

**DATA BASE  
MANAGEMENT SYSTEM  
(PGDCA06)  
(PG - DIPLOMA)**



**ACHARYA NAGARJUNA UNIVERSITY**

**CENTRE FOR DISTANCE EDUCATION**

**NAGARJUNA NAGAR,**

**GUNTUR**

**ANDHRA PRADESH**

**An Introduction to Database Systems**

Bipin Desai

**Modern Database Management**

F. McFadden, J. Hoffer

**An Introduction to Database Systems**

C. J. Date;

**AUTHOR:**

**Y.SURESH BABU., M.Com, M.C.A.,  
Lecturer,  
Dept. Of Computer Science,  
JKC College.  
GUNTUR.**

### 1.14. Summary:

A database management system (DBMS) is a computer program designed to manage a database; a large set of structured data, and run operations on the data requested by numerous users. Typical examples of DBMS use include accounting, human resources and customer support systems. The primary goal of a DBMS is to provide an environment that is both convenient and efficient for the people to use in retrieving and storing information. The three levels of the architecture are three different views of the data. The design of each such level is considered as a schema and hence the whole design is referred as three-schema architecture. Level internal, view and external level.

Data model is a model that describes in an abstract way as to how data is represented in an information system or a database management system. Important data models are: entity-relationship, relational, network and hierarchical data models. A database administrator (DBA) is a person who is responsible for the environmental aspects of a database. A transaction is a collection of operations that performs a single logical function in a database application. Each transaction is a unit of both atomicity and consistency.

### 1.15. Technical Terms:

**Query:** A specific set of instructions for extracting particular data from a database.

**Metadata:** Data is useful when placed in some context. **Metadata** are data that describe the properties or characteristics, such as definitions, structures, and rules or constraints, of other data.

**Data Processing:** Systematically performing a series of actions with data. May be done by manual, mechanical, electromechanical, or electronic (primarily computer) means.

**Security:** The process of protecting information from unauthorized use. An example is the use of credit card numbers on the Internet to purchase merchandise and services.

### 1.16. Model Questions:

1. What is file processing? Explain the major disadvantages of file processing?
2. What is Database management system? Explain the advantages of database management system?
3. Write about different types of database users?
4. What are five main functions of Database Administrator (DBA)?

### 1.17. References:

#### Database System Concepts

Silberschatz, Korth, and Sudarshan

#### Database Management Systems

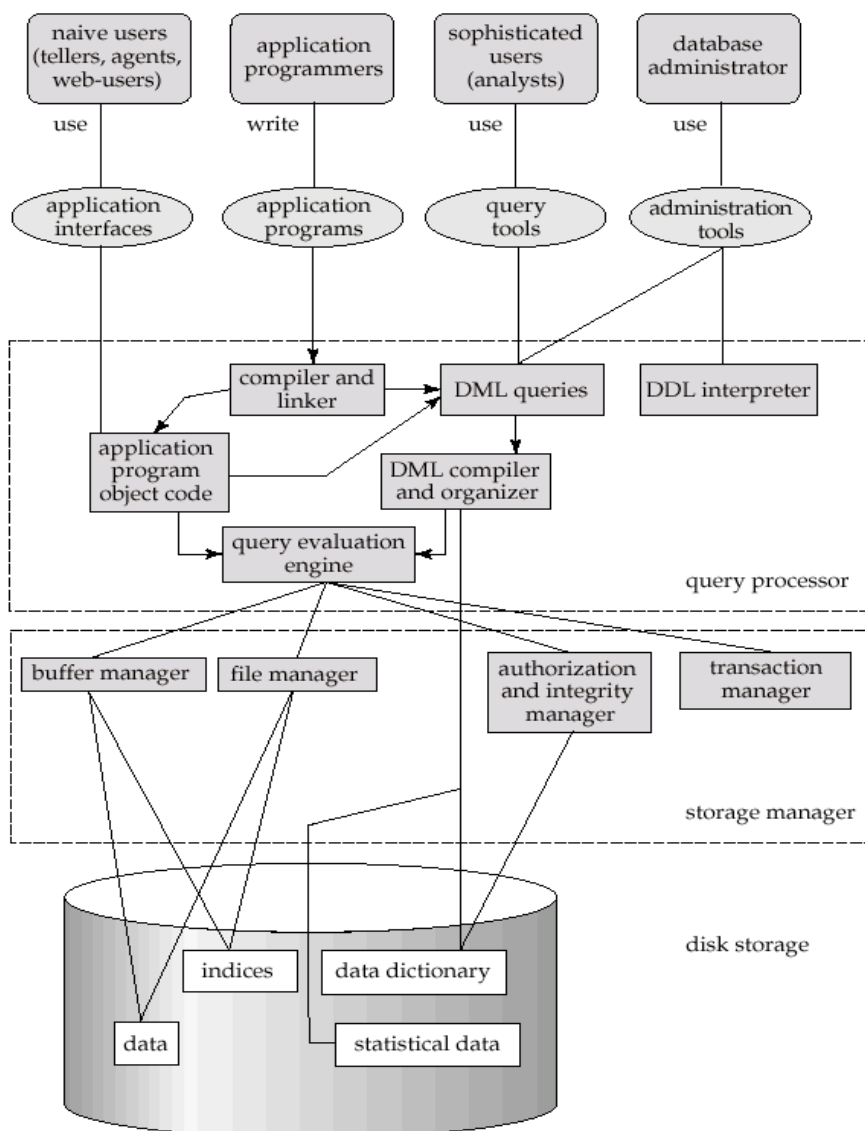
Arun K. Majumdar, Pritimoy Bhattacharyya

### 1.13. Database System Structure:

A database system is partitioned into modules that deal with each of the responsibilities of the overall system. The functional components of a database system can be broadly divided into the storage manager and the query processor components.

Storage manager is a program module that provides the interface between the low-level data stored in the database and the application programs and queries submitted to the system.

The query processor helps the database system in simplifying and facilitate to access data.



**Database system Structure**

**End Users:**

Persons who add, delete, and modify data in the database and who request and receive information from it.

**1.11. Database Administrator:**

The database administrator (DBA) is the person (or group of people) responsible for overall control of the database system. The DBA would normally have a large number of tasks related to maintaining and managing the database.

The DBA's responsibilities include the following:

**Deciding and Loading the Database Contents:**

The DBA in consultation with senior management is normally responsible for defining the conceptual schema of the database. The DBA would also be responsible for making changes to the conceptual schema of the database if and when necessary.

**Assisting and Approving Applications and Access:**

The DBA would normally provide assistance to end-users interested in writing application programs to access the database. The DBA would also approve or disapprove access to the various parts of the database by different users.

**Deciding Data Structures:**

Once the database contents have been decided, the DBA would normally make decisions regarding how data is to be stored and what indexes need to be maintained. In addition, a DBA normally monitors the performance of the DBMS and makes changes to data structures if the performance justifies them. In some cases, radical changes to the data structures may be called for.

**Backup and Recovery:**

Since the database is such a valuable asset, the DBA must make all the efforts possible to ensure that the asset is not damaged or lost. This normally requires a DBA to ensure that regular backups of a database are carried out and in case of failure (or some other disaster like fire or flood), suitable recovery procedures are used to bring the database up with as little down time as possible.

**Monitor Actual Usage:**

The DBA monitors actual usage to ensure that policies laid down regarding use of the database are being followed. The usage information is also used for performance tuning.

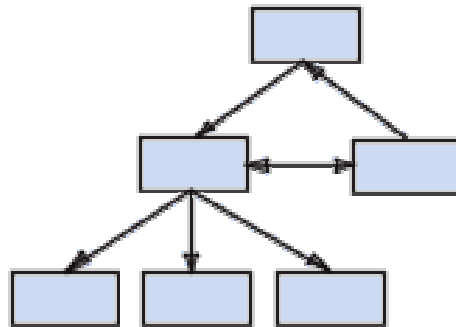
**1.12. Transaction Management:**

A *transaction* is a collection of operations that performs a single logical function in a database application. Each transaction is a unit of both atomicity and consistency.

Transaction management component ensures that the database remains in a consistent (correct) state despite system failures (e.g., power failures and operating system crashes) and transaction failures.

Concurrency control manager controls the interaction among the concurrent transactions, to ensure the consistency of the database

Symbolic representation:



### Physical Data Models:

The physical data model is used to describe data at lowest level. Unlike other data models, physical data models are not much in use. Two widely known ones are:

- Unifying data model.
- Frame-memory data model.

## 1.9. Database Languages :

A database system provides a data definition language to specify the database schema and a data manipulation language to express database queries and updates. In practice, the data definition and data manipulation languages are not two separate languages, instead they simply form parts of a single database languages, such as widely used SQL language.

## 1.10. Database Users:

A major goal of a database system is to retrieve information from and store new information in the database. People who work with a database can be categorized as database users and database administrators.

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

### Naive Users:

Naïve users are unsophisticated users who interact with the system by invoking one of the application programs that have been written previously.

### Application Programmers:

Application programmers are computer professionals who write application programs. Application programmers can choose from many tools to develop user interfaces.

### Sophisticated Users:

Sophisticated users interact with the system without writing programs. Instead, they form their requests in a database query language.

### Specialized Users:

Specialized users are sophisticated users who write specialized database applications that do not fit into the traditional data processing framework

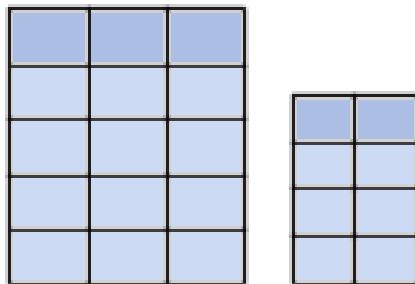
The three widely acceptable record based data models are:

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

### Relational Data Model:

The relational data model uses the concept of relations to represent each and every file of a system. Relations are nothing but two-dimensional arrays (or tables) that represent data in a most efficient way. (Or) A relational data model is a collection of tables and associated relationships among those data. Each table is a combination of multiple rows and columns, and each column has unique name. In other words a relational data model is exactly, a way of looking at data i.e., creation and manipulation of data. In another way, we can say that a database management system that manages information in terms of tables is nothing but RDBMS.

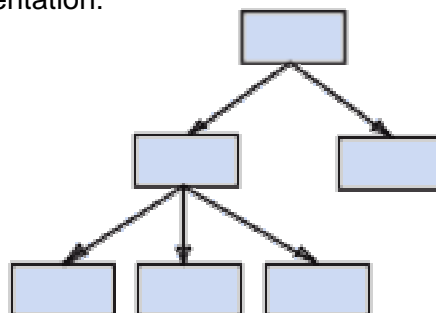
Symbolic Representation is :



### Hierarchical Data Model:

The hierarchical data model is similar to network data model in the sense that data and relationships among the data are represented as records and links respectively. In this data model, unlike network data model, records are organized in terms of tree structure.

Symbolic representation:



### Network Data Model:

Data in network data model is represented by a collection of records and relationships among data represented by links, which are viewed as pointers. Literally the records are represented as graphs.

**Logical or Conceptual Schema:**

The conceptual view is the information model of the enterprise and contains the view of the whole enterprise without any concern for the physical implementation. In this level, we describe what data are to be stored and what relationships exist among the data. The database administrators use this level of abstraction.

**Internal Schema:**

The lowest level of data abstraction describes how the data is actually stored. In this level, complex data structures of low level are described in detail. In other words, the internal view is the view about the actual physical storage of data. It tells us what data is stored in the database and how.

**1.8. Data Models:**

A data model tells about the underlying structure of a database. It is a collection of conceptual tools like describing the data, data relationships, data semantics and consistency constraints. The various data models that have been proposed fall into three categories:

- Object based data model.
- Record based data model.
- Physical data model.

**Object Based Data Model:**

This data model is used in describing data at the logical and view levels and it specify fairly flexible structures. There is another classification in this model:

- ◆ Entity-Relationship model.
- ◆ Object- oriented data model.
- ◆ Semantic data model.
- ◆ Functional data model.

**Entity Relationship Data Model:**

The entity-relationship (E-R) data model is based on a perception of real world that consists of a collection of basic objects called entities and relationships among these objects.

An entity is a thing or object or physical construct. An employee, a student, a product are considered as entities. A relationship is an association of several entities.

**Object Oriented Data Model:**

Like E-R data model, the object oriented data model is also based on a collection of objects. An object is an instance, which holds a set of values within itself. An object may contain the body of the operations that work on the data. The bodies are known to be methods. Unlike E-R data model, each object has its own unique identity, independent of the values that it contains.

**Record Based Data model:**

Record based data models are also used for describing data at logical level and view level as well. These data models are used to specify the overall structure of the database and to provide a higher-level implementation. Record based data models, as they are named, maintain fixed format records of several types.



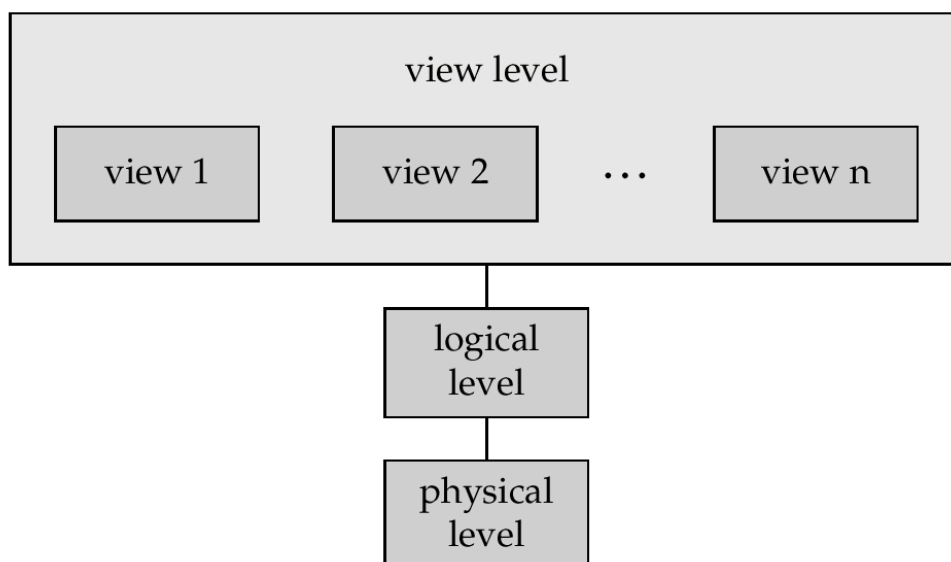
## 1.7. Views Of Data:

A Database is a collection of interrelated files and set of programs that allow users to access and modify these files. The main objective of three-schema architecture is to separate database from application programs.

A commonly used view of data approach is the three-level architecture suggested by ANSI/SPARC (American National Standards Institute/Standards Planning and Requirements Committee). Under this approach, a database is considered as containing data about an *enterprise*.

The three levels of the architecture are three different views of the data. The design of each such level is considered as a schema and hence the whole design is referred as three-schema architecture. With the three schemas, program data independency can corresponding operation independency can be achieved. This process is also referred to as data abstraction.

- Internal Level or Internal Schema or Physical Level.
- Conceptual Level or Conceptual Schema or Logical Level.
- External Level or External Schema or View Level.



### Three Levels Of Data Abstraction

The three level . . . . . ie information meaning (conceptual view) from the external data representation and from the physical data structure layout.

#### External Schema:

The highest level of data abstraction is nothing but external schema. It describes only a part of the entire database. It is basically presentation level. Several users of the database have their own view of data. Each has a separate design of approach. In general, the end users and even the application programmers are only interested in a subset of the database. For example, a department head may only be interested in the departmental finances and student enrolments but not the library information. The librarian would not be expected to have any interest in the information about academic staff. The payroll office would have no interest in student enrolments.

**Improved Data Accessibility and Responsiveness:**

End users without programming experience can retrieve and display data (using SQL).

**Reduced Program Maintenance:**

Data are independent of the application programs that use them, and either one can be changed without a change in the other.

**Data Integrity:**

The term **data integrity** refers to the degree to which data is accurate and reliable. **Integrity Constraints** are rules that all data must follow. For example if month is a field, then a number greater than 12 is invalid. Similar examples are number of days in a month, number of hours in a day, etc. Other invalid values could be pay rates, temperatures (too high or too low) etc.

**1.6.2. Disadvantages of the Database Approach:****New, Specialized Personnel:**

New individuals need to be hired and/or trained, and frequently retrained or upgraded to implement databases.

**Installation, Management Cost and Complexity:**

A multi-user DBMS is a large and complex suite of software that has a high initial cost and require a staff of trained personnel to install and operate. A substantial annual maintenance and support costs are needed. Hardware and Data Communications systems may need upgrading. Security software is often required to ensure proper concurrent of shared data.

**Conversion Costs:**

Old file processing system converted to modern database technology will cost money and time.

**Need for Explicit Backup and Recovery:**

Data may be damaged or destroyed, due to hardware failure, physical damage caused by fires or floods, and software or human errors. A **backup**, or copy, must be made periodically. DBMS's include backup routines or rely on system utilities. **Recovery** is replacing the damaged database with good backup. Users have to reenter data of any transactions lost since the last backup

**Organizational Conflict:**

Conflicts on data definitions, data formats and coding, rights to update shared data, and associated issues are difficult to resolve.

**Security:**

In addition to User ID and password, specific privileges can be assigned to each user, defining that user's access to the data. **Read-only privilege** permits that user only to look at the data; no changes are allowed. **Update privilege** allows the user to make changes to the data. DBMS has privileges at the field level; a user may be able to change some fields, just look at others, and not even see some fields.

Client/server computing and Internet applications became important in the 1990's. Multimedia data, including graphics, sound, images, and video also became more common, and so object-oriented databases were introduced. Combination of relational and object-oriented databases known as object-relational databases are now available. In the future multidimensional data will become more important.

## 1.6. Database Approach:

In traditional file processing each user defines and implements the files needed for specific application as part of the programming application. In database approach a single repository of data is maintained and accessed by various users. It emphasizes the integration and sharing of the data through out the organization.

### 1.6.1. Advantages Of The Database Approach:

#### Program-Data Independence:

The separation of data description (metadata) from the application programs that use the data leads to **data independence**. Data descriptions are stored in a central location called the *repository*. Organization's data can change and evolve without changing the application programs that process the data.

#### Minimal Data Redundancy:

Traditionally, information systems were developed using a file-processing approach. Each application had its own files, and data was not shared among applications, resulting in a great deal of **data redundancy**, or repetition of the same data value.

The database approach was developed to minimize data redundancy by creating separate files for each entity. Files are referred to as **tables**, and a **database** is a collection of related tables. Data files are integrated into a single logical structure. While not completely eliminating redundancy, the designer can control the type and amount of redundancy.

#### Improved Data Consistency:

Obtained by reducing redundancy. Updating data values is simplified, as each value is stored in one place only. Storage is not wasted.

#### Improved Data Sharing:

Database is designed as a shared resource. Authorized users are granted permission to use the database, and provided with user views to facilitate this use.

#### Improved Productivity of Application Development:

Reduction of the cost and time for developing new business applications. The programmer can concentrate on the specific functions required for the new application and DBMS provides a number of high-level productivity tools such as forms and report generators and high-level languages that automate some of the activities of database design and implementation.

#### Enforcement of Standards:

Standards include naming conventions, data quality standards, and uniform procedures for accessing, updating, and protecting data.

**Atomicity Problem:**

In general a transaction is atomic, i.e., it must be either completely done or undone. As the computer system is an electronic device, it may be subject to fail sometimes. If a failure occurred during the execution of a transaction, it may lead to data inconsistency. For example, consider bank transaction to transfer an amount of Rs.1000 from account A to account B. If a failure occurred in the middle, it may be possible that Rs.1000 may be removed from account A and was not credited to account B. Clearly, we say that while transferring the amount both credit and debit are to be done simultaneously.

**Concurrent Access Anomalies:**

Concurrent access may be done to the same transactions in multi-user environments. It may again lead to inconsistency. For example, consider bank account A containing Rs.5000. If two customers withdraw funds (say Rs.1000 and Rs.2000 respectively) from the account at the same time, the transaction may leave incorrect data. The system may show the same balance Rs.5000 to both the customers and they may feel that they can withdraw a maximum amount of Rs.4500 leaving the remaining amount as minimum balance. It may lead to inconsistent data.

**Security Problem:**

The entire database must not be available to all the database users. If the access is provided, improper and illegal operations may be performed over the database, which in turn leave inconsistent data. Hence certain security measures like individual user and their respective passwords are to be imposed over the database.

**1.4. What is Database Management System [DBMS] :**

A Database Management System (DBMS) is a software package to facilitate the creation and maintenance of a computerized database. A Database System (DB) is a DBMS together with the data itself.

(Or)

A Database Management system consists of a collection of interrelated data and a set of programs to access those data. The collection of data usually referred as database. Here a database holds information regarding an enterprise.

(Or)

A Database Management System is a general-purpose software system that facilitates the processes defining, constructing and manipulating databases for various applications.

(Or)

A Database Management System is a computerized record keeping system that lets the user to perform various operations over the database.

**1.5 Evaluation of Database Systems :**

DBMS were first introduced during the 1960's. This was called "proof-of-concept" period in which the feasibility of managing vast amounts of data with DBMS was demonstrated.

The DBMS became a commercial reality in the 1970's. The Hierarchical and network models were introduced in this decade. The relational model was first defined by E. F. Codd an IBM research fellow in 1970, and became commercially successful in the 1980's.

Database applications are widely used. Here are some representative applications:

- ❑ Banking
- ❑ Airlines
- ❑ Universities
- ❑ Credit card Transactions
- ❑ Telecommunications
- ❑ Finance
- ❑ Sales
- ❑ Manufacturing
- ❑ Human Resources.

### **1.3. Traditional File Processing System :**

In the beginning of computer-based data processing, there were no databases. To be useful for business applications, computers must be able to store, manipulate, and retrieve large files of data. So, an organization's information was stored as groups of records in separate files. Computer file processing systems were developed for this purpose.

A file is a collection of records. A record in turn is a collection of several interrelated data items. In early days, user data is managed in terms of physical files in disks.

#### **1.3.1 Disadvantages of File Processing System :**

The file processing system has several disadvantages. They are:

##### **Data Redundancy:**

Data redundancy means duplication of data. As the data may be stored in several files, it may be repeated in multiple files, which leads to memory wastage and access cost. It in turn leads to data inconsistency, i.e., in various copies of the same data, one updating may lead to the necessary changes in all the remaining copies. It becomes tedious for the user.

##### **Difficulty In Accessing Data:**

As no special application programs are available at that time, it becomes tedious for the user to access the data in this system. In other words we can say that the conventional file processing system does not allow us to access needed data in convenient and efficient manner.

##### **Data Isolation:**

Because the data are scattered over the memory in terms of various files, and the files may be in various format, it is difficult to write new application programs to retrieve the necessary data.

##### **Integrity Problems:**

Data validity is the most vital aspect in DBMS. To check the validity of the data, certain consistency constraints are to be imposed. Such constraints are difficult to be enforced in traditional file processing system. For example, salary of an employee should not be less than or equal to zero

## 1.1 INTRODUCTION

Any organization uses a computer to store and process information because it hopes for speed, accuracy, efficiency, economy etc., beyond what could be achieved using clerical methods. The objectives of using a Database Management System (DBMS) must in essence be the same although the justifications may be more indirect. Early computer applications were based on existing clerical methods and stored information was partitioned in much the same way as manual files. But the computer's processing speed gave a potential for RELATING data from different sources to produce valuable management information, provided that some standardization could be imposed over departmental boundaries.

### Data:

Data refers to facts, symbols, events, entities, variable, names or any other, which has little meaning.

Database that contains the facts like Faculty Name, City, College etc,

For Ex :

Suresh            Guntur            JKC College

The data may contain facts, text, images, sound and video segments.

### Information:

Processed data is known as Information.

### Database:

It is an organized collection of logically related data. The data are structured so as to be easily stored, manipulated and retrieved by users. For better retrieval and sorting, each record is usually organized as a set of data elements (facts). The items retrieved in answer to queries become information that can be used to make decisions.

For example a student may maintain a small database, which includes contacts like student number, name, address, phone no etc, in his computer.

### Meta Data:

Meta Data describes the structure of the primary database and it also describes the properties or characteristics of data. These properties may include data definitions, data structures, rules or constraints.

(Or)

Data about the data. The metadata describe the properties of data, but not include that data.

## 1.2. Database System Applications:

Database System Application is an application program that is used to perform a series of database activities on behalf of database users.

The basic operations or activities are:

- CREATE
- READ
- UPDATE
- DELETE

## **Lesson 1**

# **INTRODUCTION**

### **1.0 Objectives:**

The major objective of this lesson is to provide a strong formal foundation in basic database concepts and technology.

After reading this chapter, you should understand:

- To define data, information and database.
- To define database management system and structure.
- To give an introduction to conventional data processing and database management system.
- Systematic database design approaches covering conceptual design, logical design and an overview of physical design.
- To describe various database models.
- To introduce the concepts of transactions and transaction processing.

### **Structure of The Lesson:**

- 1.1. Introduction**
- 1.2. Database System Applications**
- 1.3. Traditional File Processing System**
  - 1.3.1. Disadvantages of File Processing System**
- 1.4. What is Database Management System [DBMS]**
- 1.5. Evaluation Of Database Systems**
- 1.6. Database Approach**
  - 1.6.1. Advantages of the Database Approach**
  - 1.6.2. Disadvantages of the Database Approach**
- 1.7. Views Of Data**
- 1.8. Data Models**
- 1.9. Database Languages**
- 1.10. Database Users**
- 1.11. Database Administrator**
- 1.12. Transaction Management**
- 1.13. Database System Structure**
- 1.14. Summary**
- 1.15. Technical terms**
- 1.16. Model Questions**
- 1.17. References**

## Lesson 2

# Entity Relationship Model-1

## 2.0 Objectives

The main objective of this lesson is to develop the skills necessary for the design and evaluation of database management systems based on the Entity-Relational database model.

