects into the transient collection. The program can then manipulate the objects

in the transient collection, and once the program terminates, the transient collection ceases to exist.

In general, numerous collections—transient or persistent—may contain objects of the same type. The inheritance model discussed in this section is very simple. As we will see in the ODMG model distinguishes between type inheritance—called *interface inheritance* and denoted by a colon (:)—and the *extent inheritance* constraint— denoted by the keyword EXTEND.

5.1.9 INHERITANCE

SQL has rules for dealing with **type inheritance** (specified via the **UNDER** keyword). In general, both attributes and instance methods (operations) are inherited. The phrase **NOT FINAL** must be included in a UDT if subtypes are allowed to be created under that UDT (see Figure 11.4(a) and (b), where PERSON_TYPE, STUDENT_TYPE, and EMPLOYEE_TYPE are declared to be NOT FINAL). Associated with type inheritance are the rules for overloading of function implementations and for resolution of function names. These inheritance rules can be summarized as follows:

- All attributes are inherited.
- The order of supertypes in the UNDER clause determines the inheritance hierarchy.
- An instance of a subtype can be used in every context in which a supertype instance is used.
- A subtype can redefine any function that is defined in its supertype, with the restriction that the signature be the same.
- When a function is called, the best match is selected based on the types of all arguments.
- For dynamic linking, the runtime types of parameters is considered.

Consider the following examples to illustrate type inheritance, which are illustrated in Figure. Suppose that we want to create two subtypes of PERSON_TYPE: EMPLOYEE_TYPE and STUDENT_TYPE. In addition, we also create a subtype MANAGER_TYPE that inherits all the attributes (and methods) of EMPLOYEE_TYPE but has an additional attribute DEPT_MANAGED. These subtypes are shown in Figure.

In general, we specify the local attributes and any additional specific methods for the subtype, which inherits the attributes and operations of its supertype. Another facility in SQL is **table inheritance** via the supertable/subtable facility. This is also specified using the keyword **UNDER.** Here, a new record that is inserted into a subtable, say

the MANAGER table, is also inserted into its supertables EMPLOYEE and PERSON. Notice that when a record is inserted in MANAGER, we must provide values for all its inherited attributes. INSERT, DELETE, and UPDATE operations are appropriately propagated.

In the ODMG object model, two types of inheritance relationships exist: behavior only inheritance and state plus behavior inheritance. **Behavior inheritance** is also known as *ISA* or *interface inheritance*, and is specified by the colon (:) notation.30 Hence, in the ODMG object model, behavior inheritance requires the supertype to be an interface,whereas the subtype could be either a class or another interface.

The other inheritance relationship, called **EXTENDS inheritance**, is specified by the keyword **extends**. It is used to inherit both state and behavior strictly among classes, so both the supertype and the subtype must be classes. Multiple inheritance via extends is not permitted. However, multiple inheritance is allowed for behavior inheritance via the colon (:) notation. Hence, an interface may inherit behavior from several other interfaces. A class may also inherit behavior from several interfaces via colon (:) notation, in addition to inheriting behavior and state from *at most one* other class via extends.

5.1.10 COMPLEX OBJECTS

Unstructured complex object:

- These is provided by a DBMS and permits the storage and retrieval of large objects that are needed by the database application.
- Typical examples of such objects are bitmap images and long text strings (such as documents); they are also known as binary largeobjects, or BLOBs for short.
- This has been the standard way by which Relational DBMSs havedealt with supporting complex objects, leaving the operations on those objects outside the RDBMS.

Structured complex object:

This differs from an unstructured complex object in that the object's structure is defined by repeated application of the type constructors provided by the OODBMS. Hence, the object structure is defined and known to the OODBMS. The OODBMS also defines methodsor operations on it.

5.1.11 OBJECT-ORIENTED DBMS

Object oriented databases or object databases incorporate the object data model to define data structures on which database operations such as CRUD can be performed. They store objects rather than data such as integers and strings. The relationship between various data is implicit tothe object and manifests as object attributes and methods. Object database management systems extend the object programming language with transparently persistent data, concurrency control, data recovery, associative queries, and other database capabilities. The Object-Oriented Database System Manifesto by Malcolm Atkinson mandates that an object oriented database system should satisfy two criteria: it should be a DBMS, and it should be an object-oriented system Thus OODB implements OO concepts such as object identity ,polymorphism, encapsulation and inheritance to provide access to persistent objects using any OO-programming language The tight integration between object orientation and databases provides programmers a unified environment when dealing with complex data such as 2D and 3D graphics. Object oriented databasesare designed to work well with object oriented programming languages such as Python, Java, and Objective-C.

5.1.12 LANGUAGES AND DESIGN

Developed by ODMG, Object Query Language allows SQL-like queries to be performed on a OODB. Like SQL, it is a declarativelanguage. Based loosely on SQL, OQL includes additional language constructs which allow for object oriented design such as operation invocation and inheritance. Query Structures look very similar in SQL and OQL but the results returned are different. Example: OQL query to obtain Voter names who are from the state of Colorado

```
Select distinct v.name
From voters v
Where v.state = “Colorado”
```

Voter Id	Name	State
V1	George Love	Colorado
V2	Winnie the Pooh	Florida
V3	John Lewis Hall	Colorado

Result from SQL table with rows

Name
George Love
John Lewis Hall

Result from OQL collection of objects

String	String
George Love	John Lewis Hall

- More example of OQL with integration to OO Language:
- Create objects as in OO languages and then make them persistent using the set() method on the database.

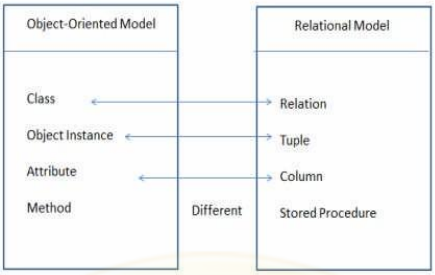
```
Person p1 = new Person(“Pikes Peak", 78);
```

db.set(p1);

- Retrieve by age (null default for string)

Person p = new Person (null, 35);

ObjectSet<Person> result = db.get(p);



5.1.13 ODMG MODEL

The **ODMG object model** is the data model upon which the object definition language (ODL) and object query language (OQL) are based. It is meant to provide a standard data model for object databases, just as SQL describes a standard data model for relational databases. It alsoprovides a standard terminology in a field where the same terms were sometimes used to describe different concepts. Many of the concepts in the ODMG model have already been discussed in Section, and we assume the reader has read this section. We will point out whenever the ODMG terminology differs from that used in Section. **Objects and Literals.**Objects and literals are the basic building blocks of the object model. The main difference between the two is that an object has both an object identifier and a **state** (or current value), whereas a literal has a value(state) but *no object identifier*. In either case, the value can have a complex structure. The object state can change over time by modifying theobject value. A literal is basically a constant value, possibly having a complex structure, but it does not change. An **object** has five aspects: identifier, name, lifetime, structure, and creation.

1. The **object identifier** is a unique system-wide identifier (or **Object_id**). Every object must have an object identifier.

2. Some objects may optionally be given a unique **name** within a particular ODMS—this name can be used to locate the object, and the system should return the object given that name. Obviously, not all individual objects will have unique names. Typically, a few objects, mainly those that hold collections of objects of a particular object type—such as *extents*—will have a name. These names are used as **entry points** to the database; that is, by locating these objects by their unique name, the user can then locate other objects that are referenced from these objects. Other important objects in the application may also

have unique names, and it is possible to give *more than one* name to an object. All names within a particular ODMS must be unique.

3. The **lifetime** of an object specifies whether it is a *persistent object* (that is, a database object) or *transient object* (that is, an object in an executing program that disappears after the program terminates). Lifetimes are independent of types—that is, some objects of a particular type may be transient whereas others may be persistent.

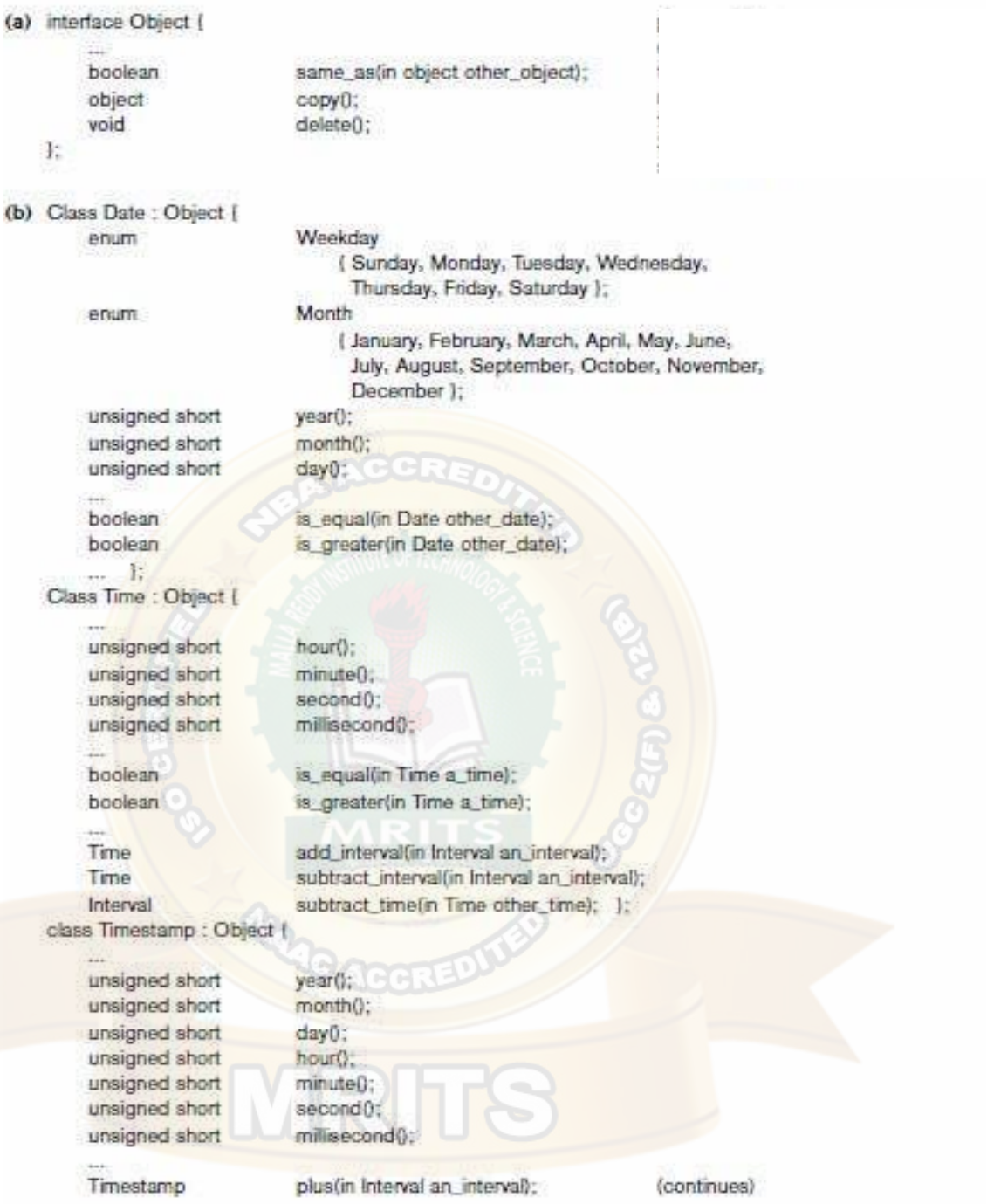
4. The **structure** of an object specifies how the object is constructed by using the type constructors. The structure specifies whether an object is *atomic* or not. An **atomic object** refers to a single object thatfollows a user-defined type, such as Employee or Department. If an object is not atomic, then it will be composed of other objects. For example, a *collection object* is not an atomic object, since its state will be a collection of other objects. The term *atomic object* is different from how we defined the *atom constructor* in Section, which referred to all values of built-in data types. In the ODMG model, an atomic object is any *individual user-defined object*. All values of the basic built-in data types are considered to be *literals*.

5. Object **creation** refers to the manner in which an object can be created. This is typically accomplished via an operation *new* for a special Object_Factory interface. We shall describe this in more detail later in this section.

In the object model, a **literal** is a value that *does not have* an object identifier. However, the value may have a simple or complex structure. There are three types of literals: atomic, structured, and collection.

Atomic literals correspond to the values of basic data types and are predefined. The basic data types of the object model include long, short, and unsigned integer numbers (these are specified by the keywords **long**, **short**, **unsigned long**, and **unsigned short** in ODL), regular and double precision floating point numbers (**float**, **double**), Boolean values (**boolean**), single characters (**char**), character strings (**string**), and enumeration types (**enum**), among others.

Structured literals correspond roughly to values that are constructed using the tuple constructor described in Section. The built-in structured literals include Date, Interval, Time, and Timestamp. Additional user-defined structured literals can be defined as needed by each application. User-defined structures are created using the **STRUCT** keyword in ODL, as in the C and C++ programming languages.



```
(a) interface Object {
        ...
        boolean             same_as(in object other_object);
        object              copy();
        void                delete();
    };

(b) Class Date : Object {
        enum                Weekday
                                { Sunday, Monday, Tuesday, Wednesday,
                                  Thursday, Friday, Saturday };
        enum                Month
                                { January, February, March, April, May, June,
                                  July, August, September, October, November,
                                  December };
        unsigned short      year();
        unsigned short      month();
        unsigned short      day();
        ...
        boolean             is_equal(in Date other_date);
        boolean             is_greater(in Date other_date);
        ...   };
    Class Time : Object {
        ...
        unsigned short      hour();
        unsigned short      minute();
        unsigned short      second();
        unsigned short      millisecond();
        ...
        boolean             is_equal(in Time a_time);
        boolean             is_greater(in Time a_time);
        ...
        Time                add_interval(in Interval an_interval);
        Time                subtract_interval(in Interval an_interval);
        Interval            subtract_time(in Time other_time);   };
    class Timestamp : Object {
        ...
        unsigned short      year();
        unsigned short      month();
        unsigned short      day();
        unsigned short      hour();
        unsigned short      minute();
        unsigned short      second();
        unsigned short      millisecond();
        ...
        Timestamp           plus(in Interval an_interval);          (continues)
```

Figure 5.1.4 Overview of the interface definitions for part of the ODMG object mode.
a) The basic object interface inherited by all objects b) Some standard interfaces for structured literals

```
    Timestamp           minus(in Interval an_interval);
    boolean             is_equal(in Timestamp a_timestamp);
    boolean             is_greater(in Timestamp a_timestamp);
    ... };
    class Interval :    Object {
    unsigned short      day();
    unsigned short      hour();
    unsigned short      minute();
    unsigned short      second();
    unsigned short      millisecond();
    ...
    Interval            plus(in Interval an_interval);
    Interval            minus(in Interval an_interval);
    Interval            product(in long a_value);
    Interval            quotient(in long a_value);
    boolean             is_equal(in interval an_interval);
    boolean             is_greater(in interval an_interval);
    ... };
```

(c)

```
interface Collection : Object {
    ...
    exception           ElementNotFound{ Object element; };
    unsigned long       cardinality();
    boolean             is_empty();
    ...
    boolean             contains_element(in Object element);
    void                insert_element(in Object element);
    void                remove_element(in Object element)
                            raises(ElementNotFound);
    iterator            create_iterator(in boolean stable);
    ... };
interface Iterator {
    exception           NoMoreElements();
    ...
    boolean             at_end();
    void                reset();
    Object              get_element() raises(NoMoreElements);
    void                next_position() raises(NoMoreElements);
    ... };
interface set : Collection {
    set                 create_union(in set other_set);
    ...
    boolean             is_subset_of(in set other_set);
    ... };
interface bag : Collection {
    unsigned long       occurrences_of(in Object element);
```

Figure 5.1.4 Overview of the interface definitions for part of the ODMG object mode.

a) The basic object interface inherited by all objects b) Some standard interfaces for structured literals

c) Interfaces for collections and iterators

```
    bag                 create_union(in Bag other_bag);
    ...   };
interface list : Collection {
    exception           Invalid_Index{unsigned_long index; };
    void                remove_element_at(in unsigned long index)
                            raises(InvalidIndex);
    Object              retrieve_element_at(in unsigned long index)
                            raises(InvalidIndex);
    void                replace_element_at(in Object element, in unsigned long index)
                            raises(InvalidIndex);
    void                insert_element_after(in Object element, in unsigned long index)
                            raises(InvalidIndex);
    ...
    void                insert_element_first(in Object element);
    ...
    void                remove_first_element() raises(ElementNotFound);
    ...
    Object              retrieve_first_element() raises(ElementNotFound);
    ...
    list                concat(in list other_list);
    void                append(in list other_list);
};
interface array     : Collection {
    exception           invalid_Index{unsigned_long index; };
    exception           Invalid_Size{unsigned_long size; };
    void                remove_element_at(in unsigned long index)
                            raises(InvalidIndex);
    Object              retrieve_element_at(in unsigned long index)
                            raises(InvalidIndex);
    void                replace_element_at(in unsigned long index, in Object element)
                            raises(InvalidIndex);
    void                resize(in unsigned long new_size)
                            raises(InvalidSize);
};
struct association { Object key; Object value; };
interface dictionary : Collection {
    exception           DuplicateName{string key; };
    exception           KeyNotFound{Object key; };
    void                bind(in Object key, in Object value)
                            raises(DuplicateName);
    void                unbind(in Object key) raises(KeyNotFound);
    Object              lookup(in Object key) raises(KeyNotFound);
    boolean             contains_key(in Object key);
};
```

Collection literals specify a literal value that is a collection of objects or values but the collection itself does not have an Object_id. The collections in the object model can be defined by the *type generators* **set**<*T*>, **bag**<*T*>, **list**<*T*>, and **array**<*T*>, where *T* is the type of objects or values in the collection.

Another collection type is **dictionary**<*K, V*>, which is a collection of associations <*K, V*>, where each *K* is a key (a unique search value) associated with a value *V*; this can be used to create an index on a collection of values *V*.

Figure gives a simplified view of the basic types and type generators of the object model. The notation of ODMG uses three concepts: interface, literal, and class. Following the ODMG terminology, we use the word **behavior** to refer to *operations* and **state** to refer to *properties* (attributes and relationships). An **interface** specifies only behavior of an object type and is typically **noninstantiable** (that is, no objects are created

corresponding to an interface). Although an interface may have state properties (attributes and relationships) as part of its specifications, these *cannot* be inherited from the interface. Hence, an interface serves to define operations that can be *inherited* by other interfaces, as well as by classes that define the user-defined objects for a particular application. A **class** specifies both state (attributes) and behavior (operations) of an object type, and is **instantiable**. Hence, database and application objects are typically created based on the user-specified class declarations that form a database schema. Finally, a **literal** declaration specifies state but no behavior. Thus, a literal instance holds a simple or complex structured value but hasneither an object identifier nor encapsulated operations. Figure is a simplified version of the object model. For the full specifications, see Cattell etc all. (2000).We will describe some of the constructs shown in Figure as we describe the object model. In the object model, all objects inherit the basic interface operations of Object, shown in Figure (a); these include operations such as copy (creates a new copy of the object), delete (deletes the object), and same_as (compares the object's identity to another object). In general, operations are applied to objects using the **dot notation**. For example, given an object *O*, to compare it with another object *P*, we write *O*.same_as(*P*) The result returned by this operation is Boolean and would be true if the identity of *P* is the same as that of *O*, andfalse otherwise. Similarly, to create a copy *P* of object *O*, we write = *O*.copy() An alternative to the dot notation is the **arrow notation:** *O*–>same_as(*P*) or *O*–>copy().

