



PRATHYUSHA
ENGINEERING COLLEGE

(An ISO 9001:2000 Certified Institution)

Aranvoyaluppam

(Affiliated to Anna University, Chennai)

LECTURE NOTES
FOR
CS6302- DATABASE MANAGEMENT SYSTEMS
2013 Regulation

B.E COMPUTER SCIENCE AND ENGINEERING
III SEMESTER
ACADEMIC YEAR: 2016-2017

Prepared by

Ms.H.VIDHYA

PRATHYUSHA ENGINEERING COLLEGE

VISION

To emerge as a premier technical, engineering and management institution in the country by imparting quality education and thus facilitate our students to blossom in to dynamic professional so that they play a vital role for the progress of the nation and for a peaceful co-existence of our fellow human being.

MISSION

Prathyusha Engineering College will strive to emerge as a premier Institution in the country by

- To provide state of art infrastructure facilities
- Imparting quality education and training through qualified, experienced and committed members of the faculty
- Empowering the youth by providing professional leadership
- Developing centers of excellence in frontiers areas of Engineering, Technology and Management
- Networking with Industry, Corporate and Research Organizations
- Promoting Institute-Industry partnership for the peace and prosperity of the nation

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

VISION

Our Vision is to build a strong teaching & research environment in the field of computer science and engineering for developing a team of young dynamic computer science engineers, researchers, future entrepreneurs who are adaptive to respond to the challenges of 21st century. Our commitment lies in producing disciplined human individuals, capable of contributing solutions to solve problems faced by our society.

MISSION

- To provide a quality undergraduate and graduate education in both the theoretical and applied foundations of computer science and engineering.
- To train the students to effectively apply this education to solve real-world problems, thus amplifying their potential for lifelong high-quality careers and gives them a competitive advantage in the ever-changing and challenging global work environment of the 21st century.
- To initiate collaborative real-world industrial projects with industries and academic institutions to inculcate facilities in the arena of Research & Development
- To prepare them with an understanding of their professional and ethical responsibilities

PROGRAMME EDUCATIONAL OBJECTIVES

PEO-1:

To train the graduates to be excellent in computing profession by updating technical skill-sets and applying new ideas as the technology evolves.

PEO-2:

To enable the graduates to excel in professional career and /or higher education by acquiring knowledge in mathematical, computing and engineering principles.

PEO-3:

To enable the graduates to be competent to grasp, analyze, design, and create new products and solutions for the real time problems that are technically advanced, economically feasible and socially acceptable.

PEO- 4:

To enable the graduates to pursue a productive career as a member of multi-disciplinary and cross-functional teams, with an appreciation for the value of ethic and cultural diversity and an ability to relate engineering issues to broader social context.

PROGRAMME OUTCOMES

1. An ability to apply knowledge of computing, mathematics, science and engineering fundamentals appropriate to the discipline.
2. An ability to analyze a problem, and identify and formulate the computing requirements appropriate to its solution.
3. An ability to design, implement, and evaluate a computer-based system, process, component, or program to meet desired needs with appropriate consideration for public health and safety, cultural, societal and environmental considerations.
4. An ability to design and conduct experiments, as well as to analyze and interpret data.
5. An ability to use current techniques, skills, and modern tools necessary for computing practice.
6. An ability to analyze the local and global impact of computing on individuals, organizations, and society.
7. Knowledge of contemporary issues.
8. An understanding of professional, ethical, legal, security and social issues and responsibilities.
9. An ability to function effectively individually and on teams, including diverse and multidisciplinary, to accomplish a common goal.
10. An ability to communicate effectively with a range of audiences.
11. Recognition of the need for and an ability to engage in continuing professional development.
12. An understanding of engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects.
13. An ability to analyze the efficiency and the performance of the software with respect to meet the requirements and specifications of the expected outcome.
14. An ability to use simulation tools to get experimental results for the Real-Time system.

OBJECTIVES:

- To expose the students to the fundamentals of Database Management Systems.
- To make the students understand the relational model.
- To familiarize the students with ER diagrams.
- To expose the students to SQL.
- To make the students to understand the fundamentals of Transaction Processing and Query Processing.
- To familiarize the students with the different types of databases.
- To make the students understand the Security Issues in Databases.

UNIT I INTRODUCTION TO DBMS**10**

File Systems Organization - Sequential, Pointer, Indexed, Direct - Purpose of Database System-

Database System Terminologies-Database characteristics- Data models – Types of data models – Components of DBMS- Relational Algebra. LOGICAL DATABASE DESIGN: Relational DBMS - Codd's Rule - Entity-Relationship model - Extended ER Normalization – Functional Dependencies, Anomaly- 1NF to 5NF- Domain Key Normal Form – Denormalization

UNIT II SQL & QUERY OPTIMIZATION**8**

SQL Standards - Data types - Database Objects- DDL-DML-DCL-TCL-Embedded SQL- Static Vs Dynamic SQL - QUERY OPTIMIZATION: Query Processing and Optimization - Heuristics and Cost Estimates in Query Optimization.

UNIT III TRANSACTION PROCESSING AND CONCURRENCY CONTROL**8**

Introduction-Properties of Transaction- Serializability- Concurrency Control – Locking Mechanisms- Two Phase Commit Protocol-Dead lock.

UNIT IV TRENDS IN DATABASE TECHNOLOGY**10**

Overview of Physical Storage Media – Magnetic Disks – RAID – Tertiary storage – File Organization – Organization of Records in Files – Indexing and Hashing –Ordered Indices – B+ tree Index Files – B tree Index Files – Static Hashing – Dynamic Hashing - Introduction to Distributed Databases- Client server technology- Multidimensional and Parallel databases- Spatial and multimedia databases- Mobile and web databases- Data Warehouse-Mining- Data marts.

UNIT V ADVANCED TOPICS**9**

DATABASE SECURITY: Data Classification-Threats and risks – Database access Control – Types of Privileges –Cryptography- Statistical Databases.- Distributed Databases-Architecture-Transaction Processing-Data Warehousing and Mining- Classification-Association rules-Clustering-Information Retrieval- Relevance ranking- Crawling and Indexing the Web- Object Oriented Databases-XML Databases.

TOTAL: 45 PERIODS

OUTCOMES:

At the end of the course, the student should be able to:

- Design Databases for applications.
- Use the Relational model, ER diagrams.
- Apply concurrency control and recovery mechanisms for practical problems.
- Design the Query Processor and Transaction Processor.
- Apply security concepts to databases.

TEXT BOOK:

1. Ramez Elmasri and Shamkant B. Navathe, "Fundamentals of Database Systems", Fifth Edition, Pearson Education, 2008.

REFERENCES:

1. Abraham Silberschatz, Henry F. Korth and S. Sudharshan, "Database System Concepts", Sixth Edition, Tata Mc Graw Hill, 2011.
2. C.J.Date, A.Kannan and S.Swamynathan, "An Introduction to Database Systems", Eighth Edition, Pearson Education, 2006.
3. Atul Kahate, "Introduction to Database Management Systems", Pearson Education, New Delhi, 2006.
4. Alexis Leon and Mathews Leon, "Database Management Systems", Vikas Publishing House Private Limited, New Delhi, 2003.
5. Raghu Ramakrishnan, "Database Management Systems", Fourth Edition, Tata Mc Graw Hill, 2010.
6. G.K.Gupta, "Database Management Systems", Tata Mc Graw Hill, 2011.
7. Rob Cornell, "Database Systems Design and Implementation", Cengage Learning,

UNIT-I

UNIT I INTRODUCTION TO DBMS

10

File Systems Organization - Sequential, Pointer, Indexed, Direct - Purpose of Database System- Database System Terminologies-Database characteristics- Data models – Types of data models – Components of DBMS- Relational Algebra.

LOGICAL DATABASE DESIGN: Relational DBMS - Codd's Rule - Entity-Relationship model - Extended ER Normalization – Functional Dependencies, Anomaly- 1NF to 5NF- Domain Key Normal Form – Denormalization

File Organization

A file system stores multiple files. As mentioned previously, a file and its metadata are usually treated as nameless objects. But, users refer to files by their names. The FMS does the mapping from filenames to file objects. Also, where there are a large number of files, we need a way to organize files so that for a given filename, we can locate the file with relative ease. The file system uses specialized files called directories (or folders) to organize files. A directory is a special file that stores information about the subdirectories and files it holds; it stores the names of the files/subdirectories. The primary function of a directory is to help the system map filenames to file objects.

File Access Mechanisms

File access mechanism refers to the manner in which the records of a file may be accessed. There are several ways to access files

- Sequential access
- Direct/Random access
- Indexed sequential access
- Pointer

Sequential access

A sequential access is that in which the records are accessed in some sequence i.e the information in the file is processed in order, one record after the other. This access method is the most primitive one. Example: Compilers usually access files in this fashion.

Direct/Random access

- Random access file organization provides, accessing the records directly.
- Each record has its own address on the file with by the help of which it can be directly accessed for reading or writing.
- The records need not be in any sequence within the file and they need not be in adjacent locations on the storage medium.

Indexed sequential access

- This mechanism is built up on base of sequential access.
- An index is created for each file which contains pointers to various blocks.
- Index is searched sequentially and its pointer is used to access the file directly.

Purpose of Database system:

The typical file-processing system is supported by a conventional operating system. The system stores permanent records in various files, and it needs different application programs to extract records from, and add records to, the appropriate files.

A file processing system has a number of major **disadvantages**.

1. Data redundancy and inconsistency:

In file processing, every user group maintains its own files for handling its data-processing applications.

Example:

Consider the UNIVERSITY database. Here, two groups of users might be the course registration personnel and the accounting office. The accounting office also keeps data on registration and related billing information, whereas the registration office keeps track of student courses and grades. Storing the same data multiple times

is called data redundancy. This redundancy leads to several problems.

- Need to perform a single logical update multiple times.
- Storage space is wasted.
- Files that represent the same data may become inconsistent.

Data inconsistency is the various copies of the same data may no longer agree. Example: One user group may enter a student's birth date erroneously as JAN-19-1984, whereas the other user groups may enter the correct value of JAN-29-1984.

2. Difficulty in accessing data

File-processing environments do not allow needed data to be retrieved in a convenient and efficient manner.

Example: Suppose that one of the bank officers needs to find out the names of all customers who live within a particular area. The bank officer has now two choices: either obtains the list of all customers and extracts the needed information manually or ask a system programmer to write the necessary application program. Both alternatives are obviously unsatisfactory. Suppose that such a program is written, and that, several days later, the same officer needs to trim that list to include only those customers who have an account balance of \$10,000 or more. A program to generate such a list does not exist. Again, the officer has the preceding two options, neither of which is satisfactory.

3. Data isolation

Because data are scattered in various files, and files may be in different formats, writing new application programs to retrieve the appropriate data is difficult.

4. Integrity problems

The data values stored in the database must satisfy certain types of consistency constraints.

Example: The balance of certain types of bank accounts may never fall below a prescribed amount (\$25). Developers enforce these constraints in the system by adding appropriate code in the various application programs.

5. Atomicity problems

Atomic means the transaction must happen in its entirety or not at all. It is difficult to ensure atomicity in a conventional file-processing system. Example:

Consider a program to transfer \$50 from account A to account B. If a system failure occurs during the execution of the program, it is possible that the \$50 was removed from account A but was not credited to account B, resulting in an inconsistent database state.

6. Concurrent - access anomalies

For the sake of overall performance of the system and faster response, many systems allow multiple users to update the data simultaneously. In such an environment, interaction of concurrent updates is possible and may result in inconsistent data. To guard against this possibility, the system must maintain some form of supervision. But supervision is difficult to provide because data may be accessed by many different application programs that have not been coordinated previously. Example: When several reservation clerks try to assign a seat on an airline flight, the system should ensure that each seat can be accessed by only one clerk at a time for assignment to a passenger.

7. Security problems

Enforcing security constraints to the file processing system is difficult.

Database System Terminology

Database – A collection (or list) of information. A database is comprised of one or more lists (called tables) of data organized by columns, rows, and cells.

Tables – The view that displays the data base as a combinations of rows (records) and columns (fields). The cells contain the bits and pieces of data for each record in each field. The first row of a table is reserved for the field names.

Field names – Identify the different categories in a database. The top row is reserved for field names. Examples of field names are First name, last name, address, city, state, zip, phone number.

Field – Categories in a database. Fields are displayed in columns. For Example, in a database, the zip field contains all the zip codes from each of the records. These are the bits and pieces of data.

Records – Related information that is separated by columns or fields. A name and address are considered one record in the database. A second Name and address are a different record.

Cells - The intersection of columns and rows that contain the data for each record

Data - All of the records of information in a database including the field names.

Data + Field Names = Records All Records = a Database.

Objects – Enables you to find, view, display, and print data differently, based on your needs. The most commonly used objects are tables, queries, forms and reports.

- Tables show all records in a spreadsheet format
- Queries allow you to ask questions of the one or more tables and show only the information you ask for
- Forms display one record at a time
- Reports give and organize why of presenting information.

Characteristics of the data in the Database Management System

1. Sharing of the data takes place amongst the different type of the users and the applications.
2. Data exists permanently.
3. Data must be very much correct in the nature and should also be in accordance with the real world entity that they represent.
4. Data can live beyond the scope of the process that has created it.
5. Data is not at all repeated.
6. Changes that are made in the schema at one level should not at all affect the other levels.

Advantages of the Database Management System

1. Helps in reducing the complex nature of the systems environment due to the central control or the management of the data, access, utilization and the security.
2. Reduces the data redundancy and also the inconsistency as the same data elements are not at all repeated.
3. Also promotes the integrity of the data throughout the system or an organization.
4. Provides for the central control of the data creation and also for the definition.
5. Reduces or completely finishes the confusion that creeps in to the data.
6. Reduces the costs relating the program development and the maintenance.
7. Separates the logical view and the physical arrangement.
8. Reduces the program data dependence.
9. Permits the ad hoc queries.

10. Provides flexibility in the information systems.
11. Increases access and availability of the information.

Data models:

The data model is a collection of conceptual tools for describing data, data relationships, data semantics, and consistency constraints. A data model provides a way to describe the design of a data base at the physical, logical and view level.

The purpose of a data model is to represent data and to make the data understandable.

According to the types of concepts used to describe the database structure, there are three data models:

1. An external data model, to represent each user's view of the organization.
2. A conceptual data model, to represent the logical view that is DBMS independent.
3. An internal data model, to represent the conceptual schema in such a way that it can be understood by the DBMS.

Categories of data model:

1. Record-based data models
2. Object-based data models
3. Physical-data models.

The first two are used to describe data at the conceptual and external levels, the latter is used to describe data at the internal level.

1. Record - Based data models

In a record-based model, the database consists of a number of fixed format records possibly of differing types. Each record type defines a fixed number of fields, each typically of a fixed length. There are three types of record-based logical data model.

- Hierarchical data model.
- Network data model
- Relational data model

Hierarchical data model

In the hierarchical model, data is represented as collections of records and relationships are represented by sets. The hierarchical model allows a node to have

only one parent. A hierarchical model can be represented as a tree graph, with records appearing as nodes, also called segments, and sets as edges.

Network data model

In the network model, data is represented as collections of records and relationships are represented by sets. Each set is composed of at least two record types:

- An owner record that is equivalent to the hierarchical model's parent
- A member record that is equivalent to the hierarchical model's child A set represents a 1: M relationship between the owner and the member.

Relational data model:

The relational data model is based on the concept of mathematical relations. Relational model stores data in the form of a table. Each table corresponds to an entity, and each row represents an instance of that entity. Tables, also called relations are related to each other through the sharing of a common entity characteristic.

Example Relational DBMS, DB2, oracle, MS SQL-server.

2. Object - Based Data Models

Object-based data models use concepts such as entities, attributes, and relationships.

An entity is a distinct object in the organization that is to be represents in the database. An attribute is a property that describes some aspect of the object, and a relationship is an association between entities. Common types of object-based data model are:

- Entity - Relationship model
- Object - oriented model
- Semantic model

Entity - Relationship Model:

The ER model is based on the following components:

- **Entity:** An entity was defined as anything about which data are to be collected and stored. Each row in the relational table is known as an entity instance or entity occurrence in the ER model. Each entity is described by a set of attributes that describes particular characteristics of the entity.

Object oriented model:

In the object-oriented data model (OODM) both data and their relationships are contained in a single structure known as an object. An object is described by its factual content. An object includes information about relationships between the facts within the object, as well as information about its relationships with other objects. Therefore, the facts within the object are given greater meaning. The OODM is said to be a semantic data model because semantic indicates meaning. The OO data model is based on the following components: An object is an abstraction of a real-world entity. Attributes describe the properties of an object.

Components of a DBMS

The DBMS accepts the SQL commands generated from a variety of user interfaces, produces query evaluation plans, executes these plans against the database, and returns the answers. As shown, the major software modules or components of DBMS are as follows:

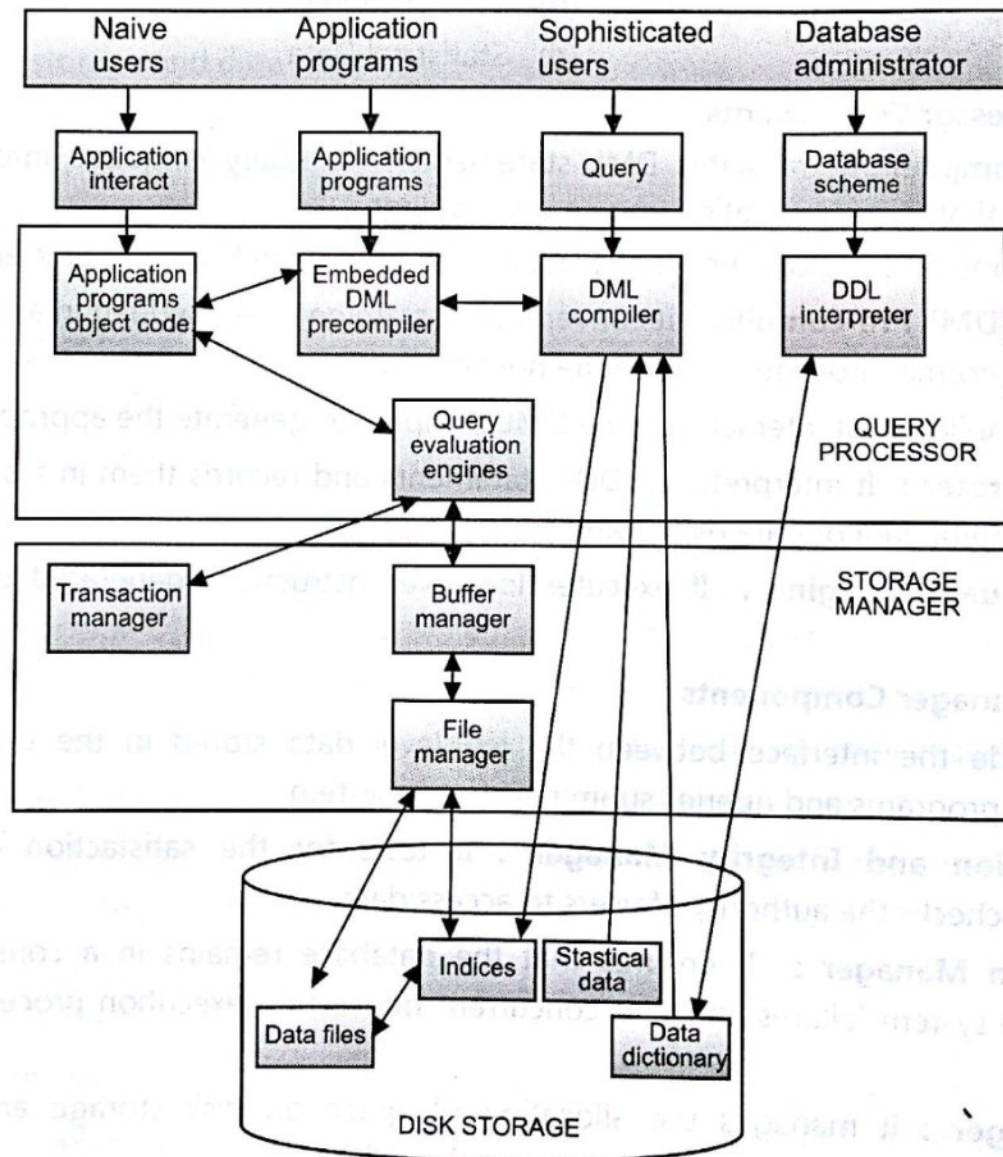
(i) Query processor: The query processor transforms user queries into a series of low level instructions. It is used to interpret the online user's query and convert it into an efficient series of operations in a form capable of being sent to the run time data manager for execution. The query processor uses the data dictionary to find the structure of the relevant portion of the database and uses this information in modifying the query and preparing an optimal plan to access the database.

(ii) Run time database manager: Run time database manager is the central software component of the DBMS, which interfaces with user-submitted application programs and queries. It handles database access at run time. It converts operations in user's queries coming. Directly via the query processor or indirectly via an application program from the user's logical view to a physical file system. It accepts queries and examines the external and conceptual schemas to determine what conceptual records are required to satisfy the user's request. It enforces constraints to maintain the consistency and integrity of the data, as well as its security. It also

PRATHYUSHA ENGINEERING COLLEGE

performs backing and recovery operations. Run time database manager is sometimes referred to as the database control system and has the following components:

- **Authorization control:** The authorization control module checks the authorization of users in terms of various privileges to users.
- **Command processor:** The command processor processes the queries passed by authorization control module.



System structure

Integrity checker: It checks the integrity constraints so that only valid data can be entered into the database.

Query optimizer: The query optimizers determine an optimal strategy for the query execution.

- **Transaction manager:** The transaction manager ensures that the transaction properties should be maintained by the system.

- **Scheduler:** It provides an environment in which multiple users can work on same piece of data at the same time in other words it supports concurrency.

(iii) Data Manager: The data manager is responsible for the actual handling of data in the database. It provides recovery to the system which that system should be able to recover the data after some failure. It includes Recovery manager and Buffer manager. The buffer manager is responsible for the transfer of data between the main memory and secondary storage (such as disk or tape). It is also referred as the cache manger.

Users

The users are the people who manage the databases and perform different operations on the databases in the database system. There are three kinds of people who play different roles in database system

1. Application Programmers
2. Database Administrators
3. Naïve User
4. Sophisticated User

Application Programmers

The people who write application programs in programming languages (such as Visual Basic, Java, or C++) to interact with databases are called Application Programmer.

Database Administrators

A person who is responsible for managing the overall database management system is called database administrator or simply DBA.

End-Users

The end-users are the people who interact with database management system to perform different operations on database such as retrieving, updating, inserting, deleting data etc.

Execution Process of a DBMS

As show, conceptually, following logical steps are followed while executing users to request to access the database system:

- (i) Users issue a query using particular database language, for example, SQL commands.
- (ii) The passes query is presented to a query optimizer, which uses information about how the data
an efficient execution plan for the evaluating the query.
- (iii) The DBMS accepts the users SQL commands and analyses them.
- (iv) The DBMS produces query evaluation plans, that is, the external schema for the user, the corresponding external/conceptual mapping, the conceptual schema, the conceptual/internal mapping, and the storage structure definition. Thus, an evaluation\ plan is a blueprint for evaluating a query.
- (v) The DBMS executes these plans against the physical database and returns the answers to the user.

Using components such as transaction manager, buffer manager, and recovery manager, the DBMS supports concurrency and recovery.

Relational algebra

Relational algebra is a procedural query language, which takes instances of relations as input and yields instances of relations as output. It uses operators to perform queries. An operator can be either unary or binary. They accept relations as their input and yields relations as their output. Relational algebra is performed recursively on a relation and intermediate results are also considered relations.

Fundamental operations of Relational algebra:

- Select
- Project
- Union
- Set different
- Cartesian product
- Rename

These are defined briefly as follows:

Select Operation (σ)

Selects tuples that satisfy the given predicate from a relation.

Notation $\sigma_p(r)$

Where p stands for selection predicate and r stands for relation. p is propositional logic formulae which may use connectors like and, or and not. These terms may use relational operators like: $=, \neq, \geq, <, >, \leq$.

For example:

$\sigma_{\text{subject}=\text{"database"}}(\text{Books})$

Output : Selects tuples from books where subject is 'database'.

$\sigma_{\text{subject}=\text{"database"} \text{ and } \text{price}=\text{"450"}}(\text{Books})$

Output : Selects tuples from books where subject is 'database' and 'price' is 450.

$\sigma_{\text{subject}=\text{"database"} \text{ and } \text{price} < \text{"450"} \text{ or } \text{year} > \text{"2010"}}(\text{Books})$

Output : Selects tuples from books where subject is 'database' and 'price' is 450 or the publication year is greater than 2010, that is published after 2010.

Project Operation (Π)

Projects column(s) that satisfy given predicate.

Notation: $\Pi_{A_1, A_2, A_n}(r)$

Where a_1, a_2, a_n are attribute names of relation r .

Duplicate rows are automatically eliminated, as relation is a set.

for example:

$\Pi_{\text{subject, author}}(\text{Books})$

Selects and projects columns named as subject and author from relation Books.

Union Operation (\cup)

Union operation performs binary union between two given relations and is defined as:

$$r \cup s = \{ t \mid t \in r \text{ or } t \in s \}$$

Notation: $r \cup s$

Where r and s are either database relations or relation result set (temporary relation).

For a union operation to be valid, the following conditions must hold:

- r, s must have same number of attributes.
- Attribute domains must be compatible.

Duplicate tuples are automatically eliminated.

$$\Pi_{\text{author}}(\text{Books}) \cup \Pi_{\text{author}}(\text{Articles})$$

Output : Projects the name of author who has either written a book or an article or both.

Set Difference ($-$)

The result of set difference operation is tuples which present in one relation but are not in the second relation.

Notation: $r - s$

Finds all tuples that are present in r but not s .

$$\Pi_{\text{author}}(\text{Books}) - \Pi_{\text{author}}(\text{Articles})$$

Output: Results the name of authors who has written books but not articles.

Cartesian Product (\times)

Combines information of two different relations into one.

Notation: $r \times s$

Where r and s are relations and there output will be defined as:

$$r \times s = \{ q \mid q \in r \text{ and } t \in s \}$$

Π author = 'tutorialspoint'(Books X Articles)

Output : yields a relation as result which shows all books and articles written by tutorialspoint.

Rename operation (ρ)

Results of relational algebra are also relations but without any name. The rename operation allows us to rename the output relation. rename operation is denoted with small greek letter rho ρ

Notation: $\rho_x(E)$

Where the result of expression E is saved with name of x.

Additional operations are:

- Set intersection
- Assignment
- Natural join

Relational Calculus

In contrast with Relational Algebra, Relational Calculus is non-procedural query language, that is, it tells what to do but never explains the way, how to do it.

Relational calculus exists in two forms:

Tuple relational calculus (TRC)

Filtering variable ranges over tuples

Notation: $\{ T \mid \text{Condition} \}$

Returns all tuples T that satisfies condition.

For Example:

$\{ T.\text{name} \mid \text{Author}(T) \text{ AND } T.\text{article} = \text{'database'} \}$

Output: returns tuples with 'name' from Author who has written article on 'database'.

TRC can be quantified also. We can use Existential (\exists) and Universal Quantifiers (\forall)

For example:

$\{ R \mid \exists T \in \text{Authors}(T.\text{article} = \text{'database'} \text{ AND } R.\text{name} = T.\text{name}) \}$

Output : the query will yield the same result as the previous one.

Domain relational calculus (DRC)

In DRC the filtering variable uses domain of attributes instead of entire tuple values (as done in TRC, mentioned above).

Notation:

$\{ a_1, a_2, a_3, \dots, a_n \mid P(a_1, a_2, a_3, \dots, a_n) \}$

where a_1, a_2 are attributes and P stands for formulae built by inner attributes.

For example:

$\{ \langle \text{article}, \text{page}, \text{subject} \rangle \mid \in \text{TutorialsPoint} \wedge \text{subject} = \text{'database'} \}$

Output: Yields Article, Page and Subject from relation TutorialsPoint where Subject is database.

Just like TRC, DRC also can be written using existential and universal quantifiers. DRC also involves relational operators.

Expression power of Tuple relation calculus and Domain relation calculus is equivalent to Relational Algebra.

LOGICAL DATABASE DESIGN

Relational DBMS

Concepts

- A database that is perceived by the user as a collection of two-dimensional tables
- SQL is used to manipulate relational databases

The Relational Database Concept

- Proposed by Dr. Codd in 1970
- The basis for the relational database management system (RDBMS)
- The relational model contains the following components:
- Collection of objects or relations

- Set of operations to act on the relations
- Data integrity for accuracy and consistency

Advantages

- ❖ Rigorous design methodology (normalization, set theory)
- ❖ All other database structures can be reduced to a set of relational tables
 - Mainframe databases use Network and Hierarchical methods to store and retrieve data.
 - Access to the data is hard-coded
 - It is very difficult to extract data from this type of database without some pre-defined access path.
 - Extremely fast retrieval times for multi-user, transactional environment.
- ❖ Ease the use compared to other database systems
- ❖ Modifiable - new tables and rows can be added easily
- ❖ The relational join mechanism
 - Based on algebraic set theory - a set is a group of common elements where each member has some unique aspect or attribute
 - very flexible and powerful
- ❖ Fast Processing
 - Faster processors, multi-threaded operating and parallel servers
 - Indexes, fast networks and clustered disk arrays
 - 57,000 simultaneous users (Oracle/IBM)

Disadvantages

- Expensive solutions that require thorough planning
- Easy to create badly designed and inefficient database designs if there is not any proper data analysis prior to implementation

Codd's Rule

Dr. Edgar Frank Codd (August 19, 1923 – April 18, 2003) was an computer scientist, while working for IBM he invented the relational model for database management (theoretical basis for relational databases). Codd proposed thirteen rules (numbered zero to twelve) and said that if a Database Management System meets these rules, it can be called as a Relational Database Management System. These rules are called as Codd's 12 rules. Hardly any commercial product follows all.

Rule Zero

- The system must qualify as relational, as a database, and as a management system. For a system to qualify as a relational database management system (RDBMS), that system must use its relational facilities (exclusively) to manage the database.
- The other 12 rules derive from this rule. The rules are as follows :

Rule 1 : The information rule: All information in the database is to be represented in one and only one way, namely by values in column positions within rows of tables.

Rule 2 : The guaranteed access rule: All data must be accessible. This rule is essentially a restatement of the fundamental requirement for primary keys. It says that every individual scalar value in the database must be logically addressable by specifying the name of the containing table, the name of the containing column and the primary key value of the containing row.

Rule 3 : Systematic treatment of null values: The DBMS must allow each field to remain null (or empty). Specifically, it must support a representation of "missing information and inapplicable information" that is systematic, distinct from all regular values (for example, "distinct from zero or any other number", in the case of numeric values), and independent of data type. It is also implied that such representations must be manipulated by the DBMS in a systematic way.

Rule 4 : Active online catalog based on the relational model: The system must support an online, inline, relational catalog that is accessible to authorized users by means of their regular query language. That is, users must be able to access the database's structure (catalog) using the same query language that they use to access the database's data.

Rule 5 : The comprehensive data sub language rule: The system must support at least one relational language that

1. Has a linear syntax
2. Can be used both interactively and within application programs,
3. Supports data definition operations (including view definitions), data manipulation operations (update as well as retrieval), security and integrity constraints, and transaction management operations (begin, commit, and rollback).

Rule 6 : The view updating rule: All views those can be updated theoretically, must be updated by the system.

Rule 7 : High-level insert, update, and delete: The system must support set-at-a-time insert, update, and delete operators. This means that data can be retrieved from a relational database in sets constructed of data from multiple rows and/or multiple tables. This rule states that insert, update, and delete operations should be supported for any retrievable set rather than just for a single row in a single table.

Rule 8 : Physical data independence: Changes to the physical level (how the data is stored, whether in arrays or linked lists etc.) must not require a change to an application based on the structure.

Rule 9 : Logical data independence: Changes to the logical level (tables, columns, rows, and so on) must not require a change to an application based on the structure. Logical data independence is more difficult to achieve than physical data independence.

Rule 10 : Integrity independence: Integrity constraints must be specified separately from application programs and stored in the catalog. It must be possible to change such constraints as and when appropriate without unnecessarily affecting existing applications.

Rule 11 : Distribution independence: The distribution of portions of the database to various locations should be invisible to users of the database. Existing applications should continue to operate successfully :

1. when a distributed version of the DBMS is first introduced; and
2. when existing distributed data are redistributed around the system.

Rule 12: The non subversion rule: If the system provides a low-level (record-at-a-time) interface, then that interface cannot be used to subvert the system, for example, bypassing a relational security or integrity constraint.

Entity-Relationship Model

Introduction to ER Modelling

- An Entity-relationship model (ERM) is an abstract and conceptual representation of data.
- ER modelling is a DBmodelling method, used to produce a type of conceptual schema of a system.
- Diagrams created by this process are called ER diagrams
- Sequence: Conceptual data model(i.e. ER) is, at a later stage (called logical design), mapped to a logical data model , (e.g.relational model); this is mapped to a physical model in physical design.
- ER Model used to interpret, specify & document requirements for DBs irrespective of DBMS being used.

ER Definitions

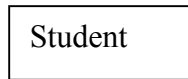
- Entity (Instance):

–An instance of a physical object in the real world.

–Entity Class: Group of objects of the same type.

–E.g. Entity Class “Student”, Entities “John”, “Trish” etc

- Attributes:



–Properties of Entities that describe their characteristics

- Types:

- *Simple*: Attribute that is not divisible, e.g. age.



- *Composite*: Attribute composed of several simple attributes, e.g. address (house number, street, district)



- *Multiple* : Attribute with a set of possible values for the same entity, e.g. Phone (home, mobile etc.) or email



- *Key*: Uniquely Ids the Entity e.g. PPSN, Chassis No.