After reading this chapter, you should understand :

- What is an entity ?
- What is a Relation ?
- Various types of Attributes
- Relationship sets
- Mapping Cardinalities

### Structure of the Lesson:

- 2.1 Basic Concepts
- 2.2 Entity Sets
- 2.3 Relationship sets
- 2.4 Mapping Constraints
- 2.5 Summary
- 2.6 Technical Terms
- 2.7 Model Questions
- 2.8 References

## 2.1 Basic Concepts:

The Entity-Relationship (ER) model was originally proposed by Peter in 1976. A basic component of the model is the Entity-Relationship diagram, which is used to visually represent data objects. The entity-relationship model can be used as a basis for unification of different views of data: the network model, the relational model and the entity set model. Entity-Relationship (ER) modeling is an important step in information system design and software engineering.

The **E-R entity-relationship** model is a detailed, logical representation of the data for an organization or for a business area. The E-R model is expressed in terms of entities in the business environment, the relationships among the entities, and the attributes of both the entities and the relationship.

The E-R data model employs three basic notations: entity sets, relationship sets, and attributes.



## 2.2 Entity Sets

An entity is a person, place, object event or concept in the user environment about which the organization wishes to maintain data.

(Or)

The **Entity-Relationship (ER) model**, a high-level data model that is useful in developing a conceptual design for a database.

(Or)

An **entity** is an object that exists and is distinguishable from other objects.

For instance, “ramu” with Faculty-ID 3456 is an entity, as he can be uniquely identified as one particular Faculty in the College.

Person : EMPLOYEE, STUDENT, PATIENT

Place : STORE, WAREHOUSE, STATE

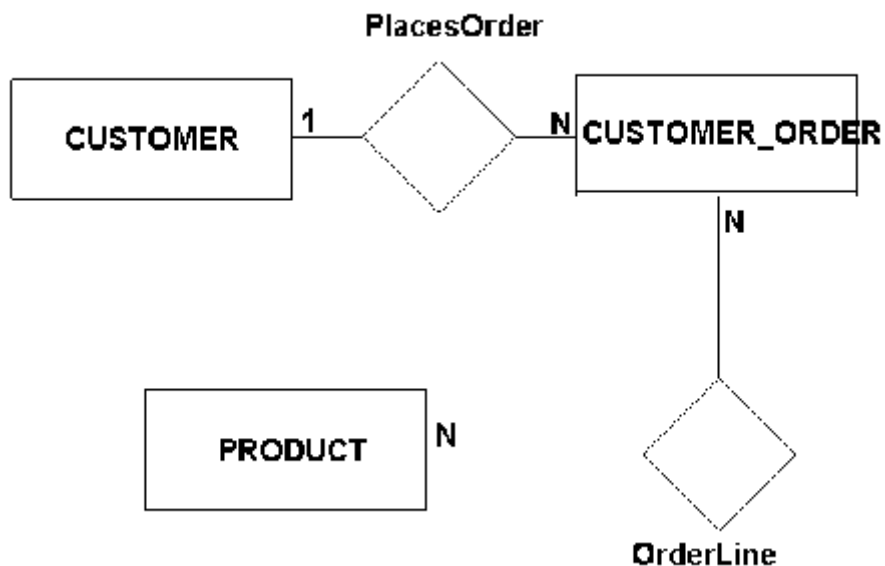
Object : MACHINE, BUILDING, AUTOMOBILE

Event : SALE, REGISTRATION, RENEWAL

Concept: ACCOUNT, COURSE, WORK CENTRE

Consider the example entities shown in below figure:

**CUSTOMER\_ORDER, CUSTOMER, PRODUCT.**



Distinguish between an **entity** and **instance** (or **occurrence**) of an entity for each of the entities shown in this example.

*Entities* are the principal data object about which information is to be collected. Entities are usually recognizable concepts, either concrete or abstract, such as person, places, things, or events, which have relevance to the database.

Entities are classified as independent or dependent. An *independent entity* is one that does not rely on another for identification. A *dependent entity* is one that relies on another for identification.

An *entity occurrence* (also called an instance) is an individual occurrence of an entity. An occurrence is analogous to a row in the relational table.

Entity sets is a set of entities of the same type that share the same properties, or attributes. The set of all persons who are students at a given Institute can be defined as the entity set student.

The entity-relationship model is based on a perception of the world as consisting of a collection of basic **objects** (entities) and **relationships** among these objects.

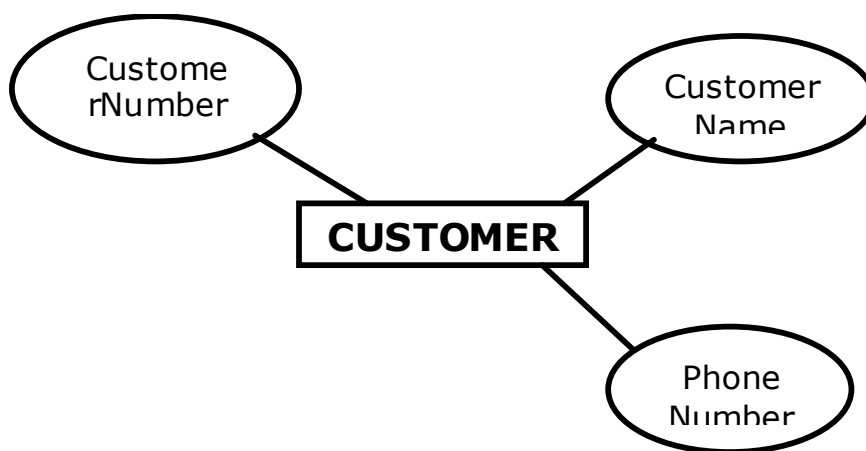
### Attribute:

Attributes are also termed Properties. Attributes or Properties are characteristics of an entity. Examples: CustomerNumber, OrderNumber, OrderDate, ProductNumber.

Attributes describe the entity of which they are associated. A particular instance of an attribute is a value.

The domain of an attribute is the collection of all possible values an attribute can have. The domain of Name is a character string.

In the example shown below, attribute names are shown as a combination of upper and lower case characters inside bubbles. Here customer is an entity. Customer Name, Customer Number and Phone Number are properties or attributes



The following attribute types, as used in the E-R model can characterize an attribute.

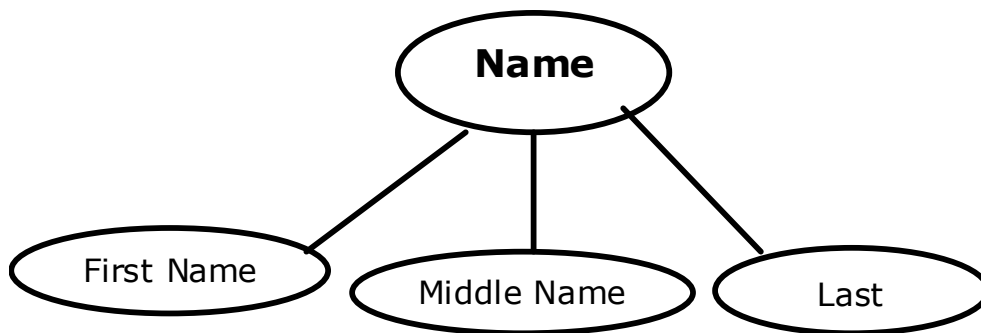
### Simple Attributes:

An Attribute is one that cannot be divided into sub parts. (Or) A **simple attribute** is one component that is atomic.

### Composite Attribute:

A **composite attribute** has multiple components, each of which is atomic or composite. (Or)

An attribute can be broken down into component parts. The most common example is Name, which can usually be broken down into the First Name, Middle Name, and Last Name.



Another example is the attribute, Address, which can be broken down into Street, City, State, and ZipCode.

### Single Valued Attributes:

An entity attribute that holds exactly one value is a **single-valued attribute**.

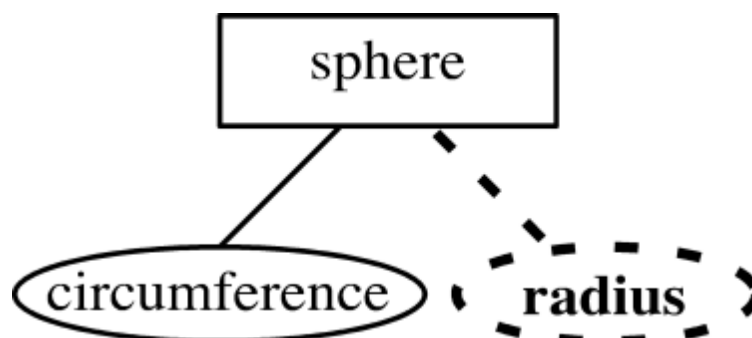
### Multivalued Attributes:

A multivalued attribute is an attribute that may take on more than one value for a given entity instance. For example, the employee entity type in given picture has an attribute name Skill, whose values record the skill (or skills) for that employee.

### Derived Attributes:

An attribute whose values can be calculated from related attribute values. (Or) A **derived attribute** can be obtained from other attributes or related entities

For example, the radius of a sphere can be determined from the circumference.



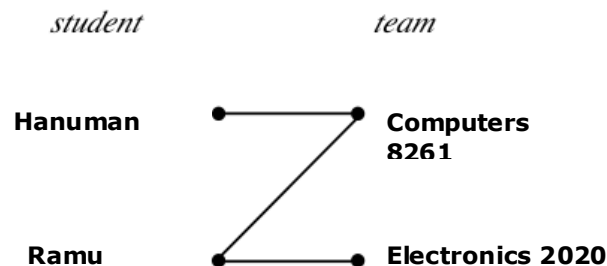
Another Example, you store the STUDENT **DateOfBirth** attribute. The **Age** attribute can be computed by subtracting the current **System Date** from the **DateOfBirth** attribute.

## 2.3. Relationship Sets:

A **relationship type** is a set of associations among entity types.

For example, the *student* entity type is related to the *team* entity type because each student is a

member of a team. In this case, a **relationship** or **relationship instance** is an ordered pair of a specific student and the student's particular Computers team, such as (Hanuman, Computers), where computers 8261 is Hanuman's team number.



We arrange the diagram so that the relationship reads from left to right, “a student is a member of a team”. Alternatively, we can arrange the components from top to bottom



**ER diagram notation for relationship type, *MemberOf***

A **relationship set** is a set of relationships of the same type.

**Formally** it is a mathematical relation on  $n \geq 2$  (possibly non-distinct) sets. If  $E_1, E_2, \dots, E_n$  are entity sets, then a relationship set R is a **subset** of

$\{(c_1, c_2, \dots, c_n) \mid c_1 \in E_1, c_2 \in E_2, \dots, c_n \in E_n\}$  where  $(c_1, c_2, \dots, c_n)$  is a relationship.

For example, consider the two entity sets *customer* and *account*. We define the relationship *CustAcct* to denote the association between customers and their accounts. This is a **binary** relationship set. Going back to our formal definition, the relationship set *CustAcct* is a subset of all the possible customer and account pairings.

This is a binary relationship. Occasionally there are relationships involving more than two entity sets.

The **role** of an entity is the function it plays in a relationship. For example, the relationship *works-for* could be ordered pairs of *employee* entities. The first employee takes the role of manager, and the second one will take the role of worker.

A relationship may also have **descriptive** attributes. For example, *date* (last date of account access) could be an attribute of the *CustAcct* relationship set.

## 2.4. Mapping Constraints:

An E-R scheme may define certain constraints to which the contents of a database must conform.

A **mapping cardinality** is a data constraint that specifies how many entities an entity can be related to in a relationship set.

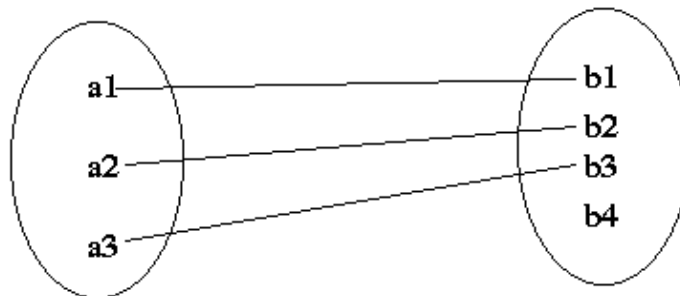
Example: A student can only work on two projects, the number of students that work on one project is not limited.

A **binary relationship set** is a relationship set on two entity sets. Mapping cardinalities on binary relationship sets are simplest.

Consider a binary relationship set R on entity sets A and B. There are four possible mapping cardinalities in this case:

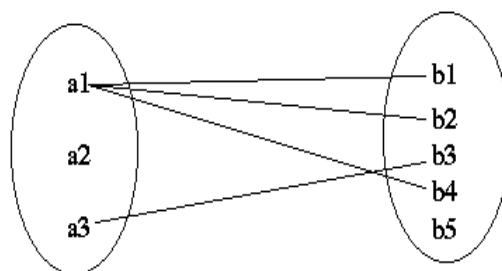
### One-to-One:

An entity in A is related to at most one entity in B, and an entity in B is related to at most



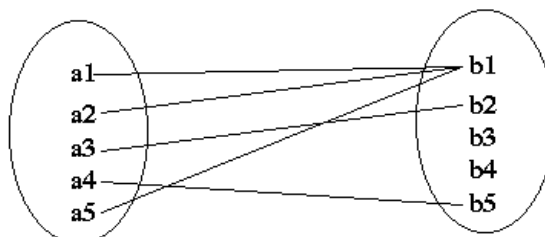
### One-to-Many:

An number of entity in B is related to at most one entity in A, and an entity in B is related to at most



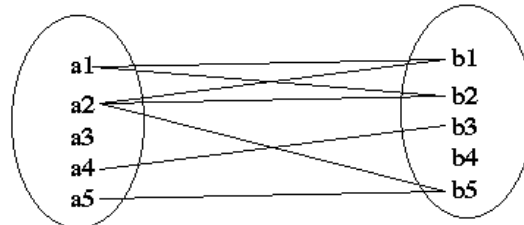
### Many-to-One:

An entity in A is related to at most one entity in B, but an entity in B is related to any number of entities in A.



**Many-to-Many:**

An entity in A is related to any number of entities in B, but an entity in B is related to any number of entities in A.



The appropriate mapping cardinality for a particular relationship set depends on the real world being modeled.

The Entity-Relationship Model is a conceptual data model that views the real world as consisting of entities and relationships. The model visually represents these concepts by the Entity-Relationship diagram. The basic constructs of the ER model are entities, relationships, and attributes. Entities are concepts, real or abstract, about which information is collected. Relationships are associations between the entities. Attributes are properties, which describe the entities. Next, we will look at the role of data modeling in the overall database design process and a method for building the data model. An attribute may be simple, composite, single valued, multivalued and derived attribute.

**2.6. Technical Terms:**

**Entity:** An **entity** is an object that exists and is distinguishable from other objects.

**Attribute:** Attributes describe the entity of which they are associated. A particular instance of an attribute is a value.

**Domain:** The domain of an attribute is the collection of all possible values an attribute can have.

**Simple attribute:** An Attribute that cannot be divided into sub parts.

**Composite Attribute:** A composite attribute has multiple components, each of which is atomic or composite.

**Multivalued Attribute:** A multivalued attribute is an attribute that may take on more than one value for a given entity instance.

**Relationship Type:** Relationship type is a set of associations among entity types.

**Mapping Cardinality:** A mapping cardinality is a data constraint that specifies how many entities an entity can be related to in a relationship set.

**2.7. Model Questions:**

1. What is an attribute? Explain various types of attribute?
2. Explain Entity set and Relationship set?
3. Write about mapping Cardinalities?

## **2.8. References:**

**Database System Concepts** Silberschatz, Korth, and Sudarshan

**Database Management Systems** Arun K. Majumdar, Pritimoy Bhattacharyya

**An Introduction to Database Systems** Bipin Desai

**Modern Database Management** F. McFadden, J. Hoffer

**An Introduction to Database Systems** C. J. Date;

**AUTHOR:**

**Y.SURESH BABU.**, M.Com, M.C.A.,  
**Lecturer, Dept. Of Computer Science,**  
**JKC College. GUNTUR.**

## Lesson 3

# Entity Relationship Model-2

### 3.0 Objectives:

The main objective of this lesson is to introduce the concept of keys and symbols for drawing E-R diagrams.

After reading this chapter, you should understand:

- What is a key?
- Super, candidate and primary keys
- E-R notations
- Generalization and Specialization
- Alternative E-R notations

### Structure of the Lesson:

- 3.1. Keys
- 3.2. Entity – Relationship Diagram
- 3.3. Weak and Strong Entity Sets
- 3.4. Specialization
- 3.5. Generalization
- 3.6. Attribute Inheritance
- 3.7. Aggregation
- 3.8. Alternative E-R Notations
- 3.9. Summary
- 3.10. Technical terms
- 3.11. Model Questions
- 3.12. References

### 3.1. Keys:

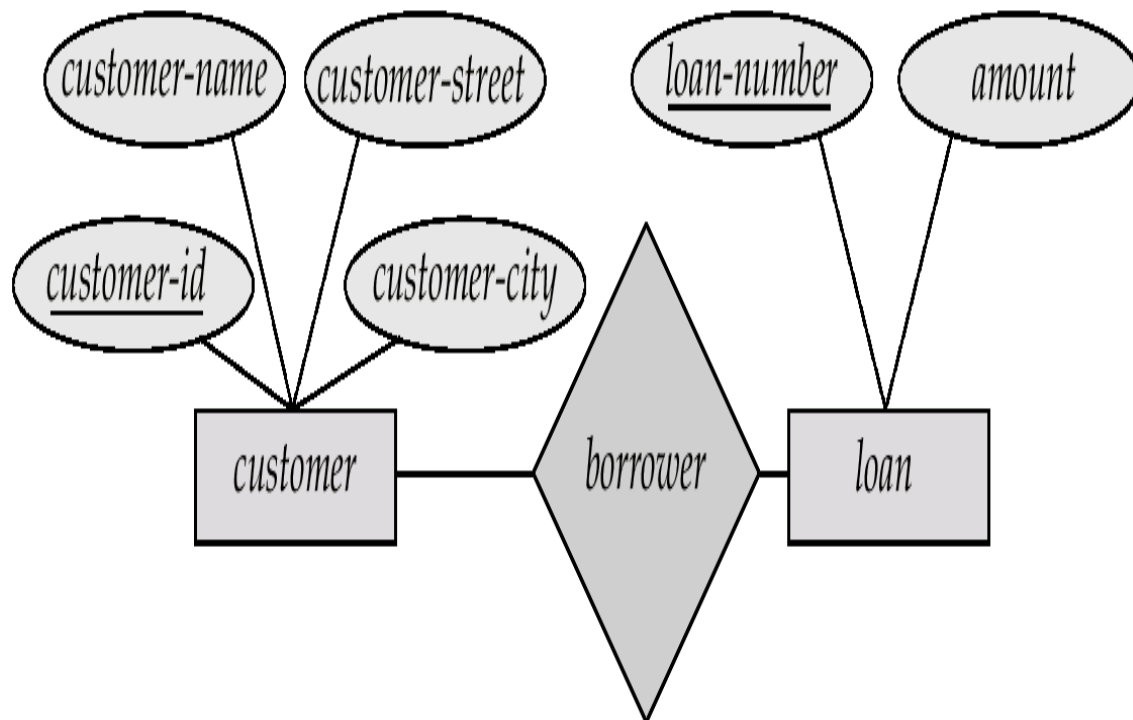
A key is an attribute or a combination of attributes, which identifies the record set uniquely. A key which contains more than one attribute is called composite key.

#### Super Key:

A **superkey** is a set of one or more attributes which taken collectively, allow us to identify uniquely an entity in the entity set

For example, in the entity set *customer*, *customer-name* and *S.I.N.* is a superkey. Note that *customer-name* alone is not, as two customers could have the same name. A superkey may contain extraneous attributes, and we are often interested in the smallest superkey.





A **Candidate Key** is an attribute or set of attributes that uniquely identify an instance of an entity, e.g., a student could be identified by their Social Security Number or by an assigned Student Identification Number. (Or)

A superkey for which no subset is a superkey is called a **candidate key**.

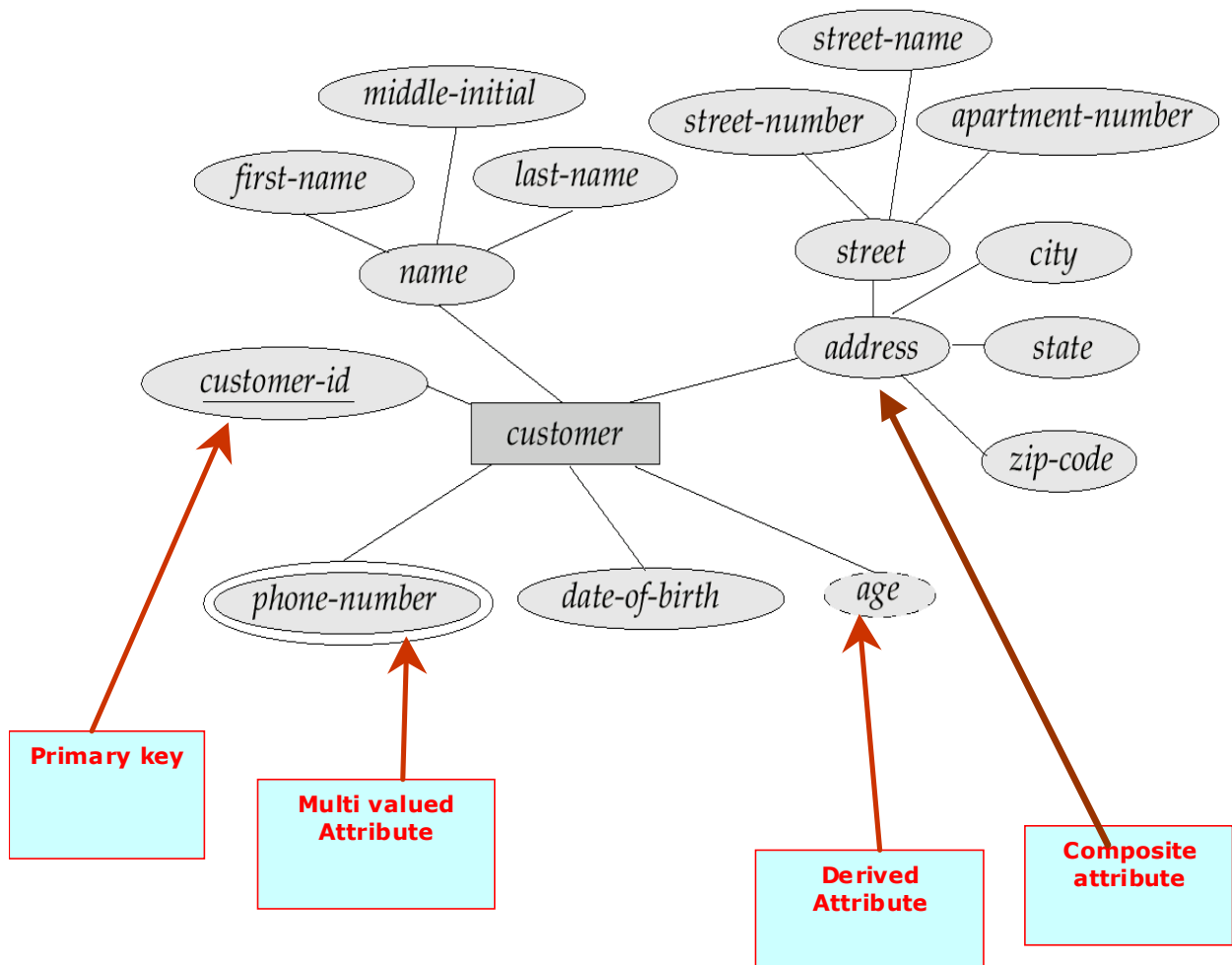
Some entities have more than one candidate key. Sometimes it is necessary to combine attributes to form a **composite key** out of two or more attributes.

In the above example customer\_id, loan\_number are candidate keys.

### Primary Key:

A **primary key** is a candidate key (there may be more than one) chosen by the Data Base designer to identify entities in an entity set.

A primary key is usually a single attribute, but may be a composite key (combination of two or more attributes). The primary key selection must follow certain criteria.



