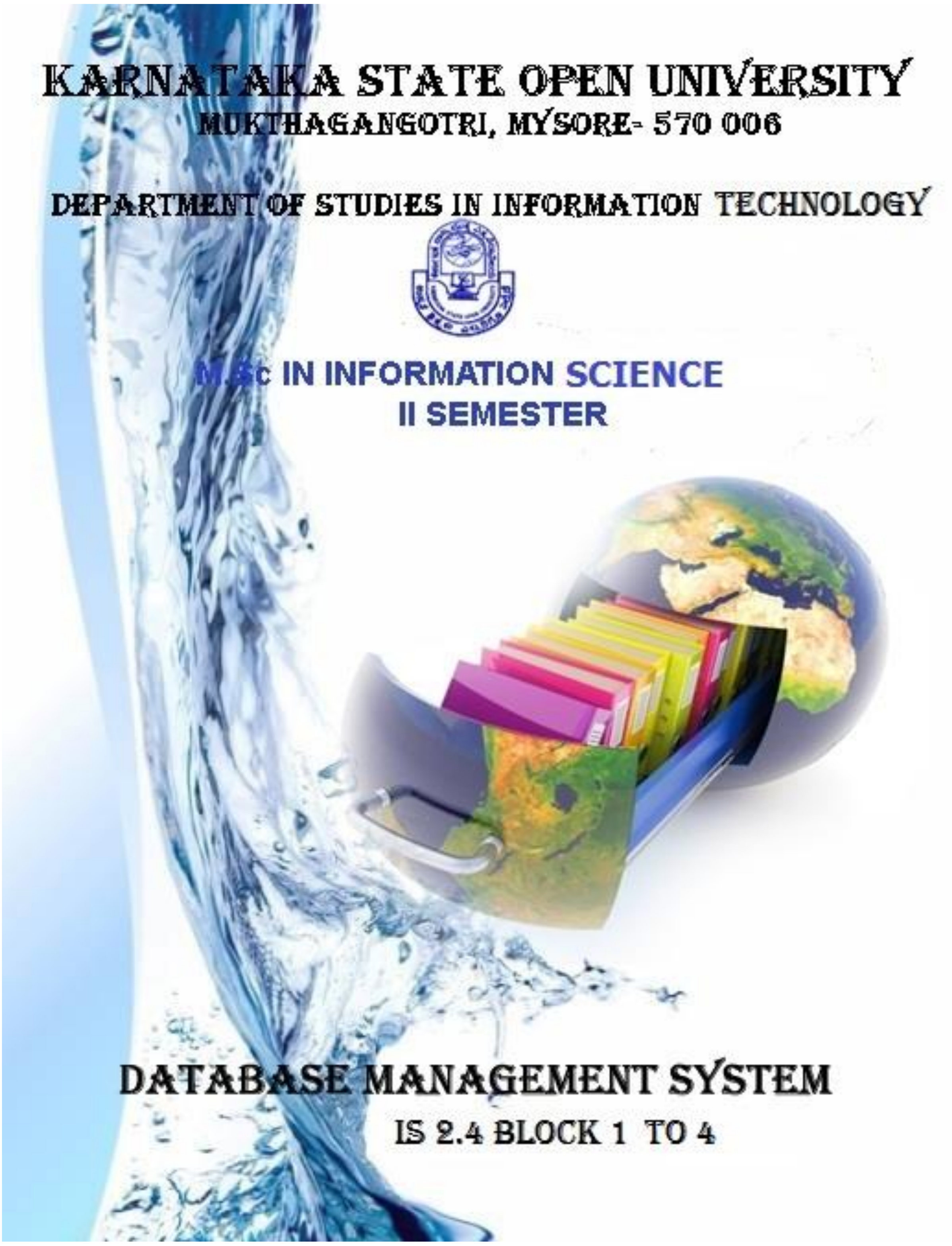
KARNATAKA STATE OPEN UNIVERSITY

MUKTHAGANGOTRI, MYSORE- 570 006

DEPARTMENT OF STUDIES IN INFORMATION TECHNOLOGY

M.Sc IN INFORMATION SCIENCE

II SEMESTER



DATABASE MANAGEMENT SYSTEM

IS 2.4 BLOCK 1 TO 4

IS 2.4

Data Base Management System

Preface

Data and information are valuable resources and which requires efficient organization to store and retrieve. This study material provides a new and comprehensive treatment of databases, dealing with the complete syllabuses for both an introductory course and an advanced course on databases. It offers a balanced view of concepts, languages and architectures, with reference to current technology on database management systems (DBMSs).

The study material is composed of four main modules and an additional module containing practical aspects. Modules I and II are designed to expose students to the principles of data management and for teaching them how to master two main skills: how to query a database (and write software that involves database access) and how to design its schema structure. These are the fundamental aspects of designing and manipulating a database that are required in order to make effective use of database technology. Modules III and IV are written in accordance to make the readers understand the concept of concurrency control, recovery query processing and optimization of database. At the end of the study material, in the last unit advanced database application related concepts are introduced to give the readers some idea.

A practical part of database management system are discuses in their respective units and also a set of practical programs are given in an manual to readers to have hands on experience in designing and maintaining a database.

Wish you happy reading.

Karnataka State Open University

Mukthagangothri, Mysore – 570 006

Second Semester M.Sc in Information Science

Data Base Management Systems

Module 3

Module 4

UNIT 1: OVERVIEW OF DBMS

Structure:

1.0 OBJECTIVES

At the end of this unit you will be able to know:

- History behind DBMS
- The problems with conventional file systems
- Advantages of DBMS
- Levels of abstraction

1.1 INTRODUCTION

A database is collection of interrelated data of an enterprise. Where as a **Database Management System** (**DBMS**) is a set of computer programs (a software) that controls the creation, maintenance, and the use of one or more database. DBMS is designed to assist in the collection and maintenance of utility data in large numbers. It is a system that makes information available when required. It allows different user application programs to easily access the same database.

DBMS contains information about a particular enterprise such as collection of interrelated data, Set of programs to access the data and an environment that is both *convenient* and *efficient* to use. Databases used to be about stuff like employee records, bank records. But today, the field also covers all the largest sources of data, with many new ideas such as:

- Web search
- Data mining
- Scientific and medical databases
- Integrating information

Databases are behind almost everything you do on the Web such as Google searches, Queries at Amazon, eBay, etc. Databases often have unique concurrency-control problems like many activities (transactions) at the database at all times.

1.2 PURPOSE OF DATABASE SYSTEM

In the early days, database applications were built directly on top of file systems. Drawbacks of using file systems to store data:

Data redundancy and inconsistency

- Different programmers will create files & application programs over a long period of time.
- Multiple file formats, duplication of information in different files.

Difficulty in accessing data

- Need to write a new program to carry out each new task.

Data isolation — multiple files and formats.

Integrity problems

- Integrity constraints (e.g. account balance > 0) become buried in program code rather than being stated explicitly.
- Hard to add new constraints or change existing ones.

Atomicity of updates

- Failures may leave database in an inconsistent state with partial updates carried out.
- Example: Transfer of funds from one account to another should either complete or not happen at all.

Concurrent access by multiple users

- Concurrent accessed needed for performance.

- Uncontrolled concurrent accesses can lead to inconsistencies.
- Example: Two people reading a balance and updating it at the same time.

Security problems

- Hard to provide user access to some, but not all, data.

Database systems offer solutions to all the above problems.

1.3 CHARACTERISTICS OF DATABASE APPROACH

Self-describing nature of a database system:

A DBMS **catalog** stores the description of a particular database (e.g. data structures, types, and constraints).The description is called **meta-data**. This allows the DBMS software to work with different database applications.

Insulation between programs and data:

Allows changing data structures and storage organization without having to change the DBMS access programs.

Data Abstraction:

A **data model** is used to hide storage details and present the users with a conceptual view of the database. Programs refer to the data model constructs rather than data storage details

Support of multiple views of the data:

Each user may see a different view of the database, which describes **only** the data of interest to that user.

Sharing of data and multi-user transaction processing:

Allowing a set of **concurrent users** to retrieve from and to update the database. *Concurrency control* within the DBMS guarantees that each **transaction** is correctly executed or aborted. *Recovery* subsystem ensures each completed transaction has its effect permanently recorded in the database. **OLTP** (Online Transaction Processing) is a major part of database applications. This allows hundreds of concurrent transactions to execute per second.

1.4 ADVANTAGE OF USING DATABASE SYSTEM

There are many advantages of DBMS. The important ones are:

- Greater flexibility
- Greater processing power
- Fits the needs of many medium to large-sized organizations
- Storage for all relevant data
- Provides user views relevant to tasks
- Ensures data integrity by managing transactions (ACID test = atomicity, consistency, isolation, durability)
- Supports simultaneous access
- Enforces design criteria in relation to data format and structure
- Provides backup and recovery
- Advanced security

The goals of a DBMS

There are many goals of DBMS. They are:

- Data storage, retrieval, and update (while hiding the internal physical implementation details)
- A user-accessible catalog
- Transaction support
- Concurrency control services (multi-user update functionality)
- Recovery services (damaged database must be returned to a consistent state)
- Authorization services (security)
- Support for data integrity services (i.e. constraints)
- Services to promote data independence
- Utility services (i.e. importing, monitoring, performance, record deletion, etc.)

The components to facilitate the goals of a DBMS may include the following:

- Query processor
- Data Manipulation Language preprocessor

- Database manager (software components to include authorization control, command processor, integrity checker, query optimizer, transaction manager, scheduler, recovery manager, and buffer manager)
- Data Definition Language compiler
- File manager
- Catalog manager

1.5 DATA ABSTRACTION

The purpose of a database system is to provide users with an abstract view of the system. The system hides certain details of how data is stored and created and maintained. The complexity is hidden from database users.

There are three levels of data abstraction: namely

1. Physical level.
2. Logical level.
3. View level.

Physical level: It deals with how the data is stored physically and where it is stored in database. This is the lowest level of abstraction.

Logical level: It describes what information or data is stored in the database (like what is the data type or what is the format of data? or the relationships among data).

Analogy: record declaration:

type *customer* = **record**

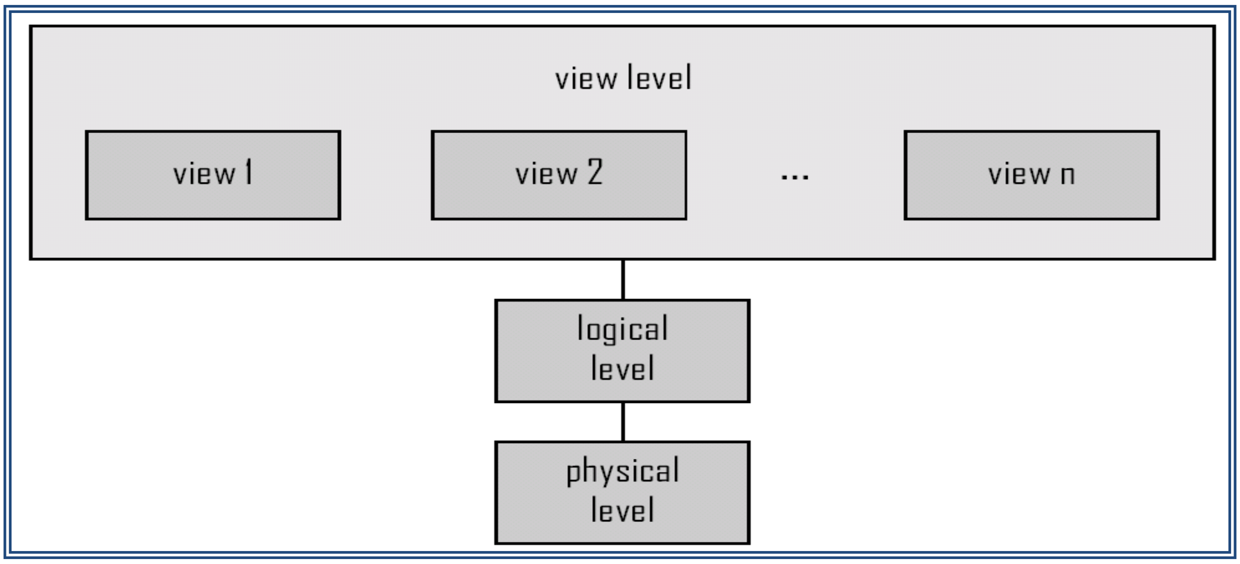
customer_id : string;
customer_name : string;
customer_street : string;
customer_city : integer;

end;

View level: Describes *part* of the database for a particular group of users. End users work on view level. There can be different views of a database. This is the highest level of abstraction.

Three Levels of Data Abstraction:

An architecture for a database system



1.6 DATA MODELS

A data model is a collection of conceptual tools for describing data, it model consists of a set of conceptual tools for *describing* data, data relationships, data semantics, and consistency requirements.

Three parts of data model are:

1. Structure of the data.

    Examples:

    - relational model = tables;
    - entitly/relationship model = entities + relationships between them.
    - semistructured model = trees/graphs.

2. Operations on data.

3. Constraints on the date.

All the data models that have been proposed can be grouped into three groups, namely: 1.object-based logical models.

2. record-based logical data models.

3. Physical data models.

Object-Based Logical Models:

Object-based logical models are used in describing data at the conceptual and view levels. They are characterized by the facts that they provide fairly flexible structuring capabilities and allow data consistency to be specified explicitly. A more widely used object-based model is the entity-relationship model. The entity-relationship model is based on a perception of a real world.

Record-Based Logical Models:

Record-based logical models are also used in describing data at the conceptual and view levels. In the record-based logical models, the database is structured in fixed format records of several types. In the record-based logical models, the three most widely accepted models are the relational, network and hierarchical models.

Physical Data Models:

The physical data models are used to describe data at the lowest level. Among a very few physical data models, unifying and frame memory are widely known. Physical data models represent the design of data while also taking into account both the constraints and facilities of a particular database management system. Generally, it is taken from a logical data model. Although it can also be engineered in reverse from a particular database implementation.

Physical data model represents how the model will be built in the database. A physical database model shows all table structures, including column name, column data type, column constraints, primary key, foreign key, and relationships between tables. Features of a physical data model include:

- Specification of all tables and columns.
- Foreign keys are used to identify relationships between tables.
- Denormalization may occur based on user requirements.
- Physical considerations may cause the physical data model to be quite different from the logical data model.

Physical data model will be different for different RDBMS.

1.7 INSTANCES AND SCHEMA

In DBMS Instance is the information stored in the Database at a particular moment and Schema is the overall Design of the Database.

INSTANCES:

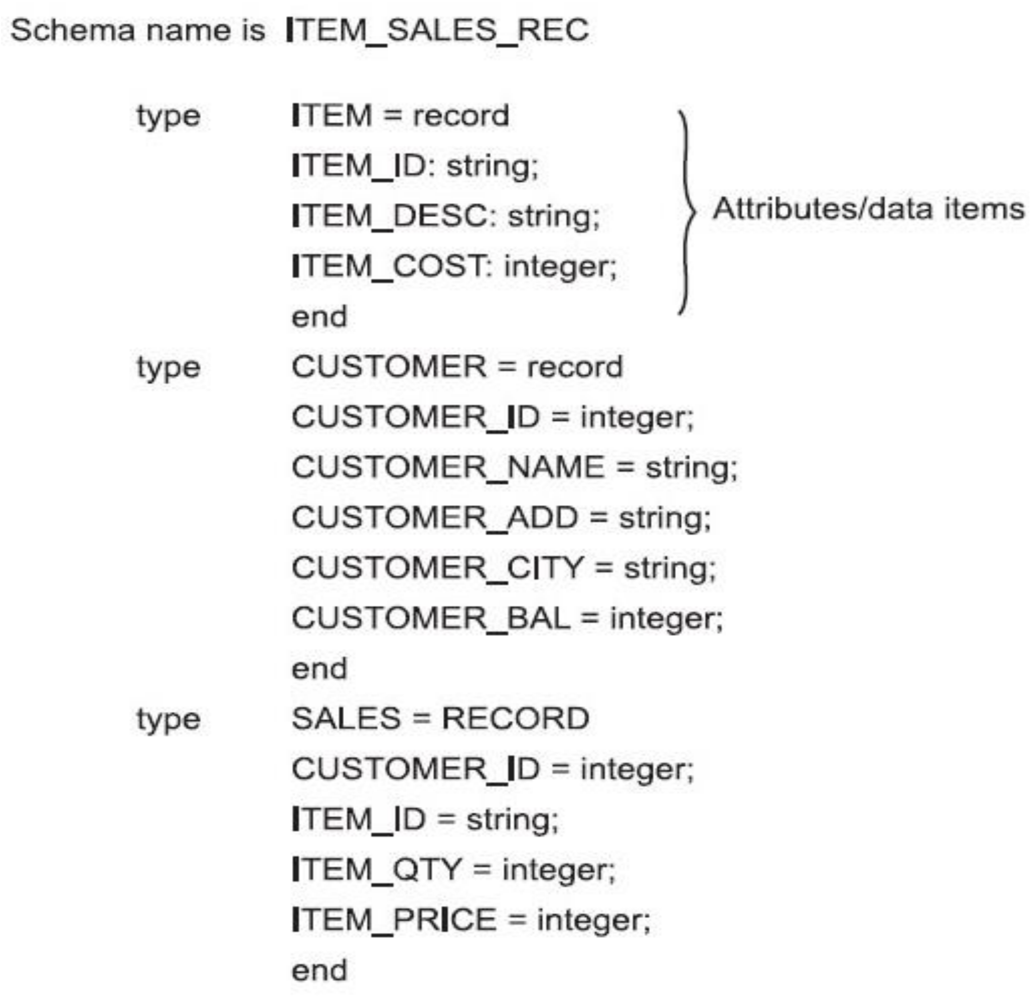
The data in the database at a particular moment of time is called an instance or a database state. In a given instance, each schema construct has its own current set of instances. Many instances or database states can be constructed to correspond to a particular database schema. Every time we update (i.e., insert, delete or modify) the value of a data item in a record, one state of the database changes into another state.

The following figure shows an instance of the ITEM relation in a database schema.

ITEM		
ITEM-ID	**ITEM_DESC**	**ITEM_COST**
1111A	Nutt	3
1112A	Bolt	5
1113A	Belt	100
1144B	Screw	2

SCHEMA:

A schema is plan of the database that give the names of the entities and attributes and the relationship among them. A schema includes the definition of the database name, the record type and the components that make up the records. Alternatively, it is defined as a framework into which the values of the data items are fitted. The values fitted into the frame-work changes regularly but the format of schema remains the same e.g., consider the database consisting of three files ITEM, CUSTOMER and SALES. The data structure diagram for this schema is shown in following figure:



```
Schema name is  ITEM_SALES_REC

          type       ITEM = record
                     ITEM_ID: string;
                     ITEM_DESC: string;          Attributes/data items
                     ITEM_COST: integer;
                     end
          type       CUSTOMER = record
                     CUSTOMER_ID = integer;
                     CUSTOMER_NAME = string;
                     CUSTOMER_ADD = string;
                     CUSTOMER_CITY = string;
                     CUSTOMER_BAL = integer;
                     end
          type       SALES = RECORD
                     CUSTOMER_ID = integer;
                     ITEM_ID = string;
                     ITEM_QTY = integer;
                     ITEM_PRICE = integer;
                     end
```

Generally, a schema can be partitioned into two categories, i.e.,

1. Logical schema
2. Physical schema.

The logical schema is concerned with exploiting the data structures offered by the DBMS so that the schema becomes understandable to the computer. It is important as programs use it to construct applications.

The physical schema is concerned with the manner in which the conceptual database get represented in the computer as a stored database. It is hidden behind the logical schema and can usually be modified without affecting the application programs.

The DBMS's provide DDL and DSDL to specify both the logical and physical schema.

Subschema:

subschema is a subset of the schema having the same properties that a schema has. It identifies a subset of areas, sets, records, and data names defined in the database schema available to user sessions. The subschema allows the user to view only that part of the database that is of interest to him. The subschema defines the portion of the database as seen by the application programs and the application programs can have different view of data stored in the database.

The different application programs can change their respective subschema without affecting other's subschema or view.

1.8 DATA INDEPENDENCE

Separation of data from processing, either so that changes in the size or format of the data elements require no change in the computer programs processing them or so that these changes can be made automatically by the database management system.

These are the techniques that allow data to be changed without affecting the applications that process it. There are two kinds of data independence. The first type is data independence for data, which is accomplished in a database management system (DBMS). It allows the database to be structurally changed without affecting most existing programs. Programs access data in a DBMS by field and are concerned with only the data fields they use, not the format of the complete record. Thus, when the record layout is updated (fields added, deleted or changed in size), the only programs that must be changed are those that use those new fields.

The ability to modify a scheme definition in one level without affecting a scheme definition in a higher level is called data independence.

There are two kinds:

1. Logical data independence.
2. Physical data independence

Logical data independence:

- The ability to modify the conceptual scheme without causing application programs to be rewritten.

- Immunity of external schemas to changes in the conceptual schema.
- Usually done when logical structure of database is altered.

Physical data independence:

- The ability to modify the internal scheme without having to change the conceptual or external schemas.
- Modifications at this level are usually to improve performance.

1.9 DATABASE LANGUAGE

The database is an intermediate link between the physical database, computer and the operating system and the users. To provide the various facilities to different types of users, a DBMS normally provides one or more specialized programming languages called database languages.

The DBMS mainly provides two database languages, namely, data definition language and data manipulation language to implement the databases. Data definition language (DDL) is used for defining the database schema. The DBMS comprises DDL compiler that identifies and stores the schema description in the DBMS catalog. Data manipulation language (DML) is used to manipulate the database.

Data Description Language (DDL):

As the name suggests, this language is used to define the various types of data in the database and their relationship with each other. In DBMSs where no strict separation between the levels of the database is maintained, the data definition language is used to define the conceptual and internal schemas for the database. On the other hand, in DBMSs, where a clear separation is maintained between the conceptual and internal levels, the DDL is used to specify the conceptual schema only. In such DBMSs, a separate language, namely, storage definition language (SDL) is used to define the internal schema. Some of the DBMSs that are based on true three-schema architecture use a third language, namely, view definition language (VDL) to define the external schema.

The DDL statements are also used to specify the integrity rules (constraints) in order to maintain the integrity of the database. The various integrity constraints are domain constraints, referential integrity, assertions and authorization. These constraints are discussed in detail in subsequent chapters. Like any other programming language, DDL also accepts input in the form of instructions (statements) and generates the description of schema as output. The output is placed in the data dictionary, which is a special type of table containing metadata. The DBMS refers the data dictionary before reading or modifying the data. Note that the database users cannot update the data dictionary; instead it is only modified by database system itself.

The basic functions performed by DDL are:

- Create tables, files, databases and data dictionaries.
- Specify the storage structure of each table on disk.
- Integrity constraints on various tables.
- Security and authorization information of each table.
- Specify the structure of each table.
- Overall design of the Database.

Data Manipulation Language (DML):

A language that enables users to access or manipulate data (retrieve, insert, update, delete) as organized by a certain Data Model is called the Data Manipulation Language (DML).

Once the database schemas are defined and the initial data is loaded into the database, several operations such as retrieval, insertion, deletion, and modification can be applied to the database. The DBMS provides data manipulation language (DML) that enables users to retrieve and manipulate the data. The statement which is used to retrieve the information is called a **query**. The part of the DML used to retrieve the information is called a query language. However, query language and DML are used synonymously though technically incorrect. The DML are of two types, namely, *non-procedural DML* and *procedural DML*.

The non-procedural or high-level or declarative DML enables to specify the complex database operations concisely. It requires a user to specify *what* data is required without specifying *how* to retrieve the required data. For example, SQL (Structured Query Language) is a non-procedural query language as it enables user to easily define the structure or modify the data in the database without specifying the details of *how* to manipulate the database. The high-level DML statements can either be entered interactively or embedded in a general purpose programming language. On the other hand, the procedural or low-level DML requires user to specify *what* data is required and *how* to access that data by providing step-by-step procedure. For example, relational algebra is procedural query language, which consists of set of operations such as select, project, union, etc., to manipulate the data in the database.

1.10 DATABASE MANAGER

Database Manager is a part of the database management system (DBMS) that handles the organisation, storage and retrieval of the data. A database manager may work with traditional programming languages, such as COBOL and BASIC, or may work only with its proprietary programming language

A database manager links two or more files together and is the foundation for developing routine business systems. Contrast with file manager, which works with only one file at a time and is typically used interactively on a personal computer for managing personal, independent files, such as name and address lists.

1.11 DATABASE ADMINISTRATOR

The people responsible for managing databases are called database administrators. Each database administrator, dubbed DBA for the sake of brevity, may be engaged in performing various database manipulation tasks such as archiving, testing, running, security control, etc., all related to the environmental side of the databases. Their job is very important, since in today's world, almost all of the information a company uses is kept in databases. Due to the fact that most databases use a different approach to storing and handling data, there are different types of DBAs. Most of the major vendors who provide database solutions also offer courses to certify the DBA.

Duties of Data Administrator:

The exact set of database administration duties of each DBA is dependent on his/her job profile, the IT policies applied by the company he/she works for and last but not least - the concrete parameters of the database management system in use. A DBA must be able to think logically to solve all problems and to easily work in a team with both DBA colleagues and staff with no computer training.

1. Frequent updates:

Some of the basic database management tasks of a DBA supplement the work of a system administrator and include hardware/ software configuration, as well as installation of new DBMS software versions. Those are very important tasks since the proper installation and configuration of the database management software and its regular updates are crucial for the optimal functioning of the DBMS and hence - of the databases, since the new releases often contain bug fixes and security updates.

2. Data analysis and security

Database security administration and data analysis are among the major duties of a DBA. He/she is responsible for controlling the DBMS security by adding and removing users, managing database quotas and checking for security issues. The DBA is also engaged in analyzing database contents and improving the data storage efficiency by optimizing the use of indexes, enabling the 'Parallel Query' execution, etc.

3. Database design

Apart from the tasks related to the logical and the physical side of the database management process, DBAs may also take part in database design operations. Their main role is to give developers recommendations about the DBMS specificities, thus helping them avoid any eventual database performance issues. Other important tasks of the DBAs are related to data modelling aimed at optimizing the system layout, as well as to the analysis and creation of new databases.

1.12 DATABASE USER

Users are differentiated by the way they expect to interact with the system.

- Application programmers: interact with system through DML calls.
- Specialized users: write specialized database applications that do not fit into the traditional data processing framework.
- Sophisticated users: form requests in a database query language
- Naive users: invoke one of the permanent application programs that have been written previously.

1.13 SUMMARY

A database is collection of interrelated data of an enterprise. Where as a **Database Management System** (**DBMS**) is a set of computer programs (a software) that controls the creation, maintenance, and the use of one or more database. In this unit, we considered the historical perspective of DBMS. We also discussed the problems with conventional file processing compared to DBMS. We ended with levels of data abstraction.

1.14 KEYWORD

Database: A database is collection of interrelated data of an enterprise
DBMS: DBMS stands for **Database Management System**, is a set of computer programs.
4GL: It stands for Fourth-generation programming language.
Abstraction: hiding certain details of how data is stored and created and maintained.

1.15 UNIT-END EXERCISES AND ANSWERS

1. What is the about of database system?
2. Give a brief explanation about characteristic of database approach?
3. What are the advantages of DBMS?
4. What are the goals of DBMS?
5. Discuss levels of abstraction?
6. Explain about Database languages?
7. What is the role of Database manager, Database administrator and Database user?

Answers: SEE

1. 1.2
2. 1.3
3. 1.4
4. 1.4
5. 1.5
6. 1.9
7. 1.10, 1.11, 1.12

1.16 SUGGESTED READING

1. Fundamentals of Database Systems By Ramez Elmasri, Shamkant B. Navathe, Durvasula V.L.N. Somayajulu, Shyam K.Gupta
2. Database System Concepts By Avi Silberschatz, Henry F. Korth , S. Sudarshan
3. Database Management Systems By Raghu Ramakrishnan and Johannes Gehrke

UNIT 2: DATA MODELLING

Structure:

2.0 OBJECTIVES

At the end of this unit you will be able to know:

- About Entity sets, Attributes , keys and Relationships
- About data modelling
- Type role and structural constraints and Weak and Strong entity types
- About Entity-Relationship Diagram, its design and EER.

2.1 INTRODUCTION

Data modelling is a method used to define and analyze data requirements needed to support the business processes of an organization and by defining the data structures and the relationships between data elements.

Data modelling techniques are used:

- to manage data as a resource (migrate, merge, data quality…)

- for designing computer databases.
- to better cope with change, by allowing to make changes into the model, that will automatically induce changes in the database and programs

.

A data model is composed of three levels of modelling:

The semantic model: describes the semantics of a domain (for example, it may be a model of the interest area of an organization or industry), i.e. define the meaning of data within the context of its interrelationships and constraints with other data. It is an abstraction which defines how the stored symbols relate to the real world. Thus, the semantic model must be a true representation of the real world. A semantic model consists of entity classes, representing kinds of things of significance in the domain, and relationships assertions about associations between pairs of entity classes. A semantic model specifies the kinds of facts or propositions that can be expressed using the model. In that sense, it defines the allowed expressions in an artificial 'language' with a scope that is limited by the scope of the model.

The logical model: describes the system information, as represented by a particular data manipulation technology type: e.g. flat file system, hierarchical DBMS(IMS,...), network DBMS (IDMS, IDS2,..), relational DBMS (DB2, ORACLE, SQL SERVER,...).

A logical model: consists of descriptions of entities (called « segments » in hierarchical DBMS, « records » in network DBMS, « tables » in relational DBMS) and attributes (called « data » inhierarchical and network DBMS, « columns » in relational DBMS), data access keys, type of links beetween entities (called « sets » in network DBMS, « foreigns keys » in relational DBMS) among other things;

The physical model: describes the physical means by which data are stored in a particular DBMS product (flat files, XML files, IMS, IDS2, IDMS, ORACLE, DB2, ...) . This is Concerned with partitions, CPUs, table spaces and the like.

2.2 ENTITY SET

Entity is a thing in the real world with an independent existence and entity set is collection or set all entities of a particular entity type at any point of time.

An entity may be concrete (a person or a book, for example) or abstract (like a holiday or a concept).

An entity set is a set of entities of the same type (e.g., all persons having an account at a bank).

Entity sets need not be disjoint. For example, the entity set employee (all employees of a bank) and the entity set customer (all customers of the bank) may have members in common. An entity is represented by a set of attributes.

An *entity set* is a logical container for instances of an entity type and instances of any type derived from that entity type. The relationship between an entity type and an entity set is analogous to the relationship between a row and a table in a relational database: Like a row, an entity type describes data structure, and, like a table, an entity set contains instances of a given structure. An entity set is not a data modelling construct; it does not describe the structure of data. Instead, an entity set provides a construct for a hosting or storage environment (such as the common language runtime or a SQL Server database) to group entity type instances so that they can be mapped to a data store.

An entity set is defined within an entity container, which is a logical grouping of entity sets and association sets.

For an entity type instance to exist in an entity set, the following must be true:

- The type of the instance is either the same as the entity type on which the entity set is based, or the type of the instance is a subtype of the entity type.
- The entity key for the instance is unique within the entity set.
- The instance does not exist in any other entity set.

2.3 ATTRIBUTES AND KEYS

In general, an attribute is a property or characteristic. Color, for example, is an attribute of your hair. In programming computers, an attribute is a changeable property or characteristic of some component of a program that can be set to different values.

In a database management system (DBMS), an attribute may describe a component of the database, such as a table or a field, or may be used itself as another term for a field.

For example, below Table contains one or more columns. These column are the Attribute in DBMS. A table named "EmployeeInfo" which have the following columns: ID, Name, Address,then "ID", "Name" and "Address" are the attributes of EmployeeInfo table.

ID	Name	Adress
01	Steve	Steve street,newyork
02	John	John villa,londan
03	Merry	Food street,bankok
04	Screw	Hill station road,japan

Types of attributes:

- Composite versus simple (atomic) attributes.
- Single-valued versus multivalued attributes.
- Stored versus derived attributes.
- NULL values.
- Complex attributes.

The key is defined as the column or attribute of the database table. For example if a table has id, name and address as the column names then each one is known as the key for that table. We can also say that the table has 3 keys as id, name and address. The keys are also used to identify each record in the database table. The following are the various types of keys available in the DBMS system.

1. A **simple** key contains a single attribute.
2. A **composite key** is a key that contains more than one attribute.
3. A **candidate key** is an attribute (or set of attributes) that uniquely identifies a row. A candidate key must possess the following properties:
   - Unique identification - For every row the value of the key must uniquely identify that row.
   - Non redundancy - No attribute in the key can be discarded without destroying the property of unique identification.
4. A **primary key** is the candidate key which is selected as the principal unique identifier. Every relation must contain a primary key. The primary key is usually the key selected to identify a row when the database is physically implemented. For example, a part number is selected instead of a part description.

5. A **superkey** is any set of attributes that uniquely identifies a row. A superkey differs from a candidate key in that it does not require the non redundancy property.

6. A **foreign key** is an attribute (or set of attributes) that appears (usually) as a non key attribute in one relation and as a primary key attribute in another relation. I say *usually* because it is possible for a foreign key to also be the whole or part of a primary key.

2.3 RELATIONSHIPS

In DBMS, a relationship, in the context of databases, is a situation that exists between two relational database tables when one table has a foreign key that references the primary key of the other table. Relationships allow relational databases to split and store data in different tables, while linking disparate data items.

For Example, in a bank database a CUSTOMER_MASTER table stores customer data with a primary key column named CUSTOMER_ID; it also stores customer data in an ACCOUNTS_MASTER table, which holds information about various bank accounts and associated customers. To link these two tables and determine customer and bank account information, a corresponding CUSTOMER_ID column must be inserted in the ACCOUNTS_MASTER table, referencing existing customer ids from the CUSTOMER_MASTER table. In this case, the ACCOUNTS_MASTER table's CUSTOMER_ID column is a foreign key that references a column with the same name in the CUSTOMER_MASTER table. This is an example of a relationship between the two tables.

Several types of relationships can be defined in a database. Consider the possible relationships between employees and departments.

One-to-many and many-to-one relationships

An employee can work in only one department; this relationship is *single-valued* for employees. On the other hand, one department can have many employees; this relationship is *multi-valued* for departments. The relationship between employees (single-valued) and departments (multi-valued) is a *one-to-many* relationship.

To define tables for each one-to-many and each many-to-one relationship:

1. Group all the relationships for which the "many" side of the relationship is the same entity.

2. Define a single table for all the relationships in the group.

In the following example, the "many" side of the first and second relationships are "employees" so an employee table, EMPLOYEE, is defined.

Entity	Relationship	Entity
Employees	are assigned to	Departments
Employees	work at	Jobs
Departments	report to	(administrative) departments

In the third relationship, "departments" is on the "many" side, so a department table, DEPARTMENT, is defined.

The following tables show these different relationships.

The EMPLOYEE table:

EMPNO	WORKDEPT	JOB
000010	A00	President
000020	B01	Manager
000120	A00	Clerk
000130	C01	Analyst
000030	C01	Manager
000140	C01	Analyst
000170	D11	Designer

The DEPARTMENT table:

DEPTNO	ADMRDEPT
C01	A00
D01	A00
D11	D01

Many-to-many relationships

A relationship that is multi-valued in both directions is a many-to-many relationship. An employee can work on more than one project, and a project can have more than one employee. The questions "What does Dolores Quintana work on?", and "Who works on project IF1000?" both yield multiple answers. A many-to-many relationship can be expressed in a table with a column for each entity ("employees" and "projects"), as shown in the following example.

The following table shows how a many-to-many relationship (an employee can work on many projects, and a project can have **7**many employees working on it) is represented.

The employee activity (EMP_ACT) table:

EMPNO	**PROJNO**
000030	IF1000
000030	IF2000
000130	IF1000
000140	IF2000
000250	AD3112

One-to-one relationships

One-to-one relationships are single-valued in both directions. A manager manages one department; a department has only one manager. The questions, "Who is the manager of Department C01?", and "What department does Sally Kwan manage?" both have single answers. The relationship can be assigned to either the DEPARTMENT table or the EMPLOYEE table. Because all departments have managers, but not all employees are managers, it is most logical to add the manager to the DEPARTMENT table, as shown in the following example.

The following table shows the representation of a one-to-one relationship.

The DEPARTMENT table:

DEPTNO	**MGRNO**

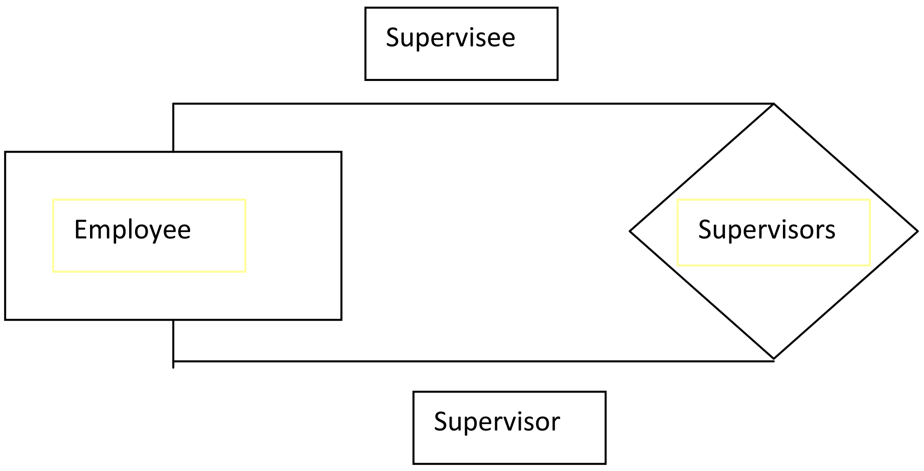
DEPTNO	MGRNO
A00	000010
B01	000020
D11	000060

2.5 RELATIONSHIPS, ROLES AND STRUCTURAL CONSTRAINTS

A relationship: A relationship is an association (combination) among the instance of one or more entity type.

Role Names: Each entity type in a relationship plays a particular role. The role name specifies the role that a participating entity type plays in the relationship and explains what the relationship means. For example, in the relationship between Employee and Department, the Employee entity type plays the employee role, and the Department entity type plays the department role. In most cases the role names do not have to be specified, but in cases where the same entity participates more than once in a relationship type in different roles, then we have to specify the role.

For example, in the Company schema, each employee has a supervisor. We need to include the relationship "Supervises". However a supervisor is also an employee. Therefore the employee entity type participates twice in the relationship, once as an employee and once as a supervisor. Therefore we can specify two roles, employee and supervisor.



Constraints on Relationship Types:

Relationship types have certain constraints that limit the possible combination of entities that may participate in relationship. An example of a constraint is that if we have the entities Doctor and Patient, the organization may have a rule that a patient cannot be seen by more than one doctor. This constraint needs to be described in the schema.

There are two main types of relationship constraints, cardinality ratio, and participation.

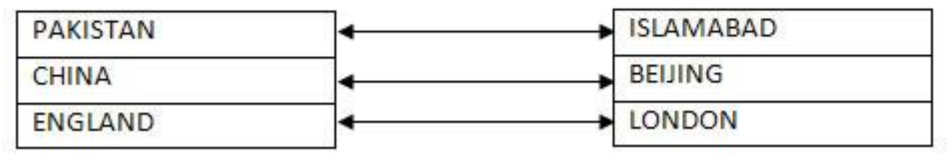
Relationship types:

Types of Relationships in RDBMS There are three types of relationships:

1. One to One relationship
2. One to Many (or Many to One) relationship
3. Many to Many relationship.

One to One relationship:

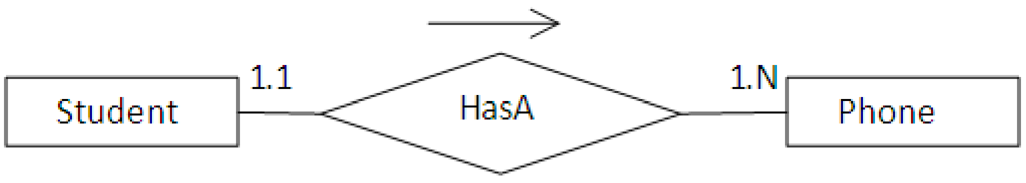
In a one-to-one relationship, each record in Table A can have only one matching record in Table B, and each record in Table B can have only one matching record in Table A



One to Many (or Many to One) relationship:

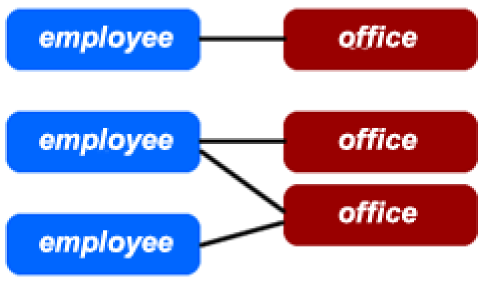
In a one-to-many relationship, a record in Table A can have many matching records in Table B, but a record in Table B has only one matching record in Table A

Consider the following relationship between *Student* and *Phone* entity. According to the relationship a student can have any number of phone numbers.



Many to Many relationship:

In a many-to-many relationship, a record in Table A can have many matching records in Table B, and a record in Table B can have many matching records in Table A



2.6 WEAK AND STRONG ENTITY TYPES

The ER model consists of different types of entities. The existence of an entity may depends on the existence of one or more other entities, such an entity is said to be existence dependent. Entities whose existence not depending on any other entities is termed as not existence dependent. Entities based on their characteristics are classified as follows.

- Strong Entities
- Weak Entities

In a relational database, an entity set that does not possess sufficient attributes to form a primary key is called a weak entity set. One which has a primary key is called a strong entity set.

The discriminator (or partial key) of a weak entity set is the set of attributes that distinguishes among all the entities of a weak entity set

The existence of a weak entity set depends on the existence of a identifying entity set. It must relate to the identifying entity set via a total, one-to-many relationship set from the identifying to the weak entity set

We depict a weak entity set by double rectangles in E-R diagram. We underline the discriminator of a weak entity set with a dashed line in E-R diagram. a weak entity is an entity that cannot be uniquely identified by its attributes alone; therefore, it must use a foreign key in conjunction with its attributes to create a primary key. The foreign key is typically a primary key of an entity it is related to.

In entity relationship diagrams a weak entity set is indicated by a bold (or double-lined) rectangle (the entity) connected by a bold (or double-lined) type arrow to a bold (or double-lined) diamond (the relationship). This type of relationship is called an identifying relationship and in IDEF1X notation it is represented by an oval entity rather than a square entity for base tables. An identifying relationship is one where the primary key is populated to the child weak entity as a primary key in that entity.

In general (though not necessarily) a weak entity does not have any items in its primary key other than its inherited primary key and a sequence number. There are two types of weak entities: associative entities and subtype entities. The latter represents a crucial type of normalization, where the super-type entity inherits its attributes to subtype entities based on the value of the discriminator.

A weak entity is one that can only exist when owned by another one. For example: a ROOM can only exist in a BUILDING. On the other hand, a TIRE might be considered as a strong entity because it also can exist without being attached to a CAR.

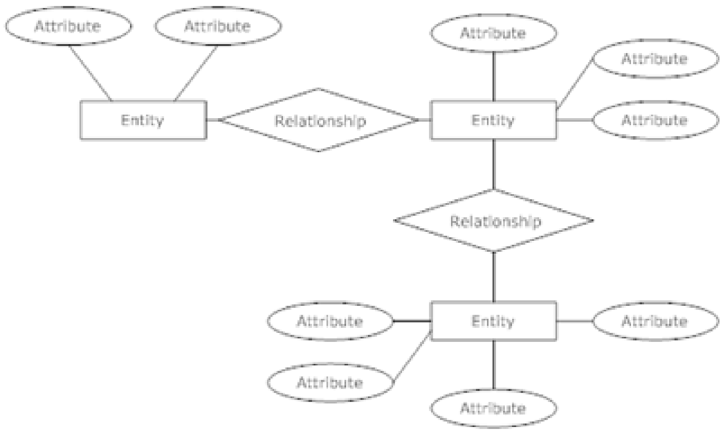
2.7 ENTITY RELATIONSHIP DIAGRAM

Entity relationship diagram is a graphical representation of entities and their relationships to each other, typically used in computing in regard to the organization of data within databases or information systems. An entity is a piece of data-an objector concept about which data is stored. A relationship is how the data is shared between entities. There are three types of relationships between entities:

- one-to-one: one instance of an entity (A) is associated with one other instance of another entity (B). For example, in a database of employees, each employee name (A) is associated with only one social security number (B).
- one-to-many: one instance of an entity (A) is associated with zero, one or many instances of another entity (B), but for one instance of entity B there is only one instance of entity A. For example, for a company with all employees working in one building, the building name (A) is associated with many different employees (B), but those employees all share the same singular association with entity A.
- many-to-many: one instance of an entity (A) is associated with one, zero or many instances of another entity (B), and one instance of entity B is associated with one, zero or many instances of entity A. For example, for a company in which all of its employees work on multiple projects, each instance of an employee (A) is associated

with many instances of a project (B), and at the same time, each instance of a project (B) has multiple employees (A) associated with it.

Entity Relationship Diagrams (ERDs) illustrate the logical structure of databases.

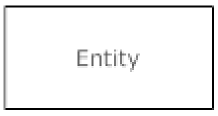


An ER Diagram

Entity Relationship Diagram Notations

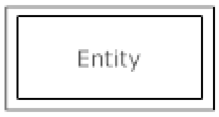
Entity

An entity is an object or concept about which you want to store information.



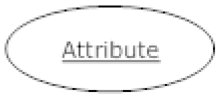
Weak Entity

A weak entity is an entity that must defined by a foreign key relationship with another entity as it cannot be uniquely identified by its own attributes alone.



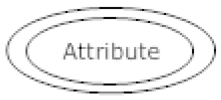
Key attribute

A key attribute is the unique, distinguishing characteristic of the entity. For example, an employee's social security number might be the employee's key attribute.



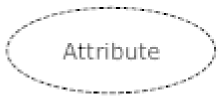
Multivalued attribute

A multivalued attribute can have more than one value. For example, an employee entity can have multiple skill values.



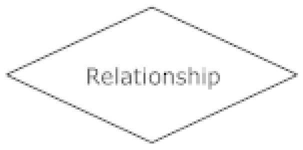
Derived attribute

A derived attribute is based on another attribute. For example, an employee's monthly salary is based on the employee's annual salary.



Relationships

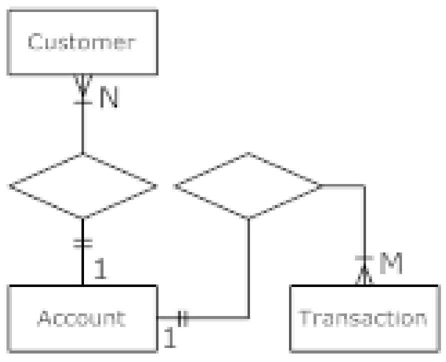
Relationships illustrate how two entities share information in the database structure.



Cardinality

Cardinality specifies how many instances of an entity relate to one instance of another entity.

Ordinality is also closely linked to cardinality. While cardinality specifies the occurrences of a relationship, ordinality describes the relationship as either mandatory or optional. In other words, cardinality specifies the maximum number of relationships and ordinality specifies the absolute minimum number of relationships.



2.8 DESIGN OF AN E-R DATABASE SCHEMA

The E-R data model provides a wide range of choice in designing a database scheme to accurately model some real-world situation.

Some of the decisions to be made are

- Using a ternary relationship versus two binary relationships.
- Whether an entity set or a relationship set best fit a real-world concept.
- Whether to use an attribute or an entity set.
- Use of a strong or weak entity set.
- Appropriateness of generalization.
- Appropriateness of aggregation

Database Design Process:

Logical Design Steps:

- It is helpful to divide the design into two steps:
- Restructuring of the Entity Relationship schema, based on criteria for the optimization of the schema and the simplification of the following step;
- Translation into the logical model, based on the features of the logical model.

Performance Analysis:

- An ER schema is restructured to optimize:

Cost of an operation is evaluated in terms of the number of occurrences of entities and relationships that are visited during the execution of an operation.

- Storage requirements. Evaluated in terms of number of bytes necessary to store the data described by the schema.
- In order to study these parameters, we need to know:
- Projected volume of data;
- Projected operation characteristics.

Analysis of Redundancies:

- A redundancy in a conceptual schema corresponds to a piece of information that can be derived (that is, obtained through a series of retrieval operations) from other data in the database.
- An Entity-Relationship schema may contain various forms of redundancy.

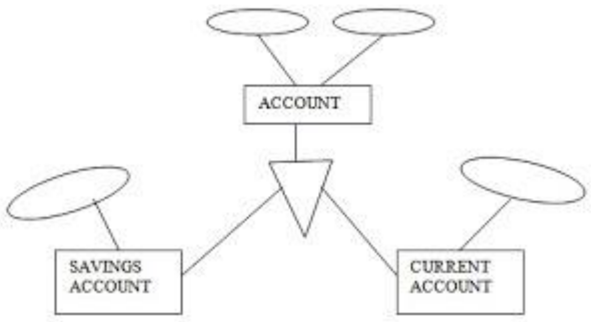
2.9 OBJECT MODELLING

An object relational database management system (ORDBMS) is a database management system with that is similar to a relational database, except that it has an object-oriented database model. This system supports objects, classes and inheritance in database schemas and query language.