–*Value Set* (or *domain*): Each simple attribute associated with a VS that may be assigned to that attribute for each individual entity, e.g. age = integer, range *18,...65+

Key/Key Attributes

Super Key - Set of attributes uniquely identifying a row
For SP{S#,P#,QTY} or {S#,P#}

- Candidate Key – (Irreducible) combination of attributes which is a unique id within a table.
For SP{S#,P#}

- Primary Key – One of the candidate keys. For SP{S#,P#}

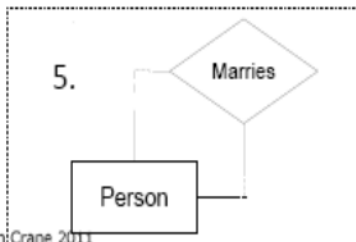
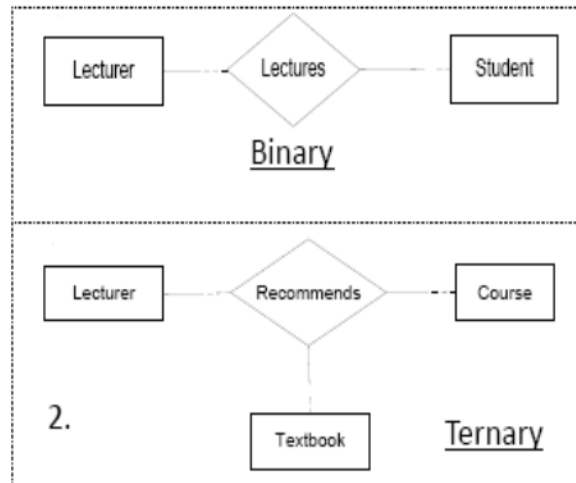
- Alternate Key – The candidate key not chosen as the primary key.

- Foreign Key – A combination of attributes in one relation whose values to equal in the primary key of another relation.

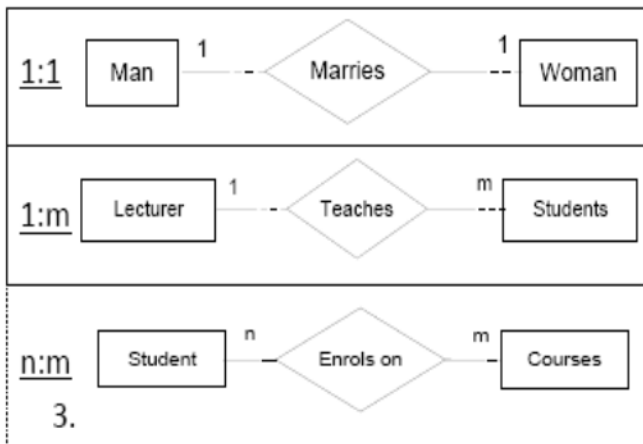
ER Definitions (cont'd)

- Relationships:**

1. Are bi-directional (ie can be put 2 ways)
2. **Degree:**
 - *binary* (i.e. involve only two entities),
 - *ternary* (i.e. involve three participating entities).
3. **Cardinality:** Entity types may be linked in more than one way.
4. May have properties (attribs).
5. Can be Recursive.



(c) Martin Crane 2011



© 2011

NAME OF THE FACULTY : Ms.H.VIDHYA

Entity

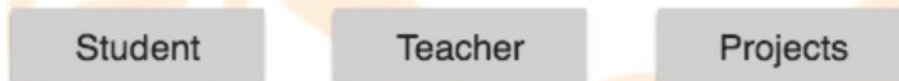
Entities are represented by means of rectangles. Rectangles are named with the entity set they represent.



[Image: Entities in a school database]

Entity

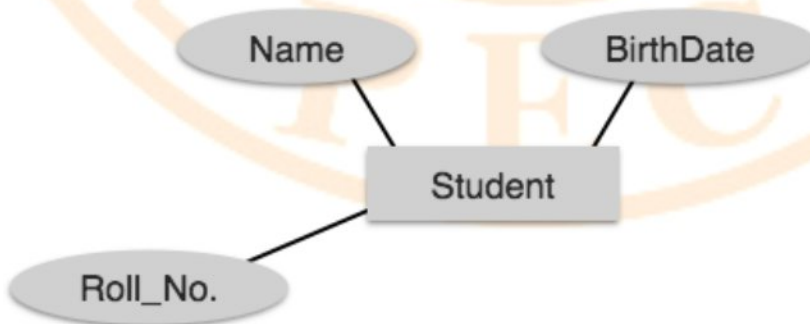
Entities are represented by means of rectangles. Rectangles are named with the entity set they represent.



[Image: Entities in a school database]

Attributes

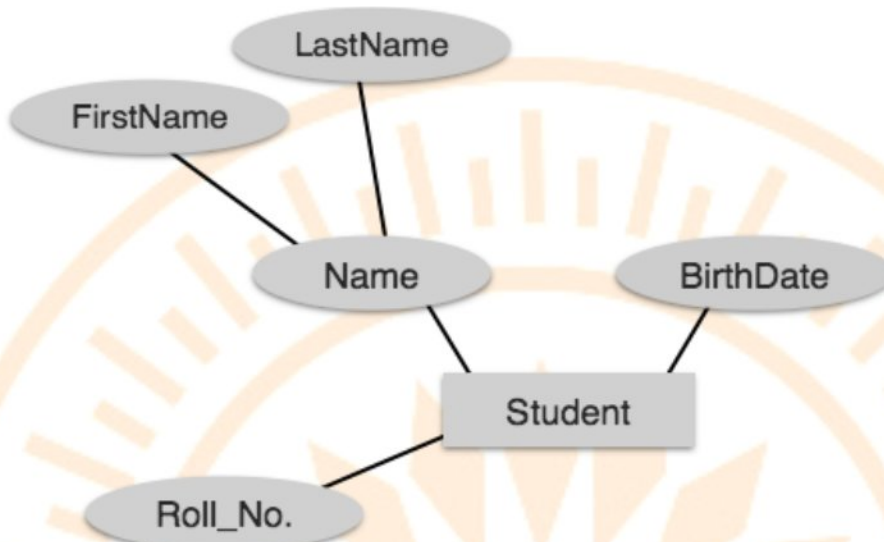
Attributes are properties of entities. Attributes are represented by means of eclipses. Every eclipse represents one attribute and is directly connected to its entity (rectangle).



[Image: Simple Attributes]

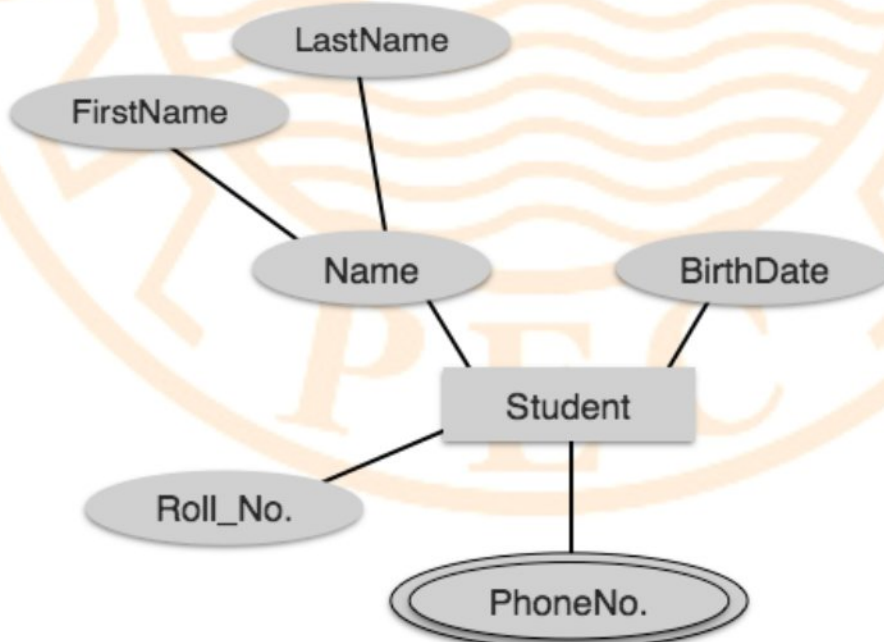
PRATHYUSHA ENGINEERING COLLEGE

If the attributes are **composite**, they are further divided in a tree like structure. Every node is then connected to its attribute. That is composite attributes are represented by eclipses that are connected with an eclipse.



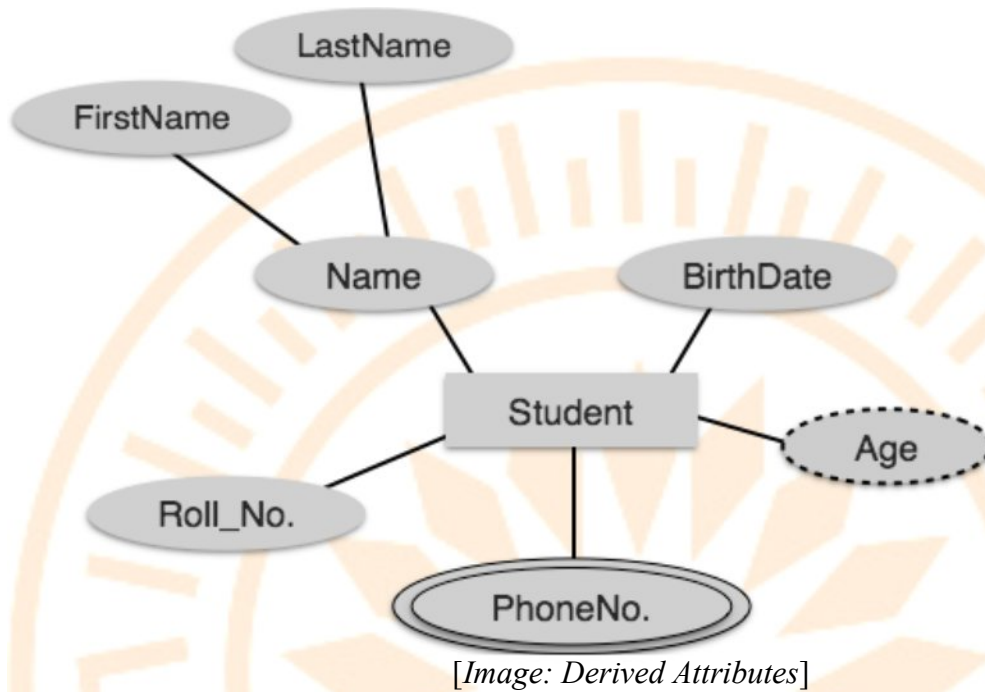
[Image: Composite Attributes]

Multivalued attributes are depicted by double eclipse.



[Image: Multivalued Attributes]

Derived attributes are depicted by dashed ellipse.



Relationship

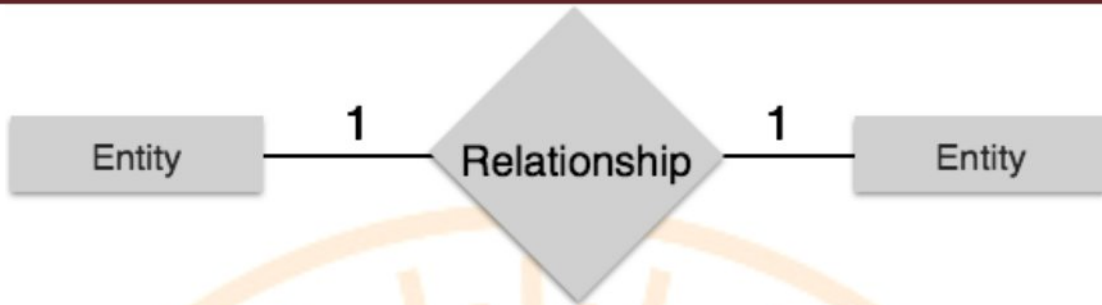
Relationships are represented by diamond shaped box. Name of the relationship is written in the diamond-box. All entities (rectangles), participating in relationship, are connected to it by a line.

Binary relationship and cardinality

A relationship where two entities are participating, is called a **binary relationship**. Cardinality is the number of instance of an entity from a relation that can be associated with the relation.

- **One-to-one**

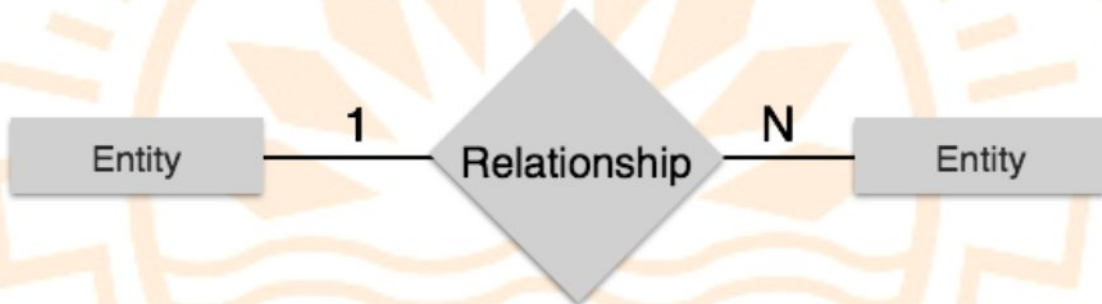
When only one instance of entity is associated with the relationship, it is marked as '1'. This image below reflects that only 1 instance of each entity should be associated with the relationship. It depicts one-to-one relationship



[Image: One-to-one]

- **One-to-many**

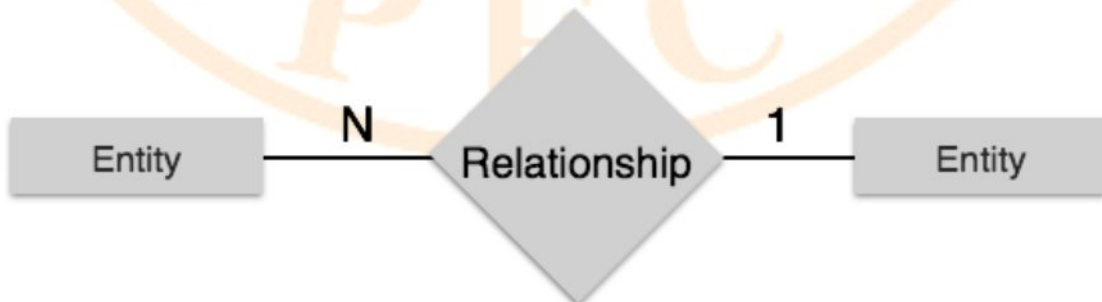
When more than one instance of entity is associated with the relationship, it is marked as 'N'. This image below reflects that only 1 instance of entity on the left and more than one instance of entity on the right can be associated with the relationship. It depicts one-to-many relationship



[Image: One-to-many]

- **Many-to-one**

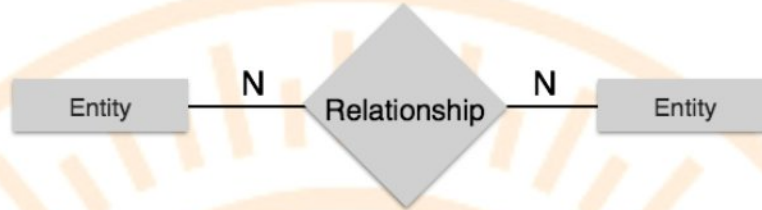
When more than one instance of entity is associated with the relationship, it is marked as 'N'. This image below reflects that more than one instance of entity on the left and only one instance of entity on the right can be associated with the relationship. It depicts many-to-one relationship



[Image: Many-to-one]

- **Many-to-many**

This image below reflects that more than one instance of entity on the left and more than one instance of entity on the right can be associated with the relationship. It depicts many-to-many relationship



[Image: Many-to-many]

Participation Constraints

- **Total Participation:** Each entity in the entity is involved in the relationship. Total participation is represented by double lines.
- **Partial participation:** Not all entities are involved in the relationship. Partial participation is represented by single line.

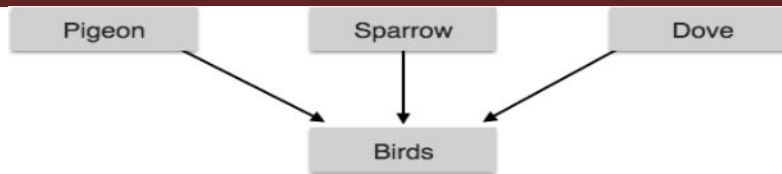
Extended ER Normalization

ER Model has the power of expressing database entities in conceptual hierarchical manner such that, as the hierarchical goes up it generalize the view of entities and as we go deep in the hierarchy it gives us detail of every entity included.

Going up in this structure is called generalization, where entities are clubbed together to represent a more generalized view. For example, a particular student named, Mira can be generalized along with all the students, the entity shall be student, and further a student is person. The reverse is called specialization where a person is student, and that student is Mira.

Generalization

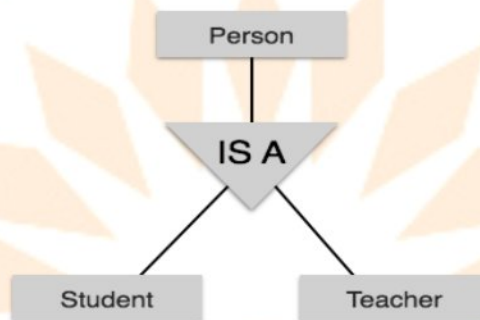
As mentioned above, the process of generalizing entities, where the generalized entities contain the properties of all the generalized entities is called Generalization. In generalization, a number of entities are brought together into one generalized entity based on their similar characteristics. For an example, pigeon, house sparrow, crow and dove all can be generalized as Birds.



[Image: Generalization]

Specialization

Specialization is a process, which is opposite to generalization, as mentioned above. In specialization, a group of entities is divided into sub-groups based on their characteristics. Take a group Person for example. A person has name, date of birth, gender etc. These properties are common in all persons, human beings. But in a company, a person can be identified as employee, employer, customer or vendor based on what role do they play in company.

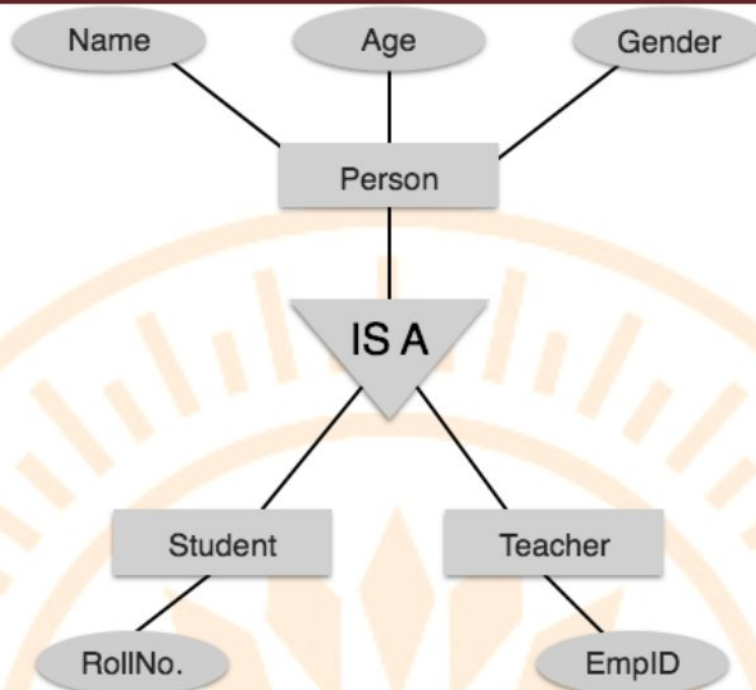


[Image: Specialization]

Similarly, in a school database, a person can be specialized as teacher, student or staff; based on what role do they play in school as entities.

Inheritance

We use all above features of ER-Model, in order to create classes of objects in object oriented programming. This makes it easier for the programmer to concentrate on what she is programming. Details of entities are generally hidden from the user, this process known as abstraction. One of the important features of Generalization and Specialization, is inheritance, that is, the attributes of higher-level entities are inherited by the lower level entities.



[Image: Inheritance]

For example, attributes of a person like name, age, and gender can be inherited by lower level entities like student and teacher etc.

Functional Dependencies

FD's are constraints on well-formed relations and represent formalism on the infrastructure of relation.

Definition: A *functional dependency* (FD) on a relation schema **R** is a constraint $X \rightarrow Y$, where X and Y are subsets of attributes of **R**.

Definition: an FD is a relationship between an attribute "Y" and a determinant (1 or more other attributes) "X" such that for a given value of a determinant the value of the attribute is uniquely defined.

- X is a determinant
- X determines Y
- Y is functionally dependent on X
- $X \rightarrow Y$
- $X \rightarrow Y$ is trivial if $Y \subseteq X$

Definition: An FD $X \rightarrow Y$ is *satisfied* in an instance **r** of **R** if for every pair of tuples, t and s : if t and s agree on all attributes in X then they must agree on all attributes in Y

PRATHYUSHA ENGINEERING COLLEGE

A key constraint is a special kind of functional dependency: all attributes of relation occur on the right-hand side of the FD:

- $SSN \rightarrow SSN, Name, Address$

Example Functional Dependencies

Let R be

NewStudent(*stuId, lastName, major, credits, status, socSecNo*)

FDs in R include

- $\{stuId\} \rightarrow \{lastName\}$, but not the reverse
- $\{stuId\} \rightarrow \{lastName, major, credits, status, socSecNo, stuId\}$
- $\{socSecNo\} \rightarrow \{stuId, lastName, major, credits, status, socSecNo\}$
- $\{credits\} \rightarrow \{status\}$, but not $\{status\} \rightarrow \{credits\}$

ZipCode \rightarrow *AddressCity*

- 16652 is Huntingdon's ZIP

ArtistName \rightarrow *BirthYear*

- Picasso was born in 1881

Autobrand \rightarrow *Manufacturer, Engine type*

- Pontiac is built by General Motors with gasoline engine

Author, Title \rightarrow *PublDate*

- Shakespeare's Hamlet was published in 1600

Trivial Functional Dependency

The FD $X \rightarrow Y$ is *trivial* if set $\{Y\}$ is a subset of set $\{X\}$

Examples: If A and B are attributes of R,

- $\{A\} \rightarrow \{A\}$
- $\{A,B\} \rightarrow \{A\}$
- $\{A,B\} \rightarrow \{B\}$
- $\{A,B\} \rightarrow \{A,B\}$

PRATHYUSHA ENGINEERING COLLEGE

are all trivial FDs and will not contribute to the evaluation of normalization.

Functional Dependency can be classified as follows:

- **Full Functional dependency** Indicates that if **A** and **B** are attributes(columns) of a table, **B** is fully functionally dependent on **A** if **B** is functionally dependent on **A**, but not on any proper subset of **A**.
E.g. StaffID---->BranchID
- **Partial Functional Dependency** Indicates that if **A** and **B** are attributes of a table, **B** is partially dependent on **A** if there is some attribute that can be removed from **A** and yet the dependency still holds.
Say for Ex, consider the following functional dependency that exists in the **Tbl_Staff** table:
StaffID,Name -----> BranchID
BranchID is functionally dependent on a subset of **A** (StaffID,Name), namely StaffID.
- **Transitive Functional Dependency:** A condition where **A**, **B** and **C** are attributes of a table such that if **A** is functionally dependent on **B** and **B** is functionally dependent on **C** then **C** is **Transitively dependent** on **A** via **B**.
Say for Ex, consider the following functional dependencies that exist in the **Tbl_Staff_Branch** table:
StaffID---->Name,Sex,Position,Sal,BranchID,Br_Address
BranchID----->Br_Address
So, StaffID attribute functionally determines Br_Address via BranchID attribute.

FD Axioms

Understanding: Functional Dependencies are recognized by analysis of the real world; no automation or algorithm. Finding or recognizing them are the database designer's task.

FD manipulations:

- **Soundness** -- no incorrect FD's are generated
- **Completeness** -- all FD's can be generated

Axiom Name	Axiom	Example
Reflexivity	if a is set of attributes, $b \subseteq a$, then $a \rightarrow b$	$SSN, Name \rightarrow SSN$
Augmentation	if $a \rightarrow b$ holds and c is a set of attributes, then	$SSN \rightarrow Name$ then $SSN, Phone \rightarrow Name, Phone$

PRATHYUSHA ENGINEERING COLLEGE

	$ca \rightarrow cb$	
Transitivity	if $a \rightarrow b$ holds and $b \rightarrow c$ holds, then $a \rightarrow c$ holds	$SSN \rightarrow Zip$ and $Zip \rightarrow City$ then $SSN \rightarrow City$
Union or Additivity *	if $a \rightarrow b$ and $a \rightarrow c$ holds then $a \rightarrow bc$ holds	$SSN \rightarrow Name$ and $SSN \rightarrow Zip$ then $SSN \rightarrow Name, Zip$
Decomposition or Projectivity*	if $a \rightarrow bc$ holds then $a \rightarrow b$ and $a \rightarrow c$ holds	$SSN \rightarrow Name, Zip$ then $SSN \rightarrow Name$ and $SSN \rightarrow Zip$
Pseudotransitivity*	if $a \rightarrow b$ and $cb \rightarrow d$ hold then $ac \rightarrow d$ holds	$Address \rightarrow Project$ and $Project, Date \rightarrow Amount$ then $Address, Date \rightarrow Amount$
(NOTE)	$ab \rightarrow c$ does NOT imply $a \rightarrow b$ and $b \rightarrow c$	

*Armstrong's Axioms (basic axioms)

Anomalies

An anomaly is an inconsistent, incomplete, or contradictory state of the database

- **Insertion anomaly** – user is unable to insert a new record of data when it should be possible to do so because not all other information is available.
- **Deletion anomaly** – when a record is deleted, other information that is tied to it is also deleted
- **Update anomaly** – a record is updated, but other appearances of the same items are not updated

Redundancy leads to the following anomalies:

Update anomaly: A change in *Address* must be made in several places. Updating one fact may require updating multiple tuples.

Deletion anomaly: Deleting one fact may delete other information. Suppose a person gives up all hobbies. Do we:

Set Hobby attribute to null? No, since *Hobby* is part of key

Delete the entire row? No, since we lose other information in the row

Insertion anomaly: To record one fact may require more information than is available. *Hobby* value must be supplied for any inserted row since *Hobby* is part of key

Objectives of Normalization

Develop a good description of the data, its relationships and constraints

Produce a stable set of relations that

- Is a faithful model of the enterprise
- Is highly flexible
- Reduces redundancy-saves space and reduces inconsistency in data
- Is free of update, insertion and deletion anomalies

Normal Forms

- First normal form -1NF
- Second normal form-2NF
- Third normal form-3NF
- Boyce-Codd normal form-BCNF
- Fourth normal form-4NF
- Fifth normal form-5NF
- Domain/Key normal form-DKNF

Each is contained within the previous form – each has stricter rules than the previous form

Limitations of E-R Designs

E-R modeling provides a set of guidelines, but does not result in a unique database schema.

Nor does it provide a way of evaluating alternative schemas.

Normalization theory provides a mechanism for analyzing and refining the schema produced by an E-R design, or any other design.

Redundancy

Dependencies between attributes within a relation cause redundancy

Ex. All addresses in the same town have the same zip code

SSN	Name	Town	Zip
1234	Joe	Huntingdon	16652
2345	Mary	Huntingdon	16652

PRATHYUSHA ENGINEERING COLLEGE

3456	Tom	Huntingdon	16652
5948	Harry	Alexandria	16603

There's clearly redundant information stored here.

Consistency and integrity are harder to maintain even in this simple example, e.g., ensuring the fact that the zip code always refers the same city and the city is spelled consistently.

Note we don't have a zip code to city fact stored unless there is a person from that zipcode

Redundancy and Other Problems

Set-valued or multi-valued attributes in the E-R diagram result in multiple rows in corresponding table

Example: Person (*SSN*, *Name*, *Address*, ***Hobbies***)

- A person entity with multiple hobbies yields multiple rows in table Person
- Hence, the association between *Name* and *Address* for the same person is stored redundantly
- *SSN* is key of entity set, but (*SSN*, *Hobbies*) is key of corresponding relation below

The relation Person can't describe people without hobbies
but more important is the replication of what would be the key value

SSN	Name	Address	Hobbies
1111	Joe	123 Main	hiking
1111	Joe	123 Main	biking
2222	Mary	321 Elm	lacross

Normalization

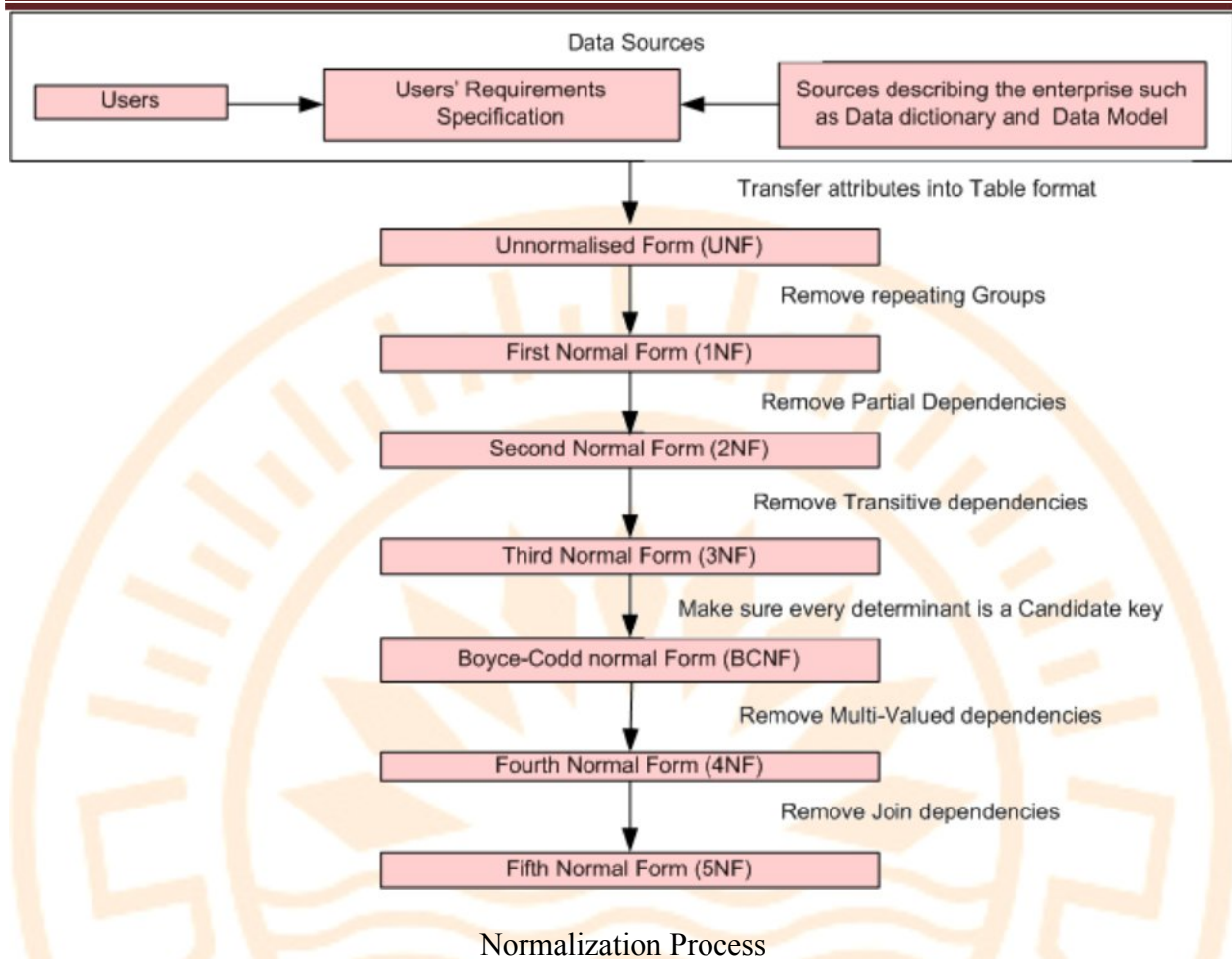
Database Normalization is a step wise formal process that allows us to decompose Database Tables in such a way that both Data Redundancy and Update Anomalies(see above for more info on update anomalies) are minimised.

It makes use of **Functional Dependencies** that exist in a table (relation, more formally) and the primary key or Candidate Keys in analysing the tables. Three normal forms were initially proposed called First normal Form (1NF), Second normal Form (2NF), and Third normal Form (3NF). Subsequently R.Boyce and E.F.Codd introduced a stronger definition of 3NF called Boyce-Codd Normal Form(BCNF).

PRATHYUSHA ENGINEERING COLLEGE

With the exception of 1NF, all these normal forms are based on Functional dependencies among the attributes of a table. Higher normal forms that go beyond BCNF were introduced later such as Fourth Normal Form (4NF) and Fifth Normal Form (5NF). However these later normal forms deal with situations that are very rare.

- **First Normal Form (1NF)** The only thing that is required for a table to be in 1NF is to contain only atomic values (intersection of each row and column should contain one and only one value).this is sometimes referred to as : Eliminate Repeating groups.
- **Second Normal Form (2NF)** A Table is said to be in 2NF if it is in 1NF and there are no partial dependencies i.e. every non primary key attribute of the Table is fully functionally dependent on the primary key.
- **Third Normal Form (3NF)** A Table that is in 1NF and 2NF and in which no non primary key attribute is transitively dependent on primary key. **Boyce-codd Normal Form (BCNF)** A Table is in BCNF if and only if every determinant(it is an attribute or a group of attributes on which some other attribute is fully functionally dependent, see functional dependency described above) is a candidate key. BCNF is a stronger form of 3NF.
The difference between 3NF and BCNF is that for a Functional dependency $A \rightarrow B$, 3NF allows this dependency in a table if attribute **B** is a primary key attribute and attribute **A** is not a **candidate key**, where as **BCNF** insists that for this dependency to remain in a table, attribute **A** must be a candidate key.
- **Fourth Normal Form (4NF)** 4NF is a stronger normal form than BCNF as it prevents Tables from containing nontrivial Multi-Valued Dependencies (MVDs) and hence data redundancy.
The Normalization of BCNF Tables to 4NF involves the removal of MVDs from the Table by placing the attribute(s) in a new Table along with the copy of the determinant(s).
- **Fifth Normal Form(5NF)** 5NF is also called **Project-Join Normal Form(PJRF)** and specifies that a 5NF Table has no **Join dependency**



Domain-key normal form (DK/NF)

Boyce-Codd normal form (BCNF), fourth normal form (4NF), and fifth normal form (5NF) are examples of such forms. Each form eliminates a possible modification anomaly but doesn't guarantee prevention of all possible modification anomalies. Domain-key normal form, however, provides such a guarantee.