E-R Diagram With Composite, Multi-valued, and Derived Attributes:

### 3.2. Entity Relationship Diagram:

**Rectangles** represent entity sets.

**Diamonds** represent relationship sets.

**Lines** link attributes to entity sets and entity sets to relationship sets.

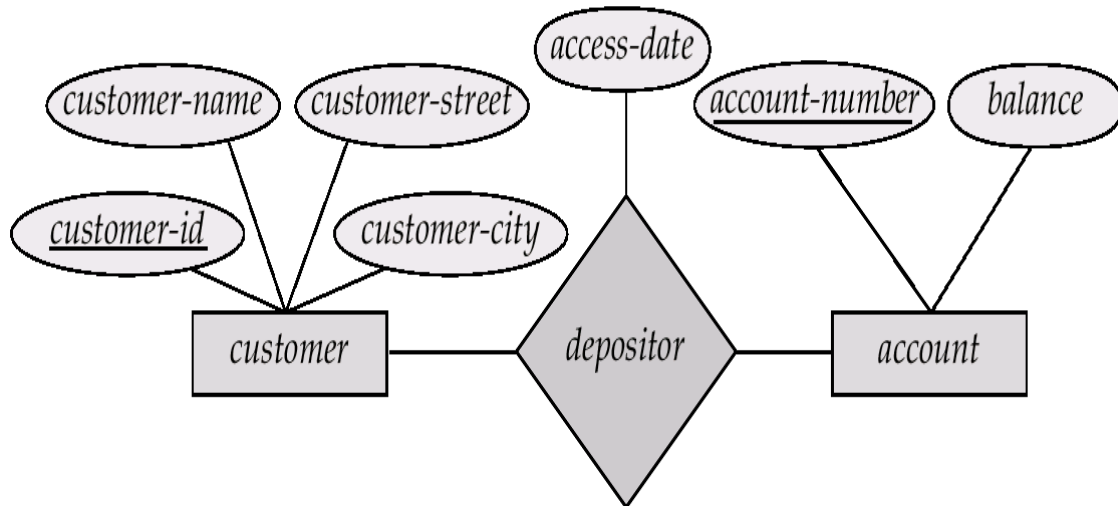
**Ellipses** represent attributes

**Double ellipses** represent *multi-valued* attributes.

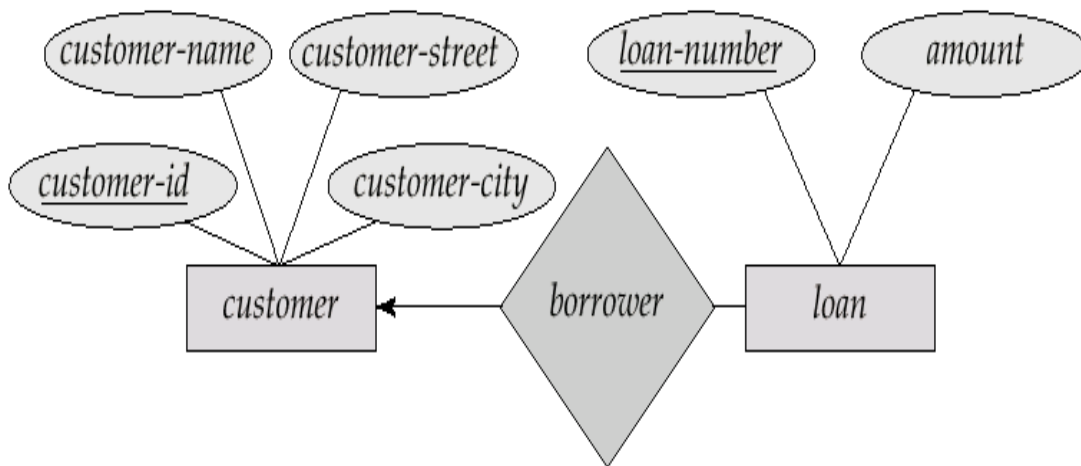
**Dashed ellipses** denote *derived* attributes.

**Underline** indicates primary key attributes.

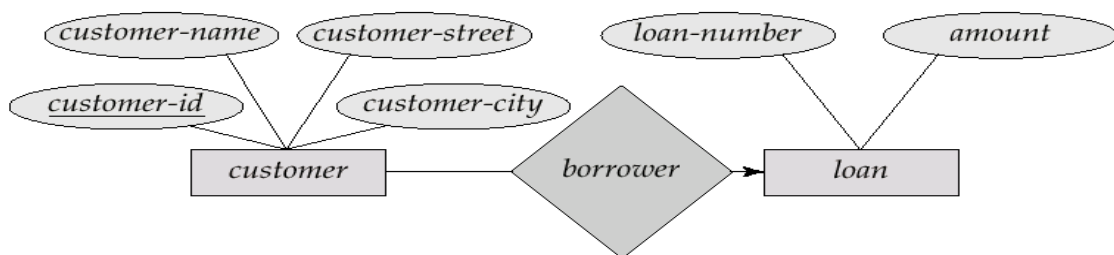
### An E-R diagram



### One-to-many from *customer* to *account*

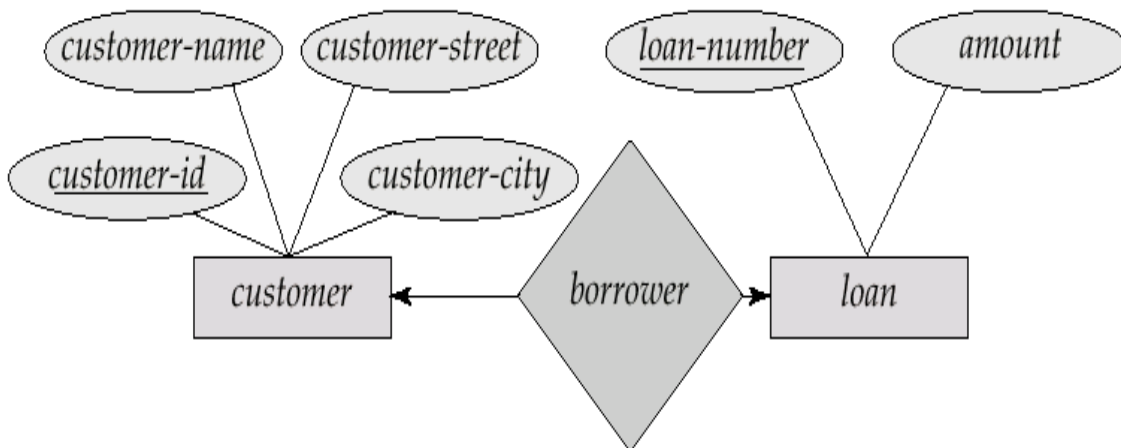


### Many-to-one from *customer* to *account*



**One-to-one from *customer* to *account*:**

We express cardinality constraints by drawing either a directed line ( $\rightarrow$ ), signifying “one,” or an undirected line ( $-$ ), signifying “many,” between the relationship set and the entity set. A customer is associated with at most one loan via the relationship *borrower*; a loan is associated with at most one customer via *borrower*



Go back and review mapping cardinalities; they express the number of entities to which an entity can be associated via a relationship. The arrow positioning is simple once you get it straight in your mind, so do some examples. Think of the arrowhead as pointing to the entity that “one” refers to.

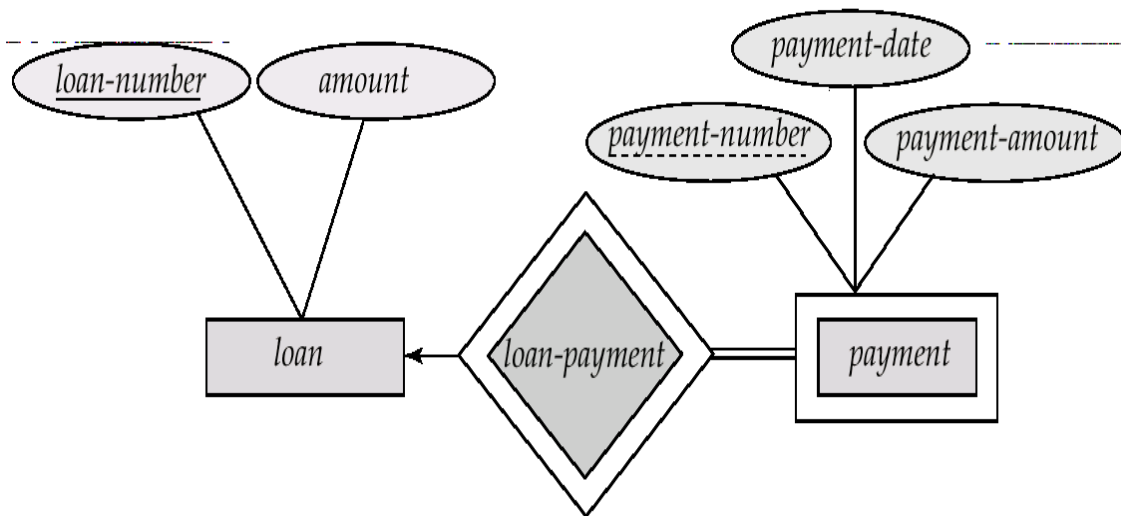
**3.3. Weak Entity and Strong Entity Sets:**

An entity set that does not possess sufficient attributes to form a primary key is called a **weak entity set**. One that does have a primary key is called a **strong entity set**.

A weak entity set does not have a primary key, but we need a means of distinguishing among the entities.

The **primary key of a weak entity set** is formed by taking the primary key of the strong entity set on which its existence depends, plus its **discriminator**.

Consider the entity set payment, which has the three attributes: payment-number, payment-date, and payment-amount. Payment numbers are typically sequential numbers, starting from 1, generated separately for each loan. Thus although each payment entity is distinct, payments for different loans may share the same payment number. Thus, this entity set does not have a primary key. It is a weak entity.



### 3.4. Specialization:

An entity set may include subgroups of entities that are distinct in some way from other entities in the set. For instance, a subset of entities set may have attributes that are not shared by all the entities in the entity set. The E-R model provides a means for representing these distinctive entity groupings.

Consider a set, a person with attributes name, street, and city. A person may be further divided into one of the following:

- \* Customer
- \* Employee

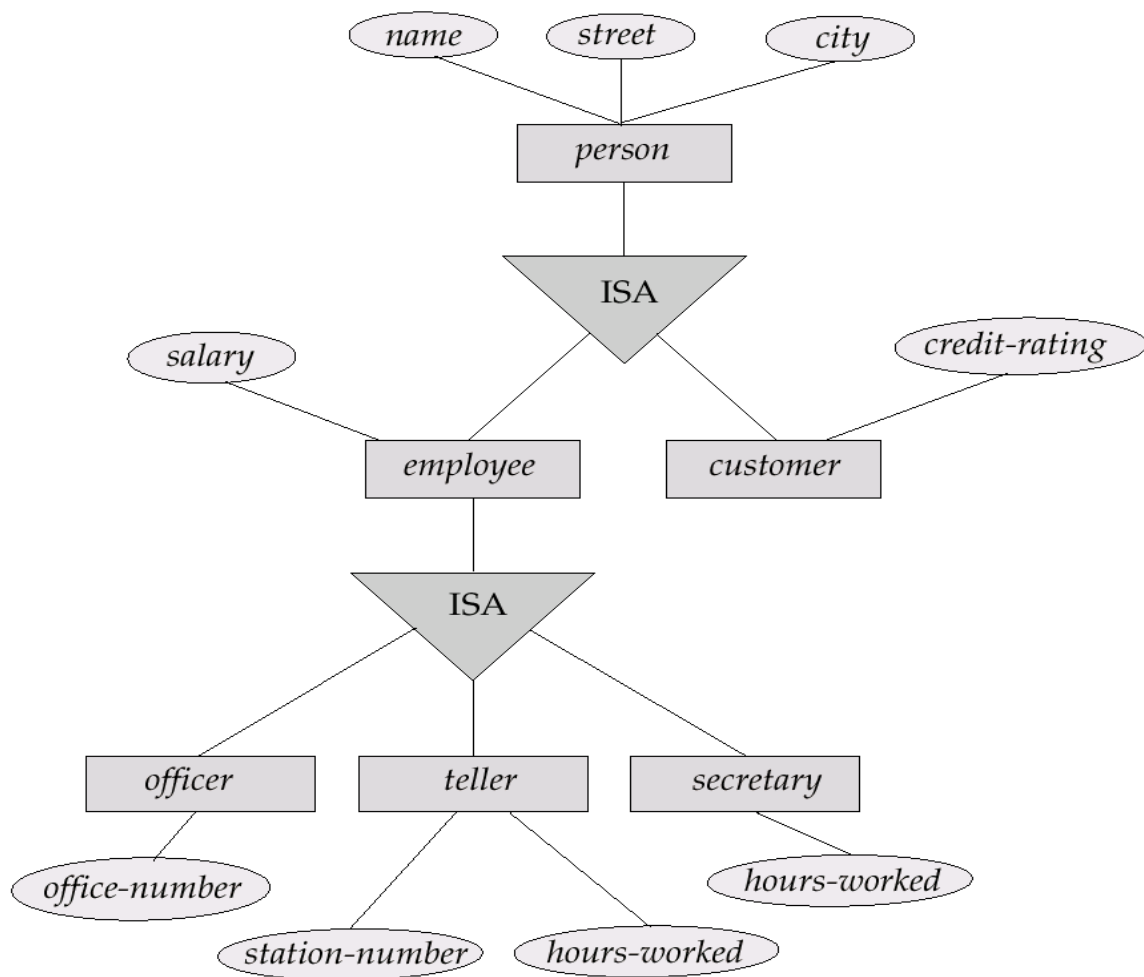
Each of these person types is described by a set of attributes that includes all the attributes of each entity set person plus possibly additional attributes. For example customer entities may be described further by the attribute customer\_id, whereas employee entities may be described further by the attribute employee\_id and salary. The process of designing sub-groupings within an entity set is called **specialization**. The specialization of person allows us to distinguish among persons according to whether they are employees or customers.

### 3.5. Generalization:

GENERALIZATION is the reverse process (to specialization) in which we suppress the differences among several entity types, identify their common features and generalize them into a single super class, of which the original entity types are subclasses.

There are some similarities between the customer entity set and the employee entity set in the sense that they have several attributes in common. This commonality can be expressed by generalization, which is containment relationship that exists between a higher-level entity set and one or more lower level entity sets.

In E-R diagrams, specialization and generalizations, as shown in the figure below.



### Specialization And Generalization

#### 3.6. Attributes Inheritance:

A crucial property of the higher and lower-level entities created by specialization and generalization is attribute inheritance. The attributes of the higher-level entity sets are said to be inherited by the lower level entity sets. For ex: Customer and employee inherit the attributes of person.

#### 3.7. Aggregation:

Aggregation is an abstraction through which relationships are treated as higher-level entities.

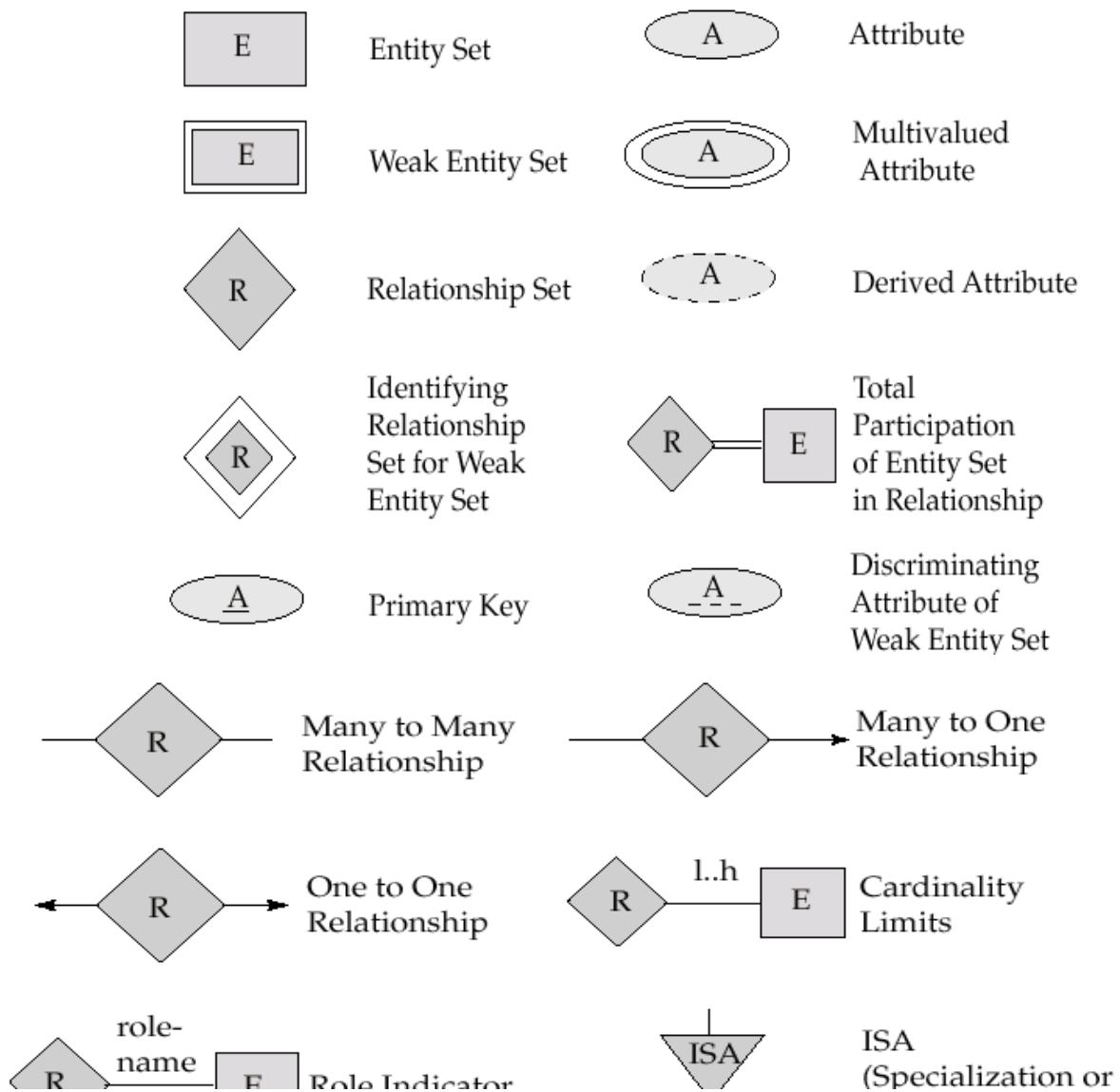
#### 3.8. Alternative E-R Notations:

There is no universal standard for E-R diagrams notation, and different books and E-R diagram notations. The following figure shows some of the alternative notations that are widely used.

### 3.9. Summary:

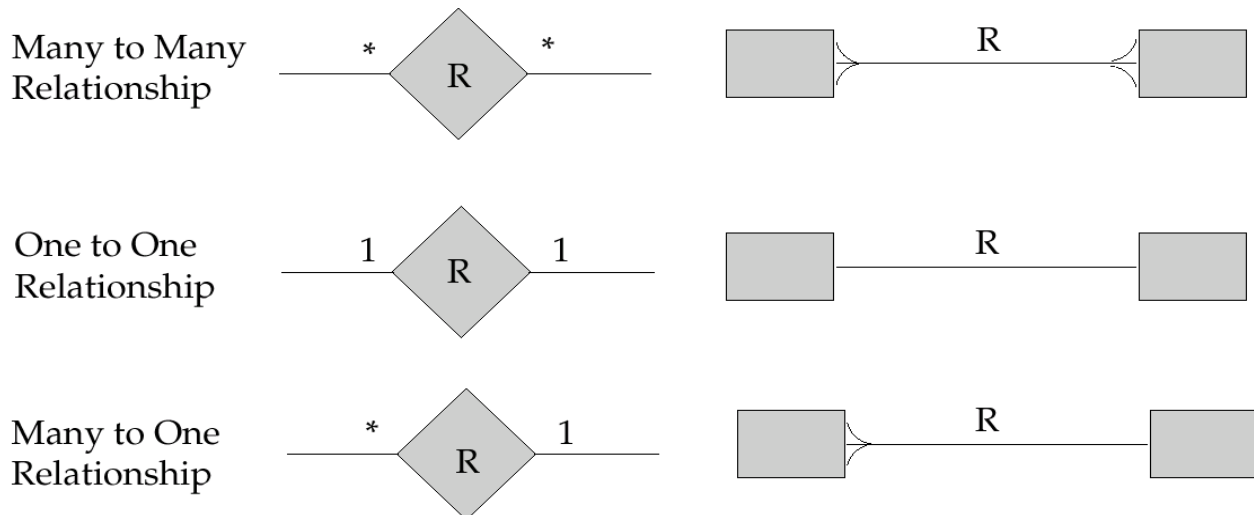
Superkey is a set of one or more attributes which, taken collectively, allow us to identify uniquely an entity in the entity set. For example, in the entity set *customer*, *customer-name* and *S.I.N.* is a superkey. A superkey may contain extraneous attributes, and we are often interested in the smallest superkey. A superkey for which no subset is a superkey is called a candidate key. A primary key is a candidate key (there may be more than one) chosen by the Database designer to identify entities in an entity set.

We can express the overall logical structure of a database graphically with an E-R diagram. Its



Entity set E with  
attributes A1, A2, A3  
and primary key A1

E
A1
A2
A3



components are: rectangles representing entity sets, Ellipses representing attributes and Diamonds representing relationship sets. Lines linking attributes to entity sets and entity sets to relationship sets. An entity set that does not possess sufficient attributes to form a primary key is called a weak entity set. One that does have a primary key is called a strong entity set. Generalization is the reverse process (to specialization) in which we suppress the differences among several entity types, identify their common features and generalize them into a single super class, of which the original entity types are subclasses.

### 3.10. Technical Terms:

**Relationship:** Association between two entities in an ERD. Each end of the relationship shows the degree of how the entities are related and the optionality.

**Generalization:** It is the reverse process (to specialization) in which we suppress the differences among several entity types, identify their common features and generalize them into a single super class, of which the original entity types are subclasses.

**Aggregation:** Aggregation is an abstraction through which relationships are treated as higher-level entities.

**Weak Entity:** An entity set that does not possess sufficient attributes to form a primary key is called a weak entity set.

**Strong Entity:** An entity that does have a primary key is called a strong entity set.



### 3.11. Model Questions:

1. Explain the distinction among the terms of primary key, candidate key and super key.
2. Define the concept of Aggregation?
3. Explain the difference between the strong entity and weak entity?
4. Write about the alternative E- R notations?
5. Explain the concept of Generalization and Specialization?

### 3.12. References:

#### **Database System Concepts**

Silberschatz, Korth, and Sudarshan

#### **Database Management Systems**

Arun K. Majumdar, Pritimoy Bhattacharyya

#### **An Introduction to Database Systems**

Bipin Desai

#### **Modern Database Management**

F. McFadden, J. Hoffer

#### **An Introduction to Database Systems**

C. J. Date;

#### **AUTHOR:**

**Y.SURESH BABU., M.Com, M.C.A.,  
Lecturer,  
Dept. Of Computer Science,  
JKC College.  
GUNTUR.**

## Lesson 4

# Relational Model

### 4.0 Objectives:

The major objective of this lesson is to provide

### Structure of the Lesson:

#### 4.1. Introduction

##### Introduction:

The relational model was formally introduced by Dr. E. F. Codd in 1970 and has evolved since then, through a series of writings. The model provides a simple, yet rigorously defined, concept of how users perceive data. The relational model represents data in the form of two-dimensional tables. Each table represents some real-world person, place, thing, or event about which information is collected.

A relational database is a collection of two-dimensional tables. The organization of data into relational tables is known as the **logical view** of the database. That is, the form in which a relational database presents data to the user and the programmer. The way the database software physically stores the data on a computer disk system is called the **internal view**. The internal view differs from product to product and does not concern us here.

##### Basic Structure:

A Relational database consists of a collection of tables, each of which is assigned a unique name. A row in a table represents a relationship among a set of values.

A **relational database** is a finite set of relation schemas (called a **database schema**) and a corresponding set of relation instances (called a **database instance**).

The relational database model represents data as a two-dimensional tables called a relations and consists of three basic components:

1. A set of domains and a set of relations
2. Operations on relations
3. Integrity rules

In the relational model, data is represented as a two-dimensional table called a *relation*. Relations have names and the columns have names called *attributes*. The elements in a column must be atomic - an elementary type such as a number, string, Date, or timestamp and from a single domain.

A relation  $r(R)$  is a mathematical relation of degree  $n$  on the domains  $\text{dom}(A_1), \text{dom}(A_2), \dots, \text{dom}(A_n)$  which is a subset of the Cartesian product of the domains that define  $R$ :

$$r(R) \subseteq (\text{dom}(A_1) \times \text{dom}(A_2) \times \dots \times \text{dom}(A_n))$$

**Example:** An employee relation is a table of names, birth dates, social security numbers, ... The contents of a relation are rarely static thus the addition or deletion of a row must be efficient.

### Database Schema:

A **database schema** is a set of relation schemas for the relations in a design. Changes to a schema or database schema are expensive thus careful thought must go into the design of a database schema.

*Relation Schema* - relationName (attribute<sub>1</sub>:dom<sub>1</sub>, ..., attribute<sub>n</sub>:dom<sub>n</sub>)

A **relation schema** e.g. employee (name, birthDate, ss#), consists of

1. The **name** of the relation. Relation names must be unique across the database.
2. The names of the *attributes* in the relation along with there associated *domain names*. An **attribute** is the name given to a column in a relation instance. All columns must be named and no two columns in the same relation may have the same name. A **domain name** is a name given to a well-defined set of values. Column values are referenced using its attribute name (A) or alternatively, the relation name followed by the attribute name (R.A)
3. The *integrity constraints (IC)*. **Integrity constraints** are restrictions on the relational instances of this schema.

### Relation Instance:

A **relation instance** is a table with rows and named columns. The rows in a relation instance (or just relation) are called **tuples**. The **cardinality** of the relation is the number of tuples in it. The names of the columns are called **attributes** of the relation. The number of columns in a relation is called the **arity** of the relation. The type constraint that the relation instance must satisfy is

1. The attribute names must correspond to the attribute names of the corresponding schema and
2. The tuple values must correspond to the domain values specified in the corresponding schema.

### Database Instance

A **database instance** is a finite set of relation instances.

Database schema example:

Movie (title, year, length, filmType)

Employee (name, birthDate, ss#, ...)