Object relational database management systems provide a middle ground between relational and object-oriented databases. In an ORDBMS, data is manipulated using queries in a query language. These systems bridge the gap between conceptual data modelling techniques such as entity relationship diagrams and object relational mapping using classes and inheritance. ORDBMSs also support data model extensions with custom data types and methods. This allows developers to raise the abstraction levels at which problem domains are viewed.

2.10 SPECIALIZATION AND GENERALIZATION

Specialization is a top-down design process, whereas generalization is a bottom-up design process. Which means you will first design the sub groupings like officer; temp-staff etc and slowly move upwards to 'employer' - 'customer' and then design the 'person' entity. In the ER diagram generalization and specialization are both represented exactly same. Consider the account entity set with attributes accno and balance. Each account can be classified as SAVINGS ACCOUNT or CURRENT ACCOUNT. Each of these is described by a set of attributes which include all the attributes of the account entity set PLUS some additional attributes , SAVINGS ACCOUNT entities are described by the attribute INTEREST RATE ,while CURRENT ACCOUNT are described by the attribute OVER DRAFT AMOUNT. There are similarities between the current account entity set and the saving account entity set in the sense that they have several attributes in common. This commonality can be expressed by generalization. Generalization is a containment relationship that exists between a higher level entity set and one or lower level entity sets .Here the ACCOUNT is the higher level entity and SAVINGS ACCOUNT & CURRENT ACCOUNT are LOWERLEVEL ENTITY SETS. In an ER diagram generalization is represented by a triangular component labelled “ISA" it stands for "is a ". For example Savings Account "is an " account .The attributes of higher level entity sets are inherited by the lower level entity sets.



ISA relationship between ACCOUNTS - SAVINGS ACCOUNT & CURRENT ACCOUNT There are two methods for transforming an ER diagram, which includes generalization, to a tabular form.

FIRST METHOD - Create a table for the higher level entity set. For each lower level entity set create a table which includes columns for each of the attributes of that entity set, plus columns for each of the attributes of the primary key of the higher level entity set.

Then we will have three tables for the above ER diagram.

a) Account with columns accno & balance
b) Savings account with columns accno & interest rate
c) Current account with columns accno. & overdraft amount

SECOND METHOD - Do not create a table for higher level entity set. For each lower level entity set create a table which includes columns for each of the attributes of that entity set, plus columns for each of the attributes of the higher level entity set.
Then we will have two tables for the above ER diagram.

a) Savings account with columns accno, balance & interest rate
b) Current Account with columns accno, balance & overdraft amount.

2.11 ENHANCED ENTITY RELATIONSHIP (EER)

Enhanced entity relationship model Includes all modelling concepts of basic ER .an additional concepts: subclasses/superclasses, specialization/generalization, categories, attribute inheritance and the resulting model is called the enhanced-ER or Extended ER (E2R or EER) model.this model is used to model applications more completely and accurately if needed also includes some object-oriented concepts, such as inheritance.

An entity type may have additional meaningful sub-groupings of its entities. For Example: EMPLOYEE may be further grouped into SECRETARY, ENGINEER, MANAGER, TECHNICIAN, SALARIED_EMPLOYEE, HOURLY_EMPLOYEE,…

Each of these groupings is a subset of EMPLOYEE entities

Each is called a subclass of EMPLOYEE

EMPLOYEE is the superclass for each of these subclasses

These are called superclass/subclass relationships.

Example: EMPLOYEE/SECRETARY, EMPLOYEE/TECHNICIAN.

Formal Definitions of EER Model (1):

- Class C: A set of entities; could be entity type, subclass, superclass, category.
- Subclass S: A class whose entities must always be subset of the entities in another class, called the superclass C of the superclass/subclass (or IS-A) relationship S/C:

  $S \subseteq C$

- Specialization Z: Z = {S1, S2,…, Sn} a set of subclasses with same superclass G; hence, G/Si a superclass relationship for i = 1, …., n.

  G is called a generalization of the subclasses {S1, S2,…, Sn}

  Z is total if we always have:

  $S1 \cup S2 \cup \ldots \cup Sn = G$;
  Otherwise, Z is partial.

  Z is disjoint if we always have:

  $Si \cap S2$ empty-set for $i \neq j$;

  Otherwise, Z is overlapping.

Formal Definitions of EER Model (2):

- Subclass S of C is predicate defined if predicate p on attributes of C is used to specify membership in S; that is, S = C[p], where C[p] is the set of entities in C that satisfy p
- A subclass not defined by a predicate is called user-defined

- Attribute-defined specialization: if a predicate A = ci (where A is an attribute of G and ci is a constant value from the domain of A) is used to specify membership in each subclass Si in Z
- Note: If ci ≠ cj for i ≠ j, and A is single-valued, then the attribute-defined specialization will be disjoint.
- Category or UNION type T

  A class that is a subset of the union of n defining superclasses
  D1, D2,…Dn, n>1:

  $T \subseteq (D1 \cup D2 \cup \ldots \cup Dn)$
  A predicate pi on the attributes of T.

  If a predicate pi on the attributes of Di can specify entities of Di that are members of T.

  If a predicate is specified on every Di: $T = (D1[p1] \cup D2[p2] \cup \ldots \cup Dn[pn]$

  Note: The definition of relationship type should have 'entity type' replaced with 'class'.

2.12 SUMMARY

Data modeling in software engineering is the process of creating a data model by applying formal data model descriptions using data modelling techniques. This unit introduced data models and some related terminologies. The data requirements are recorded as a conceptual data model with associated data definitions. We also defined relationships, roles and structural constraints in this unit.

2.13 KEYWORD

Data model: A data model is a collection of conceptual tools for describing data, data relationships, data semantics, and consistency constraints.

An attribute: An attribute is a property or characteristic of an object.

An entity: An entity is an object that exists and is distinguishable from other objects by a specific set of attributes. The existence need not be a material existence.

Entity types: An entity type defines a set of entities that have same attributes. A name and a list of attributes describe each entity type.

Entity set: An entity set is a set of entities of the same type.

Key: A key is a set attributes that is used to identify records.

Super Key: A set of attributes (columns) that is used to identify the records (rows) in a table uniquely is known as Super Key. A table can have many Super Keys

Candidate Key: It can be defined as minimal Super Key or irreducible Super Key. In other words a set attributes that identifies the records uniquely but none of its proper subsets can identify the records uniquely

Primary Key: A Candidate Key that is used by the database designer for unique identification of each row in a table is known as Primary Key. A Primary Key can consist of one or more attributes of a table

Foreign Key: A foreign key is a set of attributes in one base table that points to the candidate key (generally it is the primary key) of another table. The purpose of the foreign key is to ensure referential integrity of the data

Relationship: A relationship is an association (combination) among the instance of one or more entity type.

2.14 UNIT-END EXERCISES AND ANSWERS

1. Explain various symbols used in ER diagrams.
2. What are design issues with ER diagram? Explain
3. What are problems on ER modelling? Explain
4. Explain about EER diagram?
5. What are entity set, attribute and key?

6. Explain about Type role, structural constraints, Weak and Strong entity types?

Answers: SEE

1. 2.7
2. 2.8
3. 2.8
4. 2.11
5. 2.2, 2.3
6. 2.5, 2.6

2.15 SUGGESTED READING

- Fundamentals of Database Systems

  By Ramez Elmasri, Shamkant B. Navathe, Durvasula V.L.N. Somayajulu, Shyam K.Gupta
- Database System Concepts

  By Avi Silberschatz, Henry F. Korth , S. Sudarshan
- Database Management Systems

  By Raghu Ramakrishnan and Johannes Gehrke

UNIT 3: DATA MODELS

Structure:

3.0 OBJECTIVE

At the end of this unit you will be able to know:

- Different types of data models
- Each model with its basic concept
- Comparison of different models

3.1 INTRODUCTION

Data model is a method used to define and analyze data requirements needed to support the business processes of an organization and by defining the data structures and the relationships between data elements.

3.2 CLASSIFICATION OF DATA MODELS

A database model is a specification describing how a database is structured and used. Several such models have been suggested. Common models include:

- Hierarchical model: In this model data is organized into a tree-like structure, implying a single upward link in each record to describe the nesting, and a sort field to keep the records in a particular order in each same-level list.

- Network model: This model organizes data using two fundamental constructs, called records and sets. Records contain fields, and sets define one-to-many relationships between records: one owner, many members.
- Relational model: is a database model based on first-order predicate logic. Its core idea is to describe a database as a collection of predicates over a finite set of predicate variables, describing constraints on the possible values and combinations of values.

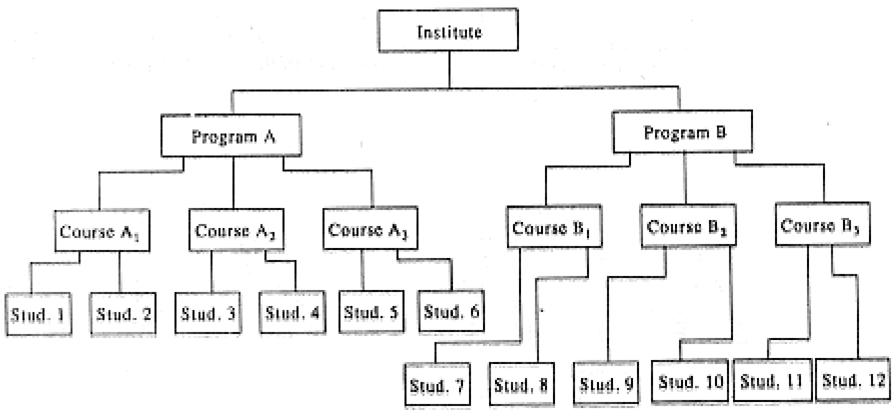
3.3 HIERARCHICAL MODELS-BASIC CONCEPT

A DBMS belonging to the hierarchical data model uses tree structures to represent relationship among records. Tree structures occur naturally in many data organizations because some entities have an intrinsic hierarchical order. For example, an institute has a number of programmes to offer. Each program has a number of courses. Each course has a number of students registered in it. The following figure depicts, the four entity types Institute, Program, Course and Student make up the four different levels of hierarchical structure.

A hierarchical database therefore consists of a collection of records, which are connected with each other through links. Each record is a collection of fields (attributes), each of which contains one data value. A link is an association between precisely two records.

A tree structure diagram serves the same purpose as an entity-relationship diagram; namely it specifies the overall logical structure of the database.

The following figure shows typical database occurrence of a hierarchical structure (tree structure).



The hierarchical data model has the following features:

· Each hierarchical tree can have only one root record type and this record type does not have a parent record type.

· The root cart have any number of child record types and each of which can itself be a root of a hierarchical sub-tree.

· Each child record type can have only one parent record type; thus a M:N relationship cannot be directly expressed between two record types.

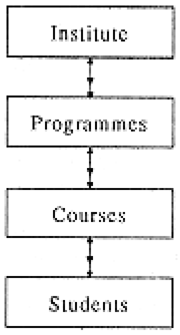
· Data in a parent record applies to all its children records

· A child record occurrence must have a parent record occurrence; deleting a parent record occurrence requires deleting its entire children record occurrence.

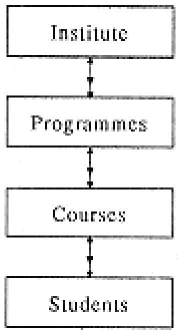
3.4 NETWORK MODELS-BASIC CONCEPT

The Database Task Group of the Conference on Data System Language (DBTG/CODASYL) formalized the network data model in the late 1960s. Their first report that has been revised a number of times, contained detailed specifications for the network data model (a model conforming to these specifications is also known as the DBTG data model). The specifications contained in the report and its subsequent revisions have been subjected to much debate and criticism. Many of the current database applications have been built on commercial DBMS systems using the DBTG model..

The popularity of the network data model coincided with the popularity of the hierarchical data model. Some data were more naturally modelled with more than one parent per child. So, the network model permitted the modelling of many-to-many relationships in data. In 1971, the Conference on Data Systems Languages (CODASYL) formally defined the network model. The basic data modelling construct in the network model is the set construct. A set consists of an owner record type, a set name, and a member record type. A member record type can have that role in more than one set, hence the multivalent concept is supported. An owner record type can also be a member or owner in another set. The data model is a simple network, and link and intersection record types (called junction records by IDMS) may exist, as well as sets between them. Thus, the complete network of relationships is represented by several pair wise sets; in each set some (one) record type is owner (at the tail of the network arrow) and one or more record types are members (at the head of the relationship arrow). Usually, a set defines a 1:M relationship, although 1:1 is permitted. The CODASYL network model is based on mathematical set theory.



A DBMS belonging to the hierarchical data model uses tree structures to represent relationship among records. Tree structures occur naturally in many data organizations because some entities have an intrinsic hierarchical order. For example, an institute has a number of programmes to offer. Each program has a number of courses. Each course has a number of students registered in it. The following figure depicts the four entity types Institute, Program, Course and Student make up the four different levels of hierarchical structure. The figure shows an example of database occurrence for an institute. A database is a collection of database occurrence.

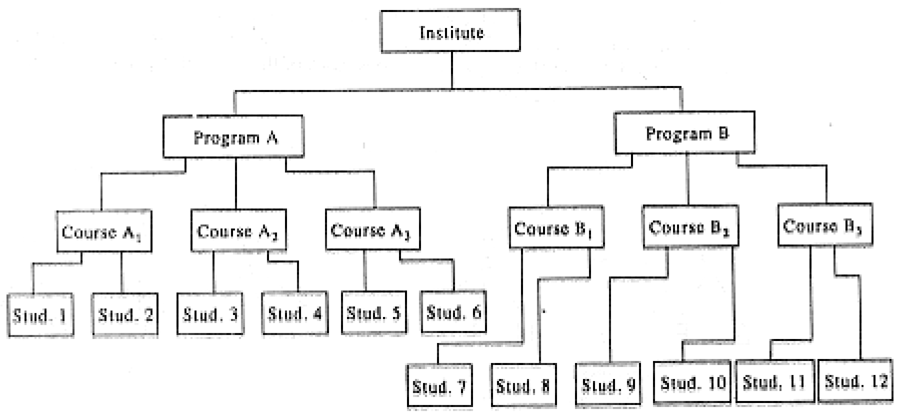


A simple Hierarchy

A hierarchical database therefore consists of a collection of records, which are connected with each other through links. Each record is a collection of fields (attributes), each of which contains one data value. A link is an association between precisely two records.

A tree structure diagram serves the same purpose as an entity-relationship diagram; namely it specifies the overall logical structure of the database.

The following figure shows typical database occurrence of a hierarchical structure (tree structure).



Database occurrence of a hierarchical structure

The hierarchical data model has the following features:

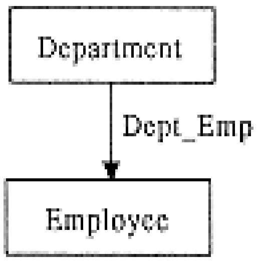
- Each hierarchical tree can have only one root record type and this record type does not have a parent record type.
- The root cart has any number of child record types and each of which can itself be a root of a hierarchical sub-tree.
- Each child record type can have only one parent record type; thus a M:N relationship cannot be directly expressed between two record types.
- Data in a parent record applies to all its children records
- A child record occurrence must have a parent record occurrence; deleting a parent record occurrence requires deleting its entire children record occurrence.

3.5 DBTG CODASYL MODELS

The DBTG model uses two different data structures to represent the database entities and relationships between the entities, namely record type and set type. A record type is used to represent an entity type. It is made up of a number of data items that represent the attributes of the entity.

A set is used to represent a directed relationship between two record types, the so-called owner record type, and the member record type. The set type, like the record type, is named and specifies that there is a one-to-many relationship (I:M) between the owner and member record types. The set type can have more than one re-cord type as its member, but only one record type is allowed to be the owner in a given set type. A database could have one or more occurrences of each of its record and set types. An occurrence of a set type consists of an occurrence of each of its record and set types. An occurrence of a set type consists of an occurrence of the owner record type and any number of occurrences of each of its member record types. A record type cannot be a member of two distinct occurrences of the same set type.

Bachman introduced a graphical means called a data structure diagram to denote the logical relationship implied by the set. Here a labelled rectangle represents the corresponding entity or record type. An arrow that connects two labelled rectangles represents a set type. The arrow direction is from the owner record type to the member record type. Figure shows two record types (DEPARTMENT and ENTLOYEE) and the set type DEPIT-EMP, with DEPARTMENT as the owner record type and EMPLOYEE as the member record type.

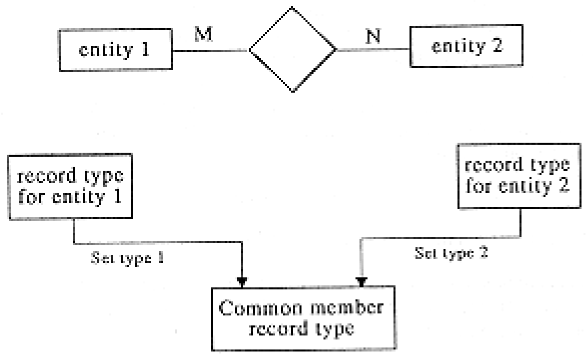


A DBTG set

The data structure diagrams have been extended to include field names in the record type rectangle, and the arrow is used to clearly identify the data fields involved in the set association. A one-to-many (I:M) relationship is shown by a set arrow that starts from the owner field in the owner record type. The arrow points to the member field within the member record type. The fields that support the relationship are clearly identified.

A logical record type with the same name represents each entity type in an E-R diagram. Data fields of the record represent the attributes of the entity. We use the term logical record to indicate that the actual implementation may be quite different.

The conversion of the E-R diagram into a network database consists of converting each I:M binary relationship into a set (a 1:1 binary relationship being a special case of a 1:M relationship). If there is a 1:M binary relationship R1 from entity type E1 to entity type E2,



Conversion of an M:N relationship into two 1:M DBTG sets

Then the binary relationship is represented by a set an instance of this would be S1 with an instance of the record type corresponding to entity E1 as the owner and one or more instances of the record type corresponding to entity E2 as the member. If a relationship has attributes, unless the attributes can be assigned to the member record type, they have to be maintained in a separate logical record type created for this purpose. The introduction of this additional record type requires that the original set be converted into two symmetrical sets, with the record corresponding to the attributes of the relationship as the member in both the sets and the records corresponding to the entities as the owners.

Each many-to-many relationship is handled by introducing a new record type to represent the relationship wherein the attributes, if any, of the relationship are stored. We then create two symmetrical I:M sets with the member in each of the sets being the newly introduced record type. The conversion of a many-to-many relationship into two one-to-many sets using a common member record type is shown in figure.

In the network model, the relationships as well as the navigation through the database are predefined at database creation time

3.6 RELATION MODELS

The relational data base approach is relatively recent and begun with a theoretical paper of Codd. In this paper it has been proposed that by using a technique called normalization the entanglement observation in the tree and network structure can be replaced by a relatively neater structure. Codd principles relate to the logical description of the data and it is important bear in mind that this is quite independent and feasible way in which the data is stored. It is only some years back that these concepts have emerged from the research development test and trial stages and are being seen as commercial projects. The attractiveness of the relational approach arouses from the simplicity in the data organization and the availability of reasonably simple to very powerful query languages. The size of the relational database approach is that all the data is expressed in terms of tables and nothing but tables. Therefore, all entities and attributes have to be expressed in rows and columns

The differences that arise in the relational approach are in setting up relationships between different tables. This actually makes use of certain mathematical operations on the relation such as projection, union, joins, etc. These operations from relational algebra and relational calculus are discussion in some more details in the second Block of this course. Similarly in order to achieve the organization of the data in terms of tables in a satisfactory manner, a technique called normalization is used.

A unit in the second block of this course describes in detail the processing of normalization and various stages including the first normal forms, second normal forms, and the third normal forms. At the moment it is sufficient to say that normalization is a technique, which helps in determining the most appropriate grouping of data items into records, segments or tuples. This is necessary because in the relational model the data items are arranged in tables, which indicate the structure, relationship, and integrity in the following manner:

1. In any given column of a table, all items are of the same kind
2. Each item is a simple number or a character string
3. All rows of a table are distinct. In other words, no two rows which are identical in every column.
4. Ordering of rows within a table is immaterial
5. The columns of a table are assigned distinct names and the ordering of these columns is immaterial
6. If a table has N columns, it is said to be of degree N. This is sometimes also referred to as the cardinality of the table. From a few base tables it is possible by setting up relations; create views, which provide the necessary information to the different users of the same database.

Advantages and Disadvantages of Relational

Approach

Advantages of Relational approach

The popularity of the relational database approach has been apart from access of availability of a large variety of products also because it has certain inherent advantages.

1. **Ease of use:** The revision of any information as tables consisting of rows and columns is quite natural and therefore even first time users find it attractive.

2. **Flexibility:** Different tables from which information has to be linked and extracted can be easily manipulated by operators such as project and join to give information in the form in which it is desired.

3. **Precision:** The usage of relational algebra and relational calculus in the manipulation of the relations between the tables ensures that there is no ambiguity, which may otherwise arise in establishing the linkages in a complicated network type database.

4. **Security:** Security control and authorization can also be implemented more easily by moving sensitive attributes in a given table into a separate relation with its own authorization controls. If authorization requirement permits, a particular attribute could be joined back with others to enable full information retrieval.

5. **Data Independence:** Data independence is achieved more easily with normalization structure used in a relational database than in the more complicated tree or network structure.

6. **Data Manipulation Language:** The possibility of responding to ad-hoc query by means of a language based on relational algebra and relational calculus is easy in the relational database approach. For data organized in other structure the query language either becomes complex or extremely limited in its capabilities.

Disadvantages of Relational Approach

One should not get carried away into believing that there can be no alternative to the RDBMS. This is not so. A major constraint and therefore disadvantage in the use of relational database system is machine performance. If the number of tables between which relationships to be established are large and the tables themselves are voluminous, the performance in responding to queries is definitely degraded. It must be appreciated that the simplicity in the relational database approach arises in the logical view. With an interactive system, for example an operation like join would depend upon the physical storage also. It is, therefore common in relational databases to tune the databases and in such a case the physical data layout would be chosen so as to give good performance in the most frequently run operations.

It therefore would naturally result in the fact that the lays frequently run operations would tend to become even more shared.

While the relational database approach is a logically attractive, commercially feasible approach, but if the data is for example naturally organized in a hierarchical manner and stored as such, the hierarchical approach may give better results. It is helpful to have a summary view of the differences between the relational and the non-relational approach in the following section.

3.7 COMPARISION OF DIFFERNT MODELS

1. **Implementation independence:** The relational model logically represents all relationships implicitly, and hence, one does not know what associations are or are not physically represented by an efficient access path (without looking at the internal data model).
2. **Logical key pointers:** The relational data model uses primary (and secondary) keys in records to represent the association between two records. Because of this models implementation independence, however, it is conceivable that the physical database (totally masked from the user of a relational database) could use address pointers or one of many other methods.
3. **Normalization theory:** Properties of a database that make it free of certain maintenance problems have been developed within the context of the relational model (although these properties can also be designed into a network data model database).
4. **High-level programming languages:** Programming languages have been developed specifically to access databases defined via the relational data model; these languages permit data to be manipulated as groups or files rather than procedurally: one record at a time.

3.8 SUMMARY

In this unit we reviewed three major traditional data models used in current DBMSs. These three models are hierarchical, network and relational.

The hierarchical model evolved from the file based system. It uses tree type data structure to represent relationship among records. The hierarchical data model restricts each record type to only parent record type. Each parent record type can have any number of children record types.

In a network model, one child record may have more than one parent nodes. A network can be converted into one or more trees by introducing redundant nodes.

The relational model is based on a collection of tables. A table is also called relation. A tree or network structure can be converted into a relational structure by separating each node in the data structure into a relation.

3.9 KEYWORD

Data model: A data model is a collection of conceptual tools for describing data, data relationships, data semantics, and consistency constraints

The relational model: The relational model is a data model, which uses relations (tables) to represent both entity information and relationship between them.

Domain: data type, set of allowable values for one or more attribute.

Relational Database: collection of normalized relations.

3.10 UNIT-END EXERCISES AND ANSWERS

1. Explain about classification of data models?

2. Explain about hierarchy model?

3. Explain about network model?

4. Explain about DBTG CODASYL model?

5. Explain about Relational model?

6. Give comparison about different?

Answers: SEE

1. 3.2

2. 3.3

3. 3.4

4. 3.5

5. 3.6

6. 3.7

3.11 SUGGESTED READING

[1] Fundamentals of Database Systems By Ramez Elmasri, Shamkant B. Navathe, Durvasula V.L.N. Somayajulu, Shyam K.Gupta

[2] Database System Concepts By Avi Silberschatz, Henry F. Korth , S. Sudarshan

[3] Database Management Systems By Raghu Ramakrishnan and Johannes Gehrke

UNIT 4: Relational Model

Structure:

4.0 OBJECTIVE

At the end of this unit you will be able to know:

- Relational model
- Relational constraints and operation
- Relational calculus
- Assertion and Triggers

4.1 INTRODUCTION

The Relational model was proposed by E.F.Codd in the year 1970. It is relatively new, compared to older models hierarchical model and network model. The relational model has become the most common commercial data model. Relations are actually the mathematical table. This unit introduces relational model concepts.

4.2 RELATIONAL MODEL

The relational database model is the most popular data model. It is very simple and easily understandable by information systems professionals and end users. Understanding a relational model is very simple since it is very similar to Entity Relationship Model. In ER model data is represented as entities similarly here data in represented in the form of relations that are depicted by use of two-dimensional tables. Also attributes are represented as columns of the table.

The basic concept in the relational model is that of a relation. In simple language, a relation is a two-dimensional table. Table can be used to represent some entity information or some relationship between them. Even the table for an entity information and table for relationship information are similar in form. Only from the type of information given in the table can tell if the table is for entity or relationship. The entities and relationships, which we studied in the ER model, are similar to relations in this model. In relational model, tables represent all the entities and relationships identified in ER model. Rows in the table represent records; and columns show the attributes of the entity. Figure 4.1 shows structure of a relation in a relational model.

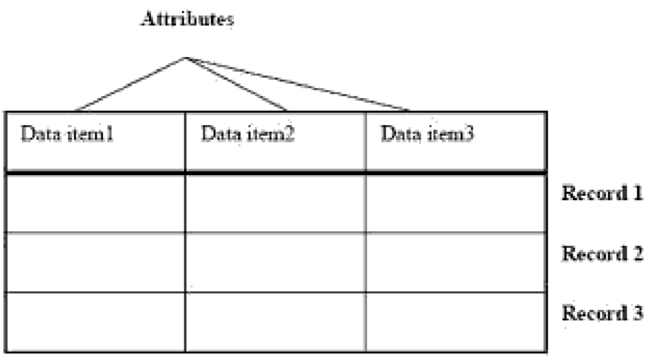


Figure 4.1 Structure of a relation in a relational model.

Structural Part of Relation

- **Relation:** a table with columns and rows.
- **Attribute /field**: named column
- **Tuple:** row of relation (a record)

- **Degree:** number of attributes in a relation. If the column/attribute is only one then relation is of degree one also known as unary. If a relation has two columns then it's known as degree 2 also known as binary.
- **Cardinality:** number of rows (tuples) in a relation.
- **Domain:** data type .Set of allowable values for one or more attribute.
- **Relational Database:** collection of normalized relations.

Basic Structure:

A relation is basically a subset of all possible rows of their corresponding domains. In a general a relation of n attributes must be a subset of:

$D_1 \times D_2 \times \ldots. \times D_{n-1} \times D_n$

Tables are essentially relations. So, we can use the terms tables and relations interchangeably.

Properties of Relation:

- Each table should have a distinct name.
- Each cell should have one value (atomic value).
- Attribute names should be unique.
- Values of an attribute are from the same domain.
- Order of attribute has no significance.
- Each tuple should be distinct. It means no duplicate rows.
- Order of rows has no significance.

4.3 Enforcing Data Integrity Constraints

In Relational model, there are various types of constraints. They are explained below.

Key constraints

Key constraint is implied by the existence of candidate keys. The intension includes a specification of the attribute(s) consisting of the primary key and specification of the attribute(s) consisting alternate keys, if any. Each of these specifications implies a uniqueness constraint (by definition of candidate key). In addition, primary key specification implies a no-nulls constraint (by integrity rule).

Referential constraints

Referential constraints are constraints implied by the existence of foreign keys. The intension includes a specification of all foreign keys in the relation. Each of these specifications implies a referential constraint (by integrity rule).

Other constraints: Many other constraints are possible in theory.

Examples:

1) salary >= 20000
2) age > 20

Participation constraints

If every entity of an entity set is related to some other entity set via a relationship type, then the participation of the first entity type is total. If only few member of an entity type is related to some entity type via a relationship type, the participation is partial.

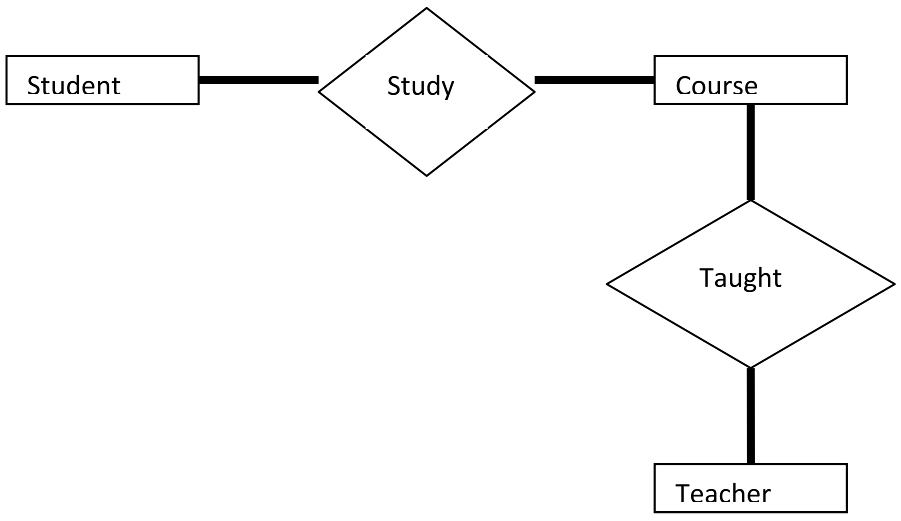
A database schema of a database system is its structure described in a formal language supported by the database management system (DBMS) and refers to the organization of data to create a blueprint of how a database will be constructed (divided into database tables). A relational database schema is the tables, columns and relationships that make up a relationaldatabase.

Schemata are generally stored in a data dictionary. Although a schema is defined in text database language, the term is often used to refer to a graphical depiction of the database structure. In other words, schema is the structure of the database that defines the objects in the database.

The Purpose of a Schema: A relational database schema helps us to organize and understand the structure of a database. This is particularly useful when designing a new database, modifying an existing database to support more functionality, or building integration between databases.

The Figure below shows the relational schema for a teaching department with three entities, namely teacher, student, and course.

There are two intersection entities in this schema: Student and Course and Teacher and Course. These handle the two many-to-many relationships: 1) between Student and Course, and 2) between Teacher and Course. In the first case, a Student may take many Courses and a Course may be taken by many Students. Similarly, in the second case, a Teacher may teach many Courses and a Course may be taught by many Teachers.



An example Relational database scheme

4.4 Relational-Algebra Operations

An operation is fundamental if it cannot be expressed with other operations. The relational algebra has six fundamental operations:

- Select (unary)
- Project (unary)
- Rename (unary)
- Cartesian product (binary)
- Union (binary)
- Set-difference (binary)

The Relational Operations produce a new relation as a result.

Let us consider the following relations (tables) as an example of a banking database in our further discussions:

customer relation over the Customer_Scheme
deposit relation over the Deposit_Scheme
borrow relation over the Borrow_Scheme
branch_scheme over the Branch_Scheme
client_realtion over the Client_Scheme

Customer table:

Customer_name	Custermer_House_number	Custermer_Locality	Custermer_City	Custermer_State
Shiva	23	Indira Nagar	Bangalore	Karnataka
Kaveri	45	Saraswathi Puram	Mysore	Karnakata
Laloo	13	Gandhi Nagar	Patna	Bihar
Kumar	56	Balaji Nagar	Tirupati	Andra Pradesh
Khan	67	Treemurthi	Delhi	Dehli
Parvathi	23	Indira Nagar	Bangalore	Karnataka
Asha	47	Netaji Nagar	Mysore	Karnataka
Ganesh	79	Thana	Mumbai	Maharastra

Deposit table:

Account_Number	Customer_Name	Branch_Name	Account_Balance
45	Kaveri	University	10000

56	Laloo	Redfort	1000000
78	Kumar	Majestic	20000
20	Shiva	Palace	30000
20	Parvathi	Palace	30000
21	Asha	Majestic	22000

Borrow table:

Loan_Number	Customer_Name	Branch_Name	Loan_Amount
10	Laloo	Saltlake	20000
13	Kumar	University	30000
13	Asha	University	30000
75	Khan	Redfort	15000
55	Laloo	Majestic	25000
66	Khan	University	40000
18	Ganesh	University	65000
11	Kuamr	Majestic	13000
61	Laloo	Redfort	22000

Branch table:

Branch_Name	Branch_City	Branch_Locality	Assets
Saltlake	Kolkata	Subhas Nagar	12000000
Redfort	Delhi	Puranadelhi	30000000
University	Mysore	Manasagangotri	1150000
Majestic	Bangalore	Gandhi Nagar	450000000
Palace	Mysore	Shivarampet	35000000

Client table:

Customer_Name	Banker_Name
Kumar	Shiva
Asha	Shiva
Laloo	Kaveri
Khan	Shiva
Asha	Rama
Laloo	Rama

Select Operation (σ):

Select is denoted by a lowercase Greek sigma (σ), with the predicate appearing as a subscript. The argument relation (table) is given in parentheses following the σ. **Select** operation selects tuples (rows) that satisfy a given predicate from the input table.

For example, let us consider the query:

“select tuples (rows) of the *borrow* relation where the branch is **MAJESTIC**”

We can express it as:

$$\sigma_{\text{Branch\_name = “MAJESTIC”}} \text{(borrow)}$$

The result of this operation consists of only one tuple as given:

Loan_Number	Customer_Name	Branch_Name	Loan_Amount
55	Laloo	Majestic	25000

We can allow comparisons using =, $\neq$, <, $\leq$, > and $\geq$ in the selection predicate.

We can also allow the logical connectives $\vee$(or) and $\wedge$(and). For example:

$$\sigma_{\text{Branch\_name = “University”} \wedge \text{ Amount> 30000}} \text{(borrow)}$$

The result of this operation is:

Loan_Number	Customer_Name	Branch_Name	Loan_Amount
66	Khan	University	40000
18	Ganesh	University	65000

The Project Operation (Π):

Projection is denoted by the Greek capital letter pi (Π). Project outputs its argument relation for the specified attributes only. The attributes to be output appear as subscripts. Since a relation is a set, duplicate rows are eliminated.

For example, let us express the query:

"find the names of branches and names of customers who have taken loans"

We can express it as:

$$\Pi_{\text{Branch\_Name, Customer\_Name}} \text{ (borrow)}$$

Branch_Name	Customer_Name
Saltlake	Laloo
University	Kumar
University	Asha
Redfort	Khan
Majestic	Laloo
University	Khan
University	Ganesh

We can also combine select and project operations as shown below:

$$\Pi_{\text{Customer\_Name}} \ (\sigma_{\text{Branch\_name = "MAJESTIC"}} \text{ (borrow))}$$

This results in the displaying of customer names who have taken loans from MAJESTIC branch.

The output of this operation is:

Customer_Name
Laloo

We can think of select as taking rows of a relation, and project as taking columns of a relation.

The Cartesian Product Operation (×):

The cartesian product of two relations is denoted by a cross ($\times$). It is a binary operation and requires two argument relations as input. For two input relations r_1 and r_2, it can be written as

$$r_1 \times r_2$$

The result of $r_1 \times r_2$ is a new relation with a tuple for each possible pairing of tuples from r_1 and r_2. In order to avoid ambiguity, the attribute names have attached to them the name of the relation from which they came. If no ambiguity will result, we drop the relation name. If r_1 has n_1 tuples, and r_2 has n_2 tuples, then $r = r_1 \times r_2$ will have $n_1 n_2$ tuples. The resulting scheme is the concatenation of the schemes of r_1 and r_2, with relation names added as mentioned. To find the clients of banker Rama and the city in which they live, we need information in both client and customer relations. We can get this by writing

$$\sigma_{\text{Banker\_Name= “Rama”}} (\text{client} \times \text{customer})$$

We want rows where client.Customer_Name = customer.Customer_Name. So we can write to get just these tuples

$$\sigma_{\text{client.Customer\_Name= customer.Customer\_Name}} (\sigma_{\text{Banker\_Name= “Rama”}} (\text{client} \times \text{customer}))$$

Finally, to get just the customer's name and city, we can use a projection as shown:

$$\Pi_{\text{client.Customer\_Name, Customer\_City}} (\sigma_{\text{client.Customer\_Name= customer.Customer\_Name}} (\sigma_{\text{Banker\_Name= “Rama”}} (\text{client} \times \text{customer})))$$

The final output of the above query is:

Customer_name	Custermer_City
Laloo	Patna
Asha	Mysore

The Rename Operation (ρ)

The rename operation solves the problems that occur with naming when performing the Cartesian product of a relation with itself.

Suppose we want to find the names of all the customers who live with “Shiva” in the same locality, same city and same state.

We can get the information of Shiva by writing

$$\Pi_{\text{Customer\_Locality, Customer\_City, Customer\_State}} (\sigma_{\text{Customer\_name =“Shiva”}} (\text{customer}))$$

To find other customers with the same information, we need to reference the customer relation again:

$$\sigma_P(\text{customer} \times \Pi_{\text{Customer\_Locality, Customer\_City, Customer\_State}}(\sigma_{\text{Customer\_name} = \text{"Shiva"}}(\text{customer})))$$

Where ***P*** is a selection predicate requiring Customer_Locality, Customer_City, and Customer_State values to be equal.

The problem is how do we distinguish between the two Locality values, two City values and two State values appearing in the Cartesian product, as both come from a customer relation. This leads to an ambiguity. The solution is use the rename operator, denoted by the Greek letter rho(ρ).

To rename a relation, the general express is:

$$\rho_x(r)$$

to get the relation ***r*** under the name of ***x***.

By using this to rename one of the two customer relations we can remove the ambiguity.

$$\Pi_{\text{customer.Customer.Name}}(\sigma_{\text{cust2.Customer\_Locality = customer.Customer\_Locality} \wedge \text{cust2. Customer\_City = customer.Customer.city} \wedge \text{cust2.Customer\_State = customer.Customer.State}}(\text{customer} \times (\Pi_{\text{Customer\_Locality, Customer\_City, Customer\_State}}(\sigma_{\text{Customer\_Name} = \text{"Shiva"}}(\rho_{\text{cust2}}(\text{customer}))))))$$

The output of the above query is:

Customer_name
Shiva
Parvathi

The Union Operation ($\cup$):

The Union operation is a binary operation. This operation is denoted $\cup$ as in set theory. It returns the union (set union) of two compatible relations (tables). For a union operation $r \cup s$ to be legal, we require that r and s must have the same number of attributes and of the same type. To find all customers of the MAJESTIC branch, we must find everyone who has a loan or an account or both at the branch.

We need both borrow and deposit relations for this. We can express it using union operator as below:

$$\Pi_{\text{Customer\_Name}} \left(\sigma_{\text{Branch\_Name} = \text{“MAJESTIC”}} \text{ borrow}\right)$$

$$\cup$$

$$\Pi_{\text{Customer\_Name}} \left(\sigma_{\text{Branch\_Name} = \text{“MAJESTIC”}} \text{ depsoit}\right)$$

The output of the above query is:

Customer_name
Kumar
Laloo

As in all set operations, duplicates if any are eliminated.

The Set Difference Operation ($-$):

The Set difference operation is a binary operation. It is denoted by the minus sign ($-$). It returns tuples that are in one relation (first relation), but not in another(second relation).

Thus $r - s$ results in a relation containing tuples that are in r but not in s.

To find customers of the MAJESTIC branch who have an account there but no loan, we express it as:

$$\Pi_{\text{Customer\_Name}} \left(\sigma_{\text{Branch\_Name} = \text{“MAJESTIC”}} \text{ depsoit }\right)$$

$-$

$$\Pi_{Customer\_Name}\ (\sigma_{Branch\_Name = \text{“MAJESTIC”}}\ borrow\ )$$

The output of the above query is:

Customer_name
Asha

4.5 Extended Relational Algebra Operations

Additional relational operations are those which are defined in terms of the fundamental operations. They do not add power to the algebra, but are useful to simplify common queries. The list of additional operations consists of set-intersection, natural join, division, and assignment operations.

The Set Intersection Operation:

Set intersection is denoted by $\cap$ and returns a relation that contains tuples that are in both of its argument relations. The general format of this operation is:

$$r \bigcap s = r - (r - s)$$

To find all customers having both a loan and an account at the REDFORT branch, we can write it as:

$$\Pi_{Customer\_Name}\ (\sigma_{Branch\_Name = \text{“REDFORT”}}\ depsoit\ )$$

$$\bigcap$$

$$\Pi_{Customer\_Name}\ (\sigma_{Branch\_Name = \text{“REDFORT”}}\ borrow\ )$$

The output of the above query is:

Customer_name
Laloo

The Natural Join Operation:

Very often we want to simplify queries on a Cartesian product. For example, “to find all customers having a loan at the bank and the cities in which they live”, we need borrow and customer relations:

The query expression is:

$\Pi_{\text{borrow.Customer\_Name, customer.Customer\_City \_}}$ $(\sigma_{\text{borrow.Customer\_Name= customer\_Customer\_Nanme}}$ (borrow $\times$ customer))

Our selection predicate obtains only those tuples pertaining to Customer_name. This type of operation is very common, so we have the natural join, denoted by a $\bowtie$ sign. Natural join combines a Cartesian product and a selection into one operation. It performs a selection forcing equality on those attributes that appear in both relation schemes. Duplicates are removed as in all relation operations. To illustrate this, we can rewrite the above query as

$\Pi_{\text{Customer\_Name, Customer\_City}}$ (borrow $\bowtie$ customer)

The resulting relation is:

Customer_name	Custermer_City
Azay	Patna
Kumar	Tirupati
Khan	Delhi
Parvathi	Bangalore
Asha	Mysore
Ganesh	Mumbai

Now let us make a more formal definition of natural join. Consider R and S to be sets of attributes. We can denote attributes appearing in both relations by $R \cap S$. We can denote attributes in either or both relations by $R \cup S$. Consider two relations $r(R)$ and $s(S)$. The natural join of r and s, denoted by $r \bowtie s$ is a relation on scheme $R \cup S$. It is a projection onto $R \cup S$ of a selection on $r \times s$ where the predicate requires $r.A = s.A$ for each attribute A in $R \cap S$. Formally,

$$r \bowtie s = \Pi_{R \cup S}(\sigma_{r.A_1 = s.A_1 \wedge r.A_2 = s.A_2 \wedge \ldots \wedge r.A_n = s.A_n}(r \times s))$$

Where $R \cap S = \{A_1, A_2, \ldots, A_n\}$.

To find the assets and names of all branches which have depositors living in Bangalore, we need customer, deposit, and branch relations and we can express the query as:

$\Pi_{\text{Branch\_Name, assets}}$ ($\sigma_{\text{Customer\_city =“Bangalore”}}$ (customer $\bowtie$ deposit $\bowtie$ branch))

Note that $\bowtie$ is associative.

To find all customers who have both an account and a loan at the MAJESTIC branch, we can express it as:

$\Pi_{\text{Customer\_Name}}$ ($\sigma_{\text{Branch\_Name =“MAJESTIC”}}$

(borrow $\bowtie$ deposit))

This is equivalent to the set intersection version. We can see that there can be several ways to write a query in the relational algebra. If two relations $r(R)$ and $s(S)$ have no attributes in common, then $R \cap S = \emptyset$, and $r \bowtie s = r \times s$.

The Division Operation:

Division operation, denoted $\div$ is suited to queries that include the phrase ``for all''. Suppose we want to “find all the customers who have an account at all branches located in Mysore”.

Strategy: think of it as three steps. We can obtain the names of all branches located in Mysore by:

$r_1 = \Pi_{\text{Branch\_Name}}$ ($\sigma_{\text{Branch\_City =“Mysore”}}$ (branch))

We can also find all Customer_Name, Branch_Name pairs for which the customer has an account by

$r_2 = \Pi_{\text{Customer\_Name, Branch\_Name}}$ ((deposit))

Now we need to find all customers who appear in r_2 with every branch name in r_1.

The divide operation provides exactly those customers:

$\Pi_{\text{Customer\_Name, Branch\_Name}}$ ((deposit))

$\div$

Π_{Branch_Name} ($\sigma_{Branch\_City = \text{"Mysore"}}$ (branch))

Which is simply $r_2 \div r_1$.

Now, we can formally define division operation as: Let $r(R)$ and $s(S)$ be relations. Let $S \subseteq R$. The relation $r \div s$ is a relation on scheme $R - S$. A tuple t is in $r \div s$ if for every tuple t_s in s there is a tuple t_r in r satisfying both of the following:

t_r [S] = t_s [S]

t_r [R - S] = t[R - S]

These conditions say that the $R - S$ portion of a tuple t is in $r \div s$ if and only if there are tuples with the $r - s$ portion and the S portion in r for every value of the S portion in relation S. The division operation can be defined in terms of the fundamental operations.

$$r \div s = \Pi_{R-S}(r) - \Pi_{R-S}((\Pi_{R-S}(r) \times s) - r)$$

The Assignment Operation

Sometimes it is useful to be able to write a relational algebra expression in parts using a temporary relation variable. The assignment operation, denoted by $\leftarrow$, works like assignment in a programming language. We could rewrite our division definition as

$$temp \leftarrow \Pi_{R-S}(r)$$
$$temp - \Pi_{R-S}((temp \times s) - r)$$

Note that no extra relation is added to the database, but the relation variable created can be used in subsequent expressions. Assignment to a permanent relation would constitute a modification to the database.

4.6 Relational Calculus

A relational calculus expression creates a new relation, which is specified in terms of variables that range over rows of the stored database relations (in tuple calculus) or over columns of the stored relations (in domain calculus).

In a calculus expression, there is no order of operations to specify how to retrieve the query result a calculus expression specifies only what information the result should contain. This is the main distinguishing feature between relational algebra and relational calculus. Relational calculus is considered to be a nonprocedural language. This differs from relational algebra, where we must write a sequence of operations to specify a retrieval request, hence relational algebra can be considered as a procedural way of stating a query.

Tuple Relational Calculus

The tuple relational calculus is based on specifying a number of tuple variables. Each tuple variable usually ranges over a particular database relation, meaning that the variable may take as its value any individual tuple from that relation. A simple tuple relational calculus query is of the form

{t | COND(t)}

where t is a tuple variable and COND (t) is a conditional expression involving t. The result of such a query is the set of all tuples t that satisfy COND (t).

Example: To find the first and last names of all employees whose salary is above $50,000, we can write the following tuple calculus

expression:

{t.FNAME, t.LNAME | EMPLOYEE(t) AND

t.SALARY>50000}

The condition EMPLOYEE(t) specifies that the range relation of tuple variable t is EMPLOYEE. The first and last name

(PROJECTION πFNAME, LNAME) of each EMPLOYEE tuple t that

satisfies the condition t.SALARY>50000 (SELECTION

σ SALARY >50000) will be retrieved.

4.7 Assertion and Triggers

An assertion is a piece of SQL which makes sure a condition is satisfied or it stops action being taken on a database object. It could mean locking out the whole table or even the whole database.To make matters more confusing - a trigger could be used to enforce a check constraint and in some DBs can take the place of an assertion (by allowing you to run code un-related to the table being modified). A common mistake for beginners is to use a check constraint when a trigger is required or a trigger when a check constraint is required. An example: All new customers opening an account must have a balance of $100; however, once the account is opened their balance can fall below that amount. In this case you have to use a trigger because you only want the condition evaluated when a new record is inserted.

TRIGGERs are pieces of executable code of which it can be declared to the DBMS that those should be executed every time a certain kind of update operation (insert/delete/update) gets done on a certain table. Because triggers can raise exceptions, they are a MEANS for implementing the same thing as an ASSERTION. However, with triggers, it's still the programmer who has to do all the coding, and not make any mistake. An example of a trigger in plain English might be something like: before updating a customer record, save a copy of the current record. Which would look something like:

```
CREATE TRIGGER triggerName

AFTER UPDATE

   INSERT INTO CustomerLog (blah, blah, blah)

   SELECT blah, blah, blah FROM deleted
```

4.8 SUMMARY

A relational algebra is an algebra defined over relations. The inputs are relations and the output is also a relation. The relational algebra is a procedural query language. In this unit, we discussed theory of relational algebra, the fundamental operations, and the additional operation of relational algebra. We also studied how to express some queries in relational algebra.