5.1.14 OBJECT DEFINITION LANGUAGES (ODL)

After our overview of the ODMG object model in the previous section, we now show how these concepts can be utilized to create an object database schema using the object definition language ODL.

The ODL is designed to support the semantic constructs of the ODMG object model and is independent of any particular programming language. Its main use is to create object specifications—that is, classes and interfaces. Hence, ODL is not a full programming language. A user can specify a database schema in ODL independently of any programming language, and then use the specific language bindings to specify how ODL constructs can be mapped to constructs in specific programming languages, such as C++, Smalltalk, and Java. We will give an overview of the C++ binding in Section. Figure (b) shows a possible object schema for part of the UNIVERSITY database. We will describe the concepts of ODLusing this example, and the one in Figure. The graphical notation for Figure (b) is shown in Figure (a) and can be considered as a variation of EER diagrams with the added concept of interface inheritance but without several EER concepts, such as categories (union types) and attributes of relationships. Figure shows one possible set of ODL class definitions for the UNIVERSITY database. In general, there may be several possible mappings from an object schema diagram (or EER schema diagram) into ODL classes. We will discuss these options further in Section. Figure

shows the straightforward way of mapping part of the UNIVERSITY database from Chapter. Entity types are mapped into ODL classes, and inheritance is done using **extends**. However, there is no direct way to map categories (union types) or to do multiple inheritance. In Figure the classes PERSON, FACULTY, STUDENT, and GRAD_STUDENT have the extents PERSONS, FACULTY, STUDENTS, and GRAD_STUDENTS, respectively. Both FACULTY and STUDENT **extends** PERSON and GRAD_STUDENT **extends** STUDENT. Hence, the collection of STUDENTS (and the collection of FACULTY) will be constrained to be a subset of the collection of PERSONs at any time. Similarly, the collection of GRAD_STUDENTs will be a subset of STUDENTs. At the same time, individual STUDENT and FACULTY objects will inherit the properties (attributes and relationships) and operations of PERSON, and individual GRAD_STUDENT objects will inherit those of STUDENT.

The classes DEPARTMENT, COURSE, SECTION, and CURR_SECTION in Figure are straightforward mappings of the corresponding entity types in Figure

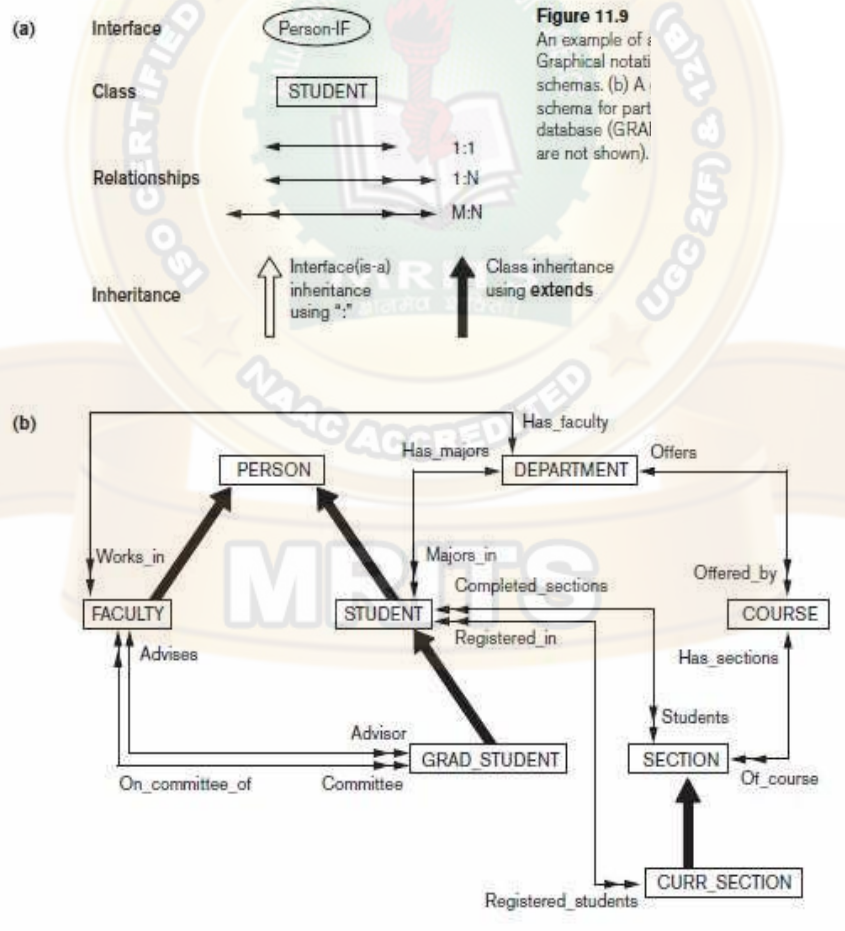
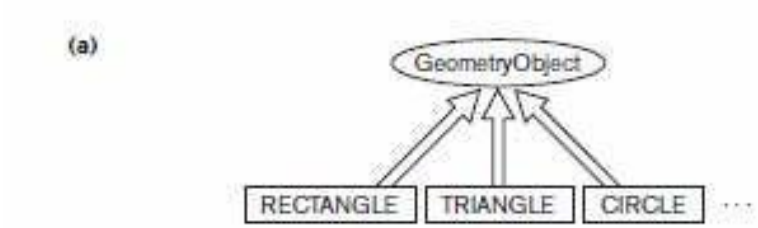


Figure 3.1.5 An example of a database schema a) Graphical notation for representing ODL schemas. b) A graphical object database schema for part of the UNIVERSITY database (GRADE and DEGREE classes are not shown 3.1.3(b)

```
class PERSON
(   extent      PERSONS
    key         Ssn )
(   attribute   struct Pname (  string  Fname,
                                string  Mname,
                                string  Lname )     Name;
    attribute   string                              Ssn;
    attribute   date                                Birth_date;
    attribute   enum Gender(M, F)                   Sex;
    attribute   struct Address (  short   No,
                                  string  Street,
                                  short   Apt_no,
                                  string  City,
                                  string  State,
                                  short   Zip )     Address;
    short       Age();   );
class FACULTY extends PERSON
(   extent      FACULTY )
(   attribute   string      Rank;
    attribute   float       Salary;
    attribute   string      Office;
    attribute   string      Phone;
    relationship  DEPARTMENT    Works_in inverse DEPARTMENT::Has faculty;
    relationship  set<GRAD_STUDENT> Advises inverse GRAD_STUDENT::Advisor;
    relationship  set<GRAD_STUDENT> On_committee_of inverse GRAD_STUDENT::Committee;
    void        give_raise(in float raise);
    void        promote(in string new rank); );
class GRADE
(   extent      GRADES )
(
    attribute   enum GradeValues{A,B,C,D,F,I, P} Grade;
    relationship  SECTION Section inverse SECTION::Students;
    relationship STUDENT Student inverse STUDENT::Completed_sections; );
class STUDENT extends PERSON
(   extent      STUDENTS )
(   attribute   string      Class;
    attribute   DEPARTMENT      Minors_in;
    relationship  DEPARTMENT Majors_in inverse DEPARTMENT::Has_majors;
    relationship  set<GRADE> Completed_sections inverse GRADE::Student;
    relationship  set<CURR_SECTION> Registered_in INVERSE CURR_SECTION::Registered_students;
    void        change_major(in string dname) raises(dname_not_valid);
    float       gpa();
    void        register(in short secno) raises(section_not_valid);
    void        assign_grade(in short secno; IN GradeValue grade)
                    raises(section_not_valid,grade_not_valid); );
```

Figure 3.1.6 Possible ODL schema for the UNIVERSITY database in figure 3.1.3(b)

```
class DEGREE
(   attribute     string          College;
    attribute     string          Degree;
    attribute     string          Year;     );
class GRAD_STUDENT extends STUDENT
(   extent        GRAD_STUDENTS   )
(   attribute     set<DEGREE>         Degrees;
    relationship  FACULTY Advisor inverse FACULTY::Advises;
    relationship  set<FACULTY>    Committee inverse FACULTY::On_committee_of;
    void          assign_advisor(in string Lname; in string Fname)
                      raises(faculty_not_valid);
    void          assign_committee_member(in string Lname; in string Fname)
                      raises(faculty_not_valid);  );
class DEPARTMENT
(   extent        DEPARTMENTS
    key           Dname )
(   attribute     string          Dname;
    attribute     string          Dphone;
    attribute     string          Doffice;
    attribute     string          College;
    attribute     FACULTY         Chair;
    relationship  set<FACULTY> Has_faculty inverse FACULTY::Works_in;
    relationship  set<STUDENT> Has_majors inverse STUDENT::Majors_in;
    relationship  set<COURSE> Offers inverse COURSE::Offered_by;  );
class COURSE
(   extent        COURSES
    key           Cno )
(   attribute     string          Cname;
    attribute     string          Cno;
    attribute     string          Description;
    relationship  set<SECTION> Has_sections inverse SECTION::Of_course;
    relationship  <DEPARTMENT> Offered_by inverse DEPARTMENT::Offers; );
class SECTION
(   extent        SECTIONS )
(   attribute     short           Sec_no;
    attribute     string          Year;
    attribute     enum Quarter{Fall, Winter, Spring, Summer}
                      Qtr;
    relationship  set<GRADE> Students inverse GRADE::Section;
    relationship  course Of_course inverse COURSE::Has_sections;  );
class CURR_SECTION extends SECTION
(   extent        CURRENT_SECTIONS )
(   relationship  set<STUDENT> Registered_students
                      inverse STUDENT::Registered_in
    void          register_student(in string Ssn)
                      raises(student_not_valid, section_full);  );
```




```
(b) interface GeometryObject
    (   attribute     enum        Shape{RECTANGLE, TRIANGLE, CIRCLE, ... }
                                      Shape;
        attribute     struct      Point (short x, short y) Reference_point;
        float         perimeter();
        float         area();
        void          translate(in short x_translation; in short y_translation);
        void          rotate(in float angle_of_rotation); };
    class RECTANGLE : GeometryObject
    (   extent        RECTANGLES  )
    (   attribute     struct      Point (short x, short y) Reference_point;
        attribute     short       Length;
        attribute     short       Height;
        attribute     float       Orientation_angle; };
    class TRIANGLE : GeometryObject
    (   extent        TRIANGLES  )
    (   attribute     struct      Point (short x, short y) Reference_point;
        attribute     short       Side_1;
        attribute     short       Side_2;
        attribute     float       Side1_side2_angle;
        attribute     float       Side1_orientation_angle; };
    class CIRCLE : GeometryObject
    (   extent        CIRCLES  )
    (   attribute     struct      Point (short x, short y) Reference_point;
        attribute     short       Radius; };
```

Figure 3.1.7 An illustration of interface inheritance via ":". (a) Graphical schema representation
(b) Corresponding interface and class definitions in ODL.

However, the class GRADE requires some explanation. The GRADE class corresponds to the M:N relationship between STUDENT and SECTION in Figure. The reason it was made into a separate class (rather than as a pair of inverse relationships) is because it includes the relationship attribute Grade.

Hence, the M:N relationship is mapped to the class GRADE, and a pair of 1:N relationships, one between STUDENT and GRADE and the other between SECTION and GRADE. These relationships are represented by the following relationship properties: Completed_sections of STUDENT; Section and Student of GRADE; and Students of SECTION. Finally, the class DEGREE is used to represent the composite, multivalued attribute degrees of GRAD_STUDENT. Because the previousexample does not include any interfaces, only classes, we now utilize a different example to illustrate interfaces and interface (behavior)

inheritance. Figure 3.1.3(a) is part of a database schema for storing geometric objects. An interface GeometryObject is specified, with operations to calculate the perimeter and area of a geometric object, plus operations to translate (move) and rotate an object. Several classes (RECTANGLE, TRIANGLE, CIRCLE,...) inherit the Geometry Object interface. Since Geometry Object is an interface, it is *noninstantiable*— that is, no objects can be created based on this interface directly. However, objects of type RECTANGLE, TRIANGLE, CIRCLE, ... can be created, and these objects inherit all the operations of the GeometryObject interface. Note that with interface inheritance, only operations are inherited, not properties (attributes, relationships). Hence, if a property is needed in the inheriting class, it must be repeated in the class definition, as with the Reference_point attribute in Figure. Notice that the inheritedoperations can have different implementations in each class. For example, the implementations of the area and perimeter operations may be different for RECTANGLE, TRIANGLE, and CIRCLE. *Multiple inheritance* of interfaces by a class is allowed, as is multiple inheritance of interfaces by another interface. However, with the **extends** (class) inheritance, multiple inheritance is *not permitted.* Hence, a class can inherit via **extends** from at most one class (in addition to inheriting from zero or more interfaces).

5.1.15 OBJECT QUERY LANGUAGES (OQL)

The object query language OQL is the query language proposed for the ODMG object model. It is designed to work closely with the programming languages for rich an ODMG binding is defined, such as C++, Smalltalk, and Java. Hence, an OQL query embedded into one of these programming languages can return objects that match the type system of that language. Additionally, the implementations of class operations in an ODMG schema can have their code written in these programming languages. The OQL syntax for queries is similar to the syntax of the relational standard query language SQL, with additional features for ODMG concepts, such as object identity, complex objects, operations, inheritance, polymorphism, and relationships.

In Section we will discuss the syntax of simple OQL queries and the concept of using named objects or extents as database entry points. Then, in Section we will discuss the structure of query results and the use of path expressions to traverse relationships among objects. Other OQL features for handling object identity, inheritance, polymorphism, and other object-oriented concepts are discussed in Section. The examples to illustrate OQL queries are based on the UNIVERSITY database schema given in Figure. The basic OQL syntax is a select ... from ... where ... structure, as it is for SQL. For example, the query to retrieve the names of all departments in the college of 'Engineering' can be written as follows:

```
Q0: select   D.Dname
    from     D in DEPARTMENTS
    where    D.College = 'Engineering';
```

In general, an **entry point** to the database is needed for each query, which can be any *named persistent object.* For many queries, the entry point is the name of the extent of a class. Recall that the extent name is considered to be the name of a persistent object whose type is a collection (in most cases, a set) of objects from the class. Looking at the extent names in Figure, the named object DEPARTMENTS is of type set<DEPARTMENT>; PERSONS is of type set<PERSON>; FACULTY is of type set<FACULTY>; and so on.

The use of an extent name—DEPARTMENTS in Q0—as an entry point refers to a persistent collection of objects. Whenever a collection is referenced in an OQL query, we should define an **iterator variable**42—*D* in Q0—that ranges over each object in the collection. In many cases, as in Q0, the query will select certain objects from the collection, based on the conditions specified in the where clause. In Q0, only persistent objects *D* in the collection of DEPARTMENTS that satisfy the condition *D*.College = 'Engineering' are selected for the query result. For each selected object *D*, the value of *D*.Dname is retrieved in the query result. Hence, the *type of the result* for Q0 is bag<string> because the type of each Dname value is string (even though the actual result is a set because Dname is a key attribute). In general, the result of a query would be of type bag for select ... from ... and of type set for select distinct ... from ... , as in SQL (adding the keyword distinct eliminates duplicates). Using the example in Q0,there are three syntactic options for specifying iterator variables:

D **in** DEPARTMENTS
DEPARTMENTS *D*
DEPARTMENTS **AS** *D*

We will use the first construct in our examples. The named objects used as database entry points for OQL queries are not limited to the names of extents. Any named persistent object, whether it refers to an atomic (single) object or to a collection object, can be used as a database entry point.

5.2.1 INTRODUCTION TO TEMPORAL DATABASE

Temporal databases, in the broadest sense, encompass all database applications that require some aspect of time when organizing their information. Hence, they provide a good example to illustrate the need for developing a set of unifying concepts for application developers to use. Temporal database applications have been developed since the early days of database usage.

```
T1: CREATE TRIGGER Total_sal1
    AFTER UPDATE OF Salary ON EMPLOYEE
    REFERENCING OLD ROW AS O, NEW ROW AS N
    FOR EACH ROW
    WHEN  ( N.Dno IS NOT NULL )
    UPDATE DEPARTMENT
    SET Total_sal = Total_sal + N.salary - O.salary
    WHERE Dno = N.Dno;

T2: CREATE TRIGGER Total_sal2
    AFTER UPDATE OF Salary ON EMPLOYEE
    REFERENCING OLD TABLE AS O, NEW TABLE AS N
    FOR EACH STATEMENT
    WHEN  EXISTS ( SELECT *FROM N WHERE N.Dno IS NOT NULL ) OR
          EXISTS ( SELECT * FROM O WHERE O.Dno IS NOT NULL )
    UPDATE DEPARTMENT AS D
    SET D.Total_sal = D.Total_sal
    + ( SELECT SUM (N.Salary) FROM N WHERE D.Dno=N.Dno )
    - ( SELECT SUM (O.Salary) FROM O WHERE D.Dno=O.Dno )
    WHERE Dno IN ( ( SELECT Dno FROM N ) UNION ( SELECT Dno FROM O ) );
```

Figure 5.1.9 Trigger T1 illustrating the syntax for defining triggers in SQL-99.

However, in creating these applications, it is mainly left to the application designers and developers to discover, design, program, and implement the temporal concepts they need. There are many examples of applications where some aspect of time is needed to maintain the information in a database. These include *healthcare*, where patient histories need to be maintained; *insurance*, where claims and accident histories are required as well as information about the times when insurance policies are in effect; *reservation systems* in general (hotel,airline, car rental, train, and so on), where information on the dates and times when reservations are in effect are required; *scientific databases*, where data collected from experiments includes the time when each data ismeasured; and so on. Even the two examples used in this book may be easily expanded into temporal applications. In the COMPANY database, we may wish to keep SALARY, JOB, and PROJECT histories on each employee.

In the UNIVERSITY database, time is already included in the SEMESTER and YEAR of each SECTION of a COURSE, the grade history of a STUDENT, and the information on research grants. In fact, it is realistic to conclude that the majority of database applications have some temporal information. However, users often attempt to simplify or ignore temporal aspects because of the complexity that they add to their applications. In this section, we will introduce some of the concepts that have been developed to deal with the complexity of temporal database applications. An overview of how time is represented in databases, the different types of temporal information, and some of the different dimensions of time that may be needed. This section gives some additionaloptions for representing time that are possible in database models that allow complex-structured objects, such as object databases. Section introduces operations for querying temporal databases, and gives a brief

overview of the TSQL2 language, which extends SQL with temporal concepts. Section focuses on time series data, which is a type of temporal data that is very important in practice.