Department(deptName, empSSNo, employeeName, function)

### Keys:

It is frequently the case that within a given relation there is one attribute with values that are unique within the relation and thus can be used to identify the tuples of that relation. For example, attribute P# of the PART relation has this property-each PART tuple contains a distinct P# value, and this value may be used to distinguish that tuple from all others in the relation. P# is said to be the **primary key** for PART.

Not every relation will have a single-attribute primary key. However, every relation will have some combination of attributes that, when taken together, have the unique identification property; a “combination” consisting of a single attribute is merely a special case. In the following relation SP for example, the combination (S#, P#) has this property.

**SP**

S#	P#	QTY
S1	P1	300
S1	P2	200
S1	P3	400
S1	P4	200
S1	P5	100
S1	P6	100
S2	P1	300
S2	P2	400
S3	P2	200
S4	P2	200
S4	P4	300
S4	P5	400

Occasionally we may encounter a relation in which there is more than one attribute combination possessing the unique identification property, and hence **more than one candidate key**. SUPPLIER is such a relation. Here the situation is that, for all time, each supplier has a unique supplier number and a unique supplier name. In such a case we may arbitrarily choose one of the candidates, say S#, as the primary key for the relation. A candidate key that is not the primary key, such as SNAME in the example, is called an **alternate key**

**PART****SUPPLIER**

P#	PNAME	COLOR	WEIGHT	CITY	S#	SNAME	STATUS	CITY
P1	Nut	Red	12	London	S1	Smith	20	London
P2	Bolt	Green	17	Paris	S2	Jones	10	Paris
P3	Screw	Blue	17	Rome	S3	Blake	30	Paris
P4	Screw	Red	14	London	S4	Clark	20	London
P5	Cam	Blue	12	Paris	S5	Adams	30	Athens
P6	Cog	Red	19	London				

So far we have considered the primary key from a purely formal point of view, that is, purely as an identifier for tuples in a relation, without paying any heed to how those tuples are interpreted. Typically, however, those tuples represent entities in the real world, and the primary key really serves as a unique identifier for those entities. For example, the tuples in the SUPPLIER relation represent individual suppliers, and values of the S# attribute actually identifies those suppliers, not just the tuples that represent them. This interpretation leads us to impose the following rule.

### Integrity Rule 1 (Entity Integrity)

No component of a primary key value may be null.

The rationale behind this rule is as follows. By definition all entities must be distinguishable—that is, they must have a unique identification of some kind. Primary keys perform the unique identification function in a relational database. An identifier (primary key value) that was wholly null would be a contradiction in terms; in effect, it would be saying that there was some entity that did not have any unique identification—i.e., was not distinguishable from other entities (and if two entities are not distinguishable from each other, then by definition there are not two entities but only one). Analogous arguments suggest that partially null identifiers should also be prohibited.

Similar considerations lead us to a second integrity rule. It is common for one relation to include references to another. For example, relation SP includes references to both the SUPPLIER relation and the PART relation, via its S# and P# attributes. It is clear that if a tuple of SP contains a value for S#, say s, then a tuple for suppliers should exist in SUPPLIER (otherwise, the SP tuple would apparently be referring to a nonexistent supplier); and similarly for parts. We can make these notions precise as follows.

First, we introduce the notion of **primary domain**. A given domain may optionally be designated as primary if and only if there exists some single-attribute primary key defined on that domain. Second, any relation including an attribute that is defined on a primary domain (for example, relation SP) must obey the following constraint.

### Integrity Rule 2 (Referential Integrity)

Let D be a primary domain, and let R1 be a relation with an attribute A that is defined on D. Then, at any given time, each value of A in R1 must be either (a) null, or (b) equal to V, say, where V is the primary key value of some tuple in some relation R2 (R1 and R2 not necessarily distinct) with primary key defined on D.

(We note that R2 must exist, by definition of primary domain. We also note that the constraint is trivially satisfied if A is the primary key of R1.)

An attribute such as A is sometimes called a **foreign key**. For example attribute P# of relation SP is a foreign key, since its values are values of the primary key of the PART relation. Keys, primary and foreign, provide a means of representing relationships between tuples; note, however that not all such “relationship” attributes are keys).

### Query Languages:

A query language is a language in which a user requests information from the database. These languages are typically of a level higher than that of a standard programming language. Query language can be categorized as being either procedural or non-procedural. In a procedural language, the user instructs the system to perform a sequence of operations on the database to compute the desired result. In a non-procedural language, the user describes the information desired without giving a specific procedure for obtaining that information.

Most commercial relational-database systems offer a query language that includes elements of both the procedural and non-procedural approaches. The relational algebra is procedural, whereas the **tuple relational calculus** and the **domain relational calculus** are non-procedural.

## Relational Algebra:

A Relational algebra is a notation for representing the types of operations, which can be performed on relational databases. It is used in a RDBMS as the intermediate language for query optimization. Thus an understanding of it is useful for database implementation and for database tuning.

A **relation** is a set of  $k$ -tuples, for some  $k$  called the arity of the relation. In general, names are given to the components of the tuple (a tuple corresponds to a record - Pascal or structure - C with fields corresponding to the names of the components). Note: this definition implies that each tuple is unique. Each relation is described by a schema which consists of a relation name and a list of attribute names - relation-name(attribute-list).  $R(A_1, \dots, A_n)$ ,  $R.A_i$ .

A **relational algebra** is an algebraic language based on a small number of *operators*, which operate on relations (tables). It is the intermediate language used by a RDBMS. Queries are expressed by applying special operators to relations.

### Fundamental Operations:

#### The Select Operation:

The Select Operation selects the tuples that satisfy a given condition or predicate

A Greek Letter **Sigma** can denote the Select operation

For example, find all employees born after 1st Jan 1950:

```
SELECT dob > 01/JAN/1950 (employee)
```



#### The Project Operation:


The project operation is a unary operation that returns its argument relation, with the specified attributes only. The resultant relation does not have any duplicate rows.

Project is denoted by Greek letter pi.



#### The Union Operation:

The union operation allows combining the data from two relations.

It is denoted by 

It creates the set union of two compatible relations.

For a union operation  $r \cup s$  to be valid, we require that the following conditions hold.

- Both relations must have the same number of columns.
- The names of the attributes are the same in both relations.
- Attributes with the same name in both relations have the same domain.

#### Set Difference:

The set difference operation, denoted by  $-$ , allows to find tuples that are in one relation but are not in another. The expression  $r-s$  results in a relation containing those tuples in  $r$  but not in  $s$ . for set

difference operations, we must ensure that the set difference are taken between compatible relations. Therefore, for a set difference operation  $r-s$  to be valid, we require that the relations  $r$  and  $s$  be of the same arity and that the domain of the  $l$ 'th attribute of  $r$  and  $l$ 'th attribute of  $s$  be the same.

### Cartesian product:

Cartesian product operation, denoted by a cross (**X**), allows us to combine information from any two relations.

### Renaming:

The attribute names in the attribute list replace the attribute names of the relation.

### Derived operators:

#### Set intersection:

Finds the common tuples in two relations with like attributes.

#### Divide:

Takes two relations, with attributes  $\{X_1, \dots, X_N, Y_1, \dots, Y_M\}$  and  $\{Y_1, \dots, Y_M\}$  respectively, and returns a relation with attributes  $\{X_1, \dots, X_N\}$  representing all the tuples in the first with matched every tuple in the second relation.

#### Join:

Creates new relation from all combinations of tuples in two relations with some matching, While this relation has the potential to be computationally expensive the join-condition typically allows the operation to be relatively inexpensive.

- The join defined above is called a *theta-join*.
- Equijoins are joins where the join-condition only involves equalities.

### Natural Joins:

The *natural join* of two relations  $R$  and  $S$ , denoted  $R \bowtie S$  is only those tuples of  $R \times S$  that agree on some list of attributes.

The natural join may be defined by

1. Compute  $R \times S$
2. For each attribute  $A$  that names both a column in  $R$  and a column in  $S$ , select from  $R \times S$  those tuples whose values agree in the columns for  $R.A$  and  $S.A$ .
3. For each attribute  $A$  above, project out the column  $S.A$  and call the remaining column  $R.A$ , simply  $A$ . (example:  $\text{employee}(\text{id}, \text{name})$ ,  $\text{salary}(\text{id}, \text{salary})$ ; the natural join  $\text{employee-salary}(\text{id}, \text{name}, \text{salary})$ )

The *theta join* of two relations  $R$  and  $S$  denoted  $R \bowtie_C S$  is only those tuples of  $R \times S$  that satisfy the condition  $C$ .

1. Compute  $R \times S$
2. Select from the product only those tuples that satisfy the condition.

## Renaming

$\tilde{S}(A_1, \dots, A_n)(R)$  is the same relation as  $R$  but its name is  $S$  with the attributes named.  $A_1, \dots, A_n$ .

**Relational Calculus:**

The relational calculus is based on the *first order logic*. There are two variants of the relational calculus:

- The *Domain Relational Calculus* (DRC), where variables stand for components (attributes) of the tuples.
- The *Tuple Relational Calculus* (TRC), where variables stand for tuples.

**Tuple Relational Calculus:**

The queries used in TRC are of the following form:  $x(A) \wedge F(x)$  where  $x$  is a tuple variable  $A$  is a set of attributes and  $F$  is a formula. The resulting relation consists of all tuples  $t(A)$  that satisfy  $F(t)$ .

The SQL language is based on the tuple relational calculus, which in turn is a subset of classical predicate logic. Queries in the TRC all have the form:

$$\{QueryTarget \mid QueryCondition\}$$

The *QueryTarget* is a tuple variable, which ranges over tuples of values. The *QueryCondition* is a logical expression such that

- It uses the *QueryTarget* and possibly some other variables.
- If a concrete tuple of values is substituted for each occurrence of the *QueryTarget* in *QueryCondition*, the condition evaluates to a Boolean value of *true* or *false*.

The result of a TRC query with respect to a database instance is the set of all choices of values for the query variable that make the query condition a true statement about the database instance. The relation between the TRC and logic is in that the *QueryCondition* is a logical expression of classical first-order logic.

**Domain Relational Calculus:**

Queries in the DRC have the form:

$$\{X_1, \dots, X_n \mid Condition\}$$

The  $X_1, \dots, X_n$  are a list of domain variables. The condition is a logical expression of classical first-order logic.

**Relational Algebra vs. Relational Calculus:**

The relational algebra and the relational calculus have the same *expressive power*; i.e. all queries that can be formulated using relational algebra can also be formulated using the relational calculus and vice versa. E. F. Codd first proved this in 1972. This proof is based on an algorithm by which an arbitrary expression of the relational calculus can be reduced to a semantically equivalent expression of relational algebra.



It is sometimes said that languages based on the relational calculus are “higher level” or “more declarative” than languages based on relational algebra because the algebra (partially) specifies the order of operations while the calculus leaves it to a compiler or interpreter to determine the most efficient order of evaluation.

## Summary:

### Technical terms:

**Relation** - a set of tuples.

**Tuple** - a collection of attributes, which describe some real world entity.

**Attribute** - a real world role played by a named domain.

**Domain** - a set of atomic values.

**Set** - a mathematical definition for a collection of objects, which contains no duplicates.

## References:

### Database System Concepts

Silberschatz, Korth, and Sudarshan

### Database Management Systems

Arun K. Majumdar, Pritimoy Bhattacharyya

### An Introduction to Database Systems

Bipin Desai

### Modern Database Management

F. McFadden, J. Hoffer

### An Introduction to Database Systems

C. J. Date;

### AUTHOR:

**Y.SURESH BABU.**, M.Com, M.C.A.,  
Lecturer,  
Dept. Of Computer Science,  
JKC College.  
GUNTUR.

## Lesson 5

# SQL - 1

### 5.0 Objectives:

The objective of this chapter is to introduce the main concepts of data storage and retrieval in the context of database information systems using Structured Query Language (SQL).

After reading this chapter, you should understand:

- What is SQL?
- To define features of SQL
- Structure of SQL
- Use various functions based on the data types to perform calculations on data.
- Basic clauses including SELECT, WHERE, FROM
- String Operations
- Set Operations
- Aggregate Functions
- Nested Subqueries.

### Structure of the Lesson:

- 5.1. Introduction
- 5.2. Features of SQL
- 5.3. Basic Structure
  - 5.3.1. Rename Operation
  - 5.3.2. Tuple Variable
  - 5.3.3. String Operators
  - 5.3.4. Ordering the display of Tupels
  - 5.3.5. Duplicates
- 5.4. Set Operations
- 5.5. Aggregate functions
- 5.6. Null Values
- 5.7. Nested Subqueries
- 5.8. Summary
- 5.9. Technical Terms
- 5.10. Model Questions
- 5.11. References

## 5.1. Introduction:

Oracle is a Relational Database Management System (RDBMS). Oracle being RDBMS, stores data in tables called relations. These relations are two-dimensional representation of data, where rows called tuples represent records and columns called attributes represent pieces of information contained in the record.

Oracle provides a rich set of tools to allow design and maintenance of the database. Major tools are,

RDBMS Kernel	: Database Engine
SQL	: Structured Query Language
SQL * PLUS	: Addition to SQL
PL / SQL	: Procedural Language SQL, allows Procedural processing of SQL statements
SQL * DBA	: Database Administrator's tool set
DEVELOPER 2000	: ORACLE'S GUI tool for Forms.

## 5.2. Features of SQL:

SQL (pronounced "ess-que-el") stands for Structured Query Language. SQL is used to communicate with a database. According to ANSI (American National Standards Institute), it is the standard language for relational database management systems. SQL statements are used to perform tasks such as update data on a database, or retrieve data from a database. Some common relational database management systems that use SQL are: Oracle, Sybase, Microsoft SQL Server, Access, Ingres, etc. Although most database systems use SQL, most of them also have their own additional proprietary extensions that are usually used on their system only. However, the standard SQL commands such as "Select", "Insert", "Update", "Delete", "Create", and "Drop" can be used to accomplish almost everything that one needs to do with a database. This tutorial will provide you with the instruction on the basics of each of these commands as well as allow you to put them to practice using the SQL Interpreter.

### Features:

- SQL is an English like language.
- SQL is a non-procedural language.
- SQL processes set of records rather than a single record at a time.
- SQL provides commands for a variety of tasks including querying data.
- Inserting, Updating and Deleting rows in a table.
- Creating, Modifying and Deleting database objects.
- Controlling access to a database and database objects.
- A range of users including DBA, Application programmer, management personal and types of end users can use SQL.

### Database Objects:

Each user owns a single schema. Schema Objects can be created and maintained with SQL.

The following are the list of schema objects:

- Tables
- Indexes
- Views
- Clusters
- Sequences
- Database triggers
- Stored functions and procedures
- Packages.

### 5.3. Basic Structure:

The Basic structure of an SQL expression consists of select, from and where clauses.

**Select** clause lists attributes to be copied - corresponds to relational algebra project. **From** clause corresponds to Cartesian product - lists relations to be used. **Where** clause corresponds to selection predicate in relational algebra.

#### Typical query has the form

**select**  $A_1, A_2, \dots, A_n$

**from**  $T_1, T_2, \dots, T_n$

**where**  $p$

Where each  $A_i$  represents an attribute, each  $r_i$  relation, and P is a predicate. This is equivalent to the relational algebra expression

$$\pi_{A_1, A_2, \dots, A_n} (\sigma_p (r_1 \times r_2 \times \dots \times r_m))$$

If the where clause is omitted, the predicate P is true.

The list of attributes can be replaced with a \* to select all. SQL forms the Cartesian product of the relations named, performs a selection using the predicate, then projects the result onto the attributes named. The result of an SQL query is a relation. SQL may internally convert into more efficient expressions

The relation schemes for the banking example used throughout the textbook are:

- *Branch-scheme* = (*bname*, *bcity*, *assets*)
- *Customer-scheme* = (*cname*, *street*, *ccity*)
- *Depositor-scheme* = (*cname*, *account#*)

- *Account-scheme* = (*bname*, *account#*, *balance*)
- *Loan-scheme* = (*bname*, *loan#*, *amount*)
- *Borrower-scheme* = (*cname*, *loan#*)

Finding the names of all branches in the *account* relation.

```
select bname
from account
```

**distinct** vs. **all**: Elimination or non-elimination of duplicates. For example, finding the names of all branches in the *account* relation.

```
select distinct bname
from account
```

By default, duplicates are not removed. We can state it explicitly using **all**.

For example

```
select all bname
from account
select * means select all the attributes.
```

Arithmetic operations can also be in the selection list. The predicates can be more complicated, and can involve:

- Logical connectives **and**, **or** and **not**.
- Arithmetic expressions on constant or tuple values.
- The **between** operator for ranges of values.

For example to find account number of accounts with balances between \$90,000 and \$100,000.

```
select account#
from account
where balance between 90000 and 100000.
```

### 5.3.1. Rename Operation:

SQL provides a mechanism for renaming both relations and attributes. It uses the **as** clause, taking the form

Old-name as new-name

The **as** clause can appear in both the **select** and **from** clauses.

EX:

```
select distinct cname, borrower.loan# as loan_id
```

```
from borrower, loan
```

```
where borrower.loan# = loan.loan# and bname= "SFU"
```

### 5.3.2. Tuple Variables:

A tuple variable in SQL must be associated with a particular relation. Tuple variables are defined in the **from** clause by way of the **as** clause.

```
select distinct cname, T.loan#  
from borrower as S, loan as T  
where S.loan# = T.loan#
```

We define a tuple variable in the from clause by placing it after the name of the relations with which it is associated, with the keyword **as** in between .

The tuple variables are most useful for comparing two tuples in same relation.

### 5.3.3. String Operators:

The most commonly used operation on strings is pattern matching using the operator **like**. String matching operators **%** (any substring) and **\_** (underscore, matching any character).

Ex : “**\_\_\_%**” matches any string with at least 3 characters.

Patterns are case sensitive, e.g., “Jim” does not match “jim”. Use the keyword **escape** to define the *escape* character.

Ex : like “**ab%tely%\%**” *escape* “\” matches all the strings beginning with “ab” followed by a sequence of characters and then “tely” and then “% \”.

Backslash overrides the special meaning of these symbols. We can use **not like** for string mismatching.

Ex : Find all customers whose street includes the substring “Main”.

```
select cname  
from customer  
where street like "%Main%"
```

SQL also permits a variety of functions on character strings, such as concatenating (using “||”), extracting substrings, finding the length of strings, converting between upper case and lower case, and so on.

### 5.3.4. Ordering the Display of Tuples

SQL allows the user to control the order in which tuples are displayed. **order by** makes tuples appear in sorted order (ascending order by default). **desc** specifies descending order. **asc** specifies ascending order.

```
select *
      from loan
      order by amount desc, loan# asc
```

Sorting can be costly, and should only be done when needed.

### 5.3.5. Duplicates:

Formal query languages are based on mathematical relations. Thus no duplicates appear in relations. As duplicate removal is expensive, SQL allows duplicates. To remove duplicates, we use the **distinct** keyword. To ensure that duplicates are not removed, we use the **all** keyword.

*Multiset* (bag) versions of relational algebra operators.

If there are  $c_1$  copies of tuples  $t_1$  in  $r_1$  and  $t_1$  satisfies selection  $\sigma_\theta$  then there are  $c_1$  copies of  $t_1$  in  $\sigma_\theta(r_1)$

for each copy of tuple  $t_1$  in  $r_1$  there is a copy of tuple  $\Pi_A(t_1)$  in  $\Pi_A(r_1)$

If there are  $c_1$  copies of tuple  $t_1$  in  $r_1$  and  $c_2$  copies of tuple  $t_2$  in  $r_2$ , there is  $c_1 \times c_2$  copies of tuple  $t_1.t_2$  in  $r_1 \times r_2$

An SQL query of the form

```
Select A1, A2, ..., An
      from T1, T2, ..., Tm
      Where P
```

is equivalent to the algebra expression

$$\Pi_{A_1, A_2, \dots, A_n} (\sigma^P(T_1 \times T_2 \times \dots \times T_m))$$

using the multiset versions of the relational operators  $\sigma$ ,  $\Pi$  and  $\times$

## 5.4. Set Operations:

SQL has the set operations **union**, **intersect** and **except** operate on relations and correspond to the relational-algebra operations  $\cup$ ,  $\cap$  and  $-$

We shall now construct queries involving the UNION, INTERSECT and EXCEPT operations of two sets; the set of all customers who have an account at the bank which can be derived by

**Select** customer-name **from** depositor

And the set of customers who have a loan at the bank, which can be derived by

**Select** customer-name **from** borrower

**1. Union:**

Return all distinct rows retrieved by either of the queries.

Ex: select job from emp union select desg from employee;

**2. Union All:**

Returns all rows (including duplicate) retrieved by either of the queries.

**3. Intersect:**

Returns only rows retrieved by both of the queries.

## 5.6. Aggregate Functions:

The aggregate functions are the functions that take a collection (a set or multiset) of values as input and return a single value.

- Average value — avg
- Minimum value — min
- Maximum value — max
- Total sum of values — sum
- Number in group — count

The input to sum and avg must be collection of numbers, but the other operators can operate on collections of nonnumeric data types, such as strings.

Ex: **select** *bname*, **avg** (*balance*) **from** *account*  
**group by** *bname*

**select** *bname*, **count** (**distinct** *cname*) **from** *account*, *depositor*  
**where** *account.account#* = *depositor.account#*  
**group by** *bname*

**select** *bname*, **avg** (*balance*) **from** *account*  
**group by** *bname* **having** **avg** (*balance*) > 1200

**select** *depositor.cname*, **avg** (*balance*)



```

from depositor, account, customer
  where depositor.cname = customer.cname and
account.account# = depositor.account#
and ccity="Vancouver" group by depositor.cname
  having count (distinct account#) > 3

```

If a **where** clause and a **having** clause appear in the same query, the **where** clause predicate is applied first. Tuples satisfying **where** clause are placed into groups by the group by clause. The **having** clause is applied to each group. Groups satisfying the **having** clause are used by the **select** clause to generate the result tuples. If no **having** clause is present, the tuples satisfying the **where** clause are treated as a single group.

### 5.7. Null Values:

With insertions, we saw how **null** values might be needed if values were unknown. Queries involving nulls pose problems. If a value is not known, it cannot be compared or be used as part of an aggregate function.

All comparisons involving **null** are false by definition. However, we can use the keyword **null** to test for null values:

```

select distinct loan# from loan

  where amount is null

```

### 5.8. Nested Sub Queries:

**SQL** provides a mechanism for nesting subqueries. A subquery is a **select-from-where** expression that is nested within another query. A common use of subqueries is to perform the tests for set membership, make set comparisons and determine set cardinality.

**SET MEMBERSHIP:** The **in** connective tests for set membership, where the set is a collection of values produced by a select clause. The **not in** connective tests for the absence of set membership.

Examples:

```

select distinct cname from borrower where cname in
(select cname from account where bname= "SFU")

```

we use the **not in** construct in the similar way.

### SET COMPARISON: (SOME/ANY or ALL)

These operators may be used in **WHERE** or **HAVING** clauses for sub-queries, that returns more than one row. These Operators compares a value with each value returned by a sub-query and returns a value.



### 5.9. Summary:

SQL is a query language that allows access to data residing in relational database management systems (RDBMS), such as Sybase, Oracle, Informix, DB2, Microsoft SQL Server, Access and many others. To retrieve information users execute '*queries*' to pull the requested information from the database using criteria that is defined by the user. SELECT is the most important and the most complex SQL statement. You can use it and the SQL statements INSERT, UPDATE, and DELETE to manipulate data. You can use the SELECT statement to retrieve data from a database, as part of an INSERT statement to produce new rows, or as part of an UPDATE statement to update information. A query, in its simplest form is constructed using the following basic query statements SELECT, FROM, WHERE and ORDER BY. The SELECT clause defines what columns or fields you want to see in your results, the FROM clause defines from what table the columns reside in, the WHERE clause defines any special criteria that must be met in order to be displayed, and finally the ORDER BY clause in which you define the sequence you want to display the results. While the only two query clauses that are required are SELECT and FROM, they are almost always accompanied by the WHERE and ORDER BY clauses to restrict the amount of data retrieved and to present it in an orderly fashion. Oracle SQL supports the following four set operations: UNION, MINUS, INTERSECT.

### 5.10. Technical terms:

**SQL:** SQL (Structured Query Language) is a standard interactive and programming language for getting information from and updating a database.

**Schema:** A description of the data represented within a database.

**Attribute:** Characteristic of an entity/object, eg: the *name* of a person.

**Data Definition Language:** Language sub-system of a data management system that is used to define the structure of the database.

**Domain:** A domain is the set of allowable values for one or more attributes.

**Degree:** The degree of a relation is the number of attributes it contains.

**Nested Query:** Nested query is when a SELECT statement embedded within another SELECT statement.

### 5.11. Model Questions:

1. What is SQL? Explain Features of SQL?
2. Write about structure of SQL?
3. What are various string operations?
4. What are various Set operations?
5. Explain various Aggregate functions with example queries?

## 5.12. References:

**Database System Concepts**

Silberschatz, Korth, and Sudarshan

**Database Management Systems**

Arun K. Majumdar, Pritimoy Bhattacharyya

**An Introduction to Database Systems**

Bipin Desai

**Modern Database Management**

F. McFadden, J. Hoffer

**An Introduction to Database Systems**

C. J. Date;

**AUTHOR:**

**Y.SURESH BABU., M.Com, M.C.A.,  
Lecturer,  
Dept. Of Computer Science,  
JKC College.  
GUNTUR.**

## Lesson 6

# SQL - 2

### 6.0 Objectives:

The objective of this chapter is to introduce the main concepts of data storage and retrieval in the context of database information systems using Structured Query Language (SQL).