4.9 KEYWORDS

Relational algebra: A relational algebra is an algebra defined over relations.
Fundamental operation: An operation is fundamental if it cannot be expressed with other operations.
Additional operation: An operation is additional if it cannot be expressed with the fundamental operations

4.10 UNIT-END EXERCISES AND ANSWERS

1. What are the fundamental operations in relational algebra?
2. Explain select, project, Cartesian product, rename, union, and set difference operations with examples.
3. What are additional relational operations? Explain.
4. Express some example queries in relational calculus.
5. Define assertion and trigger.

Answer: SEE

1. 4.2
2. 4.3
3. 4.4
4. 4.6
5. 4.7

4.11 SUGGESTED READINGS

[1] Fundamentals of Database Systems By Ramez Elmasri, Shamkant B. Navathe, Durvasula V.L.N. Somayajulu, Shyam K.Gupta

[2] Database System Concepts By Avi Silberschatz, Henry F. Korth , S. Sudarshan

[3] Database Management Systems By Raghu Ramakrishnan and Johannes Gehrke

Module 2

UNIT 5: Commercial Query Languages

Structure:

5.0 OBJECTIVES

At the end of this unit you will be able to know:

- Basics of SQL
- Write SQL queries to examine the data in the rows and columns of relational tables.
- Aggregation, views in SQL.
- Embedded SQL and Procedural extension to SQL
- Introduction to Query-by-example(QBE).
- Additional Features of SQL

5.1 INTRODUCTION

SQL is a standard query language for accessing and manipulating databases. SQL stands for Structured Query Language. It is an ANSI (American National Standards Institute) standard. Note that are there are many different versions of the SQL language. The original version was developed at IBM's San Jose Research Laboratory. It was originally called Sequel and implemented as part of System R project. Over a period of time its name has changed to SQL.

This unit presents the basics of the SQL language, and together with the succeeding units on SQL, provides a detailed introduction to the SQL language. It provides the basic concepts needed to understand the syntax of the language.

5.2 INTRODUCTION TO SQL

We can do the followings with SQL:

- Execute queries against a database
- Retrieve data from a database
- Insert records in a database
- Update records in a database
- Delete records from a database
- Create new databases
- Create new tables in a database
- Create stored procedures in a database
- Create views in a database
- Set permissions on tables, procedures, and views

SQL Syntax:

To understand SQL syntax let us consider a database table first.

Database Tables: A database contains one or more tables. Each table is identified by a name. Tables contain records (rows) with data.

Below is an example of a table called "persons", which contains three records.

Person_Id	Name	House_Number	Street	Locality	City
1	Rama	24	Palace	Saraswathipuram	Mysore
2	Sita	38	Sastri	Jayanagar	Bangalore
3	Odeyar	45	Irwin	Agrahara	Mysore

SQL Statements:

Most of the actions you need to perform on a database are done with SQL statements.

The following SQL statement will select all the records in the "persons" table:

SELECT *

FROM persons

Note that SQL is not case sensitive. In general an SQL statement looks like:

Select attribute1, attribute2, …. attributen

from table1, table2, … tablem

[where condition]

Here there are three clauses: select, from and where (where is optional). If where clause is omitted, then the condition is true. The result of an SQL query expression is a relation.

For example, let us consider the query "Find all persons who live in Saraswathipuram locality of Mysore City"

This can be expressed in SQL as:

select Name

from person

where Locality=”Saraswathipuram” and City=”Mysore”

The output will be:

Name
Rama

5.3 Basic queries in SQL

SQL Data Manipulation Language (DML) and Data Manipulation Language (DDL): SQL has two parts: DML and DDL.

The query and update commands form the DML part of SQL. They include the following commands:

- SELECT: extracts data from a database
- UPDATE: updates data in a database
- DELETE: deletes data from a database
- INSERT INTO: inserts new data into a database

The DDL part of SQL can be used to create or delete database tables. It can also be used to define indexes (keys), to specify links between tables, and to impose constraints between tables. The most important DDL statements in SQL are:

- CREATE DATABASE: creates a new database
- ALTER DATABASE: modifies a database
- CREATE TABLE: creates a new table
- ALTER TABLE: modifies a table
- DROP TABLE: deletes a table
- CREATE INDEX: creates an index
- DROP INDEX: deletes an index

SQL SELECT Statement: The SELECT statement is used to select data from a database. The result can be stored in a result table.

SQL SELECT Syntax:

SELECT attribute_name(s)

FROM table_name(s)

For example, "Find all the person names" can be expressed in SQL as:

```
select Name
from person
```

The result is:

Name
Rama
Sita
Odeyar

To select all attributes of a table, we can use SELECT *. For example, let us say we want to select all the columns from the "persons" table. We express it as:

Select * from person

The result is:

Person_Id	Name	House_Number	Street	Locality	City
1	Rama	24	Palace	Saraswathipuram	Mysore
2	Sita	38	Sastri	Jayanagar	Bangalore
3	Odeyar	45	Irwin	Agrahara	Mysore

The asterisk (*) is a quick way of selecting all columns. The result-set will look like the original input relation:

The SELECT DISTINCT Statement:

In SQL duplicates are not eliminated by default. In order to remove duplicates we can use the DISTINCT keyword to return only distinct (different) values.

The syntax is:

SELECT DISTINCT attribute(s)

FROM table_name(s)

Now, in the result relation, the duplicates if are eliminated.

For example,

select City from person

Will have the result:

City
Mysore
Bangalore
Mysore

Whereas Select distinct City from person

Will have the result:

City
Mysore
Bangalore

The WHERE Clause: The WHERE clause is used to extract only those records that fulfill a specified criterion

WHERE Syntax

SELECT column_name(s) FROM table_name(s) WHERE condition

For example,

Select Name from person where City=Mysore”

Will have the result

Name
Rama
Odeyar

With the WHERE clause, we can use the following operators:

Operator	Meaning(Description)
=	Equal
<>	Not equal (in some versions of SQL, != is used)
>	Greater than
<	Less than
>=	Greater than or equal
<=	Less than or equal
BETWEEN	Between an inclusive range
LIKE	Search for a pattern
IN	If we know the exact value we want to return for at least one of the columns

The AND & OR Operators: The AND & OR operators are used to filter records based on more than one condition. The AND operator displays a record if both the first condition and the second condition are true. The OR operator displays a record if either the first condition or the second condition is true

For example,

i) To express the query :"Find all persons who live in the Palace street of Mysore City"

SELECT Name FROM Persons WHERE Street="Palace" AND City="Mysore"

The result is:

Name
Rama

ii) To express the query : “Find all persons who live in the Palace street or Irwin street”

SELECT Name FROM Persons WHERE Street=”Palace” OR Street =”Irwin”

The result is:

Name
Rama
Odeyar

Combining AND & OR

For example,

iii) To express the query: “Find all persons who live in the Palace street or Irwin street of Mysore City”

SELECT Name FROM Persons WHERE (Street=”Palace” OR Street =”Irwin”) AND City=”Mysore”

The result is:

Name
Rama
Odeyar

The ORDER BY Keyword: The ORDER BY keyword is used to sort the result-set by a specified column. The ORDER BY keyword sorts the records in ascending order by default. If we want to sort the records in a descending order, we can use the DESC keyword. If we want to sort the records in a ascending order, we can use the ASC keyword.

ORDER BY Syntax:

SELECT attribute_name(s) FROM table_name(s) ORDER BY attribute_name(s) ASC|DESC

For example: “Find all person names in ascending order”

SELECT Name FROM Persons ORDER BY Name

The result is:

Name
Odeyar
Rama
Sita

To display the same in descending order of Names:

SELECT Name FROM persons ORDER BY Name DESC

The result is:

Name
Sita
Rama
Odeyar

5.4 ADVANCE QUERIES IN SQL

The Advanced Query element allows you to execute more complex query statements or any other SQL statements. This element can be used to manage your entities or to execute any other statement in the database.

An advanced query can have:

- Input parameters: You might need to use a parameter in your query. You manage input parameters in the Advanced query editor.
- Output parameter: The output parameters of an advanced query are the record list (List output parameter) and the query count (Count output parameter) that you can manage in the Expression or Variable editors.

The type of the elements in the record list is always a Structure defined in the advanced query editor.

The query count has the total number of records returned by the query without any Max. Records limitation.

5.5 FUNCTION IN SQL

Functions in SQL enable you to perform feats such as determining the sum of a column or converting all the characters of a string to uppercase. By the end of the day, you will understand and be able to use all the following:

- Aggregate functions
- Date and time functions
- Arithmetic functions
- Character functions
- Conversion functions
- Miscellaneous functions

These functions greatly increase your ability to manipulate the information you retrieved using the basic functions of SQL that were described earlier this week. The first five aggregate functions, COUNT, SUM, AVG, MAX, and MIN, are defined in the ANSI standard. Most implementations of SQL have extensions to these aggregate functions, some of which are covered today. Some implementations may use different names for these functions.

5.6 AGGREGATION

These functions are also referred to as group functions. They return a value based on the values in a column. (After all, you wouldn't ask for the average of a single field.) The examples in this section use the table TEAMSTATS:

INPUT:

```
SQL> SELECT * FROM TEAMSTATS;
```

OUTPUT:

```
NAME        POS  AB   HITS  WALKS  SINGLES  DOUBLES  TRIPLES  HR  SO
---------   ---  ---  ----  -----  -------  -------  -------  --  --
JONES       1B   145  45    34     31       8        1        5   10
DONKNOW     3B   175  65    23     50       10       1        4   15
WORLEY      LF   157  49    15     35       8        3        3   16
DAVID       OF   187  70    24     48       4        0        17  42
HAMHOCKER   3B   50   12    10     10       2        0        0   13
CASEY       DH   1    0     0      0        0        0        0   1
```

6 rows selected.

COUNT

The function COUNT returns the number of rows that satisfy the condition in the WHERE clause. Say you wanted to know how many ball players were hitting under 350. You would type

INPUT/OUTPUT:

```
SQL> SELECT COUNT(*)
 2 FROM TEAMSTATS
 3 WHERE HITS/AB < .35;
COUNT(*)   --------
                    4
```

SUM

SUM does just that. It returns the sum of all values in a column. To find out how many singles have been hit, type

INPUT:

```
SQL> SELECT SUM(SINGLES) TOTAL_SINGLES
2 FROM TEAMSTATS;
```

OUTPUT:

```
TOTAL_SINGLES -------------
                             174
```

To get several sums, use

INPUT/OUTPUT:

```
SQL> SELECT SUM(SINGLES) TOTAL_SINGLES, SUM(DOUBLES)
TOTAL_DOUBLES,
SUM(TRIPLES) TOTAL_TRIPLES, SUM(HR) TOTAL_HR
 2 FROM TEAMSTATS;
TOTAL_SINGLES TOTAL_DOUBLES TOTAL_TRIPLES TOTAL_HR
------------- ------------- ------------- --------
 174              32                 5          29
```

To collect similar information on all 300 or better players, type

INPUT/OUTPUT:

```
SQL> SELECT SUM(SINGLES) TOTAL_SINGLES, SUM(DOUBLES)
TOTAL_DOUBLES,
SUM(TRIPLES) TOTAL_TRIPLES, SUM(HR) TOTAL_HR
```

```
2 FROM TEAMSTATS
3 WHERE HITS/AB >= .300;
TOTAL_SINGLES TOTAL_DOUBLES TOTAL_TRIPLES TOTAL_HR
------------- ------------- ------------- --------
 164            30            5            29
```

To compute a team batting average, type

INPUT/OUTPUT:

```
SQL> SELECT SUM(HITS)/SUM(AB) TEAM_AVERAGE
2 FROM TEAMSTATS;
TEAM_AVERAG------------
 .                  33706294
```

SUM works only with numbers.

AVG

The AVG function computes the average of a column. To find the average number of strike outs, use this:

INPUT:

```
SQL> SELECT AVG(SO) AVE_STRIKE_OUTS
 2 FROM TEAMSTATS;
```

OUTPUT:

```
AVE_STRIKE_OUTS ---------------
                          16.166667
```

MAX

If you want to find the largest value in a column, use MAX. For example, what is the highest number of hits?

INPUT:

```
SQL> SELECT MAX(HITS)
 2 FROM TEAMSTATS;
```

OUTPUT:

```
MAX(HITS)---------70
```

MIN

MIN does the expected thing and works like MAX except it returns the lowest member of a column. To find out the fewest at bats, type

INPUT:

```
SQL> SELECT MIN(AB)
2 FROM TEAMSTATS;
```

OUTPUT:

```
MIN(AB)---------
                1
```

The following statement returns the name closest to the beginning of the alphabet:

INPUT/OUTPUT:

```
SQL> SELECT MIN(NAME)
 2 FROM TEAMSTATS;
MIN(NAME)---------------
                     CASEY
```

You can combine MIN with MAX to give a range of values. For example:

INPUT/OUTPUT:

```
SQL> SELECT MIN(AB), MAX(AB)
 2 FROM TEAMSTATS;
 MIN(AB) MAX(AB)-------- --------
                                  1       187
```

This sort of information can be useful when using statistical functions. NOTE: As we mentioned in the introduction, the first five aggregate functions are described in the ANSI standard. The remaining aggregate functions have become de facto standards, present in all important implementations of SQL.

5.7 UPDATES IN SQL

The SQL UPDATE Query is used to modify the existing records in a table. WHERE clause can be used with UPDATE query to update selected rows otherwise all the rows would be affected.

Syntax:

The basic syntax of UPDATE query with WHERE clause is as follows:

```
UPDATE table_name
SET column1 = value1, column2 = value2...., columnN = valueN
WHERE [condition];
```

You can combine N number of conditions using AND or OR operators.

Example:

Consider CUSTOMERS table is having following records:

```
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+ ---------- +----- +-----------   +----------   +
|  1 |    Ramesh   |  32 | Ahmedabad  |  2000.00 |
|  2 |      Khilan   |  25 | Delhi             |  1500.00 |
|  3 |      kaushik  |  23 | Kota              |  2000.00 |
|  4 |      Chaitali |  25 | Mumbai            |  6500.00 |
|  5 |      Hardik   |  27 | Bhopal            |  8500.00 |
|  6 |     Komal     |  22 | MP                |  4500.00 |
|  7 |      Muffy    |  24 | Indore            | 10000.00 |
```

Following is an example which would update ADDRESS for a customer whose ID is 6:

```
SQL> UPDATE CUSTOMERS
SET ADDRESS = 'Pune'
WHERE ID = 6;
```

Now CUSTOMERS table would have following records:

```
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  6 | Komal    |  22 | Pune      |  4500.00 |
|  7 | Muffy    |  24 | Indore    | 10000.00 |
```

If you want to modify all ADDRESS and SALARY column values in CUSTOMERS table, you do not need to use WHERE clause and UPDATE query would be as follows:

```
SQL> UPDATE CUSTOMERS
SET ADDRESS = 'Pune', SALARY = 1000.00;
```

Now CUSTOMERS table would have following records:

```
| ID | NAME     | AGE | ADDRESS | SALARY  |
+----+----------+-----+---------+---------+
|  1 | Ramesh   |  32 | Pune    | 1000.00 |
|  2 | Khilan   |  25 | Pune    | 1000.00 |
|  3 | kaushik  |  23 | Pune    | 1000.00 |
|  4 | Chaitali |  25 | Pune    | 1000.00 |
|  5 | Hardik   |  27 | Pune    | 1000.00 |
|  6 | Komal    |  22 | Pune    | 1000.00 |
|  7 | Muffy    |  24 | Pune    | 1000.00 |
```

5.8 VIEWS IN SQL

A view is often referred to as a virtual table. Views are created by using the CREATE VIEW statement. After the view has been created, you can use the following SQL commands to refer to that view:

- SELECT
- INSERT
- INPUT
- UPDATE
- DELETE

An index is another way of presenting data differently than it appears on the disk. Special types of indexes reorder the record's physical location within a table. Indexes can be created on a column within a table or on a combination of columns within a table. When an index is used, the data is presented to the user in a sorted order, which you can control with the CREATE INDEX statement. You can usually gain substantial performance improvements by indexing on the correct fields, particularly fields that are being joined between tables.

Using Views you can encapsulate complex queries. After a view on a set of data has been created, you can treat that view as another table. However, special restrictions are placed on modifying the data within views. When data in a table changes, what you see when you query the view also changes. Views do not take up physical space in the database as tables do.

The syntax for the CREATE VIEW statement isSYNTAX:

```
CREATE VIEW <view_name> [(column1, column2...)] AS
SELECT <table_name column_names>
FROM <table_name>
```

A Simple View

Let's begin with the simplest of all views. Suppose, for some unknown reason, you want to make a view on the BILLS table that looks identical to the table but has a different name. (We call it DEBTS.) Here's the statement:

INPUT:

```
SQL> CREATE VIEW DEBTS AS
SELECT * FROM BILLS.
```

SQL View Processing

Views can represent data within tables in a more convenient fashion than what actually exists in the database's table structure. Views can also be extremely convenient when performing several complex queries in a series (such as within a stored procedure or application program). To solidify your understanding of the view and the SELECT statement, the next section examines the way in which SQL processes a query against a view. Suppose you have a query that occurs often, for example, you routinely join the BILLS table with the BANK_ACCOUNTS table to retrieve information on your payments.

INPUT:

```
SQL> SELECT BILLS.NAME, BILLS.AMOUNT, BANK_ACCOUNTS.BALANCE,
 2 BANK_ACCOUNTS.BANK FROM BILLS, BANK_ACCOUNTS
 3 WHERE BILLS.ACCOUNT_ID = BANK_ACCOUNTS.ACCOUNT_ID;
```

5.9 Embedded SQL and 4GLs

Embedded SQL commonly refers to what is technically known as Static SQL.

Static and Dynamic SQL

Static SQL means embedding SQL statements directly within programming code. This code cannot be modified at runtime. In fact, most implementations of Static SQL require the use of a pre-compiler that fixes your SQL statement at runtime. Both Oracle and Informix have developed Static SQL packages for their database systems. These products contain pre-compilers for use with several languages, including the following:

- C
- Pascal
- Ada
- COBOL
- FORTRAN

Some advantages of Static SQL are

- Improved runtime speed
- Compile-time error checking

The disadvantages of Static SQL are that

- It is inflexible.
- It requires more code (because queries cannot be formulated at runtime).
- Static SQL code is not portable to other database systems (a factor that you should always consider). If you print out a copy of this code, the SQL statements appear next

to the C language code (or whatever language you are using). Program variables are bound to database fields using a pre-compiler command.

This example illustrates the use of Static SQL in a C function. Please note that the syntax used here does not comply with the ANSI standard. This Static SQL syntax does not actually comply with any commercial product, although the syntax used is similar to that of most commercial products.INPUT:

```
BOOL Print_Employee_Info (void)
{
int Age = 0;
char Name[41] = "\0";
char Address[81] = "\0";
/* Now Bind Each Field We Will Select To a Program Variable */
#SQL BIND(AGE, Age)
#SQL BIND(NAME, Name);
#SQL BIND(ADDRESS, Address);
/* The above statements "bind" fields from the database to variables
from the program.
 After we query the database, we will scroll the records returned
and then print them to the screen */
#SQL SELECT AGE, NAME, ADDRESS FROM EMPLOYEES;
#SQL FIRST_RECORD
if (Age == NULL)
{
 return FALSE;
}
while (Age != NULL)
{
 printf("AGE = %d\n, Age);
 printf("NAME = %s\n, Name);
 printf("ADDRESS = %s\n", Address);
 #SQL NEXT_RECORD
}
return TRUE;
}
```

5.10 Procedural extension to SQL

One way to introduce Procedural extension into SQL is to begin by describing standard Structured Query Language, or SQL. SQL is the language that enables relational database users to communicate with the database in a straightforward manner. You can use SQL commands to query the database and modify tables within the database. When you write an SQL statement, you are telling the database what you want to do, not how to do it. The query optimizer decides the most efficient way to execute your statement. If you send a series of SQL statements to the server in standard SQL, the server executes them one at a time in chronological order.PL/SQL is Oracle's procedural language; it comprises the standard language of SQL and a wide array of commands that enable you to control the execution of SQL statements according to different conditions. PL/SQL can also handle runtime errors. Options such as loops and IF...THEN statements give PL/SQL the power of third-generation programming languages. PL/SQL allows you to write interactive, user-friendly programs that can pass values into variables. You can also use several predefined packages, one of which can display messages to the user.

PL/SQL is a block-structured language, meaning that PL/SQL programs are divided and written in logical blocks of code. Within a PL/SQL block of code, processes such as data manipulation or queries can occur. The following parts of a PL/SQL block are discussed in this section:

● The DECLARE section contains the definitions of variables and other objects such as constants and cursors. This section is an optional part of a PL/SQL block.

● The PROCEDURE section contains conditional commands and SQL statements and is where the block is controlled. This section is the only mandatory part of a PL/SQL block.

● The EXCEPTION section tells the PL/SQL block how to handle specified errors and user-defined exceptions.

This section is an optional part of a PL/SQL block.

5.11 Introduction to Query-by-example(QBE)

Query-by-Example (QBE) is another language for querying (and, like SQL, for creating and modifying) relational data. It is different from SQL, and from most other database query languages, in having a graphical user interface that allows users to write queries by creating example tables on the screen. A user needs minimal information to get started and the whole language contains relatively few concepts. QBE is especially suited for queries that are not too complex and can be expressed in terms of a few tables.

5.12 SUMMARY

SQL is a standard query language for accessing and manipulating databases. SQL stands for Structured Query Language. It is a procedure oriented language. This unit has presented the basics of the SQL language. Here have discussed the data definitions, constraints and schemes changes in SQL. We have expressed many queries in SQL.

5.13 KEYWORDS

SQL: It stands for Structured Query Language.

Database Table: A database contains one or more tables. Each table is identified by a name. Tables contain records (rows) with data.

View: A view is a virtual relation.

5.14 UNIT-END EXERCISES AND ANSWERS

1. Write a note SQL syntax, with suitable examples.
2. Express some queries in SQL.
3. Define advance queries in SQL
4. Explain Functions in SQL
5. Explain about views ad embedded SQL.
6. Define procedural extension to SQL ad QBE.

Answer: SEE

1.	5.2
2.	5.3
3.	5.4

4.	5.5
5.	5.8,5.9
6.	5.10,5.11

5.15 SUGGESTED READINGS

[1] Fundamentals of Database Systems By Ramez Elmasri, Shamkant B. Navathe, Durvasula V.L.N. Somayajulu, Shyam K.Gupta

[2] Database System Concepts By Avi Silberschatz, Henry F. Korth , S. Sudarshan

[3] Database Management Systems By Raghu Ramakrishnan and Johannes Gehrke

UNIT 6: Database Design

Structure:

6.0 OBJECTIVES

At the end of this unit you will be able to know:

- Database Design
- Functional Dependence
- Normal Forms;
- Normalization
- Second normal form
- Third normal form
- Boyce-Codd normal form

6.1 INTRODUCTION

The goal of relational database design is to generate a set of relation schemes with some desirable properties. These schemes generated should allow us to store information without unnecessary redundancy. They should also allow us to retrieve information easily. Here, we deal with the goodness of a database design.

6.2 Database design process

The six main phases of the database design process:

- Requirements connection and analysis - involves the collection of information concerning the intended use of the database
- Conceptual database design - The goal of this phase is to produce a conceptual schema for the database that is independent of a specific DBMS. In addition to specifying the conceptual schema, we should specify as many of the know database applications or transactions as possible. These applications should also be specified using a notation that is independent of any specific DBMS.
- Choice of a DBMS

The following costs must be considered when selecting a DBMS:

a) Software acquisition cost.

b) Maintenance cost. The cost of receiving standard maintenance service from the vendor and for keeping the DBMS version up to date.

c) Hardware acquisition cost. New hardware such as additional memory, terminal, disk units, or specialized DBMS storage may be needed.

d) Database creation and conversion cost: The cost of using the DBMS software either to create the database system form scratch or to convert an existing system to the new DBMS software.

e) Personnel cost. New positions of database administrator (DBA) and staff are being created in most companies using DBMS's.

f) Training cost.

g) Operating cost. The cost of continued operation of the database system is typically not worked into an evaluation of alternative because it is incurred regardless of the DBMS selected.

- Data model mapping (also called logical database design) - During this phase we map (or transform) the conceptual schema from the high - level data model used in phase 2 into the data model of the DBMS chosen in phase 3. In addition, the design of external schemas (views) for specific applications is often done during this phase.
- Physical database design- During this phase we design the specifications for the stored database in terms of physical storage structures, record placement, and access paths. This corresponds to the design of the internal schema in the terminology of the three-level DBMS architecture.
- Database implementation

6.3 Relational database design

When we design a relational database, there may be some undesirable properties. A good database design must concentrate on removing or minimizing undesirable properties.

Some of the informal database schema measures are:

- Keeping the semantics of the attributes
- Reducing the repetition of information
- Avoiding inability to represent certain information
- Avoiding the generation of spurious tuples

6.3.1 Keeping the semantics of the attributes:

When we design a relation schema, we must assume that attributes in the schema have certain real world meaning and a proper interpretation associated with them. It is advisable to design a schema so that it is easy to explain its meaning. When doing so, keep the attributes of one entity type and relationship type in a single relation. It is not advisable to combine attributes from multiple entity types and relationship types into a single relation. If we mix the attributes of multiple entities and relationships, it will lead semantic ambiguity. And the resulting relation cannot be easily explained.

6.3.2 Reducing the repetition of information:

One important goal of schema design is to minimize the storage space used by the relations. In a relational database, repetition of Information is a condition where the values of one attribute are determined by the values of another attribute in the same relation, and both values are repeated throughout the relation. Repetition of information has two problems associated with it. 1) It wastes storage 2) It makes updating the relation more difficult. This is a bad relational database design

Consider an alternate scheme for branch scheme and deposit scheme as:

Deposit_Info_Scheme and the corresponding relation as shown:

Branch_Name	Account_No	Customer_Name	Balance	Assets	City
University	20	Rama	12,000	**35,00,00,000**	**Mysore**
University	25	Sita	11,000	**35,00,00,000**	**Mysore**
Majestic	35	Krishna	16,000	**55,00,00,000**	**Bangalore**
Majestic	56	Gocinda	55,999	**55,00,00,000**	**Bangalore**
Majestic	70	Hamsa	76,988	**55,00,00,000**	**Bangalore**
:					
:					

Note that for branch name, the assets value and the city value are repeated for every account in that branch.

6.3.3 Inability to represent certain information:

Consider the above relation scheme, in which it is not possible to represent any branch information, if there are no account holders in that branch. Inability to represent information is a condition where a relationship exists among only a proper subset of the attributes in a relation. This is bad relational database design.

6.3.4 Avoiding the generation of spurious tuples:

A spurious tuple is a record in database that gets created when two tables are joined badly. We may get spurious tuples when we join tables on attributes that are neither primary keys nor foreign keys

For example, consider a student relation scheme and the corresponding relation as:

Student:

Roll_Number	Name	Age	Sex	Total_Marks
1234	Rama	50	M	450
4321	Sita	50	F	480

Suppose we decompose the student relation into:

student_part1

Roll_Number	Name	Age
1234	Rama	50
4321	Sita	50

student_part2

Age	Sex	Total_Marks
50	M	450
50	F	480

Now, consider the natural join of student_part1 and student_part2

We get:

Roll_Number	Name	Age	Sex	Total_Marks
1234	Rama	50	M	450
1234	**Rama**	**50**	**F**	**480**
4321	Sita	50	M	450
4321	**Sita**	**50**	**F**	**480**

Note that the tuples with bold are spurious and got generated because of bad decomposition.

The above discussion shows that we must produce the relation schemes (database design) which address the above issues.

A good database design must achieve:

- No repetition of information (if at all possible)
- Lossless join. (is a must)
- Dependency preservation. (if at all possible)

Note that achieving all of them always, may not be possible. To what extent we can achieve them depends upon how good our database design is. In order to understand the goodness and badness of relational schemas and their analysis, we start with functional dependency which can be used as tool.

6.4 Anomalies in a database

Databases are designed to collect data and sort or present it in specific ways to the end user.Database anomalies-unmatched or missing information caused by limitations or flaws within a given database. Entering or deleting information cause issues if the database is limited or has ‘bugs’.

6.4.1 Modification/Update Anomalies

They are the data inconsistencies that resulted from data redundancy or partial update. The Problems resulting from data redundancy in database table are known as update anomalies. If a modification is not carried out on all the relevant rows, the database will become inconsistent. Any database insertion, deletion or modification that leaves the database in an inconsistent state is said to have caused an update anomaly.

6.4.2 Insertion Anomalies

To insert the information into the table, one must enter the correct details; information must be consistent with the values for the other rows; missing or incorrectly formatted entries are two of the more common insertion errors. Most developers acknowledge that this will happen and build in error codes that tell you exactly what went wrong.

6.4.3 Deletion Anomalies

Issues with data being deleted either – when attempting to delete and being stopped by an error or – by the unseen drop off of data. If we delete a row from the table that represents the last piece of data, the details about that piece are also lost from the Database. These are the least likely to be caught or to stop you from proceeding. As many deletion errors go unnoticed for extended periods of time, they could be the most costly in terms of recovery

6.5 Functional dependencies

A functional dependency (FD) is a constraint between two sets of attributes in a relation from a database.

Definition: Let R be a relation scheme (a set of attributes). Let α and β be subsets of R. We say that α functionally determines β (written as $\alpha \rightarrow \beta$) if each α value is associated with precisely one β value. α is called determinant set and β the dependent set.

The meaning of FD: $\alpha \rightarrow \beta$:: α functionally determines β in a relation R if , and only if, whenever two tuples of r(R) agree on their α - value, they must agree on their β -value.
i.e., for any two tuples t_1 and t_2 in r if $t_1[\alpha] = t_2[\alpha]$, they must also have $t_1[\beta] = t_2[\beta]$.
Thus, a candidate key is a minimal set of attributes that functionally determines all of the attributes in a relation.
If $\alpha \rightarrow \beta$ in R, it may not be necessary to have $\beta \rightarrow \alpha$.
For example, consider the following relation scheme:

Branch_Name	Assets	Branch_City
Majestic	20,00,00,000	Bangalore
Saraswathipuram	10,00,00,000	Mysore
Redfort	30,00,00,000	Delhi
Noriman	12,00,00,000	Bombay
Shivaji	30,00,00,000	Bombay

Note that the following functional dependencies, which are true in the given relation scheme.
Branch_name → Assets
Branch_name → Branch_City

But **Branch_City → Branch_name** is not true. Similarly **Assets → Branch_name** is not true.
Trivial functional dependency: A functional dependency FD: $\alpha \rightarrow \beta$ is called trivial if β is a subset of α.

6.6 Canonical cover

While updating the database we need to verify all the functional dependencies. In order to minimize this verification cost, we make use of canonical cover, which is defined as:
A functional depending set F_C for F is Canonical Cover (minimal or irreducible) if the set has following three properties:

1. Each right set of a functional dependency of F_C contains only one attribute (unique).
2. Each left set of a functional dependency of F_C is irreducible. It means that reducing any one attribute from left set will change the content of F_C (contains no extraneous attribute in the left side).
3. Reducing any functional dependency will change the content of F_C.

Every set of functional dependencies has a canonical cover. Now instead of testing F, we can test $F_{C.}$

Algorithm to obtain Canonical Cover:

Let F be given set of Fds.

1. Set $F_C := F$
2. Replace each FD $\alpha \rightarrow \{A_1, A_2,\ldots,A_n\}$ in F_C by the n FDs $\alpha \rightarrow A_1$, $\alpha \rightarrow A_2$, …, $\alpha \rightarrow A_n$.
3. For each FD $\alpha \rightarrow A$ in F_C for each attribute B that is an element of α if $\{ \{ F_C - \{ \alpha \rightarrow A \} \} \cup \{ (\alpha - \{B\}) \rightarrow A \}$ is equivalent to F, then replace $\alpha \rightarrow A$ with $(\alpha - \{B\}) \rightarrow A$ in F_C
4. For each remaining FD $\alpha \rightarrow A$ in F_C if $\{F – \{ \alpha \rightarrow A \} \}$ is equivalent to F, them remove $\alpha \rightarrow A$ from F_C.

6.7 Normal forms

The concept of normalization was introduced by Edgar F. Codd. Normalization is the process of organizing data to minimize redundancy to improve storage efficiency, data integrity, and scalability. The goal of database normalization is to decompose relations with anomalies in order to produce smaller, well-structured relations. The objective is to isolate data so that insertions, deletions, and modifications of a field value can be made in just one table and then propagated through the rest of the database via the defined relationships.

In the relational model, methods exist for quantifying how efficient a database is. These classifications are called normal forms (or NF), and there are algorithms for converting a given database between them. Normalization generally involves splitting existing tables into multiple ones, which must be re-joined or linked each time a query is issued.

Normalization: In the design of a relational database management system, the process of organizing data to minimize redundancy is called normalization.

6.8 First Normal Form

First normal form (1NF) sets the very basic rules for an organized database:

The rules are:

1. Eliminate duplicative columns from the same table.

2. Create separate tables for each group of related data and identify each row with a unique column or set of columns (the primary).

The rules mean the following:

The first rule dictates that we must not duplicate data within the same row of a table. This concept is referred to as the atomicity of a table. Tables that satisfy with this rule are said to be atomic.

For example consider teaching department database that stores the class_teacher-student relationship. Assume that each class_teacher may have one or more students while each student may have only one class_teacher.

We might create a table with the following fields:

- Class_teacher
- Student1
- Student2
- Student3
- Student4

However, recall the first rule imposed by 1NF: eliminate duplicative columns from the same table. Clearly, the Student1-Student4 columns are duplicative. Look at the problem with this table. If a class_teacher has only one student – the Stduent2-Student4 columns are simply wasted storage space (a precious database commodity). Furthermore, imagine the case where a class_teacher already has 4 students – what happens if she takes on another student? The whole table structure would require modification. At this point, a second bright idea usually occurs to database novices: We do not want to have more than one column and we want to allow for a flexible amount of data storage. Let us try something like this:

- Class_teacher
- Students

Where the Student field contains multiple entries in the form "Rama, Sita, Asha, John, Joe"

This solution is closer, but it also has problem. The Student column is still duplicative and non-atomic. What happens when we need to add or remove a student? We need to read and write the entire contents of the table. It becomes a problem if class teacher has more students. Also, it complicates the process of selecting data from the database in future queries.

Here is a table that satisfies the first rule of 1NF:

- Class_teacher
- Student

In this case, each student has a single entry, but class teacher may have multiple entries.

Now, let us look at the second rule: identify each row with a unique column or set of columns (the primary key)? We may select student column as primary key. In fact, the student column is a good candidate for a primary key. However, the data that we may receive to store in our table makes this a less than ideal solution. What happens if we get another student named Rama? How do we store his class_teacher-student relationship in the database?

It is advisable to use a truly unique identifier (such as a student ID) as a primary key. Our final table would look like this:

- Class_teacher ID
- Student ID

Now, our table is in first normal form.

6.9 Second Normal Form and Third Normal Form

A 1NF table is in 2NF if and only if, given any candidate key K and any attribute A that is not a constituent of a candidate key, A depends upon the whole of K rather than just a part of it.

Formally a 1NF table is in 2NF if and only if all its non-prime attributes are functionally dependent on the whole of every candidate key (A non-prime attribute is one that does not belong to any candidate key.).

Note that when a 1NF table has no composite candidate keys (candidate keys consisting of more than one attribute), the table is automatically in 2NF.

Consider the following table:

Student_name	Skill	Address
Rama	Java	123, Palace Street, Mysore
Rama	C++	123, Palace Street, Mysore
Rama	Linux	123, Palace Street, Mysore
Sita	Pascal	321, Racecourse Road, Bangalore
Sita	C	321, Racecourse Road, Bangalore
John	C++	456, Sastri Street, Dehli

Quing	Windows	345, MG Road, Bangalore

Neither {Student_name} nor {Skill} is a candidate key.

This is because a given student might need to appear more than once (he might have multiple Skills), and a given Skill might need to appear more than once (it might be possessed by multiple students). Only the composite key {Student_name, Skill} qualifies as a candidate key for the table (assuming all students names are unique).

The remaining attribute, Address, is dependent on only part of the candidate key, namely Student_name. Therefore the table is not in 2NF. Note the redundancy in the way Addresses are represented: we are told three times that Rama's address is 123, Palace Street, Mysore, and twice that Sita's address 321, Racecourse Road, Bangalore. This redundancy makes the table vulnerable to update anomalies.

A 2NF alternative to this design would represent the same information in two tables: an "Students" table with candidate key {Student_name}, and an "Student' Skills" table with candidate key {Student_name, Skill}:

Students table:

Student_name	Address
Rama	123, Palace Street, Mysore
Rama	123, Palace Street, Mysore
Rama	123, Palace Street, Mysore
Sita	321, Racecourse Road, Bangalore
Sita	321, Racecourse Road, Bangalore
John	456, Sastri Street, Dehli
Quing	345, MG Road, Bangalore

Student' Skills table:

Student_name	Skill
Rama	Java
Rama	C++
Rama	Linux
Sita	Pascal
Sita	C

John	C++
Quing	Windows

Third Normal Form

Let R be relation scheme (a set of attributes). Let α and β be subsets of R. Let F be set functional dependencies holding on R and F^+ be the closure of F.

A relation scheme R is in Third Normal Form (3NF) with respect to a set F of functional dependencies if, for all functional dependencies in F^+ of the form $\alpha \rightarrow \beta$, at least one the following holds:

- $\alpha \rightarrow \beta$ is a trivial FD
- α is a superkey for R
- Each attribute A in β - α is contained in a candidate key.

For example, consider the relation scheme R = (A, B, C) with F = {C → A, (B, A) → C}.

Note that R is in 3 NF, because {B, A} is candidate key and for FD C → A, A ∈ {B, A}.

6.10 Boyce-Codd Normal Form

Let R be relation scheme (a set of attributes). Let α and β be subsets of R. Let F be set functional dependencies holding on R and F^+ be the closure of F.

A relation scheme R is in Boyce-Codd Normal Form (BCNF) with respect to a set F of functional dependencies if, for all functional dependencies in F^+ of the form $\alpha \rightarrow \beta$, at least one the following holds:

- $\alpha \rightarrow \beta$ is a trivial FD
- α is a superkey for R

For example, consider the relation scheme R = (A, B, C) with F = {C → A, (B, A) → C}.

Note that R is not in BCNF, because C → A is neither trivial nor C is a superkey. Where as R_1 = {A,C} and R_2 = {B, C} are both in BCNF.

6.11 Reduction of an E-R schema to Tables

A database that conforms to an E-R database schema can be represented by a collection of tables. For each entity set and for each relationship set, there is a unique table. A table is a chart with rows and columns. The set of all possible rows is the *Cartesian product* of all columns.

A row is also known as a *tuple* or a *record*. A table has an unlimited number of rows. Each column is also known as a *field*.

Strong Entity Sets

It is common practice for the table to have the same name as the entity set. There is one column for each attribute.

Weak Entity Sets

There is one column for each attribute, plus the attribute(s) the form the primary key of the strong entity set that the weak entity set depends upon.

Relationship Sets

We represent a relationship with a table that includes the attributes of each of the primary keys plus any descriptive attributes (if any).

There is a problem that if one of the entities in the relationship is a weak entity set, there would be no unique information in the relationship table, and therefore may be omitted.

Another problem can occur if there is an existance dependency. In that case, you can combine the two tables.

Multivalued Attributes

When an attribute is multivalued, remove the attribute from the table and create a new table with the primary key and the attribute, but each value will be a separate row.

Generalization

Create a table for the higher-level entity set. For each lower-level entity set, create a table with the attributes for that specialization and include the primary key from the higher-level entity set.

6.12 SUMMARY

The goal of relational database design is to generate a set of relation schemes with some desirable properties. In this unit we have discussed about the informal design guidelines for relational schemes. We have also discussed about functional dependency.

Normalization is the process of organizing data to minimize redundancy to improve storage efficiency, data integrity, and scalability. The goal of database normalization is to decompose relations with anomalies in order to produce smaller, well-structured relations. In this unit we have discussed about Normalizations and different normal forms.

6.13 KEYWORDS

Functional Dependency: A functional dependency (FD) is a constraint between two sets of attributes in a relation from a database

Canonical cover: A functional depending set F_C for F is Canonical Cover if it satisfies minimal (irreducible) property.
Normalization: The process of organizing data to minimize redundancy.

1NF: First Normal Form
2NF: Second Normal Form
3NF: Third Normal Form
BCNF: Boyce-Codd Normal Form

6.14 UNIT-END EXERCISES AND ANSWERS

1. What are the steps in data design process and explain about relational database design?
2. Explain the concept functional dependency.
3. What is Anomalies in a database and conical cover?
4. What is normalization?
5. Define 1NF.
6. Define 2NF and Explain 3NF
7. Explain BCNF
8. Explain Reduction of an E-R schema to Tables

Answers: SEE

1.	6.2,6.3
2.	6.5
3.	6.4,6.6
4.	6.7
5.	6.8

6. 6.9
7. 6.10
8. 6.11

6.15 SUGGESTED READINGS

[1] Fundamentals of Database Systems By Ramez Elmasri, Shamkant B. Navathe, Durvasula V.L.N. Somayajulu, Shyam K.Gupta

[2] Database System Concepts By Avi Silberschatz, Henry F. Korth , S. Sudarshan

[3] Database Management Systems By Raghu Ramakrishnan and Johannes Gehrke

[4] An Introduction to Database Systems C.J.Date

UNIT 7: File Organization, Indexing and Hashing

Structure:

7.0 OBJECTIVES

At the end of this unit you will be able to know:

- File Organizations
- Hashing
- Indexing
- Buffer Management
- Describe a number of different types of indexes commonly found in modern database environments;
- Understand the data structures which can support the various indexes;
- Be aware of the proper ways in which indexes are used;

- Understand the typical approaches used in industry to improve database performance through indexing.

7.1 INTRODUCTION

Data storage refers to computer data storage which includes memory, components, devices and media that retain digital computer data used for computing for some interval of time. Data storage is device that records (stores) or retrieves (reads) information from any medium.

Databases are typically stored on magnetic disks as files of records. In this unit we deal with the organization of databases in storage and the techniques for accessing them efficiently.
The databases must be stored physically on some computer storage medium. Computer storage media form a hierarchy that includes two main categories.
Primary storage: This includes computer main memory and cache memories. This storage media can be operated directly by the CPU. Primary storage provides fast access to data but is limited by size, volatile in nature and costly too.

7.2 Overview of file organization techniques

In this unit, we are concerned with obtaining data representation for files on external storage devices so that required functions (e.g. retrieval, update) may be carried out efficiently. The particular organization most suitable for any application will depend upon such factors as the kind of external storage available, types of queries allowed, number of keys, mode of retrieval and mode of update.

7.3 Secondary storage devices

The basic question is why does a DBMS store data on external storage?
The answer is a DBMS stores data on external storage because the quantity of data is vast, and must persist across program executions. These external storages are called secondary storage.

Secondary storage: This includes tapes, magnetic disks and optical disks. These devices provide slower access to data than do primary storage devices. But they provide large storage and cheaper.

In secondary storage cannot be directly processed by the CPU. For processing it should be brought to primary storage.

Storage Hierarchy: There is a range of memory and storage devices within the computer system. The following list starts with the slowest devices and ends with the fastest.

VERY SLOW;

Punch cards (obsolete)

Punched paper tape (obsolete)

FASTER:

Bubble memory

Floppy disks

MUCH FASTER:

Magnetic tape

Optical discs (CD-ROM, DVD-ROM, MO, etc.)

Magnetic disks with movable heads

Magnetic disks with fixed heads (obsolete)

Low-speed bulk memory

FASTEST:

Flash memory

Main memory

Cache memory

Microcode

Registers

Storage of Databases:

The data in most of the databases is large and must be stored persistently over long period of time. The data stored in database is accessed and processed repeatedly during this period. Most databases are stored on magnetic disks permanently. Some of the reasons for this are:

- Databases are generally too large to fit entirely in main memory.
- The data in databases required to be stored permanently on nonvolatile storage.
- The cost of storage very less compared to primary storage.

It is possible to design effective databases with acceptable performances by understanding the properties and characteristics of magnetic disks and the way data files can be organized on disks.

For backing up of the databases, magnetic tapes are frequently used as storage medium, because the storage on magnetic tapes costs very less compared disks. But access to data on tape is very slow. Data stored on tape is offline. It requires loading of data whenever data is needed. In contrast, the disks are online devices that can be accessed directly at any time.

When the DBA and database designer design, implement, and operate a database on specific DBMS must know the advantages and disadvantages of each storage technique.

7.4 Heap files and sorted files

A heap file is an unordered set of records. Rows are simply appended to the end of the file as they are inserted. Hence the file is unordered. Deleted rows will create gaps in file. File then must be periodically compacted to recover space. Tuples are stored only as a series of bytes. It is up to upper layers to make sense of them.

Heap file services:

The basic services are:

- create a data record giving the data size needed and return its identifier
- delete a data record giving its identifier
- read a data record giving identifier and returning associated data
- write a data record giving identifier and associated data

Sequential scans on heap files are also supported.

Heap file structure:

Heap file contains a list of one header record followed by data or free record.

Header
Record
Record
…………
Record

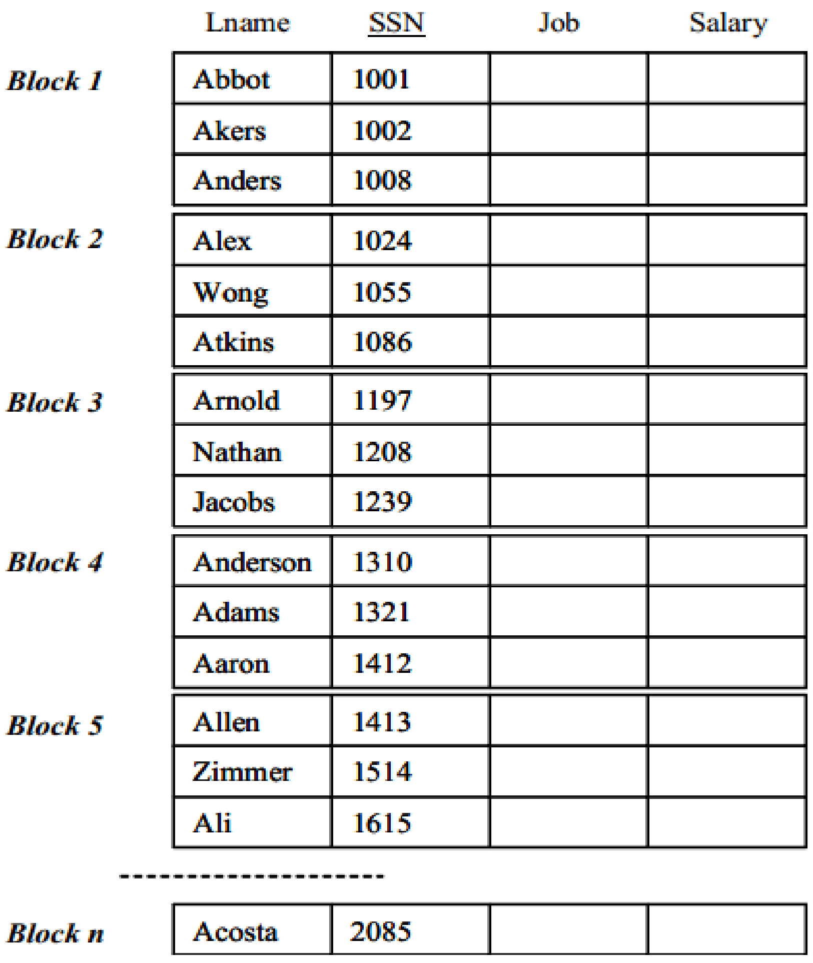
Each data or free record has:

- A type indicator (free or data record)
- An area size
- And a previous record position in file

Area size and previous record position in file make able to linearly go to previous or next record.

Sorted files

The simplest method to make efficient searching is to organize the table in a Sorted File. The entire file is sorted by increasing value of one of the attributes, called the ordering attribute (or ordering field). If the ordering field) is also a key attribute, it is called the ordering key. Figure 7 shows an example of a table sorted by the "SSN" attribute.



	Lname	SSN	Job	Salary
Block 1	Abbot	1001		
	Akers	1002		
	Anders	1008		
Block 2	Alex	1024		
	Wong	1055		
	Atkins	1086		
Block 3	Arnold	1197		
	Nathan	1208		
	Jacobs	1239		
Block 4	Anderson	1310		
	Adams	1321		
	Aaron	1412		
Block 5	Allen	1413		
	Zimmer	1514		
	Ali	1615		

Block n	Acosta	2085		

A table (only part of the data is shown) sorted by ‘SSN’ as ordering attribute.

To Search for a record with a given value of the ordering attribute lets us binary search.