5.2.2 TIME ONTOLOGY

For temporal databases, time is considered to be an *ordered sequence* of **points** in some **granularity** that is determined by the application. For example, suppose that some temporal application never requires time units that are less than one second. Then, each time point represents one second using this granularity. In reality, each second is (short) *time duration*, not a point, since it may be further divided into milliseconds, microseconds, and so on. Temporal database researchershave used the term **chronon** instead of point to describe this minimal granularity for a particular application. The main consequence of choosinga minimum granularity—say, one second—is that events occurring within the same second will be considered to be *simultaneous events*, even though in reality they may not be.

Because there is no known beginning or ending of time, one needs a reference point from which to measure specific time points. Various calendars are used by various cultures (such as Gregorian (western), Chinese, Islamic, Hindu, Jewish, Coptic, and so on) with different reference points. A **calendar** organizes time into different time units for convenience. Most calendars group 60 seconds into a minute, 60 minutes into an hour, 24 hours into a day (based on the physical time of earth's rotation around its axis), and 7 days into a week. Further grouping of days into months and months into years either follow solar or lunar natural phenomena, and are generally irregular. In the Gregorian calendar, which is used in most western countries, days are grouped into months that are 28, 29, 30, or 31 days, and 12 months are grouped into a year. Complex formulas are used to map the different time units to one another.

In SQL2, the temporal data types (see Chapter 4) include DATE (specifying Year, Month, and Day as YYYY-MM-DD), TIME (specifying Hour, Minute, and Second as HH:MM:SS), TIMESTAMP (specifying a Date/Time combination, with options for including subsecond divisions if they are needed), INTERVAL (a relative time duration, such as 10 days or 250 minutes), and PERIOD (an *anchored* time duration with a fixed starting point, such as the 10-day period from January 1, 2009, to January 10, 2009, inclusive). **Event Information versus Duration (or State)Information.** A temporal database will store information concerning when certain events occur, or when certain facts are considered to be true. There are several different types of temporal information. **Point events** or **facts** are typically associated in the database with a **single time point** in some granularity. For example, a bank deposit event may be associated with the timestamp when the deposit was made, or the total monthly sales of a product (fact) may be associated with a particular month (say, February 2010). Note that even though such events or facts may have

different granularities, each is still associated with a *single time value* in the database. This type of information is often represented as **time series data** as we will discuss. **Duration events** or **facts**, on the other hand, are associated with a specific **time period** in the database. For example, an employee may have worked in a company from August 15, 2003 until November 20, 2008.

A **time period** is represented by its **start** and **end time points** [START-TIME, ENDTIME]. For example, the above period is represented as [2003-08-15, 2008-11-20]. Such a time period is ofteninterpreted to mean the *set of all time points* from starttime to end-time, inclusive, in the specified granularity. Hence, assuming day granularity, the period [2003-08-15, 2008-11-20] represents the set of all days from August 15, 2003, until November 20, 2008, inclusive.13 **Valid Time and Transaction Time Dimensions.** Given a particular event or fact that is associated with a particular time point or time period in the database, the association may be interpreted to mean different things. The most natural interpretation is that the associated time is the time that the event occurred,or the period during which the fact was considered to be true *in the real world.* If this interpretation is used, the associated time is often referred to as the **valid time**. A temporal database using this interpretation is called a **valid time database**. However, a different interpretation can be used, where the associated time refers to the time when the information was actually stored in the database; that is, it is the value of the system time clock when the information is valid *in the system.* In this case, the associated time is called the **transaction time**. A temporal database using this interpretation is called a **transaction time database**.

Other interpretations can also be intended, but these are considered to be the most common ones, and they are referred to as **time dimensions**. In some applications, only one of the dimensions is needed and in other cases both time dimensions are required, in which case the temporal database is called a **bitemporal database**. If other interpretations are intended for time, the user can define the semantics and program the applications appropriately, and it is called a **user-defined time**. The next section shows how these concepts can be incorporated into relational databases, and Section shows an approach to incorporate temporal concepts into object databases.

5.2.3 STRUCTURE AND GRANULARITY

The time domain (or ontology) specifies the basic building blocks of time. It is generally modeled as a set of time instants (or points) with an imposed partial order, e.g., (N, <). Additional axioms impose more structure on the time domain, yielding more refined time domains. Linear time advances from past to future in a step-by-step fashion. This model of time is mainly used in the database area. In contrast, AI applications often used a branching time model, which has a tree-like structure, allowing for possible futures. Time is linear from

the past to now, where it divides into several time lines; along any future path, additional branches may exist. This yields a tree-like structure rooted at now. Now marks the current time point and is constantly moving forward. The time domain can be bounded in the past and/or in the future, i.e., a first and/or last time instant exists; otherwise, it is called unbounded.The time domain can be dense, discrete, or continuous. In a discrete time domain, time instants are non-decomposable units of time with a positive duration, called chronons. A chronon is the smallest duration of time that can be represented. This time model is isomorphic to the natural numbers. In contrast, in a dense time domain, between any two instants of time,there exists another instant; this model is isomorphic to the rational numbers. Finally, continuous time is dense and does not allow "gaps" between consecutive time instants. Time instants are durationless. The continuous time model is isomorphic to the real numbers. While humans perceive time as continuous, a discrete linear time model is generally used in temporal databases for several practical reasons, e.g., measures of time are generally reported in terms of chronons, natural language referencesare compatible with chronons, and any practical implementation needs a discrete encoding of time. A limitation of a discrete time model is, for example, the inability to represent continuous phenomena.

A time granularity is a partitioning of the time domain into a finite set of segments, called granules, providing a particular discrete image of a (possibly continuous) timeline. The main aim of granularities is to support user friendly representations of time. For instance, birth dates are typically measured at the granularity of days, business appointments at the granularity of hours, and train schedules at the granularity of minutes. Multiple granularities are needed in many real-world applications.

In any specific application, the granularity of time has some practical magnitude. For instance, the time-point that a business event, like a purchase, is associated with a date, so that a day is the proper granule for most business transactions. People do not schedule themselves for intervals of less than a minute, while database transactions may be measured in milliseconds. Eventually we are limited by the precision that our hardware can recognize; fractions of microseconds are the finest grain here. We use G to denote the granularity; it is in effect an interval. The finiteness of measurement granules leads to a confusion of event times andintervals. If we limit our event measures to dates (G = 1 day), and we say that an event occurred on such and-such a day, then implicit for most of usis also that the event spanned some interval within that day. A point event is then associated with an interval of one granule length. There will be a smallest time granule G, perhaps intervals of seconds or days, whichfollow each other without gaps, and are identified by the timepoint at their beginning. True Intervals are sequences of event measuring interval

However, problems arise with this simplification. Inconsistencies occur when an inclusive interval is defined by two event time measurements with an implicit grain. First we have to round actual

measurements to the integer grain size used; then we add the granule size to the result:

$$T_G = t_f - t_s + G$$

where ts denotes the value corresponding to the start of the interval and t/the value when the interval is finished. Thus, if movie is shown daily from the 12th to the 19th of a month, there will be 19-12+1 = 8 performance days. While we are all used to performing such adjustments when computing with intervals, a database system which deals with temporal events must present consistent temporal semantics to the user to avoid confusion. We cannot use an event directly to compute an interval but always have to correct for the associated grain size. While in any one application use of a fixed grain size is feasible, problems arise if we merge information from distinct applications. A database system has to carry out the computations to satisfy the application semantics. If those include the use of finite events, then the grain size assumption made must be explicitly stated. Many granularities may need to be simultaneously active.In our formulation we will require two datatypes, infinitesimal time points for events and intervals for histories, to deal with all temporal data.

5.2.4 TEMPORAL DATA MODELS

Most of the work in the research area of temporal databases has been done in respect of the relational data model. In this section, as shown below in table 2 and Table 3, some of the most important temporal data models are compared

TABLE 2
Summary of data models in temporal perspective

model / parameter	Ariav model [1]	Ben-Zvi model [2]	Clifford and Croker model [3][4]	Gadia model [12][13]	Jensen & Snodgrass Model [19][20]
Time	VT & TT	VT & TT	VT	VT	VT & TT
Timestamp	Tuple	Tuple	Attrib	Attrib	Tuple
Functional	VTC	VTC	Valid	VTC	Valid

dependency					
Nornal form – conventional	S	NS	NS	NS	S
Nornal form – time	NS	S	S	S	NS
Query language support	NS	S	S	S	S
Optimization	NS	S	NS	S	NS
Temporal operator support	NS	S	S	S	S
Distribution	NS	NS	NS	NS	NS
Temporal keys	S	S	S	S	S

Attrib:Attribute; NS:Not supported; S:Supported;
VTC:Validity to be checked.

TABLE 3
Summary of data models in temporal perspective

model / parameter	Lorentzos model [21]	Snodgrass Model [27]	Tansel Model [29]	Vianu model [30]	Wijsen Model [31]
Time	VT	VT & TT	VT	VT	VT
Timestamp	Attrib	Tuple	Attrib	Tuple	Attrib
Functional dependency	VTC	Valid	VTC	VTC	VTC
Nornal form – conventional	S	NS	NS	S	S
Nornal form – time	NS	S	S	NS	NS
Query language support	NS	S	S	NS	NS
Optimization	NS	S	S	NS	NS
Temporal operator support	S	S	S	S	S
Distribution	NS	NS	NS	NS	NS
Temporal keys	S	S	S	S	S

Attrib:Attribute; NS:Not supported; S:Supported;
VTC:Validity to be checked.

Ariav's model used tuple time stamping with time being represented by discrete time points in bitemporal mode. The model is conceptually simplistic but difficult to implement in efficiency and reliability terms. Ben-Zvi's time relational model (TRM) was a pioneeringwork in many aspects. The most important idea of TRM is perhaps the non

first normal form (NFNF). Ben-Zvi's concept of effective time and registration time, which are now known as valid time (VT) and transaction time (TT), respectively, added new dimensions to time varying information computing. Ben-Zvi was the first to coin the term and notion of time-invariant key for his non first normal tuples, called tuple version sets in his terminology. Differentiation between an error and a change was recognized and both of them were made queriable. Also the need for fast access to current data was recognized. Clifford and Croker's model followed historical relational data model (HRDM) and tuples areheterogeneous in their temporal dimension. Unfortunately, the historical relational algebra is not a complete algebra w.r.t. HRDM. So, the cartesianproduct of three-dimensional relation (e.g. join operation) is not clear and hence results are not reliable. Gadia's model has temporal element as an appropriate datatype for time. This model assumes that key values donot change with time. Another requirement is all attributes in a tuple have the same time domain. This requirement is called homogeneity. The positive aspect of Gadia's model is that it minimizes redundancy. But when concatenation of partial temporal elements along with tuple homogeneity is implemented, the query results into incomplete or missing information. Jensen & Snodgrass model proposed bitemporal conceptual data model (BCDM), allowing to associate both valid and transaction times with data. The domains of valid and transaction times are the finite sets DVT and DTT, respectively. A valid time chronon cv is a time point belonging to DVT and a transaction time chronon ct is a time point belonging to DTT.A bitemporal chronon cb = (ct , cv) is an ordered pair consisting of a transaction time chronon and a valid time chronon. The schema of abitemporal relation R, defined on the set U = {A1,A2,...,An} of non- timestamp attributes, is of the form R = (A1,A2,...,An | T), that is, it consists of n non-timestamp attributes A1,A2,...,An, with domain dom(Ai) for each i $\in$ [1,n], and an implicit timestamp attribute T. The domain of T is (DTT $\cup${UC})$\times$ DVT , where UC is a special value that can be assumed by a transaction time chronon to express the condition "until changed". For instance, to state that a tuple valid at time cv is current in the database, the bitemporal chronon (UC, cv) must be assigned to the tuple timestamp. As a general rule, they associate a set of bitemporal chronons in the two-dimensional space with every tuple. Lorentzos's model followed interval-extended relational model (IXRM) and an interval relational algebra for the management of interval relations. The fundamental properties of a model are that it must be satisfactory and simple. Lorentzos model satisfied both aspects. However, when a model is defined, efficiency issues are of minor importance. IXRM operations require a great deal of space and time, is a point of concern. Snodgrass's Model uses temporal query language (TQuel) which is based on the predicate calculus. One of the key features of this model is when the algebra is used to implement theTQuel, the a conversion will be necessary between tuple timestamping (where each tuple is associated with a single interval) and attribute-value time-stamping (where each attribute is associated with potentially multipleintervals). This conversion is formalized in a transformation function (T). Though this model seems to be more efficient but relatively less user

friendly. Tansel's model used Attribute-value timestamping and used the concepts of time by example (TBE) and query by example (QBE). This model is quite user friendly. However, nested temporal relations are an area of concern since structuring nested temporal relations hinges upon thetype of associations between the involved entities. Vianu proposed a simple extension of the relational data model in order to describe evolution of a database over time. A database sequence is defined as a sequence of consecutive instances of the database, plus "update mappings"from one instance (the "old" one) to the next one (the "new" instance). Constraints on the evolution of attribute values of tuples (objects) over time are expressed by means of dynamic functional dependencies (DFDs), that make it possible to define dependencies between old and new values of attributes on updates. Wijsen and his colleagues temporal data model proposed three types of keys i.e. snapshot keys (SK), dynamic keys (DK) and temporal keys (TK) corresponding to snapshot functional dependency, dynamic dependency and temporal dependency, respectively. Let dom be a set of atomic values, that is, the union of disjoint domains correspondingto atomic types, att be a set of attribute names, and λ be a special attribute used to denote object identity. Moreover, let obj be an infinite set of object identifiers (OIDs) and class be a set of class names. Given a finite set of class names C, a type over C is a set {A1: τ 1, A2: τ 2,..., An: τ n}, where A1, A2,... An are distinct attribute names and each τi with $1 \leq i \leq n$, is either an atomic type or a class name in C. A schema is a pair (C,ρ), whereC is a finite set of class names and ρ is a total function that maps eachclass name in C into a type over C.

Some data models use FNF and others prefer NFNF. The choice may depend on the consideration of time as discrete or interval based or continuous. Also, the traditional Entity relationship model (ERM) can be extended for temporal data models (TDM) by considering suitable operators and constructs for their effective and efficient implementation. The traditional ERM is capable of capturing the whole temporal aspects. Many extensions [15][16] have been proposed to extend the ERM in orderto capture time varying information. For graphical representation, Unified modeling language (UML) is normally used. However, UML constructs in reference of temporal data models can be possibly used and drawn using application softwares like Rational software architecture. The temporal query language (TQuel) supports both valid time and transaction time. It also supports user defined time. Tuples are optimally time-stamped with either a single valid time stamp (if a relation models events) or a pair of valid timestamps (if a relation models intervals), along with transaction timestamps, denoting when the tuple was logically inserted into the relation. A transaction timestamp of "until changed" indicates that the tuple has not been deleted yet. A functional example of temporal database is TimeDB. It uses the extension approach with respect to the data structures. TimeDB uses a layered approach which means it was built as a front end to a commercial DBMS that translates temporal statements into standard SQL statements. This way, it is possible to support features such as persistence, consistency, concurrency, recovery etc. without having to

implement from the scratch. It is a bitemporal DBMS. TimeDBimplements the query language ATSQL2. ATSQL2 includes not only a bitemporal query language but also a bi-temporal modification, data definition and constraint specification language. TimeDB implements the temporal algebra operations using standard SQL statements. TimeDB supports a command oriented user interface

5.2.5 TEMPORAL RELATIONAL ALGEBRAS

In Temporal Datalog programs, we do not view predicates as representing an infinite collection of finite relations, nor do we manipulate finite relations at any given moment in time. Temporal relations are not first-class citizens in Temporal Datalog. Moreover, without negation, the set difference operation available in the relational algebra cannot be specified in Temporal Datalog. To alleviate these limitations, we introduce a temporal relational algebra as a query language for Temporal Datalog, which is referred to as TRA. Examples of TRA expressions are given in the next section

TRA is a "pointwise extension" of the relational algebra upon ω defined as follows. The signature of TRA includes all operators of the relational algebra, and the universe of TRA is the set of temporal relations $\cup_{n\geq 0}[\omega \to \mathcal{P}(U^n)]$ where U is the domain of interpretations of TL. Let ∇ be a unary operator of the relational algebra, and ∇_T be the corresponding operator of TRA. For any given temporal relation r, the following holds: for all $t \in \omega$,

$$\nabla_T(r)(t) = \nabla(r(t)).$$

Similarly for the binary operators. In other words, when restricted to moments in time, a pointwise operator degenerates into the corresponding operator of the relational algebra. Therefore we can explain what a pointwise operator does to its operand by looking at the individual results for each moment in time. This implies that TRA by design has the relational algebra as a special case.

Expressions, Operations An expression of TRA consists of compositions of algebraic operators and predicate symbols. Algebraic operators are applied to temporal relations and yield temporal relations as a result which can then be used as operands in other expressions. In addition to the pointwise operators, the signature of TRA is extended with temporal and aggregation operators summarized below. In the following, we assume familiarity with the notions of a comparator value and comparator formula; see [17] for details. Pointwise Operators Pointwise operators are $\cap$, $\cup$, $\times$,$-$, πX, and σF . At any given moment in time t, the outcome of a pointwise operation depends only on the values of its operands at time t. For instance, given the expression r $\cap$ s, the resulting temporal relation is the pointwise intersection of r and s. Temporal Operators Temporal operators are first, next, prev and fby[·]. The temporaloperators first and next of TRA do not behave the same way as those ofTL do. However, we use the same symbols as it is always understood fromthe context which ones are referred to. The 4 temporal operator firstfreezes a temporal relation at its initial value; next and prev shift a given

temporal relation in the corresponding direction; fby[·] does temporal splicing, i.e., cutting and pasting of temporal relations. Aggregation Operators Let $x \geq 1$. Aggregation operators are sumx, avgx , count, maxx and minx. These pointwise operators are applied to temporal relations with arbitrary arities, and produce unary temporal relations as a result. Denotational Semantics Given a Temporal Datalog program db, an expression over db contains only those predicate symbols appearing in db, and terms from the Herbrand universe of db. We assume that all expressions are legal, i.e., arities of relations given in an expression match with respect to the operations involved, and so do the types of attributes over which aggregation operations are performed. The meaning of an expression of TRA is a temporal relation, just like the meaning of apredicate symbol defined in db. Let [[E]](db) denote the denotation (meaning) of E with respect to a given temporal database db. In particular, we have that [[E]](db) is an element of $[\omega \rightarrow P(U\ k\ )]$ for some $k \geq 0$. In general, given an expression E of TRA, we have that $[[E]] \in DB \rightarrow [\ n \geq 0[\omega \rightarrow P(U\ n\ )]$ where DB is the set of Temporal Datalog programs, and U is the set of ground terms of TL. Given $db \in DB$, $s \in \omega$, $x \geq 1$, and TRA expressions A and B, the following are the definitions of the denotations of each kind of expressions of TRA.