After reading this chapter, you should understand:

- What is View?
- Understand how to create views
- Structure of complex queries
- Joined relations
- Understand Data Definition Language
- What is Dynamic SQL?

### Structure of the Lesson:

- 6.1. Views
- 6.2. Complex Queries
- 6.3. Modifications of the Database
- 6.4. Joining relations
- 6.5. Data Definition Language
- 6.6. Embedded SQL
- 6.7. Dynamic SQL
- 6.8. Summary
- 6.9. Technical Terms
- 6.10. Model Questions
- 6.11. References

### 6.1. Views:

A view is like a window through which data on tables can be viewed or changed. A view is derived from another table or view, which is referred as the base table. A view is stored as a SELECT statement only but has no data of its own. It manipulates data in the underlying base table.

A view in SQL is defined using the **create view** command:

Create view *v* as (query expression)

Where (*query expression*) is any legal query expression.

**Ex:** To create a view *all-customer* of all branches and their customers:

```
create view all-customer as
  (select bname, cname
   from depositor, account
   where depositor.account# = account.account#)
  union
  (select bname, cname from borrower, loan
   where borrower.loan# = loan.loan#)
```

## 6.2. Complex Queries:

Complex queries are often hard or impossible to write as a single SQL block or a union/intersection/difference of SQL blocks. An SQL block consists of a single **select from where** statement, possibly with **group by** and **having** clauses. There is a way composing multiple SQL blocks to express a complex query:

**Derived Relations:** SQL allows a sub query expression to be used in the **from** clause. If we use such an expression, then we must give the result relation a name, and we can rename the attribute. We do this renaming by using the **as** clause. For example to find average account balance of those branches where the average account balance is greater than \$1,000.

```
select bname, avg-balance
  from (select bname, avg(balance)
        from account group by bname)
  as result(bname, avg-balance)
  where avg-balance > 1000
```

## 6.3. Modifications of the Database:

### Deletion:

Deletion is expressed in much the same way as a query. Instead of displaying, the selected tuples are removed from the database. We can only delete whole tuples.

A deletion in SQL is of the form

```
delete from r where P
```

Tuples in **r** for which **P** is true are deleted. If the **where** clause is omitted, all tuples are deleted. A delete command operates on only relation. If we want to delete tuples from several relations, we must use one delete command for each relation.

1. Delete all of Smith's account records.

```
delete from depositor
  where cname="Smith"
```

2. Delete all loans with loan numbers between 1300 and 1500.

```
delete from loan
  where loan# between 1300 and 1500
```

3. Delete all accounts at branches located in Surrey.

```
delete from account
where bname in
(select bname from branch
where bcity="Surrey")
```

**Insertion:**

To insert data into a relation, we either specify a tuple, or write a query whose result is the set of tuples to be inserted. Attribute values for inserted tuples must be members of the attribute's domain.

Examples:

1. To insert a tuple for Smith who has \$1200 in account A-9372 at the SFU branch.  

```
insert into account
values ("SFU", "A-9372", 1200)
```
2. To provide each loan that the customer has in the SFU branch with a \$200 savings account.  

```
insert into account
select bname, loan#, 200
from loan where bname="SFU"
```

We can prohibit the insertion of null values using the SQL DDL.

**Update:**

We may wish to change a value in a tuple without changing the values in the tuple. For this purpose update statement is used.

Example: To increase all balances by 5 percent.

```
update account
set balance=balance * 1.05
```

This statement is applied to every tuple in *account*.

In general, where clause of update statement may contain any legal construct in a where clause of a select statement (including nesting). A nested select within an update may reference the relation that is being updated. As before, all tuples in the relation are first tested to see whether they should be updated, and the updates are carried out afterwards.

Example: To pay 5% interest on account whose balance is greater than average, we have

```
update account
set balance=balance * 1.05
where balance >
select avg (balance) from account
```

**Transactions:**

A transaction consists of a sequence of query and/or update statements. The SQL standard specifies that a transaction begins implicitly when an SQL statement is executed.

**Commit work:** Commits the current transaction; i.e., it makes the updates performed by the transaction become permanent in the database. After the transaction is committed, a new transaction is automatically started.

**Rollback work:** Causes the current transaction to be rolled back; i.e., it undoes all the updates performed by the SQL statements in the transaction. Thus the database state is restored to what it was before the first statement of the transaction was executed.

## 6.4. Joined Relations:

SQL provides the basic Cartesian-product mechanism for joining tuples of relations, and it also provides other mechanism for joining relations, including condition joins and natural joins.

Examples: Here there are two relations, named loan and borrower.

Loan			borrower	
bname	loan#	amount	cname	loan#
Downtown	L-170	3000	Jones	L-170
Redwood	L-230	4000	Smith	L-230
Perryridge	L-260	1700	Hayes	L-155

**join:**

*loan inner join borrower on  
loan.loan# = borrower.loan#*

Notice that the loan# will appear twice in the inner joined

bname	loan#	amount	cname	loan#
Downtown	L-170	3000	Jones	L-170
Redwood	L-230	4000	Smith	L-230

Result of *loan inner join borrower*

**left outer join:**

bname	loan#	amount	cname	loan#
Downtown	L-170	3000	Jones	L-170
Redwood	L-230	4000	Smith	L-230
Perryridge	L-260	1700	null	null

*loan left outer join borrower on loan.loan# = borrower.loan#*

Result of *loan left outer join borrower*.

**natural inner join:**

*loan natural inner join borrower*

bname	loan#	amount	cname
Downtown	L-170	3000	Jones
Redwood	L-230	4000	Smith



Result of loan natural inner join borrower.

**Join Types :** inner join, left outer join, right outer join, full outer join.

The keyword **inner** and **outer** are optional since the rest of the join type enables us to deduce whether the join is an **inner join** or an **outer join**. It also provides two other join types, These are

**cross join :** an **inner join** without a join condition.

**union join:** a **full outer join** on the “false” condition, i.e., where the **inner join** is empty.

$A_1, A_2, \dots, A_n$

**Join conditions:** natural, on predicate, using .

The use of join condition is mandatory for outer joins

but is optional for inner joins (if it is omitted, a Cartesian product results).

## 6.5. Data-Definition Language:

The SQL DDL (Data Definition Language) allows specification of not only a set of relations, but also the following information for each relation:

- The schema for each relation.
- The domain of values associated with each attribute.
- Integrity constraints.
- The set of indices for each relation.
- Security and authorization information.
- Physical storage structure on disk.

### Domain Types in SQL:

The SQL-92 standard supports a variety of built-in domain types:

- **char(n):** Fixed-length character string, with user-specified length.
- **varchar(n):** Variable-length character string, with user-specified maximum length.
- **int:** an integer (length is machine-dependent).
- **smallint:** A small integer (length is machine-dependent).
- **numeric(p, d):** A fixed-point number with user-specified precision, consists of p digits (plus a sign) and d of p digits are to the right of the decimal point. E.g., numeric(3, 1) allows 44.5 to be stored exactly but not 444.5.
- **real, double precision:** Floating-point or double-precision floating-point numbers, with machine-dependent precision.
- **float(n):** Floating-point, with user-specified precision of at least n digits.
- **date:** A calendar date, containing four digit year, month, and day of the month.
- **time:** The time of the day in hours, minutes, and seconds.

- **timestamp**: A combination of date and time. A variant, timestamp(p) can be used to specify the number of fractional digits for seconds.

**Schema Definition in SQL:** An SQL relation is defined by:

**Create table** ( $A_1, D_1, A_2, D_2, \dots, A_n, D_n$ )

**Integrity Constraint1**

.....

**Integrity Constraint1)**

where  $r$  is the relation name,  $A_i$  is the name of an attribute, and

$D_i$  is the domain of that attribute. The allowed integrity-constraints are

**primary key** : ( $A_{j1}, \dots, A_{jm}$ ) The primary key specification says that attributes ( $A_{j1}, \dots, A_{jm}$ )

from the primary key for the relation. The primary key attributes are required to be non-null and unique.

Example: create table branch ( bname char(15) not null

bcity char(30)

assets integer

primary key (*bname*)

check (*assets* >= 0))

Check(P): The check clause specifies a predicate P that must be satisfied by every tuple in the relation.

**Example :**

create table student (name char(15) not null

student-id char(10) not null

degree-level char(15) not null

check (degree-level in

("Bachelors", "Masters", "Doctorate"))))

## 6.6. Embedded SQL:

SQL provides a powerful declarative query language. However, access to a database from a general-purpose programming language is required because,

- o SQL is not as powerful as a general-purpose programming language. There are queries that cannot be expressed in SQL, but can be programmed in C, Fortran, Pascal, Cobol, etc.
- o Non-declarative actions such as printing a report, interacting with a user, or sending the result to a GUI — cannot be done from within SQL.

The SQL standard defines embedding of SQL as *embedded SQL* and the language in which SQL queries are embedded is referred as *host language*. The result of the query is made available to the program one tuple (record) at a time. To identify embedded SQL requests to the preprocessor, we use EXEC SQL statement:

EXEC SQL embedded SQL statement END-EXEC

A semi-colon is used instead of END-EXEC when SQL is embedded in C or Pascal.

### Embedded SQL statements: declare cursor, open, and fetch

#### EXEC SQL

```
declare c cursor for
    select cname, ccity
    from deposit, customer
    where deposit.cname = customer.cname
    and deposit.balance > :amount
```

END-EXEC

where amount is a host-language variable.

EXEC SQL open c END-EXEC

This statement causes the DB system to execute the query and to save the results within a temporary relation.

A series of **fetch** statement are executed to make tuples of the results available to the program.

```
EXEC SQL fetch c into :cn, :cc END-EXEC
```

The program can then manipulate the variable *cn* and *cc* using the features of the host programming language.

A single **fetch** request returns only one tuple. We need to use a **while** loop (or equivalent) to process each tuple of the result until no further tuples (when a variable in the SQLCA is set).

We need to use **close** statement to tell the database system to delete the temporary relation that held the result of the query.

```
EXEC SQL close c END-EXEC
```

Embedded SQL can execute any valid **update**, **insert**, or **delete** statements

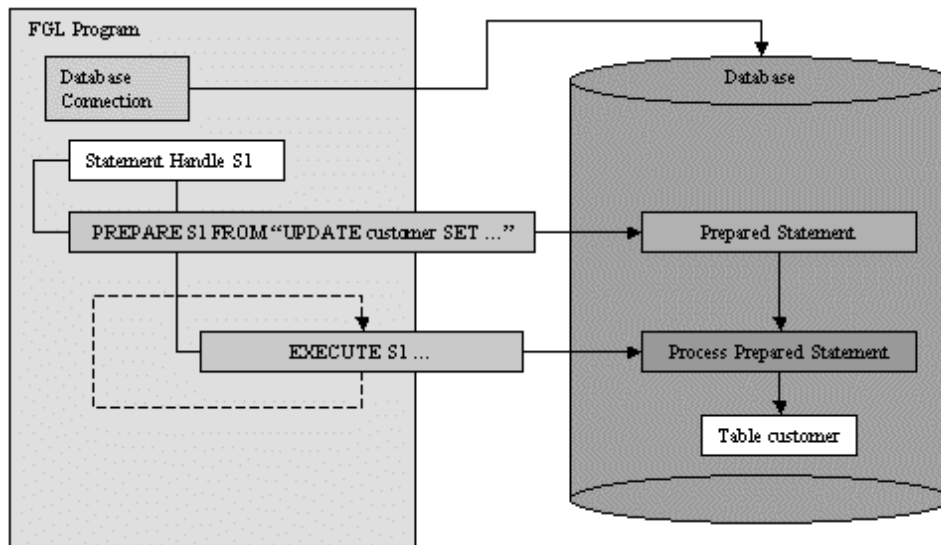
### 6.7. Dynamic SQL:

BDL includes basic SQL instructions in the language syntax, but only a limited number of SQL instructions are supported this way. Dynamic SQL Management allows you to execute any kind of SQL statement, hard coded or created at runtime, with or without SQL parameters, returning or not returning a result set.

In order to execute an SQL statement with Dynamic SQL, you must first prepare the SQL statement to initialize a statement handle, then

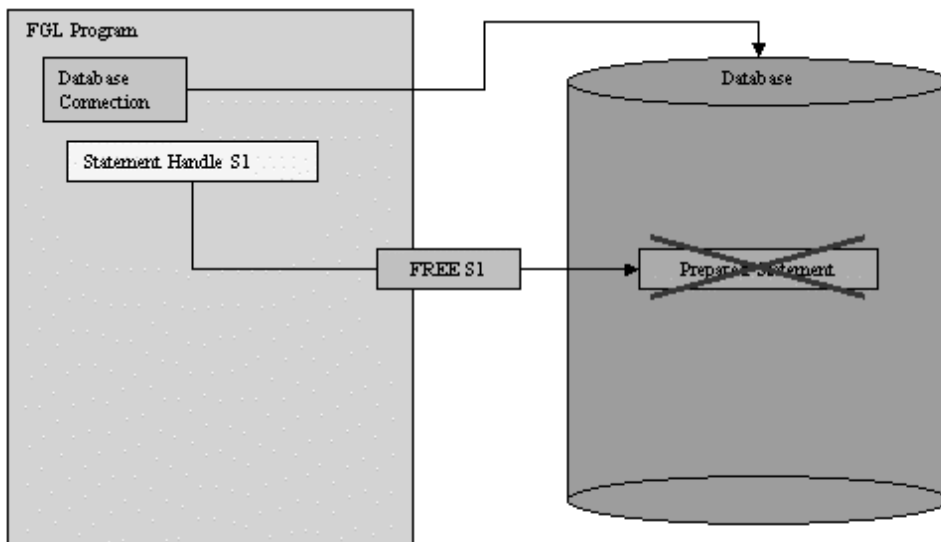
When you no longer need the prepared statement, you can free the statement handle to release allocated resources:

When using insert cursors or SQL statements that produce a result set (like SELECT), you must declare a cursor with a prepared statement handle.



Prepared SQL statements can contain SQL parameters by using ? placeholders in the SQL text. In this case, the EXECUTE or OPEN instruction supplies input values in the USING clause.

To increase performance efficiency, you can use the PREPARE instruction, together with an EXECUTE instruction in a loop, to eliminate overhead caused by redundant parsing and optimizing. For example, an UPDATE statement located within a WHILE loop is parsed each time the loop runs. If you prepare the UPDATE statement outside the loop, the statement is parsed only once, eliminating overhead and speeding statement execution.



## 6.8. Summary:

SQL View is a virtual table, which is based on SQL SELECT query. Essentially a view is very close to a real database table except for the fact that the real tables store data, while the views don't. The view's data is generated dynamically when the view is referenced.

Most database applications do a specific job. For example, a simple program might prompt the user for an employee number, then update rows in the EMP and DEPT tables. In this case, you know the makeup of the UPDATE statement at pre-compile time. That is, you know which tables might be changed, the constraints defined for each table and column, which columns might be updated, and the data type of each column.

However, some applications must accept (or build) and process a variety of SQL statements at run time. For example, a general-purpose report writer must build different SELECT statements for the various reports it generates. In this case, the statement's makeup is unknown until run time. Such statements can, and probably will, change from execution to execution. They are aptly called *dynamic* SQL statements.

Unlike static SQL statements, dynamic SQL statements are not embedded in your source program. Instead, they are stored in character strings input to or built by the program at run time. They can be entered interactively or read from a file.

Dynamic SQL allows you to write SQL that will then write and execute more SQL for you. This can be a great time saver because you can: Automate repetitive tasks, write code that will work in any database or server and write code that dynamically adjusts itself to changing conditions

## 6.9. Technical Terms:

**View:** A logical table whose data are not physically stored. You define a view to access a subset of the columns stored in a row. Access a set of columns stored in different rows or avoid creating a redundant copy of data that is already stored.

### **Join:**

The JOIN is a SQL command used to retrieve data from two or more database tables with existing relationship based upon a common attribute.

### **DDL:**

DDL Data Definition Language. A language used by a database management system which allows users to define the database, specifying data types, structures and constraints on the data. Examples are the CREATE TABLE, CREATE INDEX, ALTER, and DROP statements. Note: DDL statements will implicitly commit any outstanding transaction.

### **Dynamic SQL:**

SQL statements are created, prepared, and executed while a program is executing. It is, therefore, possible with dynamic SQL to change the SQL statement during program execution and have many variations of a SQL statement at run time.

**6.10. Model Questions:**

1. What is a view? How to create views in SQL?
2. Write short notes on Complex Queries?
3. Explain the concept of modifications of the Database?
4. How to joining relations in SQL? Explain?
5. Write short notes Embedded SQL and Dynamic SQL?

**6.11. References:****Database System Concepts**

Silberschatz, Korth, and Sudarshan

**Database Management Systems**

Arun K. Majumdar, Pritimoy Bhattacharyya

**An Introduction to Database Systems**

Bipin Desai

**Modern Database Management**

F. McFadden, J. Hoffer

**An Introduction to Database Systems**

C. J. Date;

**AUTHOR:**

**Y.SURESH BABU.**, M.Com, M.C.A.,  
Lecturer,  
Dept. Of Computer Science,  
JKC College.  
GUNTUR.

## Lesson 7

# Integrity and Security

### 7.0 Objectives:

After reading this chapter, you should understand:

- What is Domain Constraint
- What is Referential Integrity
- What are Assertion
- Various Security Issues

### Structure of the Lessons

- 7.1. Domain Constraints
- 7.2. Referential Integrity
  - 7.2.1. Referential Integrity in E-R Model
  - 7.2.2. Database Modification
  - 7.2.3. Referential Integrity in SQL
- 7.3. Assertions
- 7.4. Triggers
  - 7.4.1. Need For Triggers
  - 7.4.2. Triggers in SQL
  - 7.4.3.
- 7.5. Security and Authorization
  - 7.5.1. Security Violations
  - 7.5.2. Authorization
  - 7.5.3. Authorization and Views
  - 7.5.4. Granting Of Privileges
  - 7.5.5. Privileges in SQL
  - 7.5.6. Roles
  - 7.5.7. Limitations of SQL Authorization
- 7.6. Encryption and Authentication
- 7.7. Summary
- 7.8. Technical Terms
- 7.9. Model Questions
- 7.10. References

## 7.1. Domain Constraints:

A domain of possible values must be associated with every attribute. The number of standard domain types, such as integer types, characters types, and date/type times are defined in SQL. Declaring an attribute to be of a particular domain acts as a constraint on the values that it can take. Domain constraints are the most elementary form of integrity constraints. They are tested easily by the system whenever a new data item is entered into the database.

The definition of domain constraints not only allows us to test values inserted in the database, but also permits us to test queries to ensure that the comparisons made make sense. The **create domain** clause can be used to define new domains.

```
create domain Dollars numeric(12,2)
```

```
create domain Pounds numeric(12,2)
```

We cannot assign or compare a value of type Dollars to a value of type Pounds. However, we can convert type as below

```
(cast r.A as Pounds)
```

```
(Should also multiply by the dollar-to-pound conversion-rate)
```

SQL provides **drop domain** and **alter domain** clauses to drop or modify domains that have been created with **create domain**.

The **check clause** in SQL permits domains to be restricted in powerful ways that most programming language type systems do not permit. The check clause permits the schema designer to specify a predicate that must be satisfied by any value assigned to a variable whose type is the domain.

```
create domain hourly-wage numeric(5,2)
```

```
constraint value-test check(value > = 4.00)
```

- The domain has a constraint that ensures that the hourly-wage is greater than 4.00.
- The clause constraint value-test is optional; useful to indicate which constraint an update violated.

The check clause can also be used to restrict a domain not to contain any null values;

```
create domain AccountNumber char(10) constraint
```

```
account-number-null-test check(value not null)
```

The domain can be restricted to contain only a specified set of values by using the **in** clause.

```
Create domain AccountType char(10)
```

```
constraint account-type-test
```

```
check(value in ('Checking','Saving'))
```

## 7.2. Referential Integrity:

Ensures, a value that appears in one relation for a given set of attributes also appears for a certain set of attributes in another relation. This condition is called referential integrity.

Example: If "Perryridge" is a branch name appearing in one of the tuples in the *account* relation, then there exists a tuple in the *branch* relation for branch "Perryridge".

### Definition:

Let  $r_1(R_1)$  and  $r_2(R_2)$  be relations with primary keys  $K_1$  and  $K_2$  respectively. The subset a of  $R_2$  is a **foreign key** referencing  $K_1$  in relation  $r_1$ , if for every  $t_2$  in  $r_2$  there must be a tuple  $t_1$  in  $r_1$  such that  $t_1[K_1] = t_2[a]$ .



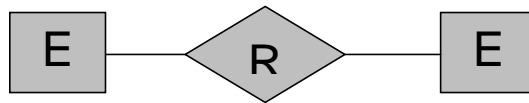
Referential integrity constraint also called **subset dependency** since its can be written as

$$\Pi_a(r_2) \subseteq \Pi_{K_1}(r_1)$$

### 7.2.1 Referential Integrity:

Referential-integrity constraints arise frequently. If we derive our relational-database schema by constructing tables from E-R diagrams, then every relation arising from a relationship set has referential-integrity constraints.

Consider relationship set  $R$  between entity sets  $E_1$  and  $E_2$ . The relational schema for  $R$  includes the primary keys  $K_1$  of  $E_1$  and  $K_2$  of  $E_2$ . Then  $K_1$  and  $K_2$  form foreign keys on the relational schemas for  $E_1$  and  $E_2$  respectively



Another source of referential-integrity constraints is weak entity sets. The relation schema for a weak entity set must include the primary key attributes of the entity set on which it depends. The relation schema for each weak entity set includes a foreign key that leads to a referential-integrity constraint.

### 7.2.2. Database Modification:

Database modifications can cause violations of referential integrity. We list here the test that we must make for each type of database modification to preserve the following referential-integrity constraint.

$$\Pi_a(r_2) \subseteq \Pi_K(r_1)$$

#### Insert:

If a tuple  $t_2$  is inserted into  $r_2$ , the system must ensure that there is a tuple  $t_1$  in  $r_1$  such that  $t_1[K] = t_2[a]$  that is

$$t_2[a] \in \Pi_K(r_1)$$

#### Delete:

If a tuple,  $t_1$  is deleted from  $r_1$ , the system must compute the set of tuples in  $r_2$  that reference  $t_1$ :

$$\sigma_{a = t_1[K]}(r_2)$$

If this set is not empty, either the delete command is rejected as an error, or the tuples that reference  $t_1$  must themselves be deleted. The latter solution may lead to cascading deletions, since tuples may reference tuples that reference  $t_1$ , and so on.

#### Update:

We must consider two cases for update; updates to the referencing relation  $r_2$  and updates to the referenced relation  $r_1$ .

If a tuple  $t_2$  is updated in relation  $r_2$  and the update modifies values for foreign key  $a$ , then a test similar to the insert case is made: Let  $t_2'$  denote the new value of tuple  $t_2$ .

The system must ensure that

$$t_2'[\alpha] \in \Pi_K(r_1)$$

If a tuple  $t_1$  is updated in  $r_1$ , and the update modifies values for the primary key ( $K$ ), then a test similar to the delete case is made. The system must compute

$$sa_{=r_1[K]}(r_2)$$

using the old value of  $t_1$  (the value before the update is applied). If this set is not empty, the update may be rejected as an error, or the update may be cascaded to the tuples in the set, or the tuples in the set may be deleted.

### 7.2.3. Referential Integrity in SQL

Foreign keys can be specified as part of the SQL **create table** statement by using the **foreign key** clause. We illustrate foreign key declarations by using the SQL DDL definitions of part of our bank database shown in table below

<b>create</b> ( <i>customer-name</i> <i>customer-street</i> <i>customer-city</i> <b>primary key</b> ( <i>customer-name</i> ))	<b>table</b> char(20), char(30), char(30),	<i>customer</i>
<b>create</b> ( <i>branch-name</i> <i>branch-city</i> <i>assets</i> <b>primary key</b> ( <i>branch-name</i> ))	<b>table</b> char(15), char(30), integer,	<i>branch</i>
<b>create</b> ( <i>account-number</i> <i>branch-name</i> <i>balance</i> <b>primary</b> <b>foreign key</b> ( <i>branch-name</i> ) <b>references</b> <i>branch</i> )	<b>table</b> char(10), char(15), integer, <b>key</b>	<i>account</i>  ( <i>account-number</i> ),
<b>create</b> ( <i>customer-name</i> char(20), <i>account-number</i> char(10), <b>primary key</b> ( <i>customer-name</i> , <i>account-number</i> ), <b>foreign key</b> ( <i>account-number</i> ) <b>references</b> <i>account</i> , <b>foreign key</b> ( <i>customer-name</i> ) <b>references</b> <i>customer</i> )	<b>table</b>	<i>depositor</i>

The primary key clause lists attributes that comprise the **primary key**. The **unique key** clause lists attributes that comprise a candidate key. The **foreign key** clause lists the attributes that comprise the foreign key and the name of the relation referenced by the foreign key. A foreign key references the primary key attributes of the referenced table. SQL supports a version of the references clause where a list of attributes of the referenced relation can be specified explicitly. The specified list of attributes must be declared as a candidate key of the referenced relation. We

can use the following short form as part of an attribute definition to declare that the attribute forms a foreign key.

Branch-name **char(15) references** branch

When a referential-integrity constraint is violated, the normal procedure is to reject the action that caused the violation.

Consider this definition of an integrity constraint on the relation *account*:

```
create table account
    ...
    foreign key(branch-name) references branch
    on delete cascade
    on update cascade
    ... )
```

Due to the **on delete cascade** clauses, if a delete of a tuple in branch results in referential-integrity constraint violation, the delete “cascades” to the account relation, deleting the tuple that refers to the branch that was deleted. The cascading updates are similar.