For a file of b blocks, look in the block number $\lceil b/2 \rceil$

If the searched value is in this block, we are done;

If the searched value is larger than the last ordering attribute value in this block,
repeat the binary search in blocks between ($\lceil b/2 \rceil$ + 1) and b;
otherwise repeat the search in blocks between 1 and ($\lceil b/2 \rceil$ - 1).

Performance: Let us consider the worst case time spent for a sorted file of size b blocks. In each iteration, we need to check one block (= t units of time); Worst case Î we do not find the record in until the last remaining block. In the 2nd iteration, at most b/2 blocks will remain to be searched; in the 3rditeration, (b/2)/2 = b/22 blocks remain. After i-iterations, b/2i-1 blocks remain to be searched. Of course, if only one block remains to be searched, no further sub-division of the list of blocks is required, namely, we stop when b/2i-1 = 1, which gives: b = 2i-1, or i = (1 + lg2b). Total number of blocks searched after iiterations is i, and the total time is t(1 + lg2b). Let's compare the relative worst-case search time between heap storage and sorted file storage. Let's assume a file occupies 8192 blocks. Stored as a heap file, the worst case time is 8192t; stored as a sorted file, the worst case time is t(1 + lg2 8192) = t(1+ 13) = 14t. The search is 8192/14 ≈585 times faster!

7.5 Indexing and Hashing- Basic concepts

Indexes are additional auxiliary access structures on already organized files (with some primary organization). Indexes speed up the retrieval of records under certain search conditions.

This unit extends the discussions we had in the previous unit. We have studied how files and records can be placed on disks, and what are the effective ways in which records can be organized in files. The three file organizations we learned in that unit were heap file, sorted file, and hash file. We have seen that there was only one goal in applying those various file organization techniques that was to provide the database system with good performance.

Database performance is again the issue around which we are going to carry out our discussions in this unit. It is not only important that a database is designed well, but part of the design process also involves ensuring that the structure of the developed system is efficient enough to satisfy users' requirements now and into the future.

Indexes play a similar role in database systems as they do in books in that they are used to speed up access to information. File structures can be affected by different indexing techniques, and they in turn will affect the performance of the databases.

It is worth emphasizing again the importance of file organizations and their related access methods. Tuning techniques can help improve performance, but only to the extent that is

allowed by a particular file organization. Indexes can help database developers build efficient file structures and offer effective access methods. When properly used and tuned, the database performance can be improved further. In fact, indexes are probably the single most important mechanism explicitly available to database developers and administrators for tuning the performance of a database.
We will introduce some new access structures called indexes, which are used to speed up the retrieval of records if certain requirements on search conditions are met. An index for a file of records works just like an index catalogue in a library.Each index (access structure) offers an access path to records. Some types of indexes, called secondary access paths, do not affect the physical placement of records on disk; rather, they provide alternative search paths for locating the records efficiently based on the indexing fields. Other types of indexes can only be constructed on a file with a particular primary organization.
It must be emphasized that different indexes have their own advantages and disadvantages. There is no universally efficient index. Each technique is best suited for a particular type of database applications.
The merits of indexes can be measured in the following aspects:

- Access types: the kind of access methods that can be supported efficiently (e.g., value-based search or range search).
- Access time: time needed to locate a particular record or a set of records.
- Insertion efficiency: how efficient is an insertion operation.
- Deletion efficiency: how efficient is a deletion operation.
- Storage overhead: the additional storage requirement by an index structure.

It is worth noting that a file of records can have more than one index, just like for books there can be different indexes such as author index and title index. An index access structure is usually constructed on a single field of record in a file, called an indexing field. Such an index typically stores each value of the indexing field along with a list of pointers to all disk blocks that contain records with that field value. The values in the index are usually sorted (ordered) so that we can perform an efficient binary search on the index.
An index file is much smaller than the data file, and therefore, searching the index using a binary search can be carried out quickly.
In a hash file organization we obtain the bucket of a record directly from its search-key value using a hash function. Hash function h is a function from the set of all search-key values K to the set of all bucket addresses B. Hash function is used to locate records for access, insertion

as well as deletion. Records with different search-key values may be mapped to the same bucket; thus entire bucket has to be searched sequentially to locate a record.

Example of Hash File Organization: There are 10 buckets, The binary representation of the i th character is assumed to be the integer i. The hash function returns the sum of the binary representations of the characters modulo 10 E.g. h(Perryridge) = 5 h(Round Hill) = 3 h(Brighton) = 3 Hash file organization of account file, using branch-name as key (See figure in next slide.)

7.6 Static Hashing

A bucket is a unit of storage containing one or more records (a bucket is typically a disk block). In a hash file organization we obtain the bucket of a record directly from its search-key value using a hash function. Hash function h is a function from the set of all search-key values K to the set of all bucket addresses B. Hash function is used to locate records for access, insertion as well as deletion. Records with different search-key values may be mapped to the same bucket; thus entire bucket has to be searched sequentially to locate a record.

7.7 Dynamic Hashing and Extendible hashing

Some hashing techniques allow the hush function to be modified dynamically to accommodate the growth or shrinking of the database. These are called dynamic hash functions.

Extendible hashing is one form of dynamic hashing, and it works in the following way:

- We choose a hash function that is uniform and random. It generates values over a relatively large range.
- The hash addresses in the address space (i.e., the range) are represented by d-bit binary integers (typically d = 32). As a result, we can have a maximum of 232 (over 4 billion) buckets.
- We do not create 4 billion buckets at once. Instead, we create them on demand, depending on the size of the file. According to the actual number of buckets created, we use the corresponding number of bits to represent their address. For example, if there are 4 buckets at the moment, we just need 2 bits for the addresses (i.e., 00, 01, 10, and 11).
- At any time, the actual number of bits used (denoted as i and called global depth) is between 0 (for one bucket) and d (for maximum 2d buckets).

- Value of i grows or shrinks with the database, and the i binary bits are used as an offset into a table of bucket addresses (called a directory). In Figure 6.3, 3 bits are used as the offset (i.e., 000, 001, 010, ..., 110, 111).
- The offsets serve as indexes pointing to the buckets in which the corresponding records are held. For example, if the first 3 bits of a hash value of a record are 001, then the record is in the bucket pointed by the 001 entry.
- It must be noted that there does not have to be a distinct bucket for each of the 2i directory locations. Several directory locations (i.e., entries) with the same first j bits (j <= i) for their hash values may contain the same bucket address (i.e., point to the same bucket) if all the records that hash to these locations fit in a single bucket.
- j is called the local depth stored with each bucket. It specifies the number of bits on which the bucket contents are based. In Figure 6.3, for example, the middle 2 buckets contain records which have been hashed based on the first 2 bits of their hash values (i.e., starting with 01 and 10), while the global depth is 3.

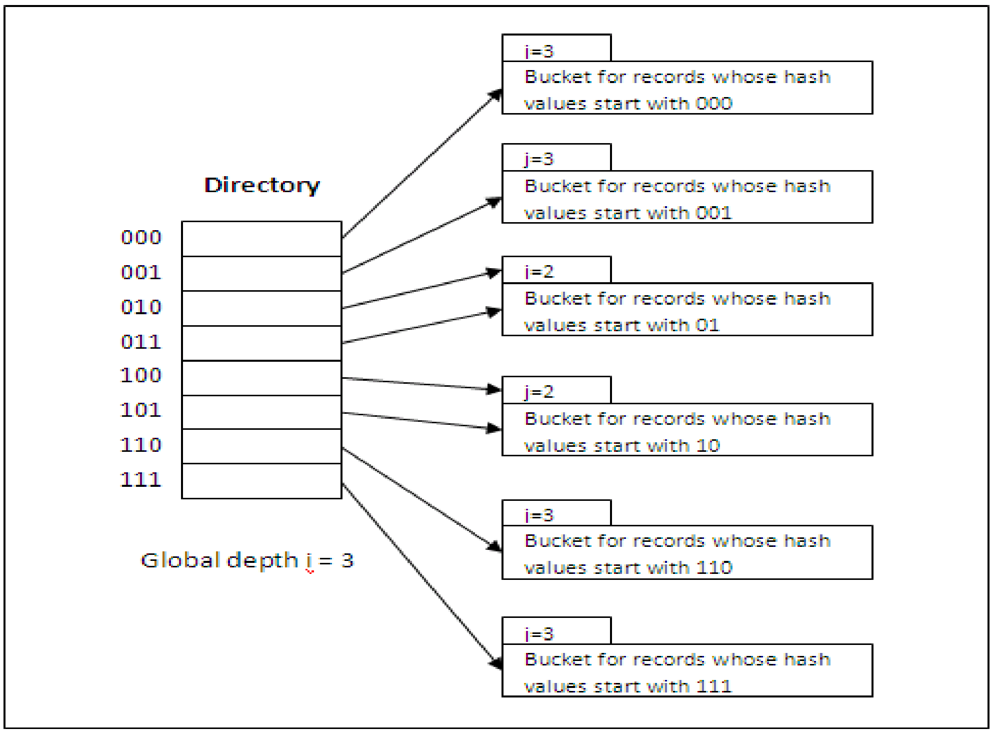
The value of i can be increased or decreased by one at a time, thus doubling or halving the number of entries in the directory table. Doubling is needed if a bucket whose local depth j is equal to the global depth i overflow. Halving occurs if i > j for all the buckets after some deletions occur.
Retrieval - to find the bucket containing the search key value K:

- Compute h(K).
- Take the first i bits of h(K).
- Look at the corresponding table entry for this i-bit string.
- Follow the bucket pointer in the table entry to retrieve the block.

Insertion - to add a new record with the hash key value K:

- Follow the same procedure for retrieval, ending up in some bucket.
- If there is still space in that bucket, place the record in it.
- If the bucket is full, we must split the bucket and redistribute the records.
- If a bucket is split, we may need to increase the number of bits we use in the hash.

To illustrate bucket splitting (see the figure below), suppose that a new record to be inserted causes overflow in the bucket whose hash values start with 01 (the third bucket in Figure X.X). The records in that bucket will have to be redistributed among two buckets: the first contains all records whose hash values start with 010, and the second contains all those whose hash values start with 011. Now the two directory entries for 010 and 011 point to the two new distinct buckets. Before the split, they point to the same bucket. The local depth of the two new buckets is 3, which is one more than the local depth of the old bucket.


Directory
000
001
010
011
100
101
110
111
Global depth i = 3
j=3
Bucket for records whose hash values start with 000
j=3
Bucket for records whose hash values start with 001
j=2
Bucket for records whose hash values start with 01
j=2
Bucket for records whose hash values start with 10
j=3
Bucket for records whose hash values start with 110
j=3
Bucket for records whose hash values start with 111

If a bucket that overflows and is split used to have a local depth j equal to the global depth i of the directory, then the size of the directory must now be doubled so that we can use an extra bit to distinguish the two new buckets. In the above figure, for example, if the bucket for records whose hash values start with 111 overflows, the two new buckets need a directory with global depth i = 4, because the two buckets are now labeled 1110 and 1111, and hence their local depths are both 4. The directory size is doubled and each of the other original entries in the directory is also split into two, both of which have the same pointer as did the original entries.

Deletion may cause buckets to be merged and the bucket address directory may have to be halved.

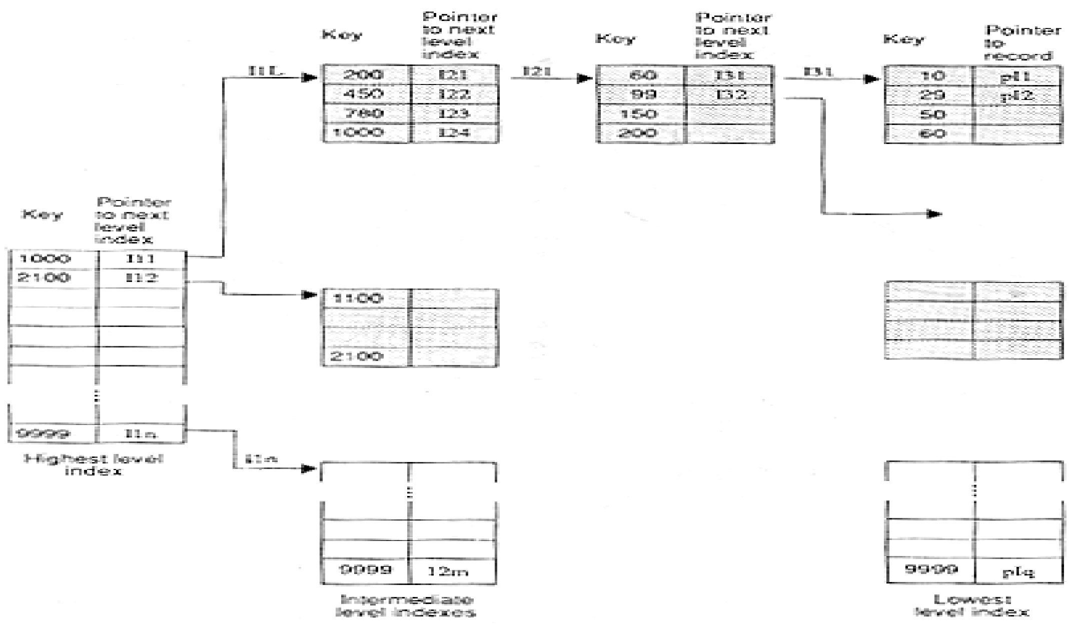
In general, most record retrievals require two block accesses – one to the directory and the other to the bucket. Extendible hashing provides performance that does not degrade as the file grows. Space overhead is minimum, because no buckets need be reserved for future use. The bucket address directory only contains one pointer for each hash value of current prefix length (i.e., the global depth). Potential drawbacks are that we need to have an extra layer in the structure (i.e., the directory) and this adds more complexities.

7.8 Single level ordered indexes

There are several types of single-level ordered indexes. A primary index is an index specified on the ordering key field of a sorted file of records. If the records are sorted not on the key field, but on a non-key field, an index can still be built which is called a clustering index. The difference lies in the fact that different records have different values in the key field, but for a non-key field, some records may share the same value. It must be noted that a file can have at most one physical ordering field. Thus, it can have at most one primary index or one clustering index, but not both. A third type of indexes, called secondary index, can be specified on any non-ordering field of a file. A file can have several secondary indexes in addition to its primary access path. Secondary indexes do not affect the physical organization of records.

7.9 Multi-level indexes

In a full indexing scheme, the address of every record is maintained in the index. For a small file, this index would be small and can be processed very efficiently in main memory.



For a large file, the index's size would pose problems. It is possible to create a hierarchy of indexes with the lowest level index pointing to the records, while the higher-level indexes point to the indexes below them (figure). The higher-level indices are small and can be moved to main memory, allowing the search to be localized to one of the larger lower level indices.

The lowest level index consists of the pair for each record in the file; this is costly in terms of space. Updates of records require changes to the index file as well as the data file. Insertion of a record requires that its pair be inserted in the index at the correct point, while deletion of a record requires that the pair be removed from the index. Therefore, maintenance of the index is also expensive. In the simplest case, updates of variable length records require that changes be made to the address field of the record entry. In a variation of this scheme, the address value in the lowest level index entry points to a block of records and the key value represents the highest key value of records in this block. Another variation of this scheme is described in the next section.

7.10 Other types of indexes

7.10.1 PRIMARY INDEXING

A primary index is built for a file (the data file) sorted on its key field and the index itself is another sorted file (called the index file) whose records (index records) are of fixed-length with two fields. The first field is of the same data type as the ordering key field of the data file, and the second field is a pointer to a disk block. The ordering key field is called the primary key of the data file. There is one index entry in the index file for each block in the data file. Each index entry has the value of the primary key field for the first record in a block and a pointer to that block as its two field values. We use the following notation to refer to an index entry i in the index file:

(K(i), P(i))

K(i) is the primary key value, and P(i) is the corresponding pointer (i.e., block address).

For example, to build a primary index on the sorted file shown below, we use the SID (Student ID) as primary key, because that is the ordering key field of the data file.

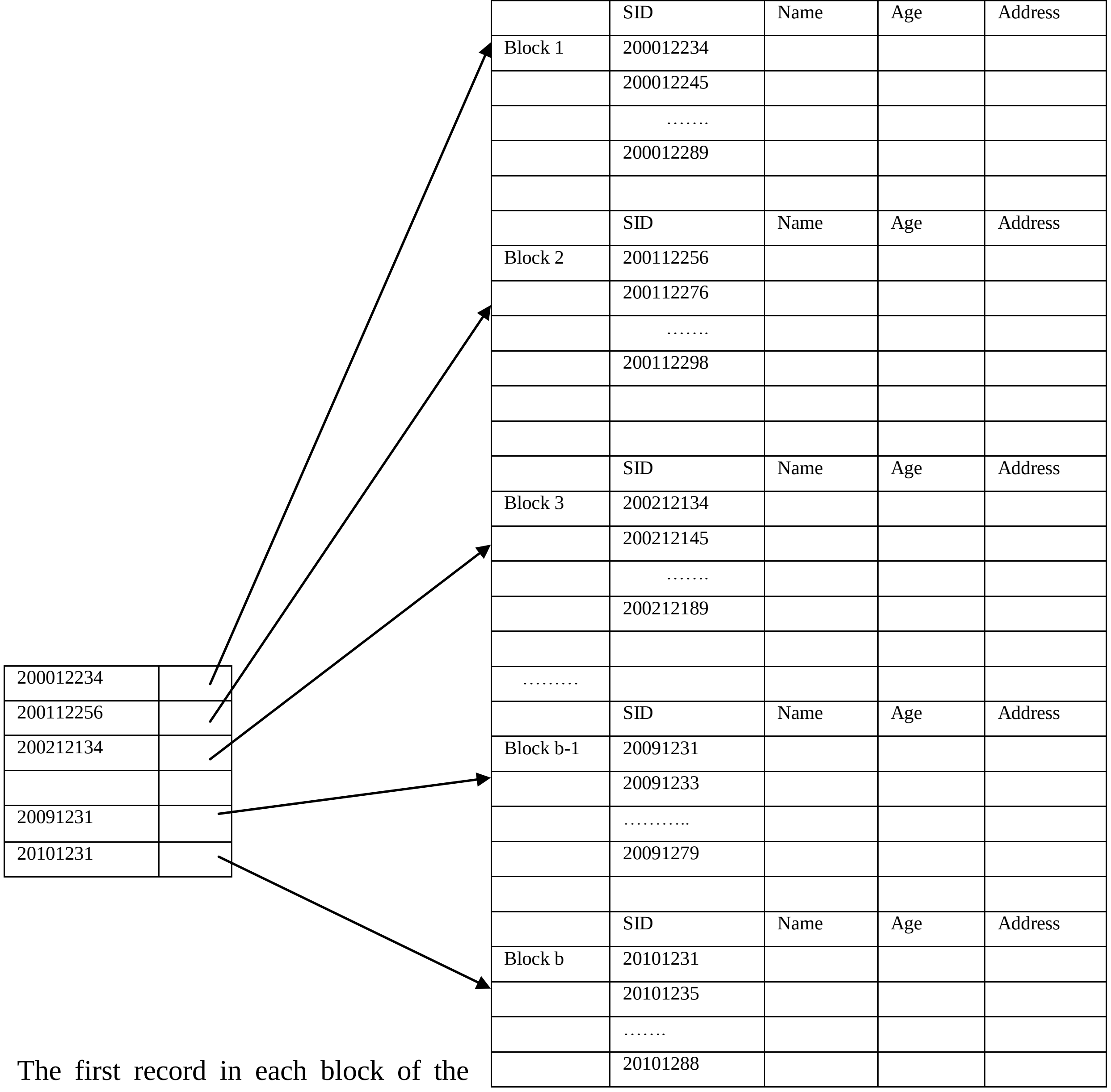
	SID	Name	Age	Address
Block 1	200012234			
	200012245			
	…….			
	200012289			

	SID	Name	Age	Address
Block 2	200112256			
	200112276			
	…….			
	200112298			
	SID	Name	Age	Address
Block 3	200212134			
	200212145			
	…….			
	200212189			
………				
	SID	Name	Age	Address
Block b-1	20091231			
	20091233			
	………..			
	20091279			
	SID	Name	Age	Address
Block b	20101231			
	20101235			
	…….			
	20101288			

Each entry in the index has an SID value and a pointer. The first two index entries of such an index file are as follows:

(K(1) = 200012234, P(1) = address of block 1)
(K(2) = 200112256, P(2) = address of block 2)
(K(3) = 200212134, P(3) = address of block 3)

The figure depicts this primary index. The total number of entries in the index is the same as the number of disk blocks in the data file. In this example, there are b blocks.



200012234	
200112256	
200212134	
20091231	
20101231	

	SID	Name	Age	Address
Block 1	200012234			
	200012245			
	…….			
	200012289			
	SID	Name	Age	Address
Block 2	200112256			
	200112276			
	…….			
	200112298			
	SID	Name	Age	Address
Block 3	200212134			
	200212145			
	…….			
	200212189			
………				
	SID	Name	Age	Address
Block b-1	20091231			
	20091233			
	………..			
	20091279			
	SID	Name	Age	Address
Block b	20101231			
	20101235			
	…….			
	20101288			

The first record in each block of the data file is called the anchor record of that block.

A primary index is an example of a sparse index in the sense that it contains an entry for each disk block rather than for every record in the data file. A dense index, on the other hand, contains an entry for every data record. In the case of a dense index, it does not require the data file to be a sorted file. Instead, it can be built on any file organization. By definition, an index file is just a special type of data file of fixed-length records with two fields.

Performance Issues:

The index file for a primary index needs significantly fewer blocks than does the file for data records for the following two reasons:

- There are fewer index entries than there are records in the data file, because an entry exists for each block rather than for each record.

- Each index entry is typically smaller in size than a data record because it has only two fields. Consequently more index entries can fit into one block. A binary search on the index file hence requires fewer block accesses than a binary search on the data file.

If a record whose primary key value is K is in the data file, then it has to be in the block whose address is P(i), where K(i) <= K < K(i+1). The i^{th} block in the data file contains all such records because of the physical ordering of the records based on the primary key field. To retrieve a record, given the value K of its primary key field, we do a binary search on the index file to find the appropriate index entry i, and then use the block address contained in the pointer P(i) to retrieve the data block.

A major problem with a primary index is insertion and deletion of records. If we attempt to insert a record in its correct position in the data file, we not only have to move records to make space for the newcomer but also have to change some index entries, since moving records may change some block anchors. Deletion of records causes similar problems in the other direction. Since a primary index file is much smaller than the data file, storage overhead is not a serious problem.

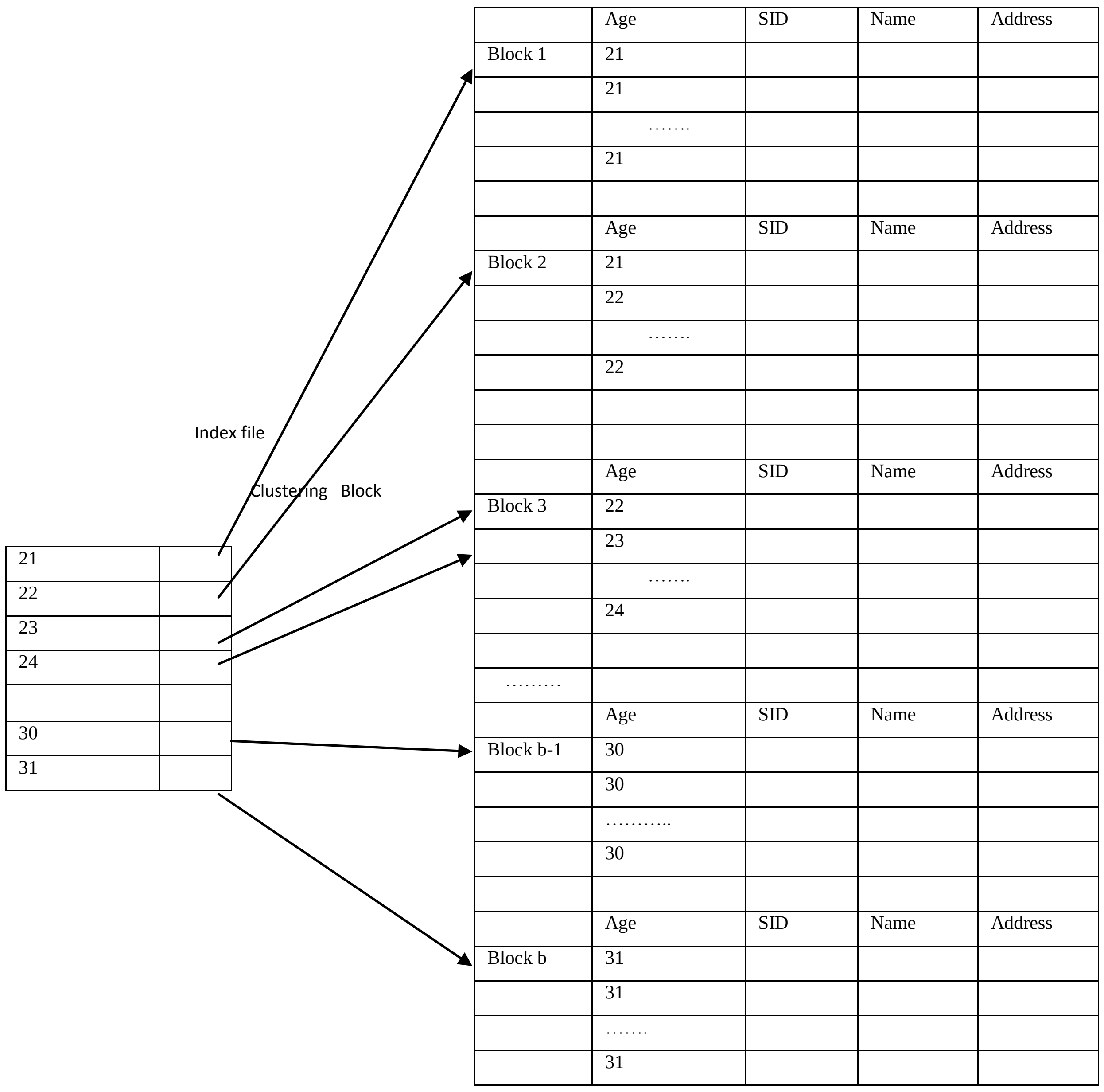
7.10.2 CLUSTERED INDEXES

If records of a file are physically ordered on a non-key field which may not have a unique value for each record, that field is called the clustering field. Based on the clustering field values, a clustering index can be built to speed up retrieval of records that have the same value for the clustering field.

A clustering index is also a sorted file of fixed-length records with two fields. The first field is of the same type as the clustering field of the data file, and the second field is a block pointer. There is one entry in the clustering index for each distinct value of the clustering field, containing the value and a pointer to the first block in the data file that holds at least one record with that value for its clustering field. The figure below illustrates an example of the Student file (sorted by their age rather than SID) with a clustering index.

In the below figure, there are 11 distinct values for Age: 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, and 31. Thus, there are 11 entries in the clustering index. As can be seen from the figure, many different records have the same Age value and can be stored in different blocks. Both Age 23 and Age 24 entries point to the 3rd block, because it stores the first record for Age 23 as well as Age 24 students.

A clustering index is another type of sparse index, because it has an entry for each distinct value of the clustering field rather than for every record in the file.



Performance Issues:

Performance improvements can be obtained by using the index to locate a record. However, the record insertion and deletion still causes similar problems to those in primary indexes, because the data records are physically ordered.

To alleviate the problem of insertion, it is common to reserve a whole block for each distinct value of the clustering field; all records with that value are placed in the block. If more than one block is needed to store the records for a particular value, additional blocks are allocated and linked together. To link blocks, the last position in a block is reserved to hold a block pointer to the next block. If there is no following block, then the block pointer will have the null value. Using this linked blocks structure, no records with different clustering field values can be stored in the same block. It also makes insertion and deletion more efficient than without the linked structure. More blocks will be needed to store records and some spaces may be wasted. That is the price to pay for improving insertion efficiency.

7.11 B-Tree index files

The B-tree is known as the balanced sort tree, which is useful for external sorting. There are strong uses of B-trees in a database system as pointed out by D.Comer (1979): "While no single scheme can be optimum for all applications, the technique of organizing a file and its index called the B -tree is, The facto, the standard organization for indexes in a database system."

The file is a collection of records. The index refers to a unique key, associated with each record. One application of B- trees is found in IBM's Virtual Storage Access Method (VSAM) file organization. Many data manipulation tasks require data storage only in main memory. For applications with a large database running on a system with limited company, the data must be stored as records on secondary memory (disks) and be accessed in pieces. The size of a record can be quite large, as shown below:

```
struct DATA
{
int ssn;
char name [80];
char address [80];
char schoold [76];
struct DATA * left; /* main memory addresses */
struct DATA * right; /* main memory addresses */
d_block d_left; /* disk block address */
d_block d_right; /* disk block address */
}
```

Advantages of B-tree indexes:

1. Because there is no overflow problem inherent with this type of organization it is good for dynamic table - those that suffer a great deal of insert / update / delete activity.
2. Because it is to a large extent self-maintaining, it is good in supporting 24-hour operation.
3. As data is retrieved via the index it is always presented in order.
4. 'Get next' queries are efficient because of the inherent ordering of rows within the index blocks.
5. B-tree indexes are good for very large tables because they will need minimal reorganization.
6. There is predictable access time for any retrieval (of the same number of rows of course) because the B-tree structure keeps itself balanced, so that there is always the same number of index levels for every retrieval. Bear in mind of course, that the number of index levels does increase both with the number of records and the length of the key value.

Because the rows are in order, this type of index can service range type enquiries, of the type below, efficiently.

SELECT ... WHERE COL BETWEEN X AND Y.

Disadvantages of B-tree indexes:

1. For static tables, there are better organizations that require fewer I/0s. ISAM indexes are preferable to B-tree in this type of environment.
2. B-tree is not really appropriate for very small tables because index look-up becomes a significant part of the overall access time.
3. The index can use considerable disk space, especially in products, which allow different users to create separate indexes on the same table/column combinations.
4. Because the indexes themselves are subject to modification when rows are updated, deleted or inserted, they are also subject to locking which can inhibit concurrency.

So to conclude this section on B-tree indexes, it is worth stressing that this structure is by far and away the most popular, and perhaps versatile, of index structures supported in the world of the RDBMS today. Whilst not fully optimized for certain activity, it is seen as the best single compromise in satisfying all the different access methods likely to be required in normal day-to-day operation.

7.12 B^+- Tree index files

In a B+ tree the leaves are linked together to form a sequence set; interior nodes exist only for the purposes of indexing the sequence set (not to index into data/records). The insertion and deletion algorithm differ slightly. Sequential access to the keys of a B-tree is much slower than sequential access to the keys of a B+ tree, since the latter are linked in sequential order by definition.

B+ tree search structure is a balanced tree in which the internal nodes direct the search and the leaf nodes contain the data entries. Leaf pages are linked using page pointers since they are not allocated sequentially. Sequence of leaf pages is called sequence set. It requires a minimum occupancy of 50% at each node except the root. If every node contains m entries and the order of the tree (a given parameter of tree) is d, the relationship d □ m □2d is true for every node except the root where it is 1 □ m .Non-leaf nodes with m index entries contain m+1 pointers to children. Leaf nodes contain data entries.

Indexes (B+ trees)

13 17 24 30

Insertion of 8 into the tree leads to a split of leftmost leaf node as well as

2* 3* 5* 7* 14* 16* 19* 20* 22*

24* 27* 29* 33* 34* 38* 39*

A b+ tree of height 1, order d=2

Insertion of 8 into the tree leads to a split of left most leaf node as well as the split of the index page to increase the height of the tree. Deletion of a record may cause a node to be at minimum occupancy and entries from an adjacent sibling are then redistributed or two nodes may need to be merged.

7.13 Bitmap index

A bitmap index is a special kind of index that stores the bulk of its data as bit arrays (bitmaps) and answers most queries by performing bitwise logical operations on these bitmaps. The most commonly used index, such as B+trees, are most efficient if the values it indexes do not repeat or repeat a smaller number of times. In contrast, the bitmap index is designed for cases where the values of a variable repeat very frequently. For example, the gender field in a customer database usually contains two distinct values: male or female. For such variables, the bitmap index can have a significant performance advantage over the commonly used trees

7.14 Hash Index

The basic idea is to use a hashing function, which maps a search-key value (of a field) into a record or bucket of records.

There are 3 alternatives for data entries:

- Actual data record (with key value k)
- <k, rid of matching data record>
- <k, list of rids of matching data records>

The Hash-based indexes are best for equality selections. They cannot support range searches. The hashing techniques can classified as static and dynamic.

Hash File Organization:

The hash file organization is based on the use of hashing techniques, which can provide very efficient access to records based on certain search conditions. The search condition must be an equality condition on a single field called hash field (e.g., ID# = 9701890, where ID# is the hash field). Often the hash field is also a key field. In this case, it is called the hash key.

Hashing Techniques:

The principle idea behind the hashing techniques is to provide a function h, called a hash function, which is applied to the hash field value of a record and computes the address of the disk block in which the record is stored. A search for the record within the block can be carried out in a buffer, as always. For most records, we need only one block transfer to retrieve that record.

Suppose K is a hash key value, the hash function h will map this value to a block address in the following form:

h(K) = address of the block containing the record with the key value K

If a hash function operates on numeric values, then non-numeric values of the hash field will be transformed into numeric ones before the function is applied.

The following are two examples of many possible hash functions:

- Hash function h(K) = K mod M: this function returns the remainder of an integer hash field value K after division by integer M. The result of the calculation is then used as the address of the block holding the record.
- A different function may involve picking some digits from the hash field value (e.g., the 2nd, 4th, and 6th digits from ID#) to form an integer, and then further calculations may be performed using the integer to generate the hash address.

The problem with most hashing functions is that they do not guarantee that distinct values will hash to distinct addresses. The reason is that the hash field space (the number of possible values a hash field can take) is usually much larger than the address space (the number of available addresses for records). For example, the hash function h(K) = K mod 7 will hash 15 and 43 to the same address 1 as shown below:

43 mod 7 = 1

15 mod 7 = 1

A collision occurs when the hash field value of a new record that is being inserted hashes to an address that already contains a different record. In this situation, we must insert the new record in some other position. The process of finding another position is called collision resolution.

The following are commonly used techniques for collision resolution:

- Open addressing: If the required position is found occupied, the program will check the subsequent positions in turn until an available space is located.
- Chaining: For this method, some spaces are kept in the disk file as overflow locations. In addition, a pointer field is added to each record location. A collision is resolved by placing the new record in an unused overflow location and setting the pointer of the occupied hash address location to the address of that overflow location. A linked list of overflow records for each hash address needs to be maintained.
- Multiple hashing: If a hash function causes a collision, a second hash function is applied. If it still does not resolve the collision, we can then use open addressing or apply a third hash function and so on.

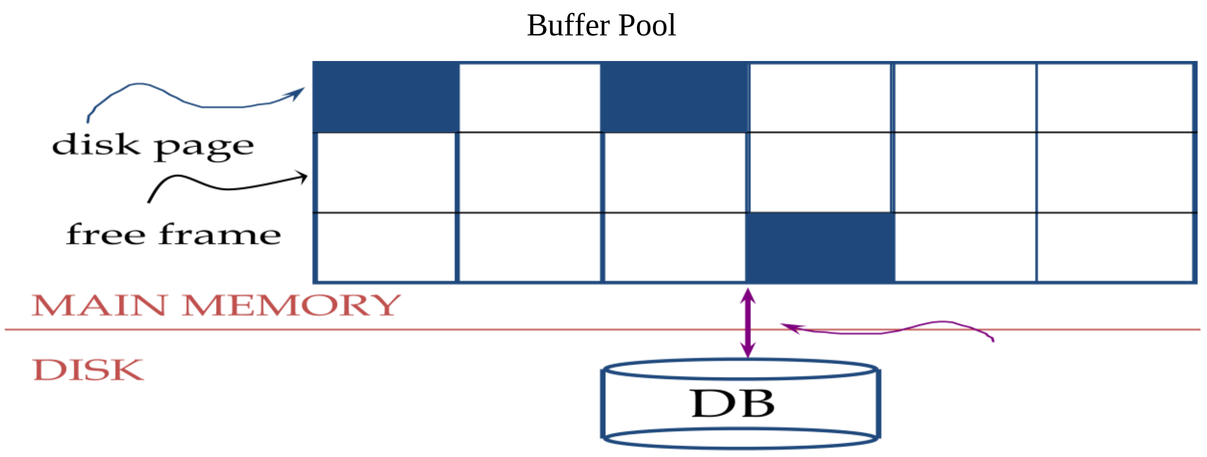
Each collision resolution method requires its own algorithms for insertion, retrieval, and deletion of records. Detailed discussions on them are beyond the scope of this unit. Interested students are advised to refer to textbooks dealing with data structures. In general, the goal of a good hash function is to distribute the records uniformly over the address space and minimise collisions while not leaving many unused locations. Also remember that a hash function should not involve complicated computing. Otherwise, it may take a long time to produce a hash address which will actually hinder performance.

7.15 Buffer management

A buffer is an 8-KB page in memory, the same size as a data or index page. Thus, the buffer cache is divided into 8-KB pages. The buffer manager manages the functions for reading data or index pages from the database disk files into the buffer cache and writing modified pages back to disk. A page remains in the buffer cache until the buffer manager needs the buffer area to read in more data. Data is written back to disk only if it is modified. Data in the buffer cache can be modified multiple times before being written back to disk

Buffer Management in a DBMS

Page Requests from Higher Levels



- Data must be in RAM for DBMS to operate on it,Buffer Mgr hides the fact that not all data is in RAM.
- When a Page is Requested, Buffer pool information table contains: <***frame#, pageid, pin_count, dirty***>
- If requested page is not in pool, Choose a frame for replacement. Only “un-pinned” pages are candidates. If frame is dirty write it to disk.
- Read requested page into chosen frame, Pin the page and return its address**.**
- Requestor of page must eventually unpin it, and indicate whether page has been modified, dirty bit is used for this. Page in pool may be requested many times, a pin count is used. To pin a page, pin_ count++, a page is a candidate for replacement iff pin count == 0 (“unpinned”).CC & recovery may entail additional I/O when a frame is chosen for replacement.

Buffer Replacement Policy:

- Frame is chosen for replacement by a replacement policy:

- Least-recently-used (LRU)
- Most-recently-used (MRU).Also called toss-immediate
  Clock.

- Policy can have big impact on # of I/O’s; depends on the access pattern.

7.16 SUMMARY

The DBMS stores data on external storage because the quantity of data is vast, and must persist across program executions. These external storages are called secondary storage.
In this unit we discussed about storage hierarchy. We also studied file organizations and indexing techniques.
Indexes are additional auxiliary access structures on already organized files (with some primary organization). Indexes speed up the retrieval of records under certain search conditions. This unit covered Primary indexing, Clustered indexes, and secondary indexing.
A database index is a data structure that improves the speed of data retrieval operations on a database table at the cost of slower writes and increased storage space. In this unit we studied different types of index data structures. We also discussed in depth hash-based indexing. There are many basic data structures that can be used to solve application problems. But to overcome their limitations, we introduced the tree data structure. We discussed B-tree , B^+ trees, and B^* trees , which are special types of the well-known tree data structure.

7.17 KEYWORDS

Bucket: A bucket is a unit of storage that can store one or more records
Heap File: A heap file is an unordered set of records.
Hash File Organization: The hash file organization is based on the use of hashing techniques.
Database index: A database index is a data structure that improves the speed of data retrieval operations on a database table at the cost of slower writes and increased storage space
Unique index: A unique index acts as a constraint on the table by preventing duplicate entries in the index and thus the backing table
Non-unique index: Use of a non unique index results in scanning all leaf blocks of the relevant part of the b-tree, when it finds a row that matches the query it will continue to scan all further leaf blocks to see if there are any other matching rows.

Bitmap index: A bitmap index is a special kind of index that stores the bulk of its data as bit arrays (bitmaps) and answers most queries by performing bitwise logical operations on these bitmaps.

B-tree: Balanced m-way search tree.

B^+ trees: A variation of B-tree with some additional properties.

B^* trees: A variation of B-trees.

7.18 UNIT-END EXERCISES AND ANSWERS

1. Explain the storage hierarchies.
2. Explain about over all file organization technique.
3. Explain hash file organization.
4. What is indexing? What are its advantages and disadvantages of indexed files?
5. Define bitmap index.
6. Explain Indexing and Hashing- Basic concepts?
7. What are the most notable problems that static hashing techniques have?
8. What are the most notable problems that dynamic hashing techniques have?
9. What are a primary index, a clustering index, and a secondary index?
10. Explain about single level ordered index?
11. Explain about multilevel ordered index?
12. Describe other index.
13. Describe the general structure of a B-tree node.
14. Describe the structure of both internal and leaf nodes of a B^+ tree.
15. Why is a B^+ tree usually preferred as an access structure to a data file rather than a B-tree?
16. What major problems may be caused by update operations on a B-tree/B^+ tree structure and why?
17. How buffer management works?
18. Describe about hash index.

Answer: SEE

1. 7.3
2. 7.2
3. 7.3,

4.	7.4
5.	7.13
6.	7.5
7.	7.6
8.	7.7
9.	7.10.1, 7.10.2
10.	7.8
11.	7.9
12.	7.10,
13.	7.11
14.	7.12
15.	7.12
16.	7.11,7.12
17.	7.15
18.	7.14

7.19 SUGGESTED READINGS

[1] Fundamentals of Database Systems By Ramez Elmasri, Shamkant B. Navathe, Durvasula V.L.N. Somayajulu, Shyam K.Gupta

[2] Database System Concepts By Avi Silberschatz, Henry F. Korth , S. Sudarshan

[3] Database Management Systems By Raghu Ramakrishnan and Johannes Gehrke

[4] An Introduction to Database Systems C.J.Date

[5] Database Management Systems Alexis Leon, Mathews, Vikas Publishing House

UNIT 8: Transaction Processing

Structure:

8.0 OBJECTIVES

At the end of this unit you will be able to know:

- Atomicity
- Durability
- Concept of a transaction
- Schedules
- Describe the nature of transactions and the reasons for designing database systems around transactions.
- Explain the causes of transaction failure.
- Analyze the problems of data management in a concurrent environment.
- Critically compare the relative strengths of different concurrency control approaches

8.1 INTRODUCTION

In this unit, we will discuss the concurrency control problem, which occurs when multiple transactions submitted by various users interfere with one another in a way that produces incorrect results. We will start the unit by introducing some basic concepts of transaction processing. Why concurrency control and recovery are necessary in a database system is then discussed. The concept of an atomic transaction and additional concepts related to transaction processing in database systems are introduced. The concepts of atomicity, consistency, isolation, and durability – the so-called ACID properties that are considered desirable in transactions are presented.

The concept of schedules of executing transactions and characterizing the recoverability of schedules is introduced with a detailed discussion of the concept of serializability of concurrent transaction executions, which can be used to define correct execution sequences of concurrent transactions.

The purpose of this unit is to introduce the fundamental technique of concurrency control, which provides database systems the ability to handle many users accessing to data simultaneously. The unit also describes the problems that arise out of the fact that users wish to query and update stored data at the same time, and the approaches developed to address these problems together with their respective strengths and weaknesses in a range of practical situations

8.2 Desirable properties of transactions

A transaction is the execution of a program that accesses or changes the contents of a database. It is a logical unit of work (LUW) on the database that is either completed in its entirety (COMMIT) or not done at all. In the later case, the transaction has to clean up its own mess, known as ROLLBACK. A transaction could be an entire program, a portion of program or a single command.

The concept of a transaction is inherently about organizing functions to manage data. A transaction may be distributed (available on different physical systems or organized into different logical sub-systems) and/or use data concurrently with multiple users for different purposes.

An example of a transaction which transfers fund from account A to account B:

Transaction to transfer Rs 500 from account A to account B:

1. read(A)
2. A := A – 50
3. write(A)
4. read(B)
5. B := B + 50
6. write(B)

Jim Gray defined these properties of a reliable transaction system in the late 1970s and developed technologies to automatically achieve them. In 1983, Andreas Reuter and Theo Haerder coined the acronym ACID to describe them.

Atomicity: The atomicity property identifies that the transaction is atomic. An atomic transaction is either fully completed, or is not begun at all. Atomicity requires that database modifications must follow an "all or nothing" rule. Any updates that a transaction might affect on a system are completed in their entirety. If for any reason an error occurs and the transaction is unable to complete all of its steps, the then system is returned to the state it was in before the transaction was started. An example of an atomic transaction is an account transfer transaction. The money is removed from account A then placed into account B. If the system fails after removing the money from account A, then the transaction processing system will put the money back into account A, thus returning the system to its original state.

Transactions can fail for several kinds of reasons:

- Hardware failure: A disk drive fails, preventing some of the transaction's database changes from taking effect.
- System failure: The user loses their connection to the application before providing all necessary information.
- Database failure: E.g., the database runs out of room to hold additional data.
- Application failure: The application attempts to post data that violates a rule that the database itself enforces such as attempting to insert a duplicate value in a column.

In the above fund transfer example:

Atomicity requirement — if the transaction fails after step 3 and before step 6, the system should ensure that its updates are not reflected in the database, else an inconsistency will result.

Consistency: It ensures the truthfulness of the database. The consistency property ensures that any transaction the database performs will take it from one consistent state to another. A transaction enforces consistency in the system state by ensuring that at the end of any transaction the system is in a valid state. If the transaction completes successfully, then all changes to the system will have been properly made, and the system will be in a valid state. If any error occurs in a transaction, then any changes already made will be automatically rolled back. This will return the system to its state before the transaction was started. Since the system was in a consistent state when the transaction was started, it will once again be in a consistent state.

Looking again at the account transfer system, the system is consistent if the total of all accounts is constant. If an error occurs and the money is removed from account A and not added to account B, then the total in all accounts would have changed. The system would no longer be consistent. By rolling back the removal from account A, the total will again be what it should be, and the system back in a consistent state.

In the above fund transfer example:

Consistency requirement – the sum of A and B is unchanged by the execution of the transaction.

Isolation: When a transaction runs in isolation, it appears to be the only action that the system is carrying out at one time. If there are two transactions that are both performing the same function and are running at the same time, transaction isolation will ensure that each transaction thinks it has exclusive use of the system. This is important in that as the transaction is being executed, the state of the system may not be consistent. The transaction ensures that the system remains consistent after the transaction ends, but during an individual transaction, this may not be the case. If a transaction was not running in isolation, it could access data from the system that may not be consistent. By providing transaction isolation, this is prevented from happening.

That is, for every pair of transactions Ti and Tj, it appears to Ti that either Tj, finished execution before Ti started, or Tj started execution after Ti finished.

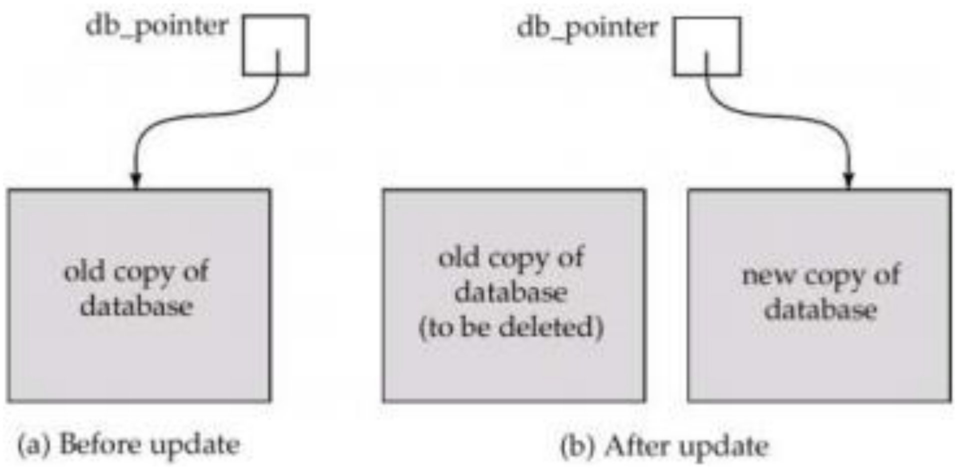
In the above fund transfer example:

Isolation requirement — if between steps 3 and 6, another transaction is allowed to access the partially updated database, it will see an inconsistent database (the sum A + B will be less than it should be). This should be ensured by isolating the effects of individual transaction.

Durability: Durability is the ability of the DBMS to recover the committed transaction updates against any kind of system failure (hardware or software). Durability is the DBMS's guarantee that once the user has been notified of a transaction's success the transaction will not be lost, the transaction's data changes will survive system failure A transaction is durable in that once it has been successfully completed all of the changes it made to the system are permanent. There are safeguards that will prevent the loss of information, even in the case of system failure. By logging the steps that the transaction performs, the state of the system can be recreated even if the hardware itself has failed. The concept of durability allows the developer to know that a completed transaction is a permanent part of the system, regardless of what happens to the system later on.
In the above fund transfer example:
Durability requirement — once the user has been notified that the transaction has completed (i.e., the transfer of the Rs 500 has taken place), the updates to the database by the transaction must persist despite failures.