1. [[p]](db) = (uM(db))(p) where p is a predicate symbol appearing in db.

2. $[[A \nabla B]](db) = [[A]](db) \nabla [[B]](db)$ where ∇ is any of $\cap$, $\cup$, $\times$ and $-$.

3. $[[\nabla A]](db) = \nabla [[A]](db)$ where ∇ is any of πX, σF , sumx, avgx, maxx, minx and count.

4. $[[\text{first } A]](db) = \lambda t.[[A]](db)(0)$.

5. $[[\text{next } A]](db) = \lambda t.[[A]](db)(t + 1)$.

6. $[[\text{prev } A]](db) = \lambda t.\ [[A]](db)(t - 1), t > 0 \varnothing, t = 0$

7. $[[A\ \text{fby}[s]\ B]](db) = \lambda t.\ [[A]](db)(t), t \leq s\ [[B]](db)(t), t > s$ Item 1 provides the link to the temporal database: the denotation of a predicate symbol is the temporal relation that the predicate represents with respect to the minimum model of db. Items 4 through 7 formalizes what temporal operators do to their operands. At time 0, the value of any expression of the form prev A is the empty set, because we cannot go into the past beyond time 0.

5.2.6 INTRODUCTION TO SPATIAL DATABASE

Spatial databases incorporate functionality that provides supportfor databases that keep track of objects in a multidimensional space. For example, cartographic databases that store maps include two-dimensional spatial descriptions of their objects—from countries and states to rivers,

cities, roads, seas, and so on. The systems that manage geographic data and related applications are known as **Geographical InformationSystems (GIS)**, and they are used in areas such as environmental applications, transportation systems, emergency response systems, and battle management. Other databases, such as meteorological databases for weather information, are three-dimensional, since temperatures and other meteorological information are related to three-dimensional spatial points. In general, a **spatial database** stores objects that have spatial characteristics that describe them and that have spatial relationships among them. The spatial relationships among the objects are important, and they are often needed when querying the database. Although a spatial database can in general refer to an *n*-dimensional space for any *n*, we will limit our discussion to two dimensions as an illustration. A spatial database is optimized to store and query data related to objects in space, including points, lines and polygons. Satellite images are a prominent example of spatial data. Queries posed on these spatial data, where predicates for selection deal with spatial parameters, are called **spatial queries**. For example, "What are the names of all bookstores within five miles of the College of Computing building at Georgia Tech?" is a spatial query. Whereas typical databases process numeric and character data,additional functionality needs to be added for databases to process spatial data types. A query such as "List all the customers located within twenty miles of company headquarters" will require the processing of spatial data types typically outside the scope of standard relational algebra and may involve consulting an external geographic database that maps the company headquarters and each customer to a 2-D map based on their address. Effectively, each customer will be associated to a <latitude, longitude> position. A traditional B+-tree index based on customers' zip codes or other nonspatial attributes cannot be used to process this query sincetraditional indexes are not capable of ordering multidimensional coordinate data. Therefore, there is a special need for databases tailored for handling spatial data and spatial queries.

5.2.7 DEFINITION

The common analytical operations involved in processing geographic or spatial data. **Measurement operations** are used to measure some global properties of single objects (such as the area, the relative size of an object's parts, compactness, or symmetry), and to measure the relative position of different objects in terms of distance and direction. **Spatial analysis** operations, which often use statistical techniques, areused to uncover *spatial relationships* within and among mapped datalayers. An example would be to create a map—known as a *prediction map*—that identifies the locations of likely customers for particular products based on the historical sales and demographic information. **Flow analysis** operations help in determining the shortest path between two points and also the connectivity among nodes or regions in a graph. **Location analysis** aims to find if the given set of points and lines lie within a given polygon (location). The process involves generating a

buffer around existing geographic features and then identifying or selecting features based on whether they fall inside or outside the boundary of the buffer. **Digital terrain analysis** is used to build three- dimensional models, where the topography of a geographical location can be represented with an *x*, *y*, *z* data model known as Digital Terrain (or Elevation) Model (DTM/DEM). The *x* and *y* dimensions of a DTM represent the horizontal plane, and *z* represents spot heights for the respective *x*, *y* coordinates. Such models can be used for analysis ofenvironmental data or during the design of engineering projects that require terrain information. Spatial search allows a user to search forobjects within a particular spatial region. For example, **thematic search** allows us to search for objects related to a particular theme or class, suchas "Find all water bodies within 25 miles of Atlanta" where the class is *water*.

There are also **topological relationships** among spatial objects. These are often used in Boolean predicates to select objects based on their spatial relationships. For example, if a city boundary is represented as a polygon and freeways are represented as multilines, a condition such as "Find all freeways that go through Arlington, Texas" would involve an *intersects* operation, to determine which freeways (lines) intersect the city boundary (polygon).

Table Common Types of Analysis for Spatial Data

Analysis Type	Type of Operations and Measurements
Measurements	Distance, perimeter, shape, adjacency, and direction
Spatial analysis/statistics	Pattern, autocorrelation, and indexes of similarity and topology using spatial and nonspatial data
Flow analysis	Connectivity and shortest path
Location analysis	Analysis of points and lines within a polygon
Terrain analysis	Slope/aspect, catchment area, drainage network
Search	Thematic search, search by region

5.2.8 TYPES OF SPATIAL DATA

This section briefly describes the common data types and models for storing spatial data. Spatial data comes in three basic forms. These forms have become a *de facto* standard due to their wide use in commercial systems.

■ **Map Data** includes various geographic or spatial features of objects in a map, such as an object's shape and the location of the object within the map. The three basic types of features are points, lines, and polygons (or areas). **Points** are used to represent spatial characteristics of objects whose locations correspond to a single 2-d coordinate (*x*, *y*, or longitude/latitude) in the scale of a particular application. Depending on the scale, some examples of point objects could be buildings, cellular towers, or stationary vehicles. Moving locations that change over time. **Lines** represent objects

having length, such as roads or rivers, whose spatial characteristics can be approximated by a sequence of connected lines. **Polygons** are used to represent spatial characteristics of objects that have a boundary, such as countries, states, lakes, or cities. Notice that some objects, such as buildings or cities, can be represented as either points or polygons, depending on the scale of detail.

■ **Attribute data** is the descriptive data that GIS systems associate with **map features**. For example, suppose that a map contains features that represent counties within a US state (such as Texas or Oregon). Attributes for each county feature (object) could include population, largestcity/town, area in square miles, and so on. Other attribute data could be included for other features in the map, such as states, cities, congressional districts, census tracts, and so on.

■ **Image data** includes data such as satellite images and aerial photographs, which are typically created by cameras. Objects of interest, such as buildings and roads, can be identified and overlaid on these images. Images can also be attributes of map features. One can add imagesto other map features so that clicking on the feature would display the image. Aerial and satellite images are typical examples of raster data.

Models of spatial information are sometimes grouped into two broad categories: *field* and *object.* A spatial application (such as remote sensing or highway traffic control) is modeled using either a field- or an object-based model, depending on the requirements and the traditional choice of model for the application. **Field models** are often used to model spatial data that is continuous in nature, such as terrain elevation, temperature data, and soil variation characteristics, whereas **object models** have traditionally been used for applications such as transportation networks, land parcels, buildings, and other objects that possess bothspatial and non-spatial attributes.

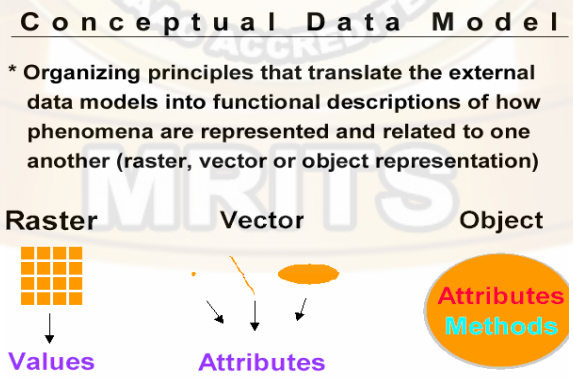
5.2.9 GEOGRAPHICAL INFORMATION SYSTEMS (GIS)

Geographic Information Systems (GIS) contain spatial information about cities, states, countries, streets, highways, lakes, rivers, and other geographical features and support applications to combine such spatial information with non-spatial data. Spatial data is stored in either raster or vector formats. In addition, there is often a temporal dimension, as when we measure rainfall at several locations over time. An important issue with spatial datasets is how to integrate data from multiple sources, since each source may record data using a different coordinate system to identify locations. Now let us consider how spatial data in a GIS is analyzed. Spatial information is almost naturally thought of as being overlaid on maps. Typical queries include "What cities lie on 1-94 between Madison and Chicago?" and "What is the shortest route from Madison to St.

Louis?" These kinds of queries can be addressed using the techniques. An emerging application is in-vehicle navigation aids. With Global Positioning System (CPS) technology, a car's location can be pinpointed, and by accessing a databa.se of local maps, a driver can receive directions from In his or her current location to a desired destination; this application also involves mobile database access! In addition, many applications involve interpolating measurements at certain locations across an entire region to obtain a *model* and combining overlapping models. For example,if -ve have measured rainfall at certain locations, we can use the Triangulated Irregular Network (TIN) approach to triangulate the region, with the locations at which we have measurements being the vertices ofthe triangles. Then, we use some form of interpolation to estimate the rainfall at points within triangles. Interpolation, triangulation, overlays, visualization of spatial data, and many other domain-specific operationsare supported in GIS products such ARC-In while spatial query processingtechniques are an important part of a GIS product, considerable additional functionality must be incorporated as well. How best to extend 0 to 1 systems with this additional functionality is an important problem yet tobe resolved. Agreeing on standards for data representation formats and coordinate system is another major challenge facing the field.

5.2.10 CONCEPTUAL DATA MODELS FOR SPATIAL DATABASES

Conceptual data model: provide the organizing principles that translate the external data models into functional descriptions of how data objects are related to one another (e.g. non-spatial: E-R model; spatial: raster, vector, object representation).



5.2.11 LOGICAL DATA MODELS FOR SPATIAL DATABASES

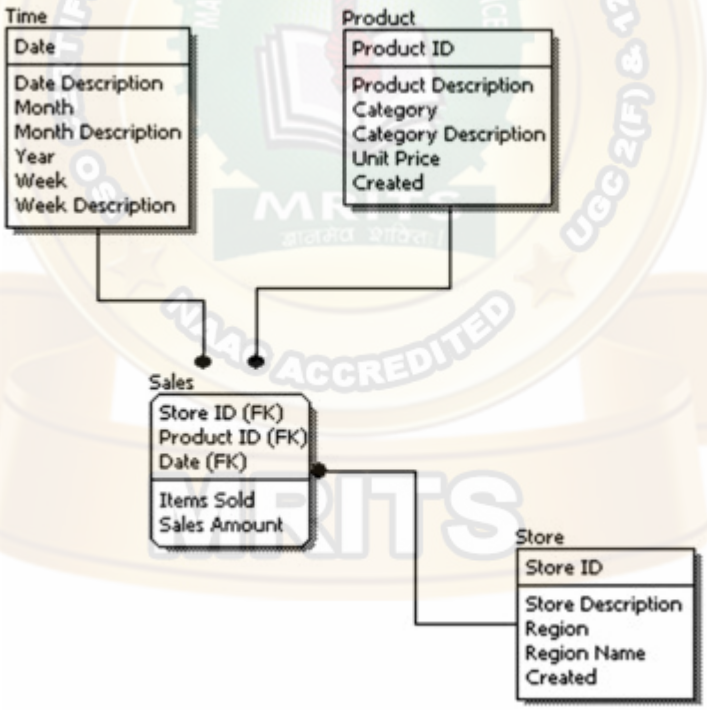
Logical data model: provide the explicit forms that the conceptual models can take and is the first step in computing (e.g. non-spatial: hierarchical, network, relational; spatial: 2-d matrix, map file, location list, point dictionary, arc/nodes).

Logical Model
(data structure)

*** Provides the explicit forms the conceptual data model can take.**

Raster:	Vector:	Object:
* 2-D matrix	* Carto spaghetti	Object design
* vertical array	* polygon file	with relational
* map file	* point dictionary	tables
* quadtree	* arc/node	

Logical data modeling involves defining the logical structure for the data in terms of the database's model. Logical modeling takes a conceptual model and seeks to produce a logical schema for a database. For example, the general definition of each relation in a DBMS is concerned with: (i) what each attribute in the relation should represent the types of data identified during conceptual modeling; (ii) which attributes are key values and (iii) how different relations are joined within a DBMS and defined during logical data modeling.



5.2.12 RASTER AND VECTOR MODEL

Data structures are complex for GIS because they must include information pertaining to entities with respect to: position, topological relationships, and attribute information. It is the topologic and spatial aspects of GIS that distinguish it from other types of data bases.

Introduction: There are presently three types of representations for geographic data: raster vector, and objects.

raster - set of cells on a grid that represents an entity (entity --> symbol/color --> cells).
vector - an entity is represented by nodes and their connecting arc or line segment (entity --> points, lines or areas --> connectivity)

object - an entity is represented by an object which has as one of its attributes spatial information.

Raster Data model

Definition: realization of the external model which sees the world as a continuously varying surface (field) through the use of 2-D Cartesian arrays forming sets of thematic layers. Space is discredited into a set of connected two dimensional units called a tessellation.

Map overlays: separate set of Cartesian arrays or "overlays" for each entity.
Logical data models: 2-D array, vertical array, and Map file

Each overlay is a 2-D matrix of points carrying the value of a single attribute. Each point is represented by a vertical array in which each array position carries a value of the attribute associated with the overlay.

Map file - each mapping unit has the coordinates for cell in which itoccurs (greater structure, many to one relationship). Compact methods for coding

Vertical array not conducive to compact data coding because it references different entities in sequence and it lacks many to one relationship. The third structure references a set of points for a region (or mapping unit) and allows for compaction. Chain codes: a region is defined in terms of origin and (0 - 3) for E, N, W, S (Map file) (binary data).

+ reduced storage.
+ area, perimeter, shape est.
- overlay difficult.

Run-length codes: row #, begin, end (Map file entity #, # pix (2-D matrix).
+ reduce storage.
- overlay difficult.

Block codes: 2-D rle, regions stored using origin and radius.
+ reduced storage.
+ U & I of regions easy.

Quadtrees: recursive decomposition of a 2-D array into quads until the next subdivision yields a region containing a single entity.
+ reduced storage.
+ variable resolution.
+ overlay of variable resolution data.
+ fast search.

Morton Sequencing
Morton Sequencing Overlay
Morton Homework

Vector data model

Definition: realization of the discrete model of real world using structures for storing and relating points, lines and polygons in sets of thematic layers.

a. Introduction

- ✓ represents an entity as exact as possible.
- ✓ coordinate space continuous (not quantized like raster).
- ✓ Structured as a set of thematic layers

b. Representation

- ✓ Point entities: geographic entities that are positioned by a single x,y coordinate. (historic site, wells, rare flora. The data record consists for x,y - attribute.
- ✓ Line Entity: (rivers, roads, rail) all linear feature are made up of line segments. a simple line 2 (x,y) coordinates.
- ✓ An arc or chain or string is a set of n (x,y) coordinate pairs that describe a continuous line. The shorter the line segments the closer the chain will approximate a continuous curve. Data record n(x,y).
- ✓ A line network gives information about connectivity between line segments in the form of pointers or relations contained in the data structure. Often build into nodes pointers to define connections and angles indicating orientation of connections (fully defines topology).

Area Entity: data structures for storing regions. Data types, land cover, soils, geology, land tenure, census tract, etc.

Cartographic spaghetti or "connect the dots". Early development in automated cartography, a substitute for mechanical drawing. Numerical storage, spatial structure evident only after plotting, not in file.

- ✓ Location list
- ✓ describe each entity by specifying coordinates around its perimeter.

- ✓ shared lines between polygons.
- ✓ polygon sliver problems.
- ✓ no topology (neighbor and island problems).
- ✓ error checking a problem.

c. Point dictionary

Unique points for entire file, no sharing of lines as in location lists (eliminate sliver problem) but still has other problems. expensive searches to construct polygons.

d. Dime Files (Dual Independent Mapping and Encoding)

Designed to represent points lines and areas that form a city though a complete representation of network of streets and other linear features. allowed for topologically based verification.

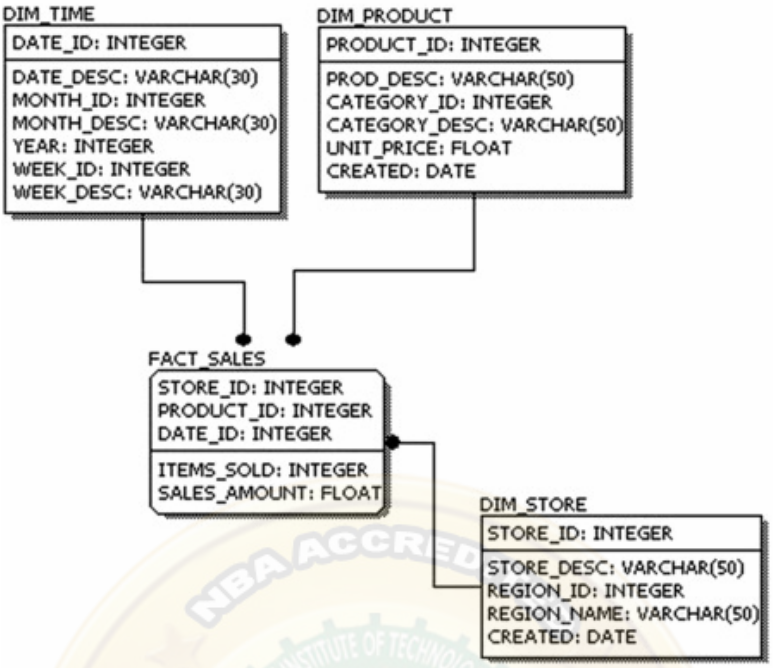
no systems of directories linking segments together (maintenance problem).

e. Arc/node

Same topological principles as the DIME system. DIME defined by line segments, chains based on records of uncrossed boundary lines (curved roads a problem for DIME). chains or boundaries serve the topological function of connecting two end points called a node and separating two zones. points between zones cartographically not topologically required (generalization possible). solves problems discussed above (neighbor, dead ends, weird polygons). can treat datainput and structure independently.