If there is a chain of foreign-key dependencies across multiple relations, with on delete cascade specified for each dependency, a deletion or update at one end of the chain can propagate across the entire chain. If a cascading update to delete causes a constraint violation that cannot be handled by a further cascading operation, the system aborts the transaction. As a result, all the changes caused by the transaction and its cascading actions are undone.

Referential integrity is only checked at the end of a transaction. Intermediate steps are allowed to violate referential integrity provided later steps remove the violation. Otherwise it would be impossible to create some database states, e.g. insert two tuples whose foreign keys point to each other.

The Null values in foreign key attributes complicate SQL referential integrity semantics, and are best prevented using **not null**. If any attribute of a foreign key is null, the tuple is defined to satisfy the foreign key constraint.

An **assertion** is a predicate expressing a condition that we wish the database always to satisfy. The domain constraints and referential-integrity constraints are special forms of assertions.

An assertion in SQL takes the form

```
create assertion <assertion-name> check <predicate>
```

Here is how the two examples of constraints can be written. Since SQL does not provide a “for all X, P(X)” construct (where P is predicate), we are forced to implement the construct by the equivalent “not exists” X such that not P(X)” construct, which can be written in SQL. We write

```
create assertion sum-constraint check
(not exists (select * from branch
            where (select sum(amount) from loan
                  where loan.branch-name =
                    branch.branch-name)
            >= (select sum(amount) from account
              where loan.branch-name =
                branch.branch-name)))
```

Every loan has at least one borrower who maintains an account with a minimum balance or \$1000.00.

```
create assertion balance-constraint check
(not exists (
  select * from loan
    where not exists (
      select *
        from borrower, depositor, account
        where loan.loan-number = borrower.loan-number
          and borrower.customer-name = depositor.customer-name
          and depositor.account-number = account.account-number
          and account.balance >= 1000)))
```

When an assertion is made, the system tests it for validity, and tests it again on every update that may violate the assertion. This testing may introduce a significant amount of overhead; hence assertions should be used with great care.

## 7.4. Triggers :

A **trigger** is a statement that is executed automatically by the system as a side effect of a modification to the database. To design a trigger mechanism

1. Specify the conditions under which the trigger is to be executed.
2. Specify the actions to be taken when the trigger executes.

The above model of triggers is referred to as the **event-condition-action model** for triggers. The database stores triggers just as if they were regular data, so that they are persistent and are accessible to all database operations. Once we enter a trigger into the database, the database system takes on the responsibility of executing it whenever the specified event occurs and corresponding condition is satisfied.

### 7.4.1. Need For Triggers :

Triggers are useful mechanisms for altering humans or for doing certain tasks automatically when certain conditions are met. For example a warehouse wishes to maintain a minimum inventory of each item; when the inventory level of an item falls below the minimum level, an order should be placed automatically. This is how the business rule can be implemented by triggers. On an update of the inventory level of an item, the trigger should compare the level with the minimum inventory level for the item, and if the level is at or below the minimum, a new order is added to an order relation.

### 7.4.2. Triggers in SQL :

The trigger definition specifies that the trigger is initiated after any update of the relation account is executed. An SQL update statement could update multiple tuples of the relation, and the **for each row** clause in trigger code would then explicitly iterate over each updated row. The **referencing row as** clause creates a variable *nrow* (called a **transition variable**), which stores the value of an updated row after the update.

```
create trigger overdraft-trigger after update on account
referencing new row as nrow
for each row
```

```

when nrow.balance < 0
begin atomic
  insert into borrower
    (select customer-name, account-number
     from depositor
     where nrow.account-number =
           depositor.account-number);
  insert into loan values
    (n.row.account-number, nrow.branch-name,
     – nrow.balance);
  update account set balance = 0
  where account.account-number = nrow.account-number
end

```

The **when** statement specifies a condition, namely  $nrow.balance < 0$ . The system executes the rest of the trigger body only for the tuple that satisfy the condition. The **begin atomic ... end** clause serves to collect multiple SQL statements into a single compound statement. The two **insert** statements with the **begin ... end** structure carry out the specific tasks of creating new tuples in the borrower and loan relations to represent the new loan. The **update** statement serves to set the account balance back to 0 from its earlier negative value.

The triggering event and actions can take many forms:

Triggering event can be **insert**, **delete** or **update**. Triggers on update can be restricted to specific attributes.

**Example :** **create trigger** *overdraft-trigger* **after update of** *balance* **on** *account*

Values of attributes before and after an update can be referenced

**referencing old row as** : for deletes and updates

**referencing new row as** : for inserts and updates

Triggers can be activated before an event, which can serve as extra constraints. Example, convert blanks to null.

```

create trigger setnull-trigger before update on r
referencing new row as nrow
for each row
  when nrow.phone-number = ''
  set nrow.phone-number = null

```

## 7.5. Security and Authorization:

The data stored in the database need protection from unauthorized access and malicious destruction or alteration, in addition to the protection against accidental introduction of inconsistency that integrity constraints provide.

### 7.5.1. Security Violations:

The forms of malicious access are:

- Unauthorized reading of data (theft of information)
- Unauthorized modification of data
- Unauthorized destruction of data

Database security refers to protection from malicious access. Absolute protection of the database from malicious abuse is not possible, but the cost to the perpetrator can be made high enough to deter most if not all attempts to access the database without proper authority.

We must take several security levels to protect the database, these are

**Database system:** Some database system users may be authorized to access only a limited protection of the database. Other user may be allowed to issue queries only. It is the responsibility of the database system to ensure that these authorization restrictions are not violated.

**Operating System:** The weakness in the operating-system security may serve as a means of unauthorized access to the database.

**Network:** Since almost all database systems allow remote access through terminals or networks, software-level security within the network software is as important as physical security, both on the Internet and in private networks.

**Physical:** Sites with computer systems must be physically secured against armed or surreptitious entry by intruders.

**Human:** Users must be authorized carefully to reduce the chance of any user giving access to an intruder in exchange for a bribe or other favors.

Security at all these levels must be maintained if database security is to be ensured. A weakness at a low level of security (physical or human) allows circumvention of strict high-level (database) security measures.

### 7.5.2. Authorization:

We may assign a user several forms of authorization on parts of the database. For example,

- **Read authorization** - allows reading, but not modification of data.
- **Insert authorization** - allows insertion of new data, but not modification of existing data.
- **Update authorization** - allows modification, but not deletion of data.
- **Delete authorization** - allows deletion of data.

In addition to these forms authorization for access to data, we may grant a user authorization to modify the database schema:

- **Index authorization** - allows creation and deletion of indices.
- **Resources authorization** - allows creation of new relations.
- **Alteration authorization** - allows addition or deletion of attributes in a relation.
- **Drop authorization** - allows deletion of relations.

### 7.5.3. Authorization and Views :

Users can be given authorization on views, without being given any authorization on the relations used in the view definition. Ability of views to hide data serves both to simplify usage of the system and to enhance security by allowing users access only to data they need for their job. A combination of relational-level security and view-level security can be used to limit a user's access to precisely the data that user needs.

For example, a bank clerk needs to know the names of the customers of each branch, but is not authorized to see specific loan information. Thus, the clerk must be denied to have access to the *loan* relation, but grant access to the view *cust-loan*, which consists only of the names of customers

and the branches at which they have a loan. This can be defined as

```
create view cust-loan as
  select branchname, customer-name
  from borrower, loan
  where borrower.loan-number = loan.loan-number
```

The clerk is authorized to see the result of the query:

```
select *
from cust-loan
```

When the query processor translates the result into a query on the actual relations in the database, we obtain a query on *borrower* and *loan*. Authorization must be checked on the clerk's query before query processing replaces a view by the definition of the view.

Creation of view does not require **resources** authorization since no real relation is being created. The creator of a view gets only those privileges that provide no additional authorization beyond that he already had. For example, if creator of view *cust-loan* had only **read** authorization on *borrower* and *loan*, he gets only **read** authorization on *cust-loan*.

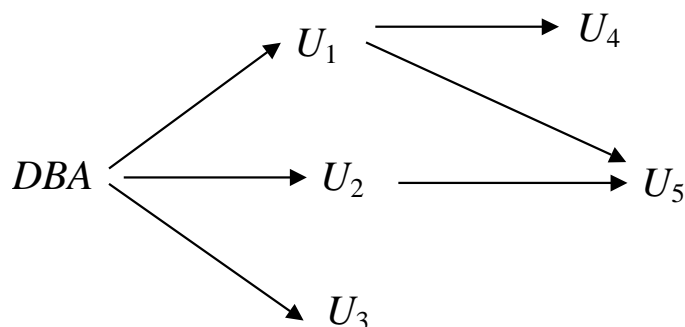
#### 7.5.4. Granting of Privileges:

A user who has been granted some form of authorization may be allowed to pass on this authorization to others users. However, we must be careful how authorization may be passed among users, to ensure that such authorization can be revoked at some future time.

The passage of authorization from one user to another may be represented by an authorization graph. The nodes of this graph are the users. The root of the graph is the database administrator. Consider graph for update authorization on loan.

An edge  $U_i \diamond U_j$  indicates that user  $U_i$  has granted update authorization on loan to  $U_j$ .

for update



A user has an authorization if and only if there is a path from the root of the authorization graph down to the node representing the user.

Suppose if Database Administrator decides to revoke the authorization of user  $U_1$ , since  $U_4$  has authorization from  $U_1$ , that authorization should be revoked as well. However,  $U_5$  was granted authorization by both  $U_1$  and  $U_2$ . If  $U_2$  eventually revokes authorization from  $U_5$ ,  $U_5$  loses the authorization.

### 7.5.5. Privileges in SQL :

SQL offers a fairly powerful mechanism for defining authorizations. Privileges is one of the mechanism which includes **delete**, **insert**, **select** and **update**. The select privilege corresponds to the **read** privilege. SQL also includes a reference privileges that permits a user/role to declare foreign keys when creating relations.

The SQL data definition language includes commands to grant and revoke privileges. The **grant** statement is used to confer authorization. The basic form of the statement is

```
grant <privilege list>
      on <relation name or view name> to <user list>
```

<user list> is a user-id and it is public, which allows all valid users the privilege granted. Granting a privilege on a view does not imply granting any privileges on the underlying relations. The grantor of the privilege must already hold the privilege on the specified item (or be the database administrator).

The following grant statement grants users  $U_1$ ,  $U_2$ , and  $U_3$  select authorization on the relation. It allows read access to relation, or the ability to query using the view. For example,

**grant** users  $U_1$ ,  $U_2$ , and  $U_3$  **select** authorization on the branch relation:

```
grant select on branch to  $U_1$ ,  $U_2$ ,  $U_3$ 
```

**insert**: The ability to insert tuples.

**update**: The ability to update using the SQL update statement.

**delete**: The ability to delete tuples.

**references**: Ability to declare foreign keys when creating relations.

**usage**: Authorizes a user to use a specified domain

**all privileges**: Used as a short form for all the allowable privileges

### 7.5.6. Roles:

Roles permit common privileges for a class of users, can be specified just once by creating a corresponding "role". Privileges can be granted to or revoked from roles, just like user. Roles can be assigned to users, and even to other roles. Roles can be created by SQL 1999 as follows:

```
create role teller
create role manager

grant select on branch to teller
grant update (balance) on account to teller
grant all privileges on account to manager
grant teller to manager
grant teller to alice, bob
grant manager to
```

Thus the privileges of a user or a role consists of

- All privileges directly granted to the user/role.
- All privileges granted to roles that have been granted to the user/role.



### 7.5.7. Limitations of SQL Authorization:

The current SQL standards for authorization have some shortcomings. Suppose, we cannot restrict students to see only (the tuples storing) their own grades, but not the grades of anyone else, authorization must then be at the level of individual tuples, which is not possible in the SQL standards authorization.

Furthermore, with the growth in Web access to databases, database accesses come primarily from application servers. End users don't have database user id's, they are all mapped to the same database user id. All end-users of an application (such as a web application) may be mapped to a single database user. The task of authorization in above cases on the application program has no support from SQL.

- **Benefit:** Fine grained authorizations, such as to individual tuples, can be implemented by the application.
- **Drawback:** Authorization must be done in application code, and may be dispersed all over an application
- Checking for absence of authorization loopholes becomes very difficult since it requires reading large amounts of application code

### 7.6. Encryption and Authentication :

The various provisions that a database system may take for authorization may still not provide sufficient protection for highly sensitive data. In such cases, data may be stored in encrypted form. It is not possible for **encrypted** data to be read unless the reader knows how to decipher (**decrypt**) them. Encryption also forms the basis of good schemes for authenticating user to a database.

**Encryption:** There are a vast number of techniques for the encryption of data. Simple encryption techniques may not provide adequate security, since it may be easy for an unauthorized user to break the code. As an example of a weak encryption technique, consider the substitution of each character with the next character in the alphabet, thus

Perryridge

Becomes

Qfsszsjehf

Properties of good encryption technique:

- Relatively simple for authorized users to encrypt and decrypt data.
- Encryption scheme depends not on the secrecy of the algorithm but on the secrecy of a parameter of the algorithm called the encryption key.
- Extremely difficult for an intruder to determine the encryption key.

**Data Encryption Standard (DES)** substitutes characters and rearranges their order on the basis of an encryption key which is provided to authorized users via a secure mechanism. Scheme is no more secure than the key transmission mechanism since the key has to be shared.

**Advanced Encryption Standard (AES)** is a new standard replacing DES, and is based on the Rijndael algorithm, but is also dependent on shared secret keys.

**Public-key encryption** is based on each user having two keys:

- *public key* – publicly published key used to encrypt data, but cannot be used to decrypt data.

- *private key* — key known only to individual user, and used to decrypt data.

Encryption scheme is such that it is impossible or extremely hard to decrypt data given only the public key. The RSA public-key encryption scheme is based on the hardness of factoring a very large number (100's of digits) into its prime components.

**Authentication:** Authentication refers to the task of verifying the identity of a person/software connecting to a database. The simplest form of Authentication consists of a secret password, which must be presented when a connection is opened to a database. Password based authentication is widely used, but is susceptible to sniffing on a network. If an eavesdropper is able to “sniff” the data being sent over the network, she may be able to find the password as it is being sent across the network. Once the eavesdropper has a user name and password, she can connect to the database, pretending to be the legitimate user.

A more secure scheme involves a **challenge-response** system. The database system sends a challenge string to the user. The user encrypts the challenge string using a secret password as encryption key, and then returns the result. The database system can verify the authenticity of the user by decrypting string with the same secret password, and checking the result with the original challenge string. This scheme ensures that no passwords travel across the network. User can use public-key encryption system by DB sending a message encrypted using user's public key, and user decrypting and sending the message back.

The interesting application of public-key encryption is in **digital signatures** to verify authenticity of data; digital signatures play the electronic role of physical signatures on documents. The private key is used to sign data, and the signed data can be made public. Anyone can verify them by the public key, but no one could have generated the signed data without having the private key.

## 7.7. Summary:

In earlier chapters, we considered several forms of constraints, including key declarations and the declaration of the form of a relationship. In this chapter, we considered several additional forms of constraints, and discussed mechanisms for ensuring the maintenance of these constraints. Domain constraints specify the set of possible values that may be associated with an attribute. Such constraints may also prohibit the use of null values for particular attributes. Referential-integrity constraints ensure a value that appears in one relation for a given set of attributes also appears for a certain set of attributes in another relation.

A user may have several forms of authorization on parts of the database. Authorization is a means by which the database system can be protected against malicious or unauthorized access. A user who has been granted some form of authority may be allowed to pass on this authority to other users. However, we must be careful about how authorization can be passed among users if we are to ensure that such authorization can be revoked at some future time.

## 7.8. Technical Terms:

**Referential Integrity:** An integrity constraint specifying that the value (or existence) of an attribute in one relation depends on the value (or existence) of an attribute in the same or another relation.

**Primary Key:** A set of one or more columns in a database table whose values, in combination, are required to be unique within the table.

**Trigger:** A set of Structured Query Language (SQL) statements that automatically “fires off” an action when a specific operation, such as changing data in a table, occurs.

**Encryption:** The process of coding data so that a specific code or key is required to restore the original data, used to make transmissions secure from unauthorized reception.

**Authorization:** The process of determining what types of activities are permitted. Usually, authorization is in the context of authentication: once you have authenticated a user, they may be authorized different types of access or activity.

### 7.9. Model Questions:

1. Write about Referential Integrity?
2. Explain the concept of Assertions?
3. What are triggers? How to create triggers in SQL?
4. Write about Security and Authorization?

### 7.10. References:

***Database System Concepts***

*Silberschatz, Korth, and Sudarshan*

***Database Management Systems***

*Arun K. Majumdar, Pritimoy Bhattacharyya*

***An Introduction to Database Systems***

*Bipin Desai*

***Modern Database Management***

*F. McFadden, J. Hoffer*

***An Introduction to Database Systems***

*C. J. Date;*

**AUTHOR:**

**Y.SURESH BABU.**, M.Com, M.C.A.,  
Lecturer,  
Dept. Of Computer Science,  
JKC College.  
GUNTUR.

## Lesson 8

# Normalization - 1

### 8.0 Objectives:

The basic objective of this Lesson is gaining knowledge of information and database design process.

Students will learn the general principles involved in designing a database that complies with the relational database model.

After reading this chapter, you should understand:

- To learn what is the purpose of database design?
- To learn basic Normal Forms
- To learn how to remove redundancies and anomalies
- To Define what a Functional Dependency is
- To learn about the Decomposition

### Structuer of the Lesson :

- 8.1.1. Introduction
- 8.1.2. Rules For Data Normalization
- 8.1.3. First Normal Form
- 8.1.4. Pitfalls in Relational – Database Design
- 8.1.5. Functional Dependencies
  - 8.1.5.1. Basic Concepts
  - 8.1.5.2. Closure Set of Functional Dependencies
  - 8.1.5.3. Closure Of Attribute Sets
  - 8.1.5.4. Canonical Cover
- 8.1.6. Desirable Properties Of Decomposition
  - 8.1.6.1. Loss Less Decomposition
  - 8.1.6.2. Dependency Preservation
- 8.1.7. Third Normal Form
- 8.1.8. Decomposition Algorithm
- 8.1.9. Summary
- 8.1.10. Technical Terms
- 8.1.11. Model Questions
- 8.1.12. References

## 8.1. Introduction:

Whenever we design databases we are faced with a number of problems relating to things like data integrity, security, efficiency. We are also faced with problems relating to the structure of the data we are planning to use.

Normalization is a design technique that is widely used as a guide in designing relational databases. Normalization is essentially a two-step process that puts data into tabular form by removing repeating groups and then removes duplicated data from the relational tables. Normalization theory is based on the concepts of **normal forms**. A relational table is said to be a particular normal form if it satisfied a certain set of constraints. There are currently five normal forms that have been defined. In this section, we will cover the first three normal forms that were defined by E. F. Codd.

Normalization theory is built around the concept of normal forms. A relation is said to be in a particular normal form if it satisfies a certain specified set of constraints.

For example, a relation is said to be in first normal form (abbreviated 1NF) if and only if it satisfies the constraint that it contains atomic values only (thus every normalized relation is in 1NF, which accounts for the "first"). Numerous normal forms have been defined.

## 8.2. Rules For Data Normalization :

### 1NF:

Eliminate Repeating Groups - Make a separate table for each set of related attributes, and give each table a primary key.

### 2NF

Eliminate Redundant Data - If an attribute depends on only part of a multi-valued key, remove it to a separate table.

### 3 NF:

Eliminate Columns Not Dependent On Key - If attributes do not contribute to a description of the key, remove them to a separate table.

### BCNF:

Boyce-Codd Normal Form - If there are non-trivial dependencies between candidate key attributes, separate them out into distinct tables.

### 4 NF:

Isolate Independent Multiple Relationships - No table may contain two or more 1:n or n:m relationships that are not directly related.

### 5 NF:

Isolate Semantically Related Multiple Relationships - There may be practical constraints on information that justify separating logically related many-to-many relationships.

## 8.3. First Normal Form:

The first of the normal forms that we study, **first normal**, imposes a very basic requirement on relations; unlike the other normal forms, it does not require additional information such as functional dependencies.

A domain is **atomic** if elements of the domain are considered to be indivisible units. We say that a relation schema  $R$  is in **first normal form** (1NF) if the domains of all attributes of  $R$  are atomic.

A set of names is an example of a non atomic value. For example, if the schema of a relation *employee* included an attribute *children* whose domain elements are sets of names, the schema would not be in first normal form.

*Composite attributes, such as an attribute address with component attributes street and city, also have non atomic domains.*

Integers are assumed to be atomic, so the set of integers is an atomic domain; the set of all sets of integers is a non atomic domain. The distinction is that we do not normally consider integers to have subparts, but we consider sets of integers to have subparts—namely, the integers making up the set. But the important issue is not what the domain itself is, but rather how we use domain elements in our database.

The domain of all integers would be non atomic if we considered each integers to be an ordered list of digits.

As a practical illustration of the point, consider an organization that assigns employees identification numbers of the following form: The first two letters specify the department and the remaining four digits are a unique number within the department for the employee. Examples of such numbers would be CS0012 and EE1127. Such identification numbers can be divided into smaller units, and are therefore non atomic. If a relation schema had an attribute whose domain consists of identification numbers encoded as above the schema would not be in first normal form.

When such identification numbers are used, the department of an employee can be found by writing code that breaks up the structure of an identification number. Doing so requires extra programming, and information gets encoded in the application program rather than in the database. Further problems arise if such identification numbers are used as primary keys: When an employee changes department, the employees identification number must be changed everywhere it occurs, which can be a difficult task, or the code that interprets the number would give a wrong results.

The use of set valued attributes can lead to designs with redundant storage of data, which in turn can result inconsistencies. For instance, instead of the relationship between accounts and customers being represented as a separate relations *depositor*, a database designer may be tempted to store a set *owners* with each account, and a set of *accounts* with each customer. Whenever an account is created, or the set of owners of an account is updated, the update has to be performed at two paces; failure to perform both updates can leave the database in an inconsistent state. Keeping only one of these sets would avoid repeated information, but would complicate some queries. Set valued attributes are also more complicated to write queries with, and, or complicated to reason about.

#### 8.4. Pitfalls in Relational -Database Design:

Before we continue our discussion of normal forms, let us look at what can go wrong in bad database design. Among the undesirable properties that a bad design may

- Repetition of information
- Inability to represent certain information

We shall discuss these problems with the help of a modified database design for our banking example: suppose the information concerning loans is kept in one single relation, *lending*, which is defined over the relation schema

*Lending-schema* = (*branch-name*, *branch-city*, *assets*,  
*customer-name*, *loan-number*, *amount*)

Below table, shows an instance of the relation *lending*(*lending-schema*). A tuple *t* in the *lending* relation has the following intuitive meaning:

- *t[assets]* is the asset figure for the branch named *t[branch-name]*.
- *t[branch-city]* is the city in which the branch named *t[branch-name]* is located.
- *t[loan-number]* is the number assigned to a loan made by the branch named *t[branch-name]* to the customer named *t[customer-name]*.
- *t[amount]* is the amount of the loan whose numbers is *t[loan-name]*.

Suppose that we wish to add a new loan to our database. Say, the loan is made by the Perryridge branch to Adams an amount of \$1500. Let the *loan-number* be L-31. In our design, we need a tuple with values on all the attributes of *lending-schema*. Thus, we must repeat the asset and city data for the Perryridge branch, and must ass the tuple. (Perryridge, Horseneck, 1700000,Adams, L-31,1500)

<i>branch-name</i>	<i>branch-city</i>	<i>assets</i>	<i>customer-name</i>	<i>loan-number</i>	<i>amount</i>
Downtown	Brooklyn	9000000	Jones	L-17	1000
Redwood	Palo Alto	2100000	Smith	L-23	2000
Perryridge	Horseneck	1700000	Hayes	L-15	1500
Downtown	Brooklyn	9000000	Jackson	L-14	1500
Mianus	Horseneck	400000	Jones	L-93	500
Round Hill	Horseneck	8000000	Turner	L-11	900
Pownal	Bennington	300000	Williams	L-29	1200
North Town	Rye	3700000	Hayes	L-16	1300
Downtown	Brooklyn	9000000	Johnson	L-18	2000
Perryridge	Horseneck	1700000	Glenn	L-25	2500
Brighton	Brooklyn	7100000	Brooks	L-10	2200

### Sample Lending Relation

To the *lending* relation in general, the set and city data for a branch must appear once for each loan made by that branch.

The repetition of information in our alternative design is undesirable. Repeating information wastes space. Furthermore, it complicates updating the database. Suppose, for example, that the assets of the Perryridge branch change from 1700000 to 1900000. Under our original design, one tuple of the *branch* relations needs to be changed. Under our alternative design, many tuples of the *lending* relations need to be changed. Thus, updates are more costly under the alternative design than under the original design. When we perform the update in the alternative database, we must ensure that every tuple pertaining to the Perryridge branch is updated, or else our database will show two different asset values for the Perryridge branch.

That observation is central to understanding why the alternative design is bad. We know that a bank branch has a unique value of assets, so given a branch name we can uniquely identify the assets value. On the other hand, we know that a branch may make many loans, so given a branch name we cannot uniquely determine a loan number.

In other words, we say that the *functional dependency*

$$\text{Branch-name} \rightarrow \text{assets}$$

Holds on *Lending-schema*, but we do not expect the functional dependency *branch-name*  $\rightarrow$  *loan-number* to hold. The fact that a branch has a particular value of assets, and the fact that a branch makes a loan are independent.