A relation is in *domain-key normal form (DK/NF)* if every constraint on the relation is a logical consequence of the definition of keys and domains. A *constraint* in this definition is any rule that's precise enough that you can evaluate whether or not it's true. A *key* is a unique identifier of a row in a table. A *domain* is the set of permitted values of an attribute.

Look at this database, which is in 1NF, to see what you must do to put that database in DK/NF.

PRATHYUSHA ENGINEERING COLLEGE

SALES		
Customer_ID	Product	Price
1001	Laundry detergent	12
1007	Toothpaste	3
1010	Chlorine bleach	4
1024	Toothpaste	3

Table: SALES (Customer_ID, Product, Price)

Key: Customer_ID

Constraints:

- Customer_ID determines Product
- Product determines Price
- Customer_ID must be an integer > 1000

To enforce Constraint 3 (that Customer_ID must be an integer greater than 1000), you can simply define the domain for Customer_ID to incorporate this constraint. That makes the constraint a logical consequence of the domain of the CustomerID column. Product depends on Customer_ID, and Customer_ID is a key, so you have no problem with Constraint 1, which is a logical consequence of the definition of the key.

Constraint 2 *is* a problem. Price depends on (is a logical consequence of) Product, and Product isn't a key. The solution is to divide the SALES table into two tables. One table uses Customer_ID as a key, and the other uses Product as a key. The database, besides being in 3NF, is also in DK/NF.

Design your databases so they're in DK/NF if possible. If you can do that, enforcing key and domain restrictions causes all constraints to be met, and modification anomalies aren't possible. If a database's structure is designed to prevent you from putting it into DK/NF, then you have to build the constraints into the application program that uses the database. The database itself doesn't guarantee that the constraints will be met.

Denormalization

Denormalization is the process of attempting to optimize the read performance of a database by adding redundant data or by grouping data.^{[1][2]} In some cases, denormalization helps cover up the inefficiencies inherent in relational database software. A relational normalized

database imposes a heavy access load over physical storage of data even if it is well tuned for high performance.

A normalized design will often store different but related pieces of information in separate logical tables (called relations). If these relations are stored physically as separate disk files, completing a database query that draws information from several relations (a *join operation*) can be slow. If many relations are joined, it may be prohibitively slow. There are two strategies for dealing with this. The preferred method is to keep the logical design normalized, but allow the database management system (DBMS) to store additional redundant information on disk to optimize query response. In this case it is the DBMS software's responsibility to ensure that any redundant copies are kept consistent. This method is often implemented in SQL as indexed views (Microsoft SQL Server) or materialized views (Oracle). A view represents information in a format convenient for querying, and the index ensures that queries against the view are optimized.

The more usual approach is to denormalize the logical data design. With care this can achieve a similar improvement in query response, but at a cost—it is now the database designer's responsibility to ensure that the denormalized database does not become inconsistent. This is done by creating rules in the database called *constraints*, that specify how the redundant copies of information must be kept synchronized. It is the increase in logical complexity of the database design and the added complexity of the additional constraints that make this approach hazardous. Moreover, constraints introduce a trade-off, speeding up reads (SELECT in SQL) while slowing down writes (INSERT, UPDATE, and DELETE). This means a denormalized database under heavy write load may actually offer *worse* performance than its functionally equivalent normalized counterpart.

A denormalized data model is not the same as a data model that has not been normalized, and denormalization should only take place after a satisfactory level of normalization has taken place and that any required constraints and/or rules have been created to deal with the inherent anomalies in the design. For example, all the relations are in third normal form and any relations with join and multi-valued dependencies are handled appropriately.

Examples of denormalization techniques include:

- Materialized views, which may implement the following:
 - Storing the count of the "many" objects in a one-to-many relationship as an attribute of the "one" relation
 - Adding attributes to a relation from another relation with which it will be joined
- Star schemas, which are also known as fact-dimension models and have been extended to snowflake schemas
- Prebuilt summarization or OLAP cubes

Denormalization techniques are often used to improve the scalability of Web applications.

UNIT II SQL & QUERY OPTIMIZATION

8

SQL Standards - Data types - Database Objects- DDL-DML-DCL-TCL-Embedded SQL-Static Vs Dynamic SQL - QUERY OPTIMIZATION: Query Processing and Optimization - Heuristics and Cost Estimates in Query Optimization.

SQL Standards

The SQL-86 standard was the initial version. The IBM Systems Application Architecture (SAA) standard for SQL was released in 1987. As people identified the need for more features, updated versions of the formal SQL standard were developed, called SQL-89 and SQL-92.

The latest version of the SQL standard, called SQL:1999, adds a variety of features to SQL. We have seen many of these features in earlier chapters, and will see a few in later chapters. The standard is broken into several parts:

- SQL/Framework (Part 1) provides an overview of the standard.
- SQL/Foundation (Part 2) defines the basics of the standard: types, schemas, tables, views, query and update statements, expressions, security model, predicates, assignment rules, transaction management and so on.
- SQL/CLI (Call Level Interface) (Part 3) defines application program interfaces to SQL.
- SQL/PSM (Persistent Stored Modules) (Part 4) defines extensions to SQL to make it procedural.
- SQL/Bindings (Part 5) defines standards for embedded SQL for different embedding languages.

There are several other parts underdevelopment, including

- Part 7: SQL/Temporal deals with standards for temporal data.
- Part 9: SQL/MED (Management of External Data) defines standards for interfacing an SQL system to external sources. By writing wrappers, system designers can treat external data sources, such as files or data in nonrelational databases, as if they were “foreign” tables.

- Part 10: SQL/OLB (Object Language Bindings) defines standards for embedding SQL in Java.

The missing numbers (Parts 6 and 8) cover features such as distributed transaction processing and multimedia data, for which there is as yet no agreement on the standards.

The multimedia standards propose to cover storage and retrieval of text data, spatial data, and still images.

DDL

Data Definition Language (DDL) statements are used to define the database structure or schema. Some examples:

- CREATE - to create objects in the database
- ALTER - alters the structure of the database
- DROP - delete objects from the database
- TRUNCATE - remove all records from a table, including all spaces allocated for the records are removed
- RENAME - rename an object

Data Definition:

The Data Definition Language is used to create an object(e.g.table),alter the structure of an object and also to drop the object created. Now let us look at the concepts related to Definition language.

Table Definition:

A table is a unit of storage which holds data in the form of rows and columns. The Data Definition Languages used for table definition can be classified into following categories

*Create Table Command.

*Alter Table Command.

(i)Add

(ii)Modify

*Drop Table Command.

*Truncate Table Command.

*Rename Table Command.

Table Creation

This is used to create a table in oracle.

Syntax:

Create table <table name> (column1 datatype(size),column2 datatype(size),...);

Example:

```
SQL> create table emp(regno number(5), name varchar(20));
```

Table created.

DESC

This command is used to display the structure of the table.

```
SQL> desc emp;
```

Name	Null?	Type
REGNO		NUMBER(5)
NAME		VARCHAR2(20)

Alter Table Command

This command is used to modify that already existing table but we can not change or alter the table name. The alter commands are:

ADD

This command is used to add column to existing table. It is used to modify the column in already existing table.

Syntax

```
Alter table <tablename> add (column datatype(size));
```

Example

```
SQL> alter table emp add(dept varchar(10));
```

Table altered.

```
SQL> desc emp;
```

Name	Null?	Type
REGNO		NUMBER(5)
NAME		VARCHAR2(20)
DEPT		VARCHAR2(10)

MODIFY

This command is used to modify the column in the table.

Syntax:

```
Alter table <tablename> modify (column datatype(size));
```

Example

```
SQL> alter table emp modify(dept varchar(20));
```

Table altered.

```
SQL> desc emp;
```

Name	Null?	Type
REGNO		NUMBER(5)
NAME		VARCHAR2(20)
DEPT		VARCHAR2(20)

RENAME

This command is used to rename the table name .

Syntax

Rename <old tablename> to <newtablename>;

Example

SQL> rename emp to emp1;

Table renamed.

TRUNCATE

This command is used to delete the data or content present in the table.

Syntax

Truncate table <tablename>;

Example

SQL> truncate table emp1;

Table truncated.

DROP

This command is used to drop the table completely from oracle.

Syntax

Drop table <tablename>;

Example

SQL> drop table emp1;

Table dropped.

DML

Data Manipulation Language (DML) statements are used for managing data within schema objects. Some examples:

- SELECT - retrieve data from the a database
- INSERT - insert data into a table
- UPDATE - updates existing data within a table
- DELETE - deletes all records from a table, the space for the records remain

(1)INSERT

This DML command is used to insert the details into the table.

Syntax

```
insert into tablename values ('&field1', '&field2',... );
```

Example

```
SQL> insert into scott. vidhya values(1111,'mala','a');
```

1 row created.

(2)SELECT

This command is used to show the details present in a table.

Syntax

```
select * from <table name >;
```

Example

```
select * from scott. vidhya;
```

REGNO	NAME	GRADE
1000	Sunil	a
1001	Allen	b
1002	Vidhya	b
1003	Jaya	b

(3) UPDATE

This command is used to change a value of field in a row.

Syntax

Update <tablename> set < field='new value'> where <field='oldvalue'>;

Example

SQL> Update stu set grade=a where name='jaya';

1 rows updated

(4)DELETE

This command is used to delete a row in table.

Syntax

Delete from <tablename> where< fieldname='value'>;

Example

SQL> delete from scott. vidhya where grade='c';

2 rows deleted.

DCL

Data Control Language (DCL) statements. Some examples:

- GRANT - gives user's access privileges to database
- REVOKE - withdraw access privileges given with the GRANT command

TCL

Transaction Control (TCL) statements are used to manage the changes made by DML statements. It allows statements to be grouped together into logical transactions.

TCL Commands are : (i)Save point (ii)Commit (iii)Rollback.

(I)SAVEPOINT:

It is used to set a mark on the table.(identify a point in a transaction to which you can later roll back)

Syntax:

Savepoint <savepoint name>;

(II) COMMIT:

This TCL command is used to make changes permanent.(save work done)

Syntax:

Commit;

(III)ROLLBACK:

This TCL command is used to perform an undo operation.(restore database to original since the last COMMIT)

Syntax:

Rollback to <savepoint name>;

More DML queries

GROUP BY

To group the records based on some attribute

Syntax:

Select <attribute list> from <table name> group by <attribute name>

Ex : Select avg(salary) from emp group by city;

ORDER BY

The ORDER BY keyword sorts the records in ascending order by default. To sort the records in a descending order, you can use the DESC keyword.

Syntax :

Select <attribute name> from <table name> order by <attribute name> asc / desc;

Ex.

- 1) SELECT * FROM Customers ORDER BY Country;
- 2) SELECT * FROM Customers ORDER BY Country desc;

AGGREGATE FUNCTIONS

Count(*) – To count the record in the table

Count(x) – to count the specific object in the table.

Aggregate Function	Returns...
min(x)	the smallest value in column x
max(x)	the largest value in column x
avg(x)	the average value in column x
stdev(x)	the standard deviation of the values in column x
count(x)	the number of values in column x
count(*)	the number of records in the table being searched

Ex.

- **Select min(marks) from student;**
(retrieve the minimum mark from student table)
- **Select max(marks) from student;**
(retrieve the maximum mark from student table)
- **Select avg(marks) from student;**
(retrieve the average mark from student table)
- **Select sum(marks) from student;**
(retrieve the sum mark from student table)

- **Select stdev(marks) from student;**

(retrieve the standard deviation of marks from student table)

- **Select count(*) from student;**

(retrieve the number of records from student table)

Embedded SQL

SQL statements can be embedded in a general-purpose programming language. The programming language in which the sql statements are to be embedded is called the host language.

Retrieving Single Tuples with Embedded SQL

Within an embedded SQL command, we may refer variables(Host language variables). These are called shared variables because they used in both sql queries and also in the host language. shared variables are prefixed with (:) when they are in an SQL statement.

Example for Shared variables

- 0) int loop ;
- 1) EXEC SQL BEGIN DECLARE SECTION ;
- 2) varchar dname [16], fname [16], lname [16], address [31] ;
- 3) char ssn [10], bdate [11], sex [2], minit [2] ;
- 4) float salary, raise ;
- 5) int dno, dnumber ;
- 6) int SQLCODE ; char SQLSTATE [6] ;
- 7) EXEC SQL END DECLARE SECTION ;

Connecting to the Database. The SQL command for establishing a connection to a database has the following form:

CONNECT TO <server name>AS <connection name>

AUTHORIZATION <user account name and password> ;

The programmer or user can use the <connection name> to change from the currently active connection to a different one by using the following command:

SET CONNECTION <connection name> ;

Once the connection is no longer needed, it can be terminated by the following command :

Disconnect <connection name>;

Communicating between the Program and the DBMS Using SQLCODE and SQLSTATE.

The two special communication variables that are used by the DBMS to communicate exception or error conditions to the program are SQLCODE and SQLSTATE.

After each database command is executed, the DBMS returns a value in SQLCODE. A value of 0 indicates that the statement was executed successfully by the DBMS. If $\text{SQLCODE} > 0$ (or, more specifically, if $\text{SQLCODE} = 100$), this indicates that no more data (records) are available in a query result. If $\text{SQLCODE} < 0$, this indicates some error has occurred.

//Program Segment E1:

```
0) loop = 1 ;
1) while (loop) {
2) prompt("Enter a Social Security Number: ", ssn) ;
3) EXEC SQL
4) select Fname, Minit, Lname, Address, Salary
5) into :fname, :mininit, :lname, :address, :salary
6) from EMPLOYEE where Ssn = :ssn ;
7) if (SQLCODE == 0) printf(fname, mininit, lname, address, salary)
8) else printf("Social Security Number does not exist: ", ssn) ;
9) prompt("More Social Security Numbers (enter 1 for Yes, 0 for No): ", loop) ;
10) }
```

When a single record is retrieved, the programmer can assign its attribute values directly to C program variables in the **into** clause. When a single record is retrieved, the programmer can assign its attribute values directly to C program variables in the **cursor** is used to allow tuple-at-a-time processing of a query result by the host language program.

Retrieving Multiple Tuples with Embedded SQL Using Cursors

We can think of a cursor as a pointer that points to a single tuple (row) from the result of a query that retrieves multiple tuples. An OPEN CURSOR command fetches the query result from the

database and sets the cursor to a position before the first row in the result of the query. This becomes the current row

for the cursor. Subsequently FETCH commands are issued in the program; each FETCH moves the cursor to the next row in the result of the query, making it the current row and copying its attribute values into the host language program variables specified in the FETCH command by an INTO clause.

CLOSE CURSOR command is issued to indicate that we are done with processing the result of the query associated with that cursor.

```
0) prompt("Enter the Department Name: ", dname) ;
1) EXEC SQL
2) select Dnumber into :dnumber
3) from DEPARTMENT where Dname = :dname ;
4) EXEC SQL DECLARE EMP CURSOR FOR
5) select Ssn, Fname, Minit, Lname, Salary
6) from EMPLOYEE where Dno = :dnumber
7) FOR UPDATE OF Salary ;
8) EXEC SQL OPEN EMP ;
9) EXEC SQL FETCH from EMP into :ssn, :fname, :minit, :lname, :salary ;
10) while (SQLCODE == 0) {
11) printf("Employee name is:", Fname, Minit, Lname) ;
12) prompt("Enter the raise amount: ", raise) ;
13) EXEC SQL
14) update EMPLOYEE
15) set Salary = Salary + :raise
16) where CURRENT OF EMP ;
17) EXEC SQL FETCH from EMP into :ssn, :fname, :minit, :lname, :salary ;
18) }
19) EXEC SQL CLOSE EMP ;
```

the clause FOR UPDATE OF in the cursor declaration and list the names of any

attributes that will be updated by the program. If rows are to be deleted, the keywords FOR UPDATE must be added without specifying any attributes. In the embedded UPDATE (or DELETE) command, the condition WHERE CURRENT OF <cursor name> specifies that the current tuple referenced by the cursor is the one to be updated (or deleted).

General Options for a Cursor Declaration. Several options can be specified when declaring a cursor. The general form of a cursor declaration is as follows:

```
DECLARE <cursor name> [ INSENSITIVE ] [ SCROLL ] CURSOR  
[ WITH HOLD ] FOR <query specification>  
[ ORDER BY <ordering specification> ]  
[ FOR READ ONLY | FOR UPDATE [ OF <attribute list> ] ] ;
```

The general form of a FETCH command is as follows, with the parts in square brackets being optional:

```
FETCH [ [ <fetch orientation> ] FROM ] <cursor name> INTO <fetch target list> ;
```

The ORDER BY clause orders the tuples so that the FETCH command will fetch them in the specified order.

STATIC SQL

Static SQL statements do not change from execution to execution. The full text of static SQL statements are known at compilation, which provides the following benefits:

- Successful compilation verifies that the SQL statements reference valid database objects.
- Successful compilation verifies that the necessary privileges are in place to access the database objects.

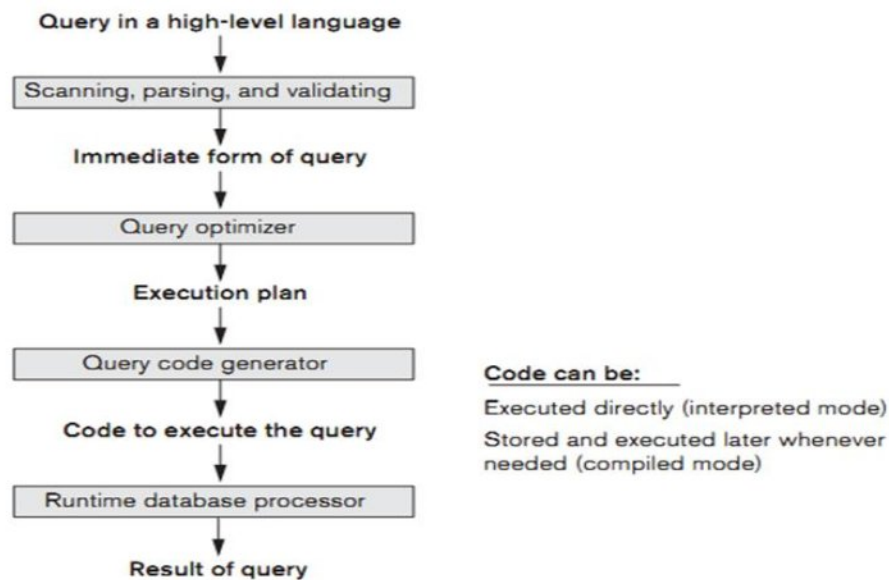
DYNAMIC SQL

Dynamic SQL to execute DML statements in which the exact SQL statement is not known until runtime

Dynamic SQL to create applications that execute dynamic queries, which are queries whose full text is not known until runtime. Many types of applications need to use dynamic queries, including:

- Applications that allow users to input or choose query search or sorting criteria at runtime
- Applications that allow users to input or choose optimizer hints at run time
- Applications that query a database where the data definitions of tables are constantly changing
- Applications that query a database where new tables are created often.

Query Processing and Optimization



Process for heuristics optimization

1. The parser of a high-level query generates an initial internal representation;
2. Apply heuristics rules to optimize the internal representation.
3. A query execution plan is generated to execute groups of operations based on the access paths available on the files involved in the query.

The main heuristic is to apply first the operations that reduce the size of intermediate results.

E.g., Apply SELECT and PROJECT operations before applying the JOIN or other binary operations.

Heuristic Optimization of Query Trees:

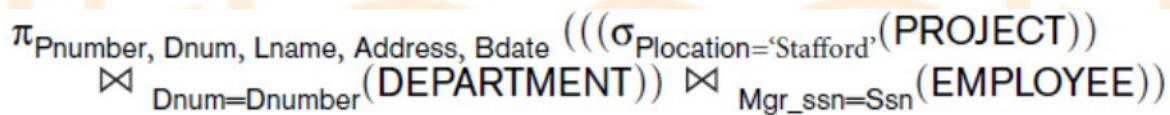
- The same query could correspond to many different relational algebra expressions — and hence many different query trees.

The task of heuristic optimization of query trees is to find a **final query tree** that is efficient to execute.

A **query tree** is a tree data structure that corresponds to a relational algebra expression.

It represents the input relations of the query as leaf nodes of the tree, and represents the relational algebra operations as internal nodes. An execution of the query tree consists of executing an internal node operation whenever its operands are available and then replacing that internal node by the relation that results from executing the operation. The order of execution of operations starts at the leaf nodes, which represents the input database relations for the query, and ends at the root node, which represents the final operation of the query. The execution terminates when the root node operation is executed and produces the result relation for the query.

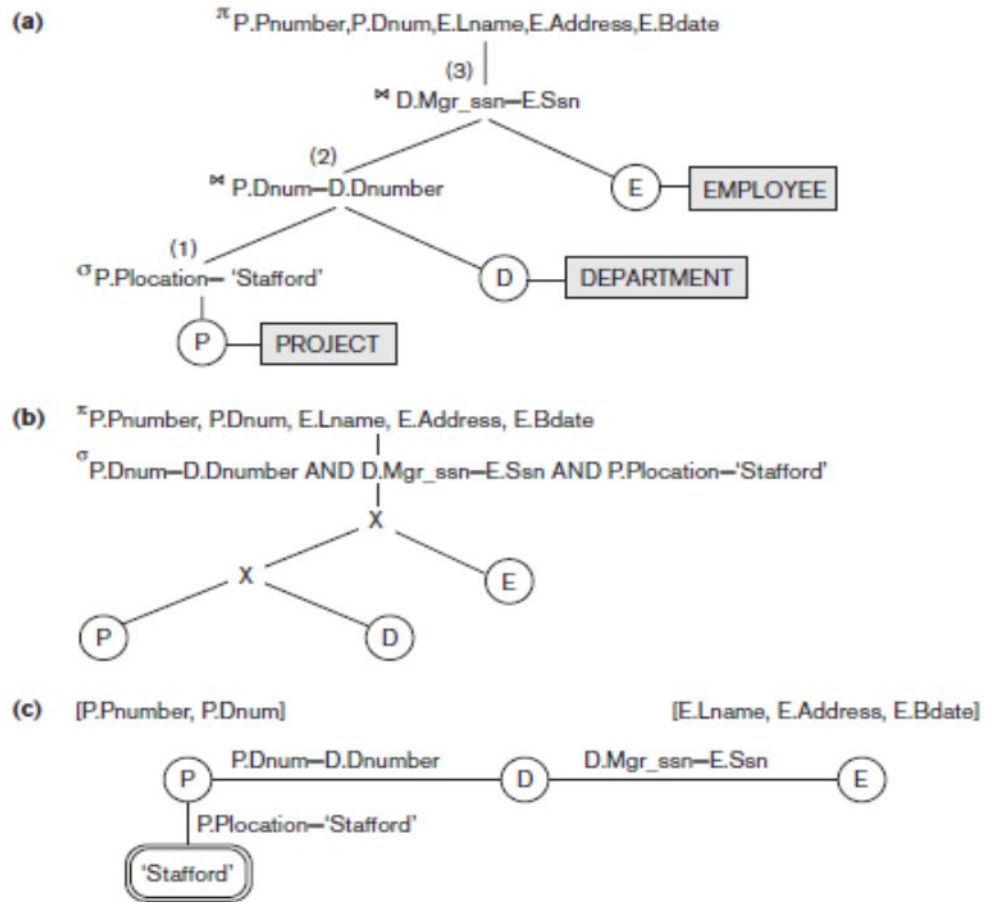
Example 1:



The diagram shows a relational algebra expression: $\pi_{Pnumber, Dnum, Lname, Address, Bdate}(((\sigma_{Plocation='Stafford'}(PROJECT)) \bowtie_{Dnum=Dnumber} (DEPARTMENT)) \bowtie_{Mgr_ssn=Ssn} (EMPLOYEE))$. It represents a query that projects specific attributes from the PROJECT, DEPARTMENT, and EMPLOYEE tables, filtered by location and manager-employee relationships.

This corresponds to the following SQL query:

Q : SELECT P.Pnumber, P.Dnum, E.Lname, E.Address, E.Bdate FROM PROJECT AS P, DEPARTMENT AS D, EMPLOYEE AS E WHERE P.Dnum=D.Dnumber AND D.Mgr_ssn=E.Ssn AND P.Plocation= 'Stafford';

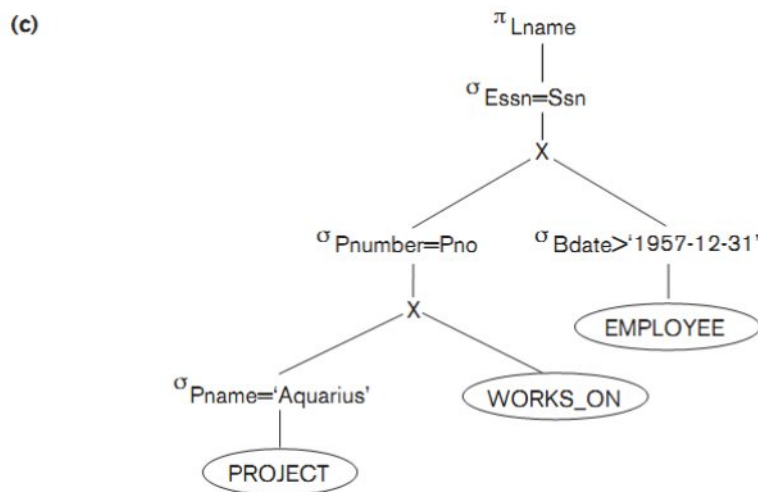
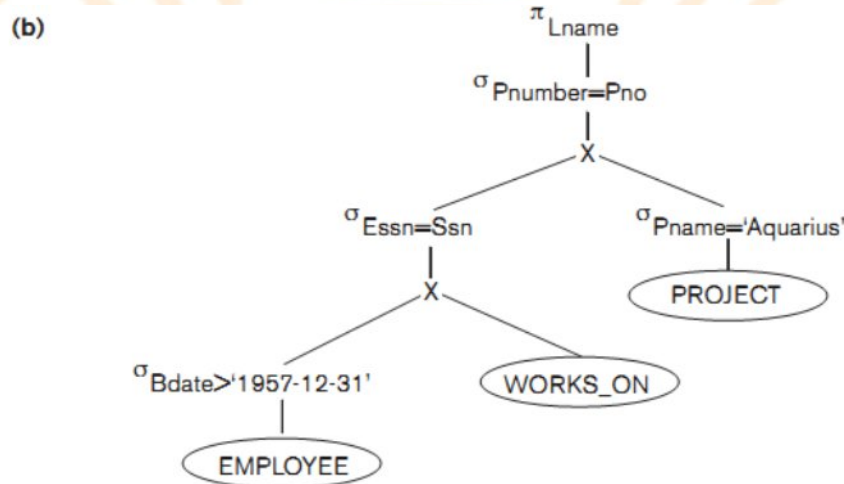
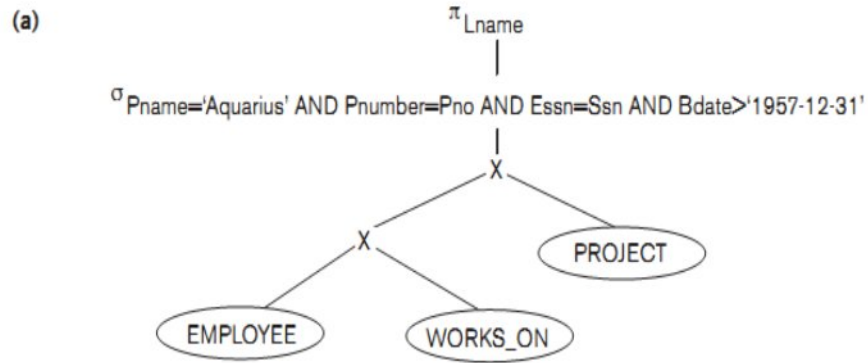


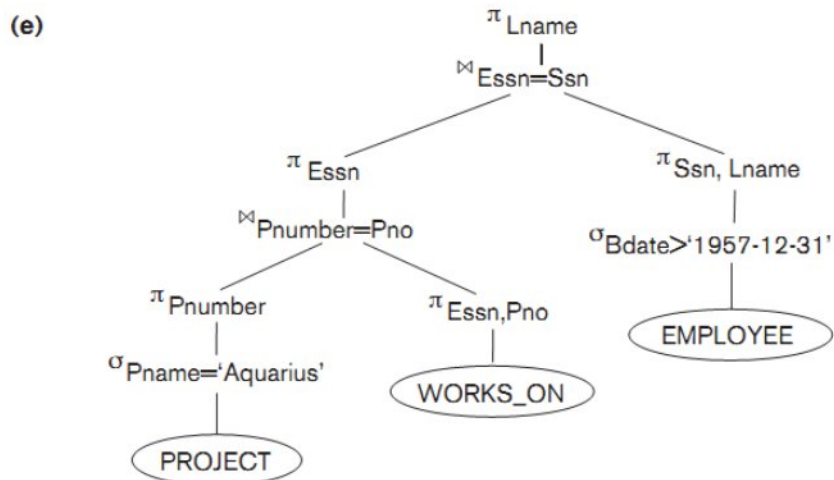
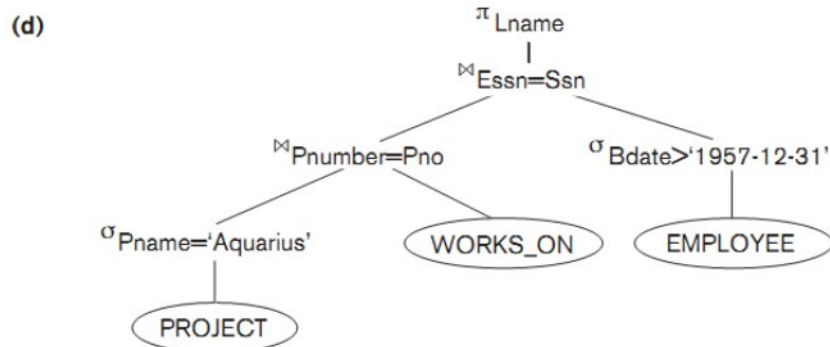
Two query trees for the query Q (a) Query tree corresponding to the relational algebra expression for Q (b) Initial (canonical) query tree for SQL query Q (c) Query graph for Q

Example 2:

Query: SELECT LNAME FROM EMPLOYEE, WORKS_ON, PROJECT WHERE PNAME = 'AQUARIUS' AND PNMUBER=PNO AND ESSN=SSN AND BDATE > '1957-12-31';

- Initial (canonical) query tree for SQL query.
- Moving SELECT operations down the query tree.
- Applying the more restrictive select operation first
- Replacing CARTESIAN PRODUCT AND select with join operations.
- Moving PROJECT operations down the query tree.





General Transformation Rules for Relational Algebra Operations. There are many rules for transforming relational algebra operations into equivalent ones.

Some transformation rules

1. Cascade of A conjunctive selection condition can be broken up into a cascade (that is, a sequence) of individual operations:

2. Commutativity of . The operation is commutative:

3. Cascade of In a cascade (sequence) of operations, all but the last one can be ignored:

4. Commuting with σ_c . If the selection condition c involves only those attributes A_1, \dots, A_n in the projection list, the two operations can be commuted:

5. Commutativity of σ_c and \bowtie . The join operation is commutative, as is the σ_c operation:

$R \bowtie \sigma_c S = \sigma_c (R \bowtie S)$
 $R \bowtie S = \sigma_c (R \bowtie S)$

6. Commuting σ_c with \bowtie (or \bowtie with σ_c). If all the attributes in the selection condition c involve only the attributes of one of the relations being joined—say, R —the two operations can be commuted as follows:

$(R \bowtie S) \sigma_c = (R \sigma_c) \bowtie S$

7. Commuting π_L with \bowtie (or \bowtie with π_L). Suppose that the projection list is $L = \{A_1, \dots, A_n, B_1, \dots, B_m\}$, where A_1, \dots, A_n are attributes of R and B_1, \dots, B_m are attributes of S . If the join condition c involves only attributes in L , the two operations can be commuted as follows:

8. Commutativity of set operations. The set operations \cup and \cap are commutative but \setminus is not.

9. Associativity of \cup , \cap , \setminus , and \bowtie . These four operations are individually associative; that is, if θ stands for any one of these four operations (throughout the expression), we have:

10. Commuting with set operations. The operation commutes with \cup , \cap , and \times . If θ stands for any one of these three operations (throughout the expression), we have:

11. The θ operation commutes with \cup .

12. Converting a θ sequence into θ . If the condition c of a θ that follows a \times corresponds to a join condition, convert the θ sequence into a θ as follows:

Outline of a Heuristic Algebraic Optimization Algorithm

The steps of the algorithm are as follows:

1. Using Rule 1, break up any SELECT operations with conjunctive conditions into a cascade of SELECT operations. This permits a greater degree of freedom in moving SELECT operations down different branches of the tree.
2. Using Rules 2, 4, 6, and 10 concerning the commutativity of SELECT with other operations, move each SELECT operation as far down the query tree as is permitted by the attributes involved in the select condition.
3. Using Rules 5 and 9 concerning commutativity and associativity of binary operations, rearrange the leaf nodes of the tree.

4. Using Rule 12, combine a CARTESIAN PRODUCT operation with a subsequent SELECT operation in the tree into a JOIN operation, if the condition represents a join condition.
5. Using Rules 3, 4, 7, and 11 concerning the cascading of PROJECT and the commuting of PROJECT with other operations, break down and move lists of projection attributes down the tree as far as possible by creating new PROJECT operations as needed
6. Identify subtrees that represent groups of operations that can be executed by a single algorithm.

Summary of Heuristics for Algebraic Optimization.