8.3 Implementation of Atomicity and durability

The recovery-management component of a database system can support atomicity
and durability by a variety of schemes. We first consider a simple, but extremely inefficient, scheme called the shadow copy scheme. This scheme, which is based on making copies of the database, called shadow copies, assumes that only one transaction is active at a time. The scheme also assumes that the database is simply a file on disk. A pointer called db-pointer is maintained on disk; it points to the current copy of the database.
In the shadow-copy scheme, a transaction that wants to update the database first creates a complete copy of the database. All updates are done on the new database copy, leaving the original copy, the shadow copy, untouched. If at any point the transaction has to be aborted, the system merely deletes the new copy. The old copy of the database has not been affected.
If the transaction completes, it is committed as follows. First, the operating system is asked to make sure that all pages of the new copy of the database have been written out to disk. (Unix systems use the flush command for this purpose.) After the operating system has written all the pages to disk, the database system updates the pointer db-pointer to point to the new copy of the database; the new copy then becomes the current copy of the database. The old copy of the database is then deleted.



8.4 Concurrent executions

A transaction goes through well-defined stages in its life (always terminating). These stages are:

- Inactive
- Active, the initial state; the transaction stays in this state while it is executing (may read and write)
- Partially committed, after the final statement has been executed. (no errors during execution)
- Failed, after the discovery that normal execution can no longer proceed (errors)
- Aborted, after the transaction has been rolled back and the database restored to its state prior to the start of the transaction. Two options after it has been aborted:
    - restart the transaction – only if no internal logical error
    - kill the transaction
- Committed, after successful completion.

These states are shown in the Figure below

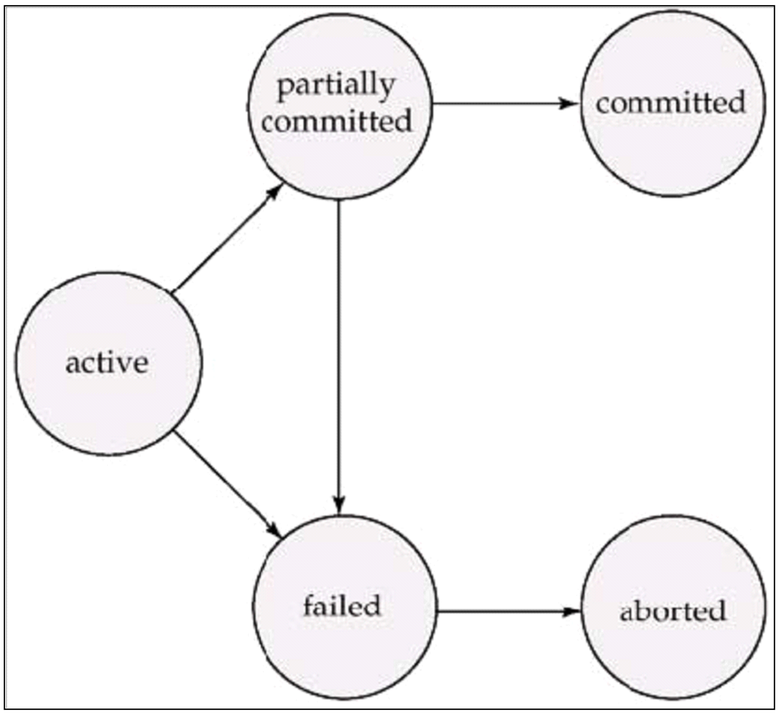


Figure: Transaction State Diagram

Concurrent Executions: Multiple transactions are allowed to run concurrently in the system. Advantages are:

- Increased processor and disk utilization, leading to better transaction throughput: one transaction can be using the CPU while another is reading from or writing to the disk
- Reduced average response time for transactions: short transactions need not wait behind long ones.

Concurrency control schemes – mechanisms to achieve isolation, i.e., to control the interaction among the concurrent transactions in order to prevent them from destroying the consistency of the database

8.5 Schedules and recoverability

A schedule is a sequence that indicates the chronological order in which instructions of concurrent transactions are executed.

- A schedule for a set of transactions must consist of all instructions of those transactions
- Must preserve the order in which the instructions appear in each individual transaction.

Example Schedules:

Let T1 transfer Rs 500 from A to B, and T2 transfer 20% of the balance from A to B. The following is a serial schedule (Schedule 1), in which T1 is followed by T2.

T1	T2
read(A);	
A := A – 500;	
write(A);	
read(B);	
B := B + 500;	
write(B)	
	read(A);
	temp := A * 0.2 ;
	A := A – temp ;
	write(A);
	read(B);
	B := B + temp;
	write(B);

Schedule 1: A serial schedule in which T1 is followed by T2.

The following is also a serial schedule (Schedule2), in which T2 is followed by T1.

T1	T2
	read(A);
	temp := A * 0.2 ;
	A := A – temp ;
	write(A);
	read(B);
	B := B + temp;
	write(B);
read(A);	
A := A – 500;	
write(A);	
read(B);	

B := B + 500;
write(B)	

Schedule 2: A serial schedule in which T2 is followed by T1.

Types of schedule

There are various types of schedules. From the scheduling point of view, we ignore operations other than read and write instructions, and we assume that transactions may perform arbitrary computations on data in local buffers in between reads and writes. Our simplified schedules consist of only read (represented by R) and write (represented by W) instructions.

Instructions li and lj of transactions Ti and Tj respectively, conflict if and only if there exists some item Q accessed by both li and lj, and at least one of these instructions wrote Q.

1. li = read(Q), lj = read(Q). li and lj don't conflict.
2. li = read(Q), lj = write(Q). They conflict.
3. li = write(Q), lj = read(Q). They conflict
4. li = write(Q), lj = write(Q). They conflict

Intuitively, a conflict between li and lj forces a (logical) temporal order between them. If li and lj are consecutive in a schedule and they do not conflict, their results would remain the same even if they had been interchanged in the schedule.

Serial Schedule:

A schedule is said serial if the transactions are executed in a non-interleaved fashion (see example Schedules 1 and 2, i.e., a serial schedule is one in which no transaction starts before a running transaction ends).

8.6 Serializability of schedules

Serializable:

A schedule that is equivalent (in its outcome) to a serial schedule has the serializability property.

Conflicting actions:

Two or more actions are said to be in conflict if:

1. The actions belong to different transactions.
2. At least one of the actions is a write operation.
3. The actions access the same object (read or write).

The following set of actions is conflicting (assume that T1, T2, and T3 are transactions and X is a data object):

- T1:R(X), T2:W(X), T3:W(X)

While the following sets of actions are not (non-conflicting):

- T1:R(X), T2:R(X), T3:R(X)
- T1:R(X), T2:W(Y), T3:R(X)

Conflict equivalence:

The schedules S1 and S2 are said to be conflict-equivalent if the following conditions are satisfied:

1. Both schedules S1 and S2 involve the same set of transactions (including ordering of actions within each transaction).
2. The order of each pair of conflicting actions in S1 and S2 are the same.

Conflict-serializable

A schedule is said to be conflict-serializable when the schedule is conflict-equivalent to one or more serial schedules.

For example:

T1	T2
R(A)	
	R(A)
W(B)	
	W(B)

Schedule 4: Which is conflict-equivalent to the serial schedule <T1,T2>, but not <T2,T1>.

View equivalence:

Two schedules S1 and S2 are said to be view-equivalent when the following conditions are satisfied:

1. If the transaction Ti in S1 reads an initial value for object X, so does the transaction Ti in S2.
2. If the transaction Ti in S1 reads the value written by transaction Tj in S1 for object X, so does the transaction Ti in S2.
3. If the transaction Ti in S1 is the final transaction to write the value for an object X, so is the transaction Ti in S2.

View - Serializable

A schedule is said to be view-serializable if it is view-equivalent to some serial schedule. Note that by definition, all conflict-serializable schedules are view-serializable.

T1	T2
R(A)	
	R(A)
W(B)	

Schedule 5: A view serializable schedule (which is also conflict-serializable)

Notice that the Schedule 5 is both view-serializable and conflict-serializable at the same time. There are however view-serializable schedules that are not conflict-serializable: those schedules with a transaction performing a blind write:

T1	T2	T3
R(A)		
	W(A)	
W(A)		
		W(A)

Schedule 6: a view serializable schedule

Notice that the Schedule 6 is view-serializable and but not conflict-serializable, but it is view-serializable since it has a view-equivalent serial schedule <T1, T2, T3>.

Since determining whether a schedule is view-serializable is NP-complete, view-serializability has little practical interest.

Complete Schedule: A complete schedule is one that contains an abort or commit actions for every transaction that occurs in the schedule. For example Schedule 7 is a complete schedule.

Schedules Involving Aborted Transactions:

When you abort a transaction, all database modifications performed under the protection of the transaction are discarded.

Unrecoverable schedule:

Consider the Schedule 7, aborted transactions being undone completely – we have to do cascading abort. We have to abort changes made by T2, but T2 is already committed – we say this schedule is an Unrecoverable schedule. What we need is Recoverable schedule.

T1	T2
R(A)	
W(A)	
	R(A)
	W(A)
	R(B)
	W(B)
Abort	Commit

Schedule 7: An Unrecoverable schedule

Recoverable Schedules:

In recoverable schedule, transactions commit only after all transactions whose changes they read commit. In such a case, we can do cascading abort.

For example consider the Schedule 8. Note that T2 cannot commit before T1, therefore when T1 aborts, we can abort T2 as well.

T1	T2
R(A)	
W(A)	
	R(A)
	W(A)
	R(B)
	W(B)
Abort	

Schedule 8: A Recoverable schedule

Cascadeless Aborts: (Avoids cascading aborts -rollbacks)

A single transaction abort leads to a series of transaction rollback. Strategy to prevent cascading aborts is to disallow a transaction from reading uncommitted changes from another transaction in the same schedule.

For example: Schedule 9 is Recoverable, Schedule 10 is Recoverable schedule with cascading aborts, Schedule 11 is a Recoverable schedule with cascadeless aborts.

T1	T2
R(A)	
W(A)	
	R(A)
	W(A)
Commit	
	Commit

Schedule 9: A Recoverable schedule

T1	T2
R(A)	
W(A)	
	R(A)
	W(A)

Abort	
	Abort

Schedule 10: A Recoverable schedule with cascading aborts.

T1	T2
	R(A)
R(A)	
W(A)	
	W(A)
Abort	
	Commit

Schedule 11: A Recoverable schedule with cascade less aborts.

8.7 concurrency control

There are a variety of mechanisms to implement concurrency control. They can be categorized as:

- **Optimistic** - A concurrency control scheme is considered optimistic when locks are acquired and released over a very short period of time at the end of a transaction. Delay the checking of whether a transaction meets the isolation and other integrity rules (e.g., serializability and recoverability) until its end, without blocking any of its (read, write) operations ("...and be optimistic about the rules being met..."), and then abort a transaction to prevent the violation, if the desired rules are to be violated upon its commit. An aborted transaction is immediately restarted and re-executed, which incurs an obvious overhead (versus executing it to the end only once). If not too many transactions are aborted, then being optimistic is usually a good strategy.
- **Pessimistic** - A concurrency control scheme is considered pessimistic when it locks a given resource early in the data-access transaction and does not release it until the transaction is closed. Block an operation of a transaction, if it may cause violation of the rules, until the possibility of violation disappears. Blocking operations is typically involved with performance reduction.

- **Semi-optimistic** - Block operations in some situations, if they may cause violation of some rules, and do not block in other situations while delaying rules checking (if needed) to transaction's end, as done with optimistic.

8.8 Serializability algorithms

SSI Algorithm

Serializable transaction are implemented using Serializable Snapshot Isolation (SSI), based on the work of Cahill, et al. Fundamentally, this allows snapshot isolation to run as it has, while monitoring for conditions which could create a serialization anomaly.

SSI is based on the observation that each snapshot isolation anomaly corresponds to a cycle that contains a "dangerous structure" of two adjacent rw-conflict edges:

T_{in} ----$_{rw}$---> T_{pivot} ----$_{rw}$---> T_{out}

SSI works by watching for this dangerous structure, and rolling back a transaction when needed to prevent any anomaly. This means it only needs to track rw-conflicts between concurrent transactions, not wr- and ww-dependencies. It also means there is a risk of false positives, because not every dangerous structure corresponds to an actual serialization failure.

The PostgreSQL implementation uses two additional optimizations:

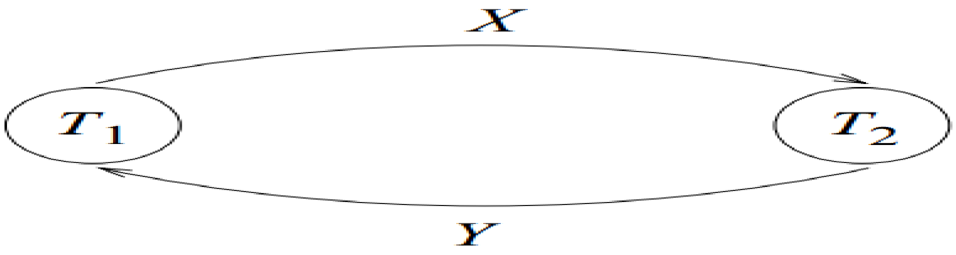
1. T_{out} must commit before any other transaction in the cycle. We only roll back a transaction if T_{out} commits before T_{pivot} and T_{in}.
2. If T_{in} is read-only, there can only be an anomaly if T_{out} committed before T_{in} takes its snapshot. This optimization is an original one. Proof:

   - Because there is a cycle, there must be some transaction T_0 that precedes T_{in} in the serial order. (T_0 might be the same as T_{out}).
   - The dependency between T_0 and T_{in} can't be a rw-conflict, because T_{in} was read-only, so it must be a ww- or wr-dependency. Those can only occur if T_0 committed before T_{in} started.
   - Because T_{out} must commit before any other transaction in the cycle, it must commit before T_0 commits -- and thus before T_{in} starts.

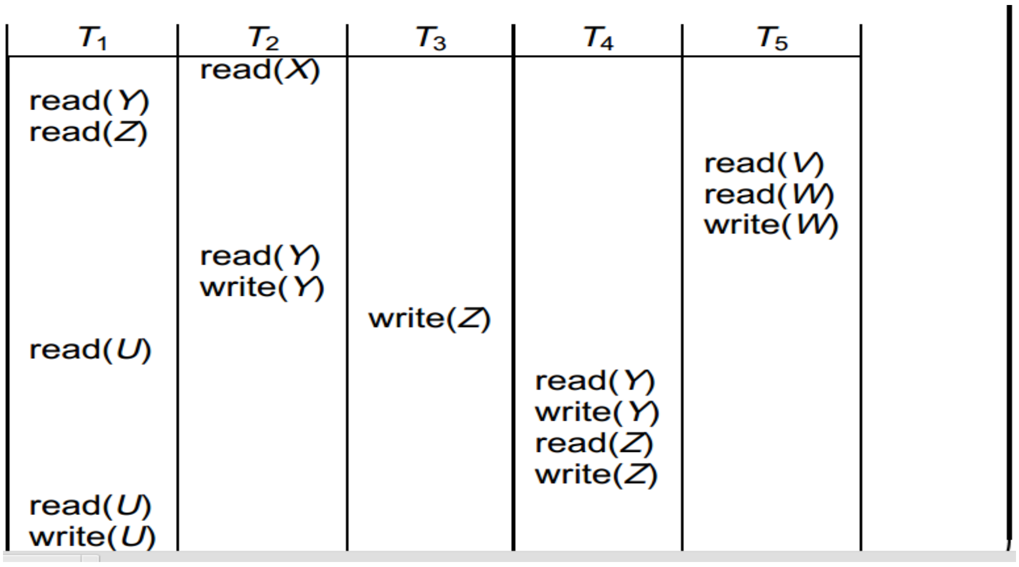
8.9 Testing for Serializability

Consider some schedule of a set of transactions T1, T2, ... , Tn. Precedence graph — a directed graph where the vertices are the transactions (names). We draw an arc from Ti to Tj if the two transactions conflict and Ti accessed the data item on which the conflict arose earlier. We may label the arc by the item that was accessed.

Example 1

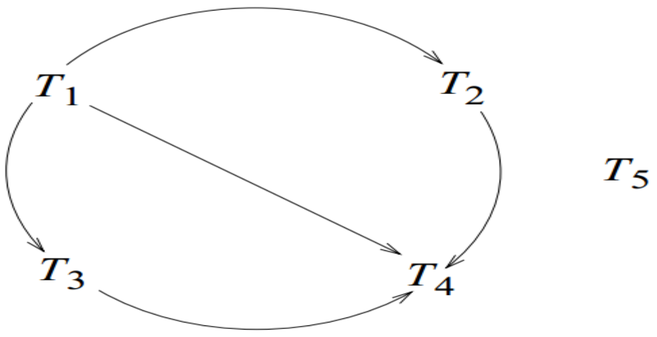


Example of schedule (schedule A)



T_1	T_2	T_3	T_4	T_5
	read(X)			
read(Y)				
read(Z)				
				read(V)
				read(W)
				write(W)
	read(Y)			
	write(Y)			
		write(Z)		
read(U)				
			read(Y)	
			write(Y)	
			read(Z)	
			write(Z)	
read(U)				
write(U)				

Precedence Graph for Schedule A



Test for Conflict Serializability:

- A schedule is conflict serializable if and only if its precedence graph is acyclic.
- Cycle-detection algorithms exist which take order n^2 time, where n is the number of vertices in the graph. (Better algorithms take order n + e where e is the number of

edges.) If precedence graph is acyclic, the serializability order can be obtained by a topological sorting of the graph. This is a linear order consistent with the partial order of the graph.

- For example, a serializability order for Schedule A would be
  $T5 \rightarrow T1 \rightarrow T3 \rightarrow T2 \rightarrow T4$.

Test for View Serializability:

- The precedence graph test for conflict serializability must be modified to apply to a test for view serializability.
- Construct a labelled precedence graph. Look for an acyclic graph which is derived from the labelled precedence graph by choosing one edge from every pair of edges with the same non-zero label. Schedule is view serializable if and only if such an acyclic graph can be found.
- The problem of looking for such an acyclic graph falls in the class of NP-complete problems. Thus existence of an efficient algorithm is unlikely. However practical algorithms that just check some sufficient conditions for view serializability can still be used.

8.10 SUMMARY

Many criteria can be used to classify DBMS. One of which, is the number of users supported by the system the concept of an atomic transaction and additional concepts related to transaction processing in database systems are introduced. The concepts of atomicity, consistency, isolation, and durability – the so-called ACID properties that are considered desirable in transactions are presented. The concept of schedules of executing transactions and characterizing the recoverability of schedules is introduced with a detailed discussion of the concept of serializability of concurrent transaction executions.

In this unit, we introduced the fundamental technique of concurrency control, which provides database systems the ability to handle many users accessing to data simultaneously. The unit also described the problems that arise out of the fact that users wish to query and update stored data at the same time, and the approaches developed to address these problems together with their respective strengths and weaknesses in a range of practical situations.

8.11 KEYWORDS

ACID: ACID is an acronym for atomicity, consistency, isolation, and durability

Schedule: A schedule is a sequence that indicates the chronological order in which instructions of concurrent transactions are executed.

Serial Schedule: A schedule is said serial if the transactions are executed in a non-interleaved fashion

Serializable: A schedule that is equivalent (in its outcome) to a serial schedule has the serializability property.

Conflict-serializable: A schedule is said to be conflict-serializable when the schedule is conflict-equivalent to one or more serial schedules.

Concurrent execution: Concurrent execution means running side by side or parallely of transactions

Serializability: In concurrency control of databases a transaction schedule has the serializability property, if its outcome is equal to the outcome of its transactions executed serially.

8.12 UNIT-END EXERCISES AND ANSWERS

1. Explain the concept of a transaction
2. Explain the ACID properties in detail.
3. What is schedule? Explain different types of schedules
4. What is meant by interleaved concurrent execution of database transactions in a multiuser system? Discuss why concurrency control is needed, and give informal examples.
5. Explain how to implement atomicity and durability
6. Explain the motivation for concurrent execution of transactions.
7. Explain about Serializability of schedules.
8. Explain Serializability algorithm.
9. Discuss about Testing for Serializability

Answers: SEE

1.	8.1
2.	8.2
3.	8.5
4.	8.3, 8.7
5.	8.3

6.	8.4
7.	8.6
8.	8.8
9.	8.9

8.13 SUGGESTED READINGS

[1] Fundamentals of Database Systems By Ramez Elmasri, Shamkant B. Navathe, Durvasula V.L.N. Somayajulu, Shyam K.Gupta

[2] Database System Concepts By Avi Silberschatz, Henry F. Korth , S. Sudarshan

[3] Database Management Systems By Raghu Ramakrishnan and Johannes Gehrke

[4] An Introduction to Database Systems C.J.Date

[5] Database Management Systems Alexis Leon, Mathews, Vikas Publishing House

Module – 3

UNIT-9: Concurrency Control

Structure

9.0 Objectives

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

- Describe Locking techniques of Concurrency control
- Define concurrency control and recovery are necessary in a database system
- Critically compare the relative strengths of different concurrency control approaches

9.1Introduction

Concurrency control in Database management systems, other transactional objects, and related distributed applications (e.g., Grid computing and Cloud computing) ensures that database transactions are performed concurrently without violating the data integrity of the respective databases. Thus concurrency control is an essential element for correctness in any system where two database transactions or more, executed with time overlap, can access the same data, e.g., virtually in any general-purpose database system. A well-established concurrency control theory exists for database systems: serializability theory, which allows effectively designing and analyzing concurrency control methods and mechanisms.

To ensure correctness, A DBMS usually guarantees that only serializable transaction schedules are generated, unless serializability is intentionally relaxed. For maintaining correctness in cases of failed (aborted) transactions (which can always happen for many reasons) schedules also need to have the recoverability property. A DBMS also guarantees that no effect of committed transactions is lost, and no effect of aborted (rolled back) transactions remains in the related database.

9.2Overview of Concurrency Control

Concurrency control is a database management systems (DBMS) concept that is used to address conflicts with the simultaneous accessing or altering of data that can occur with a multi-user system. Concurrency control, when applied to a DBMS, is meant to coordinate simultaneous transactions while preserving data integrity. The Concurrency is about to control the multi-user access of Database

Illustrative Example

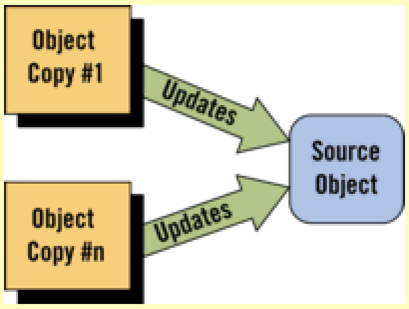
To illustrate the concept of concurrency control, consider two travellers who go to electronic kiosks at the same time to purchase a train ticket to the same destination on the same train. There's only one seat left in the coach, but without concurrency control, it's possible that both travellers will end up purchasing a ticket for that one seat. However, with concurrency control, the database wouldn't allow this to happen. Both travellers would still be able to access the train seating database, but concurrency control would preserve data accuracy and allow only one traveller to purchase the seat.This example also illustrates the importance of addressing this issue in a multi-user database. Obviously, one could quickly run into problems with the inaccurate data that can result from several transactions occurring simultaneously and writing over each other. The following section provides strategies for implementing concurrency control.

The majority of software development today, as it has been for several decades, focuses on multiuser systems. In multiuser systems, there is always a danger that two or more users will attempt to update a common resource, such as shared data or objects, and it's the responsibility of the developers to ensure that updates are performed appropriately. Consider an airline reservation system, for example. A flight has one seat left, and you and I are trying to reserve that seat at the same time. Both of us check the flight status and are told that a seat is still available. We both enter our payment information and click the reservation button at the same time. What should happen? If the system works only one of us will be given a seat and the other will be told that there is no longer a seat available. An effort called concurrency control makes this happen.

Why we need concurrency control?

The problem stems from the fact that to support several users working simultaneously with the same object, the system must make copies of the object for each user, as indicated in Figure 1(below). The source object may be a row of data in a relational database and the copies may be a C++ object in an object database, Regardless of the technology involved, you need to synchronize changes—updates, deletions and creations—made to the copies, ensuring the transactional integrity of the source.

Object Concurrency Control Diagram



Concurrencycontrol synchronizes updates to an object.

Concurrency Control Strategies:

There are three basic object concurrency control strategies: pessimistic, optimistic and truly optimistic. Pessimistic concurrency control locks the source for the entire time that a copy of it exists, not allowing other copies to exist until the copy with the lock has finished its transaction. The copy effectively places a write lock on the appropriate source, performs some work, then applies the appropriate changes to the source and unlocks it. This is a brute force approach to concurrency that is applicable for small-scale systems or systems where concurrent access is very rare: Pessimistic locking doesn't scale well because it blocks simultaneous access to common resources.

Optimistic concurrency control takes a different approach, one that is more complex but is scalable. With optimistic control, the source is uniquely marked each time it's initially accessed by any given copy. The access is typically a creation of the copy or a refresh of it. The user of the copy then manipulates it as necessary. When the changes are applied, the copy briefly locks the source, validates that the mark it placed on the source has not been updated by another copy, commits its changes and then unlocks it. When a copy discovers that the mark on the source has been updated—indicating that another copy of the source has since accessed it—we say that a collision has occurred. A similar but alternative strategy is to check the mark placed on the object previously, if any, to see that it is unchanged at the point of update. The value of the mark is then updated as part of the overall update of the source. The software is responsible for handling the collision appropriately, strategies for which are described below. Since it's unlikely that separate users will access the same object simultaneously, it's better to handle the occasional collision than to limit the size of the system. This approach is suitable for large systems or for systems with significant concurrent access.

Truly optimistic concurrency control, is the most simplistic—it's also effectively unusable for most systems. With this approach, no locks are applied to the source and no unique marks are placed on it; the software simply hopes for the best and applies the changes to the source as they occur. This approach means your system doesn't guarantee transactional integrity if it's possible that two or more users can manipulate a single object simultaneously. Truly optimistic concurrency control is only appropriate for systems that have no concurrent update at all, such as information-only Web sites or single user systems.

9.3Locking techniques

One of the main techniques used to control concurrent execution of transactions (that is to provide serializable execution of transactions) is based on the concept of locking data items. A lock is a variable associated with a data item in the database and describes the status of that data item with respect to possible operations that can be applied to the item. Generally speaking, there is one lock for each data item in the database. The overall purpose of locking is to obtain maximum concurrency and minimum delay in processing transactions.

Locking: The idea of locking is simple: when a transaction needs an assurance that some object, typically a database record that it is accessing in some way, will not change in some unpredictable manner while the transaction is not running on the CPU, it acquires a lock on that object. The lock prevents other transactions from accessing the object. Thus the first transaction can be sure that the object in question will remain in a stable state as long as the transaction desires.

Types of Locks:

- Binary Locks
- Shared/Exclusive (or Read/Write) Locks
- Conversion of Locks

Binary Lock

A binary lock can have two states or values: locked and unlocked (or 1 and 0, for simplicity). A distinct lock is associated with each database item X. If the value of the lock on X is 1, item X cannot be accessed by a database operation that requests the item. If the value of the lock on X is 0, the item can be accessed when requested. We refer to the current value (or state) of the lock associated with item X as LOCK(X). Two operations, lock_item and unlock_item, are used with binary locking. A transaction requests access to an item X by first issuing a lock_item(X) operation. If LOCK(X) = 1, the transaction is forced to wait. If LOCK(X) = 0, it is set to 1 (the transaction locks the item) and the transaction is allowed to access item X. When the transaction is through using the item, it issues an unlock_item(X) operation, which sets LOCK(X) to 0 (unlocks the item) so that X may be accessed by other transactions. Hence, a binary lock enforces mutual exclusion on the data item.

Note: The lock_item and unlock_item operations must be implemented as indivisible units (atomic unit operations)

If the simple binary locking scheme described here is used, every transaction must obey the following rules:

1. A transaction T must issue the operation lock_item(X) before any read_item(X) or write_item(X) operations are performed in T.

2. A transaction T must issue the operation unlock_item(X) after all read_item(X) and write_item(X) operations are completed in T.

3. A transaction T will not issue a lock_item(X) operation if it already holds the lock on item X.

4. A transaction T will not issue an unlock_item(X) operation unless it already holds the lock on item X.

Between the lock_item(X) and unlock_item(X) operations in transaction T, T is said to hold the lock on item X. At most one transaction can hold the lock on a particular item.

Disadvantages:

It is too restrictive for database items, because at most one transaction can hold a lock on a given item. We should allow several transactions to access the same item X if they all access X for reading purposes only. However, if a transaction is to write an item X, it must have exclusive access to X.

Shared/Exclusive (or Read/Write) Locks

To recover from binary lock disadvantage, a different type of lock called a multiple-mode lock is used. In this scheme, called shared/exclusive or read/write locks, there are three locking operations: read_lock(X), write_lock(X), and unlock(X). A lock associated with an item X, LOCK(X), now has three possible states: "read-locked," "write-locked," or "unlocked." A read-locked item is also called share-locked, because other transactions are allowed to read the item, whereas a write-locked item is called exclusive-locked, because a single transaction exclusively holds the lock on the item.

Note: Each of the three operations should be considered indivisible.

When we use the shared/exclusive locking scheme, the system must enforce the following rules:

1. A transaction T must issue the operation read_lock(X) or write_lock(X) before any read_item(X) operation is performed in T.

2. A transaction T must issue the operation write_lock(X) before any write_item(X) operation is performed in T.

3. A transaction T must issue the operation unlock(X) after all read_item(X) and write_item(X) operations are completed in T.

4. A transaction T will not issue a read_lock(X) operation if it already holds a read (shared) lock or a write (exclusive) lock on item X. This rule may be relaxed.

5. A transaction T will not issue a write_lock(X) operation if it already holds a read (shared) lock or write (exclusive) lock on item X. This rule may be relaxed.

6. A transaction T will not issue an unlock(X) operation unless it already holds a read (shared) lock or a write (exclusive) lock on item X.

Conversion of Locks

Sometimes it is desirable to relax conditions 4 and 5 in the preceding list in order to allow lock conversion; that is, a transaction that already holds a lock on item X is allowed under certain conditions to convert the lock from one locked state to another. For example, it is possible for a transaction T to issue a read_lock(X) and then later on to upgrade the lock by issuing a write_lock(X) operation. If T is the only transaction holding a read lock on X at the time it issues the write_lock(X) operation, the lock can be upgraded; otherwise, the transaction must wait. It is also possible for a transaction T to issue a write_lock(X) and then later on to downgrade the lock by issuing a read_lock(X) operation.

Lock compatibility:

Lock Compatibility Matrix lists compatibility between lock types. + means compatible, - means incompatible.

Table 1. Lock Compatibility Matrix

	Shared	Update	Exclusive
Shared	+	+	-

Update	+	-	-
Exclusive	-	-	-

Use of the Locking Scheme

Using binary locks or multiple-mode locks in transactions as described earlier does not guarantee serializability of schedules in which the transactions participate. For example, two simple transactions T1 and T2 are shown below.

T_1	T_2
read_lock(Y);	read_lock(X);
read_item(Y);	read-item(X);
unlock(Y);	unlock(X);
write_lock(X);	write_lock(Y);
read_item(X);	read_item(Y);
X:=X+Y;	Y:=X+Y;
write_item(X);	write_item(Y);
unlock(X);	unlock(Y);

Schedule 1

Assume initially, X = 20 and Y = 30, the result of serial schedule T1 followed by T2 is X = 50 and Y = 80; and the result of serial schedule T2 followed by T1 is X = 70 and Y = 50. The figure below shows an example where although the multiple-mode locks are used, a nonserializable schedule may still result.

T_1	T_2
read_lock(Y);	
read_item(Y);	
unlock(Y);	
	read_lock(X);
	read-item(X);
	unlock(X);
	write_lock(Y);
	read_item(Y);
	Y:=X+Y;
	write_item(Y);
	unlock(Y);
write_lock(X);	
read_item(X);	
X:=X+Y;	
write_item(X);	
unlock(X);	

Schedule 2

The reason for this nonserializable schedule to occur is that the items Y in T1 and X in T2 were unlocked too early. To guarantee serializability, we must follow an additional protocol concerning the positioning of locking and unlocking operations in every transaction.The best known protocol, two-phase locking, is described below.

9.4 Lock based protocols

Basic 2PL

A transaction is said to follow the two-phase locking protocol (basic 2-PL protocol) if all locking operations (read_lock, write_lock) precede the first unlock operation in the transaction. Such a transaction can be divided into two phase: an expanding (or growing) phase, during which new locks on items can be acquired but none can be released; and a shrinking phase, during which existing locks can be released but no new locks can be acquired.

Transactions T1 and T2 shown in the Schedule 2 do not follow the 2-PL protocol. This is because the write_lock(X) operation follows the unlock(Y) operation in T1, and similarly the write_lock(Y) operation follows the unlock(X) operation in T2. If we enforce 2-PL, the transactions can be rewritten as T1' and T2', as shown below.

T_1'	T_2'
read_lock(Y);	read_lock(X);
read_item(Y);	read-item(X);
write_lock(X);	write_lock(Y);
unlock(Y);	unlock(X);
read_item(X);	read_item(Y);
X:=X+Y;	Y:=X+Y;
write_item(X);	write_item(Y);
unlock(X);	unlock(Y);

Schedule 3

Now the schedule involves interleaved operations shown in Schedule 3 is not permitted. This is because T1' will issue its write_lock(X) before it unlocks Y; consequently, when T2' issues its read_lock(X), it is forced to wait until T1' issues its unlock(X) in the schedule.

It can be proved that, if every transaction in a schedule follows the basic 2-PL, the schedule is guaranteed to be serializable, obviating the need to test for serializability of schedules any more. The locking mechanism, by enforcing 2-PL rules, also enforces serializability.

It can be proved that, if every transaction in a schedule follows the basic 2-PL, the schedule is guaranteed to be serializable, obviating the need to test for serializability of schedules any more. The locking mechanism, by enforcing 2-PL rules, also enforces serializability.

Another problem that may be introduced by 2-PL protocol is deadlock. The formal definition of deadlock will be discussed below. Here, an example is used to give you an intuitive idea about the deadlock situation. The two transactions that follow the 2-PL protocol can be interleaved as shown here.

Time	T_1’	T_2’
1	read_lock(Y);	
2	read_item(Y);	
3		read_lock(X);
4		read-item(X);
5	write_lock(X);	
	wait	
6		write_lock(Y);
		wait
	...	...

Schedule 4

At time step 5, it is not possible for T1’ to acquire an exclusive lock on X as there is already a shared lock on X held by T2’. Therefore, T1’ has to wait. Transaction T2’ at time step 6 tries to get an exclusive lock on Y, but it is unable to as T1’ has a shared lock on Y already. T2’ is put in waiting too. Therefore, both transactions wait fruitlessly for the other to release a lock. This situation is known as a deadlock.

Conservative 2PL

A variation of the basic 2-PL is conservative 2-PL also known as static 2-PL, which is a way of avoiding deadlock. The conservative 2-PL requires a transaction to lock all the data items it needs in advance. If at least one of the required data items cannot be obtained then none of the items are locked. Rather, the transaction waits and then tries again to lock all the items it needs. Although conservative 2-PL is a deadlock-free protocol, this solution further limits concurrency.

Strict 2PL

In practice, the most popular variation of 2-PL is strict 2-PL, which guarantees strict schedule. (Strict schedules are those in which transactions can neither read nor write an item X until the last transaction that wrote X has committed or aborted). In strict 2-PL, a transaction T does not release any of its locks until after it commits or aborts. Hence, no other transaction can read or write an item that is written by T unless T has committed, leading to a strict schedule for recoverability. Notice the difference between conservative and strict 2-PL, the former must lock all items before it starts, whereas the latter does not unlock any of its items until after it terminates (by committing or aborting). Strict 2-PL is not deadlock-free unless it is combined with conservative 2-PL.

In summary, all type 2-PL protocols guarantee serializability (correctness) of a schedule but limit concurrency. The use of locks can also cause two additional problems: deadlock and livelock. Conservative 2-PL is deadlock free.

9.5Time stamp based protocols

A timestamp is a tag that can be attached to any transaction or any data item, which denotes a specific time on which the transaction or data item had been activated in any way. We, who use computers, must all be familiar with the concepts of “Date Created” or “Last Modified” properties of files and folders. Well, timestamps are things like that.

A timestamp can be implemented in two ways. The simplest one is to directly assign the current value of the clock to the transaction or the data item. The other policy is to attach the value of a logical counter that keeps incrementing as new timestamps are required.

The timestamp of a transaction denotes the time when it was first activated. The timestamp of a data item can be of the following two types:

W-timestamp (Q): This means the latest time when the data item Q has been written into.

R-timestamp (Q): This means the latest time when the data item Q has been read from.

These two timestamps are updated each time a successful read/write operation is performed on the data item Q.

How should timestamps be used?

The timestamp ordering protocol ensures that any pair of conflicting read/write operations will be executed in their respective timestamp order. This is an alternative solution to using locks.

For Read operations:

1. If TS (T) < W-timestamp (Q), then the transaction T is trying to read a value of data item Q which has already been overwritten by some other transaction. Hence the value which T wanted to read from Q does not exist there anymore, and T would be rolled back.

2. If TS (T) >= W-timestamp (Q), then the transaction T is trying to read a value of data item Q which has been written and committed by some other transaction earlier. Hence T will be allowed to read the value of Q, and the R-timestamp of Q should be updated to TS (T).

For Write operations:

1. If TS (T) < R-timestamp (Q), then it means that the system has waited too long for transaction T to write its value, and the delay has become so great that it has allowed another transaction to read the old value of data item Q. In such a case T has lost its relevance and will be rolled back.
2. Else if TS (T) < W-timestamp (Q), then transaction T has delayed so much that the system has allowed another transaction to write into the data item Q. in such a case too, T has lost its relevance and will be rolled back.
3. Otherwise the system executes transaction T and updates the W-timestamp of Q to TS (T).

9.6Commit 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 itmust abort at all sites. To ensure this property, the transaction coordinator of T must execute a commit protocol. Among the simplest and most widely used commit protocols is the two-phasecommit protocol (2PC), which is described below.

Two-Phase Commit

We first describe how the two-phase commit protocol (2PC) operates during normal operation, then describe how it handles failures and finally how it carries out recoveryand concurrency control.

Consider a transaction T initiated at site Si, where the transaction coordinator is Ci.

The Commit Protocol

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

- Phase 1. Ci adds the record<prepare T> to the log, and forces the log onto stable storage. It then sends a prepare T message to all sites at which T executed. On receiving such a message, the transaction manager at that site determines whether it is willing to commit its portion of T. If the answer is no, it adds a record <no T> to the log, and then responds by sending an abort T message to Ci. If the answer is yes, it adds a record <ready T> to the log, and forces the log (with all the log records corresponding to T) onto stable storage. The transaction manager then replies with a ready T message to Ci.

- Phase 2. When Ci receives responses to the prepare T message from all the sites, or when a prespecified interval of time has elapsed since the prepare T message was sent out, Ci can determine whether the transaction T can be committed or aborted. Transaction T can be committed if Ci received a ready T message from all the participating sites. Otherwise, transaction T must be aborted. Depending on the verdict, either a record <commit T> or a record<abort T> is added to the log and the log is forced onto stable storage. At this point, the fate of the transaction has been sealed. 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.

9.7Optimistic technique

In all the concurrency control techniques we have discussed so far, a certain degree of checking is done before a database operation can be executed. For example, in locking, a check is done to determine whether the item being accessed is locked. In timestamp ordering, the transaction timestamp is checked against the read and write timestamps of the item. Such checking represents overhead during transaction execution, with the effect of slowing down the transactions.

In optimistic concurrency control techniques, also known as validation or certification techniques, no checking is done while the transaction is executing. Several proposed concurrency control methods use the validation technique. We will describe only one scheme here. In this scheme, updates in the transaction are not applied directly to the database items until the transaction reaches its end. During transaction execution, all updates are applied to local copies of the data items that are kept for the transaction (Note 6). At the end of transaction execution, a validation phase checks whether any of the transaction's updates violate serializability. Certain information needed by the validation phase must be kept by the system. If serializability is not violated, the transaction is committed and the database is updated from the local copies; otherwise, the transaction is aborted and then restarted later.

There are three phases for this concurrency control protocol:

1. Read phase: A transaction can read values of committed data items from the database.

However, updates are applied only to local copies (versions) of the data items kept in the transaction workspace.

2. Validation phase: Checking is performed to ensure that serializability will not be violated if the transaction updates are applied to the database.
3. Write phase: If the validation phase is successful, the transaction updates are applied to the database; otherwise, the updates are discarded and the transaction is restarted.

The idea behind optimistic concurrency control is to do all the checks at once; hence, transaction execution proceeds with a minimum of overhead until the validation phase is reached. If there is little interference among transactions, most will be validated successfully. However, if there is much interference, many transactions that execute to completion will have their results discarded and must be restarted later. Under these circumstances, optimistic techniques do not work well. The techniques are called "optimistic" because they assume that little interference will occur and hence that there is no need to do checking during transaction execution.

9.8Time stamp ordering multi version concurrency control

In this method, several versions, of each data item X are maintained. For each version, the value of version and the following two timestamps are kept:

1. read_TS: The read timestamp of is the largest of all the timestamps of transactions that have successfully read version .
2. write_TS: The write timestamp of is the timestamp of the transaction that wrote the value of version .

Whenever a transaction T is allowed to execute a write_item(X) operation, a new version of item X is created, with both the write_TS and the read_TS set to TS(T). Correspondingly, when a transaction T is allowed to read the value of version Xi, the value of read_TS() is set to the larger of the current read_TS() and TS(T).

To ensure serializability, the following two rules are used:

1. If transaction T issues a write_item(X) operation, and version i of X has the highest write_TS() of all versions of X that is also less than or equal to TS(T), and read_TS() > TS(T), then abort and roll back transaction T; otherwise, create a new version of X with read_TS() = write_TS() = TS(T).
2. If transaction T issues a read_item(X) operation, find the version i of X that has the

highest write_TS() of all versions of X that is also less than or equal to TS(T); then return the value of to transaction T, and set the value of read_TS() to the larger of TS(T) and the current read_TS().

As we can see in case 2, a read_item(X) is always successful, since it finds the appropriate version to read based on the write_TS of the various existing versions of X. In case 1, however, transaction T may be aborted and rolled back. This happens if T is attempting to write a version of X that should have been read by another transaction T whose timestamp is read_TS(); however, T has already read version Xi, which was written by the transaction with timestamp equal to write_TS(). If this conflict occurs, T is rolled back; otherwise, a new version of X, written by transaction T, is created. Notice that, if T is rolled back, cascading rollback may occur. Hence, to ensure recoverability, a transaction T should not be allowed to commit until after all the transactions that have written some version that T has read have committed.

9.9Deadlock handling

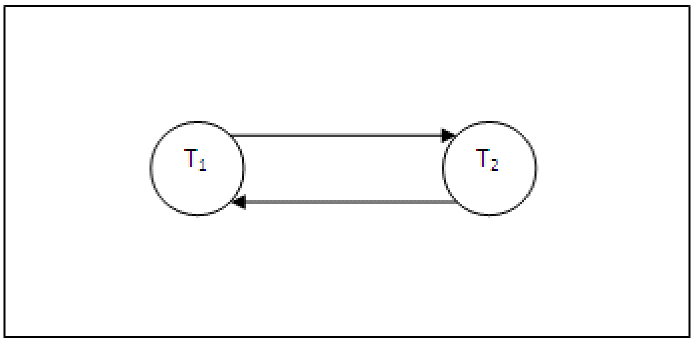
Deadlock occurs when each of two transactions is waiting for the other to release the lock on an item. A simple example is shown in Schedule 4 above, where the two transactions T1' and T2' are deadlocked in a partial schedule; T1' is waiting for T2' to release item X, while T2' is waiting for T1' to release item Y. Meanwhile, neither can proceed to unlock the item that the other is waiting for, and other transactions can access neither item X nor item Y. Deadlock is also possible when more than two transactions are involved.

Deadlock Detection with Wait-for Graph

A simple way to detect a state of deadlock is to construct a wait-for graph. One node is created in the graph for each transaction that is currently executing in the schedule. Whenever a transaction Ti is waiting to lock an item X that is currently locked by a transaction Tj, create a directed edge (Ti Tj). When Tj releases the lock(s) on the items that Ti was waiting for, the directed edge is dropped from the waiting-for graph. We have a state of deadlock if and only if the wait-for graph has a cycle. Recall this partial schedule introduced previously.

Time	T_1'	T_2'
1	read_lock(Y);	
2	read_item(Y);	
3		read_lock(X);
4		read-item(X);
5	write_lock(X);	
	wait	
6		write_lock(Y);
		wait
	...	...

The wait-for graph for the above partial schedule is shown below.



$$\begin{tikzcd} T_1 \arrow[r, bend left] & T_2 \arrow[l, bend left] \end{tikzcd}$$

One problem with using the wait-for graph for deadlock detection is the matter of determining when the system should check for deadlock. Criteria such as the number of concurrently executing transactions or the period of time several transactions have been waiting to lock items may be used to determine that the system should check for deadlock.

When we have a state of deadlock, some of the transactions causing the deadlock must be aborted. Choosing which transaction to abort is known as victim selection. The algorithm for victim selection should generally avoid selecting transactions that have been running for a long time and that have performed many updates, and should try instead to select transactions that have not made many changes or that are involved in more than one deadlock cycle in the wait-for graph. A problem known as cyclic restart may occur, where a transaction is aborted and restarted only to be involved in another deadlock. The victim selection algorithm can use higher priorities for transactions that have been aborted multiple times so that they are not selected as victims repeatedly.

Livelock

Another problem that may occur when we use locking is livelock. A transaction is in a state of livelock if it cannot proceed for an indefinite period while other transactions in the system continue normally. This may occur if the waiting scheme for locked items is unfair, giving priority to some transactions over others. The standard solution for livelock is to have a fair waiting scheme. One such scheme uses a first-come-first-serve queue; transactions are enabled to lock an item in the order in which they originally requested to lock the item. Another scheme allows some transactions to have priority over others but increases the priority of a transaction the longer it waits, until it eventually gets the highest priority and proceeds.

9.10 Summary

Concurrency control is the management of contention for data resources. Concurrency control ensures that database transactions are performed concurrently without violating the data integrity of the respective databases. This unit discussed Lock-based Concurrency Control, Two Phase Locking Protocols and its variations, Deadlocks, and livelocks.

9.11Keywords

Commit protocols, Lock based protocols, Concurrency control, Deadlock handling;

9.12Exercises

1. What is meant by interleaved concurrent execution of database transactions in a multiuser system?
2. Discuss why concurrency control is needed, and give informal examples.
3. Explain different types of anomalies due to concurrent execution of transactions.
4. Explain the motivation for concurrent execution of transactions.

9.13Reference

1. Fundamentals of Database Systems By RamezElmasri, Shamkant B. Navathe, Durvasula V.L.N. Somayajulu, ShyamK.Gupta
2. Database System Concepts By AviSilberschatz, Henry F. Korth , S. Sudarshan

3. Database Management Systems By Raghu Ramakrishnan and Johannes Gehrke

UNIT-10: Recovery mechanisms

Structure

10.0 Objectives

After going through this unit, you will be able to: Define Database Backup and Recovery

- Describe about Recovery mechanisms
- Define Crashes due to hardware malfunction
- Define Recovery from transaction failure

10.1Introduction

Our emphasis is on conceptually describing several different approaches to recovery. For descriptions of recovery features in specific systems, the reader should consult the bibliographic reference. Recovery techniques are often intertwined with the concurrency control mechanisms. Certain recovery techniques are best used with specific concurrency control methods. We will attempt to discuss recovery concepts independently of concurrency control mechanisms, but we will discuss the circumstances under which a particular recovery mechanism is best used with a certain concurrency control protocol.

10.2Recovery mechanisms

Database systems, like any other computer systems, are subject to failures. Despite this, any organization that depends upon a database must have that database available when it is required. Therefore, any DBMS intended for a serious business or organizational user must have adequate facilities for fast recovery after failure. In particular, whenever a transaction is submitted to a DBMS for execution, the system must ensure that either all the operations in the transaction are completed successfully and their effect is recorded permanently in the database or the transaction has no effect whatsoever on the database/on any other transactions.

The understanding of the methods available for recovering from such failures is therefore essential to any serious study of database. This unit describes the methods available for recovering from a range of problems that can occur throughout the life of a database system. These mechanisms include automatic protection mechanisms built into the database software itself, and non-automatic actions available to people responsible for the running of the database system, both for the backing up of data and recovery for a variety of failure situations.

Recovery techniques are intertwined with the concurrency control mechanisms: certain recovery techniques are best used with specific concurrency control methods. Assume for the most part that we are dealing with a large multi-user database: small systems typically provide little or no support for recovery; in these systems recovery is regarded as a user problem.

10.3 Crash recovery

Recovery ensures database is fault tolerant, and not corrupted by software, system or media failure.