## 8.5. Functional Dependencies:

Functional dependencies play a key role in differentiating good database designs from bad database designs. A **functional dependency** is a type of constraint that is a generalization of the notion

### 8.5.1. Basic Concepts:

Functional dependencies are constraints on the set of legal relations. They allow us to express facts about the enterprise that we are modeling with our database. We defined the notion of a *superkey* as follows. Let  $R$  be a relation schema. A subset  $K$  of  $R$  is a **superkey** of  $R$  if, in any legal relation  $r(R)$ , for all pairs  $t_1$  and  $t_2$  of tuples in such that  $t_1 \neq t_2$ , then  $t_1[k] \neq t_2[k]$ . That is, no two tuples in any legal relation  $r(R)$  may have the same value on attribute set  $k$ .

The notion of functional dependency generalizes the notion of super key. Consider a relation schema  $R$ , let  $\alpha \subseteq R$  and  $\beta \subseteq R$ . The **functional dependency**

$$\alpha \rightarrow \beta$$

Holds on schema  $R$  if, in any legal relations  $r(R)$ , for all pairs of tuple  $t_1$  and  $t_2$  in  $r$  such that  $t_1[\alpha] = t_2[\alpha]$ , it is also the case that  $t_1[\beta] = t_2[\beta]$ ,

Using the functional-dependency notation, we say that  $K$  superkey of  $R$  if  $K \rightarrow R$ . That is  $K$  is a super key if, whenever  $t_1[k] = t_2[k]$ , it is also the case that  $t_1[R] = t_2[R]$ , (that is,  $t_1 = t_2$ ).

Functional dependencies allow us to express constraints that we cannot express with super keys. Consider the schema

Loan-info-schema = (customer-name, loan-number,  
branch-name, amount)

Which is simplification of the lending-schema that we saw earlier. The set of functional dependencies that we expect to hold on this relation schema is

loan-number  $\rightarrow$  amount

loan-number  $\rightarrow$  branch-name

We would not, however, expect the functional dependency

loan-number  $\rightarrow$  customer-name

to hold, since, in general, a given loan can be made to more than one customer (for example to both members of a husband-wife pair).



1. To test relations to see whether they are legal under a given set of functional dependencies. If a relation  $r$  is legal under a set  $F$  of functional dependencies, we say that  $r$  **satisfies**  $F$ .
2. To specify constraints on the set of legal relations. We shall thus concern ourselves with only those relations that satisfy a given set of functional dependencies. If we wish to constrain ourselves to relations on schema  $R$  that satisfy a set  $F$  of functional dependencies, we say that  $F$  holds on  $R$ .

Consider a relation  $R$ , attribute  $Y$  of  $R$  is functionally dependent on attribute  $X$  of  $R$  if and only if each  $X$ -value in  $R$  has associated with it precisely one  $Y$ -value in  $R$  (at any one time). It is represented by  **$R.X \rightarrow R.Y$**

The concept of functional dependencies is the basis for the first three normal forms. A column,  $Y$ , of the relational table  $R$  is said to be **functionally dependent** upon column  $X$  of  $R$  if and only if each value of  $X$  in  $R$  is associated with precisely one value of  $Y$  at any given time.  $X$  and  $Y$  may be composite. Saying that column  $Y$  is functionally dependent upon  $X$  is the same as saying the values of column  $X$  identify the values of column  $Y$ . If column  $X$  is a primary key, then all columns in the relational table  $R$  must be functionally dependent upon  $X$ .

A shorthand notation for describing a functional dependency is:

$$R.x \rightarrow R.y$$

Which can be read as in the relational table named  $R$ , column  $x$  functionally determines (identifies) column  $y$ .

**Full functional dependence** applies to tables with composite keys. Column  $Y$  in relational table  $R$  is fully functional on  $X$  of  $R$  if it is functionally dependent on  $X$  and not functionally dependent upon any subset of  $X$ . Full functional dependence means that when a primary key is composite, made of two or more columns, then the other columns must be identified by the entire key and not just some of the columns that make up the key.

### 8.5.2 Closure Set of Functional Dependencies:

We need to consider *all* functional dependencies that hold. Given a set  $F$  of functional dependencies, we can prove that certain other ones also hold. We say these ones are **logically implied** by  $F$ .

Suppose we are given a relation scheme  $R=(A,B,C,G,H,I)$ , and the set of functional dependencies:

$$A \rightarrow B$$

$$A \rightarrow C$$

$$CG \rightarrow H$$

$$CG \rightarrow I$$

$$B \rightarrow H$$

Then the functional dependency  $A \rightarrow H$  is logically implied.

To see why, let  $t_1$  and  $t_2$  be tuples such that

$$t_1[A] = t_2[A]$$

As we are given  $A \rightarrow B$  it follows that we must also have

$$t_1[B] = t_2[B]$$

Further, since we also have  $B \rightarrow H$  we must also have

$$t_1[H] = t_2[H]$$

Thus, whenever two tuples have the same value on  $A$

they must also have the same value on  $H$  and we can say that  $A \rightarrow H$

The **closure** of a set  $F$  of functional dependencies is the set of all functional dependencies logically implied by  $F$ .

We denote the closure of  $F$  by  $F^+$ . To compute  $F^+$ , we can use some rules of inference called **Armstrong's Axioms**:

- **Reflexivity rule:** if  $\alpha$  is a set of attributes and  $\beta \subseteq \alpha$ , then  $\alpha \rightarrow \beta$  holds.
- **Augmentation rule:** if  $\alpha \rightarrow \beta$  holds, and  $\gamma$  is a set of attributes, then  $\gamma\alpha \rightarrow \gamma\beta$  holds.
- **Transitivity rule :** if  $\alpha \rightarrow \beta$  holds, and  $\beta \rightarrow \gamma$  holds, and  $\alpha \rightarrow \gamma$  holds.

These rules are **sound** because they do not generate any incorrect functional dependencies. They are also **complete** as they generate all of  $F^+$ .

To make life easier we can use some additional rules, derivable from Armstrong's Axioms:

- **Union rule:** if  $\alpha \rightarrow \beta$  and  $\alpha \rightarrow \gamma$ , then  $\alpha \rightarrow \beta\gamma$  holds.
- **Decomposition rule:** if  $\alpha \rightarrow \beta\gamma$  holds, then  $\alpha \rightarrow \beta$  and  $\alpha \rightarrow \gamma$  both hold
- **Pseudotransitivity rule :** if  $\alpha \rightarrow \beta$  holds, then  $\gamma\beta \rightarrow \delta$  holds, then  $\alpha\gamma \rightarrow \delta$  holds.

Applying these rules to the scheme and set  $F$  mentioned above, we can derive the following

- $A \rightarrow H$  as we saw by the transitivity rule
- $CG \rightarrow HI$  by the union rule.
- $AG \rightarrow I$  by several steps:
- Note that  $A \rightarrow C$  holds
- Then  $AG \rightarrow CG$  by the augmentation rule
- Now by transitivity,  $AG \rightarrow I$

### 8.5.3. Closure of Attributes Sets :

To test whether a set of attributes  $\alpha$  is a superkey, we need to find the set of attributes functionally determined by  $\alpha$

1. Let  $\alpha$  be a set of attributes. We call the set of attributes determined by  $\alpha$  under a set of  $F$  functional dependencies the **closure** of  $\alpha$  under  $F$ , denoted  $\alpha^+$ .

2. The following algorithm computes  $\alpha^+$

result =  $\alpha$

while (changes to result) do

  for each functional dependency  $\beta \rightarrow \gamma$

    in  $F$  do

      begin

        if  $\beta \subseteq \text{result}$

          then result := result  $\cup \gamma$

      end

3. If we use this algorithm on our example to calculate  $(AG^+)$  then we find;
- We start with result = AG
  - causes *result* to become ABCG
  - causes *result* to become ABCGH
  - I causes *result* to become ABCGHI.
  - The next time we execute the while loop, no new attributes are added, and the algorithm terminates.

#### 8.5.4. Canonical Cover:

To minimize the number of functional dependencies that need to be tested in case of an update we may restrict **F** to a **canonical cover**  $F_e$

A canonical cover for **F** is a set of dependencies such that **F** logically implies all dependencies in  $F_e$  and vice versa.

$F_e$  must also have the following properties:

Every functional dependency  $\alpha \rightarrow \beta$  in  $F_e$  contains no **extraneous** attributes in  $\alpha$  (ones that can be removed from  $\alpha$  without changing  $F_e^+$ ). So  $A$  is extraneous in  $\alpha$  if  $A \in \alpha$  and

$$(F_e - \{\alpha \rightarrow \beta\}) \cup \{\alpha - A \rightarrow \beta\}$$

logically implies  $F_e$

Every functional dependency  $\alpha \rightarrow \beta$  in  $F_e$  contains no **extraneous** attributes in  $\beta$  (ones that can be removed from  $\beta$  without changing  $F_e^+$ ). So  $A$  is extraneous in  $\beta$  if  $A \in \beta$

$$(F_e - \{\alpha \rightarrow \beta\}) \cup \{\alpha \rightarrow \beta - A\}$$

logically implies  $F_e$

Each left side of a functional dependency in  $F_e$  is unique. That is there are no two dependencies  $\alpha_1 \rightarrow \beta_1$  and  $\alpha_2 \rightarrow \beta_2$  in  $F_e$  such that  $\alpha_1 = \alpha_2$ .

To compute a canonical cover  $F_e$  for  $F$ ,

- o Use the union rule to replace dependencies of the form  $\alpha_1 \rightarrow \beta_1$  and  $\alpha_1 \rightarrow \beta_2$  with  $\alpha \rightarrow \beta_1\beta_2$
- o Test each functional dependency  $\alpha \rightarrow \beta$  to see if there is an extraneous attribute in  $\alpha$
- o Continue until there are no changes occurring in the loop.

An example: for the relational scheme  $R = (A, B, C)$ , and the set  $F$  of functional dependencies.

$$A \rightarrow BC$$

$$B \rightarrow C$$

$$A \rightarrow B$$

$$AB \rightarrow C$$

We will compute  $F_e$

We have two dependencies with the same left hand side.

$$A \rightarrow BC$$

$$A \rightarrow B$$

We can replace these two with just  $A \rightarrow BC$

A is extraneous in  $AB \rightarrow C$  because  $B \rightarrow C$  logically implies  $AB \rightarrow C$

Then our set is

$$A \rightarrow BC$$

$$B \rightarrow C$$

We still have an extraneous attribute on the right-hand side of the first dependency. C is extraneous in  $A \rightarrow BC$  because  $A \rightarrow B$  and  $B \rightarrow C$  logically imply that  $A \rightarrow B$

- So we end up with

$$A \rightarrow B$$

$$B \rightarrow C$$

Consider a schema

$$\text{Lending-schema} = (\text{bname}, \text{assets}, \text{bcity}, \text{loan\#}, \text{cname}, \text{amount})$$

which we saw was a bad design.

The set of functional dependencies we required to hold on this schema was:

$$\text{bname} \rightarrow \text{assets bcity}$$

$$\text{loan\#} \rightarrow \text{amount bname}$$

If we decompose it into

$$\text{Branch-schema} = (\text{bname}, \text{assets}, \text{bcity})$$

If we decompose it into

$$\text{Branch-schema} = (\text{bname}, \text{assets}, \text{bcity})$$

$$\text{Loan-info-schema} = (\text{bname}, \text{loan\#}, \text{amount})$$

$$\text{Borrow-schema} = (\text{cname}, \text{loan\#})$$

we claim this decomposition has several desirable properties

### 8.6.1. Loss Less Decomposition:

We claim the above decomposition is lossless. How can we decide whether decomposition is lossless?

- Let  $\mathbf{R}$  be a relation scheme.
- Let  $\mathbf{F}$  be a set of functional dependencies on  $\mathbf{R}$ .
- Let  $\mathbf{R}_1$  and  $\mathbf{R}_2$  form a decomposition of  $\mathbf{R}$ .
- The decomposition is a lossless-join decomposition of  $\mathbf{R}$  if at least one of the following functional dependencies are in  $\mathbf{F}^+$ 
  1.  $R_1 \cap R_2 \rightarrow R_1$
  2.  $R_1 \cap R_2 \rightarrow R_2$

Why is this true? Simply put, it ensures that the attributes involved in the natural join  $(R_1 \cap R_2)$  are a candidate key for at least one of the two relations.

This ensures that we can never get the situation where spurious tuples are generated; as for any value on the join attributes there will be a unique tuple in **one** of the relations.

We'll now show our decomposition is loss less-join by showing a set of steps that generate the decomposition:

- First we decompose *Lending-scheme* into  
*Branch-scheme* = (*bname*, *assets*, *bcity*)  
*Borrow-scheme* = (*bname*, *loan#*, *cname*, *amount*)
- Since *bname* → *assets* *bcity*, the augmentation rule for functional dependencies implies that  
*bname* → *bname assets bcity*
- Since *Branch-scheme* ∩ *Borrow -scheme* = *bname*, our decomposition is lossless join.
- Next we decompose *Borrow-scheme* into  
*Loan-info-schem* = (*bname*, *loan#*, *amount*)  
*Cust-loan-scheme* = (*cname*, *loan#*)
- As *loan#* is the common attribute, and  
*loan #* → *amount bname*  
 This is also a loss less-join decomposition

### 8.6.2. Dependency Preservation:

Another desirable property in database design is **dependency preservation**.

We would like to check easily that updates to the database do not result in illegal relations being created.

It would be nice if our design allowed us to check updates without having to compute natural joins.

To know whether joins must be computed, we need to determine what functional dependencies may be tested by checking each relation individually.

Let  $F$  be a set of functional dependencies on schema  $R$ .

Let  $\{R_1, R_2, \dots, R_n\}$  be a decomposition of  $R$ .

The **restriction** of  $F$  to  $R_i$  is the set of all functional dependencies in  $F^+$  that include only attributes of  $R_i$ .

Functional dependencies in a restriction can be tested in one relation, as they involve attributes in one relation schema.

The test of restrictions  $F_1, F_2, \dots, F_n$  is the set of dependencies that can be checked efficiently.

We need to know whether testing only the restrictions is sufficient.

$$\text{Let } F' = F_1 \dots F_2 \dots F_n$$

$F'$  is a set of functional dependencies on schema  $R$ , but in general  $F' \neq F$ .

However, it may be that  $F'^+ = F^+$

If this is so, then every functional dependency in  $F$  is implied by  $F'$ , and if  $F'$  is satisfied, then  $F$  must also be satisfied.

A decomposition having the property that  $F'^+ = F^+$  is a **dependency-preserving** decomposition.

The algorithm for testing dependency preservation follows this method:

```

compute  $F^+$ 
  for each schema  $R_i$  in  $D$  do
    begin
       $F_i$  = the restriction of  $F^+$  to  $R_i$ 
    end

```

Rather than compute  $F^+$  and  $F'^+$ , and see whether they are equal, we can do this:

Find  $F - F'$  the functional dependencies not checkable in one relation.

See whether this set is obtainable from  $F'$  by using Armstrong's Axioms.

This should take a great deal less work, as we have (usually) just a few functional dependencies to work on.

### 8.7. Third Normal Form:

A relation  $R$  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  $\alpha \rightarrow \beta$ , where  $\alpha \subseteq R$ ,  $\beta \subseteq R$ , at least one of the following holds:

- ❑  $\alpha \rightarrow \beta$  Is trivial (i.e.,  $\beta \in \alpha$ )
- ❑  $\alpha$  Is a super key for  $R$
- ❑ Each attribute  $A$  in  $\beta - \alpha$  is contained in a candidate key for  $R$

### 8.8. Decomposition Algorithm:

Let  $F_c$  be a canonical cover for  $F$ ;

$i := 0$ ;

```

for each functional dependency  $\alpha \rightarrow \beta$  in  $F_c$  do
  if none of the schemas  $R_j, 1 \leq j \leq i$  contains  $\alpha\beta$ 
    then begin
       $i := i + 1$ 
       $R_i = \alpha\beta$ 
    end

```

```

if none of the schemas  $R_j, 1 \leq j \leq i$  contains a candidatekey for R
  then begin
    i:= i+1;
     $R_i :=$  any candidate key for R;
  end
return  $(R_1, R_2, \dots, R_i)$ 
 $F' = \theta$ 
for each restriction  $F_i$  do
  begin
     $F' = F' \cup F_i$ 
  end
compute  $F'^+$ ;
if  $(F'^+ = F^+)$  then return (true)
else return (false);

```

We can now show that our decomposition of *Lending-schema* is dependency preserving.

The functional dependency

$\text{bname} \rightarrow \text{assets bcity}$

can be tested in one relation on *Branch-schema*.

The functional dependency

$\text{loan\#} \rightarrow \text{amount bname}$

can be tested in *Loan-schema*.

As the above example shows, it is often easier not to apply the algorithm shown to test dependency preservation, as computing  $F^+$  takes exponential time.

### An Easier Way To Test For Dependency Preservation

Really we only need to know whether the functional dependencies in  $F$  and not in  $F'$  are implied by those in  $F'$ .

In other words, are the functional dependencies not easily checkable logically implied by those that are?

### 8.9. Summary :

The main purpose of normalization is to maintain well-organized data in terms of tables by reducing redundancy, update anomalies and to simplify the enforcement of integrity constraints. Last but not least, to provide a good design that is easy to understand and provides base to extensibility.

First normal form: A table is in the first normal form if it contains no repeating columns.

Second normal form: A table is in the second normal form if it is in the first normal form and contains only columns that are dependent on the whole (primary) key.

Third normal form: A table is in the third normal form if it is in the second normal form and contains only columns that are not transitively dependent on the primary key.

When you follow these rules, the tables of the model are in the third normal form, according to E. F. Codd, the inventor of relational databases. When tables are not in the third normal form, either redundant data exists in the model, or problems exist when you attempt to update the tables.

### 8.10. Technical Terms:

**Normalization:** A series of steps followed to obtain a database design that allows for efficient access and storage of data. These steps reduce data redundancy and the chances of data becoming inconsistent.

**Transitive dependency:** Let R be a relation and let a, b and c are the attributes of R then we say that R satisfies transitive dependency if there exists  $a \rightarrow b$  and  $b \rightarrow c$  and consequently  $a \rightarrow c$ .

**Functional Dependency:** many-to-one relationship shared by columns of values in database tables. A functional dependency from column X to column Y is a constraint that requires two rows to have the same value for the Y column if they have the same value for the X column.

**Non-Loss Decomposition:** without losing of data, dividing the relation into multiple number of relations called Non loss Decomposition.

### 8.11. Model Questions:

1. What does Data Normalization mean? What are the rules for Normalization?
2. Explain the First normal Form [1NF] with an example?
3. Explain the concept of functional Dependency?
4. What is Decomposition? Write about Loss-less Decomposition?

### 8.12. References:

***Database System Concepts***

*Silberschatz, Korth, and Sudarshan*

***Database Management Systems***

*Arun K. Majumdar, Pritimoy Bhattacharyya*

***An Introduction to Database Systems***

*Bipin Desai*

***Modern Database Management***

*F. McFadden, J. Hoffer*

***An Introduction to Database Systems***

*C. J. Date;*

**AUTHOR:**

**Y.SURESH BABU.,** M.Com, M.C.A.,  
Lecturer,  
Dept. Of Computer Science,  
JKC College.  
GUNTUR.



## Lesson 9

# Normalization-2

### 9.0 Objectives:

After reading this chapter, you should understand:

- To learn about advanced Normal Forms
- To differentiate 3NF and BCNF
- To learn about Fourth Normal form
- To know about Multivalued Dependencies
- To introduce more normal forms fifth and domain-key NF

### Structure Of the Lesson:

- 9.1. Advanced Normalization
- 9.2. Comparison Between 3 NF and BCMF
- 9.3. Fourth Normal Form
  - 9.3.1. Multivalued Dependencies
  - 9.3.2. Definition of Fourth Normal Form
- 9.4. More Normal Forms
- 9.5. Summary
- 9.6. Technical Terms
- 9.7. Model Questions
- 9.8. References

### 9.1. Advanced Normalization:

After 3NF, all normalization problems involve only tables, which have three or more columns, and all the columns are keys. Many practitioners argue that placing entities in 3NF is generally sufficient because it is rare that entities that are in 3NF are not in 4NF and 5NF. They further argue that the benefits gained from transforming entities into 4NF and 5NF are so slight that it is not worth the effort. However, advanced normal forms are presented because there are cases where they are required. ]

### 9.2. Comparison Between 3 NF and BCMF:

Of the two normal forms for relational-database schemas, 3NF and BCNF, there are advantages to 3NF in that we know that it is always possible to obtain a 3NF design without sacrificing a loss less join or dependency preservation. Nevertheless, there are disadvantages to 3NF: If we do not eliminate all transitive relations schema dependencies, we may have to use null values to

represent some of the possible meaningful relationships among the data items, and there is the problem, consider again the *Banker-schema* and its associated functional dependencies. Since  $banker-name \rightarrow branch-name$ , we may want to represent the relationship between values for *banker-name* and values for *branch-name* in our database. If we are to do so, however, either there must be a corresponding value for *customer-name*, or we must use a null value for the attribute *customer-name*.

<i>customer-name</i>	<i>banker-name</i>	<i>branch-name</i>
Jones	Johnson	Perryridge
Smith	Johnson	Perryridge
Hayes	Johnson	Perryridge
Jackson	Johnson	Perryridge
Curry	Johnson	Perryridge
Turner	Johnson	Perryridge

### An instance of Banker-schema.

As an illustration of the repetition of information problem, consider the instance of Banker-schema in the Banker-schema relation. Notice that the information indicating that Johnson is working at the Perryridge branch is repeated.

Recall that our goals of database design with functional dependencies are:

1. BCNF
2. Loss less join
3. Dependency preservation

Since it is not always possible to satisfy all three, we may be forced to choose BCNF and dependency preservation with 3NF.

It is worth noting that SQL does not provide a way of specifying functional dependencies, except for the special case of declaring super keys by using the primary key or unique constraints. It is possible, although a little complicated, to write assertions that enforce a functional dependency-preserving decomposition; if we use standard SQL we would not be able to efficiently test a functional dependency whose left-hand side is not a key.

Although testing functional dependencies may involve a join if the decomposition is not dependency preserving, we can reduce the cost by using materialized views, which many database systems support. Given BCNF decomposition that is not dependency preserving, we consider each dependency in a minimum cover  $F_c$  that is not preserved in the decomposition. For each such dependency  $\alpha \rightarrow \beta$  we define materialized views that computes a join of all relations in the decomposition, and projects the results on  $\alpha\beta$  The functional dependency can be easily tested on the materialized view, by means of a constraint **unique** ( $\alpha$ ) On the negative side there is a space and time overhead due to the materialized view, but on the positive side, the application programmer need not worry about

writing code to keep redundant data consistent on updates. It is the job of the database system to maintain the materialized view, keep up to date when the database is updated.

Thus, in case we are not able to get a dependency-preserving BCNF decomposition, it is generally preferable to opt for BCNF, and use techniques such as materialized views to reduce the cost of checking functional dependencies.

### 9.3. Fourth Normal Form:

Some relation schemas, even though they are in BCNF, do not seem to be sufficiently normalized, in the sense that they still suffer from the problem of repetition of information. Consider again our banking example: Assume that, in an alternative design for the bank database schema, we have the schema:

BC-schema=(loan-number, customer-name, customer-street, customer-city)

The astute reader will recognize this schema as a non-BCNF schema because of the functional dependency.

Customer- name  $\rightarrow$  customer—street customer-city

That we asserted earlier, and because customer- name is not a key for BC-schema. However, assume that our bank is attracting wealthy customers who have several addresses. (Say a winter home and a summer home). Then we no longer wish to enforce the functional dependency Customer- name  $\rightarrow$  customer—street customer-city. If we remove this functional dependency, we find BC-schema to be in BCNF with respect to our modified set of functional dependencies. Yet, even though BC-schema is now in BCNF, we still have the problem of repetition of information that we had earlier.

To deal with this problem, we must define a new form of constraint, called a *multivalued dependency*. As we did for functional dependencies, we shall use multivalued dependencies to define a normal form for relation schemas. This normal form, called fourth normal form (4NF), is more restrictive than BCNF. We shall see that every 4 NF schema is also in BCNF, but there are BCNF schemas that are not in 4 NF.

#### 9.3.1. Multivalued Dependencies:

Functional dependencies rule out certain tuples from being in a relation. If  $A \rightarrow B$ , then we cannot have two tuples with the same  $A$  value but different  $B$  values. Multivalued dependencies, on the other hand, don't rule out the existence of certain tuples. Instead, they require that other tuples of a certain form be present in the relation.

For this reason, functional dependencies sometimes are referred to as **equality-generating dependencies**, and multivalued dependencies are referred to as **tupel-generating dependencies**.