- The main heuristic is to apply first the operations that reduce the size of intermediate results.
- This includes performing as early as possible SELECT operations to reduce the number of tuples and Performing PROJECT operations as early as possible to reduce the number of attributes—by moving SELECT and PROJECT operations as far down the tree as possible.
- The latter rule is accomplished through reordering the leaf nodes of the tree among themselves while avoiding Cartesian products, and adjusting the rest of the tree appropriately.

Converting Query Trees into Query Execution Plans

With **materialized evaluation**, the result of an operation is stored as a temporary relation (that is, the result is physically materialized).

For instance, the JOIN operation can be computed and the entire result stored as a temporary relation, which is then read as input by the algorithm that computes the PROJECT operation, which would produce the query result table. On the other hand,

With **pipelined evaluation**, as the resulting tuples of an operation are produced, they are forwarded directly to the next operation in the query sequence.

For example, as the selected tuples from DEPARTMENT are produced by the SELECT operation, they are placed in a buffer; the JOIN operation algorithm would then consume the tuples from the buffer, and those tuples that result from the JOIN operation are pipelined to the projection operation algorithm. The advantage of pipelining is the cost savings in not having to write the intermediate results to disk and not having to read them back for the next operation.

Cost Components for Query Execution

The cost of executing a query includes the following components:

1. Access cost to secondary storage.

I/O Cost: This is the cost of transferring (reading and writing) data blocks between secondary disk storage and main memory buffers.

Searching Cost: The cost of searching for records in a disk file depends on the type of access structures on that file, such as ordering, hashing, and primary or secondary indexes.

In addition, factors such as whether the file blocks are allocated contiguously on the same disk cylinder or scattered on the disk affect the access cost.

2. Disk storage cost. This is the cost of storing on disk any intermediate files that are generated by an execution strategy for the query.

3. Computation cost. This is the cost of performing in-memory operations on the records within the data buffers during query execution. Such operations include searching for and sorting records, merging records for a join or a sort operation, and performing computations on field values. This is also known as CPU (central processing unit) cost.

4. Memory usage cost. This is the cost pertaining to the number of main memory buffers needed during query execution.

5. Communication cost. This is the cost of shipping the query and its results from the database site to the site or terminal where the query originated. In distributed databases (see Chapter 25), it would also include the cost of transferring tables and results among various computers during query evaluation.

UNIT – III

Introduction-Properties of Transaction- Serializability- Concurrency Control – Locking Mechanisms-Two Phase Commit Protocol-Dead lock.

Transaction :

A transaction is an atomic logical unit of work that should either be completed in its entirety or not done at all. For recovery purposes, the system needs to keep track of when each transaction starts, terminates, and commits or aborts.

Transaction Properties

There are four important properties of transaction that a DBMS must ensure to maintain data in the case of concurrent access and system failures. These are:

Atomicity: (all or nothing)

A transaction is said to be atomic if a transaction always executes all its actions in one step or not executes any actions at all. It means either all or none of the transactions operations are performed.

Consistency: (No violation of integrity constraints)

A transaction must preserve the consistency of a database after the execution. The DBMS assumes that this property holds for each transaction. Ensuring this property of a transaction is the responsibility of the user.

Isolation: (concurrent changes invisibles)

The transactions must behave as if they are executed in isolation. It means that if several transactions are executed concurrently the results must be same as if they were executed serially in some order. The data used during the execution of a transaction cannot be used by a second transaction until the first one is completed.

Durability: (committed update persist)

The effect of completed or committed transactions should persist even after a crash. It means once a transaction commits, the system must guarantee that the result of its operations will never be lost, in spite of subsequent failures.

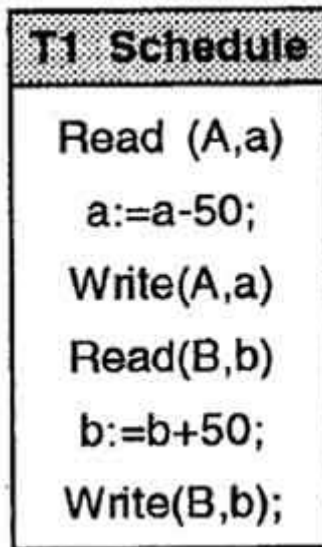
The acronym ACID is sometimes used to refer above four properties of transaction that we have presented here: Atomicity, Consistency, Isolation, and Durability.

Example

In order to understand above properties consider the following example:

Let, T_i is a transaction that transfers Rs 50 from account A to account B. This transaction can be defined as:





Atomicity

Suppose that, just prior to execution of transaction T_i the values of account A and B are Rs.1000 and Rs.2000.

Now, suppose that during the execution of T_i , a power failure has occurred that prevented the T_i to complete successfully. The point of failure may be after the completion Write (A,a) and before Write(B,b). It means that the changes in A are performed but not in B. Thus the values of account A and B are Rs.950 and Rs.2000 respectively. We have lost Rs.50 as a result of this failure.

Now, our database is in inconsistent state.

The reason for this inconsistent state is that our transaction is completed partially and we save the changes of uncommitted transaction. So, in order to get the consistent state, database must be restored to its original values i.e. A to Rs.1000 and B to Rs.2000, this leads to the concept of atomicity of transaction. It means that in order to maintain the consistency of database, either all or none of transaction's operations are performed.

In order to maintain atomicity of transaction, the database system keeps track of the old values of any write and if the transaction does not complete its execution, the old values are restored to make it appear as the transaction never executed.

PRATHYUSHA ENGINEERING COLLEGE

Consistency

The consistency requirement here is that the sum of A and B must 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.

Isolation

If several transactions are executed concurrently (or in parallel), then each transaction must behave as if it was executed in isolation. It means that concurrent execution does not result an inconsistent state.

For example, consider another transaction T2, which has to display the sum of account A and B. Then, its result should be Rs.3000.

Let's suppose that both T1 and T2 perform concurrently, their schedule is shown below:

T1	T2	Status
Read (A,a) a:=a-50 Write(A,a)		value of A i.e.1000 is copied to local variable a local variable a=950 value of local variable a 950 is copied to database item A
	Read(A,a) Read(B,b) display(a+b)	value of database item A 950 is copied to a value of database item B 2000 is copied to b 2950 is displayed as sum of accounts A and B
Read(B,b) b:=b+50 Write(B,b)		value of data base item B 2000 is copied to local variable b local variable b=2050 value of local variable b 2050 is copied to database item B

The above schedule results inconsistency of database and it shows Rs.2950 as sum of accounts A and B instead of Rs.3000. The problem occurs because second concurrently running transaction T2, reads A and B at intermediate point and computes its sum, which results inconsistent value.

Isolation property demands that the data used during the execution of a transaction cannot be used by a second transaction until the first one is completed.

A solution to the problem of concurrently executing transaction is to execute each transaction serially 'that is one after the other. However, concurrent execution of transaction provides significant performance benefits, so other solutions are developed they allow multiple transactions to execute concurrently.

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 will 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, even if there is a system failure after the transaction completes execution. Ensuring durability is the responsibility of a component of the database system called the recovery-management component.

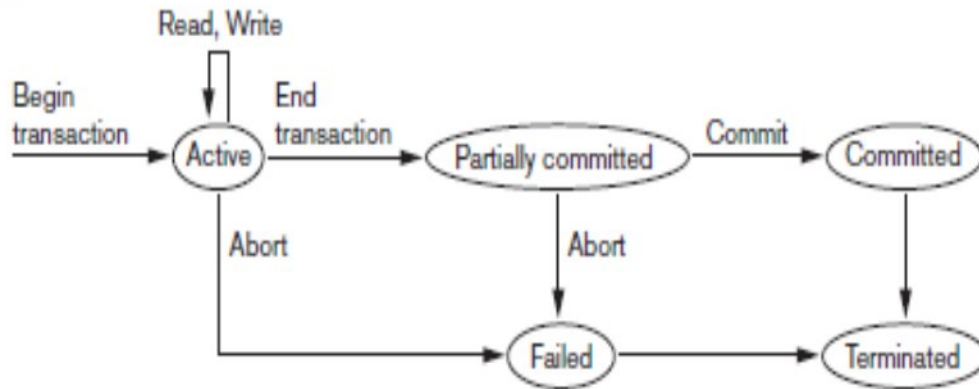
States of Transaction

A transaction must be in one of the following states:

- **Active:** the initial state, the transaction stays in this state while it is executing.
- **Partially committed:** after the final statement has been executed.
- **Failed:** when the normal execution can no longer proceed.
- **Aborted:** after the transaction has been rolled back and the database has been restored to its state prior to the start of the transaction.
- **Committed:** after successful completion.

The state diagram corresponding to a transaction is shown in Figure.

State transition diagram illustrating the states for transaction execution.



We say that a transaction has committed only if it has entered the committed state. Similarly, we say that a transaction has aborted only if it has entered the aborted state. A transaction is said to have terminated if it has either committed or aborted.

A transaction starts in the active state. When it finishes its final statement, it enters the partially committed state. At this point, the transaction has completed its execution, but it is still possible that it may have to be aborted, since the actual output may still be temporarily hiding in main memory and thus a hardware failure may preclude its successful completion.

The database system then writes out enough information to disk that, even in the event of a failure, the updates performed by the transaction can be recreated when the system restarts after the failure. When the last of this information is written out, the transaction enters the committed state.

For recovery purpose, the recovery manager of the DBMS needs to keep track of the following operations:

- **BEGIN_TRANSACTION.** This marks the beginning of transaction execution.

- **READ or WRITE.** These specify read or write operations on the database items that are executed as part of a transaction.

■ **END_TRANSACTION.** This specifies that READ and WRITE transaction operations have ended and marks the end of transaction execution. However, at this point it may be necessary to check whether the changes introduced by the transaction can be permanently applied to the database (committed) or whether the transaction has to be aborted because it violates serializability

or for some other reason.

■ **COMMIT_TRANSACTION.** This signals a *successful end* of the transaction so that any changes (updates) executed by the transaction can be safely **committed** to the database and will not be undone.

■ **ROLLBACK (or ABORT).** This signals that the transaction has *ended unsuccessfully*, so that any changes or effects that the transaction may have applied to the database must be **undone**.

Concurrent Executions

There are two good reasons for allowing concurrency:

- **Improved throughput and resource utilization.**
- **Reduced waiting time.**

The motivation for using concurrent execution in a database is essentially the same as the motivation for using **multiprogramming** in an operating system.

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 $T1$ followed by $T2$. The final values of accounts A and B , after the execution \$855 and \$2145, respectively

T_1	T_2
<code>read(A)</code> <code>A := A - 50</code> <code>write(A)</code> <code>read(B)</code> <code>B := B + 50</code> <code>write(B)</code>	<code>read(A)</code> <code>temp := A * 0.1</code> <code>A := A - temp</code> <code>write(A)</code> <code>read(B)</code> <code>B := B + temp</code> <code>write(B)</code>

Schedule 1—a serial schedule in which T_1 is followed by T_2 .

that is, the sum $A + B$ —is preserved after the execution of both transactions. 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 15.4. Again, as expected, the sum $A + B$ is preserved, and the final values of accounts A and B are \$850 and \$2150, respectively.

T_1	T_2
<code>read(A)</code> <code>A := A - 50</code> <code>write(A)</code> <code>read(B)</code> <code>B := B + 50</code> <code>write(B)</code>	<code>read(A)</code> <code>temp := A * 0.1</code> <code>A := A - temp</code> <code>write(A)</code> <code>read(B)</code> <code>B := B + temp</code> <code>write(B)</code>

Schedule 2—a serial schedule in which T_2 is followed by T_1 .

The execution sequences just described are called **schedules**. They represent the chronological order in which instructions are executed in the system. Thus, the number of possible schedules for a set of n transactions is much larger than $n!$.

These schedules are **serial**: Each serial schedule consists of a sequence of instructions from various transactions, where the instructions belonging to one single transaction appear together in that schedule. Thus, for a set of n transactions, there exist $n!$ different valid serial schedules.

Suppose that the two transactions are executed concurrently. One possible schedule appears in the following schedule.

T ₁	T ₂
read(A)	
A := A - 50	
write(A)	
	read(A)
	temp := A * 0.1
	A := A - temp
	write(A)
read(B)	
B := B + 50	
write(B)	
	read(B)
	B := B + temp
	write(B)

Schedule 3—a concurrent schedule equivalent to schedule 1

Concurrency Problems

If locking is not available and several users access a database concurrently, problems may occur if their transactions use the same data at the same time. Concurrency problems include:

- Lost or buried updates.
- Uncommitted dependency ([dirty read](#)).
- Inconsistent analysis ([nonrepeatable read](#)).
- Phantom reads.

Lost Updates

Lost updates occur when two or more transactions select the same row and then update the row based on the value originally selected. Each transaction is unaware of other transactions. The last update overwrites updates made by the other transactions, which results in lost data.

For example, two editors make an electronic copy of the same document. Each editor changes the copy independently and then saves the changed copy, thereby overwriting the original document. The editor who saves the changed copy last overwrites changes made by the first

editor. This problem could be avoided if the second editor could not make changes until the first editor had finished.

Uncommitted Dependency (Dirty Read)

Uncommitted dependency occurs when a second transaction selects a row that is being updated by another transaction. The second transaction is reading data that has not been committed yet and may be changed by the transaction updating the row.

For example, an editor is making changes to an electronic document. During the changes, a second editor takes a copy of the document that includes all the changes made so far, and distributes the document to the intended audience. The first editor then decides the changes made so far are wrong and removes the edits and saves the document. The distributed document contains edits that no longer exist, and should be treated as if they never existed. This problem could be avoided if no one could read the changed document until the first editor determined that the changes were final.

Inconsistent Analysis (Nonrepeatable Read)

Inconsistent analysis occurs when a second transaction accesses the same row several times and reads different data each time. Inconsistent analysis is similar to uncommitted dependency in that another transaction is changing the data that a second transaction is reading. However, in inconsistent analysis, the data read by the second transaction was committed by the transaction that made the change. Also, inconsistent analysis involves multiple reads (two or more) of the same row and each time the information is changed by another transaction; thus, the term nonrepeatable read.

For example, an editor reads the same document twice, but between each reading, the writer rewrites the document. When the editor reads the document for the second time, it has changed. The original read was not repeatable. This problem could be avoided if the editor could read the document only after the writer has finished writing it.

Phantom Reads

Phantom reads occur when an insert or delete action is performed against a row that belongs to a range of rows being read by a transaction. The transaction's first read of the range of rows shows a row that no longer exists in the second or succeeding read, as a result of a deletion by a

different transaction. Similarly, as the result of an insert by a different transaction, the transaction's second or succeeding read shows a row that did not exist in the original read. For example, an editor makes changes to a document submitted by a writer, but when the changes are incorporated into the master copy of the document by the production department, they find that new unedited material has been added to the document by the author. This problem could be avoided if no one could add new material to the document until the editor and production department finish working with the original document.

Serializability

The database system must control concurrent execution of transactions, to ensure that the database state remains consistent.

Since transactions are programs, it is computationally difficult to determine exactly what operations a transaction performs and how operations of various transactions interact. we consider only two operations:

read and write. We thus assume that, between a read(Q) instruction and a write(Q) instruction on a data item Q , a transaction may perform an arbitrary sequence of operations on the copy of Q that is residing in the local buffer of the transaction.

Two different forms of schedule equivalence are **Conflict serializability** and **view serializability**.

Consider the following

1. $I_i = \text{read}(Q)$, $I_j = \text{read}(Q)$. The order of I_i and I_j does not matter, since the same value of Q is read by T_i and T_j , regardless of the order.
2. $I_i = \text{read}(Q)$, $I_j = \text{write}(Q)$. If I_i comes before I_j , then T_i does not read the value of Q that is written by T_j in instruction I_j . If I_j comes before I_i , then T_i reads the value of Q that is written by T_j . Thus, the order of I_i and I_j matters.
3. $I_i = \text{write}(Q)$, $I_j = \text{read}(Q)$. The order of I_i and I_j matters for reasons similar to those of the previous case.
4. $I_i = \text{write}(Q)$, $I_j = \text{write}(Q)$. Since both instructions are write operations, the order of these instructions does not affect either T_i or T_j . However, the value obtained by the next read(Q) instruction of S is affected, since the result of

only the latter of the two write instructions is preserved in the database. If there is no other write(Q) instruction after I_i and I_j in S , then the order of I_i and I_j directly affects the final value of Q in the database state that results from schedule S .

Thus, only in the case where both I_i and I_j are read instructions does the relative order of their execution not matter.

We say that I_i and I_j **conflict** if they are operations by different transactions on the same data item, and at least one of these instructions is a write operation.

T_1	T_2
read(A)	
write(A)	
	read(A)
	write(A)
read(B)	
write(B)	
	read(B)
	write(B)

Schedule 3—showing only the read and write instructions.

The write(A) instruction of T_1 conflicts with the read(A) instruction of T_2 . However, the write(A) instruction of T_2 does not conflict with the read(B) instruction of T_1 , because the two instructions access different data items.

So, We continue to swap nonconflicting instructions:

- Swap the read(B) instruction of T_1 with the read(A) instruction of T_2 .
- Swap the write(B) instruction of T_1 with the write(A) instruction of T_2 .
- Swap the write(B) instruction of T_1 with the read(A) instruction of T_2 .

T_1	T_2
read(A)	
write(A)	
	read(A)
read(B)	
	write(A)
write(B)	
	read(B)
	write(B)

Schedule 5—schedule 3 after swapping of a pair of instructions.

The final result of these swaps, the following schedule is a serial schedule,

T_1	T_2
read(A)	
write(A)	
read(B)	
write(B)	
	read(A)
	write(A)
	read(B)
	write(B)

Schedule 6—a serial schedule that is equivalent to schedule 3.

If a schedule S can be transformed into a schedule S_1 by a series of swaps of nonconflicting instructions, we say that S and S_1 are **conflict equivalent**.

View Serializability

The schedules S and S_1 are said to be **view equivalent** if three conditions are met:

1. For each data item Q , if transaction T_i reads the initial value of Q in schedule S , then transaction T_i must, in schedule S_1 , also read the initial value of Q .
2. For each data item Q , if transaction T_i executes $\text{read}(Q)$ in schedule S , and if that value was produced by a $\text{write}(Q)$ operation executed by transaction T_j , then the $\text{read}(Q)$ operation of

transaction T_i must, in schedule S_2 , also read the value of Q that was produced by the same $\text{write}(Q)$ operation of transaction T_j .

3. For each data item Q , the transaction (if any) that performs the final $\text{write}(Q)$ operation in schedule S must perform the final $\text{write}(Q)$ operation in schedule S_2 .

Conditions 1 and 2 ensure that each transaction reads the same values in both schedules and, therefore, performs the same computation. Condition 3, coupled with conditions 1 and 2, ensures that both schedules result in the same final system state.

In the following schedule, transactions T_4 and T_6 perform $\text{write}(Q)$ operations without having performed a $\text{read}(Q)$ operation. Writes of this sort are called **blind writes**. Blind writes appear in any view-serializable schedule that is not conflict serializable.

T_3	T_4	T_6
$\text{read}(Q)$		
$\text{write}(Q)$	$\text{write}(Q)$	
		$\text{write}(Q)$

Schedule 9—a view-serializable schedule.

Lock-Based Protocols

One way to ensure serializability 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 T_i has obtained a **shared-mode lock** (denoted by S) on item Q , then T_i can read, but cannot write, Q .

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

	S	X
S	true	false
X	false	false

Lock-compatibility matrix

Starvation

Suppose a transaction T_2 has a shared-mode lock on a data item, and another transaction T_1 requests an exclusive-mode lock on the data item. Clearly, T_1 has to wait for T_2 to release the shared-mode lock. Meanwhile, a transaction T_3 may request a shared-mode lock on the same data item. The lock request is compatible with the lock granted to T_2 , so T_3 may be granted the shared-mode lock. At this point T_2 may release the lock, but still T_1 has to wait for T_3 to finish. But again, there may be a new transaction T_4 that requests a shared-mode lock on the same data item, and is granted the lock

before T_3 releases it. In fact, it is possible that there is a sequence of transactions that each requests a shared-mode lock on the data item, and each transaction releases the lock a short while after it is granted, but T_1 never gets the exclusive-mode lock on the data item. The transaction T_1 may never make progress, and is said to be **starved**.

We can avoid starvation of transactions by granting locks in the following manner: When a transaction T_i requests a lock on a data item Q in a particular mode M , the concurrency-control manager grants the lock provided that

1. There is no other transaction holding a lock on Q in a mode that conflicts with M .
2. There is no other transaction that is waiting for a lock on Q , and that made its lock request before T_i .

Thus, a lock request will never get blocked by a lock request that is made later.

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.

Another variant of two-phase locking is the **rigorous two-phase locking protocol**, which requires that all locks be held until the transaction commits. we can easily verify that, with rigorous two-phase locking, transactions can be serialized in the order in which they commit.

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, on the basis of read and write requests from the transaction:

- When a transaction T_i issues a $\text{read}(Q)$ operation, the system issues a lock-S(Q) instruction followed by the $\text{read}(Q)$ instruction.
- When T_i issues a $\text{write}(Q)$ operation, the system checks to see whether T_i already holds a shared lock on Q . If it does, then the system issues an upgrade(Q) instruction, followed by the $\text{write}(Q)$ instruction. Otherwise, the system issues a lock-X(Q) instruction, followed by the $\text{write}(Q)$ instruction.
- All locks obtained by a transaction are unlocked after that transaction commits or aborts.

Timestamps

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 of the counter when the transaction enters the system.

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 $\text{write}(Q)$ successfully.
- **R-timestamp(Q)** denotes the largest timestamp of any transaction that executed $\text{read}(Q)$ successfully.

These timestamps are updated whenever a new $\text{read}(Q)$ or $\text{write}(Q)$ instruction is executed.

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 T_i issues $\text{read}(Q)$.

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

b. If $\text{TS}(T_i) \geq \text{W-timestamp}(Q)$, then the read operation is executed, and $\text{Rtimestamp}(Q)$ is set to the maximum of $\text{R-timestamp}(Q)$ and $\text{TS}(T_i)$.

2. Suppose that transaction T_i issues $\text{write}(Q)$.

a. If $\text{TS}(T_i) < \text{R-timestamp}(Q)$, then the value of Q that T_i 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 T_i back.

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

c. Otherwise, the system executes the write operation and sets $\text{W-timestamp}(Q)$ to $\text{TS}(T_i)$.

The modification to the timestamp-ordering protocol, called **Thomas' write rule**,

Suppose that transaction T_i issues $\text{write}(Q)$.

1. If $TS(T_i) < R\text{-timestamp}(Q)$, then the value of Q that T_i 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 T_i back.
2. If $TS(T_i) < W\text{-timestamp}(Q)$, then T_i 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 $W\text{-timestamp}(Q)$ to $TS(T_i)$.

Deadlock

A system is in dead lock state if there exists a set of transactions such that every transaction in the set is waiting for another transaction in the set. More precisely, there exists a set of waiting transactions $\{T_0, T_1, T_2, \dots, T_n\}$ such that T_0 is waiting for a data item that T_1 holds, and T_1 is waiting for a data item that T_2 holds, T_n is waiting for a data item that T_0 holds. None of the transactions can make progress in such situation.

There are two principal methods to dealing with deadlock problem

- **Deadlock prevention** – To ensure that the system will never enter into the deadlock state.
- **Deadlock detection and recovery** – The system allowed to enter into the deadlock state and then try to recover the system using recovery schemes.

Dead lock Prevention

There are two approaches in deadlock prevention ,

- 1) Ensures that no cyclic waits can occur by ordering the requests for locks , or requiring all locks to be acquired together.
- 2) It is closer to the recovery scheme , and performs transaction roll back instead of waiting for a lock, whenever the wait could potentially result in dead lock.

Simple scheme for first approach is that each transaction can lock its data items before it starts its execution .

But there are two disadvantages in this scheme,

- 1) It is often hard to predict that what are the data item needs to be locked at the time of starting the execution.
- 2) Data item utilization may be low, some of the data items may be locked but unused for a long time.

The Second approach for preventing deadlocks is to use pre-emption and transaction rollbacks.

To control the preemption , we assign a unique timestamp to each transaction . The system uses the timestamp to decide whether a transaction is should wait or roll back. If a transaction is rolled back it retains its old timestamp when restarted.

Two dead lock prevention schemes using timestamps have been proposed.

- 1) Wait – Die : It is a non- preemptive technique
- 2) Wound – wait : It is a preemptive technique

Whenever the system rolls back transactions, it is important to ensure that there is no starvation that is , no transaction gets rolled back repeatedly and is never allowed to make progress.

Deadlock Detection and Recovery

An algorithm that examines the system state periodically to determine whether a deadlock has occurred or not. If the dead lock occurred in the system it must be recovered from the deadlock. To do so, the system must

- Maintain information about the current allocation of data as well as the requests of the transactions.
- Provide an algorithm that uses this information to determine whether the system has entered a deadlock state.

- Recover from the deadlock when the detection algorithm determines that the deadlock exists.

Deadlock detection

Deadlock can be described precisely in terms of a directed graph called a wait for graph. This graph consists of a pair $G=(V,E)$ where V is a set of vertices and E is a set of edges.

V - all transactions.

E – Ordered pair $\{ T_i \rightarrow T_j \}$ implying that the transaction T_i is waiting for transaction T_j to release a data item that it needs.

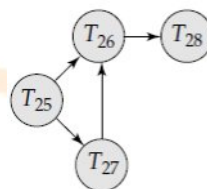
Wait for Graph

When transaction T_i requests a data item currently being held by transaction T_j , then the edge $T_i \rightarrow T_j$ is inserted in the wait-for graph. This edge is removed only when transaction T_j is no longer holding a data item needed by transaction T_i .

A deadlock exists in the system if and only if the wait-for graph contains a cycle. Each transaction involved in the cycle is said to be deadlocked. To detect deadlocks, the system needs to maintain the wait-for graph, and periodically to invoke an algorithm that searches for a cycle in the graph.

Consider the wait-for graph in which depicts the following situation:

- Transaction T_{25} is waiting for transactions T_{26} and T_{27} .
- Transaction T_{27} is waiting for transaction T_{26} .
- Transaction T_{26} is waiting for transaction T_{28} .

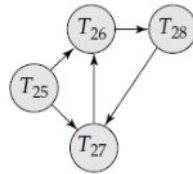


Wait-for graph with no cycle.

Since the graph has no cycle, the system is not in a deadlock state.

Suppose now that transaction T_{28} is requesting an item held by T_{27} . The edge $T_{28} \rightarrow T_{27}$ is added to the wait-for graph, resulting in the new system state in the following graph.

Now, the graph contains the cycle $T_{26} \rightarrow T_{28} \rightarrow T_{27} \rightarrow T_{26}$



Wait-for graph with a cycle.

implying that transactions T_{26} , T_{27} , and T_{28} are all deadlocked.

Consequently, the question arises: When should we invoke the detection algorithm?

The answer depends on two factors:

1. How often does a deadlock occur?
2. How many transactions will be affected by the deadlock?

If deadlocks occur frequently, then the detection algorithm should be invoked more frequently.

Recovery form deadlock

Three actions need to be taken ,

1. Selection of a victim. Given a set of deadlocked transactions, we must determine which transaction (or transactions) to roll back to break the deadlock. We should roll back those transactions that will incur the minimum cost. Unfortunately, the term *minimum cost* is not a precise one.

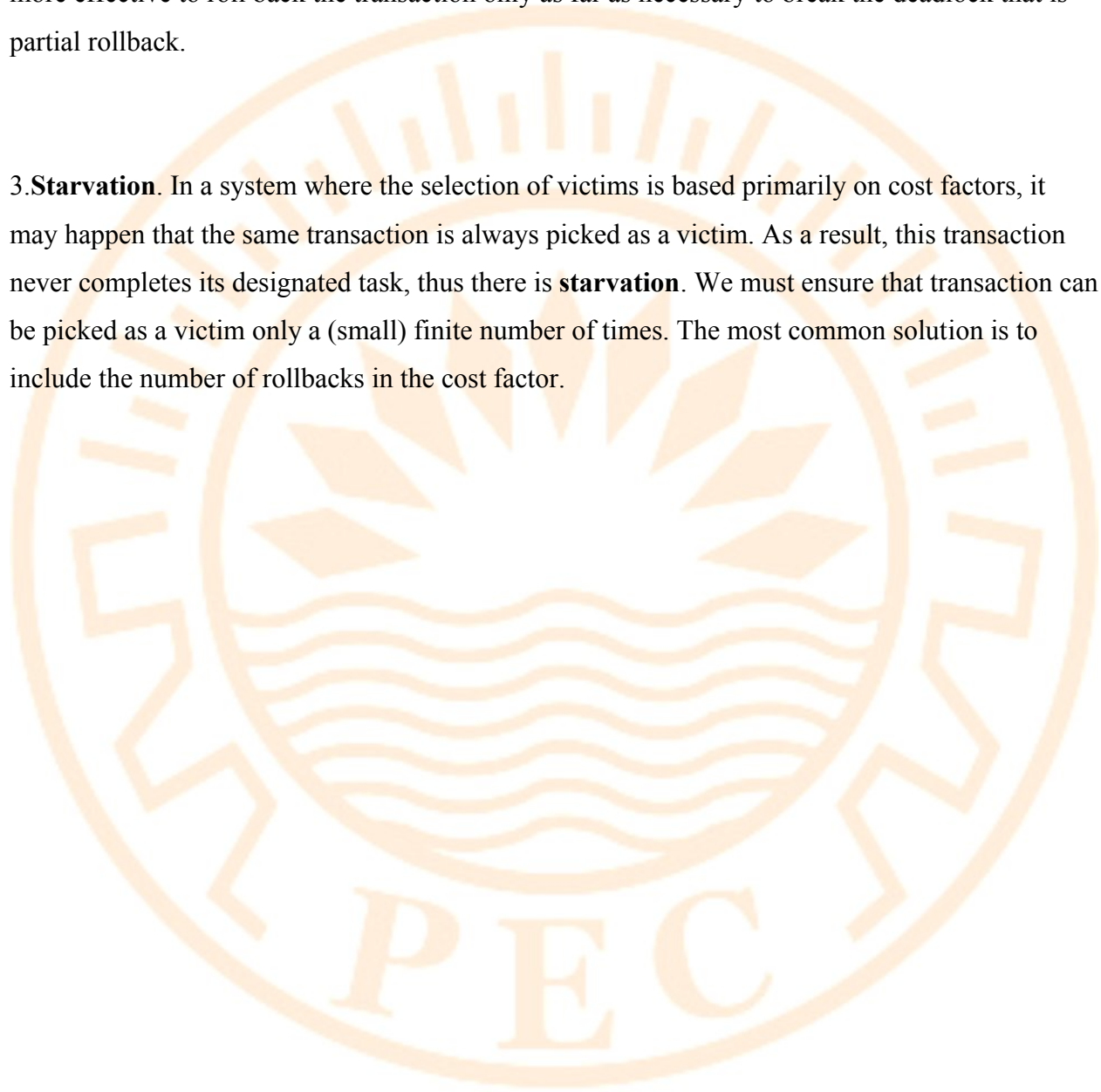
Many factors may determine the cost of a rollback, including

- a. How long the transaction has computed, and how much longer the transaction will compute before it completes its designated task.
- b. How many data items the transaction has used.
- c. How many more data items the transaction needs for it to complete.
- d. How many transactions will be involved in the rollback.

2. Rollback. Once we have decided that a particular transaction must be rolled back, we must determine how far this transaction should be rolled back.

The simplest solution is a **total rollback**: Abort the transaction and then restart it. However, it is more effective to roll back the transaction only as far as necessary to break the deadlock that is partial rollback.

3. Starvation. In a system where the selection of victims is based primarily on cost factors, it may happen that the same transaction is always picked as a victim. As a result, this transaction never completes its designated task, thus there is **starvation**. We must ensure that transaction can be picked as a victim only a (small) finite number of times. The most common solution is to include the number of rollbacks in the cost factor.



UNIT – IV

UNIT IV TRENDS IN DATABASE TECHNOLOGY 10

Overview of Physical Storage Media – Magnetic Disks – RAID – Tertiary storage – File Organization – Organization of Records in Files – Indexing and Hashing – Ordered Indices – B+ tree Index Files – B tree Index Files – Static Hashing – Dynamic Hashing - Introduction to Distributed Databases- Client server technology- Multidimensional and Parallel databases- Spatial and multimedia databases-Mobile and web databases- Data Warehouse-Mining- Data marts.

Overview of Physical Storage Media

Several types of data storage exist in most computer systems. These storage media are classified by the speed with which data can be accessed, by the cost per unit of data to buy the medium, and by the medium's reliability.

→ **Cache.** The cache is the fastest and most costly form of storage. Cache memory is small; its use is managed by the computer system hardware. We shall not be concerned about managing cache storage in the database system.

→ **Main memory.** The storage medium used for data that are available to be operated on is main memory. The general-purpose machine instructions operate on main memory. The contents of main memory are usually lost if a power failure or system crash occurs.

→ **Flash memory.** Also known as *electrically erasable programmable read-only memory (EEPROM)*, flash memory differs from main memory in that data survive power failure. Reading data from flash memory takes less than 100 nanoseconds (a nanosecond is 1/1000 of a microsecond), which is roughly as fast as reading data from main memory. Writing data to flash memory is more complicated—data can be written once, which takes about 4 to 10 microseconds, but cannot be overwritten directly.

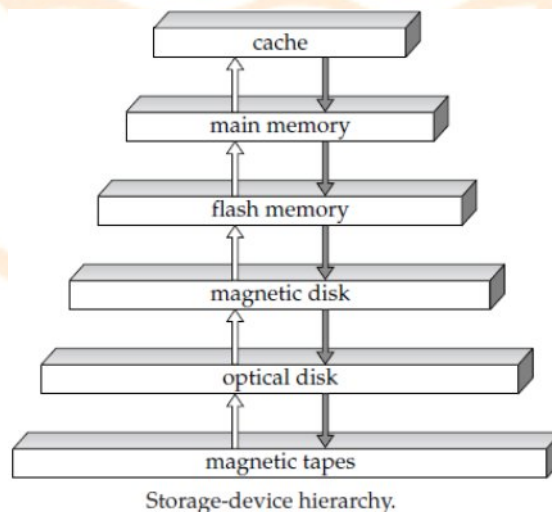
→ **Magnetic-disk storage.** The primary medium for the long-term on-line storage of data is the magnetic disk. Usually, the entire database is stored on magnetic disk. The system must move

the data from disk to main memory so that they can be accessed. After the system has performed the designated operations, the data that have been modified must be written to disk. The size of magnetic disks currently ranges from a few gigabytes to 80 gigabytes.