Recovery Manager: When a DBMS is restarted after crashes, the recovery manager must bring the database to a consistent state. It maintains log information during normal execution of transactions for use during crash recovery. Recovery Manager is responsible for ensuring transaction atomicity and durability.

– Ensures atomicity by undoing the actions of transactions that do not commit.
– Ensures durability by making sure that all actions of committed transactions survive system crashes and media failure.

Need for Recovery

Whenever a transaction is submitted to a DBMS for execution, the system is responsible for making sure that either:

1. all the operations in the transaction are completed successfully and the effect is recorded permanently in the database, or
2. the transaction has no effect whatsoever on the database or on any other transactions.

The DBMS must not permit some operations of a transaction T to be applied to the database while other operations of T are not. This may happen if a transaction fails after executing some of its operations but before executing all of them.

Transaction Problems

The practical aspects of transactions are about keeping control. There are a variety of causes of transaction failure. These may include:

1. Concurrency control enforcement: concurrency control method may abort the transaction, to be restarted later, because it violates serializability, or because several transactions are in a state of deadlock.
2. Local error detected by the transaction: during transaction executions, certain conditions may occur that necessitates cancellation of the transaction. (e.g., an account with insufficient funds may cause a withdrawal transaction from that account to be cancelled.) This may be done by a programmed ABORT in the transaction itself.
3. A transaction or system error: due to some operation in the transaction may cause it to fail, such as integer overflow, division by zero, erroneous parameter values or logical programming errors.
4. User interruption of the transaction during its execution, e.g., by issuing a control-C in a VAX/VMS or UNIX environment.

5. System software errors that result in abnormal termination or destruction of the database management system.
6. Crashes due to hardware malfunction resulting in loss of internal (main and cache) memory (otherwise known as system crashes).
7. Disk malfunctions such as read or write malfunction or a disk read/write head crash. This may happen during a read or write operation of the transaction.
8. Natural physical disasters and catastrophes such as fires, earthquakes or power surges; sabotages, intentional contamination with computer viruses, or destruction of data or facilities by operators or users.

Failures of types 1 to 6 are more common than those of types of 7 or 8. Whenever a failure of type 1 through 6 occurs, the system must keep sufficient information to recover from the failure. Disk failure or other catastrophic failures of 7 or 8 do not happen frequently; if they do occur, it is a major task to recover from these types of failure.

10.4Recovery from transaction failure

Start a Transaction

In SQL, a transaction can be started either implicitly or explicitly. A transaction starts implicitly when certain types of SQL statements are executed, such as the SELECT, DELETE, UPDATE, and CREATE TABLE statements. These types of statements must be executed within the context of a transaction. If a transaction is not active, one is initiated.

Transactions can also be initiated explicitly by using the START TRANSACTION statement. The START TRANSACTION statement serves two purposes: to set the transaction's properties, and to initiate the transaction. In terms of setting the properties, the START TRANSACTION statement works just like the SET TRANSACTION statement. You can set the access level, the isolation level, and the diagnostics size. As for initiating a transaction, you simply execute the START TRANSACTION statement.
The syntax for the START TRANSACTION statement is similar to the SET TRANSACTION statement, as you can see in the following syntax

```
START TRANSACTION <mode>[ { , <mode> } . . . ]
```

After you specify the START TRANSACTION keywords, you must specify one or more transaction modes. As with the SET TRANSACTION statement, you can include only one mode for each type.

Terminate a Transaction

Earlier in this unit, you learned that a transaction can be initiated either explicitly or implicitly. The same thing is true for ending a transaction. You can explicitly commit or roll back a transaction, which then terminates the transaction, or the transaction is terminated implicitly when circumstances force that termination.

In SQL, there are four primary circumstances that will terminate a transaction:

- A ROLLBACK statement is explicitly defined in the transaction. When the statement is executed, actions are undone, the database is returned to the state it was in when the transaction was initiated, and the transaction is terminated. If the ROLLBACK statement references a savepoint, only the actions taken after the savepoint are undone, and the transaction is not terminated.
- A COMMIT statement is explicitly defined in the transaction. When the statement is executed, all transaction-related changes are saved to the database, and the transaction is terminated.
- The program that initiated the transaction is interrupted, causing the program to abort. In the event of an abnormal interruption, which can be the result of hardware or software problems, all changes are rolled back, the database is returned to its original state, and the transaction is terminated. A transaction terminated in this way is similar to terminating a transaction by using a ROLLBACK statement.
- The program successfully completes its execution. All transaction-related changes are saved to the database, and the transaction is terminated. Once these changes are committed, they cannot be rolled back. A transaction terminated in this way is similar to terminating a transaction by using a COMMIT statement.

As you can see, the ROLLBACK and COMMIT statements allow you to explicitly terminate a transaction, whereas a transaction is terminated implicitly when the program ends or is interrupted. These methods of termination ensure that data integrity is maintained and the database is protected. No changes are made to the database unless the transaction is complete.

10.5Recovery in a Centralized DBMS

The recovery in a distributed DBMS is more complicated than in a centralized DBMS, as it requires ensuring the atomicity of global transactions as well as of local transactions. Therefore, it is necessary to modify the commit and abort processing in a distributed DBMS, so that a global transaction does not commit or abort until all sub transactions of it have successfully committed or aborted. In addition, the distributed recovery protocols must have the capability to deal with different types of failures such as site failures, communication link failures and network partitions. Termination protocols are unique to distributed database systems. In a distributed DBMS, the execution of a global transaction may involve several sites, and if one of the participating sites fails during the execution of the global transaction, termination protocols are used to terminate the transaction at the other participating sites. Similarly, in the case of network partition, termination protocols are used to terminate the active transactions that are being executed at different partitions.

One desirable property for distributed recovery protocols is independency. An independent recovery protocol decides how to terminate a transaction that was executing at the time of a failure without consulting any other site. Moreover, the distributed recovery protocols should cater for different types of failures in a distributed system to ensure that the failure of one site does not affect processing at another site. In other words, operational sites should not be left blocked. Protocols that obey this property are known as non-blocking protocols. It is preferable that the termination protocols are non-blocking. A non-blocking termination protocol allows a transaction to terminate at the operational sites without waiting for recovery of the failed site.

10.6Database recovery techniques based on immediate and deferred updates

In all the concurrency control techniques we have discussed so far, a certain degree of checking is done before a database operation can be executed. For example, in locking, a check is done to determine whether the item being accessed is locked. In timestamp ordering, the transaction timestamp is checked against the read and write timestamps of the item. Such checking represents overhead during transaction execution, with the effect of slowing down the transactions.

Crash or instance recovery recovers a database to its transaction-consistent state just before instance failure. Crash recovery recovers a database in a single-instance configuration and instance recovery recovers a database in an Oracle Parallel Server configuration. If all instances of an Oracle Parallel Server database crash, then Oracle performs crash recovery.

Recovery from instance failure is automatic, requiring no DBA intervention. For example, when using the Oracle Parallel Server, another instance performs instance recovery for the failed instance. In single-instance configurations, Oracle performs crash recovery for a database when the database is restarted, that is, mounted and opened to a new instance. The transition from a mounted state to an open state automatically triggers crash recovery, if necessary.

Crash or instance recovery consists of the following steps:

1. Rolling forward to recover data that has not been recorded in the datafiles, yet has been recorded in the online redo log, including the contents of rollback segments. This is called cache recovery.

2. Opening the database. Instead of waiting for all transactions to be rolled back before making the database available, Oracle allows the database to be opened as soon as cache recovery is complete. Any data that is not locked by unrecovered transactions is immediately available.

3. Marking all transactions system-wide that were active at the time of failure as DEAD and marking the rollback segments containing these transactions as PARTLY AVAILABLE.

4. Rolling back dead transactions as part of SMON recovery. This is called transaction recovery.
5. Resolving any pending distributed transactions undergoing a two-phase commit at the time of the instance failure.

6. As new transactions encounter rows locked by dead transactions, they can automatically roll back the dead transaction to release the locks. If you are using Fast-Start Recovery, just the data block is immediately rolled back, as opposed to the entire transaction.

10.7ARIES recovery algorithm

ARIES (Algorithms for Recovery and Isolation Exploiting Semantics') uses a steal/no-force approach for writing, and it is based on three concepts:

- Write-ahead logging
- Repeating history during redo - ARIES will retrace all actions of the database system prior to the crash to reconstruct the database state when the crash occurred. Transactions that were uncommitted at the time of the crash (active transactions) are undone.
- Logging changes during undo - will prevent ARIES from repeating the completed undo operations if a failure occurs during recovery, which causes a restart of the recovery process.

There are 3 phases in the Aries recovery algorithm:

- **Analysis:** Scan the log forward (from the most recent checkpoint) to identify all transactions that were active, and all dirty pages in the buffer pool at the time of the crash.
- **Redo:** Redo all updates to dirty pages in the buffer pool, as needed, to ensure that all logged updates are in fact carried out and written to disk.
- **Undo:** The writes of all transactions that were active at the crash are undone

At the end all committed updates and only those updates are reflected in the database.
Some care must be taken to handle the case of a crash occurring during the recovery process!

10.8Shadow paging

This recovery scheme does not require the use of a log in a single-user environment. In a multiuser environment, a log may be needed for the concurrency control method. Shadow paging considers the database to be made up of a number of fixed-size disk pages (or disk blocks)—say, n—for recovery purposes. A directory with n entries is constructed, where the i th entry points to the i th database page on disk. The directory is kept in main memory if it is not too large, and all references—reads or writes—to database pages on disk go through it. When a transaction begins executing, the current directory—whose entries point to the most recent or current database pages on disk—is copied into a shadow directory. The shadow directory is then saved on disk while the current directory is used by the transaction.

During transaction execution, the shadow directory is never modified. When a write_item operation is performed, a new copy of the modified database page is created, but the old copy of that page is not overwritten. Instead, the new page is written elsewhere—on some previously unused disk block. The current directory entry is modified to point to the new disk block, whereas the shadow directory is not modified and continues to point to the old unmodified disk block. For pages updated by the transaction, two versions are kept. The old version is referenced by the shadow directory, and the new version by the current directory.

To recover from a failure during transaction execution, it is sufficient to free the modified database pages and to discard the current directory. The state of the database before transaction execution is available through the shadow directory, and that state is recovered by reinstating the shadow directory. The database thus is returned to its state prior to the transaction that was executing when the crash occurred, and any modified pages are discarded. Committing a transaction corresponds to discarding the previous shadow directory. Since recovery involves neither undoing nor redoing data items, this technique can be categorized as a NO-UNDO/NO-REDO technique for recovery.

In a multiuser environment with concurrent transactions, logs and checkpoints must be incorporated into the shadow paging technique. One disadvantage of shadow paging is that the updated database pages change location on disk. This makes it difficult to keep related database pages close together on disk without complex storage management strategies. Furthermore, if the directory is large, the overhead of writing shadow directories to disk as transactions commit is significant. A further complication is how to handle garbage collection when a transaction commits. The old pages referenced by the shadow directory that have been updated must be released and added to a list of free pages for future use. These pages are no longer needed after the transaction commits. Another issue is that the operation to migrate between current and shadow directories must be implemented as an atomic operation.

10.9Buffer management

In this section, we consider several subtle details that are essential to the implementation of a crash-recovery scheme that ensures data consistency and imposes aminimalamount of overhead on interactions with the database.

Log-Record Buffering

So far, we have assumed that every log record is output to stable storage at the time it is created. This assumption imposes a high overhead on system execution for severalreasons: Typically, output to stable storage is in units of blocks. In most cases, a log record is much smaller than a block. Thus, the output of each log record translates toa much larger output at the physical level. The output of a block to stable storage may involve several output operations at thephysical level.

The cost of performing the output of a block to stable storage is sufficiently high that it is desirable to output multiple log records at once. To do so, we write logrecords to a log buffer in main memory, where they stay temporarily until they are output to stable storage. Multiple log records can be gathered in the log buffer, andoutput to stable storage in a single output operation. The order of log records in the stable storage must be exactly the same as the order in which they were written tothe log buffer.

As a result of log buffering, a log record may reside in only main memory (volatile storage) for a considerable time before it is output to stable storage. Since such logrecords are lost if the system crashes, we must impose additional requirements on the recovery techniques to ensure transaction atomicity:

• Transaction Ti enters the commit state after the <Ti commit> log record has been output to stable storage.
• Before the <Ti commit> log record can be output to stable storage, all log records pertaining to transaction Ti must have been output to stable storage.
• Before a block of data in main memory can be output to the database (in nonvolatile storage), all log records pertaining to data in that block must havebeen output to stable storage.

This rule is called the write-ahead logging (WAL) rule. (Strictly speaking,the WAL rule requires only that the undo information in the log have been output to stable storage, and permits the redo information to be written later.The difference is relevant in systems where undo information and redo information are stored in separate log records.)

The three rules state situations in which certain log records must have been output to stable storage. There is no problem resulting from the output of log records earlierthan necessary. Thus, when the system finds it necessary to output a log record to stable storage, it outputs an entire block of log records, if there are enough log recordsin main memory to fill a block. If there are insufficient log records to fill the block, all log records in main memory are combined into a partially full block, and are outputto stable storage. Writing the buffered log to disk is sometimes referred to as a log force.

10.10 Summary

The major objective of a recovery manager is to employ an appropriate technique for recovering the system from a failure, and to preserve the database in the consistent state that existed prior to the failure.Reliability refers to the probability that the system under consideration does not experience any failures in a given time period.Availability refers to the probability that the system can continue its normal execution according to the specification at a given point in time in spite of failures.There are different types of failures that may occur in a distributed database environment such as site failure, link failure, loss of message and network partition.Two distributed recovery protocols are 2PC protocol and 3PC protocol. 2PC is a blocking protocol whereas 3PC is a non-blocking protocol.The termination protocols that are used to handle network partition can be classified into two categories: pessimistic protocols and optimistic protocols. In the case of non-replicated databases, the termination protocols that are used to deal with network partition are pessimistic. In the case of replicated databases, both types of termination protocols can be used.

10.11Keywords

Recovery mechanisms, Crash recovery, ARIES, Buffer management, Shadow paging

10.12Exercises

1. Discuss the different types of transaction failures
2. How are buffering and caching techniques used by the recovery subsystem?
3. How can recovery handle transaction operations that do not affect the database, such as the printing of reports by a transaction?
4. Describe the three phases of the ARIES recovery method.
5. Describe the two-phase commit protocol for multidatabase transactions.

10.13Reference

1. Database System Concepts By AviSilberschatz, Henry F. Korth , S. Sudarshan
2. Database Management Systems By Raghu Ramakrishnan and Johannes Gehrke
3. Database Management Systems Alexis Leon, Mathews, Vikas Publishing House

UNIT-11: Query Interpretation

Structure

11.0 Objectives

After going through this unit, you will be able to: Answer Query Processing and Optimization

- Describe Query interpretation
- Define Log-Record Buffering
- Define Shadow paging

11.1Introduction

The query processor turns user queries and data modification commands into a query plan, which is a sequence of operations (or algorithm) on the database. It translates high level queries to low level commands.

Decisions taken by the query processor:

- Which of the algebraically equivalent forms of a query will lead to the most efficient algorithm?
- For each algebraic operator what algorithm should we use to run the operator?
- How should the operators pass data from one to the other? (eg, main memory buffers, disk buffers)

Oracle supports a large variety of processing techniques in its query processing engine. Here we briefly describe the important ones.

Execution Methods:

A variety of methods exist to access data:

Full-Table Scan: In Oracle, a full-table scan is performed by reading all of the table rows, block by block, until the high-water mark for the table is reached. As a general rule, full-table scans should be avoided unless the SQL query requires a majority of the rows in the table.

Fast Full Index Scan: The fast full index scan is an alternative to a full table scan when there is an index that contains all the columns that are needed for the query

Index scan: Oracle accesses index based on the lowest cost of execution path available with index scan. The query processor creates a start and/or stop key from conditions in the query and uses it to scan to a relevant part of the index.

Index join: If a query needs only a small subset of the columns of a wide table, but no single index contains all those columns, the processor can use an index join to generate the relevant information without accessing the table, by joining several indices that together contain the needed columns.

Cluster and hash cluster access: The processor accesses the data by using the cluster key.

Query optimization in Oracle:

Oracle carries out query optimization in several stages.

A user query will go through 3 phases:

Parsing: At this stage the syntax and semantics are checked, at the end you have a parse tree which represents the query's structure. The statement is normalized so that it can be processed more efficiently, once all the checks have completed it is considered a valid parse tree and is sent to the logical query plan generation stage.

Optimization: The optimizer is used at this stage, which is a cost-based optimizer, this chooses the best access method to retrieve the requested data. It uses statistics and any hints specified in the SQL query. The CBO produces an optimal execution plan for the SQL statement.

The optimization process can be divided in two parts:

1) Query rewrite phase: The parse tree is converted into an abstract logical query plan, the various nodes and branches are replaced by operators of relational algebra.

2) Execution plan generation phase: Oracle transforms the logical query plan into a physical query plan, the physical query or execution plan takes into account the following factors

- The various operations (joins) to be performed during the query
- The order in which the operations are performed
- The algorithm to be used for performing each operation
- The best way to retrieve data from disk or memory
- The best way to pass data from operation to another during the query

The optimizer may well come up with multiple physical plans, all of which are potential execution plans. The optimizer then chooses among them by estimating the costs of each possible plan (based on table and index statistics) and selecting the plan with the lowest cost. This is called the cost-based query optimization (CBO).

Query Transformation: Query processing requires the transformation of your SQL query into an efficient execution plan, Query optimization requires that the best execution plan is the one executed, the goal is to use the least amount of resources possible (CPU and I/O), the more resources a query uses the more impact it has on the general performance of the database.

Some of the major types of transformation and rewrites supported by Oracle are View merging, complex view merging, subquery flattening, materialized view rewrite, star transformation.

Access Path Selection: In relational database management system (RDBMS) terminology, Access Path refers to the path chosen by the system to retrieve data after a structured query language (SQL) request is executed. Access path selection can make a tremendous impact on the overall performance of the system.

Optimization of access path selection maybe gauged using cost formulas with I/O and CPU utilization weight usually considered. Generally, query optimizers evaluate the available paths to data retrieval and estimate the cost in executing the statements using the determined paths or a combination of these paths. Access paths selection for joins where data is taken from than one table is basically done using the nested loop and merging scan techniques. Because joins are more complex, there are some other considerations for determining access path selections for them.

11.2Query interpretation

The amount of data handled by a database management system increases continuously, and it is no longer unusual for a DBMS to manage data sizes of several hundred gigabytes to terabytes. In this context, a critical challenge in a database environment is to develop an efficient query processing technique, which involves the retrieval of data from the database. A significant amount of research has been dedicated to develop highly efficient algorithms for processing queries in different databases, but here the attention is restricted on relational databases. There are many ways in which a complex query can be executed, but the main objective of query processing is to determine which one is the most cost-effective. This complicated task is performed by a DBMS module, called query processor. In query processing, the database users generally specify what data is required rather than the procedure to follow to retrieve the required data. Thus, an important aspect of query processing is query optimization. A query typically has many possible execution strategies, and the process of choosing a suitable one for processing a query is known as query optimization.

A query is expressed by using a high-level language such as SQL (Structured Query Language) in a relational data model. The main function of a relational query processor is to transform a high-level query into an equivalent lower-level query (relational algebra), and the transformation must achieve both correctness and efficiency. The lower-level query actually implements the execution strategy for the given query. In a centralized DBMS, query processing can be divided into four main steps: query decomposition (consisting of scanning, parsing and validation), query optimization, code generation and query execution. In the query decomposition step, the query parser checks the validity of the query and then translates it into an internal form, usually a relational algebraic expression or something equivalent. The query optimizer examines all relational algebraic expressions that are equivalent to the given query and chooses the optimum one that is estimated to be the cheapest. The code generator generates the code for the access plan selected by the query optimizer and the query processor actually executes the query.

1. Why do we need to optimize?
   - A high-level relational query is generally non-procedural in nature.
   - It says ``what", rather than ``how" to find it.
   - When a query is presented to the system, it is useful to find an efficient method of finding the answer, using the existing database structure.
   - Usually worthwhile for the system to spend some time on strategy selection.
   - Typically can be done using information in main memory, with little or no disk access.
   - Execution of the query will require disk accesses.
   - Transfer of data from disk is slow, relative to the speed of main memory and the CPU
   - It is advantageous to spend a considerable amount of processing to save disk accesses.

2. Do we really optimize?
   - Optimizing means finding the best of all possible methods.
   - The term ``optimization" is a bit of a misnomer here.
   - Usually the system does not calculate the cost of all possible strategies.
   - Perhaps ``query improvement" is a better term.
3. Two main approaches:
   - Rewriting the query in a more effective manner.
   - Estimating the cost of various execution strategies for the query.
   - Usually both strategies are combined.
   - The difference in execution time between a good strategy and a bad one may be huge.
   - Thus this is an important issue in any DB system.

- In network and hierarchical systems, optimization is left for the most part to theapplication programmer.
- Since the DML language statements are embedded in the host language, it is not easy to transform a hierarchical or network query to another one, unless one has knowledge about the entire application program.
- As a relational query can be expressed entirely in a relational query language without the use of a host language, it is possible to optimize queries automatically.
- SQL is suitable for human use, but internally a query should be represented in a more useful form, like the relational algebra.

4. So, first the system must translate the query into its internal form. Then optimization begins:
   - Find an equivalent expression that is more efficient to execute.
   - Select a detailed strategy for processing the query. (Choose specific indices to use, and order in which tuples are to be processed, etc.)
5. Final choice of a strategy is based primarily on the number of disk accesses required.

11.3Equivalence of Expressions

Recovery ensures database is fault tolerant, and not corrupted by software, system or media failure.

The first step in selecting a query-processing strategy is to find a relational algebra expression that is equivalent to the given query and is efficient to execute.We'll use the following relations as examples:

Customer-scheme = (cname, street, ccity)
Deposit-scheme = (bname, account#, name, balance)
Branch-scheme = (bname, assets, bcity)

We will use instances customer, deposit and branch of these schemes.

Selection Operation

Consider the query to find the assets and branch-names of all banks who have depositors livingin Port Chester. In relational algebra, this is

$\Pi_{bname,assets}(\sigma_{ccity="Port\ Chester"}$
(customer $\bowtie$ deposit $\bowtie$ branch))

- This expression constructs a huge relation, of which we are only interested in a few tuples.

  customer $\bowtie$ deposit $\bowtie$ branch
- We also are only interested in two attributes of this relation.
- We can see that we only want tuples for which ccity = ``Port Chester".
- Thus we can rewrite our query as:

$\Pi_{bname,assets}((\sigma_{ccity="Port\ Chester"}(customer))$
$\bowtie deposit \bowtie branch)$
- This should considerably reduce the size of the intermediate relation.

Projection Operation

1. Like selection, projection reduces the size of relations. It is advantageous to **apply projections early**. Consider this form of our example query:

   $\Pi_{bname,assets}$
   $(((\sigma_{ccity="Port\ Chester"}(customer))$
   $\bowtie deposit) \bowtie branch)$

2. When we compute the subexpression

   $((\sigma_{ccity="Port\ Chester"}(customer)) \bowtie deposit)$

   we obtain a relation whose scheme is
   (*cname, ccity, bname, account#, balance*)

3. We can eliminate several attributes from this scheme. The only ones we need to retain are those that
   - appear in the result of the query **or**
   - are needed to process subsequent operations.
4. By eliminating unneeded attributes, we reduce the number of columns of the intermediate result, and thus its size.
5. In our example, the only attribute we need is *bname* (to join with *branch*). So we can rewrite our expression as:

   $\Pi_{bname,assets}$
   $((\Pi_{bname}((\sigma_{ccity="Port\ Chester"}(customer))$
   $\bowtie deposit)) \bowtie branch)$

6. Note that there is no advantage in doing an early project on a relation before it is needed for some other operation:
   - We would access every block for the relation to remove attributes.

- Then we access every block of the reduced-size relation when it is actually needed.
- We do more work in total, rather than less!

11.4Algorithm for executing query operations

The function of the optimizer is to translate certain SQL statements (SELECT, INSERT, UPDATE, or DELETE) into an efficient access plan made up of various relational algebra operators (join, duplicate elimination, union, and so on). The operators within the access plan may not be structurally equivalent to the original SQL statement, but the access plan's various operators will compute a result that is semantically equivalent to that SQL request.

Relational algebra operators in access plans

An access plan consists of a tree of relational algebra operators which, starting at the leaves of the tree, consume the base inputs to the query (usually rows from a table) and process the rows from bottom to top, so that the root of the tree yields the final result. Access plans can be viewed graphically for ease of comprehension. See Reading execution plans, and Reading graphical plans.

SQL Anywhere supports multiple implementations of these various relational algebra operations. For example, SQL Anywhere supports three different implementations of inner join: nested loops join, merge join, and hash join. Each of these operators can be advantageous to use in specific circumstances: some of the parameters that the query optimizer analyzes to make its choice include the amount of table data in cache, the characteristics and selectivity of the join predicate, the sortedness of the inputs to the join and the output from it, the amount of memory available to perform the join, and a variety of other factors.

SQL Anywhere may dynamically, at execution time, switch from the physical algebraic operator chosen by the optimizer to a different physical algorithm that is logically equivalent to the original. Typically, this alternative access plan is used in one of two circumstances:

- When the total amount memory used to execute the statement is close to a memory governor threshold, then a switch is made to a strategy that may execute more slowly, but that frees a substantial amount of memory for use by other operators (or other requests). When this occurs, the QueryLowMemoryStrategy property is incremented. This information also appears in the graphical plan for the statement.

For information about the QueryLowMemoryStrategy property, see Connection properties. The amount of memory that can be used by an operator dependent upon the multiprogramming level of the server, and the number of active connections.

For more information about how the memory governor and the multiprogramming level, see:

- Threading in SQL Anywhere
- Configuring the database server's multiprogramming level
- The memory governor

➢ If, at the beginning of its execution, the specific operator (a hash inner join, for example) determines that its inputs are not of the expected cardinality as that computed by the optimizer at optimization time. In this case, the operator may switch to a different strategy that will be less expensive to execute. Typically, this alternative strategy utilizes index nested loops processing. For the case of hash join, the QueryJHToJNLOptUsed property is incremented when this switch occurs. The occurrence of the join method switch is also included in the statement's graphical plan. For information about the QueryJHToJNLOptUsed property, see Connection properties.

Parallelism during query execution

➢ SQL Anywhere supports two different kinds of parallelism for query execution: inter-query, and intra-query. Inter-query parallelism involves executing different requests simultaneously on separate CPUs. Each request (task) runs on a single thread and executes on a single processor.

➢ Intra-query parallelism involves having more than one CPU handle a single request simultaneously, so that portions of the query are computed in parallel on multi-processor hardware. Processing of these portions is handled by the Exchange algorithm. See Exchange algorithm (Exchange).

➢ Intra-query parallelism can benefit a workload where the number of simultaneously-executing queries is usually less than the number of available processors. The maximum degree of parallelism is controlled by the setting of the max_query_tasks option. See max_query_tasks option.

- ➢ The optimizer estimates the extra cost of parallelism (extra copying of rows, extra costs for co-ordination of effort) and chooses parallel plans only if they are expected to improve performance.
- ➢ Intra-query parallelism is not used for connections with the priority option set to background. See priority option.
- ➢ Intra-query parallelism is not used if the number of server threads that are currently handling a request (ActiveReq server property) recently exceeded the number of CPU cores on the computer that the database server is licensed to use. The exact period of time is decided by the server and is normally a few seconds. See Database server properties.

Parallel execution

Whether a query can take advantage of parallel execution depends on a variety of factors:

- The available resources in the system at the time of optimization (such as memory, amount of data in cache, and so on)
- The number of logical processors on the computer
- The number of disk devices used for the storage of the database, and their speed relative to that of the processor and the computer's I/O architecture.
- The specific algebraic operators required by the request. SQL Anywhere supports five algebraic operators that can execute in parallel:

- ✓ parallel sequential scan (table scan)
- ✓ parallel index scan
- ✓ parallel hash join, and parallel versions of hash semijoin and anti-semijoin
- ✓ parallel nested loop joins, and parallel versions of nested loop semijoin and anti-semijoin
- ✓ parallel hash filter
- ✓ parallel hash group by

A query that uses unsupported operators can still execute in parallel, but the supported operators must appear below the unsupported ones in the plan (as viewed in dbisql). A query where most of the unsupported operators can appear near the top is more likely to use parallelism. For example, a sort operator cannot be parallelized but a query that uses an ORDER BY on the outermost block may be parallelized by positioning the sort at the top of the plan and all the parallel operators below it. In contrast, a query that uses a TOP n and ORDER BY in a derived table is less likely to use parallelism since the sort must appear somewhere other than the top of the plan.

By default, SQL Anywhere assumes that any dbspace resides on a disk subsystem with a single platter. While there can be advantages to parallel query execution in such an environment, the optimizer's I/O cost model for a single device makes it difficult for the optimizer to choose a parallel table or index scan unless the table data is fully resident in the cache. However, by calibrating the I/O subsystem using the ALTER DATABASE CALIBRATE PARALLEL READ statement, the optimizer can then cost with greater accuracy the benefits of parallel execution. The optimizer is much more likely to choose execution plans with some degree of parallelism for multiple spindles.

When intra-query parallelism is used for an access plan, the plan contains an Exchange operator whose effect is to merge (union) the results of the parallel computation of each subtree. The number of subtrees underneath the Exchange operator is the degree of parallelism. Each subtree, or access plan component, is a database server task. The database server kernel schedules these tasks for execution in the same manner as if they were individual SQL requests, based on the availability of execution threads (or fibers). This architecture means that parallel computation of any access plan is largely self-tuning, in that work for a parallel execution task is scheduled on a thread (fiber) as the server kernel allows, and execution of the plan components is performed evenly.

11.5Heuristics of Query Optimization

The heuristic optimization process uses a set of rules that have been defined to guarantee good execution plans. Thus, the effectiveness of a heuristic optimizer to produce good plans is based solely on the effectiveness and completeness of its rules. The following paragraphs describe the rules used to create the DBXP query optimizer. Although these rules are very basic, when applied to typical queries the resulting execution is near optimal with fast performance and accurate results.

Some basic strategies were used to construct the query tree initially. Specifically, all executions take place in the query tree node. Restrictions and projections are processed on a branch and do not generate intermediate relations. Joins are always processed as an intersection of two paths. A multiway join would be formed using a series of two-way joins. The following rules represent the best practices for forming a set of heuristics to generate good execution plans. The DBXP optimizer is designed to apply these rules in order to transform the query tree into a form that ensures efficient execution.

11.6Catalog information for cost estimation of queries

Catalog Information for Cost Estimation

Information about relations and attributes:

NR: number of tuples in the relation R.

BR: number of blocks that contain tuples of the relation R. SR: size of a tuple of R.

FR: blocking factor; number of tuples from R that t into one block (FR = dNR=BRe)

V(A; R): number of distinct values for attribute A in R. SC(A; R): selectivity of attribute A average number of tuples of R that satisfy an equality condition on A.
SC(A; R) = NR=V(A; R).
Information about indexes:

HTI: number of levels in index I (B+-tree).
LBI: number of blocks occupied by leaf nodes in index I (rst-level blocks).
ValI: number of distinct values for the search key.

Some relevant tables in the Oracle system catalogs:

USER TABLES	USER TAB COLUMNS	USER INDEXES
NUM ROWS	NUM DISTINCT	BLEVEL
BLOCKS	LOW VALUE	LEAF BLOCKS
EMPTY BLOCKS	HIGH VALUE	DISTINCT KEYS
AVG SPACE	DENSITY	AVG LEAF BLOCKS PER KEY
CHAIN CNT	NUM BUCKETS	

AVG ROW LEN LAST ANALYZED

Measures of Query Cost

There are many possible ways to estimate cost, e.g., based on disk accesses, CPU time, or communication overhead.

Disk access is the predominant cost (in terms of time); relatively easy to estimate; therefore, number of block transfers from/to disk is typically used as measure.

{ Simplifying assumption: each block transfer has the same cost.

Cost of algorithm (e.g., for join or selection) depends on database buer size; more memory for DB buer reduces disk accesses. Thus DB buer size is a parameter for estimating cost.

We refer to the cost estimate of algorithm S as cost(S). We do not consider cost of writing output to disk.

11.7Algebraic manipulation

Relational Algebra comprises a set of basic operations. An operation is the application of an operator to one or more source (or input) relations to produce a new relation as a result. This is illustrated in Figure 8.1 below. More abstractly, we can think of such an operation as a function that maps arguments from specified domains to a result in a specified range. In this case, the domain and range happen to be the same, i.e. relations.

Relational Algebra is a procedural language. It specifies the operations to be reformed on. Existing relations in derived result relations. Therefore, it defines the complete schema for each of the result relations. The relational algebraic operations can be divided into basic set-oriented operations and relational-oriented operations. The former are the traditional set operations, the latter, those for performing joins. Selection, projection, and division.

Relational Algebra is a collection of operations to manipulate relations. Each operation takes one or more relations as its operands and produce another relation as its results. Some mainly used operations are join, selection and projection. Relational algebra is a procedural language. It specifies the operations to be performed on existing relations to derive relations. The relational algebra operations can be divided into basic set oriented operations and relational oriented operations. The former are the traditional set operations, the latter are joins, selections, projection and division.

Basic Operations

Basic operations are the traditional set operations: union, difference, intersection and cartesian product. Three of these four basic operations - union, intersection, and difference require that operand relations be union compatible. Two relations are union compatible if they have the same parity and one-to-one correspondence of the attributes with the corresponding attributes defined over the same domain. The cartesian product can be defined on any two relations. Two relations P and Q are said to be union compatible if both P and Q are of the same degree n and the domain of the correspondingn attributes are identical,

i.e. if P= P [P1,……. Pn] and Q = [Q1, ...Qn] then

Dom (Pi) = Dom (Qi) for i = (1,2,n)

Where Dom (Pi) represents the domain of the attribute Pi. Some basic operations used in Relational Algebra are:

Traditional Set Operations: Further traditional set operations are subdivided as:

(a) UNION

(b) INTERSECTION

(c) DIFFERENCE

(d) CARTESIAN PRODUCT

Relational Set Operators: Similarly further Relational Set Operations subdivided as:

(a) PROJECTION

(b) SELECTION

(c) JOIN

(d) DIVISION

Traditional Set Operations

(i) UNION (U): The union of two relations A and B is done by UNION command as:

A UNION B

It is the set of all types belonging to either A or B or both. Let us consider set A and B as:

Table A:

Rn	Name
101	Amrit
102	Amar
103	Vickey
104	Raj
110	Joyti
112	Banny

Table B:

Rn	Name
103	**Vickey**
106	Jimmy
107	Ruchi
110	Joyti
104	Raj

If P and Q are two sets, then R is the resultant set by union operations. The R is resented by:

$$|R| = |P| + |Q|$$

For example, let us consider A be the set of suppliers tuples for suppliers in London and B is the set of supplier who supply part P1. Then A UNION B is the set of supplier samples for suppliers who are either located in London city or supply part P1 (or both). It is denoted by the symbol U (union). We can combine it as:

A U B (in mathematical form.)

R: A ∪ B

Rn	Name
101	Amrit
103	Vickey
102	Amar
104	Raj
106	Jimmy
107	Ruchi
110	Joyti
112	Banny

(ii) Intersection (∩): The intersection operation selects the common tuples from the two relations. It is denoted by the symbol ∩. The intersection of two relations and B is defined as:

or A INTERSECT B

For example, if A and B are two sets, then intersection between these two are in R1.

R : A ∩ B

Rn	Name
103	Vickey
104	Raj
110	Joyti

(iii) Difference (—): The difference operation removes common tuples from t first relation. The difference between two relations A and B be defined as:

A MINUS B

It is the set of all tuples belonging to set A but not belonging to B. It is denoted by (-). We can represent it as A - B. For example, from the above said two A and B sets, the difference between A and B be represented as:

R : A – B

Rn	Name
101	Amrit
102	Ruchi
112	Banny

(iv) Cartesian Product: It is denoted by 'X' or 'x' (Cross). The cartesian product of two relations A and B is defined as:

A TIMES B

Or A x B

The extended cartesian or simply the cartesian product of two relations is the concatenation of tuples belonging to the two relations. A new resultant relation schema is created consisting of all possible combinations of the tuples is represented as:

R = P x Q

For example, let us consider A be the set of all supplier number and B is set of all part number. Then A TIMES B is the set of all possible supplier number / part number pairs as:

Table A:

S#	Sname
S1	Mona
S2	Nidhi
S3	Vikas

Table B:

S#	P#
S1	P1
S1	P2
S2	P3
S2	P4

R: A x B

S#	Sname	P#
S1	Mona	P1
S1	Mona	P2
S2	Nidhi	P3
S2	Nidhi	P4

Relational Set Operations:

The basic set operations, which provide a very limited data manipulating facility have been supplemented by the definition of the following operations:

(i) Projection (x): The projection of a relation is defined as the projection of all its tuples over some set of attributes i.e. it yields a vertical subset of a relation. The projection operation is used to either reduce the number of attributes in the resultant or the reorder attributes. For example, if P is the table, then we can project on the field name as and get resultant projected table:

R = π P (Names)

Rn	Name
101	Mona
103	Nidhi
104	Vikas
106	Geeta
107	Raju

$\xrightarrow{\pi \text{ (names)}}$

Name
Mona
Nidhi
Vikas
Geeta
Raju

(ii) Selection (σ): Selection is the selection of some tuples based on some condition. It is horizontal subset of relation. It is denoted by s. It reduces the number of tuples. from a relation, e.g. if P is the relation then R is the resultant table after selection on P. Condition is to select all tuples having roll no. < 105.

R = s P (RN < 105)

Rn	Name
101	Mona
103	Nidhi
104	Vikas

(iii) Join (S or $\bowtie$) The join operator allows the combining of two relations to form a single new relations. These are of three types:

(i) Theta Join

(ii) Natural Join

(iii) Equi Join

Theta Join is the joining of two tables on the basis of a condition. Natural Join is the joining of two tables without any condition and equality. Equi Join is the joining of two tables of both having common equal key field. For example, if S and P are two tables and these are joined on CITY field as S.CITY and P.CITY.

(iv) Division (+): The division operator divides a dividend relation A of degree m + n by a divider relation B of degree n. It will produce a result relation of degree m. Suppose A is relational table of supplier having supplier number and B is the relational tables of different types of parts, then A DIVIDE BY B gives the resultant table R.

11.8Shadow paging

This recovery scheme does not require the use of a log in a single-user environment. In a multiuser environment, a log may be needed for the concurrency control method. Shadow paging considers the database to be made up of a number of fixed-size disk pages (or disk blocks)—say, n—for recovery purposes. A directory with n entries is constructed, where the i th entry points to the i th database page on disk. The directory is kept in main memory if it is not too large, and all references—reads or writes—to database pages on disk go through it. When a transaction begins executing, the current directory—whose entries point to the most recent or current database pages on disk—is copied into a shadow directory. The shadow directory is then saved on disk while the current directory is used by the transaction.

During transaction execution, the shadow directory is never modified. When a write_item operation is performed, a new copy of the modified database page is created, but the old copy of that page is not overwritten. Instead, the new page is written elsewhere—on some previously unused disk block. The current directory entry is modified to point to the new disk block, whereas the shadow directory is not modified and continues to point to the old unmodified disk block. For pages updated by the transaction, two versions are kept. The old version is referenced by the shadow directory, and the new version by the current directory.

To recover from a failure during transaction execution, it is sufficient to free the modified database pages and to discard the current directory. The state of the database before transaction execution is available through the shadow directory, and that state is recovered by reinstating the shadow directory. The database thus is returned to its state prior to the transaction that was executing when the crash occurred, and any modified pages are discarded. Committing a transaction corresponds to discarding the previous shadow directory. Since recovery involves neither undoing nor redoing data items, this technique can be categorized as a NO-UNDO/NO-REDO technique for recovery.

In a multiuser environment with concurrent transactions, logs and checkpoints must be incorporated into the shadow paging technique. One disadvantage of shadow paging is that the updated database pages change location on disk. This makes it difficult to keep related database pages close together on disk without complex storage management strategies. Furthermore, if the directory is large, the overhead of writing shadow directories to disk as transactions commit is significant. A further complication is how to handle garbage collection when a transaction commits. The old pages referenced by the shadow directory that have been updated must be released and added to a list of free pages for future use. These pages are no longer needed after the transaction commits. Another issue is that the operation to migrate between current and shadow directories must be implemented as an atomic operation.

11.9Buffer management

In this section, we consider several subtle details that are essential to the implementation of a crash-recovery scheme that ensures data consistency and imposes aminimalamount of overhead on interactions with the database.

Log-Record Buffering

So far, we have assumed that every log record is output to stable storage at the time it is created. This assumption imposes a high overhead on system execution for severalreasons: Typically, output to stable storage is in units of blocks. In most cases, a log record is much smaller than a block. Thus, the output of each log record translates toa much larger output at the physical level. The output of a block to stable storage may involve several output operations at thephysical level.

The cost of performing the output of a block to stable storage is sufficiently high that it is desirable to output multiple log records at once. To do so, we write logrecords to a log buffer in main memory, where they stay temporarily until they are output to stable storage. Multiple log records can be gathered in the log buffer, andoutput to stable storage in a single output operation. The order of log records in the stable storage must be exactly the same as the order in which they were written tothe log buffer.

As a result of log buffering, a log record may reside in only main memory (volatile storage) for a considerable time before it is output to stable storage. Since such logrecords are lost if the system crashes, we must impose additional requirements on the recovery techniques to ensure transaction atomicity:

• Transaction Ti enters the commit state after the <Ti commit> log record has been output to stable storage.
• Before the <Ti commit> log record can be output to stable storage, all log records pertaining to transaction Ti must have been output to stable storage.
• Before a block of data in main memory can be output to the database (in nonvolatile storage), all log records pertaining to data in that block must havebeen output to stable storage.

This rule is called the write-ahead logging (WAL) rule. (Strictly speaking,the WAL rule requires only that the undo information in the log have been output to stable storage, and permits the redo information to be written later.The difference is relevant in systems where undo information and redo information are stored in separate log records.)

The three rules state situations in which certain log records must have been output to stable storage. There is no problem resulting from the output of log records earlierthan necessary. Thus, when the system finds it necessary to output a log record to stable storage, it outputs an entire block of log records, if there are enough log recordsin main memory to fill a block. If there are insufficient log records to fill the block, all log records in main memory are combined into a partially full block, and are outputto stable storage. Writing the buffered log to disk is sometimes referred to as a log force.

11.10 Summary

The most complex database internal technology an optimizer. You learned how to expand the concept of the query tree to incorporate a query optimizer that uses the tree structure in the optimization process. More importantly, you discovered how to construct a heuristic query optimizer. The knowledge of heuristic optimizers should provide you with a greater understanding of the DBXP engine and how it can be used to study database technologies in more depth. It doesn't get any deeper than an optimizer. In this, chapter will complete the DBXP engine by linking the heuristic query optimizer using the query tree class to an execution process that surprise also uses the query tree structure.

11.11Keywords

Query interpretation, Equivalence of expressions, Query Optimization, Query operations

11.12Exercises

1. Explain various query optimization
2. Explain Query interpretation process
3. Discuss different types of query operations.

11.13Reference

1. Fundamentals of Database Systems By RamezElmasri, Shamkant B. Navathe, Durvasula V.L.N. Somayajulu, ShyamK.Gupta
2. Database System Concepts By AviSilberschatz, Henry F. Korth , S. Sudarshan
3. Database Management Systems By Raghu Ramakrishnan and Johannes Gehrke
4. Database Management Systems Alexis Leon, Mathews, Vikas Publishing House

Unit: 12 Evolution of an Information system

Structure

12.0 Objectives

After going through this unit, you will be able toexplain :Advanced Database Applications

- Describe Navigation Database System
- Discuss MIS as a technique for making programmed decisions.
- Define different Types of transaction processing system.

12.1Introduction

Many advanced computer-based applications, such as computer-aided design and manufacturing (CAD/CAM), network management, financial instruments trading, medical informatics,office automation, and software development environments (SDEs), are data-intensive in the sense that they generate and manipulate large amounts of data (e.g., all the software artifacts inan SDE). It is desirable to base these kinds of application systems on data management capabilities similar to those provided by database management systems (DBMSs) for traditionaldata processing. These capabilities include adding, removing, retrieving and updating data from on-line storage, and maintaining the consistency of the information stored in a database. Consistencyin a DBMS is maintained if every data item satisfies specific consistency constraints, which are typically implicit in data processing, although known to the implementors of the applications,and programmed into atomic units called transactions that transform the database from one consistent state to another. Consistency can be violated by concurrent access by multipletransactions to the same data item. A DBMS solves this problem by enforcing a concurrency control policy that allows only consistency-preserving schedules of concurrent transactionsto be executed.

We use the term advanced database applications to describe application systems, such as the ones mentioned above, that utilize DBMS capabilities. They are termed advanced to distinguish
them from traditional database applications, such as banking and airline reservations systems, in which the nature of the data and the operations performed on the data are amenable toconcurrency control mechanisms that enforce the classical transaction model. Advanced applications, in contrast, place different kinds of consistency constraints, and, in general, the classicaltransaction model is not applicable. For example, network management, financial instruments trading and medical informatics may require real-time processing, while CAD/CAM, officeautomation and SDEs involve long interactive database sessions and cooperation among multiple database users. Conventional concurrency control mechanisms appropriate for traditionalapplications are not applicable "as is" in these new domains.

12.2Evolution of an Information system

Data processing has provided voluminous printouts of internal data, such as accounting and financial data, production schedules, inventories of raw materials, work in process and finished goods sales, by territory, products and customers; and external, environmental data, such as marketing data of consumer demand census of population, economic forecasts, population shifts, and reports of social condition. With this explosion of data came the realization that mangers might not need more data but what they did need was relevant timely data.

Information theory provided the needed breakthrough by pointing out that data and information are not synonymous. Data are facts, statistics, opinions, or predictions classified on some basis for storage, processing, and retrieval. These data may be printed out as messages, but all data or messages may not be relevant to managers in the performance of their functions. Information, on the other hand, is data that are relevant to the needs of managers in performing their functions. A computer can store technically vast amounts of data â€“ more than managers can comprehend or use. The semantic aspect provides meaning to the data so that managers can concentrate on the portion that will make them more effective.

Messages that shed no new light on subjects of interest to the manager may be printed and transmitted. Some messages contain so much interference that they jam the reception of pertinent information. In short, managers may be hindered by excess data and frustrated by meaningless messages, much as consumers are frustrated by junk mail. The more probable a message is, the less information it gives. Early uses of computers in data processing were primarily in handling large masses of routine activities but they did not enable managers to distinguish relevant information from volumes of data with little meaning or of little interest to the receiver.

Computer technology has developed greater speeds, storage capacity, and improved methods of printing outputs so that managers are often inundated with data. But they needed better information, not more data. Furthermore, managers often installed computers merely because others were purchasing or renting computers and because it was the â€œinthingâ€• to do. In short they obtained equipment before they determined their needs.

This realistic discussion of the evolution of information system demonstrates the result of the rapid improvements in technology ahead and of the ability of human beings to make efficient and effective use of the improvements It also serves to caution prospective managers that the expected technological improvements in the next decade might result in a repetition of past mistakes.With a clear understanding of their needs for information, managers can then design for handling information specifically to aid them in performing their managerial functions. In summary, in order for managers to manage information they (1) must identify their specific and explicit needs, (2) design a system that will satisfy these needs, and (3) only then select the computer hardware to efficiently implement the system.

12.3Decision making and MIS

Todays managers depend on information systems for decision making. The managers have handful of data around them but manually they cannot process the data accurately and with in the short period of time available to them due to heavy competition in modern world. Therefore mangers depend on information systems.

The concept of MIS:

Management: Management has been defined in a variety of ways, but for our purposes it comprises the process or activities what managers do in the operation of their organization: Plan, Organize, Initiate and Control operations.

Information:

Data are facts and figures that are not currently being used in a decision processes and usually take the form of historical records that are recorded and filed without immediate intent to retrieve for decision making

Information consists of data that have been retrieved, processed or otherwise used for information or inference purposes, argument, or as a basis for forecasting or decision making.

System can be described simply as a set of elements joined together for a common objective. A subsystem is is part of a larger system with which we are concerned. All systems are part of largersystems.

The objective of an MIS (Management Information System) is to provide information for decision making on planning, initiating, organizing, and controlling the operations of the subsystems of the form and to provide a synergetic organization in the process.

Decision Support System: It is sometimes described as the next evolutionary step after *Management Information Systems (MIS)* . MIS support decision making in both structured and unstructured problem environments.. It supports decision making at all levels of the organization .IS (Information Systems) are intended to be woven into the fabric of the organizations , not standing alone. IS support all aspects of the decision making process.MIS are made of people, computers, procedures, databases, interactive query facilities and so on.

They are intended to be evolutionary/adaptive and easy for people to use.