5.2.13 PHYSICAL DATA MODELS FOR SPATIAL DATABASES

Physical data modeling involves mapping the conceptual and logical models into a database implementation (Fig. 4.8). The result of physical modeling is a physical schema, which is tailored to a specific DBMS. Physical modeling fills in the blanks within the logical model required for a concrete DBMS, specifying actual values (data types) of each attribute, giving working names to the relations. Physical modeling results in a working physical database definition.



5.2.14 CLUSTERING METHODS (SPACE FILLING CURVES)

Spatial data tends to be highly correlated. For example, people with similar characteristics, occupations, and backgrounds tend to cluster together in the same neighborhoods. The three major spatial data mining techniques are spatial classification, spatial association, and spatial clustering.

■ **Spatial classification.** The goal of classification is to estimate the value of an attribute of a relation based on the value of the relation's other attributes. An example of the spatial classification problem is determining the locations of nests in a wetland based on the value of other attributes (for example, vegetation durability and water depth); it is also called the *location prediction problem*. Similarly, where to expect hotspots in crime activity is also a location prediction problem.

■ **Spatial association. Spatial association rules** are defined in terms of spatial predicates rather than items. A spatial association rule is of the form

$$P_1 \wedge P_2 \wedge \ldots \wedge P_n \Rightarrow Q_1 \wedge Q_2 \wedge \ldots \wedge Q_m,$$

where at least one of the P_i's or Q_j's is a spatial predicate. For example, the rule

$$\text{is\_a}(x, \text{country}) \wedge \text{touches}(x, \text{Mediterranean}) \Rightarrow \text{is\_a}(x, \text{wine-exporter})$$

(that is, a country that is adjacent to the Mediterranean Sea is typically a wine exporter) is an example of an association rule, which will have a certain support *s* and confidence *c*.

Spatial colocation rules attempt to generalize association rules to point to collection data sets that are indexed by space. There are several crucial differences between spatial and nonspatial associations including:

1. The notion of a transaction is absent in spatial situations, since data is embedded in continuous space. Partitioning space into transactions would lead to an overestimate or an underestimate of interest measures, for example, support or confidence.

2. Size of item sets in spatial databases is small, that is, there are many fewer items in the item set in a spatial situation than in a nonspatial situation. In most instances, spatial items are a discrete version of continuous variables. For example, in the United States income regions may be defined as regions where the mean yearly income is within certain ranges, such as, below $40,000, from $40,000 to $100,000, and above $100,000.

■ **Spatial Clustering** attempts to group database objects so that the most similar objects are in the same cluster, and objects in different clusters are as dissimilar as possible. One application of spatial clustering is to group together seismic events in order to determine earthquake faults. An example of a spatial clustering algorithm is **density-based clustering**, which tries to find clusters based on the density of data points in a region. These algorithms treat clusters as dense regions of objects in the data space. Two variations of these algorithms are density-based spatial clustering of applications with noise (DBSCAN) and density-based clustering (DENCLUE). DBSCAN is a density-based clustering algorithmbecause it finds a number of clusters starting from the estimated density distribution of corresponding nodes.

5.2.15 STORAGE METHODS (R-TREE)

The **R-tree** is a height-balanced tree, which is an extension of the B+-tree for *k*-dimensions, where $k > 1$. For two dimensions (2-d), spatial objects are approximated in the R-tree by their **minimum bounding rectangle** (**MBR**), which is the smallest rectangle, with sides parallel to the coordinate system (*x* and *y*) axis, that contains the object. R-trees are characterized by the following properties, which are similar to the properties for B+-trees but are adapted to 2-d spatial objects. As in Section, we use *M* to indicate the maximum number of entries that can fit in an R-tree node.

1. The structure of each index entry (or index record) in a leaf node is (I, *object-identifier*), where I is the MBR for the spatial object whose identifier is *object-identifier*.

2. Every node except the root node must be at least half full. Thus, a leaf node that is not the root should contain *m* entries (I, *object-identifier*) where $M/2 <= m <= M$. Similarly, a non-leaf node that is not the root

should contain *m* entries (I, *child-pointer*) where *M*/2 <= *m* <= *M*, and I is the MBR that contains the union of all the rectangles in the node pointed at by *child-pointer*.

3. All leaf nodes are at the same level, and the root node should have at least two pointers unless it is a leaf node.

4. All MBRs have their sides parallel to the axes of the global coordinate system. Other spatial storage structures include quadtrees and their variations. **Quadtrees** generally divide each space or subspace into equally sized areas, and proceed with the subdivisions of each subspace to identify the positions of various objects. Recently, many newer spatial access structures have been proposed, and this area remains an active research area.

5.2.16 QUERY PROCESSING

Spatial Query Processing in the Euclidean Space R-trees [G84, SRF87, BKSS90] are the most popular indexes for Euclidean query processing due to their simplicity and efficiency. The R-tree can be viewed as a multi-dimensional extension of the B-tree. Figure shows an exemplary R-tree for a set of points {a,b,…,j} assuming a capacity of three entries per node. Points that are close in space (e.g., a,b) are clustered in the same leaf node (E3) represented as a minimum bounding rectangle (MBR). Nodes are then recursively grouped together following the same principle until the top level, which consists of a single root.

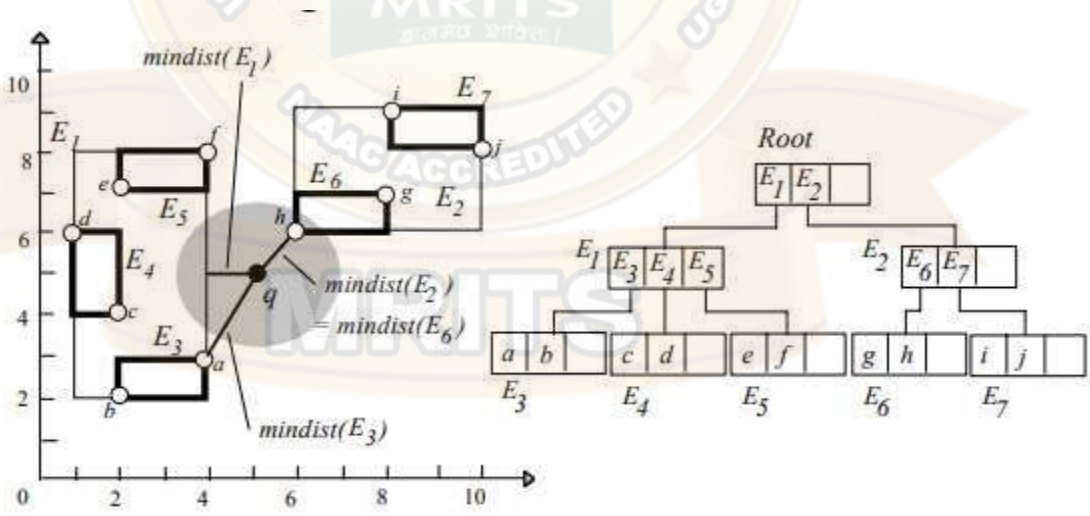


Figure 2.1: An R-tree example

R-trees (like most spatial access methods) were motivated by the need to efficiently process range queries, where the range usually corresponds to a rectangular window or a circular area around a query point. The R-tree answers the query q (shaded area) in Figure 2.1 as follows. The root is first retrieved and the entries (e.g., E1, E2) that

intersect the range are recursively searched because they may contain qualifying points. Non-intersecting entries (e.g., E3) are skipped. Note that for non-point data (e.g., lines, polygons), the R-tree provides just a filter step to prune non-qualifying objects. The output of this phase has to pass through a refinement step that examines the actual object representation to determine the actual result. The concept of filter and refinement steps applies to all spatial queries on non-point objects. A nearest neighbor (NN) query retrieves the ($k \geq 1$) data point(s) closest to a query point q. The R-tree NN algorithm proposed in [HS99] keeps a heap with the entries of the nodes visited so far. Initially, the heap contains the entries of the root sorted according to their minimum distance (mindist) from q. The entry with the minimum mindist in the heap (E1 in Figure 2.1) is expanded, i.e., it is removed from the heap and its children (E3, E4, E5) are addedtogether with their mindist. The next entry visited is E2 (its mindist is currently the minimum in the heap), followed by E6, where the actual result (h) is found and the algorithm terminates, because the mindist of all entries in the heap is greater than the distance of h. The algorithm can be easily extended for the retrieval of k nearest neighbors (kNN). Furthermore, it is optimal (it visits only the nodes necessary for obtaining the nearest neighbors) and incremental, i.e., it reports neighbors in ascending order of their distance to the query point, and can be applied when the number k of nearest neighbors to be retrieved is not known in advance. An intersection join retrieves all intersecting object pairs (s,t) from two datasets S and T. If both S and T are indexed by R-trees, the R- tree join algorithm [BKS93] traverses synchronously the two trees, following entry pairs that overlap; non-intersecting pairs cannot lead to solutions at the lower levels. Several spatial join algorithms have been proposed for the case where only one of the inputs is indexed by an R-tree or no input is indexed [RSV02]. For point datasets, where intersection joins are meaningless, the corresponding problem is the e distance join, which finds all pairs of objects (s,t) $s \in S$, $t \in T$ within (Euclidean) distance e from each other. R-tree join can be applied in this case as well, the only difference being that a pair of intermediate entries is followed if their distance is below (or equal to) e. The intersection join can be considered as a special case of the e-distance join, where e=0. Finally, a closest-pairs query outputs the ($k \geq 1$) pairs of objects (s,t) $s \in S$, $t \in T$ with the smallest (Euclidean) distance. The algorithms for processing suchqueries [CMTV00] combine spatial joins with nearest neighbor search. In particular, assuming that both datasets are indexed by R-trees, the trees aretraversed synchronously, following the entry pairs with the minimum distance. Pruning is based on the mindist metric, but this time defined between entry MBRs. As all these algorithms apply only location-based metrics to prune the search space, they are inapplicable for SNDB.

5.2.17 LET US SUM UP

Thus, we have studied basic concepts of object oriented database, object identity, encapsulation, methods, persistence and inheritance. With

this ODMG as language design mode, ODL (object Definition language) and OQL query language. Also the major aspect of spatial and temporal database, in short GIS as well.

5.2.18 LIST OF REFERENCES

- Distributed Database; Principles & Systems By Publications, Stefano Ceri and Giuseppo Pelagatti,, McGraw-Hill International Editions (1984)

- Database Management Systems, 3rd edition, Raghu Ramakrishnan and Johannes Gehrke, McGraw-Hill (2002).

- Fundamentals of Database Systems, 6thEdition, Elmasri and Navathe, Addison. Wesley (2003).
- Unifying temporal data models via a conceptual model, C.S. Jensen, M.D. Soo, and R.T. Snodgrass: Information Systems, vol. 19, no. 7, pp. 513-547, 1994.

- Spatial Databases: A Tour by Shashi Shekhar and Sanjay Chawla, Prentice Hall, 2003 (ISBN 013-017480-7)

- Principles of Multimedia Database Systems, Subramanian V. S. Elsevier Publishers, 2013.

5.2.19 UNIT END EXERCISES

1) Explain features of object oriented database with an example.

2) Describe in detail temporal databases.

3) What is a Geographical Information system? Explain Different format used to represent geographic data.

4) Write a short note on Spatial database

5) What is GIS? Explain its application.

6) Explain conceptual and logical data model for spatial databases.

7) Explain ODMG model.

❖❖❖❖

6

DEDUCTIVE, ACTIVE, MULTIMEDIA AND XML DATABASES

Unit Structure

6.1.0 OBJECTIVES

In this chapter you will learn about:

- Introduction to deductive database.
- Datalog notation, clause form and horn clauses etc.
- Basics of active database and XML database-structure schema.

- XML data storage, querying and transformation etc.
- Introduction to multimedia database system.

6.1.1 INTRODUCTION

This chapter introduces database concepts for some of the common features that are needed by advanced applications and are being used widely. We will cover *active rules* that are used in active database applications. We will also discuss *deductive databases.* It is important to note that each of these topics is very broad, and we give only a brief introduction to each.

We discuss deductive databases, an area that is at the intersection of databases, logic, and artificial intelligence or knowledge bases. A **deductive database system** includes capabilities to define **(deductive) rules**, which can deduce or infer additional information from the facts that are stored in a database. Because part of the theoretical foundation for some deductive database systems is mathematical logic, such rules are often referred to as **logic databases**. Other types of systems, referred to as **expert database systems** or **knowledge-based systems**, also incorporate reasoning and inferencing capabilities; such systems use techniques that were developed in the field of artificial intelligence, including semantic networks, frames, production systems, or rules for capturing domain-specific knowledge.

Also Multimedia databases provide features that allow users to store and query different types of multimedia information, which includes images (such as pictures and drawings), video clips (such as movies,newsreels, and home videos), audio clips (such as songs, phone messages, and speeches), and documents (such as books and articles).

6.1.2 DEDUCTIVE DATABASE

In a deductive database system we typically specify rules through a **declarative language** a language in which we specify what to achieve rather than how to achieve it. An **inference engine** (or **deduction mechanism**) within the system can deduce new facts from the database by interpreting these rules. The model used for deductive databases is closely related to the relational data model, and particularly to the domain relational calculus formalism. It is also related to the field of **logic programming** and the **Prolog** language. The deductive database work based on logic has used Prolog as a starting point. A variation of Prolog called **Datalog** is used to define rules declaratively in conjunction with an existing set of relations, which are themselves treated as literals in the language. Although the language structure of Datalog resembles that of Prolog, its operational semantics that is, how a Datalog program is executed is still different.

A deductive database uses two main types of specifications: facts and rules. **Facts** are specified in a manner similar to the way relations are specified, except that it is not necessary to include the attribute names. Recall that a tuple in a relation describes some real-world fact whose meaning is partly determined by the attribute names. In a deductive database, the meaning of an attribute value in a tuple is determined solely by its *position* within the tuple. **Rules** are somewhat similar to relational views. They specify virtual relations that are not actually stored but that can be formed from the facts by applying inference mechanisms based on the rule specifications.

The main difference between rules and views is that rules may involve recursion and hence may yield virtual relations that cannot be defined in terms of basic relational views. The evaluation of Prolog programs is based on a technique called *backward chaining*, which involves a top-down evaluation of goals. In the deductive databases that use Datalog, attention has been devoted to handling large volumes of data stored in a relational database. Hence, evaluation techniques have been devised that resemble those for a bottom-up evaluation. Prolog suffers from the limitation that the order of specification of facts and rules is significant in evaluation; moreover, the order of literals within a rule is significant. The execution techniques for Datalog programs attempt to circumvent these problems.

6.1.3 INTRODUCTION TO RECURSIVE QUERIES

As begin with a simple example that illustrates the its of SQL-92 queries with the power of recursive definitions. Let Assembly be a relation with three fields *part, subpart,* and *qty.* An example instance of Assembly is shown in Figure 4.1.1. Each tuple in Assembly indicates How many copies of a particular subpart are Contained in a given part. The first tuple indicates, for example, that (1, trike contains three wheels} The Assembly relation can be visualized as a tree, as shown in Figure. A. tuple is shown as an edge going from the part to the subpart, with the *qty* value as the edge label

part	*subpart*	*qty*
trike	\vheel	3
trike	fralne	1
frarne	seat	1
franle	pedal	1
wheel	spoke	2
\vheel	tire	1
tire	run	1
tire	tube	1

Figure 6.1.1 An instance of assembly

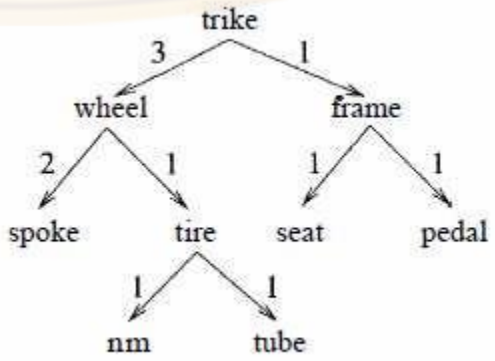


Figure 6.1.2 Assembly instance seen as a Tree

A natural question to ask is, "What are the components of a trike?" Rather surprisingly, this query is impossible to write in SQL-92. Of course, if we look at a given instance of the Assembly relation, we can write a 'query' that takes the union of the parts that are used in a trike. But such a query is not interesting---we want a query that identifies all components of a trike for *any* instance of Assembly, and such a query cannot be written in relational algebra or in SQL-92. Intuitively, the problem is that we are forced to join the Assembly relation with itself to recognize that *trike* contains *spoke* and *tire*, that is, to go one level down the Assembly tree. For each additional level, we need an additional join; two joins are needed to recognize that *trike* contains *rim*, which is asubpart of *tire*. Thus, the number of joins needed to identify all subparts of*trike* depends on the height of the Assembly tree, that is, on the given instance of the Assembly relation. No relational algebra query works forall instances; given any query, we can construct an instance whose height is greater than the number of joins in the query.

6.1.4 DATALOG NOTATION

In Datalog, as in other logic-based languages, a program is built from basic objects called **atomic formulas**. It is customary to define the syntax of logic-based languages by describing the syntax of atomic formulas and identifying how they can be combined to form a program. In Datalog, atomic formulas are **literals** of the form $p(a1, a2, ..., an)$, where p is the predicate name and n is the number of arguments for predicate p. Different predicate symbols can have different numbers of arguments, and the number of arguments n of predicate p is sometimes called the **arity** or **degree** of p. The arguments can be either constant values or variable names. As mentioned earlier, we use the convention that constant values either are numeric or start with a *lowercase* character, whereas variable names always start with an *uppercase* character.

A number of **built-in predicates** are included in Datalog, which can also be used to construct atomic formulas. The built-in predicates are of two main types: the binary comparison predicates < (less), <= (less_or_equal), > (greater), and >= (greater_or_equal) over ordered domains; and the comparison predicates = (equal) and /= (not_equal) over ordered or unordered domains. These can be used as binary predicates with the same functional syntax as other predicates for example, by writing less(X, 3) or they can be specified by using the customary infix notation $X<3$. Note that because the domains of these predicates are potentially infinite, they should be used with care in rule definitions. For example, the predicate greater(X, 3), if used alone, generates an infinite setof values for X that satisfy the predicate (all integer numbers greater than 3).

A **literal** is either an atomic formula as defined earlier called a **positive literal**—or an atomic formula preceded by **not**. The latter is a negated atomic formula, called a **negative literal**. Datalog programs can

be considered to be a *subset* of the predicate calculus formulas, which are somewhat similar to the formulas of the domain relational calculus. In Datalog, however, these formulas are first converted into what is known as **clausal form** before they are expressed in Datalog, and only formulas given in a restricted clausal form, called *Horn clauses* can be used in Datalog.

6.1.5 CLAUSE FORM AND HORN CLAUSES

A formula in the relational calculus is a condition that includes predicates called *atoms* (based on relation names). Additionally, a formula can have quantifiers namely, the *universal quantifier* (for all) and the *existential quantifier* (there exists). In clausal form, a formula must be transformed into another formula with the following characteristics:

- All variables in the formula are universally quantified. Hence, it is not necessary to include the universal quantifiers (for all)explicitly; the quantifiers are removed, and all variables in the formula are *implicitly* quantified by the universal quantifier.