→**Optical storage.** The most popular forms of optical storage are the *compact disk* (CD), which can hold about 640 megabytes of data, and the *digital video disk* (DVD) which can hold 4.7 or 8.5 gigabytes of data per side of the disk. The optical disks used in read-only compact disks (CD-ROM) or read-only digital video disk (DVD-ROM) cannot be written, but are supplied with data prerecorded.

There are “record-once” versions of compact disk (called CD-R) and digital video disk (called DVD-R), which can be written only once; such disks are also called **write-once, read-many** (WORM) disks. There are also “multiple-write” versions of compact disk (called CD-RW) and digital video disk (DVD-RW and DVD-RAM), which can be written multiple times.

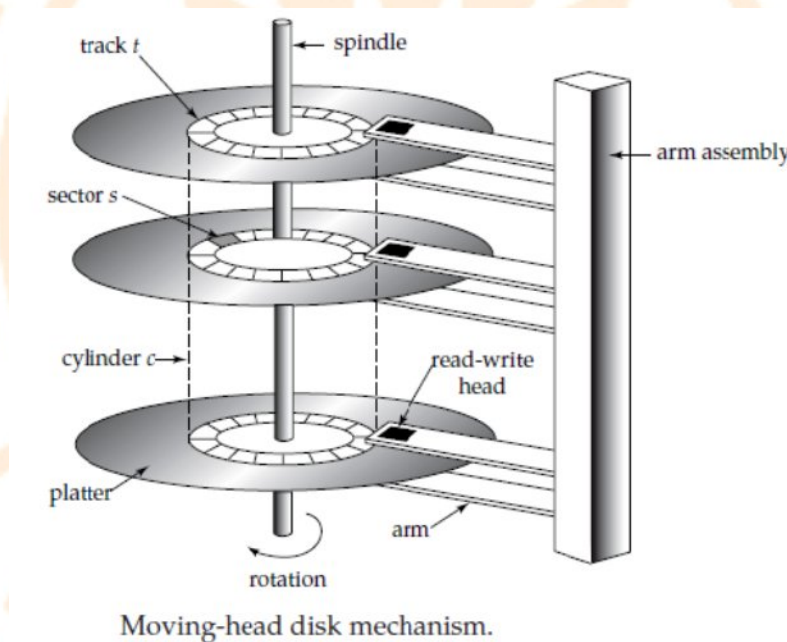
→**Tape storage.** Tape storage is used primarily for backup and archival data. Although magnetic tape is much cheaper than disks, access to data is much slower, because the tape must be accessed sequentially from the beginning. For this reason, tape storage is referred to as **sequential-access** storage. In contrast, disk storage is referred to as **direct-access** storage because it is possible to read data from any location on disk. Tapes have a high capacity (40 gigabyte to 300 gigabytes tapes are currently available)



The fastest storage media—for example, cache and main memory—are referred to as **primary storage**. The media in the next level in the hierarchy—for example, magnetic disks—are referred to as **secondary storage**, or **online storage**. The media in the lowest level in the hierarchy—for example, magnetic tape and optical-disk, jukeboxes—are referred to as **tertiary storage**, or **offline storage**.

Volatile storage loses its contents when the power to the device is removed. **Nonvolatile storage** for safekeeping of data.

Magnetic Disks



Each disk **platter** has a flat circular shape. Its two surfaces are covered with a magnetic material, and information is recorded on the surfaces. Platters are made from rigid metal or glass and are covered (usually on both sides) with magnetic recording material. We call such magnetic disks **hard disks**, to distinguish them from **floppy disks**, which are made from flexible material.

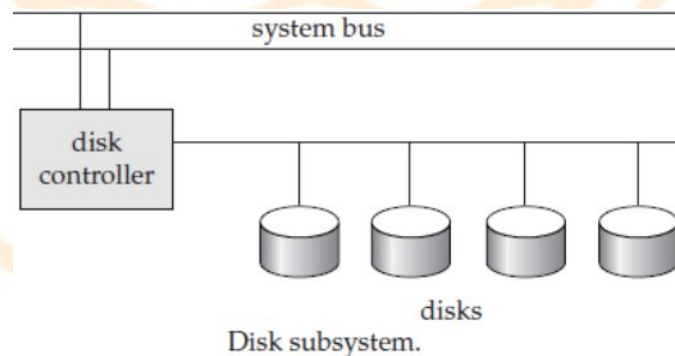
PRATHYUSHA ENGINEERING COLLEGE

The disk surface is logically divided into **tracks**, which are subdivided into **sectors**. A **sector** is the smallest unit of information that can be read from or written to the disk. In currently available disks, sector sizes are typically 512 bytes; there are over 16,000 tracks on each platter, and 2 to 4 platters per disk. The **read–write head** stores information on a sector magnetically as reversals of the direction of magnetization of the magnetic material.

A disk typically contains many platters, and the read –write heads of all the tracks are mounted on a single assembly called a **disk arm**.

Fixed-head disks and multiple-arm disks were used in high-performance mainframe systems, but are no longer in production. A *fixed-head disk* has a separate head for each track. This arrangement allows the computer to switch from track to track quickly, without having to move the head assembly, but because of the large number of heads, the device is extremely expensive.

A **disk controller** interfaces between the computer system and the actual hardware of the disk drive. Disk controllers also attach **checksums** to each sector that is written. the hecksum is computed from the data written to the sector. When the sector is read back, the controller computes the checksum again from the retrieved data and compares it with the stored checksum; if the data are corrupted, with a high probability the newly computed checksum will not match the stored checksum. If such an error occurs, the controller will retry the read several times; if the error continues to occur, the controller will signal a read failure. Another interesting task that disk controllers perform is **remapping of bad sectors**.



Performance Measures of Disks

The main measures of the qualities of a disk are capacity, access time, data-transfer rate, and reliability. **Access time** is the time from when a read or write request is issued to when data

transfer begins. To access (that is, to read or write) data on a given sector of a disk, the arm first must move so that it is positioned over the correct track, and then must wait for the sector to appear under it as the disk rotates. The time for repositioning the arm is called the **seek time**.

The **average seek time** is the average of the seek times, measured over a sequence of (uniformly distributed) random requests. Once the seek has started, the time spent waiting for the sector to be accessed to appear under the head is called the **rotational latency time**.

The **average latency time** of the disk is one-half the time for a full rotation of the disk. The access time is then the sum of the seek time and the latency, and ranges from 8 to 20 milliseconds.

The **data-transfer rate** is the rate at which data can be retrieved from or stored to the disk. Current disk systems claim to support maximum transfer rates of about 25 to 40 megabytes per second.

The final commonly used measure of a disk is the **mean time to failure (MTTF)**, which is a measure of the reliability of the disk. The mean time to failure of a disk (or of any other system) is the amount of time that, on average, we can expect the system to run continuously without any failure.

RAID

The data storage requirements of some applications are very large so, large number of disks are needed to store data for such applications. Having a large number of disks in a system presents opportunities for improving the rate at which data can be read or written, if the disks are operated in parallel.

A variety of disk-organization techniques, collectively called **redundant arrays of independent disks (RAID)**, have been proposed to achieve improved performance and reliability.

There are two main goals of parallelism in a disk system:

1. Load-balance multiple small accesses (block accesses), so that the throughput of such accesses increases.
2. Parallelize large accesses so that the response time of large accesses is reduced.

RAID Levels

Mirroring provides high reliability, but it is expensive. Striping provides high data transfer rates, but does not improve reliability.

Various other schemes provide redundancy at lower cost by combining disk striping with “parity” bits (which we describe next). These schemes have different cost–performance trade-offs.

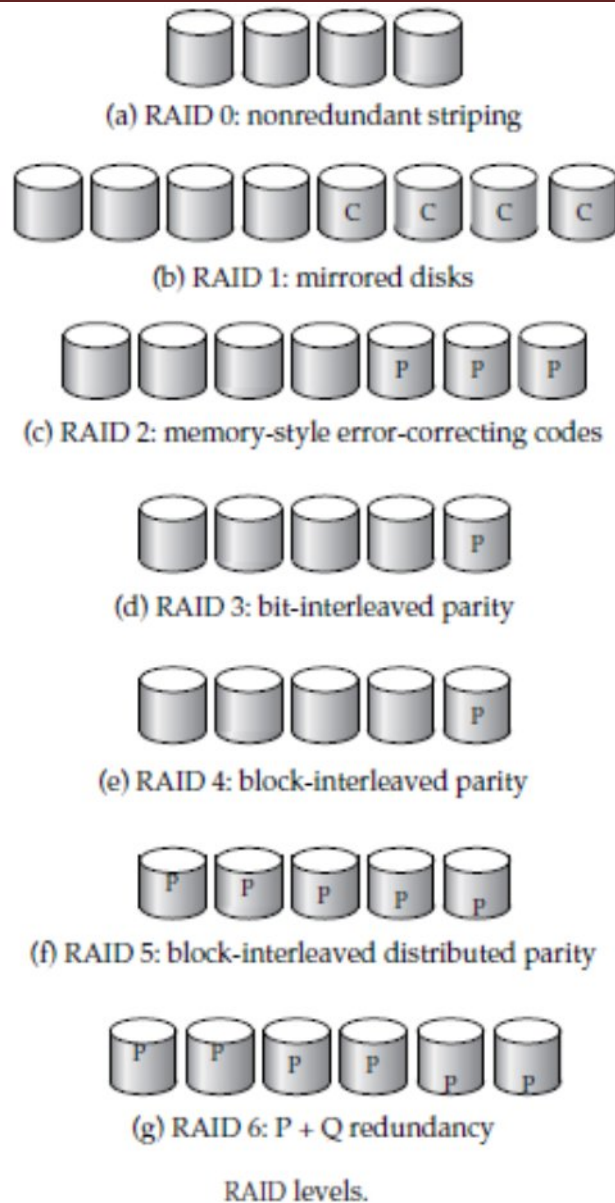
The schemes are classified into **RAID levels**.

RAID level 0 refers to disk arrays with striping at the level of blocks, but without any redundancy (such as mirroring or parity bits).

RAID level 1 refers to disk mirroring with block striping.

RAID level 2, known as memory-style error-correcting-code (ECC) organization, employs parity bits. Memory systems have long used parity bits for error detection and correction. Each byte in a memory system may have a parity bit associated with it that records whether the number of bits in the byte that are set to 1 is even (parity = 0) or odd (parity = 1). If one of the bits in the byte gets damaged (either a 1 becomes a 0, or a 0 becomes a 1), the parity of the byte changes and thus will not match the stored parity. Similarly, if the stored parity bit gets damaged, it will not match the computed parity. Thus, all 1-bit errors will be detected by the memory system.

The disks labeled P store the error correction bits. If one of the disks fails, the remaining bits of the byte and the associated error-correction bits can be read from other disks, and can be used to reconstruct the damaged data.



RAID level 3, bit-interleaved parity organization, improves on level 2 by exploiting the fact that disk controllers, unlike memory systems, can detect whether a sector has been read correctly, so a single parity bit can be used for error correction, as well as for detection.

The idea is as follows. If one of the sectors gets damaged, the system knows exactly which sector it is, and, for each bit in the sector, the system can figure out whether it is a 1 or a 0 by computing the parity of the corresponding bits from sectors in the other disks. If the parity of the remaining bits is equal to the stored parity, the missing bit is 0; otherwise, it is 1.

RAID level 3 is as good as level 2, but is less expensive in the number of extra disks (it has only a one-disk overhead).

RAID level 3 has two benefits over level 1. It needs only one parity disk for several regular disks, whereas Level 1 needs one mirror disk for every disk, and thus reduces the storage overhead.

RAID level 4, block-interleaved parity organization, uses block level striping, like RAID 0, and in addition keeps a parity block on a separate disk for corresponding blocks from N other disks. A block read accesses only one disk, allowing other requests to be processed by the other disks. Thus, the data-transfer rate for each access is slower, but multiple read accesses can proceed in parallel, leading to a higher overall I/O rate. The transfer rates for large reads is high, since all the disks can be read in parallel; large writes also have high transfer rates, since the data and parity can be written in parallel.

A single write requires four disk accesses: two to read the two old blocks, and two to write the two blocks.

RAID level 5, block-interleaved distributed parity, improves on level 4 by partitioning data and parity among all $N + 1$ disks, instead of storing data in N disks and parity in one disk. In level 5, all disks can participate in satisfying read requests, unlike RAID level 4, where the parity disk cannot participate, so level 5 increases the total number of requests that can be met in a given amount of time. For each set of N logical blocks, one of the disks stores the parity, and the other N disks store the blocks.

A parity block cannot store parity for blocks in the same disk, since then a disk failure would result in loss of data as well as of parity, and hence would not be recoverable. Level 5 subsumes level 4, since it offers better read–write performance at the same cost, so level 4 is not used in practice.

RAID level 6, the $P + Q$ redundancy scheme, is much like RAID level 5, but stores extra redundant information to guard against multiple disk failures. Instead of using parity, level 6 uses error-correcting codes such as the Reed–Solomon codes.

Choice of RAID Level

The factors to be taken into account when choosing a RAID level are

- Monetary cost of extra disk storage requirements
- Performance requirements in terms of number of I/O operations
- Performance when a disk has failed
- Performance during rebuild (that is, while the data in a failed disk is being rebuilt on a new disk)

RAID level 0 is used in high-performance applications where data safety is not critical.

Since RAID levels 2 and 4 are subsumed by RAID levels 3 and 5, the choice of RAID levels is restricted to the remaining levels.

RAID level 1 is popular for applications such as storage of log files in a database system, since it offers the best write performance. RAID level 5 has a lower storage overhead than level 1, but has a higher time overhead for writes.

RAID Level 6 is not supported currently by many RAID implementations, but it offers better reliability than level 5 and can be used in applications where data safety is very important. For applications where data are read frequently, and written rarely, level 5 is the preferred choice.

RAID level 1 is therefore the RAID level of choice for many applications with moderate storage requirements, and high I/O requirements.

RAID level 5, which increases the number of I/O operations needed to write a single logical block. If there are more disks in an array, data-transfer rates are higher, but the system would be more expensive. If there are more bits protected by a parity bit, the space overhead due to parity bits is lower, but there is an increased chance that a second disk will fail before the first failed disk is repaired, and that will result in data loss.

Tertiary Storage

In a large database system, some of the data may have to reside on tertiary storage. The two most common tertiary storage media are optical disks and magnetic tapes.

Optical Disks

Compact disks are a popular medium for distributing software, multimedia data such as audio and images, and other electronically published information. They have a fairly large

capacity (640 megabytes), and they are cheap to mass-produce. Digital video disks (DVDs) are replacing compact disks in applications that require very large amounts of data.

CD and DVD drives have much longer seek times (100 milliseconds is common) than do magnetic-disk drives, since the head assembly is heavier. Rotational speeds are typically lower than those of magnetic disks, although the faster CD and DVD drives have rotation speeds of about 3000 rotations per minute, which is comparable to speeds of lower-end magnetic-disk drives. Rotational speeds of CD drives originally corresponded to the audio CD standards, and the speeds of DVD drives originally corresponded to the DVD video standards, but current-generation drives rotate at many times the standard rate.

Data transfer rates are somewhat less than for magnetic disks. The record-once versions of optical disks (CD-R, and increasingly, DVD-R) are popular for distribution of data and particularly for archival storage of data because they have a high capacity, have a longer lifetime than magnetic disks, and can be removed and stored at a remote location. Since they cannot be overwritten, they can be used

to store information that should not be modified, such as audit trails. The multiple write versions (CD-RW, DVD-RW, and DVD-RAM) are also used for archival purposes.

Jukeboxes are devices that store a large number of optical disks (up to several hundred) and load them automatically on demand to one of a small number (usually, 1 to 10) of drives.

Magnetic Tapes

Although magnetic tapes are relatively permanent and can hold large volumes of data, they are slow in comparison to magnetic and optical disks. Even more important, magnetic tapes are limited to sequential access. Thus, they cannot provide random access for secondary-storage requirements, although historically, prior to the use of magnetic disks, tapes were used as a secondary-storage medium. Tapes are used mainly for backup, for storage of infrequently used information, and as an offline medium for transferring information from one system to another. Tapes are also used for storing large volumes of data, such as video or image data, that either do not need to be accessible quickly or are so voluminous that magneticdisk storage would be too expensive.

Tape devices are quite reliable, and good tape drive systems perform a read of the just-written data to ensure that it has been recorded correctly. Tapes, however, have limits on the number of times that they can be read or written reliably.

Tape jukeboxes, like optical disk jukeboxes, hold large numbers of tapes, with a few drives onto which the tapes can be mounted; they are used for storing large volumes of data, ranging up to many terabytes.

File Organization

A **file** is organized logically as a sequence of records. These records are mapped onto disk blocks. We need to consider ways of representing logical data models in terms of files.

One approach to mapping the database to files is to use several files, and to store records of only one fixed length in any given file. An alternative is to structure our files so that we can accommodate multiple lengths for records; however, files of fixed length records are easier to implement than are files of variable-length records. Many of the techniques used for the former can be applied to the variable-length case.

Fixed-Length Records

There are two problems with this simple approach:

1. It is difficult to delete a record from this structure. The space occupied by the record to be deleted must be filled with some other record of the file, or we must have a way of marking deleted records so that they can be ignored.

Unless the block size happens to be a multiple of 40 (which is unlikely), some records will cross block boundaries. That is, part of the record will be stored in one block and part in another. It would thus require two block accesses to read or write such a record.

record 0	A-102	Perryridge	400
record 1	A-305	Round Hill	350
record 3	A-101	Downtown	500
record 4	A-222	Redwood	700
record 5	A-201	Perryridge	900
record 6	A-217	Brighton	750
record 7	A-110	Downtown	600
record 8	A-218	Perryridge	700

File with record 2 deleted and all records moved.

When a record is deleted, we could move the record that came after it into the space formerly occupied by the deleted record, and so on, until every record following the deleted record has been moved ahead.

At the beginning of the file, we allocate a certain number of bytes as a **file header**. The header will contain a variety of information about the file. For now, all we need to store there is the address of the first record whose contents are deleted. We use this first record to store the address of the second available record, and so on. Intuitively, we can think of these stored addresses as *pointers*, since they point to the location of a record. The deleted records thus form a linked list, which is often referred to as a **free list**.

On insertion of a new record, we use the record pointed to by the header. We change the header pointer to point to the next available record. If no space is available, we add the new record to the end of the file.

Insertion and deletion for files of fixed-length records are simple to implement, because the space made available by a deleted record is exactly the space needed to insert a record. If we allow records of variable length in a file, this match no longer holds. An inserted record may not fit in the space left free by a deleted record, or it may fill only part of that space.

header				
record 0	A-102	Perryridge	400	
record 1				
record 2	A-215	Mianus	700	
record 3	A-101	Downtown	500	
record 4				
record 5	A-201	Perryridge	900	
record 6				
record 7	A-110	Downtown	600	
record 8	A-218	Perryridge	700	

File with free list after deletion of records 1, 4, and 6.

Variable-Length Records

Variable-length records arise in database systems in several ways:

- Storage of multiple record types in a file
- Record types that allow variable lengths for one or more fields
- Record types that allow repeating fields

The format of the record is

type *account-list* = **record**

branch-name : char (22);

account-info : **array** [1 ..∞] **of**

record;

account-number : char(10);

balance : real;

end

end

We define *account-info* as an array with an arbitrary number of elements. There is no limit on how large a record can be.

Byte-String Representation

A simple method for implementing variable-length records is to attach a special *endof-record* (⊥) symbol to the end of each record. We can then store each record as a string of consecutive bytes.

The byte-string representation as described in Figure 11.10 has some disadvantages:

- It is not easy to reuse space occupied formerly by a deleted record. Although techniques exist to manage insertion and deletion, they lead to a large number of small fragments of disk storage that are wasted.
- There is no space, in general, for records to grow longer. If a variable-length record becomes longer, it must be moved—movement is costly if pointers to the record are stored elsewhere in the database (e.g., in indices, or in other records), since the pointers must be located and updated. Thus, the basic byte-string representation described here is not usually used for implementing variable-length records.

0	Perryridge	A-102	400	A-201	900	A-218	700	⊥
1	Round Hill	A-305	350	⊥				
2	Mianus	A-215	700	⊥				
3	Downtown	A-101	500	A-110	600	⊥		
4	Redwood	A-222	700	⊥				
5	Brighton	A-217	750	⊥				

Byte-string representation of variable-length records.

A modified form of the byte-string representation, called the slotted-page structure, is commonly used for organizing records *within* a single block

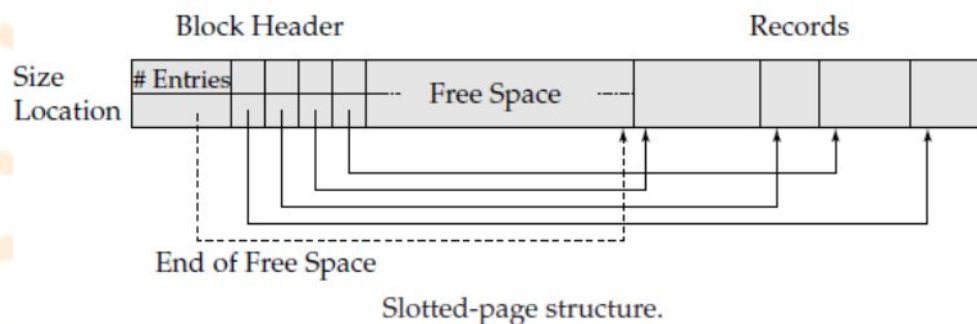
The **slotted-page structure** appears in the following figure. There is a header at the beginning of each block, containing the following information:

1. The number of record entries in the header
2. The end of free space in the block

3. An array whose entries contain the location and size of each record

The actual records are allocated *contiguously* in the block, starting from the end of the block. The free space in the block is contiguous, between the final entry in the header array, and the first record. If a record is inserted, space is allocated for it at the end of free space, and an entry containing its size and location is added to the header.

If a record is deleted, the space that it occupies is freed, and its entry is set to deleted (its size is set to -1 , for example). Further, the records in the block before the deleted record are moved, so that the free space created by the deletion gets occupied.



Fixed-Length Representation

Another way to implement variable-length records efficiently in a file system is to use one or more fixed-length records to represent one variable-length record.

There are two ways of doing this:

1. Reserved space. If there is a maximum record length that is never exceeded, we can use fixed-length records of that length. Unused space (for records shorter than the maximum space) is filled with a special null, or end-of-record, symbol.

2. List representation. We can represent variable-length records by lists of fixedlength records, chained together by pointers.

PRATHYUSHA ENGINEERING COLLEGE

The reserved-space method is useful when most records have a length close to the maximum. Otherwise, a significant amount of space may be wasted.

If we choose to apply the reserved-space method to our account example, we need to select a maximum record length. Figure shows how the file would be represented if we allowed a maximum of three accounts per branch.

0	Perryridge	A-102	400	A-201	900	A-218	700
1	Round Hill	A-305	350	⊥	⊥	⊥	⊥
2	Mianus	A-215	700	⊥	⊥	⊥	⊥
3	Downtown	A-101	500	A-110	600	⊥	⊥
4	Redwood	A-222	700	⊥	⊥	⊥	⊥
5	Brighton	A-217	750	⊥	⊥	⊥	⊥

File using the reserved-space method.

The reserved-space method is useful when most records have a length close to the maximum. Otherwise, a significant amount of space may be wasted.

In our example, some branches may have many more accounts than others. This situation leads us to consider the linked list method. To represent the file by the linked list method, we add a pointer field.

The resulting structure appears like the following figure,

0	Perryridge	A-102	400	
1	Round Hill	A-305	350	
2	Mianus	A-215	700	
3	Downtown	A-101	500	
4	Redwood	A-222	700	
5		A-201	900	
6	Brighton	A-217	750	
7		A-110	600	
8		A-218	700	



The diagram illustrates a linked list structure. Arrows from the pointer fields of records 0, 1, 2, 3, 4, 6, and 7 point to the start of records 5, 6, 7, 8, 5, 7, and 8 respectively. Record 5 is the first record for the branch A-201, and record 8 is the last record for the branch A-218.

File using linked lists.

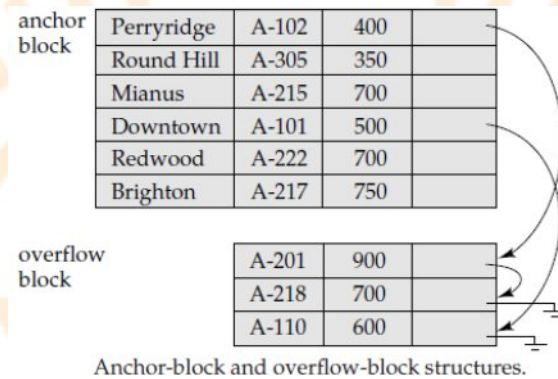
This wasted space is significant, since we expect, in practice, that each branch has a large number of accounts.

To deal with this problem, we allow two kinds of blocks in our file:

1.Anchor block, which contains the first record of a chain

2. Overflow block, which contains records other than those that are the first record of a chain.

Thus, all records *within a block* have the same length, even though not all records in the file have the same length.



Organization of Records in Files

Several of the possible ways of organizing records in files are:

- **Heap file organization.** Any record can be placed anywhere in the file where there is space for the record. There is no ordering of records. Typically, there is a single file for each relation.
- **Sequential file organization.** Records are stored in sequential order, according to the value of a “search key” of each record.
- **Hashing file organization.** A hash function is computed on some attribute of each record. The result of the hash function specifies in which block of the file the record should be placed.

A **sequential file** is designed for efficient processing of records in sorted order based on some search-key. A **search key** is any attribute or set of attributes; it need not be the primary key, or even a superkey. To permit fast retrieval of records in search-key order, we chain together records by pointers. The pointer in each record points to the next record in search-key order.

Furthermore, to minimize the number of block accesses in sequential file processing, we store records physically in search-key order, or as close to search-key order as possible.

A-217	Brighton	750	
A-101	Downtown	500	
A-110	Downtown	600	
A-215	Mianus	700	
A-102	Perryridge	400	
A-201	Perryridge	900	
A-218	Perryridge	700	
A-222	Redwood	700	
A-305	Round Hill	350	

Sequential file for *account* records.

Data Dictionary

A relational-database system needs to maintain data *about* the relations, such as the schema of the relations. This information is called the **data dictionary**, or **system catalog**.

Among the types of information that the system must store are these:

- Names of the relations
- Names of the attributes of each relation
- Domains and lengths of attributes
- Names of views defined on the database, and definitions of those views
- Integrity constraints (for example, key constraints)

Indexing and Hashing

Database system indices play the same role as book indices.

There are two basic kinds of indices:

- **Ordered indices.** Based on a sorted ordering of the values.
- **Hash indices.** Based on a uniform distribution of values across a range of buckets. The bucket to which a value is assigned is determined by a function, called a *hash function*.

Each technique must be evaluated on the basis of these factors:

- **Access types:** The types of access that are supported efficiently. Access types can include finding records with a specified attribute value and finding records whose attribute values fall in a specified range.
- **Access time:** The time it takes to find a particular data item, or set of items, using the technique in question.
- **Insertion time:** The time it takes to insert a new data item. This value includes the time it takes to find the correct place to insert the new data item, as well as the time it takes to update the index structure.
- **Deletion time:** The time it takes to delete a data item. This value includes the time it takes to find the item to be deleted, as well as the time it takes to update the index structure.
- **Space overhead:** The additional space occupied by an index structure. Provided that the amount of additional space is moderate, it is usually worthwhile to sacrifice the space to achieve improved performance.

An attribute or set of attributes used to look up records in a file is called a **search key**.

Ordered Indices

To gain fast random access to records in a file, we can use an index structure. Each index structure is associated with a particular search key. The records in the indexed file may themselves be stored in some sorted order.

Primary Index

A file may have several indices, on different search keys. If the file containing the records is sequentially ordered, a **primary index** is an index whose search key also defines the sequential order of the file. Primary indices are also called **clustering indices**.

Such files, with a primary index on the search key, are called **index-sequential files**.

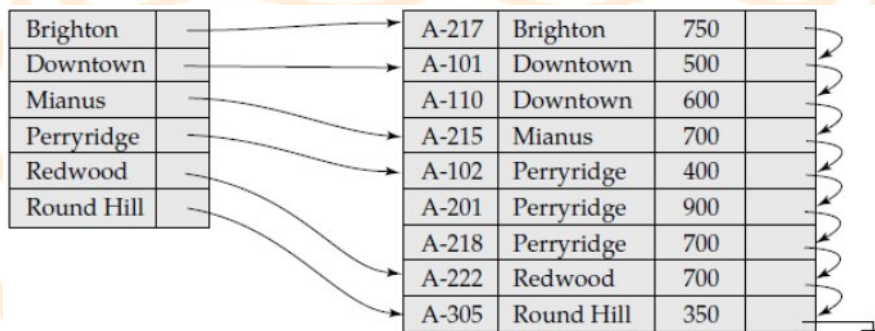
Dense and Sparse Indices

There are two types of ordered indices that we can use:

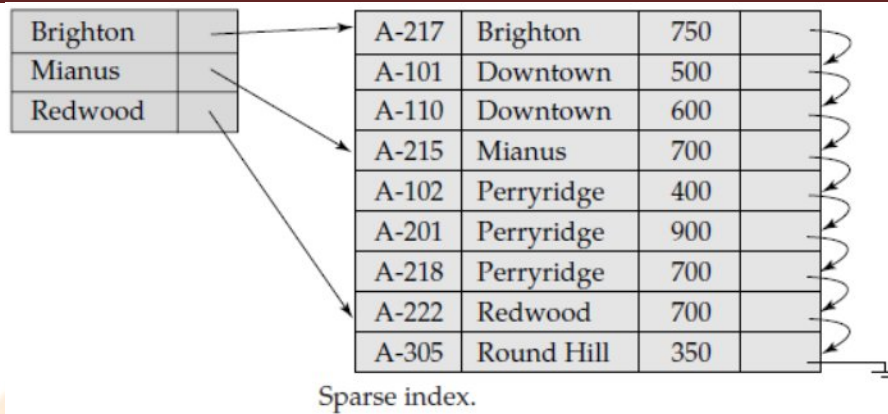
- **Dense index:** An index record appears for every search-key value in the file. In a dense primary index, the index record contains the search-key value and a pointer to the first data record with that search-key value. The rest of the records with the same search key-value would be stored sequentially after the first record, since, because the index is a primary one, records are sorted on the same search key.

Dense index implementations may store a list of pointers to all records with the same search-key value; doing so is not essential for primary indices.

- **Sparse index:** An index record appears for only some of the search-key values. As is true in dense indices, each index record contains a search-key value and a pointer to the first data record with that search-key value. To locate a record, we find the index entry with the largest search-key value that is less than or equal to the search-key value for which we are looking. We start at the record pointed to by that index entry, and follow the pointers in the file until we find the desired record.



Dense index.



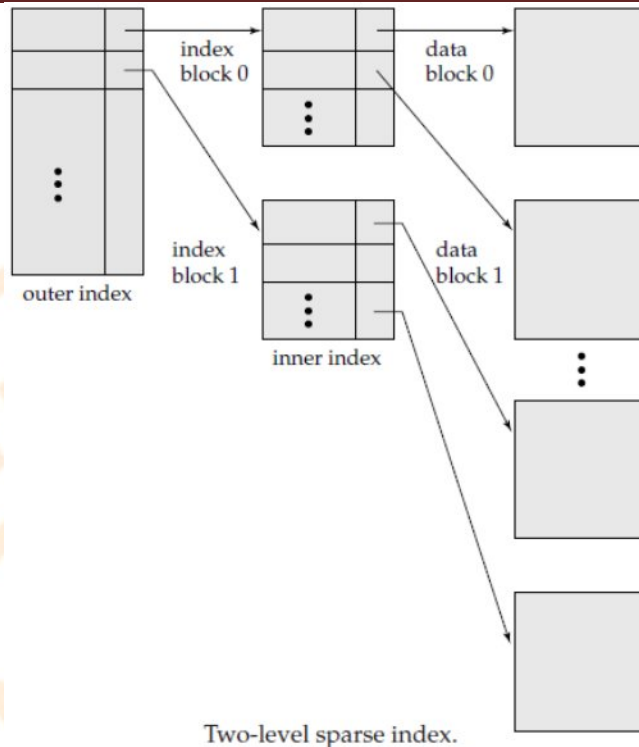
Multilevel Indices

Even if we use a sparse index, the index itself may become too large for efficient processing. It is not unreasonable, in practice, to have a file with 100,000 records, with 10 records stored in each block. If we have one index record per block, the index has 10,000 records. Index records are smaller than data records, so let us assume that 100 index records fit on a block. Thus, our index occupies 100 blocks. Such large indices are stored as sequential files on disk.

If an index is sufficiently small to be kept in main memory, the search time to find an entry is low. However, if the index is so large that it must be kept on disk, a search for an entry requires several disk block reads. Binary search can be used on the index file to locate an entry, but the search still has a large cost.

The process of searching a large index may be costly. To deal with this problem, we treat the index just as we would treat any other sequential file, and construct a sparse index on the primary index.

Assume that the outer index is already in main memory. If our file is extremely large, even the outer index may grow too large to fit in main memory. In such a case, we can create yet another level of index. Indeed, we can repeat this process as many times as necessary. Indices with two or more levels are called **multilevel** indices. Searching for records with a multilevel index requires significantly fewer I/O operations than does searching for records by binary search. Each level of index could correspond to a unit of physical storage. Thus, we may have indices at the track, cylinder, and disk levels. A typical dictionary is an example of a multilevel index.