Methods of Decision Making

Type of Decision	Methods of decision making	
	OLD	NEW
Programmed Repetitive and Routine	Habit Standard operating procedure Organization structure, policy etc	Management Information System
Non-Programmed	Judgement, Intution, Insight experience Training and Learning	Systematic Approach to problem solving & Decision making

MIS is a technique for making programmed decisions. If we include the computer and management science as integral parts or tools of computer –based information systems, the prospects for a revolution in programmed decision making are very real. Just as a manufacturing process is becoming more and more automated so is the automation of programmed decisions increasing to support this production and other information needs through out the organization.

12.4 Navigation Database System

A navigational database is a type of database in which records or objects are found primarily by following references from other objects. Navigational interfaces are usually procedural, though some modern systems like XPath can be considered to be simultaneously navigational and declarative.

Navigational access is traditionally associated with the network model and hierarchical model of database interfaces, and some have even acquired set-oriented features.[1] Navigational techniques use "pointers" and "paths" to navigate among data records (also known as "nodes"). This is in contrast to the relational model (implemented in relational databases), which strives to use "declarative" or logic programming techniques in which you ask the system for what you want instead of how to navigate to it.

For example, to give directions to a house, the navigational approach would resemble something like "Get on highway 25 for 8 miles, turn onto Horse Road, left at the red barn, then stop at the 3rd house down the road", whereas the declarative approach would resemble "Visit the green house(s) within the following coordinates...."

Hierarchical models are also considered navigational because one "goes" up (to parent), down (to leaves), and there are "paths", such as the familiar file/folder paths in hierarchical file systems. In general, navigational systems will use combinations of paths and prepositions such as "next", "previous", "first", "last", "up", "down", "owner", etc.

12.5 Data Mining and Data warehouse

Owing to the widespread use of databases and the excessive growth in their volumes, organizations are facing the problem of information overloading. Simply storing information in a data warehouse does not provide the benefits that an organization is seeking. To realize the value of a data warehouse, it is necessary to extract the knowledge hidden within the warehouse. Data mining is the non-trivial process of identifying valid, novel, potentially useful and ultimately understandable pattern in data. Data mining is concerned with the analysis of data and the use of software techniques for searching hidden and unexpected patterns and relationships in sets of data. There is increasing desire to use this new technology in the application domain, and a growing perception is that these large passive databases can be made into useful actionable information. There are numerous definitions for data mining. One such focused definition is presented in the following.

Data mining is the process of extracting valid, previously unknown, comprehensible and actionable information from large databases and using it to make crucial business decisions.

Overview

Generally, data mining (sometimes called data or knowledge discovery) is the process of analyzing data from different perspectives and summarizing it into useful information - information that can be used to increase revenue, cuts costs, or both. Data mining software is one of a number of analytical tools for analyzing data. It allows users to analyze data from many different dimensions or angles, categorize it, and summarize the relationships identified. Technically, data mining is the process of finding correlations or patterns among dozens of fields in large relational databases.

Continuous Innovation

Although data mining is a relatively new term, the technology is not. Companies have used powerful computers to sift through volumes of supermarket scanner data and analyze market research reports for years. However, continuous innovations in computer processing power, disk storage, and statistical software are dramatically increasing the accuracy of analysis while driving down the cost.

Example

For example, one Midwest grocery chain used the data mining capacity of Oracle software to analyze local buying patterns. They discovered that when men bought diapers on Thursdays and Saturdays, they also tended to buy beer. Further analysis showed that these shoppers typically did their weekly grocery shopping on Saturdays. On Thursdays, however, they only bought a few items. The retailer concluded that they purchased the beer to have it available for the upcoming weekend. The grocery chain could use this newly discovered information in various ways to increase revenue. For example, they could move the beer display closer to the diaper display. And, they could make sure beer and diapers were sold at full price on Thursdays.

Data, Information, and Knowledge

Data

Data are any facts, numbers, or text that can be processed by a computer. Today, organizations are accumulating vast and growing amounts of data in different formats and different databases. This includes:

- operational or transactional data such as, sales, cost, inventory, payroll, and accounting
- nonoperational data, such as industry sales, forecast data, and macro economic data
- meta data - data about the data itself, such as logical database design or data dictionary definitions

Information

The patterns, associations, or relationships among all this data can provide information. For example, analysis of retail point of sale transaction data can yield information on which products are selling and when.

Knowledge

Information can be converted into knowledge about historical patterns and future trends. For example, summary information on retail supermarket sales can be analyzed in light of promotional efforts to provide knowledge of consumer buying behavior. Thus, a manufacturer or retailer could determine which items are most susceptible to promotional efforts.

Data Warehouses

Dramatic advances in data capture, processing power, data transmission, and storage capabilities are enabling organizations to integrate their various databases into data warehouses. Data warehousing is defined as a process of centralized data management and retrieval. Data warehousing, like data mining, is a relatively new term although the concept itself has been around for years. Data warehousing represents an ideal vision of maintaining a central repository of all organizational data. Centralization of data is needed to maximize user access and analysis. Dramatic technological advances are making this vision a reality for many companies. And, equally dramatic advances in data analysis software are allowing users to access this data freely. The data analysis software is what supports data mining.

What can data mining do?

Data mining is primarily used today by companies with a strong consumer focus - retail, financial, communication, and marketing organizations. It enables these companies to determine relationships among "internal" factors such as price, product positioning, or staff skills, and "external" factors such as economic indicators, competition, and customer demographics. And, it enables them to determine the impact on sales, customer satisfaction, and corporate profits. Finally, it enables them to "drill down" into summary information to view detail transactional data.

With data mining, a retailer could use point-of-sale records of customer purchases to send targeted promotions based on an individual's purchase history. By mining demographic data from comment or warranty cards, the retailer could develop products and promotions to appeal to specific customer segments.

For example, Blockbuster Entertainment mines its video rental history database to recommend rentals to individual customers. American Express can suggest products to its cardholders based on analysis of their monthly expenditures.

WalMart is pioneering massive data mining to transform its supplier relationships. WalMartcaptures point-of-sale transactions from over 2,900 stores in 6 countries and continuously transmits this data to its massive 7.5 terabyte Teradata data warehouse. WalMart allows more than 3,500 suppliers, to access data on their products and perform data analyses. These suppliers use this data to identify customer buying patterns at the store display level. They use this information to manage local store inventory and identify new merchandising opportunities. In 1995, WalMart computers processed over 1 million complex data queries.

The National Basketball Association (NBA) is exploring a data mining application that can be used in conjunction with image recordings of basketball games. The Advanced Scout software analyzes the movements of players to help coaches orchestrate plays and strategies. For example, an analysis of the play-by-play sheet of the game played between the New York Knicks and the Cleveland Cavaliers on January 6, 1995 reveals that when Mark Price played the Guard position, John Williams attempted four jump shots and made each one! Advanced Scout not only finds this pattern, but explains that it is interesting because it differs considerably from the average shooting percentage of 49.30% for the Cavaliers during that game.

By using the NBA universal clock, a coach can automatically bring up the video clips showing each of the jump shots attempted by Williams with Price on the floor, without needing to comb through hours of video footage. Those clips show a very successful pick-and-roll play in which Price draws the Knick's defense and then finds Williams for an open jump shot.

How does data mining work?

While large-scale information technology has been evolving separate transaction and analytical systems, data mining provides the link between the two. Data mining software analyzes relationships and patterns in stored transaction data based on open-ended user queries. Several types of analytical software are available: statistical, machine learning, and neural networks. Generally, any of four types of relationships are sought:

Classes: Stored data is used to locate data in predetermined groups. For example, a restaurant chain could mine customer purchase data to determine when customers visit and what they typically order. This information could be used to increase traffic by having daily specials.

Clusters: Data items are grouped according to logical relationships or consumer preferences. For example, data can be mined to identify market segments or consumer affinities.

Associations: Data can be mined to identify associations. The beer-diaper example is an example of associative mining.

Sequential patterns: Data is mined to anticipate behavior patterns and trends. For example, an outdoor equipment retailer could predict the likelihood of a backpack being purchased based on a consumer's purchase of sleeping bags and hiking shoes.

Data mining consists of five major elements:

- Extract, transform, and load transaction data onto the data warehouse system.
- Store and manage the data in a multidimensional database system.
- Provide data access to business analysts and information technology professionals.
- Analyze the data by application software.
- Present the data in a useful format, such as a graph or table.

Different levels of analysis are available:

Artificial neural networks: Non-linear predictive models that learn through training and resemble biological neural networks in structure.

Genetic algorithms: Optimization techniques that use processes such as genetic combination, mutation, and natural selection in a design based on the concepts of natural evolution.

Decision trees: Tree-shaped structures that represent sets of decisions. These decisions generate rules for the classification of a dataset. Specific decision tree methods include Classification and Regression Trees (CART) and Chi Square Automatic Interaction Detection (CHAID) . CART and CHAID are decision tree techniques used for classification of a dataset. They provide a set of rules that you can apply to a new (unclassified) dataset to predict which records will have a given outcome. CART segments a dataset by creating 2-way splits while CHAID segments using chi square tests to create multi-way splits. CART typically requires less data preparation than CHAID.

Nearest neighbor method: A technique that classifies each record in a dataset based on a combination of the classes of the k record(s) most similar to it in a historical dataset (where k 1). Sometimes called the k-nearest neighbor technique.

Rule induction: The extraction of useful if-then rules from data based on statistical significance.

Data visualization: The visual interpretation of complex relationships in multidimensional data. Graphics tools are used to illustrate data relationships.

What technological infrastructure is required?

Today, data mining applications are available on all size systems for mainframe, client/server, and PC platforms. System prices range from several thousand dollars for the smallest applications up to $1 million a terabyte for the largest. Enterprise-wide applications generally range in size from 10 gigabytes to over 11 terabytes. NCR has the capacity to deliver applications exceeding 100 terabytes. There are two critical technological drivers:

Size of the database: the more data being processed and maintained, the more powerful the system required.

Query complexity: the more complex the queries and the greater the number of queries being processed, the more powerful the system required.

Relational database storage and management technology is adequate for many data mining applications less than 50 gigabytes. However, this infrastructure needs to be significantly enhanced to support larger applications. Some vendors have added extensive indexing capabilities to improve query performance. Others use new hardware architectures such as Massively Parallel Processors (MPP) to achieve order-of-magnitude improvements in query time. For example, MPP systems from NCR link hundreds of high-speed Pentium processors to achieve performance levels exceeding those of the largest supercomputers.

12.6Data Mining Techniques

The fundamental objectives of data mining that are identified by researchers are prediction and description. To predict unknown or future values of interest, prediction makes use of existing variables in the database. Description focuses on finding patterns describing the data and the subsequent presentation for user interpretation. The relative emphasis of both prediction and description differ with respect to the underlying application and the technique used. To fulfill the above-mentioned objectives, several data mining techniques have been developed. These are predictive modelling, database segmentation or clustering, link analysis and deviation detection.

Predictive Modelling

Predictive modelling is similar to the human learning experience in using observations to form a model of the important characteristics of some phenomenon. Predictive modelling is used to analyse an existing database to determine some essential features (model) about the data set. The model is developed using a supervised learning approach that has two different phases. These are training and testing. In the training phase, a model is developed using a large sample of historical data, known as the training set. In the testing phase, the model is tested on new, previously unseen, data to determine its accuracy and physical performance characteristics. Two methods are used in predictive modelling. These are classification and value prediction.

12.7 Summary

- A data warehouse is a subject-oriented, time-variant, integrated, non-volatile repository of information for strategic decision-making.
- A data web house is a distributed data warehouse that is implemented on the Web with no central data repository.
- The components of a data warehouse are operational data source, load manager, query manager, warehouse manager, detailed data, summarized data, archive/backup data, metadata, end-user access tools and data warehouse background processes.

- The three different types of database schemas that are used to represent data in warehouses are star schema, snowflake schema and fact constellation schema.
- A subset of a data warehouse that supports the requirements of a particular department or business function is called a data mart.
- OLAP is the dynamic synthesis, analysis and consolidation of large volumes of multi-dimensional data. There are three different categories of OLAP tools, which are MOLAP tools, ROLAP tools and HOLAP tools.
- Data mining is the process of extracting valid, previously unknown, comprehensible and actionable information from large volumes of multi-dimensional data and it is used to make crucial business decisions. The different data mining techniques are predictive modelling, database segmentation or clustering, link analysis and deviation detection.

12.8Keywords

Data Mining, Data warehouse, Transaction processing, Predictive Modelling

12.9Exercises

1. Define Data mining and its applications
2. Explain Navigation Database System
3. Explain different data warehouse techniques
4. Discuss Decision making and MIS

12.10Reference

1. Fundamentals of Database Systems By RamezElmasri, Shamkant B. Navathe, Durvasula V.L.N. Somayajulu, ShyamK.Gupta
2. Database System Concepts By AviSilberschatz, Henry F. Korth , S. Sudarshan
3. Database Management Systems By Raghu Ramakrishnan and Johannes Gehrke

Unit: 13 Distributed Databases

Structure

13.0 Objectives

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

- Describe Basic Distributed Databases
- Define Distributed Transactions
- Define Commit Protocols

13.1Introduction

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

13.2Distributed Databases

Distributed database system (DDBS) is a database in which storage devices are not all attached to a common CPU. It may be stored in a multiple computers located in the same physical location, or be dispersed over a network of interconnected computers. Take it simply, it is a database system that is logically centralized but physically distributed. It can be regarded as a combination of database system and computer network. Though this is an important issue in database architecture, the storage and query in the distributed database system is one of the most significant problems in the present database systems. Therefore, when preparing the lecture that focuses on the database storage and query, we inevitably encounter this topic. Here we will provide a brief introduction to DDBS, especially focusing on the query operations.

1. **Design Principles of DDBS :**C.J.Date, the early designer of relational database besides E.F.Codd, proposed the 12 design principles of DDBS which is now widely accepted and regarded as the standard definition of DDBS. They are:
(1) local autonomy; (2) no reliance on central site; (3) continuous operation; (4) location transparency and location independence; (5) fragmentation independence; (6) replication independence; (7) distributed query process; (8) distributed transaction management; (9) hardware independence; (10) operate system independence; (11) network independence; (12) DBMS (database management system) independence.As for the users, they will not feel like they are using a distributed database system. To the contrary, the main goal of DDBS design is to make the using of the system like operating on a local database. In this case, it is a real and general DDBS.

2. Data Fragmentation: The using of DDBS is to distribute the data to different servers so as to improve security and enhance the ability to deal with large scale of data, thus data fragmentation is the first step in storing a distributed database. Three types of data fragmentation are involved:

Horizontal Fragmentation: divide every tuples in the global relation into mutually exclusive sets, and each set is a fragment of the global relation. To retrieve the primal database we have to UNION the data fragments.Vertical Fragmentation: divide the attributes of the global relation into different sets. Applying the JOIN operation to the sets, we can retrieve the original relation.Mixed Fragmentation: the combination of the previous methods of fragmentation.With the methods above we are able to fragment the original relations and store them in different sites.

3. Independence of Data in DDBS: When using the DDBS, users do not have to have the knowledge about the distribution of global data. That is to say, the logical fragmentation of data, the physical location of the data fragments and the data model of every site is transparent to users. Hence, independence of data is also called data transparency. According to their relative hierarchy in the DDBS, there are three types of data transparency:

Fragmentation Transparency: users do not have to care about the data fragmentation. Even if the fragmentation changes, it will not influence on the use of the database.Location Transparency: users do not have to know about the duplicates of data fragments and where these duplicates are stored.Local Data Model Transparency: users do not have to know about the specific data model and the properties of the objects on every site.

4. Distributed Data Query Process : Query in DDBS is different from the traditional query since it involves in the communication between sites. Traditional query deal with two main costs: CPU and I/O. However, in DDBS, the time spent on data communication must be considered. It can be calculated by the following formula:

$$T(X)=C0+C1*X$$

In the formula, C0 stands for the fixed cost when communicate between two sites, C1 refers the transmission rate, with the unit "second per bit", and X is the amount of data been transmitted, with the unit "bit".For the system with high transmission speed, the main bottleneck of query efficiency is the same as query on a local database. However, in a system with low data communication rate, the time spent on the data transmission must be considered.

a) In implementing the distributed data query, there are four major processes:
b) Query decomposition: decompose the query to an expression of relational algebra.

c) Data localization: Transform the global algebraic expression into the expressions on each data fragments on different sites.

d) Global optimization: Find the best sequence of query operation, taking the cost of CPU, I/O and transmission into consideration.

e) Local optimization: It is implemented by each site to optimize the query there. It is the same as optimize the query on a local database.

5. Introduction to Semi-Join Algorithm : In the system where data transmission costs more time than data processing, an algorithm called semi-join algorithm, is applied. The operation semi-join is the combination of projection and joining. It only joins the common attributes between the two relations. Therefore, if we first get the semi-join of relation A and B, and transmit it to relation A and implement the joining operation, and then transmit the result to B and implement another joining operation, the data transmission is much less than the method of directly transmit the relation A to B and then join them.

According to the property of semi-join, if we need to join a small part in one relation to another relation, using semi-join is a desirable strategy.This is a brief introduction in the storage and query in distributed database system, and hope it will provide an overview of this prevail database system.

13.3Distributed Data Storage

A distributed database system allows applications to access data from local and remote databases. In a homogenous distributed database system, each database is an Oracle Database. In a heterogeneous distributed database system, at least one of the databases is not an Oracle Database. Distributed databases use a client/server architecture to process information requests.

This section contains the following topics:

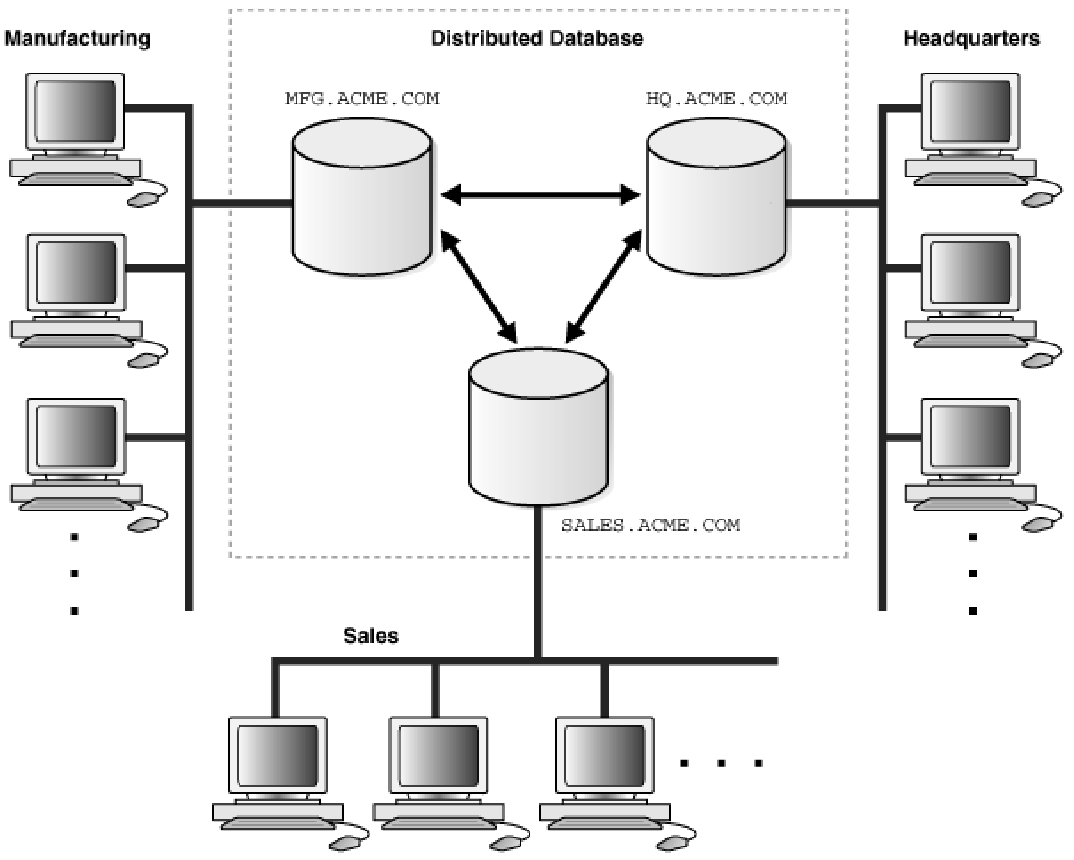
- Homogenous Distributed Database Systems
- Heterogeneous Distributed Database Systems
- Client/Server Database Architecture

Homogenous Distributed Database Systems

A homogenous distributed database system is a network of two or more Oracle Databases that reside on one or more machines. Figure illustrates a distributed system that connects three databases: hq, mfg, and sales. An application can simultaneously access or modify the data in several databases in a single distributed environment. For example, a single query from a Manufacturing client on local database mfg can retrieve joined data from the products table on the local database and the dept table on the remote hq database.

For a client application, the location and platform of the databases are transparent. You can also create synonyms for remote objects in the distributed system so that users can access them with the same syntax as local objects. For example, if you are connected to database mfg but want to access data on database hq, creating a synonym on mfg for the remote dept table enables you to issue this query:

SELECT * FROM dept;
In this way, a distributed system gives the appearance of native data access. Users on mfg do not have to know that the data they access resides on remote databases.



An Oracle Database distributed database system can incorporate Oracle Databases of different versions. All supported releases of Oracle Database can participate in a distributed database system. Nevertheless, the applications that work with the distributed database must understand the functionality that is available at each node in the system. A distributed database application cannot expect an Oracle7 database to understand the SQL extensions that are only available with Oracle Database.

Heterogeneous Distributed Database Systems

In a heterogeneous distributed database system, at least one of the databases is a non-Oracle Database system. To the application, the heterogeneous distributed database system appears as a single, local, Oracle Database. The local Oracle Database server hides the distribution and heterogeneity of the data.

The Oracle Database server accesses the non-Oracle Database system using Oracle Heterogeneous Services in conjunction with an agent. If you access the non-Oracle Database data store using an Oracle Transparent Gateway, then the agent is a system-specific application. For example, if you include a Sybase database in an Oracle Database distributed system, then you need to obtain a Sybase-specific transparent gateway so that the Oracle Database in the system can communicate with it.Alternatively, you can use generic connectivity to access non-Oracle Database data stores so long as the non-Oracle Database system supports the ODBC or OLE DB protocols.

Client/Server Database Architecture

A database server is the Oracle software managing a database, and a client is an application that requests information from a server. Each computer in a network is a node that can host one or more databases. Each node in a distributed database system can act as a client, a server, or both, depending on the situation.

In Figure, the host for the hq database is acting as a database server when a statement is issued against its local data (for example, the second statement in each transaction issues a statement against the local dept table), but is acting as a client when it issues a statement against remote data (for example, the first statement in each transaction is issued against the remote table emp in the sales database).

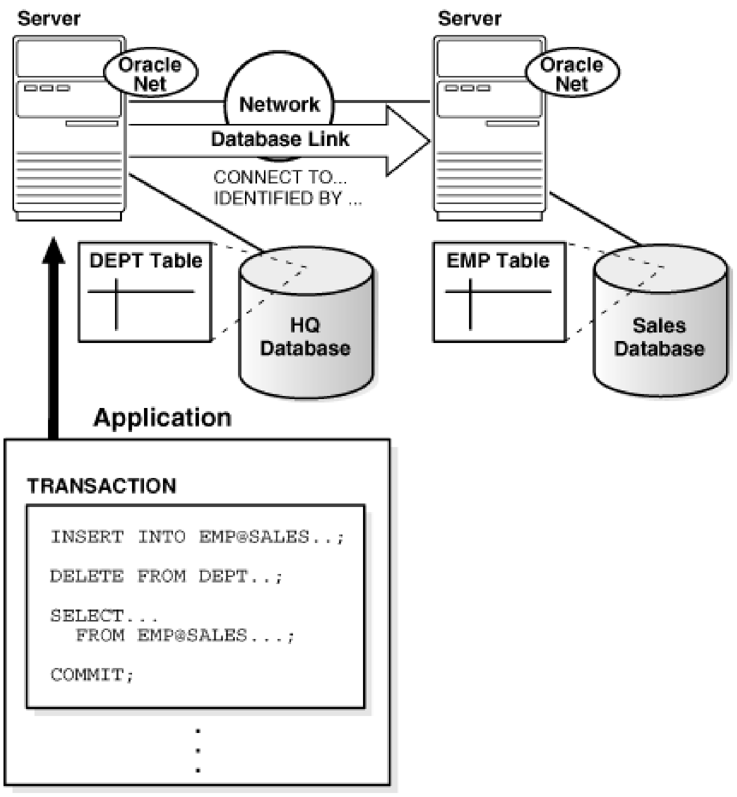
A client can connect directly or indirectly to a database server. A direct connection occurs when a client connects to a server and accesses information from a database contained on that server. For example, if you connect to the hq database and access the dept table on this database as in Figure 29-2, you can issue the following:

SELECT * FROM dept;

This query is direct because you are not accessing an object on a remote database.In contrast, an indirect connection occurs when a client connects to a server and then accesses information contained in a database on a different server. For example, if you connect to the hq database but access the emp table on the remote sales database as in Figure, you can issue the following:

SELECT * FROM emp@sales;

This query is indirect because the object you are accessing is not on the database to which you are directly connected.



13.4Distributed Query Processing (DQP)

DQP allows the tables from multiple distributed relational databases to be queried, using SQL, as if there were multiple tables in a single database. Consider two distributed data resources. The first is called MySurvey and contains survey data of people:

Data Resource: MySurvey			
Table: People			
name	**gender**	**age**	**postcode**
Alan	M	20	AB1 2GB
Bob	M	50	EH5 2TG
Cath	F	25	JK3 4RE

The second data resource is called PostcodeStats and maps postcodes to classifications of a region type:

Data Resource: PostcodeStats	
Table: PostcodeRegionType	
Postcode	**regionType**
AB1 2GB	City
FG2 5GB	Rural
EH5 2TG	City
JK3 4RE	Rural

If these two distributed relational data resources could be queried as if they were two tables in a single database then useful questions could be answered. DQP allows the creation of a virtual data resource that appears to the user to contain these two tables. By default, table names in the virtual data resource are prefixed with the original resource names so the two tables in our virtual data resource become:

Data Resource: MyDQPResource			
Table: MySurvey_People			
name	**Gender**	**age**	**postcode**
Alan	M	20	AB1 2GB
Bob	M	50	EH5 2TG
Cath	F	25	JK3 4RE

Data Resource: MyDQPResource	
Table: PostcodeStats_PostcodeRegionType	
Postcode	**regionType**
AB1 2GB	City
FG2 5GB	Rural
EH5 2TG	City
JK3 4RE	Rural

The MyDQPResource resource can be used to execute SQL queries that use both these tables. For example, to obtain the region type that Alan lives in, the SQL query is:

```
SELECT regionType
FROM MySurvey_People p JOIN PostcodeStats_PostcodeRegionType r
  ON p.postcode = r.postcode
WHERE name = 'Alan'
```

which produces result:

```
| regionType |
| City       |
```

The SQL query to find the the average age for people living in each region type would be:

```
SELECT regionType, avg(age) as avgAge
FROM MySurvey_People p JOIN PostcodeStats_PostcodeRegionType r
  ON p.postcode = r.postcode
GROUP BY regionType
```

which produces the result:

```
| regionType | avgAge |
| City       | 35.0   |
| Rural      | 25.0   |
```

The DQP federation will contain one table for each table in each of the distributed resources. Consider the following new relational resource that we now wish to add to our federation.

Data Resource: JoesSurvey			
Table: People			
name	**gender**	**age**	**postcode**
Darren	M	20	FG2 5GB
Ed	M	31	EH5 2TG
Fiona	F	29	FG2 5GB

This table will appear in our data federation as JoesSurvey_People. This will be an entirely different table from the MySurvey_People table. If we wish to consider these two tables to be portions of a single table then we must use a UNION operator in our SQL queries. For example, the query to obtain the average age for each region type now becomes:

```
SELECT regionType, avg(age) as avgAge
FROM
  ((SELECT age, postcode FROM MySurvey_People) UNION ALL
   (SELECT age, postcode FROM JoesSurvey_People)) as u
  JOIN PostcodeStats_PostcodeRegionType r ON u.postcode =
r.postcode
GROUP BY regionType
```

It will be possible in the future to combine DQP with OGSA-DAI's support of views to hide the need for union operators such as this from the client. For example, the DQP resource could be wrapped in a view resource that defines a new People table as:

```
SELECT * FROM
((SELECT name, age, postcode FROM MySurvey_People) UNION ALL
 (SELECT name, age, postcode FROM JoesSurvey_People))
```

As the time of writing the OGSA-DAI Views extension pack is not compatible with DQP. Check the OGSA-DAI website to see if a new version of the OGSA-DAI views extension pack that is compatible with DQP has been released.

The virtual federated resources that DQP provides can be configured statically at the server or dynamically by a client. Once the DQP resource has been created it can be used by clients just like any other relational resource.

13.5Distributed Transactions

A distributed transaction is a transaction that updates data on two or more networked computer systems. Distributed transactions extend the benefits of transactions to applications that must update distributed data. Implementing robust distributed applications is difficult because these applications are subject to multiple failures, including failure of the client, the server, and the network connection between the client and server. In the absence of distributed transactions, the application program itself must detect and recover from these failures.

Many aspects of a distributed transaction are identical to a transaction whose scope is a single database. For example, a distributed transaction provides predictable behavior by enforcing the ACID properties that define all transactions.

For distributed transactions, each computer has a local transaction manager. When a transaction does work at multiple computers, the transaction managers interact with other transaction managers via either a superior or subordinate relationship. These relationships are relevant only for a particular transaction.

Each transaction manager performs all the enlistment, prepare, commit, and abort calls for its enlisted resource managers (usually those that reside on that particular computer). Resource managers manage persistent or durable data and work in cooperation with the DTC to guarantee atomicity and isolation to an application.

In a distributed transaction, each participating component must agree to commit a change action (such as a database update) before the transaction can occur. The DTC performs the transaction coordination role for the components involved and acts as a transaction manager for each computer that manages transactions. When committing a transaction that is distributed among several computers, the transaction manager sends prepare, commit, and abort messages to all its subordinate transaction managers. In the two-phase commit algorithm for the DTC, phase one involves the transaction manager requesting each enlisted component to prepare to commit; in phase two, if all successfully prepare, the transaction manager broadcasts the commit decision.

In general, transactions involve the following steps:

1. Applications call the transaction manager to begin a transaction.

2. When the application has prepared its changes, it asks the transaction manager to commit the transaction. The transaction manager keeps a sequential transaction log so that its commit or abort decisions will be durable.

   - If all components are prepared, the transaction manager commits the transaction and the log is cleared.

   - If any component cannot prepare, the transaction manager broadcasts an abort decision to all elements involved in the transaction.

   - While a component remains prepared but not committed or aborted, it is in doubt about whether the transaction committed or aborted. If a component or transaction manager fails, it reconciles in-doubt transactions when it reconnects.

When a transaction manager is in-doubt about a distributed transaction, the transaction manager queries the superior transaction manager. The root transaction manager, also referred to as the global commit coordinator, is the transaction manager on the system that initiates a transaction and is never in-doubt. If an in-doubt transaction persists for too long, the system administrator can force the transaction to commit or abort.

13.6Commit Protocols

The two phase commit protocol is a distributed algorithm which lets all sites in a distributed system agree to commit a transaction. The protocol results in either all nodes committing the transaction or aborting, even in the case of site failures and message losses. However, due to the work by Skeen and Stonebraker, the protocol will not handle more than one random site failure at a time. The two phases of the algorithm are broken into the COMMIT-REQUEST phase, where the COORDINATOR attempts to prepare all the COHORTS, and the COMMIT phase, where the COORDINATOR completes the transactions at all COHORTS.

Assumptions

The protocol works in the following manner: One node is designated the coordinator, which is the master site, and the rest of the nodes in the network are called cohorts. Other assumptions of the protocol include stable storage at each site and use of a write ahead log by each node. Also, the protocol assumes that no node crashes forever, and eventually any two nodes can communicate with each other. The latter is not a big deal since network communication can typically be rerouted. The former is a much stronger assumption; suppose the machine blows up!

Basic Algorithm

During phase 1, initially the coordinator sends a query to commit message to all cohorts. Then it waits for all cohorts to report back with the agreement message. The cohorts, if the transaction was successful, write an entry to the undo log and an entry to the redo log. Then the cohorts reply with an agree message, or an abort if the transaction failed at a cohort node. During phase 2, if the coordinator receives an agree message from all cohorts, then it writes a commit record into its log and sends a commit message to all the cohorts. If all agreement messages do not come back the coordinator sends an abort message. Next the coordinator waits for the acknowledgement from the cohorts. When acks are received from all cohorts the coordinator writes a complete record to its log. Note the coordinator will wait forever for all the acknowledgements to come back. If the cohort receives a commit message, it releases all the locks and resources held during the transaction and sends an acknowledgement to the coordinator. If the message is abort, then the cohort undoes the transaction with the undo log and releases the resources and locks held during the transaction. Then it sends an acknowledgement.

Disadvantages

The greatest disadvantage of the two phase commit protocol is the fact that it is a blocking protocol. A node will block while it is waiting for a message. This means that other processes competing for resource locks held by the blocked processes will have to wait for the locks to be released. A single node will continue to wait even if all other sites have failed. If the coordinator fails permanently, some cohorts will never resolve their transactions. This has the effect that resources are tied up forever.Another disadvantage is the protocol is conservative. It is biased to the abort case rather than the complete case.

13.7 Summary

- Concurrency control is the activity of coordinating concurrent accesses to a database in a multi-user system, while preserving the consistency of the data. A number of anomalies can occur owing to concurrent accesses on data items in a DDBMS, which can lead to the inconsistency of the database. These are lost update anomaly, uncommitted dependency or dirty read anomaly, inconsistent analysis anomaly, non-repeatable or fuzzy read anomaly, phantom read anomaly and multiple-copy consistency problem. To prevent data inconsistency, it is essential to guarantee distributed serializability of concurrent transactions in a DDBMS.
- A distributed schedule or global schedule (the union of all local schedules) is said to be distributed serializable if the execution orders at each local site is serializable and local serialization orders are identical.
- The concurrency control techniques are classified into two broad categories: Pessimistic concurrency control mechanisms and Optimistic concurrency control mechanisms.
- Pessimistic algorithms synchronize the concurrent execution of transactions early in their execution life cycle.
- Optimistic algorithms delay the synchronization of transactions until the transactions are near to their completion. The pessimistic concurrency control algorithms are further classified into locking-based algorithms, timestamp-based algorithms and hybrid algorithms.
- In locking-based concurrency control algorithms, the synchronization of transactions is achieved by employing physical or logical locks on some portion of the database or on the entire database.
- In the case of timestamp-based concurrency control algorithms, the synchronization of transactions is achieved by assigning timestamps both to the transactions and to the data items that are stored in the database.
- To allow maximum concurrency and to improve efficiency, some locking-based concurrency control algorithms also involve timestamps ordering; these are called hybrid concurrency control algorithms.

13.8Keywords

Distributed Databases, Commit Protocols, Query Processing, Distributed Transactions

13.9Exercises

1. Define Commit Protocols?
2. Explain Distributed Databases in details.
3. Explain Distributed Query Processing
4. Discuss Distributed Transactions

13.10 Reference

1. Fundamentals of Database Systems By RamezElmasri, Shamkant B. Navathe, Durvasula V.L.N. Somayajulu, ShyamK.Gupta
2. Database System Concepts By AviSilberschatz, Henry F. Korth , S. Sudarshan
3. Database Management Systems Alexis Leon, Mathews, Vikas Publishing House

UNIT-14: Approaches to OODs

Structure

14.0 Objectives

After going through this unit, you will be able to: Describe Object Oriented Database Design

- Describe Approaches to OODs
- State Type and class hierarchies
- Define Complex Objects

14.1Introduction

We discuss object-oriented data models and database systems (Note 1). Traditional data models and systems, such as relational, network, and hierarchical, have been quite successful in developing the database technology required for many traditional business database applications. However, they have certain shortcomings when more complex database applications must be designed and implemented—for example, databases for engineering design and manufacturing (CAD/CAM and CIM (Note 2)), scientific experiments, telecommunications, geographic information systems, and multimedia (Note 3). These newer applications have requirements and characteristics that differ from those of traditional business applications, such as more complex structures for objects, longer-duration transactions, new data types for storing images or large textual items, and the need to define nonstandard application-specific operations. Object-oriented databases were proposed to meet the needs of these more complex applications. The object-oriented approach offers the flexibility to handle some of these requirements without being limited by the data types and query languages available in traditional database systems. A key feature of object-oriented databases is the power they give the designer to specify both the structure of complex objects and the operations that can be applied to these objects.

Another reason for the creation of object-oriented databases is the increasing use of object-oriented programming languages in developing software applications. Databases are now becoming fundamental components in many software systems, and traditional databases are difficult to use when embedded in object-oriented software applications that are developed in an object-orientedprogramming language such as C++, SMALLTALK, or JAVA. Object-oriented databases are designed so they can be directly—or seamlessly—integrated with software that is developed using object-oriented programming languages.

The need for additional data modelling features has also been recognized by relational DBMS vendors, and the newer versions of relational systems are incorporating many of the features that were proposed for object-oriented databases. This has led to systems that are characterized as object-relational or extended relational DBMSs (see Chapter 13). The next version of the SQL standard for relational DBMSs, SQL3, will include some of these features.
In the past few years, many experimental prototypes and commercial object-oriented database systems have been created. The experimental prototypes include the ORION system developed at MCC (Note 4), OPENOODB at Texas Instruments, the IRIS system at Hewlett-Packard laboratories, the ODE system at AT&T Bell Labs (Note 5), and the ENCORE/ObServer project at Brown University. Commercially available systems include GEMSTONE/OPAL of GemStone Systems, ONTOS of Ontos, Objectivity of Objectivity Inc., Versant of Versant Object Technology, ObjectStore of Object Design, ARDENT of ARDENT Software (Note 6), and POET of POET Software. These represent only a partial list of the experimental prototypes and the commercially available object-oriented database systems.

14.2Approaches to OODs

This section gives a quick overview of the history and main concepts of object-oriented databases, or OODBs for short. The OODB concepts are then explained in more detail. The term object-oriented—abbreviated by OO or O-O—has its origins in OO programming languages, or OOPLs. Today OO concepts are applied in the areas of databases, software engineering, knowledge bases, artificial intelligence, and computer systems in general. OOPLs have their roots in the SIMULA language, which was proposed in the late 1960s. In SIMULA, the conceptof a class groups together the internal data structure of an object in a class declaration. Subsequently, researchers proposed the concept of abstract data type, which hides the internal data structures and specifies all possible external operations that can be applied to an object, leading to the concept of encapsulation. The programming language SMALLTALK, developed at Xerox PARC (Note 9) in the 1970s, was one of the first languages to explicitly incorporate additional OO concepts, such as message passing and inheritance. It is known as a pure OO programming language, meaning that it was explicitly designed to be object-oriented. This contrasts with hybrid OO programming languages, which incorporate OO concepts into an already existing language. An example of the latter is C++, which incorporates OO concepts into the popular C programming language.

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

One goal of OO databases is to maintain a direct correspondence between real-world and database objects so that objects do not lose their integrity and identity and can easily be

identified and operated upon. Hence, OO databases provide a unique system-generated object identifier (OID) for each object. We can compare this with the relational model where each relation must have a primary key attribute whose value identifies each tuple uniquely. In the relational model, if the value of the primary key is changed, the tuple will have a new identity, even though it may still represent the same real-world object. Alternatively, a real-world object may have different names for key attributes in different relations, making it difficult to ascertain that the keys represent the same object (for example, the object identifier may be represented as EMP_ID in one relation and as SSN in another).
Another feature of OO databases is that objects may have an object structure of arbitrary complexity in order to contain all of the necessary information that describes the object. In contrast, in traditional database systems, information about a complex object is often scattered over many relations or records, leading to loss of direct correspondence between a real-world object and its database representation.

The internal structure of an object in OOPLs includes the specification of instance variables, which hold the values that define the internal state of the object. Hence, an instance variable is similar to the concept of an attribute, except that instance variables may be encapsulated within the object and thus are not necessarily visible to external users. Instance variables may also be of arbitrarily complex data types. Object-oriented systems allow definition of the operations or functions (behavior) that can be applied to objects of a particular type. In fact, some OO models insist that all operations a user can apply to an object must be predefined. This forces a complete encapsulation of objects. This rigid approach has been relaxed in most OO data models for several reasons. First, the database user often needs to know the attribute names so they can specify selection conditions on the attributes to retrieve specific objects. Second, complete encapsulation implies that any simple retrieval requires a predefined operation, thus making ad hoc queries difficult to specify on the fly.

To encourage encapsulation, an operation is defined in two parts. The first part, called the signature or interface of the operation, specifies the operation name and arguments (or parameters). The second part, called the method or body, specifies the implementation of the operation. Operations can be invoked by passing a message to an object, which includes the operation name and the parameters. The object then executes the method for that operation. This encapsulation permits modification of the internal structure of an object, as well as the implementation of its operations, without the need to disturb the external programs that invoke these operations. Hence, encapsulation provides a form of data and operation independence another key concept in OO systems is that of type and class hierarchies and inheritance. This permits specification of new types or classes that inherit much of their structure and operations from previously defined types or classes. Hence, specification of object types can proceed systematically. This makes it easier to develop the data types of a system incrementally, and to reuse existing type definitions when creating new types of objects.

One problem in early OO database systems involved representing relationships among objects. The insistence on complete encapsulation in early OO data models led to the argument that relationships should not be explicitly represented, but should instead be described by defining appropriate methods that locate related objects. However, this approach does not work very well for complex databases with many relationships, because it is useful to identify these relationships and make them visible to users. The ODMG 2.0 standard has recognized this need and it explicitly represents binary relationships via a pair of inverse references—that is, by placing the OIDs of related objects within the objects themselves, and maintaining referential integrity.

Some OO systems provide capabilities for dealing with multiple versions of the same object—a feature that is essential in design and engineering applications. For example, an old version of an object that represents a tested and verified design should be retained until the new version is tested and verified. A new version of a complex object may include only a few new versions of its component objects, whereas other components remain unchanged. In addition to permitting versioning, OO databases should also allow for schema evolution, which occurs when type declarations are changed or when new types or relationships are created. These two features are not specific to OODBs and should ideally be included in all types of DBMSs.

Another OO concept is operator polymorphism, which refers to an operation's ability to be applied to different types of objects; in such a situation, an operation name may refer to several distinct implementations, depending on the type of objects it is applied to. This feature is also called operator overloading. For example, an operation to calculate the area of a geometric object may differ in its method (implementation), depending on whether the object is of type triangle, circle, or rectangle. This may require the use of late binding of the operation name to the appropriate method at run-time, when the type of object to which the operation is applied becomes known.

14.3Object Identity, Object Structure, and Type Constructors

In this section we first discuss the concept of object identity, and then we present the typical structuring operations for defining the structure of the state of an object. These structuring operations are often called type constructors. They define basic data-structuring operations that can be combined to form complex object structures.

Object Identity

An OO database system provides a unique identity to each independent object stored in the database. This unique identity is typically implemented via a unique, system-generated object identifier, orOID. The value of an OID is not visible to the external user, but it is used internally by the system to identify each object uniquely and to create and manage inter-object references.

The main property required of an OID is that it be immutable; that is, the OID value of a particular object should not change. This preserves the identity of the real-world object being represented. Hence, an OO database system must have some mechanism for generating OIDs and preserving the immutability property. It is also desirable that each OID be used only once; that is, even if an object is removed from the database, its OID should not be assigned to another object. These two properties imply that the OID should not depend on any attribute values of the object, since the value of an attribute may be changed or corrected. It is also generally considered inappropriate to base the OID on the physical address of the object in storage, since the physical address can change after a physical reorganization of the database. However, some systems do use the physical address as OID to increase the efficiency of object retrieval. If the physical address of the object changes, an indirect pointer can be placed at the former address, which gives the new physical location of the object. It is more common to use long integers as OIDs and then to use some form of hash table to map the OID value to the physical address of the object.

Some early OO data models required that everything—from a simple value to a complex object—be represented as an object; hence, every basic value, such as an integer, string, or Boolean value, has an OID. This allows two basic values to have different OIDs, which can be useful in some cases. For example, the integer value 50 can be used sometimes to mean a weight in kilograms and at other times to mean the age of a person. Then, two basic objects with distinct OIDs could be created, but both objects would represent the integer value 50. Although useful as a theoretical model, this is not very practical, since it may lead to the generation of too many OIDs. Hence, most OO database systems allow for the representation of both objects and values. Every object must have an immutable OID, whereas a value has no OID and just stands for itself. Hence, a value is typically stored within an object and cannot be referenced from other objects. In some systems, complex structured values can also be created without having a corresponding OID if needed.

Object Structure

In OO databases, the state (current value) of a complex object may be constructed from other objects (or other values) by using certain type constructors. One formal way of representing such objects is to view each object as a triple (i, c, v), where i is a unique object identifier (the OID), c is a type constructor (Note 12) (that is, an indication of how the object state is constructed), and v is the object state (or current value). The data model will typically include several type constructors. The three most basic constructors are atom, tuple, and set. Other commonly used constructors include list, bag, and array. The atom constructor is used to represent all basic atomic values, such as integers, real numbers, character strings, Booleans, and any other basic data types that the system supports directly.

The object state v of an object (i, c, v) is interpreted based on the constructor c. If c = atom, the state (value) v is an atomic value from the domain of basic values supported by the system. If c = set, the state v is a set of object identifiers , which are the OIDs for a set of objects that are typically of the same type. If c = tuple, the state v is a tuple of the form , where each is an attribute name (Note 13) and each is an OID. If c = list, the value v is an ordered list of OIDs of objects of the same type. A list is similar to a set except that the OIDs in a list are ordered, and hence we can refer to the first, second, or object in a list. For c = array, the state of the object is a single-dimensional array of object identifiers. The main difference between array and list is that a list can have an arbitrary number of elements whereas an array typically has a maximum size. The difference between set and bag is that all elements in a set must be distinct whereas a bag can have duplicate elements.

This model of objects allows arbitrary nesting of the set, list, tuple, and other constructors. The state of an object that is not of type atom will refer to other objects by their object identifiers. Hence, the only case where an actual value appears is in the state of an object of type atom The type constructors set, list, array, and bag are called collection types (or bulk types), to distinguish them from basic types and tuple types. The main characteristic of a collection type is that the state of the object will be a collection of objects that may be unordered (such as a set or a bag) or ordered (such as a list or an array). The tuple type constructor is often called a structured type, since it corresponds to the struct construct in the C and C++ programming languages.

14.4Creation of Values of Complex Types

In SQL:1999 constructor functions are used to create values of structured types. A function with the same name as a structured type is a constructor function for thestructured type. For instance, we could declare a constructor for the type *Publisher*like this:

create function *Publisher* (*n* **varchar**(20), *b* **varchar**(20))

returns*Publisher*

begin

set*name* = *n*;

set*branch* = *b*;

end

We can then use *Publisher*('McGraw-Hill', 'New York') to create a value of the type*Publisher*.

SQL:1999 also supports functions other than constructors, the names of such functions must be different from the name of any structuredtype.Note that in SQL:1999, unlike in object-oriented databases, a constructor creates avalue of the type, not an object of the type. That is, the value the constructor createshas no object identity. In SQL:1999 objects correspond to tuples of a relation, and arecreated by inserting a tuple in a relation.

By default every structured type has a constructor with no arguments, which setsthe attributes to their default values. Any other constructors have to be created explicitly.There can be more than one constructor for the same structured type; althoughthey have the same name, they must be distinguishable by the number of argumentsand types of their arguments.

An array of values can be created in SQL:1999 in this way:

array['Silberschatz', 'Korth', 'Sudarshan']

We can construct a row value by listing its attributes within parentheses. For instance,if we declare an attribute *publisher1* as a row type, we can constructthis value for it:

('McGraw-Hill', 'New York')without using a constructor.

We create set-valued attributes, such as *keyword-set*, by enumerating their elementswithin parentheses following the keyword **set**.We can create multiset values just likeset values, by replacing **set** by **multiset**.3Thus, we can create a tuple of the type defined by the *books* relation as:

(’Compilers’, **array**[’Smith’, ’Jones’], *Publisher*(’McGraw-Hill’, ’New York’),
 set(’parsing’, ’analysis’))

14.5Type Hierarchies and Inheritance

Another main characteristic of OO database systems is that they allow type hierarchies and inheritance. Type hierarchies in databases usually imply a constraint on the extents corresponding to the types in the hierarchy. We first discuss type hierarchies and then the constraints on the extents. We use a different OO model in this section—a model in which attributes and operations are treated uniformly—since both attributes and operations can be inherited.

Type Hierarchies and Inheritance

In most database applications, there are numerous objects of the same type or class. Hence, OO databases must provide a capability for classifying objects based on their type, as do other database systems. But in OO databases, a further requirement is that the system permit the definition of new types based on other predefined types, leading to a type (or class) hierarchy.