- In clausal form, the formula is made up of a number of clauses, where each **clause** is composed of a number of *literals* connected by OR logical connectives only. Hence, each clause is a*disjunction* of literals.

- The *clauses themselves* are connected by AND logical connectives only, to form a formula. Hence, the **clausal form of a formula** is a *conjunction* of clauses.

It can be shown that *any formula can be converted into clausal form.* For our purposes, we are mainly interested in the form of the individual clauses, each of which is a disjunction of literals. Recall that literals can be positive literals or negative literals. Consider a clause of the form:

$$\text{NOT}(P_1) \text{ OR } \text{NOT}(P_2) \text{ OR } \ldots \text{ OR } \text{NOT}(P_n) \text{ OR } Q_1 \text{ OR } Q_2 \text{ OR } \ldots \text{ OR } Q_m \qquad (1)$$

This clause has *n* negative literals and *m* positive literals. Such a clause can be transformed into the following equivalent logical formula:

$$P_1 \text{ AND } P_2 \text{ AND } \ldots \text{ AND } P_n \Rightarrow Q_1 \text{ OR } Q_2 \text{ OR } \ldots \text{ OR } Q_m \qquad (2)$$

where ⇒ is the implies symbol. The formulas (1) and (2) are equivalent, meaning that their truth values are always the same. This is the case because if all the *Pi* literals (*i* = 1, 2, ..., *n*) are true, the formula (2) is true only if at least one of the Qi's is true, which is the meaning of the ⇒ (implies) symbol. For formula (1), if all the Pi literals (i = 1, 2, ..., n) are true, their negations are all false; so in this case formula (1) is true only if at least one of the *Qi*'s is true. In Datalog, rules are expressed as a

restricted form of clauses called **Horn clauses**, in which a clause can contain *at most one* positive literal. Hence, a Horn clause is either of the form

$$\text{NOT } (P_1) \text{ OR NOT}(P_2) \text{ OR } \ldots \text{ OR NOT}(P_n) \text{ OR } Q \quad (3)$$

or of the form

$$\text{NOT } (P_1) \text{ OR NOT}(P_2) \text{ OR } \ldots \text{ OR NOT}(P_n) \quad (4)$$

The Horn clause in (3) can be transformed into the clause

$$P_1 \text{ AND } P_2 \text{ AND } \ldots \text{ AND } P_n \Rightarrow Q \quad (5)$$

which is written in Datalog as the following rule:

$$Q :- P_1, P_2, \ldots, P_n. \quad (6)$$

The Horn clause in (4) can be transformed into

$$P_1 \text{ AND } P_2 \text{ AND } \ldots \text{ AND } P_n \Rightarrow \quad (7)$$

which is written in Datalog as follows:

$$P_1, P_2, \ldots, P_n. \quad (8)$$

A **Datalog rule**, as in (6), is hence a Horn clause, and its meaning, based on formula (5), is that if the predicates *P*1 AND *P*2 AND ... AND*Pn* are all true for a particular binding to their variable arguments, then *Q*is also true and can hence be inferred. The Datalog expression (8) can be considered as an integrity constraint, where all the predicates must be true to satisfy the query. In general, a **query in Datalog** consists of two components:

- A Datalog program, which is a finite set of rules
- A literal *P*(*X*1, *X*2, ..., *Xn*), where each *Xi* is a variable or a constant

A Prolog or Datalog system has an internal **inference engine** that can be used to process and compute the results of such queries. Prolog inference engines typically return one result to the query (that is, one set of values for the variables in the query) at a time and must be prompted to return additional results. On the contrary, Datalog returns results set-at-a-time.

6.1.6 INTERPRETATION OF MODEL

There are two main alternatives for interpreting the theoretical meaning of rules: *proof-theoretic* and *model-theoretic.* In practical systems, the inference mechanism within a system defines the exact interpretation, which may not coincide with either of the two theoretical interpretations. The inference mechanism is a computational procedure and hence provides a computational interpretation of the meaning of rules.In this section, first we discuss the two theoretical interpretations. Then we briefly discuss inference mechanisms as a way of defining the meaning of

rules. In the **proof-theoretic** interpretation of rules, we consider the facts and rules to be true statements, or **axioms**. **Ground axioms** contain no variables. The facts are ground axioms that are given to be true. Rules are called **deductive axioms**, since they can be used to deduce new facts. The deductive axioms can be used to construct proofs that derive new facts from existing facts. For example, Figure 4.1.3 shows how to prove the fact SUPERIOR(james, ahmad) from the rules and facts given in Figure.

1. SUPERIOR(*X*, *Y*) :– SUPERVISE(*X*, *Y*).	(rule 1)
2. SUPERIOR(*X*, *Y*) :– SUPERVISE(*X*, *Z*), SUPERIOR(*Z*, *Y*).	(rule 2)
3. SUPERVISE(jennifer, ahmad).	(ground axiom, given)
4. SUPERVISE(james, jennifer).	(ground axiom, given)
5. SUPERIOR(jennifer, ahmad).	(apply rule 1 on 3)
6. SUPERIOR(james, ahmad).	(apply rule 2 on 4 and 5)

Figure 6.1.3 Proving a new fact

The proof-theoretic interpretation gives us a procedural or computational approach for computing an answer to the Datalog query. The process of proving whether a certain fact (theorem) holds is known as **theorem proving**.

The second type of interpretation is called the **model-theoretic** interpretation. Here, given a finite or an infinite domain of constant values, we assign to a predicate every possible combination of values as arguments. We must then determine whether the predicate is true or false. In general, it is sufficient to specify the combinations of arguments that make the predicate true, and to state that all other combinations make the predicate false. If this is done for every predicate, it is called an **interpretation** of the set of predicates. For example, consider the interpretation shown in Figure6.1.4 for the predicates SUPERVISE and SUPERIOR. This interpretation assigns a truth value (true or false) to every possible combination of argument values (from a finite domain) for the two predicates.

An interpretation is called a **model** for a *specific set of rules* if those rules are *always true* under that interpretation; that is, for any values assigned to the variables in the rules, the head of the rules is true when we substitute the truth values assigned to the predicates in the body of the rule by that interpretation. Hence, whenever a particular substitution (binding) to the variables in the rules is applied, if all the predicates in the body of a rule are true under the interpretation, the predicate in the head of the rule must also be true. The interpretation shown in Figure is a model for the two rules shown, since it can never cause the rules to be violated. Notice that a rule is violated if a particular binding of constants to the variables makes all the predicates in the rule body true but makes the predicate in the rule head false. For example, if SUPERVISE(*a, b*) and SUPERIOR(*b, c*) are both true under some interpretation, but SUPERIOR(*a, c*) is not true, the interpretation cannot be a model for the recursive rule:

SUPERIOR(*X*, *Y*) :– SUPERVISE(*X*, *Z*), SUPERIOR(*Z*, *Y*)

In the model-theoretic approach, the meaning of the rules is established by providing a model for these rules. A model is called a **minimal model** for a set of rules if we cannot change any fact from true to false and still get a model for these rules. For example, consider the interpretation in Figure, and assume that the SUPERVISE predicate is defined by a set of known facts, whereas the SUPERIOR predicate is defined as an interpretation (model) for the rules. Suppose that we add the predicate SUPERIOR(james, bob) to the true predicates. This remains a model for the rules shown, but it is not a minimal model, since changing the truth value of SUPERIOR(james,bob) from true to false still provides us with a model for the rules. The model shown in Figure is the minimal model for the set of facts that are defined by the SUPERVISE predicate. Ingeneral, the minimal model that corresponds to a given set of facts in the model theoretic interpretation should be the same as the facts generated bythe proof.

Rules
SUPERIOR(*X*, *Y*) :– SUPERVISE(*X*, *Y*).
SUPERIOR(*X*, *Y*) :– SUPERVISE(*X*, *Z*), SUPERIOR(*Z*, *Y*).

Interpretation

Known Facts:
SUPERVISE(franklin, john) is **true**.
SUPERVISE(franklin, ramesh) is **true**.
SUPERVISE(franklin, joyce) is **true**.
SUPERVISE(jennifer, alicia) is **true**.
SUPERVISE(jennifer, ahmad) is **true**.
SUPERVISE(james, franklin) is **true**.
SUPERVISE(james, jennifer) is **true**.
SUPERVISE(*X*, *Y*) is **false** for all other possible (*X*, *Y*) combinations

Derived Facts:
SUPERIOR(franklin, john) is **true**.
SUPERIOR(franklin, ramesh) is **true**.
SUPERIOR(franklin, joyce) is **true**.
SUPERIOR(jennifer, alicia) is **true**.
SUPERIOR(jennifer, ahmad) is **true**.
SUPERIOR(james, franklin) is **true**.
SUPERIOR(james, jennifer) is **true**.
SUPERIOR(james, john) is **true**.
SUPERIOR(james, ramesh) is **true**.
SUPERIOR(james, joyce) is **true**.
SUPERIOR(james, alicia) is **true**.
SUPERIOR(james, ahmad) is **true**.
SUPERIOR(*X*, *Y*) is **false** for all other possible (*X*, *Y*) combinations

Figure 6.1.4 An interpretation that is a minimal model.

Theoretic interpretation for the same original set of ground and deductive axioms. However, this is generally true only for rules with a simple structure. Once we allow negation in the specification of rules, the correspondence between interpretations *does not* hold. In fact, with negation, numerous minimal models are possible for a given set of facts.

A third approach to interpreting the meaning of rules involves defining an inference mechanism that is used by the system to deduce facts from the rules. This inference mechanism would define a **computational interpretation** to the meaning of the rules. The Prolog logic programming language uses its inference mechanism to define the

meaning of the rules and facts in a Prolog program. Not all Prolog programs correspond to the proof-theoretic or model-theoretic interpretations; it depends on the type of rules in the program. However, for many simple Prolog programs, the Prolog inference mechanism infers the facts that correspond either to the proof-theoretic interpretation or to a minimal model under the model-theoretic interpretation.

6.1.7 LEAST MODEL SEMANTICS

We want users to be able to understand a Datalog prograrn by understanding each rule independent of other rules, with : *If the body is True, the head* is *also True.* This intuitive reading of a rule suggests that, given certain relation instances for the relation names that appear in the body of a rule, the relation instance for the relation mentioned in the head of the rule contain a certain set of tuples. If a relation Harne R. appears in the heads of several rules, the relation instance for R satisfy the intuitive reading of all these rules. However, we do not want tuples to be included in the instance for R, unless they are necessary to satisfy one of the rules defining R,. That is, we want to compute only tuples for R that aresupported by Salne rule for R. To these ideas precise, we need to introducethe concepts of models and least models. A model is a collection ofrelation instances, one instance for each relation in the program, thatsatisfies the following condition. For every rule in the program, whenever we replace each variable in the rule by a corresponding constant, the following holds:

If every tuple in the body (obtained by our replacement of variables with constants) is in the corresponding relation instance, *Then*the tuple generated for the head (by the assignment of constants to variables that appear in the head) is also in the corresponding relation instance.
Observe that the instances for the input relations are given, and the definition of a model essentially restricts the instances for the output relations. Consider the rule

```
Components(Part, Subpart) '- Assembly(Part, Part2, Qty),
                             Components(Part2, Subpart).
```

Suppose we replace the variable Part by the constant *wheel,* Part2 by *tire,* Qty by 1, and Subpart by *rim*:

```
Components(wheel, rim) '- Assembly(wheel, tire, 1),
                          Components(tire, rim).
```

Let A be an instance of Assembly and C be an instance of components. If A contains the tuple (wheel,tire, 1) and C contains the tuple *(tire, rim,),* then C tuple also contain the tuple *(wheel, rim)* for the pair of instances A. and C to be a model. of course, the instances A and must satisfy the inclusion requirement just illustrated for *every* assignmentof constants to the variables in the rule: If the tuples in the rule body are inA and C, the tuple in the head to be in C.

As an example, the instances of Assembly shown in Figure and Components had shown in Figure together form a model for the component program. Given the instance of Assembly shown in Figure, there is no justification for including the tuple (spoke, *pedal)* to the Components instance. Indeed, if we add this tuple to the componentsinstance in Figure, *We* no longer have a model for our program, as the following instance of the recursive rule demonstrates, since *(wheel, pedal)* is not in the Components instance:

```
Components(wheel, pedal) :-  Assembly(wheel, spoke, 2),
                             Components(spoke, pedal).
```

However, by also adding the tuple *(wheel, pedal)* to the Components instance, we obtain another model of the Components program. Intuitively, this is unsatisfactory since there is no justification foradding the tuple *(spoke, pedal)* in the first place, given the tuples in the Assembly instance and the rules in the program.

We address this problem by using the concept of a least model. A least model of a program is a model M such that for every other model M2 of the same program, for each relation R in the program the instance for R is contained in the instance of R. The 1nodel formed by the instances of Assembly and Components shown in Figures and is the least model for the components program with the given Assembly instance.

6.1.8 THE FIXED POINT OPERATOR

A fixpoint of a function *f* is a value *v* such that the function applied to the value returns the same value, that is, *f(v) = 'U.* Consider a function applied to a set of values that also returns a set of values. For example, we can define *double* to l)e a function that multiplies every element of the input set by two and double+ to be *double* U *identity.* T'hus, *double(* {1,2,5}) == {2,4,lO}, and *double+(* {1,2,5}) ::::::: {1,2,4.,5,lO}.The set of all even integers which happens to be an infinite set-is a fixpoint of the function *double*-+. Another fixpoint of the function double+ is the set of all integers. The first fixpoint (the set of all (even integers) is smaller than the second fixpoint (the set of all integers) because it is contained in the latter.

The least fixpoint of a function is the fixpoint that is smaller than every other fixpoint of that function. In general, it is not guaranteed that a function has a load fixpoint. For example, there may be two fixpoints, neither of which is smaller than the other. (Does *double* have a least fixpoint? What is it?) No let us turn to functions over sets of tuples, in particular, functions defined using relational algebra expressions. The Components relation can be defined by an equation.

$$Components = \pi_{1,5}(Assembly \bowtie_{2=1} Components) \cup \pi_{1,2}(Assembly)$$

1'his equation has the form1

$$Cornponents = f(Cbrnponents, Assembly)$$

where the function f is defined using a relational aJgebra expression. For a given instance of the input relation Assernbly, this can be simplified to

$$Components = f(Co1nponents)$$

The least fixpoint of *f* is an instance of Components that satisfies this equation. Clearly the projection of the first two fields of the tuples in the given instance of the input relation Assembly rnust be included in the (instance that is the) least fixpoint of Components. In addition, any tuple obtained by joining Components with Assembly and projecting the appropriate fields must also be in Components.

A little thought shows that the instance of Components that is the least fixpoint of *f* can be computed using repeated applications of the Datalog rules shown in the previous section. Indeed, applying the two Datalog rules is identical to evaluating the relational expression used in defining components. If an application generates Components tuples that are not in the current instance of the Components relation, the current instance cannot be the fixpoint.

Therefore, we add the new tuples to Components <tnd evalu<te the relational expression (equivalently, the two Datalog rules) again. This process is repeated until every tuple generated is already in the current instance of Components. Then applying the rules to the current set of tuples does not produce any new tuples, we have reached a fixpoint. If components is initialized to the empty set of tuples. We infer only tuples that we necessary by the definition of a fixpoint, and the fixpoint computed is the least fixpoint.

6.1.9 SAFE DATALOG PROGRAM

There are two main methods of defining the truth values of predicates in actual Datalog programs. **Fact-defined predicates** (or **relations**) are defined by listing all the combinations of values (the tuples) that make the predicate true. These correspond to base relations whose contents are stored in a database system. Figure shows the fact-defined predicates EMPLOYEE, MALE, FEMALE, DEPARTMENT, SUPERVISE, PROJECT, and WORKS_ON, which correspond to part of the relational database shown in Figure 4.1.5 **Rule-defined predicates** (or **views**) are defined by being the head (LHS) of one or more Datalog rules; they correspond to *virtual relations* whose contents can be inferred by the inference engine.

Figure shows a number of rule-defined predicates. A program or a rule is said to be **safe** if it generates a *finite* set of facts. The general

theoretical problem of determining whether a set of rules is safe is undecidable.