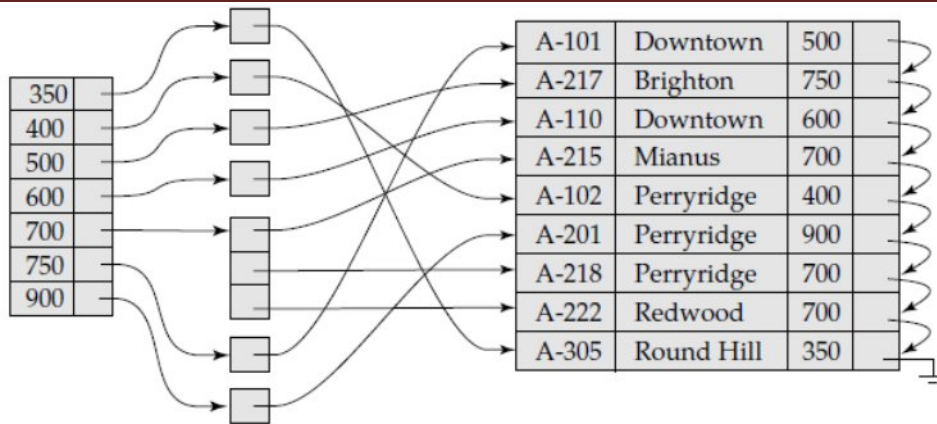
Secondary Indices

Secondary indices must be dense, with an index entry for every search-key value, and a pointer to every record in the file. A primary index may be sparse, storing only some of the search-key values, since it is always possible to find records with intermediate search-key values by a sequential access to a part of the file, as described earlier. If a secondary index stores only some of the search-key values, records with intermediate search-key values may be anywhere in the file and, in general, we cannot find them without searching the entire file.

A secondary index on a candidate key looks just like a dense primary index. We can use an extra level of indirection to implement secondary indices on search keys that are not candidate keys. The pointers in such a secondary index do not point directly to the file. Instead, each points to a bucket that contains pointers to the file.

A sequential scan in primary index order is efficient because records in the file are stored physically in the same order as the index order.

Secondary indices improve the performance of queries that use keys other than the search key of the primary index.



Secondary index on *account* file, on noncandidate key *balance*.

B+-Tree Index Files

The main disadvantage of the index-sequential file organization is that performance degrades as the file grows, both for index lookups and for sequential scans through the data. Although this degradation can be remedied by reorganization of the file, frequent reorganizations are undesirable.

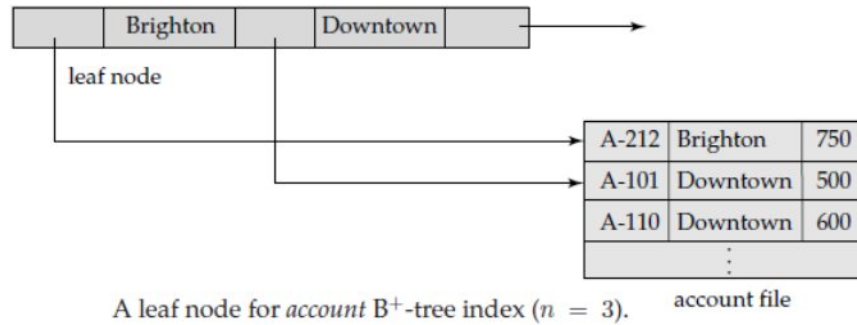
The **B+-tree** index structure is the most widely used of several index structures that maintain their efficiency despite insertion and deletion of data. A B+-tree index takes the form of a **balanced tree** in which every path from the root of the tree to a leaf of the tree is of the same length. Each nonleaf node in the tree has between $n/2$ and n children, where n is fixed for a particular tree.



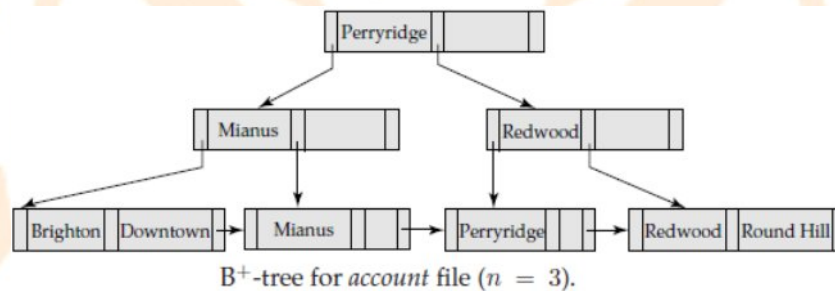
Typical node of a B⁺-tree.

Structure of a B+ Tree

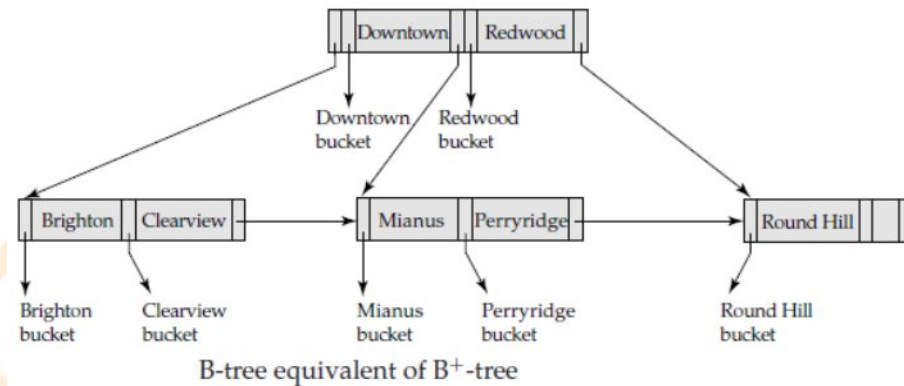
A B+-tree index is a multilevel index. The nonleaf nodes of the B+-tree form a multilevel (sparse) index on the leaf nodes. The structure of nonleaf nodes is the same as that for leaf nodes, except that all pointers are pointers to tree nodes. A nonleaf node may hold up to n pointers, and *must* hold at least $\lfloor n/2 \rfloor$ pointers. The number of pointers in a node is called the *fanout* of the node.



Let us consider a node containing m pointers. For $i = 2, 3, \dots, m - 1$, pointer P_i points to the subtree that contains search-key values less than K_i and greater than or equal to K_{i-1} . Pointer P_m points to the part of the subtree that contains those key values greater than or equal to K_{m-1} , and pointer P_1 points to the part of the subtree that contains those search-key values less than K_1 . Unlike other nonleaf nodes, the root node can hold fewer than $\lfloor n/2 \rfloor$ pointers.



the generalized B-tree in the figure, there are $n - 1$ keys in the leaf node, but there are $m - 1$ keys in the nonleaf node.



(a)



(b)

Typical nodes of a B-tree. (a) Leaf node. (b) Nonleaf node.

The number of nodes accessed in a lookup in a B-tree depends on where the search key is located. A lookup on a B⁺-tree requires traversal of a path from the root of the tree to some leaf node. In contrast, it is sometimes possible to find the desired value in a B-tree before reaching a leaf node.

B-tree has a smaller fanout and therefore may have depth greater than that of the corresponding B⁺-tree.

Deletion in a B-tree is more complicated. In a B⁺-tree, the deleted entry always appears in a leaf. In a B-tree, the deleted entry may appear in a nonleaf node. The insertion in a B-tree is only slightly more complicated than is insertion in a B⁺-tree.

The space advantages of B-trees are marginal for large indices, and usually do not outweigh the disadvantages that we have noted. Thus, many database system implementers prefer the structural simplicity of a B⁺-tree.

Static Hashing

One disadvantage of sequential file organization is that we must access an index structure to locate data, or must use binary search, and that results in more I/O operations. File organizations based on the technique of **hashing** allow us to avoid accessing an index structure.

Hash File Organization

In a **hash file organization**, we obtain the address of the disk block containing a desired record directly by computing a function on the search-key value of the record.

In hashing, the term **bucket** denotes a unit of storage that can store one or more records. A bucket is typically a disk block, but could be chosen to be smaller or larger than a disk block.

Formally, let K denote the set of all search-key values, and let B denote the set of all bucket addresses. A **hash function** h is a function from K to B . Let h denote a hash function.

To insert a record with search key K_i , we compute $h(K_i)$, which gives the address of the bucket for that record.

To perform a lookup on a search-key value K_i , we simply compute $h(K_i)$, then search the bucket with that address.

For deletion, if the search-key value of the record to be deleted is K_i , we compute $h(K_i)$, then search the corresponding bucket for that record, and delete the record from the bucket.

Hash Functions

To choose a hash function that assigns search-key values to buckets in such a way that the distribution has these qualities:

- The distribution is *uniform*. That is, the hash function assigns each bucket the same number of search-key values from the set of *all* possible search-key values.
- The distribution is *random*. That is, in the average case, each bucket will have nearly the same number of values assigned to it, regardless of the actual distribution of search-key values.

More precisely, the hash value will not be correlated to any externally visible ordering on the search-key values, such as alphabetic ordering or ordering by the length of the search keys; the hash function will appear to be random.

Hash functions require careful design. A bad hash function may result in lookup taking time proportional to the number of search keys in the file. A well-designed function gives an average-case lookup time that is a (small) constant, independent of the number of search keys in the file.

Handling of Bucket Overflows

When a record is inserted, the bucket to which it is mapped has space to store the record. If the bucket does not have enough space, a **bucket overflow** is said to occur. Bucket overflow can occur for several reasons:

- **Insufficient buckets.** The number of buckets, which we denote nB , must be chosen such that $nB > nr/fr$, where nr denotes the total number of records that will be stored, and fr denotes the number of records that will fit in a bucket. This designation, of course, assumes that the total number of records is known when the hash function is chosen.
- **Skew.** Some buckets are assigned more records than are others, so a bucket may overflow even when other buckets still have space. This situation is called bucket **skew**. Skew can occur for two reasons:

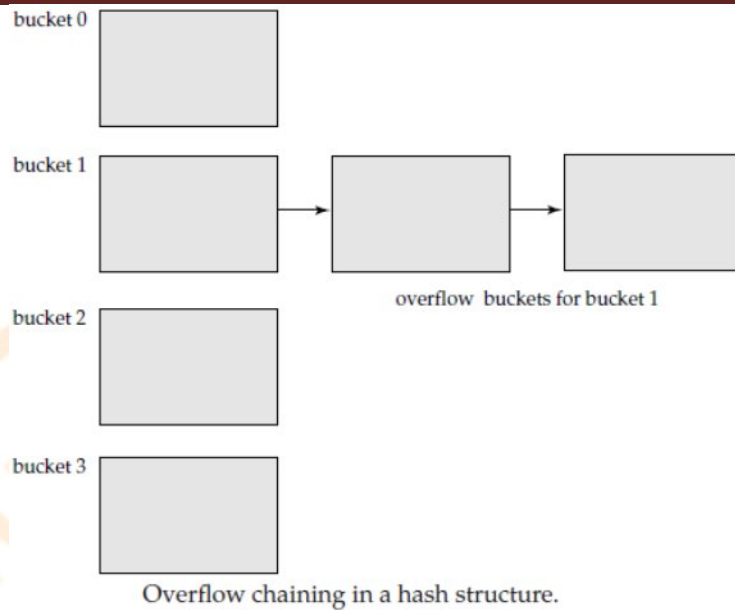
1. Multiple records may have the same search key.

2. The chosen hash function may result in nonuniform distribution of search keys.

So that the probability of bucket overflow is reduced, the number of buckets are chosen to be $(nr/fr) * (1 + d)$, where d is a fudge factor, typically around 0.2. Some space is wasted: About 20 percent of the space in the buckets will be empty. But the benefit is that the probability of overflow is reduced.

To handle the bucket overflow, we have two approaches such as

- i) **Open Hashing (Linear Probing)**
- ii) **Closed Hashing (Overflow Chaining)**



If a record must be inserted into a bucket b , and b is already full, the system provides an overflow bucket for b , and inserts the record into the overflow bucket. If the overflow bucket is also full, the system provides another overflow bucket, and so on. All the overflow buckets of a given bucket are chained together in a linked list, as in Figure. Overflow handling using such a linked list is called **overflow chaining**. It is also called as **closed hashing**.

An alternative approach, called **open hashing**, the set of buckets is fixed, and there are no overflow chains. Instead, if a bucket is full, the system inserts records in some other bucket in the initial set of buckets B . One policy is to use the next bucket (in cyclic order) that has space; this policy is called *linear probing*.

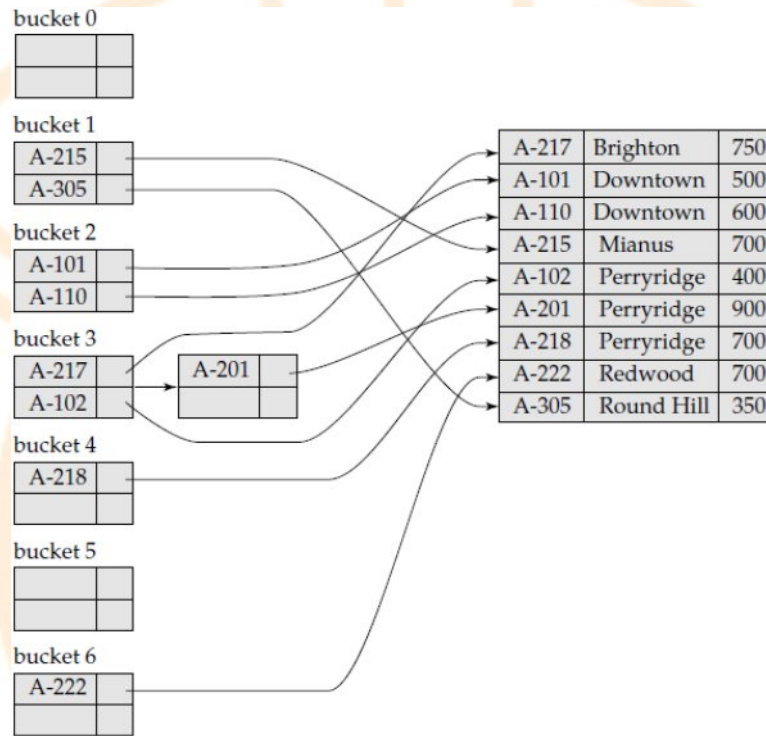
Open hashing has been used to construct symbol tables for compilers and assemblers, but closed hashing is preferable for database systems.

An important drawback to the form of hashing that we have described is that we must choose the hash function when we implement the system, and it cannot be changed easily thereafter if the file being indexed grows or shrinks. Since the function h maps search-key values to a fixed set B of bucket addresses.

Hash Indices

PRATHYUSHA ENGINEERING COLLEGE

A **hash index** organizes the search keys, with their associated pointers, into a hash file structure. We construct a hash index as follows. We apply a hash function on a search key to identify a bucket, and store the key and its associated pointers in the bucket (or in overflow buckets). Figure shows a secondary hash index on the *account* file, for the search key *account-number*.



Hash index on search key *account-number* of *account* file.

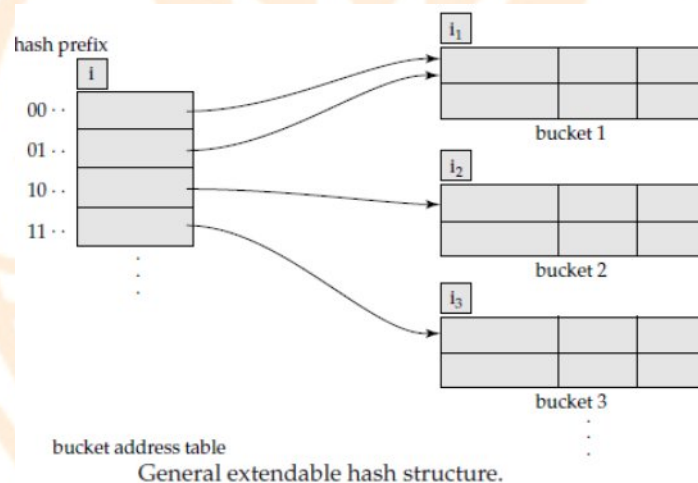
The hash function in the figure computes the sum of the digits of the account number modulo 7. The hash index has seven buckets, each of size 2 (realistic indices would, of course, have much larger bucket sizes). One of the buckets has three keys mapped to it, so it has an overflow bucket. In this example, *account-number* is a primary key for *account*, so each searchkey has only one associated pointer.

Dynamic Hashing

Most databases grow larger over time. If we are to use static hashing for such a database, we have three classes of options:

1. Choose a hash function based on the current file size. This option will result in performance degradation as the database grows.
2. Choose a hash function based on the anticipated size of the file at some point in the future. Although performance degradation is avoided, a significant amount of space may be wasted initially.
3. Periodically reorganize the hash structure in response to file growth. Such a reorganization involves choosing a new hash function, recomputing the hash function on every record in the file, and generating new bucket assignments.

Several **dynamic hashing** techniques allow the hash function to be modified dynamically to accommodate the growth or shrinkage of the database. Dynamic hashing is also called as **extendable hashing**.



We do not create a bucket for each hash value. We create buckets on demand, as records are inserted into the file. We do not use the entire b bits of the hash value initially. At any point, we use i bits, where $0 \leq i \leq b$. These i bits are used as an offset into an additional table of bucket addresses. The value of i grows and shrinks with the size of the database.

Homogeneous and Heterogeneous Databases

In a **homogeneous distributed database**, all sites have identical database management system software, are aware of one another, and agree to cooperate in processing users' requests.

In a **heterogeneous distributed database**, different sites may use different schemas, and different database management system software. The sites may

not be aware of one another, and they may provide only limited facilities for cooperation in transaction processing. The differences in schemas are often a major problem for query processing

Distributed Data Storage

Consider a relation r that is to be stored in the database. There are two approaches to storing this relation in the distributed database:

- **Replication.** The system maintains several identical replicas (copies) of the relation, and stores each replica at a different site. The alternative to replication is to store only one copy of relation r .
- **Fragmentation.** The system partitions the relation into several fragments, and stores each fragment at a different site.

Data Replication

There are a number of advantages and disadvantages to replication.

- **Availability.** If one of the sites containing relation r fails, then the relation r can be found in another site. Thus, the system can continue to process queries involving r , despite the failure of one site.
- **Increased parallelism.** In the case where the majority of accesses to the relation r result in only the reading of the relation, then several sites can process queries involving r in parallel.
- **Increased overhead on update.** The system must ensure that all replicas of a relation r are consistent; otherwise, erroneous computations may result. Thus, whenever r is updated, the update must be propagated to all sites containing replicas.

Data Fragmentation

There are two different schemes for fragmenting a relation:

horizontal fragmentation

vertical fragmentation.

Horizontal fragmentation splits the relation by assigning each tuple of r to one or more fragments.

In **horizontal fragmentation**, a relation r is partitioned into a number of subsets, r_1, r_2, \dots, r_n .

If the banking system has only two branches—Hillside and Valleyview—then there are two different fragments:

$$\begin{aligned} \text{account1} &= \sigma_{\text{branch-name} = \text{"Hillside"}}(\text{account}) \\ \text{account2} &= \sigma_{\text{branch-name} = \text{"Valleyview"}}(\text{account}) \end{aligned}$$

Horizontal fragmentation is usually used to keep tuples at the sites where they are used the most, to minimize data transfer.

We reconstruct the relation r by taking the union of all fragments; that is,

$$r = r_1 \cup r_2 \cup \dots \cup r_n$$

Vertical fragmentation splits the relation by decomposing the scheme R of relation r .

Vertical fragmentation of $r(R)$ involves the definition of several subsets of attributes R_1, R_2, \dots, R_n of the schema R so that

$$R = R_1 \cup R_2 \cup \dots \cup R_n$$

Each fragment r_i of r is defined by $r_i = \Pi_{R_i}(r)$

The fragmentation should be done in such a way that we can reconstruct relation r from the fragments by taking the natural join $r = r_1 \bowtie r_2 \bowtie r_3 \bowtie \dots \bowtie r_n$

One way of ensuring that the relation r can be reconstructed is to include the primary-key attributes of R in each of the R_i .

Distributed Transactions

The Distributed transactions must preserve the ACID properties. There are two types of transaction that we need to consider. The **local transactions** are those that access and update

data in only one local database; the **global transactions** are those that access and update data in several local databases.

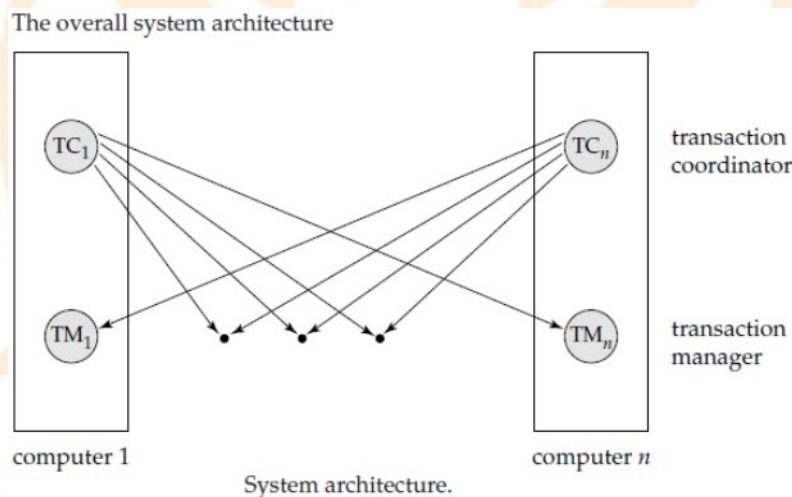
Each site has its own *local* transaction manager, whose function is to ensure the ACID properties of those transactions that execute at that site. The various transaction managers cooperate to execute global transactions.

each site contains two subsystems:

- The **transaction manager** manages the execution of those transactions (or subtransactions)

that access data stored in a local site. Note that each such transaction may be either a local transaction (that is, a transaction that executes at only that site) or part of a global transaction (that is, a transaction that executes at several sites).

- The **transaction coordinator** coordinates the execution of the various transactions (both local and global) initiated at that site.



Each transaction manager is responsible for

- Maintaining a log for recovery purposes.
- Participating in an appropriate concurrency-control scheme to coordinate the concurrent execution of the transactions executing at that site.

The transaction coordinator is responsible for

- Starting the execution of the transaction.

- Breaking the transaction into a number of subtransactions and distributing these subtransactions to the appropriate sites for execution.
- Coordinating the termination of the transaction, which may result in the transaction being committed at all sites or aborted at all sites.

System Failure Modes

The basic failure types in distributed database system is

- Failure of a site.
- Loss of messages.
- Failure of a communication link.
- Network partition.

Commit Protocols

If we are to ensure atomicity, all the sites in which a transaction T executed must agree on the final outcome of the execution. T must either commit at all sites, or it must abort at all sites. To ensure this property, the transaction coordinator of T must execute a *commit protocol*.

Two-Phase Commit

When T completes its execution—that is, when all the sites at which T has executed inform C_i that T has completed— C_i starts the 2PC protocol.

Phase 1.

- C_i adds the record $\langle \text{prepare } T \rangle$ to the log, and forces the log onto stable storage.
- It then sends a prepare T message to all sites. On receiving such a message, the transaction manager determines whether it is willing to commit its portion of T .
- If the answer is no, it adds a record $\langle \text{no } T \rangle$ to the log, and then responds by sending an abort T message to C_i .
- If the answer is yes, it adds a record $\langle \text{ready } T \rangle$ to the log, and forces the log onto stable storage.

- Then, The transaction manager replies with a ready T message to C_i .

Phase 2.

- When C_i receives responses to the prepare T message from all the sites prespecified interval of time has elapsed since the prepare T message was sent out, C_i can determine whether the transaction T can be committed or aborted.
- Transaction T can be committed if C_i received a ready T message from all the participating sites.
- Otherwise, transaction T must be aborted.
- Depending on that, either a record $\langle \text{commit } T \rangle$ or a record $\langle \text{abort } T \rangle$ is added to the log and the log is forced onto stable storage.
- Following this point, the coordinator sends either a commit T or an abort T message to all participating sites. When a site receives that message, it records the message in the log.

Distributed Database System

Homogeneous and Heterogeneous Databases

In a **homogeneous distributed database**, all sites have identical database management system software, are aware of one another, and agree to cooperate in processing users' requests.

In a **heterogeneous distributed database**, different sites may use different schemas, and different database management system software. The sites may not be aware of one another, and they may provide only limited facilities for cooperation in transaction processing. The differences in schemas are often a major problem for query processing

Distributed Data Storage

Consider a relation r that is to be stored in the database. There are two approaches to storing this relation in the distributed database:

- **Replication.** The system maintains several identical replicas (copies) of the relation, and stores each replica at a different site. The alternative to replication is to store only one copy of relation r .
- **Fragmentation.** The system partitions the relation into several fragments, and stores each fragment at a different site.

Data Replication

There are a number of advantages and disadvantages to replication.

- **Availability.** If one of the sites containing relation r fails, then the relation r can be found in another site. Thus, the system can continue to process queries involving r , despite the failure of one site.
- **Increased parallelism.** In the case where the majority of accesses to the relation r result in only the reading of the relation, then several sites can process queries involving r in parallel.
- **Increased overhead on update.** The system must ensure that all replicas of a relation r are consistent; otherwise, erroneous computations may result. Thus, whenever r is updated, the update must be propagated to all sites containing replicas.

Data Fragmentation

There are two different schemes for fragmenting a relation:

horizontal fragmentation

vertical fragmentation.

Horizontal fragmentation splits the relation by assigning each tuple of r to one or more fragments.

In **horizontal fragmentation**, a relation r is partitioned into a number of subsets, r_1, r_2, \dots, r_n .

If the banking system has only two branches—Hillside and Valleyview—then there are two different fragments:

$$account1 = \sigma_{branch-name = \text{"Hillside"}}(account)$$

$$account2 = \sigma_{branch-name = \text{"Valleyview"}}(account)$$

Horizontal fragmentation is usually used to keep tuples at the sites where they are used the most, to minimize data transfer.

We reconstruct the relation r by taking the union of all fragments; that is,

$$r = r_1 \cup r_2 \cup \dots \cup r_n$$

Vertical fragmentation splits the relation by decomposing the scheme R of relation r .

Vertical fragmentation of $r(R)$ involves the definition of several subsets of attributes R_1, R_2, \dots, R_n of the schema R so that

$$R = R_1 \cup R_2 \cup \dots \cup R_n$$

Each fragment r_i of r is defined by $r_i = \Pi_{R_i}(r)$

The fragmentation should be done in such a way that we can reconstruct relation r from the fragments by taking the natural join $r = r_1 \bowtie r_2 \bowtie r_3 \bowtie \dots \bowtie r_n$

One way of ensuring that the relation r can be reconstructed is to include the primary-key attributes of R in each of the R_i .

Distributed Transactions

The Distributed transactions must preserve the ACID properties. There are two types of transaction that we need to consider. The **local transactions** are those that access and update data in only one local database; the **global transactions** are those that access and update data in several local databases.

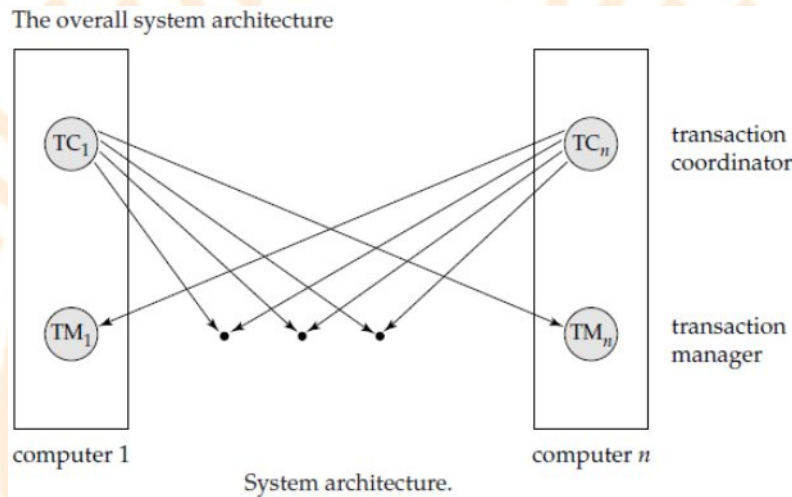
Each site has its own *local* transaction manager, whose function is to ensure the ACID properties of those transactions that execute at that site. The various transaction managers cooperate to execute global transactions.

each site contains two subsystems:

- The **transaction manager** manages the execution of those transactions (or subtransactions)

that access data stored in a local site. Note that each such transaction may be either a local transaction (that is, a transaction that executes at only that site) or part of a global transaction (that is, a transaction that executes at several sites).

- The **transaction coordinator** coordinates the execution of the various transactions (both local and global) initiated at that site.



Each transaction manager is responsible for

- Maintaining a log for recovery purposes.
- Participating in an appropriate concurrency-control scheme to coordinate the concurrent execution of the transactions executing at that site.

The transaction coordinator is responsible for

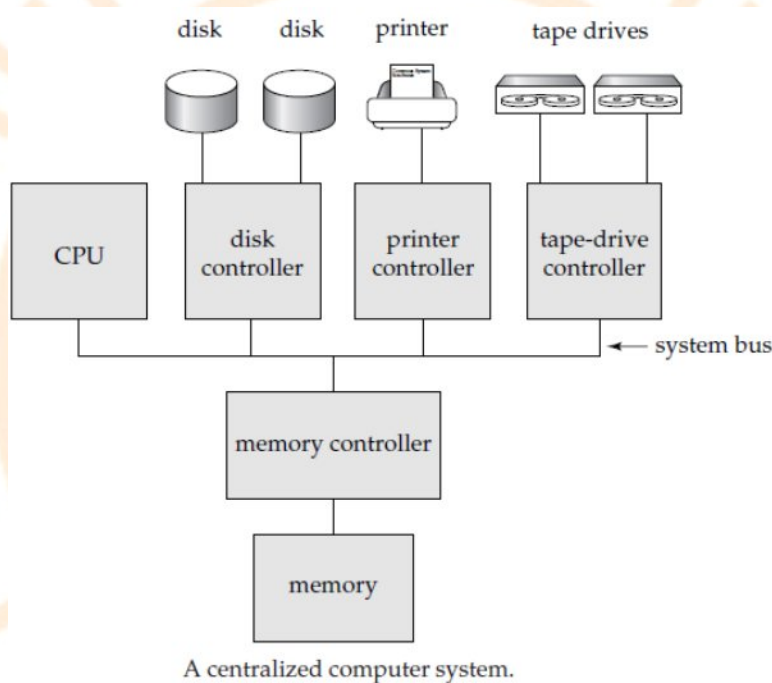
- Starting the execution of the transaction.
- Breaking the transaction into a number of subtransactions and distributing these subtransactions to the appropriate sites for execution.
- Coordinating the termination of the transaction, which may result in the transaction being committed at all sites or aborted at all sites.

System Failure Modes

The basic failure types in distributed database system is

- Failure of a site.
- Loss of messages.
- Failure of a communication link.
- Network partition.

Centralized System Architecture



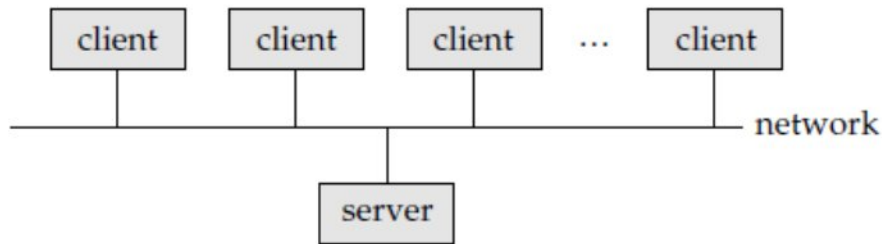
Client Server system

In client server system clients will send the request and server will send the response to the client. Now a days , centralized systems today act as **server systems** that satisfy requests generated by *client systems*. Figure shows the general structure of a client–server system.

Database functionality can be broadly divided into two parts—the front end and the back end as in the second figure. The back end manages access structures, query evaluation and optimization, concurrency control, and recovery. The front end of a database system consists of tools such as forms, report writers, and graphical user interface facilities.

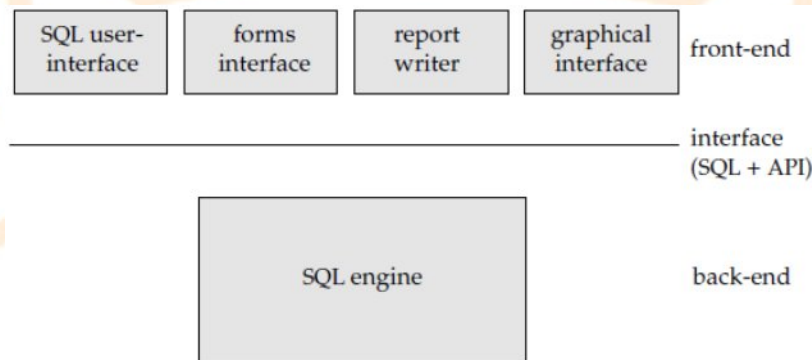
PRATHYUSHA ENGINEERING COLLEGE

The interface between the front end and the back end is through SQL, or through an application program.



General structure of a client-server system.

Some transaction-processing systems provide a **transactional remote procedure call** interface to connect clients with a server. These calls appear like ordinary procedure calls to the programmer, but all the remote procedure calls from a client are enclosed in a single transaction at the server end. Thus, if the transaction aborts, the server can undo the effects of the individual remote procedure calls.



Front-end and back-end functionality.

Server systems can be broadly categorized as transaction servers and data servers.

- **Transaction-server** systems, also called **query-server** systems, provide an interface to which clients can send requests to perform an action, in response to which they execute the action and send back results to the client. Usually, client machines ship transactions to the server systems, where those transactions are executed, and results are shipped back to

clients that are in charge of displaying the data. Requests may be specified by using SQL, or through a specialized application program interface.

- **Data-server systems** allow clients to interact with the servers by making requests to read or update data, in units such as files or pages. For example, file servers provide a file-system interface where clients can create, update, read, and delete files. Data servers for database systems offer much more functionality; they support units of data—such as pages, tuples, or objects—that are smaller than a file. They provide indexing facilities for data, and provide transaction facilities so that the data are never left in an inconsistent state if a client machine or process fails.