Typically, a type is defined by assigning it a type name and then defining a number of attributes (instance variables) and operations (methods) for the type. In some cases, the attributes and operations are together called functions, since attributes resemble functions with zero arguments. A function name can be used to refer to the value of an attribute or to refer to the resulting value of anoperation (method). In this section, we use the term function to refer to both attributes and operations of an object type, since they are treated similarly in a basic introduction to inheritance.

A type in its simplest form can be defined by giving it a type name and then listing the names of its visible (public) functions. When specifying a type in this section, we use the following format, which does not specify arguments of functions, to simplify the discussion:

TYPE_NAME: function, function, . . . , function

For example, a type that describes characteristics of a PERSON may be defined as follows:

PERSON: Name, Address, Birthdate, Age, SSN

In the PERSON type, the Name, Address, SSN, and Birthdate functions can be implemented as stored attributes, whereas the Age function can be implemented as a method that calculates the Age from the value of the Birthdate attribute and the current date.

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

EMPLOYEE: Name, Address, Birthdate, Age, SSN, Salary, HireDate, Seniority
STUDENT: Name, Address, Birthdate, Age, SSN, Major, GPA

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

The idea of defining a type involves defining all of its functions and implementing them either as attributes or as methods. When a subtype is defined, it can then inherit all of these functions and their implementations. Only functions that are specific or local to the subtype, and hence are notimplemented in the supertype, need to be defined and implemented. Therefore, we can declare EMPLOYEE and STUDENT as follows:

EMPLOYEE subtype-of PERSON: Salary, HireDate, Seniority
STUDENT subtype-of PERSON: Major, GPA

In general, a subtype includes all of the functions that are defined for its supertype plus some additional functions that are specific only to the subtype. Hence, it is possible to generate a type hierarchy to show the supertype/subtype relationships among all the types declared in the system.
As another example, consider a type that describes objects in plane geometry, which may be defined as follows:

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

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

Notice that the Area operation may be implemented by a different method for each subtype, since the procedure for area calculation is different for rectangles, triangles, and circles. Similarly, the attribute ReferencePoint may have a different meaning for each subtype; it might be the center point for RECTANGLE and CIRCLE objects, and the vertex point between the two given sides for a TRIANGLE object. Some OO database systems allow the renaming of inherited functions in different subtypes to reflect the meaning more closely.

An alternative way of declaring these three subtypes is to specify the value of the Shape attribute as a condition that must be satisfied for objects of each subtype:
RECTANGLE subtype-of GEOMETRY_OBJECT (Shape=‘rectangle’): Width, Height
TRIANGLE subtype-of GEOMETRY_OBJECT (Shape=‘triangle’): Side1, Side2, Angle
CIRCLE subtype-of GEOMETRY_OBJECT (Shape=‘circle’): Radius
Here, only GEOMETRY_OBJECT objects whose Shape=‘rectangle’ are of the subtype RECTANGLE, and similarly for the other two subtypes. In this case, all functions of the

GEOMETRY_OBJECT supertype are inherited by each of the three subtypes, but the value of the Shape attribute is restricted to a specific value for each.

Notice that type definitions describe objects but do not generate objects on their own. They are just declarations of certain types; and as part of that declaration, the implementation of the functions of each type is specified. In a database application, there are many objects of each type. When an object is created, it typically belongs to one or more of these types that have been declared. For example, a circle object is of type CIRCLE and GEOMETRY_OBJECT (by inheritance). Each object also becomes a member of one or more persistent collections of objects (or extents), which are used to group together collections of objects that are meaningful to the database application.

Inheritance

Inheritance can be at the level of types, or at the level of tables. We first considerinheritance of types, then inheritance at the level of tables.

Type Inheritance

Suppose that we have the following type definition for people:

```
create type Person
(namevarchar(20),
addressvarchar(20))
```

We may want to store extra information in the database about people who are students,and about people who are teachers. Since students and teachers are also people,we can use inheritance to define the student and teacher types in SQL:1999:

```
create type Student
under Person
(degreevarchar(20),
departmentvarchar(20))
create type Teacher
under Person
(salary integer,
departmentvarchar(20))
```

Both Student and Teacher inherit the attributes of Person—namely, name and address.Student and Teacher are said to be subtypes of Person, and Person is a supertype ofStudent, as well as of Teacher.

Methods of a structured type are inherited by its subtypes, just as attributes are.However, a subtype can redefine the effect of a method by declaring the methodagain, using overriding method in place of method in the method declaration.

Now suppose that we want to store information about teaching assistants, whoare simultaneously students and teachers, perhaps even in different departments.TheSQL:1999 standard does not support multiple inheritance. However, draft versionsof the SQL:1999 standard provided for multiple inheritance, and although the finalSQL:1999 omitted it, future versions of the SQL standard may introduce it. We baseour discussion on the draft versions of the SQL:1999 standard.

For instance, if our type system supports multiple inheritance, we can define atype for teaching assistant as follows:

```
create type TeachingAssistant
under Student, Teacher
```

TeachingAssistant would inherit all the attributes of Student and Teacher. There is aproblem, however, since the attributes name, address, and department are present inStudent, as well as in Teacher.The attributes name and address are actually inherited from a common source, Person.So there is no conflict caused by inheriting them from Student as well as Teacher.However, the attribute department is defined separately in Student and Teacher. In fact,a teaching assistant may be a student of one department and a teacher in anotherdepartment. To avoid a conflict between the two occurrences of department, we canrename them by using an as clause, as in this definition of the type TeachingAssistant:

```
create type TeachingAssistant
under Student with (department as student-dept),
Teacher with (department as teacher-dept)
```

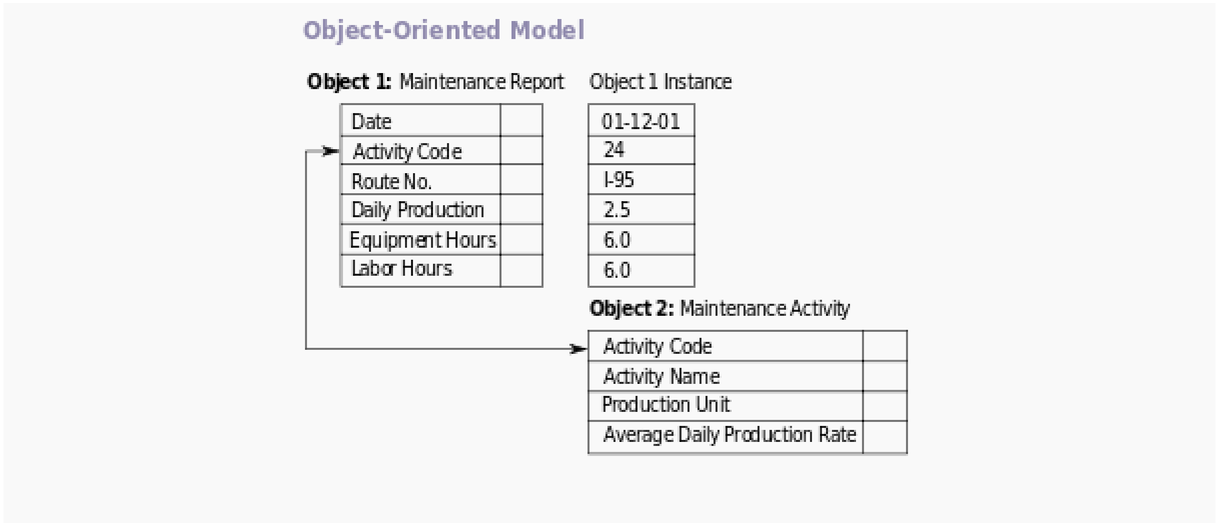
We note that SQL:1999 supports only single inheritance— that is, a type can inheritfrom only a single type; the syntax used is as in our earlier examples. Multiple inheritanceas in the TeachingAssistant example is not supported in SQL:1999. The SQL:1999standard also requires an extra field at the end of the type definition, whose valueis either final or not final.

The keyword final says that subtypes may not be createdfrom the given type, while not final says that subtypes may be created.

In SQL as in most other languages, a value of a structured type must have exactlyone "most-specific type." That is, each value must be associated with one specifictype, called its most-specific type, when it is created. By means of inheritance, itis also associated with each of the supertypes of its most specific type. For example,suppose that an entity has the type Person, aswell as the type Student. Then, the mostspecifictype of the entity is Student, since Student is a subtype of Person. However, anentity cannot have the type Student, as well as the type Teacher, unless it has a type,such as TeachingAssistant, that is a subtype of Teacher, as well as of Student.

14.6Object Relational DBMS-Overview

An object-relational database (ORD), or object-relational database management System (ORDBMS), is a database management system (DBMS) similar to a relational database, but with an object-oriented database model: objects, classes and inheritance are directly supported in database schemas and in the query language. In addition, just as with proper relational systems, it supports extension of the data model with custom data-types and methods.



Example of an object-oriented database model. Object-relational database can be said to provide a middle ground between relational databases and object-oriented databases(OODBMS). In object-relational databases, the approach is essentially that of relational databases: the data resides in the database and is manipulated collectively with queries in a query language; at the other extreme are OODBMSes in which the database is essentially a persistent object store for software written in an object-oriented programming language, with a programming API for storing and retrieving objects, and little or no specific support for querying.

The basic goal for the Object-relational database is to bridge the gap between relational databases and the object-oriented modelling techniques used in programming languages such as Java, C++, Visual Basic .NET or C#. However, a more popular alternative for achieving such a bridge is to use a standard relational database systems with some form of Object-relational mapping (ORM) software. Whereas traditional RDBMS or SQL-DBMS products focused on the efficient management of data drawn from a limited set of data-types (defined by the relevant language standards), an object-relational DBMS allows software developers to integrate their own types and the methods that apply to them into the DBMS.

The ORDBMS (like ODBMS or OODBMS) is integrated with an object-oriented programming language. The characteristic properties of ORDBMS are 1) complex data, 2) type inheritance, and 3) object behavior. Complex data creation in most SQL ORDBMSs is based on preliminary schema definition via the user-defined type (UDT). Hierarchy within structured complex data offers an additional property, type inheritance. That is, a structured type can have subtypes that reuse all of its attributes and contain additional attributes specific to the subtype. Another advantage, the object behavior, is related with access to the program objects. Such program objects have to be storable and transportable for database processing; therefore they usually are named as persistent objects. Inside a database, all the relations with a persistent program object are relations with its object identifier (OID). All of these points can be addressed in a proper relational system, although the SQL standard and its implementations impose arbitrary restrictions and additional complexity.

In object-oriented programming (OOP) object behavior is described through the methods (object functions). The methods denoted by one name are distinguished by the type of their parameters and type of objects for which they attached (method signature). The OOP languages call this the polymorphism principle, which briefly is defined as "one interface, many implementations". Other OOP principles, inheritance and encapsulation are related both, with methods and attributes. Method inheritance is included in type inheritance. Encapsulation in OOP is a visibility degree declared, for example, through the PUBLIC, PRIVATE and PROTECTED modifiers.

14.7 Summary

In this chapter we discussed the concepts of the object-oriented approach to database systems, which was proposed to meet the needs of complex database applications and to add database functionality to object-oriented programming languages such as C++. We first discussed the main concepts used in OO databases, which include the following:

- Object identity: Objects have unique identities that are independent of their attribute values.
- Type constructors: Complex object structures can be constructed by recursively applying a set of basic constructors, such as tuple, set, list, and bag.
- Encapsulation of operations: Both the object structure and the operations that can be applied to objects are included in the object class definitions.
- Programming language compatibility: Both persistent and transient objects are handled uniformly. Objects are made persistent by being attached to a persistent collection.
- Type hierarchies and inheritance: Object types can be specified by using a type hierarchy, which allows the inheritance of both attributes and methods of previously defined types.
- Extents: All persistent objects of a particular type can be stored in an extent. Extents corresponding to a type hierarchy have set/subset constraints enforced on them.
- Support for complex objects: Both structured and unstructured complex objects can be stored and manipulated.
- Polymorphism and operator overloading: Operations and method names can be overloaded to apply to different object types with different implementations.
- Versioning: Some OO systems provide support for maintaining several versions of the same object.

In the next chapter, we show how some of these concepts are realized in the ODMG standard and give examples of specific OODBMSs. We also discuss object-oriented database design and a standard for distributed objects called CORBA.

14.8Keywords

Complex Objects, Persistence, Object oriented data model, Inheritance

14.9Exercises

1. Define Persistence?

2. Explain OODs, Object oriented data model
3. Explain Type and class hierarchies
4. Discuss Complex Objects

14.10 Reference

1. Fundamentals of Database Systems By RamezElmasri, Shamkant B. Navathe, Durvasula V.L.N. Somayajulu, ShyamK.Gupta
2. Database System Concepts By AviSilberschatz, Henry F. Korth , S. Sudarshan
3. Database Management Systems Alexis Leon, Mathews, Vikas Publishing House
4. An Introduction to Database Systems C.J.Date

UNIT-15: Security & integrity threats

Structure

15.0 Objectives

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

- Describe Security and integrity threats
- Define statistical database security
- Explain Security issue based on granting/revoking of privileges

15.1Introduction

Database security concerns the use of a broad range of information security controls to protect databases (potentially including the data, the database applications or stored functions, the database systems, the database servers and the associated network links) against compromises of their confidentiality, integrity and availability. It involves various types or categories of controls, such as technical, procedural/administrative and physical.

Database security is a specialist topic within the broader realms of computer security, information security and risk management.
Security risks to database systems include, for example:

Unauthorized or unintended activity or misuse by authorized database users, database administrators, or network/systems managers, or by unauthorized users or hackers (e.g. inappropriate access to sensitive data, metadata or functions within databases, or inappropriate changes to the database programs, structures or security configurations);

Malware infections causing incidents such as unauthorized access, leakage or disclosure of personal or proprietary data, deletion of or damage to the data or programs, interruption or denial of authorized access to the database, attacks on other systems and the unanticipated failure of database services;

Overloads, performance constraints and capacity issues resulting in the inability of authorized users to use databases as intended;

Physical damage to database servers caused by computer room fires or floods, overheating, lightning, accidental liquid spills, static discharge, electronic breakdowns/equipment failures and obsolescence;
Design flaws and programming bugs in databases and the associated programs and systems, creating various security vulnerabilities (e.g. unauthorized privilege escalation), data loss/corruption, performance degradation etc.;
Data corruption and/or loss caused by the entry of invalid data or commands, mistakes in database or system administration processes, sabotage/criminal damage etc.
Many layers and types of information security control are appropriate to databases, including:
Access control
Auditing
Authentication
Encryption
Integrity controls
Backups
Application security

Traditionally databases have been largely secured against hackers through network security measures such as firewalls, and network-based intrusion detection systems. While network security controls remain valuable in this regard, securing the database systems themselves, and the programs/functions and data within them, has arguably become more critical as networks are increasingly opened to wider access, in particular access from the Internet. Furthermore, system, program, function and data access controls, along with the associated user identification, authentication and rights management functions, have always been important to limit and in some cases log the activities of authorized users and administrators. In other words, these are complementary approaches to database security, working from both the outside-in and the inside-out as it were.

Many organizations develop their own "baseline" security standards and designs detailing basic security control measures for their database systems. These may reflect general information security requirements or obligations imposed by corporate information security policies and applicable laws and regulations (e.g. concerning privacy, financial management and reporting systems), along with generally-accepted good database security practices (such as appropriate hardening of the underlying systems) and perhaps security recommendations from the relevant database system and software vendors. The security designs for specific database systems typically specify further security administration and management functions (such as administration and reporting of user access rights, log management and analysis, database replication/synchronization and backups) along with various business-driven information security controls within the database programs and functions (e.g. data entry validation and audit trails). Furthermore, various security-related activities (manual controls) are normally incorporated into the procedures, guidelines etc. relating to the design, development, configuration, use, management and maintenance of databases.

15.2Security & integrity threats

With the increase in usage of databases, the frequency of attacks against those databases has also increased. Here we look at some of the threats that database administrators actually can do something about.

Database attacks are an increasing trend these days. What is the reason behind database attacks? One reason is the increase in access to data stored in databases. When the data is been accessed by many people, the chances of data theft increases. In the past, database attacks were prevalent, but were less in number as hackers hacked the network more to show it was possible to hack and not to sell proprietary information. Another reason for database attacks is to gain money selling sensitive information, which includes credit card numbers, Social Security Numbers, etc. We previously defined database security and talked about common database security concepts. Now let's look at the various types of threats that affect database security.

Types of threats to database security

1. Privilege abuse: When database users are provided with privileges that exceeds their day-to-day job requirement, these privileges may be abused intentionally or unintentionally.

Take, for instance, a database administrator in a financial institution. What will happen if he turns off audit trails or create bogus accounts? He will be able to transfer money from one account to another thereby abusing the excessive privilege intentionally.

Having seen how privilege can be abused intentionally, let us see how privilege can be abused unintentionally. A company is providing a “work from home” option to its employees and the employee takes a backup of sensitive data to work on from his home. This not only violates the security policies of the organization, but also may result in data security breach if the system at home is compromised.

2. Operating System vulnerabilities: Vulnerabilities in underlying operating systems like Windows, UNIX, Linux, etc., and the services that are related to the databases could lead to unauthorized access. This may lead to a Denial of Service (DoS) attack. This could be prevented by updating the operating system related security patches as and when they become available.

3. Database rootkits: A database rootkit is a program or a procedure that is hidden inside the database and that provides administrator-level privileges to gain access to the data in the database. These rootkits may even turn off alerts triggered by Intrusion Prevention Systems (IPS). It is possible to install a rootkit only after compromising the underlying operating system. This can be avoided by periodical audit trails, else the presence of the database rootkit may go undetected.

4. Weak authentication: Weak authentication models allow attackers to employ strategies such as social engineering and brute force to obtain database login credentials and assume the identity of legitimate database users.

5. Weak audit trails: A weak audit logging mechanism in a database server represents a critical risk to an organization especially in retail, financial, healthcare, and other industries with stringent regulatory compliance. Regulations such as PCI, SOX, and HIPAA demand extensive logging of actions to reproduce an event at a later point of time in case of an incident. Logging of sensitive or unusual transactions happening in a database must be done in an automated manner for resolving incidents. Audit trails act as the last line of database defense. Audit trails can detect the existence of a violation that could help trace back the violation to a particular point of time and a particular user.

15.3Defense mechanisms

Having a secure system is an essential target for any organization, to achieve that, a good security policy must be designed carefully and a multilayered approach to security should be deployed. Also, it is important to denote that the database security cannot achieve without a secured environment surrounds it, i.e. secured network, operating system and application.

The most common threats that face the database are less of integrity, less of availability and less of confidentiality. To protect database against these types of threats, it is common to implement four kinds of control mechanisms; Access control, inference control, flow control, and encryption control. But these database security mechanisms are not design to detect intrusions but to avoid it; this means it is not enough to protect the database. Although applying these mechanisms, database still violate from internal and external attackers. So the quick and accurate detection of attacks on a database system is a prerequisite for fast damage assessment and recovery.

In recent years, interest in database intrusion detection system as a second line of defense in database. There are three main reasons for this. Actions deemed malicious for a DBMS are not necessarily malicious for the underlying operating system or the network. Thus designing an IDS for the latter may not be effective against database attacks. In addition, host- or network-based IDSs are mainly focused on detecting external intrusions as opposed to internal intruders, while legitimate users who abuse their privileges are the primary security threat in database systems. Therefore, SQL injection and other SQL-based attack targeted at databases cannot be effectively detected by network- or host-based IDSs. The distinctive characteristics of DBMSs, together with their widespread use and the invaluable data they hold, make it vital to detect any intrusion attempts made at the database level.

15.4Statistical database auditing & control

Statistical databases are used mainly to produce statistics on various populations. The database may contain confidential data on individuals, which should be protected from user access. However, users are permitted to retrieve statistical information on the populations, such as averages, sums, counts, maximums, minimums, and standard deviations. The techniques that have been developed to protect the privacy of individual information are outside the scope of this book. We will only illustrate the problem with a very simple example, which refers to the relation shown in Figure 22.03. This is a PERSON relation with the attributes NAME, SSN, INCOME, ADDRESS, CITY, STATE, ZIP, SEX, and LAST_DEGREE.

A population is a set of tuples of a relation (table) that satisfy some selection condition. Hence each selection condition on the PERSON relation will specify a particular population of PERSON tuples. For example, the condition SEX = ‘M’ specifies the male population; the condition ((SEX = ‘F’) AND (LAST_DEGREE = ‘M. S.’ OR LAST_DEGREE = ‘PH.D. ’)) specifies the female population that has an M.S. or PH.D. degree as their highest degree; and the condition CITY = ‘Houston’ specifies the population that lives in Houston.

Statistical queries involve applying statistical functions to a population of tuples. For example, we may want to retrieve the number of individuals in a population or the average income in the population. However, statistical users are not allowed to retrieve individual data, such as the income of a specific person. Statistical database security techniques must prohibit the retrieval of individual data. This can be controlled by prohibiting queries that retrieve attribute values and by allowing only queries that involve statistical aggregate functions such as COUNT, SUM, MIN, MAX, AVERAGE, and STANDARD DEVIATION. Such queries are sometimes called statistical queries.

In some cases it is possible to infer the values of individual tuples from a sequence of statistical queries. This is particularly true when the conditions result in a population consisting of a small number of tuples. As an illustration, consider the two statistical queries:

```
Q1:
SELECT COUNT (*) FROM PERSON
WHERE,condition.;

Q2:
SELECT AVG (INCOME) FROM PERSON
WHERE,condition.;
```

Now suppose that we are interested in finding the SALARY of ‘Jane Smith’, and we know that she has a PH.D. degree and that she lives in the city of Bellaire, Texas. We issue the statistical query Q1 with the following condition:

```
(LAST_DEGREE=‘PH.D.’ AND SEX=‘F’ AND CITY=‘Bellaire’ AND STATE=‘Texas’)
```

If we get a result of 1 for this query, we can issue Q2 with the same condition and find the INCOME of Jane Smith. Even if the result of Q1 on the preceding condition is not 1 but is a small number—say, 2 or 3—we can issue statistical queries using the functions MAX, MIN, and AVERAGE to identify the possible range of values for the INCOME of Jane Smith.

The possibility of inferring individual information from statistical queries is reduced if no statistical queries are permitted whenever the number of tuples in the population specified by the selection condition falls below some threshold. Another technique for prohibiting retrieval of individual information is to prohibit sequences of queries that refer repeatedly to the same population of tuples. It is also possible to introduce slight inaccuracies or "noise" into the results of statistical queries deliberately, to make it difficult to deduce individual information from the results. The interested reader is referred to the bibliography for a discussion of these techniques.

15.5 Security issue based on granting/revoking of privileges

The typical method of enforcing discretionary access control in a database system is based on the granting and revoking of privileges. Let us consider privileges in the context of a relational DBMS. In particular, we will discuss a system of privileges somewhat similar to the one originally developed for the SQL language. Many current relational DBMSs use some variation of this technique. The main idea is to include additional statements in the query language that allow the DBA and selected users to grant and revoke privileges.

Types of Discretionary Privileges

In SQL2, the concept of authorization identifier is used to refer, roughly speaking, to a user account (or group of user accounts). For simplicity, we will use the words user or account interchangeably in place of authorization identifier. The DBMS must provide selective access to each relation in the database based on specific accounts. Operations may also be controlled; thus having an account does not necessarily entitle the account holder to all the functionality provided by the DBMS. Informally, there are two levels for assigning privileges to use the database system:

1. The account level: At this level, the DBA specifies the particular privileges that each account holds independently of the relations in the database.
2. The relation (or table) level: At this level, we can control the privilege to access each individual relation or view in the database.

The privileges at the account level apply to the capabilities provided to the account itself and can include the CREATE SCHEMA or CREATE TABLE privilege, to create a schema or base relation; the CREATE VIEW privilege; the ALTER privilege, to apply schema changes such as adding or removing attributes from relations; the DROP privilege, to delete relations or views; the MODIFY privilege, to insert, delete, or update tuples; and the SELECT privilege, to retrieve information from the database by using a SELECT query. Notice that these account privileges apply to the account in general. If a certain account does not have the CREATE TABLE privilege, no relations can be created from that account. Account-level privileges are not defined as part of SQL2; they are left to the DBMS implementers to define. In earlier versions of SQL, a CREATETAB privilege existed to give an account the privilege to create tables (relations).

The second level of privileges applies to the relation level, whether they are base relations or virtual (view) relations. These privileges are defined for SQL2. In the following discussion, the term relation may refer either to a base relation or to a view, unless we explicitly specify one or the other. Privileges at the relation level specify for each user the individual relations on which each type of command can be applied. Some privileges also refer to individual columns (attributes) of relations. SQL2 commands provide privileges at the relation and attribute level only. Although this is quite general, it makes it difficult to create accounts with limited privileges. The granting and revoking of privileges generally follows an authorization model for discretionary privileges known as the access matrix model, where the rows of a matrix M represent subjects (users, accounts, programs) and the columns represent objects (relations, records, columns, views, operations). Each position M(i, j) in the matrix represents the types of privileges (read, write, update) that subject i holds on object j.

To control the granting and revoking of relation privileges, each relation R in a database is assigned an owner account, which is typically the account that was used when the relation was created in the first place. The owner of a relation is given all privileges on that relation. In SQL2, the DBA can assign an owner to a whole schema by creating the schema and associating the appropriate authorization identifier with that schema, using the CREATE SCHEMA command (see Section 8.1.1). The owner account holder can pass privileges on any of the owned relations to other users by granting privileges to their accounts. In SQL the following types of privileges can be granted on each individual relation R:

- SELECT (retrieval or read) privilege on R: Gives the account retrieval privilege. In SQL this gives the account the privilege to use the SELECT statement to retrieve tuples from R.
- MODIFY privileges on R: This gives the account the capability to modify tuples of R. In SQL this privilege is further divided into UPDATE, DELETE, and INSERT privileges to apply the corresponding SQL command to R. In addition, both the INSERT and UPDATE privileges can specify that only certain attributes of R can be updated by the account.
- REFERENCES privilege on R: This gives the account the capability to reference relation R when specifying integrity constraints. This privilege can also be restricted to specific attributes of R.

Notice that to create a view, the account must have SELECT privilege on all relations involved in the view definition.

Specifying Privileges Using Views

The mechanism of views is an important discretionary authorization mechanism in its own right. For example, if the owner A of a relation R wants another account B to be able to retrieve only some fields of R, then A can create a view V of R that includes only those attributes and then grant SELECT on V to B. The same applies to limiting B to retrieving only certain tuples of R; a view V can be created by defining the view by means of a query that selects only those tuples from R that A wants to allow B to access.

Propagation of Privileges Using the GRANT OPTION

Whenever the owner A of a relation R grants a privilege on R to another account B, the privilege can be given to B with or without the GRANT OPTION. If the GRANT OPTION is given, this means that B can also grant that privilege on R to other accounts. Suppose that B is given the GRANT OPTION by A and that B then grants the privilege on R to a third account C, also with GRANT OPTION. In this way, privileges on R can propagate to other accounts without the knowledge of the owner of R. If the owner account A now revokes the privilege granted to B, all the privileges that B propagated based on that privilege should automatically be revoked by the system.

It is possible for a user to receive a certain privilege from two or more sources. For example, A4 may receive a certain UPDATE R privilege from both A2 and A3. In such a case, if A2 revokes this privilege from A4, A4 will still continue to have the privilege by virtue of having been granted it from A3. If A3 later revokes the privilege from A4, A4 totally loses the privilege. Hence, a DBMS that allows propagation of privileges must keep track of how all the privileges were granted so that revoking of privileges can be done correctly and completely.

An Example

Suppose that the DBA creates four accounts—A1, A2, A3, and A4—and wants only A1 to be able to create base relations; then the DBA must issue the following GRANT command in SQL:

```
GRANT CREATETAB TO A1;
```

The CREATETAB (create table) privilege gives account A1 the capability to create new database tables (base relations) and is hence an account privilege. This privilege was part of earlier versions of SQL but is now left to each individual system implementation to define. In SQL2, the same effect can be accomplished by having the DBA issue a CREATE SCHEMA command, as follows:

```
CREATE SCHEMA EXAMPLE AUTHORIZATION A1;
```

Now user account A1 can create tables under the schema called EXAMPLE. To continue our example, suppose that A1 creates the two base relations EMPLOYEE and DEPARTMENT then A1 is the owner of these two relations and hence has all the relation privileges on each of them.

Next, suppose that account A1 wants to grant to account A2 the privilege to insert and delete tuples in both of these relations. However, A1 does not want A2 to be able to propagate these privileges to additional accounts. Then A1 can issue the following command:

```
GRANT INSERT, DELETE ON EMPLOYEE, DEPARTMENT TO A2;
```

Notice that the owner account A1 of a relation automatically has the GRANT OPTION, allowing it to grant privileges on the relation to other accounts. However, account A2 cannot grant INSERT and DELETE privileges on the EMPLOYEE and DEPARTMENT tables, because A2 was not given the GRANT OPTION in the preceding command.

Next, suppose that A1 wants to allow account A3 to retrieve information from either of the two tables and also to be able to propagate the SELECT privilege to other accounts. Then A1 can issue the following command:

```
GRANT SELECT ON EMPLOYEE, DEPARTMENT TO A3 WITH GRANT OPTION;
```

The clause WITH GRANT OPTION means that A3 can now propagate the privilege to other accounts by using GRANT. For example, A3 can grant the SELECT privilege on the EMPLOYEE relation to A4 by issuing the following command:

15.6 Summary

In this chapter we discussed several techniques for enforcing security in database systems. Security enforcement deals with controlling access to the database system as a whole and controlling authorization to access specific portions of a database. The former is usually done by assigning accounts with passwords to users. The latter can be accomplished by using a system of granting and revoking privileges to individual accounts for accessing specific parts of the database. This approach is generally referred to as discretionary access control. We presented some SQL commands for granting and revoking privileges, and we illustrated their use with examples. Then we gave an overview of mandatory access control mechanisms that enforce multilevel security. These require the classifications of users and data values into security classes and enforce the rules that prohibit flow of information from higher to lower security levels.

15.7Keywords

Security, Integrity threats, Statistical database, revoking of privileges

15.8Exercises

1. Explain various Security & integrity threats
2. Explain Security issue based on granting/revoking of privileges
3. Discuss statistical database security

15.9Reference

1. Fundamentals of Database Systems By RamezElmasri, Shamkant B. Navathe, Durvasula V.L.N. Somayajulu, ShyamK.Gupta
2. Database System Concepts By AviSilberschatz, Henry F. Korth , S. Sudarshan
3. Database Management Systems By Raghu Ramakrishnan and Johannes Gehrke
4. Database Management Systems Alexis Leon, Mathews, Vikas Publishing House
5. An Introduction to Database Systems C.J.Date

UNIT-16: Case Study on Popular Database Package Oracle

Structure

16.0 OBJECTIVES

At the end of this unit, we will be able understand:

- Overview of ORACLE
- Database Design and Querying Tools;
- SQL Variations and Extensions
- Storage and Indexing; Query
- Processing and Optimization

16.1 INTRODUCTION

The Oracle Database (simply referred as Oracle) is an object-relational database management system (ORDBMS) produced and marketed by Oracle Corporation. Larry Ellison and his friends and former co-workers Bob Miner and Ed Oates started the consultancy Software Development Laboratories (SDL) in 1977. SDL developed the original version of the Oracle software. The name Oracle comes from the code-name of a CIA-funded project. Oracle has held a leading position in the relational database market. This unit gives a subset of the features, options, and functionally of Oracle products.

16.2 DATABSE DESIGN AND QUERY TOOLS

A variety of tools provide by Oracle for database design, querying, report generation and data analysis.

Most of Oracle's tools are included in the Oracle Internet Development Suite. This suite has various aspects of application development, including tools for forms development, data modelling, reporting, and querying. The suite supports the UML standard for development modelling. It provides class modelling to generate code for the business components for java framework as well as activity modelling for general purpose control flow modelling. Below, we discuss Oracle Designer, and Oracle Builder tools.

Oracle Designer: The Oracle Designer is a major database design tool. It translates business logic and data flows into a schema definitions and procedural scripts for application logic. It supports modelling techniques such as E-R diagrams, information engineering, and object analysis and design. It stores the design in Oracle Repository, which serves a single point of metadata for the application. Forms and Reports can be generated using metadata. Oracle Repository provides configuration management for database objects, forms applications, java classes, XML files, and other types of files.

Oracle Builder: The Oracle Builder is an application development tool for data warehousing. It is tool for design and development of all aspects of a data warehouse, including schema design, data mapping and transformations, data load processing, and metadata management. It supports both 3NF and star schemas and can also import designs from Oracle Designer. Oracle Warehouse Builder (OWB) is a data warehousing-centered data modelling and data integration solution, built into every Oracle database.

Warehouse Builder delivers the following capabilities:

- Relational and dimensional modelling
- Support for OLAP and cube organized MVs

- Support for Oracle data warehousing features such as partitioning
- Data warehousing-specific operators for loading cubes and dimensions
- Oracle Spatial transformations
- Metadata management, data lineage, impact analysis and change propagation for complex designs
- Fuzzy matching and merging
- Integration with third-party name and address cleansing products

Oracle gives tools for ad-hoc querying, report generation and data analysis, including OLAP. We discuss below Oracle Discoverer, Oracle Express Server.

Oracle Discoverer: Oracle Discoverer is a web-based tool-set for ad-hoc querying, reporting, data analysis, and Web-publishing for the Oracle Database environment. It was originally a stand-alone product. However, it has become a component of the Oracle Fusion Middleware suite, and renamed Oracle Business Intelligence Discoverer. Oracle Corporation markets it as a business intelligence product. It allows users to drill up and down on result sets, pivot data, and store calculations as reports that can be published in a variety of formats such as spreadsheets or HTML. It has wizards to help end user visualize data as graphs.

Oracle Express Server: Oracle Express Server is an Oracle database that provides multidimensional views of data. It is Oracle's OLAP product offering, which allows views to be created ahead of time (MOLAP) or directly from relational databases (ROLAP). Oracle Express Server also lets users gain access to their data from an Excel spreadsheet. It supports a wide variety of analytical queries s well as forecasting, modelling, and scenario management. It can use the relational database management systems as a back end for storage or use its own multidimensional storage of the data.

Oracle is moving away from supporting a separate storage engine and moving most of the calculations into SQL. In this model where all the data reside in the relational DBMS and where any remaining calculations that cannot be performed in SQL are done in a calculation engine running on the database server.

There are many reasons for this:

- For both analytical application and the data warehousing common security model can be used.
- A relational engine can scale to much larger data sets.
- Integration of multidimensional and data warehousing modelling can be done.
- No need of training DBA for two engines.
- The RDBMS has a larger set of features and functionality in many areas as high availability, backup and recovery, and third-party tool support.

16.3 SQL VARIATIONS AND EXTENSIONS

Oracle9i supports all core SQL:1999 features fully or partially, with some minor exceptions. In addition, Oracle supports a large number of other language constructs, some of which confirm with SQL:1999, while others are Oracle-specific in syntax or functionality.

A few Oracle SQL extensions examples:

- with: Conceptually, the WITH clause allows us to give names to predefined SELECT statements inside the context of a larger SELECT statement. We can then reference the NAMED SELECT statements later.
- connect by: This clause can be used to select data that has a hierarchical relationship (usually some sort of parent->child or boss->employee or thing->parts).
- upsert and multitable inserts: The upsert operation combines update and insert. Multitable inserts allow multiple tables to be updated based on a single scan of new data.

Object-Relational Features: Oracle has extensive support for object-relational constructs, including object types, collections types, object tables, table functions, object views, methods, user-defined aggregate functions, and XML data types.

Triggers: Oracle provides several types of triggers and several options for when and how they are invoked. Triggers are a special PL/SQL construct similar to procedures. However, a procedure is executed explicitly from another block via a procedure call, while a trigger is executed implicitly whenever the triggering event happens. The triggering event is either a

INSERT, DELETE, or UPDATE command. The timing can be either BEFORE or AFTER. The trigger can be either row-level or statement-level, where the former fires once for each row affected by the triggering statement and the latter fires once for the whole statement. Oracle allows the creation of instead of triggers for views that cannot be subject to DML operations.

Oracle provides many triggers that are executed on a variety of other events, like database startup or shutdown, user login or logout, server error messages, and DDL statements such as create, alter and drop.

16.4 STORAGE AND INDEXING

Oracle database consists of information stored in files and accessed through an instance, which is a shared memory area and set of processes that interact with the data in the files.

Table Spaces: A tablespace is a logical storage unit within an Oracle database. It is logical because a tablespace is not visible in the file system of the machine on which the database resides. A tablespace, in turn, consists of at least one datafile which, in turn, are physically located in the file system of the server.

The most common tablespaces are:

- The system tablespace: It holds the data dictionary and storage for triggers and stored procedures.
- Tablespaces for used data: they are used for storing the user data.
- Temporary tablespaces: They are created during sorting the database, if sort cannot be done in memory.

Tablespaces can also be used as a means of moving data between databases.

Segments:

Datablocks, Extents and Segments are units of logical database space allocated by Oracle. At the lowest level of granularity, data is stored by Oracle in Datablocks. One datablock is equivalent to a specific number of bytes of physical database space on a disk. At the next

level of logical database storage is extent which corresponds to a specific number of contiguous data blocks. A segment is a set of extents. Space is allocated in a segment in units of extents. When existing extents are full, extra extents are allocated by Oracle as and when required. So extents need not be contiguous on the disk.
A database block does not have to be the same as an operating system block in size, but should be a multiple of it.

There are four types segments:
Data segments, Index segments, Temporary segments, and Rollback segments.
For control of space allocation and management, Oracle provides storage parameters.

Tables:

A table in Oracle is heap organized. The storage location of a row in a table is not based on the values contained in it. In Oracle a table can have columns whose data type is another table, called nested tables. The nested tables are stored separately. Oracle temporary tables store the data only during either the transaction in which the data are inserted or the user session. Another form of organization for table data is cluster. Here the rows from different tables are stored together in the same block on the basis of some common columns.

Index-Organized tables:

Oracle Index-organized tables (IOTs) are a unique style of table structure that is stored in a B-tree index structure. Besides storing the primary key values of an Oracle indexed-organized tables row, each index entry in the B-tree also stores the non-key column values. Oracle Indexed-organized tables provide faster access to table rows by the primary key or any key that is a valid prefix of the primary key. Because the non-key columns of a row are present in the B-tree leaf block itself, there is no additional block access for index blocks.

Indices: Oracle supports several types of indices. The most commonly used is a B-tree index, created on one or more columns.

Bitmap Indices:

Oracle bitmap indexes are very different from standard b-tree indexes. In bitmap structures, a two-dimensional array is created with one column for every row in the table being indexed.

Each column represents a distinct value within the bitmapped index. This two-dimensional array represents each value within the index multiplied by the number of rows in the table.

At row retrieval time, Oracle decompresses the bitmap into the RAM data buffers so it can be rapidly scanned for matching values. These matching values are delivered to Oracle in the form of a Row-ID list, and these Row-ID values may directly access the required information.

The real benefit of bitmapped indexing occurs when one table includes multiple bitmapped indexes. Each individual column may have low cardinality. The creation of multiple bitmapped indexes provides a very powerful method for rapidly answering difficult SQL queries.

As the number if distinct values increases, the size of the bitmap increases exponentially, such that an index with 100 values may perform thousands of times faster than a bitmap index on 1,000 distinct column values.

Remember that bitmap indexes are only suitable for static tables and materialized views which are updated at night and rebuilt after batch row loading. If your tables are not read-only during query time, do not consider using bitmap indexes.

Function-Based Indices:

This is an ability to index functions and use these indexes in query. In a nutshell, this capability allows you to have case insensitive searches or sorts, search on complex equations, and extend the SQL language efficiently by implementing your own functions and operators and then searching on them.

Why to use this feature:

- It is easy and provides immediate value.
- It can be used to speed up existing applications without changing any of their logic or queries.
- It can be used to supply additional functionality to applications with very little cost.

Join Indices:

In join index the key columns are not in the table that is referenced by the row-ids in the index. Oracle supports bitmap join indices primarily for use with star schemas.

In a Bitmap Index, each distinct value for the specified column is associated with a bitmap where each bit represents a row in the table. A '1' means that row contains that value, a '0' means it doesn't.

Bitmap Join Indexes extend this concept such that the index contains the data to support the join query, allowing the query to retrieve the data from the index rather than referencing the join tables. Since the information is compressed into a bitmap, the size of the resulting structure is significantly smaller than the corresponding materialized view.

Oracle allows bitmap join indices to have more than one key column and these columns can be in different tables.

The bitmap join index is extremely useful for table joins that involve low-cardinality columns. However, bitmap join indexes aren't useful in all cases. You shouldn't use them for OLTP databases because of the high overhead associated with updating bitmap indexes.

Domain Indices:

These are indexes you build and store yourself, either in Oracle or outside of Oracle. For simple data types such as integers and small strings, all aspects of indexing can be easily handled by the database system. This is not the case for documents, images, video clips and other complex data types that require content-based retrieval (CBR). The essential reason is that complex data types have application specific formats, indexing requirements, and selection predicates. For example, there are many different document encodings (such as ODA, SGML, plain text) and information retrieval (IR) techniques (keyword, full-text boolean, similarity, probabilistic, and so on). To effectively accommodate the large and growing number of complex data objects, the database system must support application specific indexing. The approach that we employ to satisfy this requirement is termed *extensible indexing*.

With Extensible indexing,

- The application defines the structure of the domain index
- The application stores the index data either inside the Oracle database or outside the Oracle database
- The application manages, retrieves and uses the index data to evaluate user queries

In effect, the application controls the structure and semantic content of the domain index. The database system interacts with the application to build, maintain, and employ the domain index. It is highly desirable for the database to handle the physical storage of domain indexes.

Application domain indexing gives the user the ability to supply new indexing technology. Most people will never make use of it this particular API to build a new index type. The inetMedia set of functionality, implemented using the Application Domain indexing feature will provide indexing on text, XML documents and images.

Partitioning: Partitioning is a key requirement of high performance, high availability database environments. Partitioning splits tables and indexes into smaller, more manageable components. Oracle Database offers the widest choice of partitioning methods available, including interval, reference, list, and range. Additionally, it provides composite partitions of two methods. Oracle Partitioning is also the foundation of Oracle's Information Lifecycle Management strategy, which aligns the business value of information to cost-effective storage tiers for large data warehousing and transaction processing applications.

Partition Options (Methodology)

Range Partitioning

Range partitioning is often used when queries use dates in their ‘where’ clauses as a major criteria to eliminate records. Both a specific date as well as date ranges can be used. However, any column often used in a where clause may be a candidate for the partition key although as the name implies, range partitioning is most useful where queries contain a bounded range such that columnA is between value1 and value2. The sizes of the individual partitions implementing range partitioning do not necessarily have to be the same. Range partitioning makes it easy to archive or delete large amounts of data.

Hash Partitioning

This method appears to be better when values are not necessarily ordered such as in an abbreviation code or large numbers of single elements are used such as with an ‘in’ statement. Hash partitions allow Oracle to create evenly sized partitions so that the work can be better distributed. One drawback to hash partitions is that partition pruning can only be done if the criterian is limited to exact values. Range queries will not be appropriate for hash partitioning. Additionally, the following operations are not permitted: split, drop, and merge. This does limit the partition's ability to be manipulated and would indicate it would be best where the table to be partitioned does not undergo much change or where it can be easily rebuilt without causing major application outage time.

Composite Partitioning

As the name implies, composite partitioning combines the previous two methods. This method seems best where there is first a primary range used and then a secondary value used, that would be more specific and non-range oriented. Composite should give even better performance than straight range partitioning if appropriate. Composite partitioning uses sub-partitioning. A sub-partition is another segment and allows further levels of parallelism as well as separate tablespace placement and storage clauses.

List partitioning

List partitioning is a partitioning technique where you specify a list of discrete values for the partitioning key in the description for each partition. This type of partitioning is useful if the data in the partitioning column have a relatively small set of discrete values.

Materialized views

A materialized view is a stored summary containing pre-computed results (originating from an SQL select statement). As the data is pre-computed, materialized views allow for faster data warehousing query answers.

Types of materialized views

There are three types of materialized views:

- Read only materialized view
- Updateable materialized view
- Writeable materialized view

Oracle uses materialized views (also known as snapshots in prior releases) to replicate data to non-master sites in a replication environment and to cache expensive queries in a data warehouse environment.

16.5 QUERY PROCESSING AND OPTIMIZATION

The query processor turns user queries and data modification commands into a query plan, which is a sequence of operations (or algorithm) on the database. It translates high level queries to low level commands.

Decisions taken by the query processor:

- Which of the algebraically equivalent forms of a query will lead to the most efficient algorithm?
- For each algebraic operator what algorithm should we use to run the operator?
- How should the operators pass data from one to the other? (eg, main memory buffers, disk buffers)

Oracle supports a large variety of processing techniques in its query processing engine. Here we briefly describe the important ones.

Execution Methods:

A variety of methods exist to access data:

Full-Table Scan: In Oracle, a full-table scan is performed by reading all of the table rows, block by block, until the high-water mark for the table is reached. As a general rule, full-table scans should be avoided unless the SQL query requires a majority of the rows in the table.

Fast Full Index Scan: The fast full index scan is an alternative to a full table scan when there is an index that contains all the columns that are needed for the query

Index scan: Oracle accesses index based on the lowest cost of execution path available with index scan. The query processor creates a start and/or stop key from conditions in the query and uses it to scan to a relevant part of the index.

Index join: If a query needs only a small subset of the columns of a wide table, but no single index contains all those columns, the processor can use an index join to generate the relevant information without accessing the table, by joining several indices that together contain the needed columns.

Cluster and hash cluster access: The processor accesses the data by using the cluster key.

Query optimization in Oracle:

Oracle carries out query optimization in several stages.

A user query will go through 3 phases:

Parsing: At this stage the syntax and semantics are checked, at the end you have a parse tree which represents the query's structure. The statement is normalized so that it can be processed more efficiently, once all the checks have completed it is considered a valid parse tree and is sent to the logical query plan generation stage.

Optimization: The optimizer is used at this stage, which is a cost-based optimizer, this chooses the best access method to retrieve the requested data. It uses statistics and any hints specified in the SQL query. The CBO produces an optimal execution plan for the SQL statement.

The optimization process can be divided in two parts:
1) Query rewrite phase: The parse tree is converted into an abstract logical query plan, the various nodes and branches are replaced by operators of relational algebra.

2) Execution plan generation phase: Oracle transforms the logical query plan into a physical query plan, the physical query or execution plan takes into account the following factors

- The various operations (joins) to be performed during the query
- The order in which the operations are performed
- The algorithm to be used for performing each operation
- The best way to retrieve data from disk or memory
- The best way to pass data from operation to another during the query

The optimizer may well come up with multiple physical plans, all of which are potential execution plans. The optimizer then chooses among them by estimating the costs of each possible plan (based on table and index statistics) and selecting the plan with the lowest cost. This is called the cost-based query optimization (CBO).

Query Transformation: Query processing requires the transformation of your SQL query into an efficient execution plan, Query optimization requires that the best execution plan is the one executed, the goal is to use the least amount of resources possible (CPU and I/O), the more resources a query uses the more impact it has on the general performance of the database.

Some of the major types of transformation and rewrites supported by Oracle are View merging, complex view merging, subquery flattening, materialized view rewrite, star transformation.

Access Path Selection: In relational database management system (RDBMS) terminology, Access Path refers to the path chosen by the system to retrieve data after a structured query language (SQL) request is executed. Access path selection can make a tremendous impact on the overall performance of the system.

Optimization of access path selection maybe gauged using cost formulas with I/O and CPU utilization weight usually considered. Generally, query optimizers evaluate the available paths to data retrieval and estimate the cost in executing the statements using the determined paths or a combination of these paths. Access paths selection for joins where data is taken from than one table is basically done using the nested loop and merging scan techniques.

Because joins are more complex, there are some other considerations for determining access path selections for them.

16.6 SUMMARY

The Oracle Database is an object-relational database management system produced and marketed by Oracle Corporation. This unit has given a subset of the features, options, and functionally of Oracle products. The unit covered Database design and querying tools, SQL Variations and Extensions, Storage and Indexing, Query processing and optimization.

16.7 KEYWORDS

Full-Table Scan, Fast Full Index Scan Index scan: Index join, Cluster and hash cluster access:

16.8 EXERCISES

1. Explain database design and querying tools supported by ORACLE.
2. What are variations in SQL?
3. How storage and indexing is managed in ORACLE
4. Discuss query processing and optimization in ORACLE

16.9 Reference

1. Fundamentals of Database Systems By RamezElmasri, Shamkant B. Navathe, Durvasula V.L.N. Somayajulu, ShyamK.Gupta
2. Database System Concepts By AviSilberschatz, Henry F. Korth , S. Sudarshan
3. Database Management Systems By Raghu Ramakrishnan and Johannes Gehrke
4. An Introduction to Database Systems C.J.Date
5. Database Management Systems Alexis Leon, Mathews, Vikas Publishing House
6. Oracle manuals.