However, one can determine the safety of restricted forms of rules. For example, the rules shown in Figure are safe. One situation where we get unsafe rules that can generate an infinite number of facts arises when one of the variables in the rule can range over an infinite domain of values, and that variable is not limited to ranging over a finite relation. Forexample, consider the following rule:

BIG_SALARY(Y) :– Y>60000

Here, we can get an infinite result if Y ranges over all possible integers. But suppose that we change the rule as follows:

BIG_SALARY(Y) :– EMPLOYEE(X), Salary(X, Y), Y>60000

In the second rule, the result is not infinite, since the values that Y can be bound to are now restricted to values that are the salary of some employee in the database—presumably, a finite set of values. We can also rewrite the rule as follows:

BIG_SALARY(Y) :– Y>60000, EMPLOYEE(X), Salary(X, Y)

```
EMPLOYEE(john).
EMPLOYEE(franklin).
EMPLOYEE(alicia).
EMPLOYEE(jennifer).
EMPLOYEE(ramesh).
EMPLOYEE(joyce).
EMPLOYEE(ahmad).
EMPLOYEE(james).

SALARY(john, 30000).
SALARY(franklin, 40000).
SALARY(alicia, 25000).
SALARY(jennifer, 43000).
SALARY(ramesh, 38000).
SALARY(joyce, 25000).
SALARY(ahmad, 25000).
SALARY(james, 55000).

DEPARTMENT(john, research).
DEPARTMENT(franklin, research).
DEPARTMENT(alicia, administration).
DEPARTMENT(jennifer, administration).
DEPARTMENT(ramesh, research).
DEPARTMENT(joyce, research).
DEPARTMENT(ahmad, administration).
DEPARTMENT(james, headquarters).

SUPERVISE(frankIln, john).
SUPERVISE(frankIln, ramesh)
SUPERVISE(frankin , joyce).
SUPERVISE(jennifer, alicia).
SUPERVISE(jennifer, ahmad).
SUPERVISE(james, franklin).
SUPERVISE(james, jennifer).
```

```
MALE(john).
MALE(franklin).
MALE(ramesh).
MALE(ahmad).
MALE(james).

FEMALE(alicia).
FEMALE(jennifer).
FEMALE(joyce).

PROJECT(productx).
PROJECT(producty).
PROJECT(productz).
PROJECT(computerization).
PROJECT(reorganization).
PROJECT(newbenefits).

WORKS_ON(john, productx, 32).
WORKS_ON(john, producty, 8).
WORKS_ON(ramesh, productz, 40).
WORKS_ON(joyce, productx, 20).
WORKS_ON(joyce, producty, 20).
WORKS_ON(franklin, producty, 10).
WORKS_ON(franklin, productz, 10).
WORKS_ON(franklin, computerization, 10).
WORKS_ON(franklin, reorganization, 10).
WORKS_ON(alicia, newbenefits, 30).
WORKS_ON(alicia, computerization, 10).
WORKS_ON(ahmad, computerization, 35).
WORKS_ON(ahmad, newbenefits, 5).
WORKS_ON(jennifer, newbenefits, 20).
WORKS_ON(jennifer, reorganization, 15).
WORKS_ON(james, reorganization, 10).
```

Figure 6.1.5 Fact predicates for part of the database

```
SUPERIOR(X, Y) :– SUPERVISE(X, Y).
SUPERIOR(X, Y) :– SUPERVISE(X, Z), SUPERIOR(Z, Y).

SUBORDINATE(X, Y) :– SUPERIOR(Y, X).

SUPERVISOR(X) :– EMPLOYEE(X), SUPERVISE(X, Y).
OVER_40K_EMP(X) :– EMPLOYEE(X), SALARY(X, Y), Y >= 40000.
UNDER_40K_SUPERVISOR(X) :– SUPERVISOR(X), NOT(OVER_40_K_EMP(X)).
MAIN_PRODUCTX_EMP(X) :– EMPLOYEE(X), WORKS_ON(X, productx, Y), Y >=20.
PRESIDENT(X) :– EMPLOYEE(X), NOT(SUPERVISE(Y, X) ).
```

Figure 6.1.6 Rule-defined predicates

In this case, the rule is still theoretically safe. However, in Prolog or any other system that uses a top-down, depth-first inference mechanism, the rule creates an infinite loop, since we first search for a value for Y and then check whether it is a salary of an employee. The result is generation of an infinite number of Y values, even though these, after a certain point, cannot lead to a set of true RHS predicates. One definition of Datalog considers both rules to be safe, since it does not depend on a particular inference mechanism. Nonetheless, it is generally advisable to write such a rule in the safest form, with the predicates that restrict possible bindings of variables placed first. As another example of an unsafe rule, consider the following rule:

```
HAS_SOMETHING(X, Y ) :– EMPLOYEE(X )
```

```
REL_ONE(A, B, C ).
REL_TWO(D, E, F).
REL_THREE(G, H, I, J ).

SELECT_ONE_A_EQ_C(X, Y, Z) :– REL_ONE(C, Y, Z).
SELECT_ONE_B_LESS_5(X, Y, Z) :– REL_ONE(X, Y, Z), Y< 5.
SELECT_ONE_A_EQ_C_AND_B_LESS_5(X, Y, Z) :– REL_ONE(C, Y, Z), Y<5

SELECT_ONE_A_EQ_C_OR_B_LESS_5(X, Y, Z) :– REL_ONE(C, Y, Z).
SELECT_ONE_A_EQ_C_OR_B_LESS_5(X, Y, Z) :– REL_ONE(X, Y, Z), Y<5.

PROJECT_THREE_ON_G_H(W, X) :– REL_THREE(W, X, Y, Z).

UNION_ONE_TWO(X, Y, Z) :– REL_ONE(X, Y, Z).
UNION_ONE_TWO(X, Y, Z) :– REL_TWO(X, Y, Z).

INTERSECT_ONE_TWO(X, Y, Z) :– REL_ONE(X, Y, Z), REL_TWO(X, Y, Z).

DIFFERENCE_TWO_ONE(X, Y, Z) :– REL_TWO(X, Y, Z) NOT(REL_ONE(X, Y, Z).

CART_PROD _ONE_THREE(T, U, V, W, X, Y, Z) :–
    REL_ONE(T, U, V), REL_THREE(W, X, Y, Z).

NATURAL_JOIN_ONE_THREE_C_EQ_G(U, V, W, X, Y, Z) :–
    REL_ONE(U, V, W), REL_THREE(W, X, Y, Z).
```

Figure 6.1.6 Predicates for illustrating relational operations

Here, an infinite number of *Y* values can again be generated, since the variable *Y* appears only in the head of the rule and hence is not limited to a finite set of values. To define safe rules more formally, we use the concept of a limited variable. A variable *X* is **limited** in a rule if (1) it appears in a regular (not built-in) predicate in the body of the rule; (2) it appears in a predicate of the form *X*=*c* or *c*=*X* or (*c*1<<=*X* and *X*<=*c*2) in the rule body, where *c*, *c*1, and *c*2 are constant values; or (3) it appears in a predicate of the form *X*=*Y* or *Y*=*X* in the rule body, where *Y* is a limited variable. A rule is said to be **safe** if all its variables are limited.

6.1.10 RECURSIVE QUERY WITH NEGATION

Unfortunately, once set-difference is allowed in the body of a rule, there may be no least , model or least fixpoint for a program. Consider the following rules:

```
Big(Part):-     Assembly(Part, Subpart, Qty), Qty> 2,
                NOT Small(Part).
Small(Part) :-  Assembly(Part, Subpart, Qty), NOT Big(Part).
```

These two rules can be thought of as an attempt to divide parts (those that are mentioned in the first column of the Assembly table) into two classes, Big and Small. The first rule defines Big to be the set of parts that use at least three copies of some subpart and are not classified as small parts. The second rule defines Small as the set of parts not classified as big parts.

If we apply these rules to the instance of Assembly shown in Figure 4.1.7, *trike* is the only part that uses at least three copies of same subpart. Should the tuple *(trike)* be in Big or Small? If we apply the first rule and then the second rule, this tuple is in Big. To apply the first rule,we consider the tuples in Assembly, choose those with Qty > 2 (which is just *(trike))*, discard those in the current instance of Srnal1 (both Big and Small are initially empty), and add the tuples that are left to Big. Therefore, an application of the first rule adds *(trike)* to Big. Proceeding similarly, we can see that if the second rule is applied before the first, *(trike)* is added to Small instead of Big. This program has two fixpoints, neither of which is smaller than the other, as shown in Figure 4.1.7. (The first fixpoint has a Big tuple that does not appear in the second fixpoint; therefore, it is not smaller than the second fixpoint. The second fixpointhas a small tuple that does not appear in the first fixpoint. Therefore it is not smaller than the first fixpoint. The order in which we apply the rules determines which fixpoint is computed; this situation is very unsatisfactory. We want users to be able to understand their queries without thinking out exactly how the evaluation proceeds. The root of the problerH is the use of NOT. When we apply the first rule, same inferences are disallowed because of the presence of tuples in small.

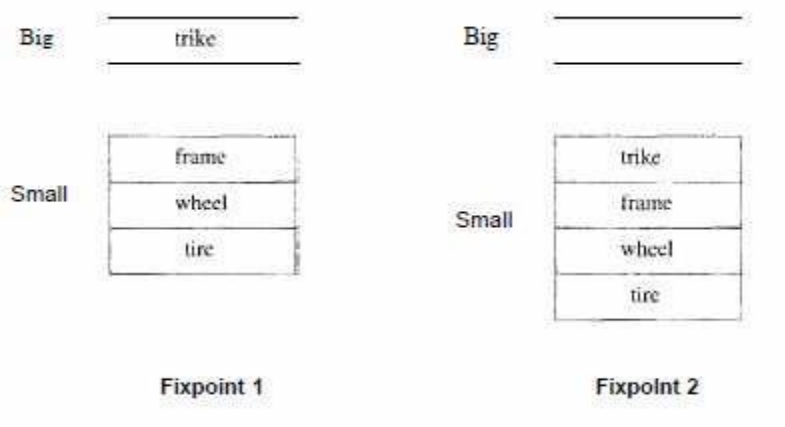


Figure 6.1.7 Two fix point for Big/Small program

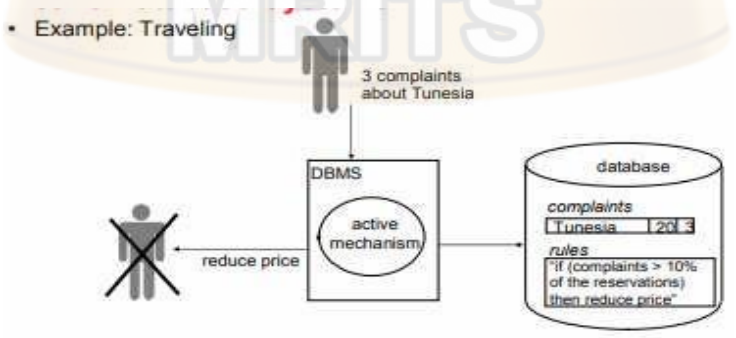
Parts that satisfy the other conditions in the body of the rule are candidates for addition to Big; we remove the parts in Small from this set of candidates. Thus some inferences that are possible if Small is empty (as it is before the second rule is applied) are disallowed if Small contains tuples (generated by applying the second rule before the first rule). Here isthe difficulty: If NOT is used, the addition of tuples to a relation can *disallow* the inference of other tuples. Without NOT, this situation can never arise; the addition of tuples to a relation can *never* disallow the inference of other tuples.

6.2 ACTIVE DATABASE

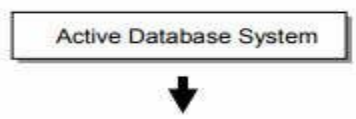
Overcome the strict separation between application programs and DBS.

• Usually only a small part of the real-world semantics can be modeled in the DBS.

• Object-oriented DBS are not enough => add active (and deductive) mechanisms to model more semantics (especially dynamic behavior) of the applications in DBS.



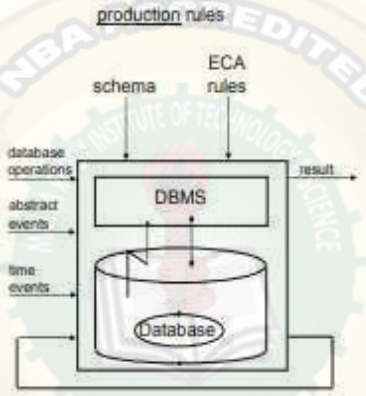
General Idea



In addition to the capabilities of passive database systems
• monitor specified situations (events & conditions) in the database or its environment
• invoke specified reactions whenever a situation occurs programs containing, e.g., database operations

Definition: Active DBS

An active database system (ADBS) is a DBS that monitors situations of interest and, when they occur, triggers an appropriate response in a timely manner. The desired behavior is expressed in production rules (also called event-condition-action rules), which are defined and stored in the DBS. This has the benefits that the rules can be shared by many application programs, and the DBS can optimize their implementation.



6.2.1 LANGUAGES FOR RULE SPECIFICATION

Rule Models and Languages
• Event Specification
• Condition Specification
• Action Specification
• Event-Condition-Action Binding
• Rule Ordering
• Rule Organization

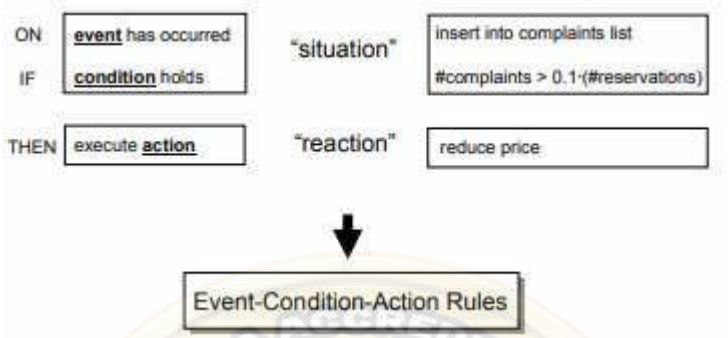
Affects data model and transaction management

6.2.2 EVENTS

- An event is something that happens at a point in time.
- Possible alternatives:
  - structure operation(insert,update,access)
  - behavior invocation(the message display is sent to an object of type widget)
  - transaction(abort,commit,begin-transacttion)

- exception(an attempt to access some data without appropriate authorization)
- clock(the first day of every month)
- external(the temperature reading goes above 30 degrees)

Production Rules (Event-Condition-Action Rules)

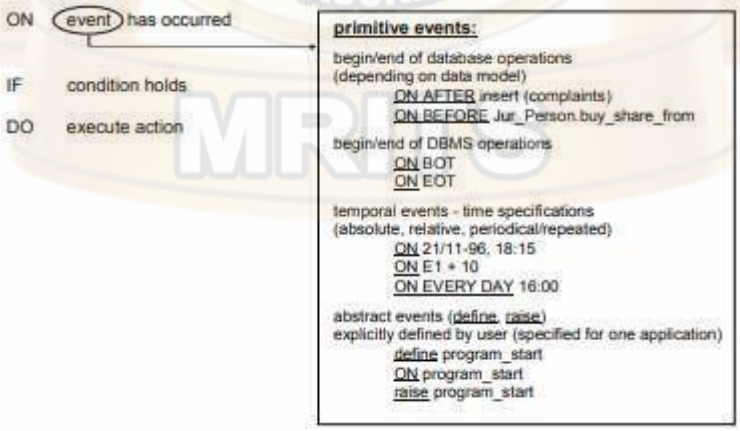


- Event Specification
- relational DBS: define rule MonitorNewEmps on insert to employee
  if ... then ...

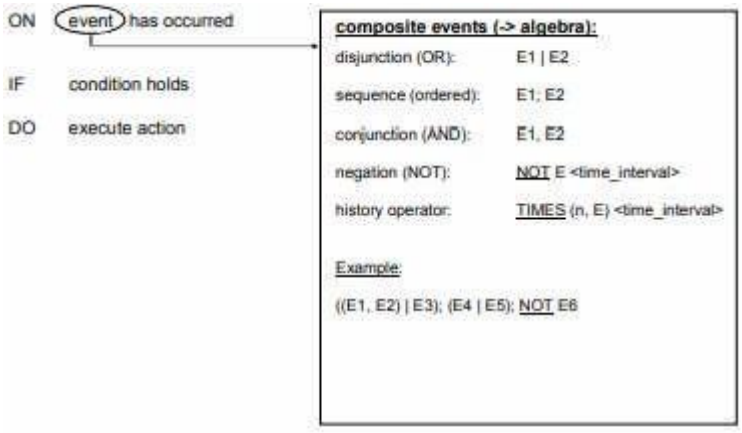
- OODBS: define rule CheckRaise on employee.salary-raise()
  if ... then ...

- rule triggered by data retrieval: define rule MonitorSalAccess on retrieve salary from employee
  if ... then ...

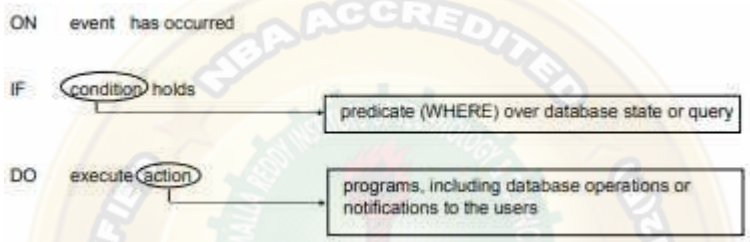
- Knowledge Model: Semantics of ECA Rules



- Knowledge Model: Semantics of ECA Rules – 2



- Knowledge Model: Semantics of ECA Rules – 3



6.2.3 CONDITIONS

- The condition indicates whether rule action should be executed.
- In ECA-rules, the condition is generally optional
- Once the triggering event has occurred, the condition may be evaluated. If condition evaluates to be true, the rule action will be executed.
- If no condition is specified, the action will be executed once the event occurs.
- The condition part of a rule specifies a predicate or query over the data in the database.
- The condition is satisfied if the predicate is true or if the query returns a nonempty answer.
- Explicit events: condition may often be omitted (in which case it is always satisfied).
- Transition conditions: allow to express conditions over changes in the database state. Example

```
define rule MonitorRaise
on update to employee.salary
if employee.salary > 1.1 * old employee.salary
then ...
```

6.2.4 ACTIONS

- The range of tasks that can be performed if the rule condition is evaluated to be true.
- It is usually a sequence of SQL statements.
- But actions may:
  - Perform some behavior invocation within the database or an external call
  - Inform the user or system administrator of some situation
  - Abort a transaction
  - Take some alternative course of action using do-instead

The action part of a production rule specifies the operations to be performed when the rule is triggered and its condition is satisfied.

```
define rule FavorNewEmps
on insert to employee
then delete employee e where e.name = employee.name
```

6.3 XML AND DATABASE

To understand XML, it is important to understand its roots as a document markup language. The term **markup** refers to anything in a document that is not intended to be part of the printed output. For example, a writer creating text that will eventually be typeset in a magazine may want to make notes about how the typesetting should be done. It would be important to type these notes in a way so that they couldbe distinguished from the actual content, so that a note like "set this word in large size, bold font" or "insert a line break here" does not end up printed in the magazine. Such notes convey extra information about the text.

In electronic document processing, a **markup language** is a formal description of what part of the document is content, what part is markup, and what the markup means. Just as database systems evolved from physical file processing to provide a separate logical view, markup languages evolved from specifying instructions for how to print parts of the document to specifying the *function* of the content. For instance, with functional markup, text representing section headings (for this section, the word "Motivation") would be marked up as being a section heading,instead of being marked up as text to be printed in large size, bold font. From the viewpoint of typesetting, such functional markup allows the document to be formatted differently in different situations. It also helps different parts of a large document, or different pages in a large Web site, to be formatted in a uniform manner. More importantly, functional markup also helps record what each part of the text represents semantically, and correspondingly helps automate extraction of key parts of documents. For the family of markup languages that includes HTML, SGML, and XML,

the markup takes the form of **tags** enclosed in angle brackets, <>. Tags are used in pairs, with <tag> and </tag> delimiting the beginning and the end of the portion of the document to which the tag refers. For example, the title of a document might be marked up as follows:

```
<title>Database System Concepts</title>
```

Unlike HTML, XML does not prescribe the set of tags allowed, and the set may be chosen as needed by each application. This feature is the key to XML's major role in data representation and exchange, whereasHTML is used primarily for document formatting.

```
<university>
    <department>
        <dept_name> Comp. Sci. </dept_name>
        <building> Taylor </building>
        <budget> 100000 </budget>
    </department>
    <department>
        <dept_name> Biology </dept_name>
        <building> Watson </building>
        <budget> 90000 </budget>
    </department>
    <course>
        <course_id> CS-101 </course_id>
        <title> Intro. to Computer Science </title>
        <dept_name> Comp. Sci </dept_name>
        <credits> 4 </credits>
    </course>
    <course>
        <course_id> BIO-301 </course_id>
        <title> Genetics </title>
        <dept_name> Biology </dept_name>
        <credits> 4 </credits>
    </course>
```

Figure 6.1.8 XML representation of (part of) university information.

```
    <instructor>
        <IID> 10101 </IID>
        <name> Srinivasan </name>
        <dept_name> Comp. Sci. </dept_name>
        <salary> 65000 </salary>
    </instructor>
    <instructor>
        <IID> 83821 </IID>
        <name> Brandt </name>
        <dept_name> Comp. Sci. </dept_name>
        <salary> 92000 </salary>
    </instructor>
    <instructor>
        <IID> 76766 </IID>
        <name> Crick </name>
        <dept_name> Biology </dept_name>
        <salary> 72000 </salary>
    </instructor>
    <teaches>
        <IID> 10101 </IID>
        <course_id> CS-101 </course_id>
    </teaches>
    <teaches>
        <IID> 83821 </IID>
        <course_id> CS-101 </course_id>
    </teaches>
    <teaches>
        <IID> 76766 </IID>
        <course_id> BIO-301 </course_id>
    </teaches>
</university>
```

Figure 6.1.9 Continuation of Figure

For example, in our running university application, department, course and instructor information can be represented as part of an XML document as in Figures 4.1.8 and 4.1.9. Observe the use of tags such as department, course, instructor, and teaches. To keep the example short, we use a simplified version of the university schema that ignores section information for courses. We have also used the tag IID to denote the identifier of the instructor, for reasons we shall see later.