Multidimensional Database

A multidimensional database (MDB) is a type of [database](#) that is optimized for [data warehouse](#) and online analytical processing ([OLAP](#)) applications. Multidimensional databases are frequently created using input from existing [relational databases](#). Whereas a relational database is typically accessed using a Structured Query Language ([SQL](#)) [query](#).

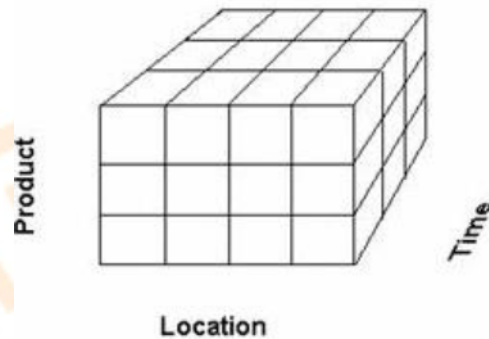
A multidimensional database allows a user to ask questions like "How many Aptivas have been sold in Nebraska so far this year?" and similar questions related to summarizing business operations and trends.

An OLAP application that accesses data from a multidimensional database is known as a [MOLAP](#)(multidimensional OLAP) application.

A multidimensional database - or a multidimensional database management system (MDDDBMS) - implies the ability to rapidly process the data in the database so that answers can be generated quickly. A number of vendors provide products that use multidimensional databases. Approaches to how data is stored and the user interface vary.

Conceptually, a multidimensional database uses the idea of a data cube to represent the dimensions of data available to a user.

For example, "sales" could be viewed in the dimensions of product model, geography, time, or some additional dimension.



Parallel DataBase

A **parallel database** system seeks to improve performance through **parallelization** of various operations, such as loading data, building indexes and evaluating queries. Although data may be stored in a distributed fashion, the distribution is governed solely by performance considerations. Parallel databases improve processing and **input/output** speeds by using multiple **CPUs** and disks in parallel.

Centralized and **client-server** database systems are not powerful enough to handle such applications. In parallel processing, many operations are performed simultaneously, as opposed to serial processing, in which the computational steps are performed sequentially.

Parallel databases can be roughly divided into two groups,

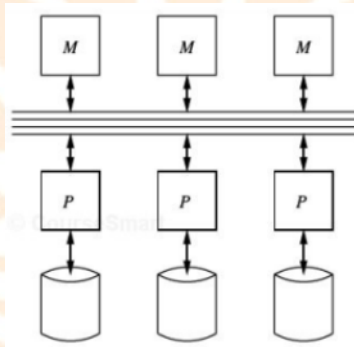
The first group of architecture is the multiprocessor architecture, the alternatives of which are the followings :

- **Shared memory architecture**, where multiple **processors** share the **main memory** space.

- **Shared disk architecture**, where each node has its own main memory, but all nodes share mass storage, usually a **storage area network**. In practice, each node usually also has multiple processors.
- **Shared nothing architecture**, where each node has its own mass storage as well as main memory.

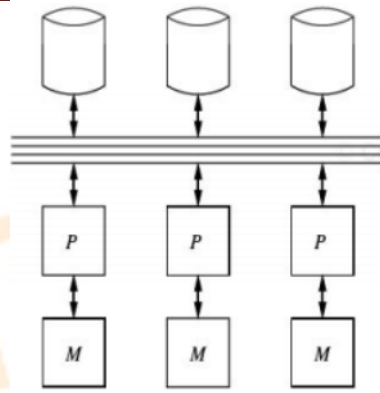
SHARED-MEMORY ARCHITECTURE

- Every processor has its own disk
- Single memory address-space for all processors
- Reading or writing to far memory can be slightly more expensive
- Every processor can have its own local memory and cache as well



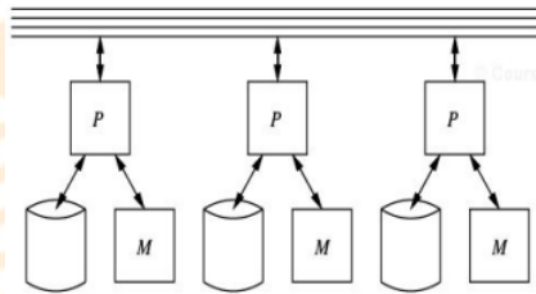
SHARED-DISK ARCHITECTURE

- Every processor has its own memory (not accessible by others)
 - All machines can access all disks in the system
 - Number of disks does not necessarily match the number of processors
- Multimedia Database



SHARED-NOTHING ARCHITECTURE

- Every machine has its own memory and disk.
- Communication is done through high speed network and switches.



Spatial Database

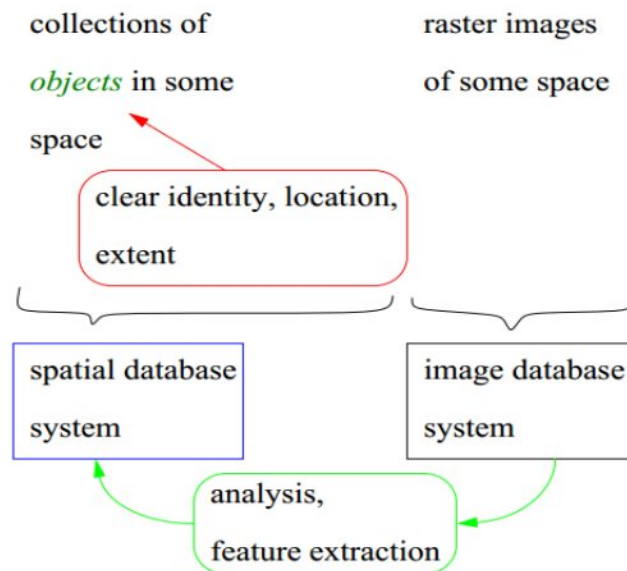
A spatial database is optimized to store and access spatial data or data that defines a geometric space. These data are often associated with geographic locations and features, or constructed features like cities. Data on spatial databases are stored as coordinates, points, lines, polygons and topology. Some spatial databases handle more complex data like three-dimensional objects, topological coverage and linear networks.

Common database systems use indexes for a faster and more efficient search and access of data. This index, however, is not fit for spatial queries. Instead, spatial databases use

something like a unique index called a spatial index to speed up database performance. Spatial indexing is very much required because a system should be able to retrieve data from a large collection of objects without really searching the whole bunch. It should also support relationships between connecting objects from different classes in a better manner than just filtering.

Aside from the indexes, spatial databases also offer spatial data types in their data model and query language. These databases require special kinds of data types to provide a fundamental abstraction and model the structure of the geometric objects with their corresponding relationships and operations in the spatial environment. Without these kind of data types, the system would not be able to support the kind of modeling a spatial database offers.

A database may contain



Multimedia Database

A multimedia database management system (MM-DBMS) is a framework that manages different types of data potentially represented in a wide diversity of formats on a wide array of media sources.

Requirements of Multimedia DBMS:

Persistence - Data objects can be saved and re-used by different transactions and program invocations

Privacy - Access and authorization control

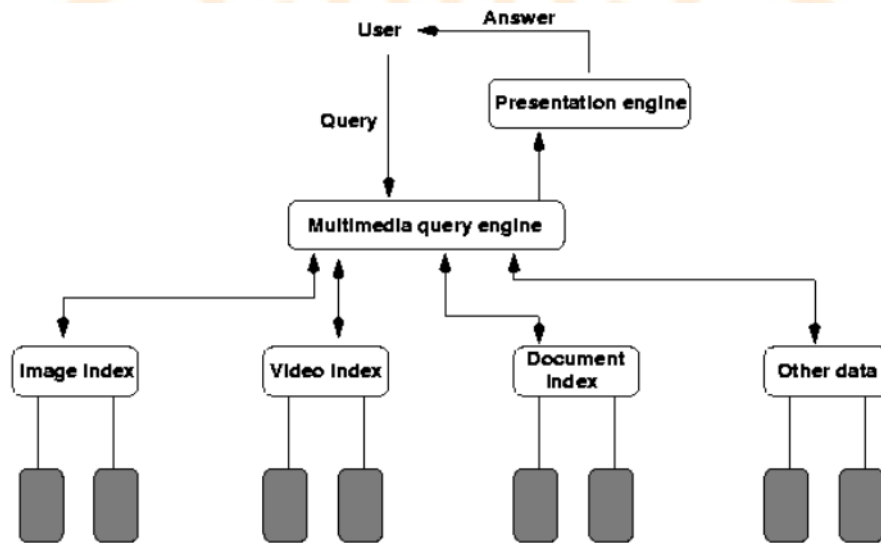
Integrity control - Ensures database consistency between transactions

Recovery - Failures of transactions should not affect the persistent data storage

Query support - Allows easy querying of multimedia data.

Based on Principle of Autonomy

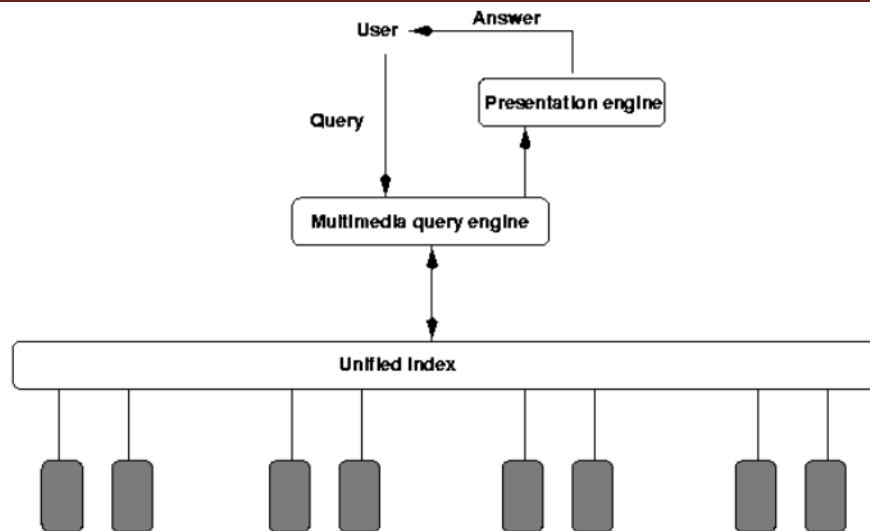
§ Each media type is organized in a media-specific manner suitable for that media type



Multimedia database Architecture

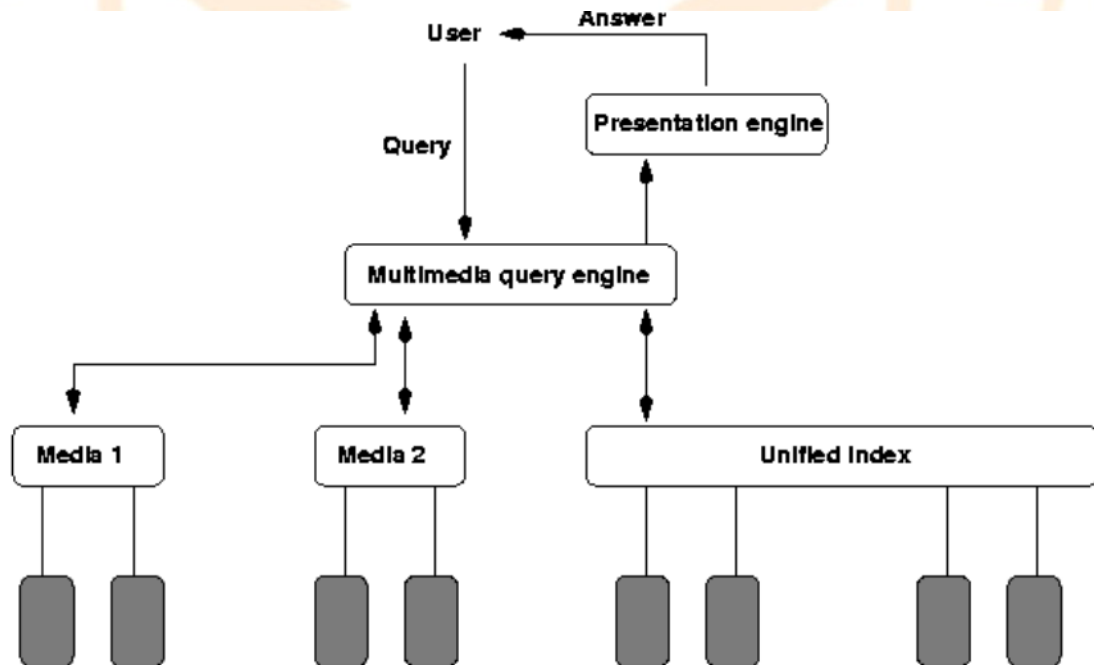
Based on Principle of Uniformity

§ A single abstract structure to index all media types



Based on Principle of Hybrid Organization

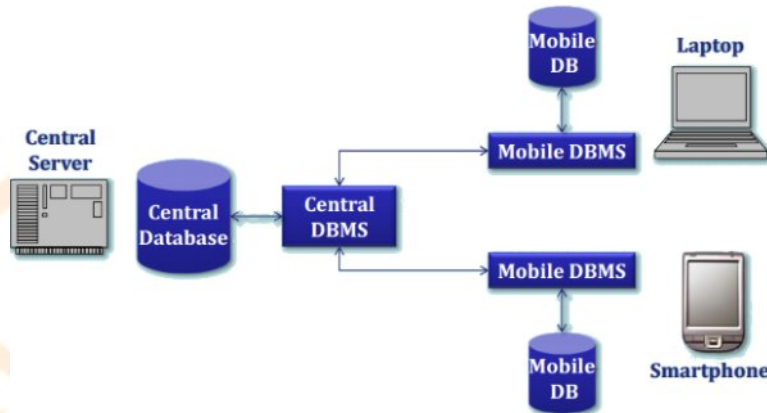
- § A hybrid of the first two. Certain media types use their own indexes, while others use the "unified" index
- § An attempt to capture the advantages of the first two approaches



Mobile Databases :

- Physically separate from the centralized server.
- Resided on mobile devices.

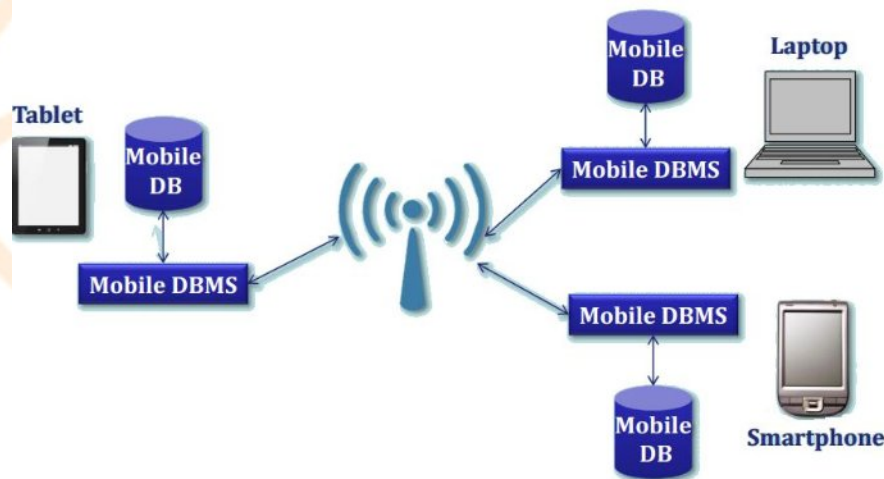
- Capable of communicating with a central database server or other mobile clients from remote devices.
- Handle local queries without connectivity.



Client - Server Mobile Databases

Requirements of Mobile Database

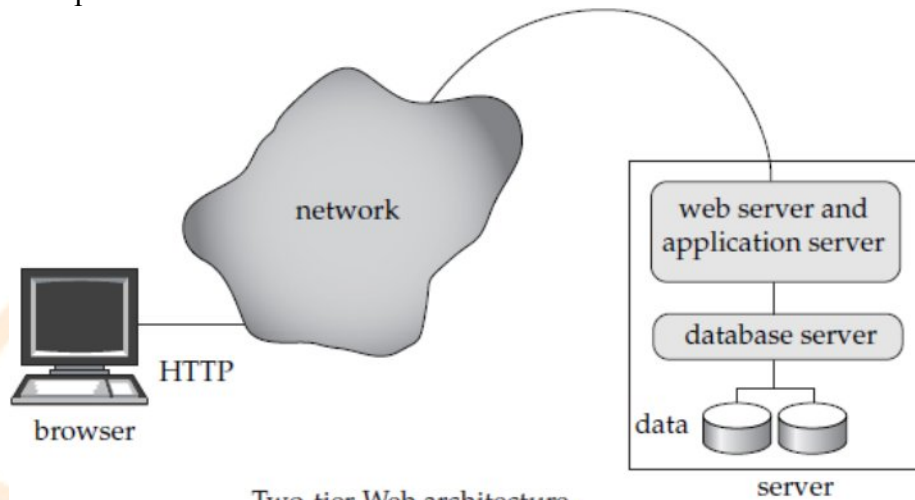
- Communicates with a centralized database server in a mode such as:
 - Wirelessly
 - Using Internet
- Replicates and Synchronizes data on the centralized server and mobile unit.
- Can capture data from various sources such as the Internet
- Manages and Analyzes the data on the mobile unit
- Can create custom mobile applications



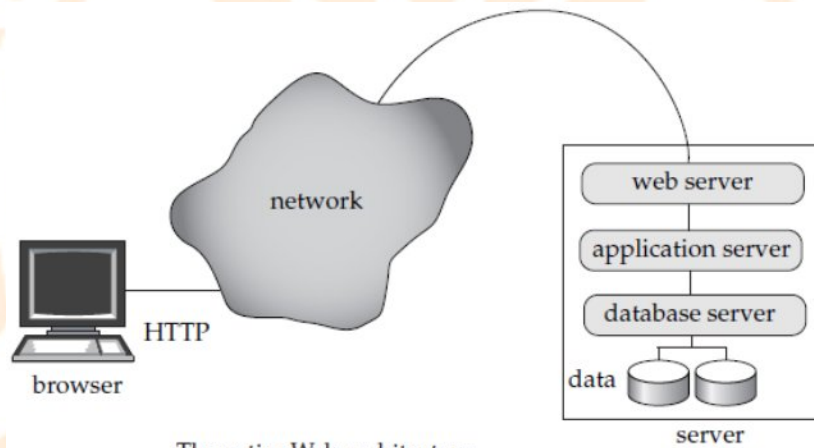
Peer - to - Peer Mobile Databases

Web Database

A database that is portable and physically separate from the corporate database server, but is capable of communicating with that corporate database server from remote sites allowing the sharing of corporate data.



Two-tier Web architecture.



Three-tier Web architecture.

Data Warehousing

A **data warehouse** is a repository (or archive) of information gathered from multiple sources, stored under a unified schema, at a single site. Once gathered, the data are stored for a long time, permitting access to historical data. Thus, data warehouses provide the user a single consolidated interface to data, making decision-support queries easier to write.

Components of a Data Warehouse

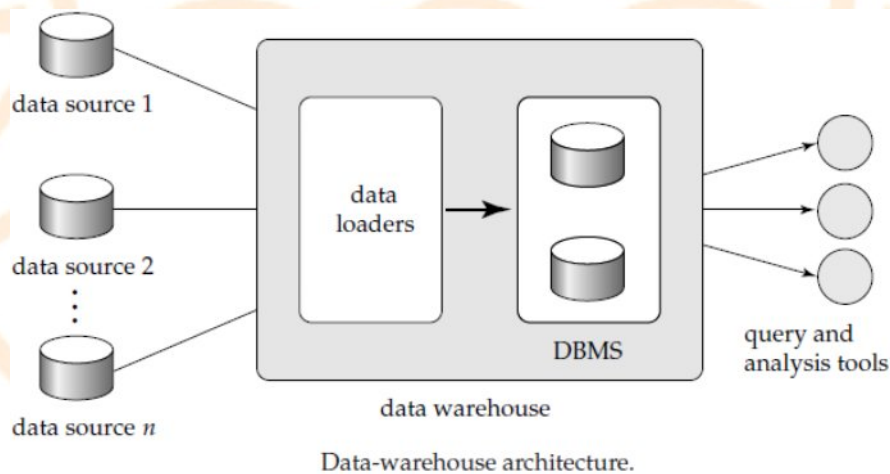
When and how to gather data. In a **source-driven architecture** for gathering data, the data sources transmit new information, either continually (as transaction processing takes place), or periodically (nightly, for example). In a **destination-driven architecture**, the data warehouse periodically sends requests for new data to the sources.

What schema to use. Data sources that have been constructed independently are likely to have different schemas. In fact, they may even use different data models. Part of the task of a warehouse is to perform schema integration, and to convert data to the integrated schema before they are stored.

Data cleansing. The task of correcting and preprocessing data is called **data cleansing**. Data sources often deliver data with numerous minor inconsistencies, that can be corrected.

How to propagate updates. Updates on relations at the data sources must be propagated to the data warehouse. If the relations at the data warehouse are exactly the same as those at the data source, the propagation is straightforward.

What data to summarize. The raw data generated by a transaction-processing system may be too large to store online. However, we can answer many queries by maintaining just summary data obtained by aggregation on a relation, rather than maintaining the entire relation.



Data Mining

Data mining deals with “knowledge discovery in databases.”

Applications of Data Mining

Prediction

The discovered knowledge has numerous applications. The most widely used applications are those that require some sort of **prediction**. For instance, when a person applies for a credit card, the credit-card company wants to predict if the person is a good credit risk. The prediction is to be based on known attributes of the person, such as age, income, debts, and past debt repayment history. Rules for making the prediction are derived from the same attributes of past and current credit card holders, along with their observed behavior, such as whether they defaulted on their creditcard dues.

Classification

Classification can be done by finding rules that partition the given data into disjoint groups. For instance, suppose that a credit-card company wants to decide whether or not to give a credit card to an applicant. The company has a variety of information about the person, such as her age, educational background, annual income, and current debts, that it can use for making a decision. Some of this information could be relevant to the credit worthiness of the applicant, whereas some may not be. To make the decision, the company assigns a creditworthiness level of excellent, good, average, or bad to each of a sample set of *current* customers according to each customer's payment history. Then, the company attempts to find rules that classify its current customers into excellent, good, average, or bad, on the basis of the information about the person, other than the actual payment history (which is unavailable for new customers). Let us consider just two attributes: education level (highest degree earned) and income. The rules may be of the following form:

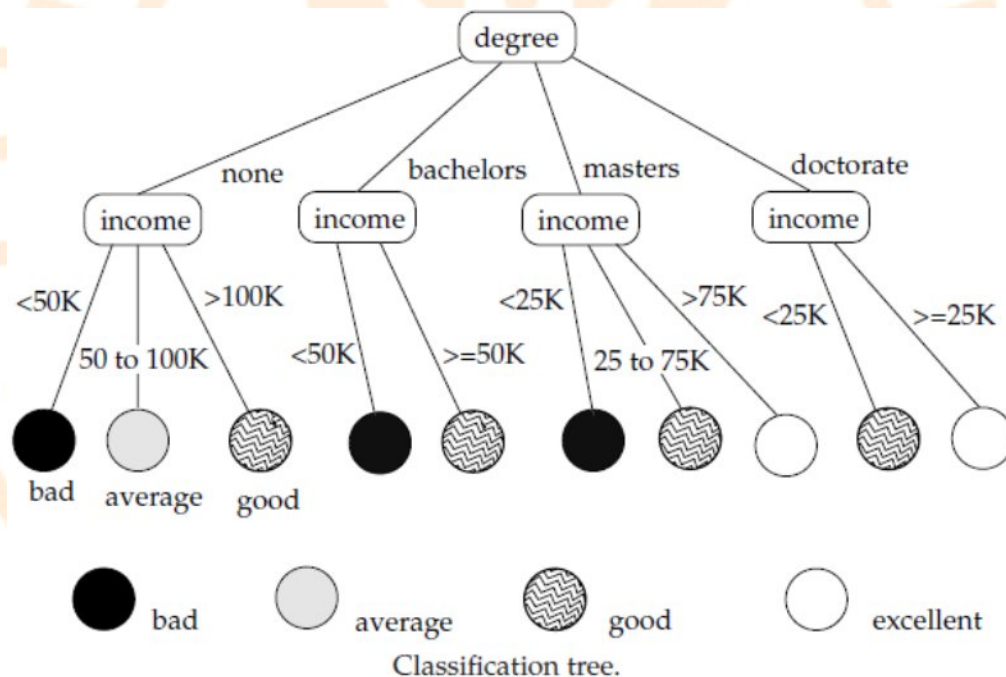
$$\text{Person } P, P.\text{degree} = \text{masters and } P.\text{income} > 75,000 \Rightarrow P.\text{credit} = \text{excellent}$$
$$\text{Person } P, P.\text{degree} = \text{bachelors or } (P.\text{income} \geq 25,000 \text{ and } P.\text{income} \leq 75,000) \Rightarrow P.\text{credit} = \text{good}$$

Similar rules would also be present for the other credit worthiness levels (average and bad).

The process of building a classifier starts from a sample of data, called a **training set**. For each tuple in the training set, the class to which the tuple belongs is already known. For instance, the training set for a credit-card application may be the existing customers, with their credit worthiness determined from their payment history. The actual data, or population, may consist of all people, including those who are not existing customers.

Decision tree classification

The decision tree classifier is a widely used technique for classification. **decision tree classifiers** use a tree; each leaf node has an associated class, and each internal node has a predicate (or more generally, a function) associated with it.



In our example,

The attribute degree is chosen, and four children, one for each value of degree, are created.

PRATHYUSHA ENGINEERING COLLEGE

The conditions for the four children nodes are degree = none, degree = bachelors, degree = masters, and degree = doctorate, respectively. The data associated with each child consist of training instances satisfying the condition associated with that child.

At the node corresponding to masters, the attribute income is chosen, with the range of values partitioned into intervals 0 to 25,000, 25,000 to 50,000, 50,000 to 75,000, and over 75,000. The data associated with each node consist of training instances with the degree attribute being masters, and the income attribute being in each of these ranges respectively. As an optimization, since the class for the range 25,000 to 50,000 and the range 50,000 to 75,000 is the same under the node *degree* = masters, the two ranges have been merged into a single range 25,000 to 75,000

All of this leads to a definition: The **best split** for an attribute is the one that gives the maximum **information gain ratio**, defined as

$$\frac{\text{Information-gain}(S, \{S_1, S_2, \dots, S_r\})}{\text{Information-content}(S, \{S_1, S_2, \dots, S_r\})}$$

Association Rules

Retail shops are often interested in **associations** between different items that people buy. Examples of such associations are:

- Someone who buys bread is quite likely also to buy milk
- A person who bought the book *Database System Concepts* is quite likely also to buy the book *Operating System Concepts*.

A grocery shop may decide to place bread close to milk, since they are often bought together, to help shoppers finish their task faster. Or the shop may place them at opposite ends of a row, and place other associated items in between to tempt people to buy those items as well, as the shoppers walk from one end of the row to the other. A shop that offers discounts on one

associated item may not offer a discount on the other, since the customer will probably buy the other anyway.

Association Rules

An example of an association rule is

$bread \Rightarrow milk$

An association rule must have an associated **population**: the population consists of a set of **instances**. In the grocery-store example, the population may consist of all grocery store purchases; each purchase is an instance. In the case of a bookstore, the population may consist of all people who made purchases, regardless of when they made a purchase. Each customer is an instance.

Rules have an associated *support*, as well as an associated *confidence*. These are defined in the context of the population:

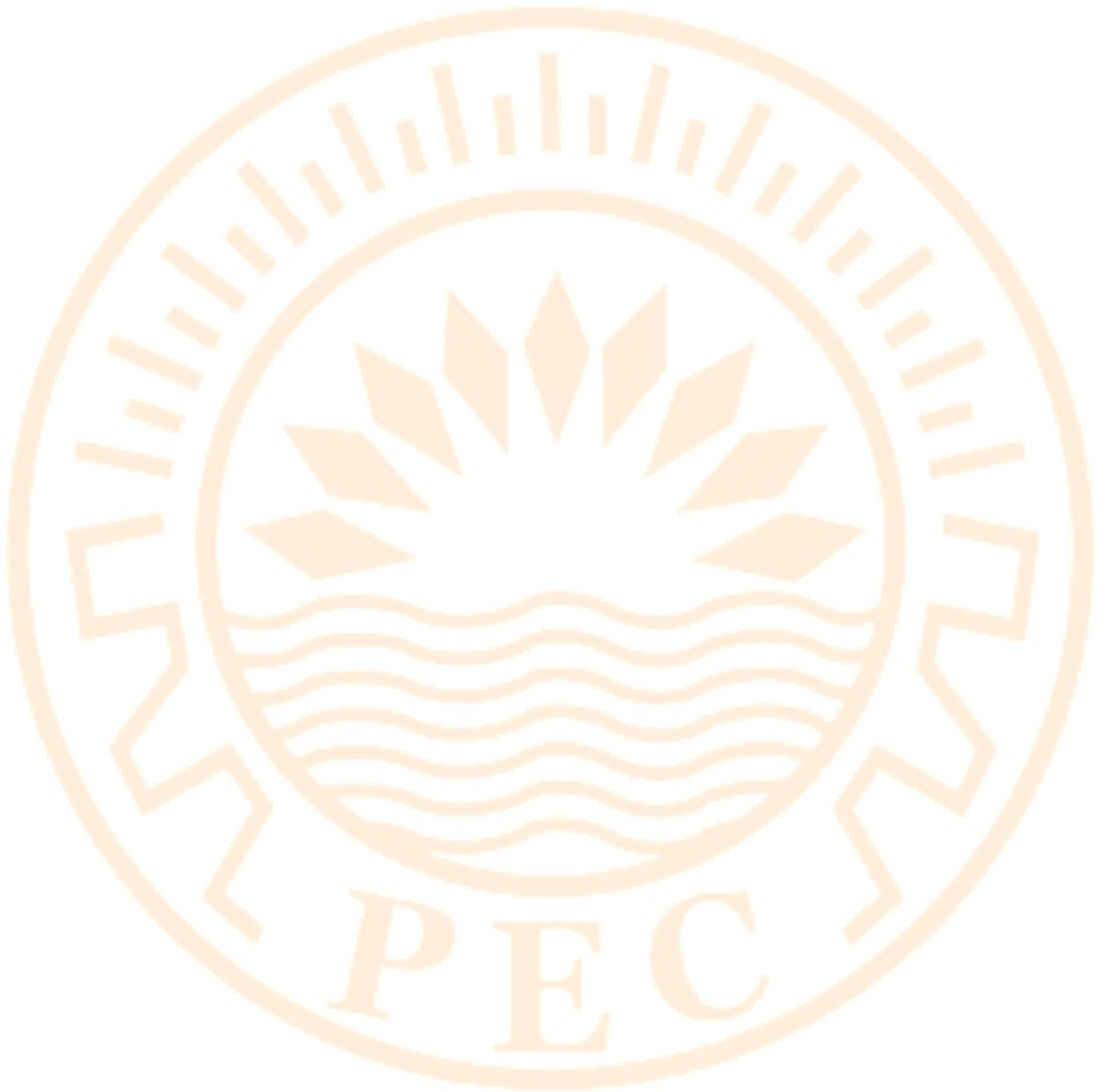
- **Support** is a measure of what fraction of the population satisfies both the antecedent and the consequent of the rule.

For instance, suppose only 0.001 percent of all purchases include milk and screwdrivers. The support for the rule $milk \Rightarrow screwdrivers$ is low.

Confidence is a measure of how often the consequent is true when the antecedent is true. For instance, the rule

$bread \Rightarrow milk$

has a confidence of 80 percent if 80 percent of the purchases that include bread also include milk.



UNIT V ADVANCED TOPICS 9

DATABASE SECURITY: Data Classification-Threats and risks – Database access Control – Types of Privileges –Cryptography- Statistical Databases.- Distributed Databases-Architecture-Transaction Processing-Data Warehousing and Mining-Classification-Association rules-Clustering-Information Retrieval- Relevance ranking-Crawling and Indexing the Web- Object Oriented Databases-XML Databases.

Cryptography

The art or science encompassing the principles and methods of transforming an intelligible message into one that is unintelligible, and then retransforming that message back to its original form

Plaintext - The original intelligible message

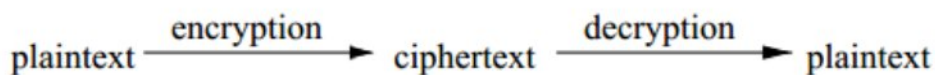
Ciphertext - The transformed message

Cipher - An algorithm for transforming an intelligible message into unintelligible by transposition and/or substitution

Key- Some critical information used by the cipher, known only to the sender & receiver

Encipher (encode)- The process of converting plaintext to ciphertext

Decipher (decode)- The process of converting ciphertext back into plaintext.



Modern cryptography concerns itself with the following four objectives:

- 1) **Confidentiality** (the information cannot be understood by anyone for whom it was unintended)
- 2) **Integrity** (the information cannot be altered in storage or transit between sender and intended receiver without the alteration being detected)

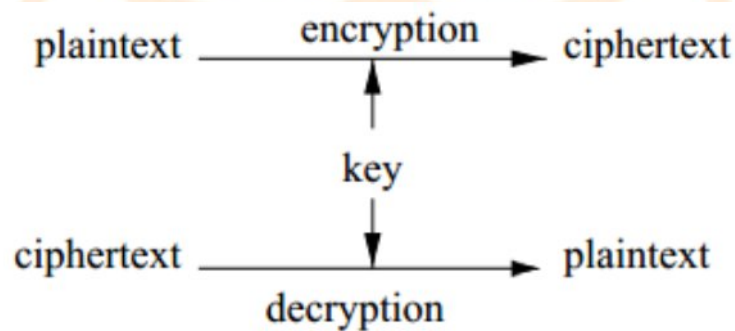
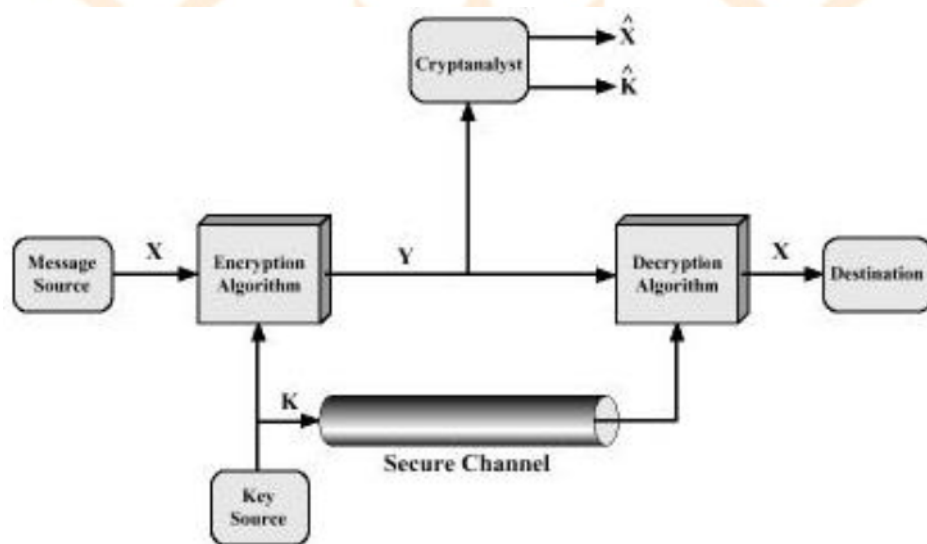
3) **Non-repudiation** (the creator/sender of the information cannot deny at a later stage his or her intentions in the creation or transmission of the information)

Secret Key Cryptography

In symmetric-key cryptography, we encode our plain text by mangling it with a secret key.

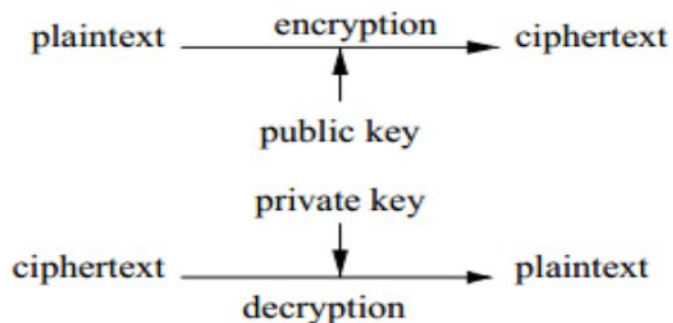
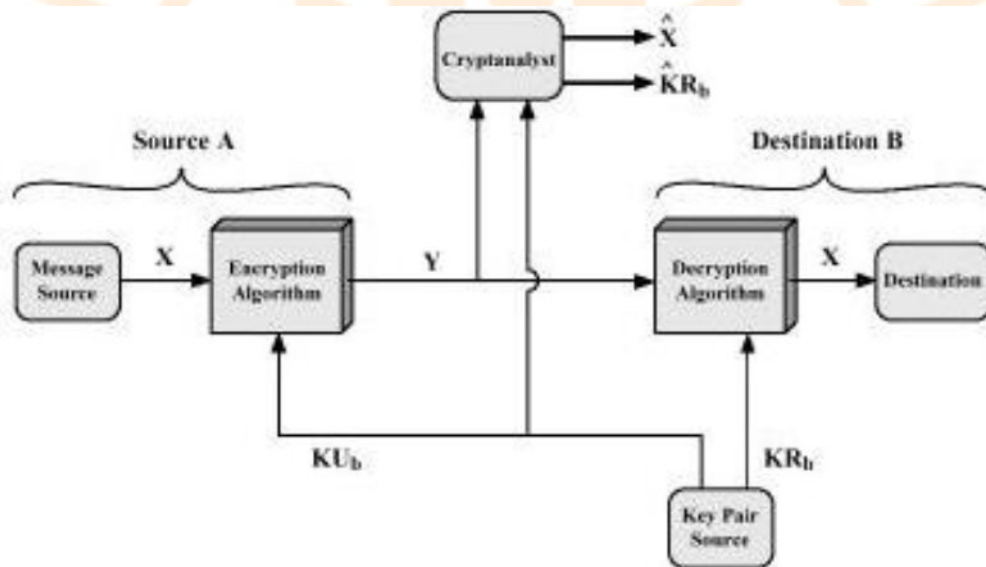
Decryption requires knowledge of the same key, and reverses the mangling.

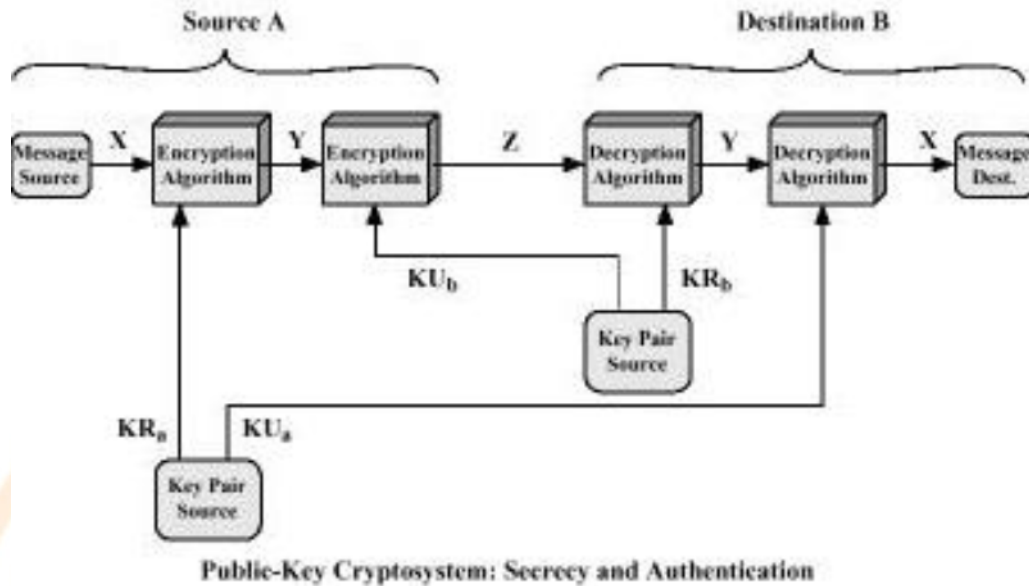
$\text{ciphertext} = \text{encrypt}(\text{plaintext}, \text{key})$
 $\text{plaintext} = \text{decrypt}(\text{ciphertext}, \text{key})$



Public Key Cryptography

Asymmetric cryptography or public-key cryptography is cryptography in which a pair of keys is used to encrypt and decrypt a message so that it arrives securely. Initially, a network user receives a public and private key pair from a certificate authority. Any other user who wants to send an encrypted message can get the intended recipient's public key from a public directory. They use this key to encrypt the message, and they send it to the recipient. When the recipient gets the message, they decrypt it with their private key, which no one else should have access to.





Statistical Databases

A **statistical database** is a [database](#) used for [statistical](#) analysis purposes. It is an [OLAP](#) (online analytical processing), instead of [OLTP](#) (online transaction processing) system. Modern decision, and classical statistical databases are often closer to the [relational model](#) than the [multidimensional](#) model commonly used in [OLAP](#) systems today.

Statistical databases typically contain parameter data and the measured data for these parameters. For example, parameter data consists of the different values for varying conditions in an experiment (e.g., temperature, time). The measured data (or variables) are the measurements taken in the experiment under these varying conditions.

Many statistical databases are sparse with many null or zero values. It is not uncommon for a statistical database to be 40% to 50% sparse.

There are two options for dealing with the sparseness:

- (1) leave the null values in there and use compression techniques to squeeze them out.
- (2) remove the entries that only have null values.

PRATHYUSHA ENGINEERING COLLEGE

Statistical databases often incorporate support for advanced statistical analysis techniques, such as correlations, which go beyond [SQL](#). They also pose unique [security](#) concerns, which were the focus of much research, particularly in the late 1970s and early to mid-1980s.

In a statistical database, it is often desired to allow query access only to aggregate data, not individual records. Securing such a database is a difficult problem, since intelligent users can use a combination of aggregate queries to derive information about a single individual.

Some common approaches are:

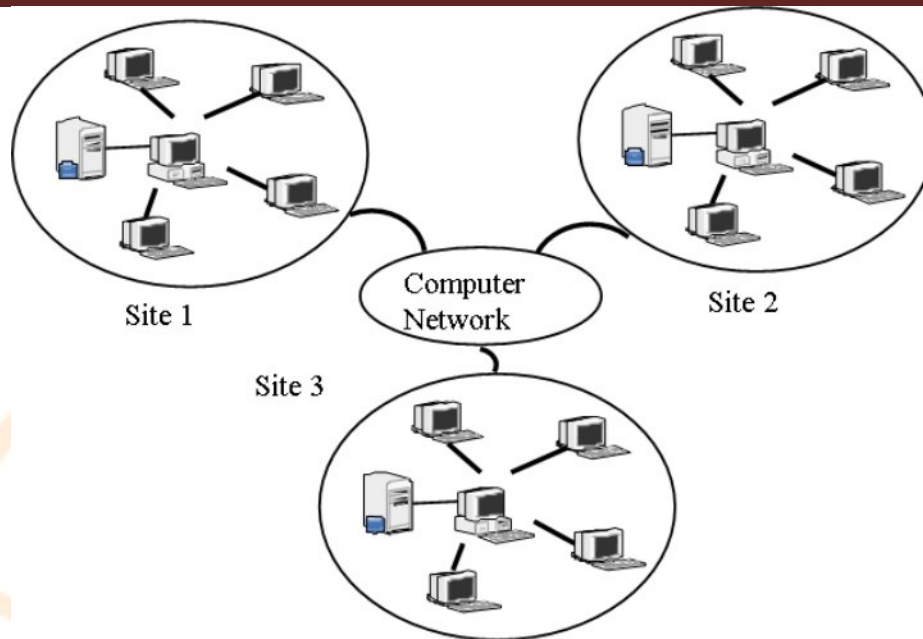
- only allowing aggregate queries (SUM, COUNT, AVG, STDEV, etc.)
- rather than returning exact values for sensitive data like income, only return which partition it belongs to (e.g. 35k-40k)
- return imprecise counts (e.g. rather than 141 records met query, only indicate 130-150 records met it.)
- don't allow overly selective WHERE clauses
- audit all users queries, so users using system incorrectly can be investigated
- use intelligent agents to detect automatically inappropriate system use

Distributed Databases

Distributed Database (DDB) is a collection of multiple, logically interrelated databases distributed over a computer network.

A **distributed database management system (DDBMS)** is the software that manages the DDB and provides an access mechanism that makes this distribution transparent to the users.

Distributed Database Architecture



Distributed Transactions

The Distributed transactions must preserve the ACID properties. There are two types of transaction that we need to consider. The **local transactions** are those that access and update data in only one local database; the **global transactions** are those that access and update data in several local databases.

Each site has its own *local* transaction manager, whose function is to ensure the ACID properties of those transactions that execute at that site. The various transaction managers cooperate to execute global transactions.

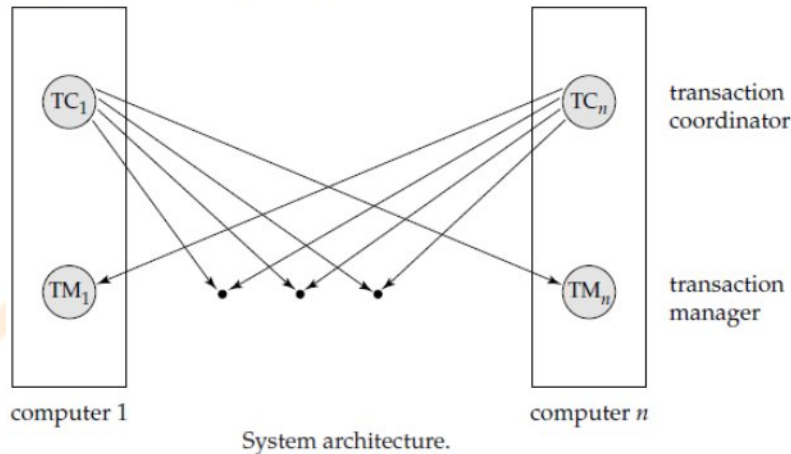
each site contains two subsystems:

- The **transaction manager** manages the execution of those transactions (or subtransactions)

that access data stored in a local site. Note that each such transaction may be either a local transaction (that is, a transaction that executes at only that site) or part of a global transaction (that is, a transaction that executes at several sites).

- The **transaction coordinator** coordinates the execution of the various transactions (both local and global) initiated at that site.

The overall system architecture



Each transaction manager is responsible for

- Maintaining a log for recovery purposes.
- Participating in an appropriate concurrency-control scheme to coordinate the concurrent execution of the transactions executing at that site.

The transaction coordinator is responsible for

- Starting the execution of the transaction.
- Breaking the transaction into a number of subtransactions and distributing these subtransactions to the appropriate sites for execution.
- Coordinating the termination of the transaction, which may result in the transaction being committed at all sites or aborted at all sites.

Commit Protocols

If we are to ensure atomicity, all the sites in which a transaction T executed must agree on the final outcome of the execution. T must either commit at all sites, or it must abort at all sites. To ensure this property, the transaction coordinator of T must execute a *commit protocol*.

Two-Phase Commit

When T completes its execution—that is, when all the sites at which T has executed inform C_i that T has completed— C_i starts the 2PC protocol.

Phase 1.

- C_i adds the record $\langle \text{prepare } T \rangle$ to the log, and forces the log onto stable storage.
- It then sends a prepare T message to all sites. On receiving such a message, the transaction manager determines whether it is willing to commit its portion of T .
- If the answer is no, it adds a record $\langle \text{no } T \rangle$ to the log, and then responds by sending an abort T message to C_i .
- If the answer is yes, it adds a record $\langle \text{ready } T \rangle$ to the log, and forces the log onto stable storage.
- Then, The transaction manager replies with a ready T message to C_i .

Phase 2.

- When C_i receives responses to the prepare T message from all the sites prespecified interval of time has elapsed since the prepare T message was sent out, C_i can determine whether the transaction T can be committed or aborted.
- Transaction T can be committed if C_i received a ready T message from all the participating sites.
- Otherwise, transaction T must be aborted.
- Depending on that, either a record $\langle \text{commit } T \rangle$ or a record $\langle \text{abort } T \rangle$ is added to the log and the log is forced onto stable storage.

Following this point, the coordinator sends either a commit T or an abort T message to all participating sites. When a site receives that message, it records the message in the log.

System Failure Modes

The basic failure types in distributed database system is

- Failure of a site.
- Loss of messages.
- Failure of a communication link.
- Network partition.

Data Warehouse

A **data warehouse** is a repository (or archive) of information gathered from multiple sources, stored under a unified schema, at a single site. Once gathered, the data are stored for a long time, permitting access to historical data. Thus, data warehouses provide the user a single consolidated interface to data, making decision-support queries easier to write.

Components of a Data Warehouse

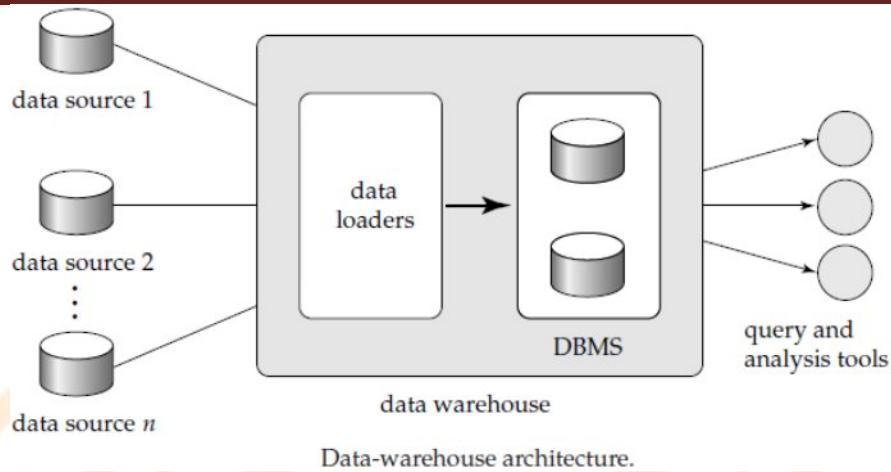
When and how to gather data. In a **source-driven architecture** for gathering data, the data sources transmit new information, either continually (as transaction processing takes place), or periodically (nightly, for example). In a **destination-driven architecture**, the data warehouse periodically sends requests for new data to the sources.

What schema to use. Data sources that have been constructed independently are likely to have different schemas. In fact, they may even use different data models. Part of the task of a warehouse is to perform schema integration, and to convert data to the integrated schema before they are stored.

Data cleansing. The task of correcting and preprocessing data is called **data cleansing**. Data sources often deliver data with numerous minor inconsistencies, that can be corrected.

How to propagate updates. Updates on relations at the data sources must be propagated to the data warehouse. If the relations at the data warehouse are exactly the same as those at the data source, the propagation is straightforward.

What data to summarize. The raw data generated by a transaction-processing system may be too large to store online. However, we can answer many queries by maintaining just summary data obtained by aggregation on a relation, rather than maintaining the entire relation.



Types of Data Warehouses

- **Enterprise Warehouse:** covers all areas of interest for an organization
- **Data Mart:** covers a subset of corporate-wide data that is of interest for a specific user group (e.g., marketing).
- **Virtual Warehouse:** offers a set of views constructed on demand on operational databases. Some of the views could be materialized (precomputed)

Data Mining

Data mining deals with “knowledge discovery in databases.”

Applications of Data Mining

Prediction

The discovered knowledge has numerous applications. The most widely used applications are those that require some sort of **prediction**. For instance, when a person applies for a credit card, the credit-card company wants to predict if the person is a good credit risk. The prediction is to be based on known attributes of the person, such as age, income, debts, and past debt repayment history. Rules for making the prediction are derived from the same attributes of past and current credit card holders, along with their observed behavior, such as whether they defaulted on their creditcard dues.

Classification

Classification can be done by finding rules that partition the given data into disjoint groups. For instance, suppose that a credit-card company wants to decide whether or not to give a credit card to an applicant. The company has a variety of information about the person, such as her age, educational background, annual income, and current debts, that it can use for making a decision. Some of this information could be relevant to the credit worthiness of the applicant, whereas some may not be. To make the decision, the company assigns a creditworthiness level of excellent, good, average, or bad to each of a sample set of *current* customers according to each customer's payment history. Then, the company attempts to find rules that classify its current customers into excellent, good, average, or bad, on the basis of the information about the person, other than the actual payment history (which is unavailable for new customers). Let us consider just two attributes: education level (highest degree earned) and income. The rules may be of the following form:

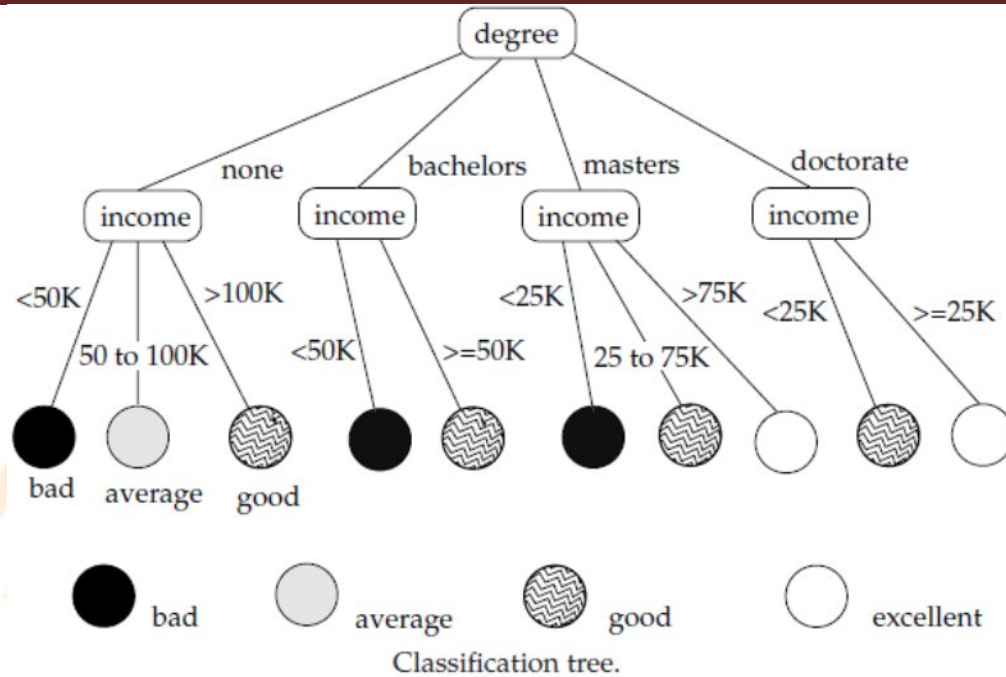
$\forall \text{ Person } P, P.\text{degree} = \text{masters and } P.\text{income} > 75,000 \Rightarrow P.\text{credit} = \text{excellent}$
 $\forall \text{ Person } P, P.\text{degree} = \text{bachelors or } (P.\text{income} \geq 25,000 \text{ and } P.\text{income} \leq 75,000) \Rightarrow P.\text{credit} = \text{good}$

Similar rules would also be present for the other credit worthiness levels (average and bad).

The process of building a classifier starts from a sample of data, called a **training set**. For each tuple in the training set, the class to which the tuple belongs is already known. For instance, the training set for a credit-card application may be the existing customers, with their credit worthiness determined from their payment history. The actual data, or population, may consist of all people, including those who are not existing customers.

Decision tree classification

The decision tree classifier is a widely used technique for classification. **decision tree classifiers** use a tree; each leaf node has an associated class, and each internal node has a predicate (or more generally, a function) associated with it.



In our example,

The attribute degree is chosen, and four children, one for each value of degree, are created.

The conditions for the four children nodes are degree = none, degree = bachelors, degree = masters, and degree = doctorate, respectively. The data associated with each child consist of training instances satisfying the condition associated with that child.

At the node corresponding to masters, the attribute income is chosen, with the range of values partitioned into intervals 0 to 25,000, 25,000 to 50,000, 50,000 to 75,000, and over 75,000. The data associated with each node consist of training instances with the degree attribute being masters, and the income attribute being in each of these ranges respectively. As an optimization, since the class for the range 25,000 to 50,000 and the range 50,000 to 75,000 is the same under the node *degree* = masters, the two ranges have been merged into a single range 25,000 to 75,000

All of this leads to a definition: The **best split** for an attribute is the one that gives the maximum **information gain ratio**, defined as

$$\text{Information-gain}(S, \{S_1, S_2, \dots, S_r\})$$

Association Rules

Retail shops are often interested in **associations** between different items that people buy.

Examples of such associations are:

- Someone who buys bread is quite likely also to buy milk
- A person who bought the book *Database System Concepts* is quite likely also to buy the book *Operating System Concepts*.

A grocery shop may decide to place bread close to milk, since they are often bought together, to help shoppers finish their task faster. Or the shop may place them at opposite ends of a row, and place other associated items in between to tempt people to buy those items as well, as the shoppers walk from one end of the row to the other. A shop that offers discounts on one associated item may not offer a discount on the other, since the customer will probably buy the other anyway.

Association Rules

An example of an association rule is

$bread \Rightarrow milk$

An association rule must have an associated **population**: the population consists of a set of **instances**. In the grocery-store example, the population may consist of all grocery store purchases; each purchase is an instance. In the case of a bookstore, the population may consist of all people who made purchases, regardless of when they made a purchase. Each customer is an instance.

Rules have an associated *support*, as well as an associated *confidence*. These are defined in the context of the population:

- **Support** is a measure of what fraction of the population satisfies both the antecedent and the consequent of the rule.

For instance, suppose only 0.001 percent of all purchases include milk and screwdrivers. The support for the rule $\text{milk} \Rightarrow \text{screwdrivers}$ is low.

Confidence is a measure of how often the consequent is true when the antecedent is true. For instance, the rule

$$\text{bread} \Rightarrow \text{milk}$$

has a confidence of 80 percent if 80 percent of the purchases that include bread also include milk.

Clustering

clustering refers to the problem of finding clusters of points in the given data. The problem of **clustering** can be formalized from distance metrics in several ways. One way is to phrase it as the problem of grouping points into k sets (for a given k) so that the average distance of points from the *centroid* of their assigned cluster is minimized.

Information Retrieval

Textual data is unstructured, unlike the rigidly structured data in relational databases. Querying of unstructured textual data is referred to as *information retrieval*. Information retrieval systems have much in common with database systems—in particular, the storage and retrieval of data on secondary storage.

Information retrieval systems are used to store and query textual data such as documents. They use a simpler data model than do database systems, but provide more powerful querying capabilities within the restricted model.

Queries attempt to locate documents that are of interest by specifying, for example, sets of keywords. The query that a user has in mind usually cannot be stated precisely; hence, information-retrieval systems order answers on the basis of potential relevance.

Ranking

PRATHYUSHA ENGINEERING COLLEGE

Finding the position of a value in a larger set is a common operation. For instance, we may wish to assign students a rank in class based on their total marks, with the rank 1 going to the student with the highest marks, the rank 2 to the student with the next highest marks, and so on. While such queries can be expressed in SQL-92, they are difficult to express and inefficient to evaluate. Ranking is done in conjunction with an **order by** specification. Suppose we are given a relation *student-marks(student-id, marks)* which stores the marks obtained by each student. The following query gives the rank of each student.

```
select student-id, rank() over (order by (marks) desc) as s-rank from student-marks
```

Multiple **rank** expressions can be used within a single select statement; thus we can obtain the overall rank and the rank within the section by using two **rank** expressions in the same **select** clause.

What happens, when ranking (possibly with partitioning) occurs along with a **group by** clause. In this case, the **group by** clause is applied first, and partitioning and ranking are done on the results of the group by. Thus aggregate values can then be used for ranking.

The presence of null values can complicate the definition of rank, since it is not clear where they should occur first in the sort order. SQL:1999 permits the user to specify where they should occur by using **nulls first** or **nulls last**, for instance

```
select student-id, rank () over (order by marks desc nulls last) as s-rank from student-marks
```

Relevance ranking

The set of all documents that satisfy a query expression may be very large; in particular, there are billions of documents on the Web, and most keyword queries on a Web search engine find hundreds of thousands of documents containing the keywords. Full text retrieval makes this problem worse: Each document may contain many terms, and even terms that are only mentioned in passing are treated equivalently with documents where the term is indeed relevant. Irrelevant documents may get retrieved as a result.

Information retrieval systems therefore estimate relevance of documents to a query, and return only highly ranked documents as answers. Relevance ranking is not an exact science, but there are some well-accepted approaches.

The first question to address is, given a particular term t , how relevant is a particular document d to the term. One approach is to use the the number of occurrences of the term in the document as a measure of its relevance, on the assumption that relevant terms are likely to be mentioned many times in a document. Just counting the number of occurrences of a term is usually not a good indicator: First, the number of occurrences depends on the length of the document, and second, a document containing 10 occurrences of a term may not be 10 times as relevant as a document containing one occurrence.

One way of measuring $r(d, t)$, the relevance of a document d to a term t , is

$$r(d, t) = \log_1 + n(d, t) / n(d)$$

where $n(d)$ denotes the number of terms in the document and $n(d, t)$ denotes the number of occurrences of term t in the document d . Observe that this metric takes the length of the document into account. The relevance grows with more occurrences of a term in the document, although it is not directly proportional to the number of occurrences.

Relevance ranking makes use of several types of information, such as:

- Term frequency: how important each term is to each document.
- Inverse document frequency.
- Site popularity. Page rank and hub/authority rank are two ways to assign importance to sites on the basis of links to the site.

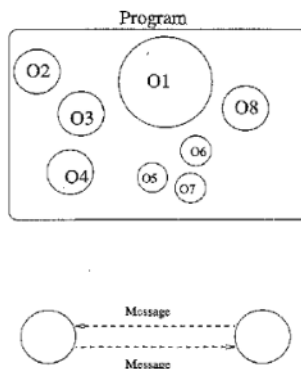
Object Oriented Databases

- Industry Trends: Integration and Sharing
- Seamless integration of operating systems, databases, languages, spreadsheets, word processors, AI expert system shells.

- Sharing of data, information, software components, products, computing environments.
- Referential sharing: Multiple applications, products, or objects share common sub-objects.

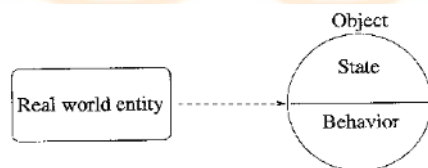
Fundamentals of Object-Oriented Approach

The object-oriented paradigm is illustrated below:



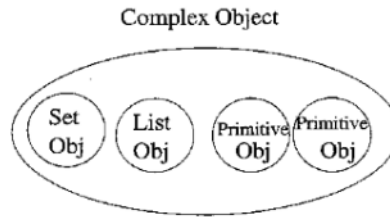
Objects and Identity

The following figure shows object with state and behavior. The state is represented by the values of the object's attributes, and the behavior is defined by the methods acting on the state of the object. There is a unique object identifier OID to identify the object.



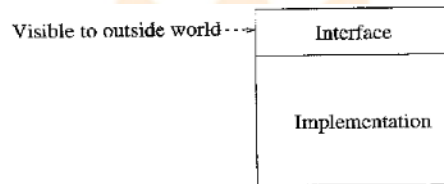
Complex Objects

Complex objects are built by applying constructors to simpler objects including: sets, lists and tuples. An example ,



Encapsulation

Encapsulation is derived from the notion of Abstract Data Type (ADT). It is motivated by the need to make a clear distinction between the specification and the implementation of an operation. It reinforces modularity and provides a form of logical data independence.



Class

A class object is an object which acts as a template.

It specifies:

A structure that is the set of attributes of the instances

A set of operations

A set of methods which implement the operations

Instantiation means generating objects,

Ex. 'new' operation in C++

Persistence of objects:

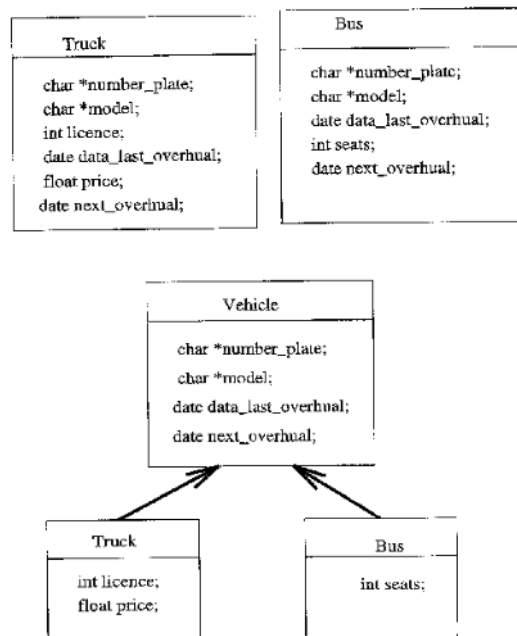
Two approaches

An implicit characteristic of all objects

An orthogonal characteristic - insert the object into a persistent collection of objects

Inheritance

A mechanism of reusability, the most powerful concept of OO programming



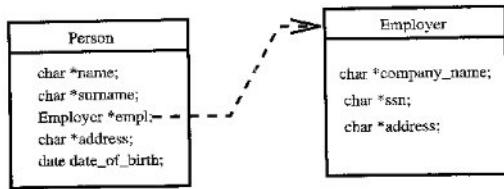
Association

Association is a link between entities in an application

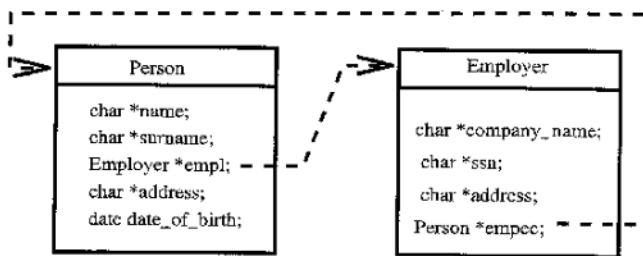
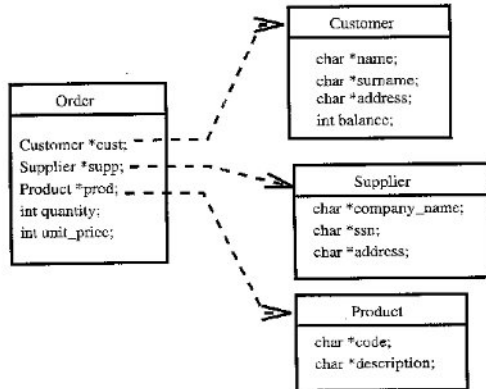
In OODB, associations are represented by means of references between objects a representation of a binary association

a representation of a ternary association

reverse reference



Representation of a binary association



ADVANTAGES OF OODB

- An integrated repository of information that is shared by multiple users, multiple products, multiple applications on multiple platforms.
- It also solves the following problems:
 1. The semantic gap: The real world and the Conceptual model is very similar.
 2. Impedance mismatch: Programming languages and database systems must be interfaced to solve application problems. But the language style, data structures, of a programming language (such as C) and the DBMS (such as

Oracle) are different. The OODB supports general purpose programming in the OODB framework.

3. New application requirements: Especially in OA, CAD, CAM, CASE, object-orientation is the most natural and most convenient.

XML Databases.

An **XML database** is a **data persistence** software system that allows data to be stored in **XML** format. These data can then be **queried**, exported and **serialized** into the desired format. XML databases are usually associated with **document-oriented databases**.

Two major classes of XML database exist:

1. **XML-enabled**: these may either map XML to traditional database structures (such as a **relational** database^[2]), accepting XML as input and rendering XML as output, or more recently support native XML types within the traditional database. This term implies that the database processes the XML itself (as opposed to relying on **middleware**).
2. **Native XML** (NXD): the internal model of such databases depends on XML and uses XML documents as the fundamental unit of storage, which are, however, not necessarily stored in the form of text files.