These tags provide context for each value and allow the semantics of the value to be identified. For this example, the XML datarepresentation does not provide any significant benefit over the traditional relational data representation; however, we use this example as our running example because of its simplicity.

```
<purchase_order>
    <identifier> P-101 </identifier>
    <purchaser>
        <name> Cray Z. Coyote </name>
        <address> Mesa Flats, Route 66, Arizona 12345, USA </address>
    </purchaser>
    <supplier>
        <name> Acme Supplies </name>
        <address> 1 Broadway, New York, NY, USA </address>
    </supplier>
    <itemlist>
        <item>
            <identifier> RS1 </identifier>
            <description> Atom powered rocket sled </description>
            <quantity> 2 </quantity>
            <price> 199.95 </price>
        </item>
        <item>
            <identifier> SG2 </identifier>
            <description> Superb glue </description>
            <quantity> 1 </quantity>
            <unit-of-measure> liter </unit-of-measure>
            <price> 29.95 </price>
        </item>
    </itemlist>
    <totalLcost> 429.85 </totalLcost>
    <payment_terms> Cash-on-delivery </payment_terms>
    <shipping_mode> 1-second-delivery </shipping_mode>
</purchaseorder>
```

Figure 6.1.10 XML representation of a purchase order.

Figure 6.1.10, which shows how information about a purchaseorder can be represented in XML, illustrates a more realistic use of XML. Purchase orders are typically generated by one organization and sent to another. Traditionally they were printed on paper by the purchaser andsent to the supplier; the data would be manually re-entered into a computer system by the supplier. This slow process can be greatly sped up by sending the information electronically between the purchaser and supplier. The nested representation allows all information in a purchase order to be represented naturally in a single document. (Real purchase orders have considerably more information than that depicted in this simplified example.) XML provides a standard way of tagging the data; the two

organizations must of course agree on what tags appear in the purchase order, and what they mean.

Compared to storage of data in a relational database, the XML representation may be inefficient, since tag names are repeated throughout the document. However, in spite of this disadvantage, an XML representation has significant advantages when it is used to exchange data between organizations, and for storing complex structured information in files:

First, the presence of the tags makes the message **self-documenting**; that is, a schema need not be consulted to understand the meaning of the text. We can readily read the fragment above, for example.

Second, the format of the document is not rigid. For example, if some sender adds additional information, such as a tag last accessed noting the last date on which an account was accessed, the recipient of the XML data may simply ignore the tag. As another example, in Figure 4.1.10, the item with identifier SG2 has a tag called unit-of-measure specified, which the first item does not.

The tag is required for items that are ordered by weight or volume, and may be omitted for items that are simply ordered by number. The ability to recognize and ignore unexpected tags allows the format of the data to evolve over time, without invalidating existing applications.

Similarly, the ability to have multiple occurrences of the same tag makes it easy to represent multi valued attributes.

Third, XML allows nested structures. The purchase order shown in Figure 6.1.10 illustrates the benefits of having a nested structure. Each purchase order has a purchaser and a list of items as two of its nested structures. Each item in turn has an item identifier, description and a price nested within it, while the purchaser has a name and address nested within it. Such information would have been split into multiple relations in a relational schema. Item information would have been stored in one relation, purchaser information in a second relation, purchase orders in a third, and the relationship between purchase orders, purchasers, and items would have been stored in a fourth relation.

The relational representation helps to avoid redundancy; for example, item descriptions would be stored only once for each item identifier in a normalized relational schema. In the XML purchase order, however, the descriptions may be repeated in multiple purchase orders that order the same item. However, gathering all information related to a purchase order into a single nested structure, even at the cost of redundancy, is attractive when information has to be exchanged with external parties.

Finally, since the XML format is widely accepted, a wide variety of tools are available to assist in its processing, including programming language APIs to create and to read XML data, browser software, and database tools.

6.3.1 STRUCTURE OF XML DATA

The fundamental construct in an XML document is the **element**. An element is simply a pair of matching start- and end-tags and all the textthat appears between them.

XML documents must have a single **root** element that encompasses all other elements in the document. In the example in Figure, the<university>element forms the root element. Further, elements in an XML document must **nest** properly. For instance:

```
<course> ... <title> ... </title> ... </course>
```

is properly nested, whereas:

```
<course> ... <title> ... </course> ... </title>
```

is not properly nested. While proper nesting is an intuitive property, we may define it more formally. Text is said to appear **in the context of** an element if it appears between the start tag and end-tag of that element. Tags are properly nested if every start-tag has a unique matching end-tag that is in the context of the same parent element.

Note that text may be mixed with the sub elements of an element, as in Figure. As with several other features of XML, this freedom makes more sense in a document-processing context than in a data-processing context, and is not particularly useful for representing more-structured data such as database content in XML.

The ability to nest elements within other elements provides an alternative way to represent information. Figure shows a representation of part of the university information from Figure 6.1.10, but with course elements nested within department elements. The nested representation makes it easy to find all courses offered by a department. Similarly, identifiers of courses taught by an instructor are nested within the instructor elements. If an instructor teaches more than one course, there would be multiple course id elements within the corresponding instructor element.

```
...
    <course>
        This course is being offered for the first time in 2009.
        <course_id> BIO-399 </course_id>
        <title> Computational Biology </title>
        <dept_name> Biology </dept_name>
        <credits> 3 </credits>
    </course>
...
```

Figure 6.1.11 Mixture of text with sub elements.

```
<university-1>
    <department>
        <dept_name> Comp. Sci. </dept_name>
        <building> Taylor </building>
        <budget> 100000 </budget>
        <course>
            <course_id> CS-101 </course_id>
            <title> Intro. to Computer Science </title>
            <credits> 4 </credits>
        </course>
        <course>
            <course_id> CS-347 </course_id>
            <title> Database System Concepts </title>
            <credits> 3 </credits>
        </course>
    </department>
    <department>
        <dept_name> Biology </dept_name>
        <building> Watson </building>
        <budget> 90000 </budget>
        <course>
            <course_id> BIO-301 </course_id>
            <title> Genetics </title>
            <credits> 4 </credits>
        </course>
    </department>
    <instructor>
        <IID> 10101 </IID>
        <name> Srinivasan </name>
        <dept_name> Comp. Sci. </dept_name>
        <salary> 65000. </salary>
        <course_id> CS-101 </coursr_id>
    </instructor>
</university-1>
```

Figure 6.1.12 Nested XML representation of university information.

Details of instructors Brandt and Crick are omitted from Figure 4.1.12 for lack of space, but are similar in structure to that for Srinivasan. Although nested representations are natural in XML, they may lead to redundant storage of data. For example, suppose details of courses taught by an instructor are stored nested within the instructor element as shown inFigure. If a course is taught by more than one instructor, course information such as title, department, and credits would be stored redundantly with every instructor associated with the course.

```
<university-2>
    <instructor>
        <ID> 10101 </ID>
        <name> Srinivasan </name>
        <dept_name> Comp. Sci.</dept_name>
        <salary> 65000 </salary>
        <teaches>
            <course>
                <course_id> CS-101 </course_id>
                <title> Intro. to Computer Science </title>
                <dept_name> Comp. Sci. </dept_name>
                <credits> 4 </credits>
            </course>
        </teaches>
    </instructor>

    <instructor>
        <ID> 83821 </ID>
        <name> Brandt </name>
        <dept_name> Comp. Sci.</dept_name>
        <salary> 92000 </salary>
        <teaches>
            <course>
                <course_id> CS-101 </course_id>
                <title> Intro. to Computer Science </title>
                <dept_name> Comp. Sci. </dept_name>
                <credits> 4 </credits>
            </course>
        </teaches>
    </instructor>
</university-2>
```

Figure 6.1.13 Redundancy in nested XML representation.

Nested representations are widely used in XML data interchange applications to avoid joins. For instance, a purchase order would store the full address of sender and receiver redundantly on multiple purchase orders, whereas a normalized representation may require a join of purchase order records with a *company address* relation to get address information.

In addition to elements, XML specifies the notion of an **attribute**. For instance, the course identifier of a course can be represented as an attribute, as shown in Figure. The attributes of an element appear as *name=value* pairs before the closing ">" of a tag. Attributes are strings and do not contain markup. Furthermore, attributes can appear only oncein a given tag, unlike sub elements, which may be repeated.

```
...
    <course course_id= "CS-101">
        <title> Intro. to Computer Science</title>
        <dept_name> Comp. Sci. </dept_name>
        <credits> 4 </credits>
    </course>
...
```

Figure6.1.14 Use of attributes.

Note that in a document construction context, the distinction between subelement and attribute is important an attribute is implicitly text that does not appear in the printed or displayed document. However, in database and data exchange applications of XML, this distinction is less

relevant, and the choice of representing data as an attribute or a subelement is frequently arbitrary. In general, it is advisable to use attributes only to represent identifiers, and to store all other data as subelements.

One final syntactic note is that an element of the form<element></element> that contains no subelements or text can be abbreviated as <element/>; abbreviated elements may, however, contain attributes. Since XML documents are designed to be exchanged between applications, a **namespace** mechanism has been introduced to allow organizations to specify globally unique names to be used as element tags in documents. The idea of a namespace is to prepend each tag or attribute with a universal resource identifier (for example, a Web address). Thus, for example, if Yale University wanted to ensure that XML documents it created would not duplicate tags used by any business partner's XML documents, it could prepend a unique identifier with a colon to each tag name. The university may use a Web URL such as: http://www.yale.edu as a unique identifier. Using long unique identifiers in every tag would be rather inconvenient, so the namespace standard provides a way to define an abbreviation for identifiers.

In Figure, the root element (university) has an attribute xmlns:yale, which declares that yale is defined as an abbreviation for the URL given above. The abbreviation can then be used in various element tags, as illustrated in the figure. A document can have more than one namespace, declared as part of the root element. Different elements can then be associated with different namespaces. A *default namespace* can be defined by using the attribute xmlns instead of xmlns:yale in the root element. Elements without an explicit namespace prefix would then belong to the default namespace.

Sometimes we need to store values containing tags without having the tags interpreted as XML tags. So that we can do so, XML allows this construct:

```
<![CDATA[<course> ···</course>]]>

<university xmlns:yale="http://www.yale.edu">
    ...
    <yale:course>
        <yale:course_id> CS-101 </yale:course_id>
        <yale:title> Intro. to Computer Science</yale:title>
        <yale:dept_name> Comp. Sci. </yale:dept_name>
        <yale:credits> 4 </yale:credits>
    </yale:course>
    ...
</university>
```

Figure 6.1.15 Unique tag names can be assigned by using namespaces.

Because it is enclosed within CDATA, the text <course> is treated as normal text data, not as a tag. The term CDATA stands for character data.

6.3.2 XML DOCUMENT SCHEMA

Databases have schemas, which are used to constrain what information can be stored in the database and to constrain the data types of the stored information. In contrast, by default, XML documents can be created without any associated schema: an element may then have any subelement or attribute. While such freedom may occasionally be acceptable given the self-describing nature of the data format, it is not generally useful when XML documents must be processed automatically as part of an application, or even when large amounts of related data are to be formatted in XML.

Here, we describe the first schema-definition language included as part of the XML standard, the *Document Type Definition*, as well as its more recently defined replacement, *XML Schema*. Another XML schema-definition language called Relax NG is also in use, but we do not cover it here; for more information on Relax NG see the references in the bibliographical notes section.

6.3.3 QUERYING AND TRANSFORMATION

Given the increasing number of applications that use XML to exchange, mediate, and store data, tools for effective management ofXML data are becoming increasingly important. In particular, tools for querying and transformation of XML data are essential to extract information from large bodies of XML data, and to convert data between different representations (schemas) in XML. Just as the output of a relational query is a relation, the output of an XML query can be an XML document. As a result, querying and transformation can be combined intoa single tool. In this section, we describe the XPath and XQuery languages:

- XPath is a language for path expressions and is actually a building block for XQuery.
- XQuery is the standard language for querying XML data. It is modeled after SQL but is significantly different, since it has to deal with nested XML data.
- XQuery also incorporates XPath expressions.

The XSLT language is another language designed for transforming XML. However, it is used primarily in document-formatting applications, rather in data management applications.

6.4 INTRODUCTION TO MULTIMEDIA DATABASE SYSTEMS

Multimedia databases provide features that allow users to store and query different types of multimedia information, which includes *images* (such as photos or drawings), *video clips* (such as movies, newsreels, or home videos), *audio clips* (such as songs, phone messages, or speeches), and *documents* (such as books or articles). The main types of database queries that are needed involve locating multimedia sources that contain certain objects of interest. For example, one may want to locate allvideo clips in a video database that include a certain person, say Michael Jackson. One may also want to retrieve video clips based on certain activities included in them, such as video clips where a soccer goal is scored by a certain player or team. The above types of queries are referred to as **content-based retrieval**, because the multimedia source is being retrieved based on its containing certain objects or activities. Hence, a multimedia database must use some model to organize and index themultimedia sources based on their contents. *Identifying the contents* of multimedia sources is a difficult and time-consuming task. There are two main approaches. The first is based on **automatic analysis** of the multimedia sources to identify certain mathematical characteristics of theircontents. This approach uses different techniques depending on the type of multimedia source (image, video, audio, or text). The second approach depends on **manual identification** of the objects and activities of interest in each multimedia source and on using this information to index the sources. This approach can be applied to all multimedia sources, but it requires a manual preprocessing phase where a person has to scan each multimedia source to identify and catalog the objects and activities it contains so that they can be used to index the sources.

In the first part of this section, we will briefly discuss some of the characteristics of each type of multimedia source—images, video, audio, and text/documents. Then we will discuss approaches for automatic analysis of images followed by the problem of object recognition in images. We end this section with some remarks on analyzing audio sources. An **image** is typically stored either in raw form as a set of pixel orcell values, or in compressed form to save space. The image *shape descriptor* describes the geometric shape of the raw image, which is typically a rectangle of **cells** of a certain width and height. Hence, each image can be represented by an *m* by *n* grid of cells. Each cell contains a pixel value that describes the cell content. In black-and-white images, pixels can be one bit. In gray scale or color images, a pixel is multiple bits.Because images may require large amounts of space, they are often stored in compressed form. Compression standards, such as GIF, JPEG, or MPEG, use various mathematical transformations to reduce the number of cells stored but still maintain the main image characteristics. Applicable mathematical transforms include Discrete Fourier Transform (DFT), Discrete Cosine Transform (DCT), and wavelet transforms. To identify

objects of interest in an image, the image is typically divided into homogeneous segments using a *homogeneity predicate.* For example, in a color image, adjacent cells that have similar pixel values are grouped into a segment. The homogeneity predicate defines conditions for automatically grouping those cells. Segmentation and compression can hence identify the main characteristics of an image.

A typical image database query would be to find images in the database that are similar to a given image. The given image could be an isolated segment that contains, say, a pattern of interest, and the query is to locate other images that contain that same pattern. There are two main techniques for this type of search. The first approach uses a **distance function** to compare the given image with the stored images and their segments. If the distance value returned is small, the probability of a match is high. Indexes can be created to group stored images that are closein the distance metric so as to limit the search space. The second approach,called the **transformation approach**, measures image similarity by having a small number of transformations that can change one image's cells to match the other image. Transformations include rotations, translations, and scaling. Although the transformation approach is more general, it is also more time-consuming and difficult. A **video source** is typically represented as a sequence of frames, where each frame is a still image. However, rather than identifying the objects and activities in every individual frame, the video is divided into **video segments**, where each segment comprises a sequence of contiguous frames that includes thesame objects/activities. Each segment is identified by its starting and ending frames. The objects and activities identified in each video segment can be used to index the segments. An indexing technique called *frame segment trees* has been proposed for video indexing. The index includes both objects, such as persons, houses, and cars, as well as activities, suchas a person *delivering* a speech or two people *talking*. Videos are alsooften compressed using standards such as MPEG. **Audio sources** include stored recorded messages, such as speeches, class presentations, or even surveillance recordings of phone messages or conversations by law enforcement. Here, discrete transforms can be used to identify the main characteristics of a certain person's voice in order to have similarity-based indexing and retrieval.

A **text/document source** is basically the full text of some article, book, or magazine. These sources are typically indexed by identifying the keywords that appear in the text and their relative frequencies. However, filler words or common words called **stopwords** are eliminated from the process. Because there can be many keywords when attempting to index a collection of documents, techniques have been developed to reduce the number of keywords to those that are most relevant to the collection. A dimensionality reduction technique called *singular value decompositions* (SVD), which is based on matrix transformations, can be used for this purpose. An indexing technique called *telescoping vector trees* (TV-trees), can then be used to group similar documents.

6.4.1 LET US SUM UP

Thus, we have studied basics of deductive database, datalog notation,clause form and horn clauses, safe datalog program etc. Also the active database with languages for rule specification and events, conditions, actions. Here with this spatial databases-clustering methods, storage methods are explained in this chapter.

6.4.2 LIST OF REFERENCES

- Distributed Database; Principles & Systems By Publications, Stefano Ceri and Giuseppo Pelagatti,, McGraw-Hill International Editions (1984)

- Database Management Systems, 4rd edition, Raghu Ramakrishnan and Johannes Gehrke, McGraw-Hill (2002).

- Fundamentals of Database Systems, 6thEdition, Elmasri and Navathe, Addison. Wesley (2004).
- Unifying temporal data models via a conceptual model, C.S. Jensen, M.D. Soo, and R.T. Snodgrass: Information Systems, vol. 19, no. 7, pp. 514-547, 1994.

- Spatial Databases: A Tour by Shashi Shekhar and Sanjay Chawla, Prentice Hall, 2004 (ISBN 014-017480-7)

- Principles of Multimedia Database Systems, Subramanian V. S. Elsevier Publishers, 2014.

6.4.3 UNIT END EXERCISES

1) Explain Active Database with an example.

2) Explain difference between structured, sem-structured and un-structured data in XML database.

3) What are three main types of XML documents? What is the use of XML DTD?

4) Explain deductive database in short.

5) Explain datalog notation.

6) Write a short note on Multimedia database system.

❖❖❖❖