Let  $R$  be a relation schema and let

$\alpha \subseteq \beta$  and  $\beta \subseteq R$  The *multivalued dependency*

$\alpha \twoheadrightarrow \beta$

$\alpha$  holds on  $R$  if in any legal relation  $r(R)$ , for all pairs for tuples  $t_1$  and  $t_2$  in  $r$  such that  $t_1[\alpha] = t_2[\alpha]$  there exist tuples  $t_3$  and  $t_4$  in  $r$  such that

$$t_1(\alpha) = t_2(\alpha) = t_3[\alpha] t_4[\alpha]$$

$$t_3[\beta] = t_1[\beta]$$

$$t_3[R - \beta] = t_2[R - \beta]$$

$$t_4[\beta] = t_2[\beta]$$

$$t_4[R - \beta] = t_1[R - \beta]$$

	$\alpha$	$\beta$	$R - \alpha - \beta$
$t_1$	$a_1 \dots a_i$	$a_{i+1} \dots a_j$	$a_{j+1} \dots a_n$
$t_2$	$a_1 \dots a_i$	$b_{i+1} \dots b_j$	$b_{j+1} \dots b_n$
$t_3$	$a_1 \dots a_i$	$a_{i+1} \dots a_j$	$b_{j+1} \dots b_n$
$t_4$	$a_1 \dots a_i$	$b_{i+1} \dots b_j$	$a_{j+1} \dots a_n$

**Tabular representation of  $\alpha \twoheadrightarrow \beta$**

This definition is less complicated than it appears to be. Figure gives a tabular picture of  $t_1, t_2, t_3$  and  $t_4$ . Intuitively, the multivalued dependency

$\alpha \twoheadrightarrow \beta$  says that the relationship between  $\alpha$  and  $\beta$  is independent of the relationship between  $\alpha$  and  $R - \beta$ . If the multivalued dependency  $\alpha \twoheadrightarrow \beta$  is a *trivial* multivalued dependency on schema  $R$  then  $\alpha \twoheadrightarrow \beta$  is trivial if  $\beta \subseteq \alpha$  or  $\beta \cup \alpha = R$ .

To illustrate the difference between functional and multivalued dependencies, we consider the BC-schema again, and the relation  $bc$ . We must repeat the loan number once for each address, for each loan a customer has.

This repetition is unnecessary, since the relationship between a customer and his address is independent of the relationship between that customer and a loan. If a customer (say, smith) has a loan (say, loan number L-23). We want that loan to be associated with all Smiths addresses. Thus, the relation of figure is illegal. To make this relation legal. We need to add the tuples (L-23, smith, main, Manchester) and (L-27, smith, north, rye) to the  $bc$  relation of figure.

Comparing the preceding example with our definition of multivalued dependency, we see that we want the multivalued dependency

Customer-name  $\twoheadrightarrow$  customer -street customer -city

To hold (The multivalued dependency customer-name  $\twoheadrightarrow$  loan-number will do as well. We shall soon see that they are equivalent).

As with functional dependencies, we shall use multivalued dependencies in two ways:

1. To test relations to determine whether they are legal under a given set of functional and multivalued dependencies.
2. To specify constraints on the set on legal relations; we shall thus concern ourselves with only those relations that satisfy a given set of functional and multivalued dependencies.

<i>loan-number</i>	<i>customer-name</i>	<i>customer-street</i>	<i>customer-city</i>
L-23	Smith	North	Rye
L-23	Smith	Main	Manchester
L-93	Curry	Lake	Horseneck

Relation bc: An example of redundancy in a BCNF relation.

<i>loan-number</i>	<i>customer-name</i>	<i>customer-street</i>	<i>customer-city</i>
L-23	Smith	North	Rye
L-23	Smith	Main	Manchester
L-93	Curry	Lake	Horseneck

An illegal bc relation.

Note that, if a relation  $r$  fails to satisfy a given multivalued dependency, we can construct a relation  $r'$  that does satisfy the multivalued dependency by adding tuples to  $r$ .

Let  $D$  denote a set of functional and multivalued dependencies. The **closure**  $D^+$  of  $D$  is the set of all functional and multivalued dependencies logically implied by  $D$ . As we did for functional dependencies, we can compute  $D^+$  from  $D$ , using the formal definitions of functional dependencies and multivalued dependencies. We can manage with such reasoning for very simple multivalued dependencies. Luckily, multivalued dependencies that occur in practice appear to be quite simple. For complex dependencies, it is better to reason about sets of dependencies by using a system of inference rules.

For the definition of multivalued dependency, we can derive the following rule:

If  $\alpha \twoheadrightarrow \beta$  and  $\alpha \rightarrow \beta$

In other words, every functional dependency is also a multivalued dependency

### 9.3.2. Definition of Fourth Normal Form:

Consider again our BC-schema example in which the multivalued dependency  $\text{customer-name} \twoheadrightarrow \text{customer-street customer-city}$  hold, but no nontrivial functional dependencies hold, although BC-schema is in BCNF, the design is not ideal, since we must repeat a customer's address information for each loan. We shall see that we can use the given multivalued dependency to improve the database design by decomposing BC-schema into a **fourth normal form** decomposition.

A relation schema  $R$  is in **fourth normal form** (4NF) with respect to a set  $D$  of functional and multivalued dependencies if, for all multivalued dependencies in  $D^+$  of the form  $\alpha \twoheadrightarrow \beta$  where  $\alpha \subseteq R$  and  $\beta \subseteq R$

$\alpha \twoheadrightarrow \beta$  is a trivial dependency

$\alpha$  is a super key for schema  $R$

A database design is in 4NF if each member of the set of relation schemas that constitutes the design is in 4NF.

Note that the definition of 4NF differs from the definition of BCNF in only the use of multivalued dependencies instead of functional dependencies. Every 4NF schema is in BCNF. To see this fact, we note that, if a schema  $R$  is not in BCNF then, there is

```

result := {R};
done := false;
compute F+;
while (not done) do
  if (there is a schema  $R_i$  in result that is not in BCNF)
    then begin
      let  $\alpha \twoheadrightarrow \beta$  be a nontrivial functional
        dependency that holds on  $R_i$ 
        such that  $\alpha \rightarrow R_i$  is not in  $F^+$ ,
        and  $\alpha \cap \beta = \emptyset$ ;
       $result := (result - R_i) \cup (R_i - \beta) \cup (\alpha, \beta)$ ;
    end
  else done := true;

```

Note: Each  $R_i$  is in BCNF, and decomposition is loss less join.

A nontrivial functional dependency  $\alpha \rightarrow \beta$

holding on  $R$ , where  $\alpha$  is not a super key. Since  $\alpha \rightarrow \beta$  implies  $\alpha \twoheadrightarrow \beta$   $R$  cannot be in 4NF.

Let  $R$  be a relation schema, and let  $R_1, R_2, R_3, \dots, R_n$

be a decomposition of  $R$ . To check if each relation schema  $R_i$  in the decomposition is in 4NF, we need to find what multivalued dependencies hold on each  $R_i$ . Recall that, for a set  $F$  of functional dependencies, the restriction  $F_i$  of  $F$  to  $R_i$  is all functional dependencies in  $F^+$  that include *only* attributes of  $R_i$ . Now consider a set  $D$  of not functional and multivalued dependencies. The restriction of  $D$  to  $R_i$  is the set  $D_i$  consisting of

1. All functional dependencies in  $D^+$  that include only attributes of  $R_i$ .
2. All multivalued dependencies of the form  $\alpha \twoheadrightarrow \beta \cap R_i$

#### 9.4. More Normal Forms:

The fourth normal form is by no means the “ultimate” normal form. As we saw earlier, multivalued dependencies help us understand and tackle some forms of repetition of information that cannot be understood in terms of functional dependencies. There are types of constraints called **join dependencies** that generalize multivalued dependencies, and lead to another normal form called **project-normal form (PJNF)** (PJNF called **fifth normal form** in some books). There is a class of even more general constraints, which leads to a normal form called **domain-key normal form**.

A practical problem with the use of these generalized constraints is that they are not only hard to reason with, but there is also no set of sound and complete inference rules of reasoning about the constraints. Hence PJNF and domain-key normal form are used quite rarely. Appendix C provides more details about these normal forms.

<i>branch-name</i>	<i>loan-number</i>
Round Hill	L-58

<i>loan-number</i>	<i>amount</i>

<i>loan-number</i>	<i>customer-name</i>
L-58	Johnson

Conspicuous by its absence from our discussion of normal forms is **second normal form (2NF)**. We have not discussed it, because it is of historical interest only.

### 9.5. Summary:

The third normal form requires that all columns in a relational table are dependent only upon the primary key. A more formal definition is: A relational table is in third normal form (3NF) if it is already in 2NF and every non-key column is non transitively dependent upon its primary key. In other words, all nonkey attributes are functionally dependent only upon the primary key. The advantage of having relational tables in 3NF is that it eliminates redundant data, which in turn saves space and reduces manipulation anomalies.

Boyce-Codd normal form (BCNF) is a more rigorous version of the 3NF dealing with relational tables that had (a) multiple candidate keys, (b) composite candidate keys, and (c) candidate keys that overlapped.

Fourth normal form (4NF) is based on the concept of *multivalued dependencies* (MVD). A Multivalued dependency occurs when in a relational table containing at least three columns, one column has multiple rows whose values match a value of a single row of one of the other columns.

### 9.6. Technical Terms:

**Domain:** A domain is the set of allowable values for one or more attributes.

**Functional Dependency:** A many-to-one relationship shared by columns of values in database tables. A functional dependency from column X to column Y is a constraint that requires two rows to have the same value for the Y column if they have the same value for the X column.

**Multivalued Dependency:** Multi-valued dependency (MVD) is a generalization of functional dependency (FD), in the sense that every FD is a MVD. ( $A \twoheadrightarrow B$ )

**Join Dependency?**

Let R be a relation, and let A, B,.....Z be arbitrary subsets of the set of attributes of R. Then we say that R satisfies the JD if and only if R is equal to the join of its projections on A, B,.....Z.

**9.7. Model Questions:**

1. Compare third normal form and BCNF?
2. What is meant by Multivalued Dependency? Explain with an example?
3. Write about Fourth Normal form?

**9.8. References:****Database System Concepts**

Silberschatz, Korth, and Sudarshan

**Database Management Systems**

Arun K. Majumdar, Pritimoy Bhattacharyya

**An Introduction to Database Systems**

Bipin Desai

**Modern Database Management**

F. McFadden, J. Hoffer

**An Introduction to Database Systems**

C. J. Date;

**AUTHOR:**

**Y.SURESH BABU.**, M.Com, M.C.A.,  
Lecturer,  
Dept. Of Computer Science,  
JKC College.  
GUNTUR.



## Lesson 10

# Object Oriented Databases

### 10.0 Objectives:

The Lesson introduces a variety of modern techniques in database. Here, we introduce the concepts of object-oriented programming and then consider the use of these concepts in database systems.

The objective of this course is to give an advanced introduction to the concepts for modeling, designing, querying and managing large databases.

After reading this chapter, you should understand:

### Structure Of the Lesson:

#### 10.1. Introduction

#### 10.2. Need for Complex Data Types

#### 10.3. The Object-Oriented Data Model

##### 10.3.1. Object Structure

##### 10.3.2. Object Classes

##### 10.3.3. Inheritance

##### 10.3.4. Multiple Inheritance

##### 10.3.5. Object Identity

##### 10.3.6. Object Containment

#### 10.4. Object-Oriented Languages

#### 10.5. Persistent Programming Languages

##### 10.5.1. Persistent of Objects

##### 10.5.2. Object Identity and Pointers

#### 10.6. Summary

#### 10.7. Technical Terms

#### 10.8. Model Questions

#### 10.9. References.

### 10.1. Introduction:

As database systems were applied to a wider range of applications including computer-aided design. Limitations imposed by the relational model emerged as obstacles. As a result, database researchers invented new data models that overcame the relational model's restrictions.

The object-oriented approach to programming was first introduced by the language Simula 67,

which was designed for programming simulations. Smalltalk awesome of the early object-oriented programming languages for general applications. Today, the languages C++ and Java are the most widely used object-oriented programming languages.

## 10.2. Need for Complex Data Types:

Traditional database applications consist of data processing tasks, such as banking and payroll management. Such applications have conceptually simple data types. The basic data items are records that are fairly small and whose fields are atomic, i.e., they are not further structured, and the first normal form holds. Further there are only a few record types.

In recent years, demand has grown for ways to deal with more complex data types. Consider for example address, while an entire address could be viewed as an atomic data item of type string. This view would hide details such the street address, city, state and postal code, which could be of interest to queries. On the other hand if an address were represented by breaking it into the components. A better alternative is to allow structured data types, which allow a type, address with sub parts street address, city, state and postal code

## 10.3. The Object-Oriented Data Model:

In this section we present the main concepts of the object-oriented data model: the structure of objects and the notions of classes, inheritance, an object identity.

### 10.3. 1. Object Structure:

An object corresponds to an entity in the E-R model. The object-oriented paradigm is based on encapsulation of data and code related to an object into a single unit, whose contents are not visible to the outside world. Conceptually, all interactions between an object and the rest of the system are via messages. Thus the interface between an object and the rest of the system is defined by a set of allowed messages

In general, an object has associated with it

- A set of **variables** that contain the data for the object; variables correspond to attributes in the E-R model.
- A set of **messages** to which the object responds; each message may have zero, one or more parameters.
- A set of **methods** each of which is a body of code to implement a message; a method returns a value as the response to the message.

The term message in an object –oriented context does not imply the use of a physical message in a computer network. Rather, it refers to the passing of requests among objects without regard to specific implementation details. The term **invoke a method** is sometimes used to denote the act of sending a message to an object and the execution of the corresponding method.

We can illustrate the motivation for using this approach by considering employee entities in a bank database. Suppose the annual salary of an employee is calculated in different ways for different employees. For instance, managers may get a bonus depending on the bank's performance; while tellers may get a bonus depending on how many hours they have worked. We can encapsulate the code for computing the salary with each employee as a method that is executed in response to an annual-salary message.

All employee objects respond to the annual-salary message, but they do so in different ways. By encapsulating within the employee object itself the information about how to compute the annual salary, all employee objects present the same interface. Since the only external interface presented by an object is the set of messages to which that object responds. This ability to modify the definition of an object without affecting the rest of the system is one of the major advantages of the object-oriented-programming paradigm.

Methods of an object can be classified as either read only or update. A read-only method does not affect the values of the variables in an object, whereas an update method may change the values of the variables. The messages to which an object responds can be similarly classified as read-only or update, depending on the method that implements the message.

In the object-oriented model, we express derived attributes of an E-R model entity as read-only messages. For example, we can express the derived attribute employment length of an employee entity as an employment length message of an employee object. The method implementing the message may determine the employment length by subtracting the start-date of the employee from the current date.

In other words, every attribute of an entity must be expressed as a variable and a pair of messages of the corresponding object in the object-oriented model. We use the variable to store the value of the attribute, one message to read the value of the attribute, and other message to update the value.

### 10.3. 2. Object Classes:

Usually there are many similar objects in a database. By similar, we mean that they respond to the same messages, use the same methods, and have variables of the same name and type.

```
Class employee {
    /*Variables*/
    String name;
    String address;
    Date start-date;
    Int salary;
    /*Messages*/
    Int annual-salary ();
    String get-name ();
    String get-address ();
    Int set-address (string new-address);
    Int employment-length ();
};
```

**Definition of the class employee.**

We group similar objects to form a **class**. Each such object is called an instance of its class. All objects in a class share a common definition, although they differ in the values assigned to the variables.

The notion of a class in the object-oriented data model corresponds to the notion of an entity set in the E-R model. Examples of classes in our bank database are employees, customers, accounts, and loans.

The class employee in pseudo code. The definition shows the variables and the messages to which the objects of the class respond. In this definition, each object of the class employee contains the variables name and address, both of which are strings; start-date, which is a date; and salary, which is an integer. Each object responds to the five messages, namely annual-salary, get-name, get-address, set-address, and employment-length. The type name before each message name indicates the type of the response to the message. Observe that the message set-address takes a parameter new-address, which specifies the new value of the address. Although we have not shown them here, the class employee would also support messages that set the name, salary, and start date.

The methods for handling the messages are usually defined separately from the class definition. The methods get-address() and set-address() would be defined, for example, by the pseudo code:

```
String get-address (){
    return address;
}
int set-address (string new-address)
{
    Address=new-address;
}
```

While the method employment-length () would be defined as:

```
Int employment-length (){
    Return today ()-start-date
}
```

Here we assume that today() is a function that returns the current date, and the operation on dates returns the interval between the two dates.

The concept of classes is similar to the concept of abstract data types. However, there are several additional aspects to the class concept, beyond those of abstract data types. To represent these additional properties, we treat each class as itself being an object. A class object includes

- A set-valued variable whose value is the set of all objects that are instances of the class.
- Implementation of a method for the message new, which creates a new instance of the class.

### 10.3. 3. Inheritance:

An object oriented database schema usually requires a large number of classes. Often, however, several classes are similar.

For example, assume that we have an object-oriented database for our bank application. We would

expect the class of bank customers to be similar to the class of bank employees, in that both define variables for name, address, and so on. However, there are variables specific to customers (credit-rating, for example). It would be desirable to define one, only if we combine employees and customers into one class.

To allow the direct representation of similarities among classes, we need to place classes in specialization hierarchy for the entity relationship model. For instance, we say that the employee is a specialization of person, because the set of all employees is a subset of the set of all persons, i.e., every employee is a person. Similarly customer is a specialization of a person.

The concept of a **class hierarchy** is similar to the concept of specialization hierarchy in the entity-relationship model. Classes that are specializations of a person class can represent employees and customers. Variables and methods specific to employees are associated with employee class. Variables and methods specific customers are associated with the customer class. Variables and methods that apply both to employees and to customers are associated with the person class

An object representing an officer contains all the variables of class officer; in addition the object representing an officer also contains all the variables of classes employee and person. This is because every officer is defined to be an employee, and since in turn every employee is defined to be a person, we can infer that every officer is also a person. We refer to this property that objects of a class contain variables defined in its super classes, as the **inheritance** of variables.

Messages and methods are inherited in the same way that variables are inherited. An important benefit of inheritance in object-oriented systems is **substitutability**. Any method of a class say, **A** can equally well be invoked with any object belonging to any sub class **B** of **A**. This characteristic leads to code reuse. Since the messages, methods, and functions do not have to be written again for objects of class **B**. For instance, the message get name() is defined for the class person, and it can be invoked with a person object, or with any object belonging to any subclass of person, such as customer or officer.

Usually we make the latter choice in object-oriented systems. It is possible to determine the set of all employee objects, in this case by taking the union of those objects associated with all subclasses of employee. Most object-oriented systems allow the specialization to be partial, i.e., they allow objects that belong to a class such as employee, but that do not belong to any of that class's sub classes

#### 10.3.4. Multiple Inheritance:

In most cases, a tree-structured organization of classes is adequate to describe applications in tree-structured organizations, each class can have at most one super class. However, there are situations that cannot be represented by a **direct a cyclic graph (Dag)** in which a class can have more than one super class.

For example, suppose employees can be either temporary or permanent. We may then create subclasses temporary and permanent, of the class employee. The subclass temporary would have an attribute termination-date specifying when the employment term ends. The subclass permanent may have a method for computing contributions to a company pension plan, which is not applicable to temporary employees.

An example of multiple inheritance, consider a university database, where a person can be a student or teacher. The university database may have class's student and teacher, which are sub classes of person, to model this situation. Now, there is also a category of students who also work

as teaching assistants; to model this situation, a class teaching-assistant may be created as a subclass of student as well as of teacher.

When multiple inheritance is used, there is potential ambiguity if the same variable or method can be inherited from more than one super class. For instance, the student class may have a variable dept identifying a student's department, and the teacher class may correspondingly have a variable dept identifying a teacher's department.

The class teaching-assistant inherits both these definitions of the variable dept, thus the use of the variable dept in the context of teaching assistants becomes ambiguous.

The result of using dept varies, depending on the particular implementation of the object-oriented model.

For example, the implementation may handle dept in one of these ways:

- Include both variables, renaming them to student.dept teacher.dept.
- Choose one or other, according to the order in which the class's student and teacher were created.
- Force the user to make a choice explicitly in the definition of the class teaching assistant.
- Treat the situation as an error.

Many object-oriented programming languages insist that an object must have a **most-specific class**, i.e., if the object belongs to many classes, it must belong to one class that is a sub class of all the other classes to which the object belongs. For example, if an object belongs to the person and teacher classes, it must belong to some class that is a sub class of both these classes. We must then use multiple inheritance to create all required sub classes such as teaching –assistant, student-council-member, teaching-assistant-council-member, and so on, to model the possibility of an object simultaneously having multiple roles.

### 10.3. 5. Object Identity:

Objects in an object-oriented database usually correspond to an entity in the enterprise being modeled by the database. An entity retains its identity even if some of its properties change over time. Likewise, an object retains its identity even if some or all of the values of variables or definitions of methods change over time. This concept of identity does not apply to tuples of a relational database. In relational systems, the tuples of a relation are distinguished only by the values that they contain.

Object identity is a stronger notion of identity than that usually found in programming languages or in data models not based on object orientation. We will illustrate several forms of identity next.

- **Value:** A data value is used for identity. This form of identity is used in relational systems. For instance, the primary key value of a tuple identifies the tuple.
- **Name:** A user-supplied name is used for identity. This form of identity is used for files in file systems. The user gives each file a name that uniquely identifies it, regardless of its contents.
- **Built-in:** A notion of identity is built into the data model or programming language, and user-supplied identifier is required. This form of identity is used in object-oriented systems. The system automatically gives each object an identifier when that object is created.

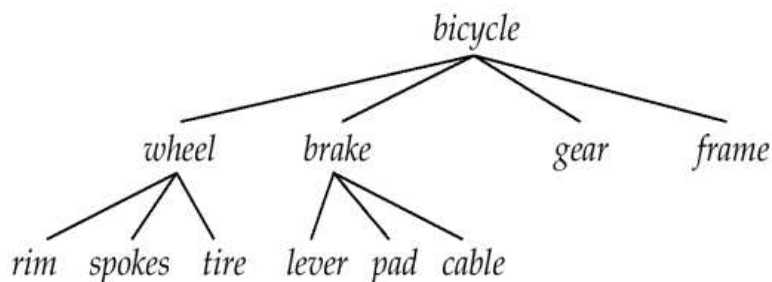
The identity of objects is a conceptual notion. Actual systems require a physical mechanism to identify objects uniquely. For identifying humans, names along with other information, such as date and place of birth, are often used as identifiers. Object-oriented systems use an **object identifier** to identify objects. Object identifiers are **unique**, i.e., each object has a single identifier, and no two objects have the same identifier. Object identifiers are not necessarily with which humans are comfortable. They could be long numbers, for example: The ability to store the identifier of an object as a field of another object is more important than having a name that is easy to remember.

### 10.3. 5. Object Containment:

References between objects can be used to model different real-world concepts. One such concept is that of *object containment*. For an illustration of containment, consider the simplified bicycle-design database. Each bicycle design contains wheels, a frame, brakes and gears. Wheels, in turn contain a rim, a set of spokes, and a tire. Each of the components of the design can be modeled as an object, and the containment of components can be modeled as containment of objects.

Objects that contain other objects are called **complex objects** or **composite objects**. There can be multiple levels of containment as shown in the figure. This situation creates a **containment hierarchy** among objects.

The hierarchy of figure shows the containment relationship among objects in a schematic way by listing class names, rather than individual objects. The links between classes must be interpreted as **is-part-of**, rather than the **is-a** interpretation of links in an inheritance hierarchy.



#### Containment Hierarchy for bicycle-design database

The above tree hierarchy does not show the variables of the various classes. Consider the bi-cycle. It may include variables, such as brand name, containing descriptive data about an instance of the class bicycle. In addition, class bi-cycle includes variables that contain a reference, or a set of references, to objects from the classes wheel, brake, gear, and frame

Containment is an important concept in object-oriented systems because it allows different users to view data at different granularities. A wheel designer can focus on instances of the class wheel without much concern about the objects of a class gear or brake. A marketing-staff person, attempting to price an entire bicycle, can reference all data pertaining to the bicycle by referencing the appropriate instance of class bicycle.

In certain applications, an object may be contained in several objects. In such cases, the containment relationship is represented by a DAG, rather than by a hierarchy.

## 10.4. Object-Oriented Languages:

Until now, we have covered the basic concepts of object orientation at an abstract level. To use the concepts in practice in a database system, however, we have to express them in some language. We can do this in either of two ways:

1. We use the concepts of object orientation purely as a design tool, and encode them into, for example, a relational database. We follow this approach when we use entity-relationship diagrams to model data, and we then manually convert the diagrams into a set of relations.
2. We incorporate the concepts of object orientation into a language that we use to manipulate the database. With this approach, there are several possible languages into which the concepts can be integrated:
  - One choice is to extend a data-manipulation language such as SQL by adding complex types and object orientation. Systems that provide object-oriented extensions to relational systems are called object-relational systems. We describe object-relational systems, and in particular object-oriented extensions to SQL.
  - Another choice is to take an existing object-oriented programming language and to extend it to deal with databases. Such languages are called **persistent programming languages**.

## 10.5. Persistent Programming Languages:

Database languages differ from traditional programming languages in that they directly manipulate data that are persistent, i.e., data that continue to exist even after the program that created it has terminated. A relation in a database and tuples in a relation are examples of persistent data. In contrast, the only persistent data that traditional programming languages directly manipulate are the files.

Access to a database is only one component of any real world application. While a data manipulation language like SQL is quite effective for accessing data, a programming language is required for implementing other components of the application such as user interfaces or communication with other computers. The traditional way of interfacing database languages to programming languages is by embedding SQL with in programming language.

A **persistent programming language** is a programming language extended with constructs to handle persistent data. Persistent programming languages can be distinguished from languages with embedded SQL in at least two ways:

1. With an embedded language, the type system of the host language usually differs from the type system of the data manipulation language. The programmer is responsible for any type conversions between the host language and SQL. Having the programmer carry out this task has several drawbacks:
  - The code to convert between objects and tuples operates outside the object-oriented type system, and hence has a higher chance of having undetected errors.
  - Conversion between the object-oriented format and the relational format of tuples in the database takes a substantial amount of code. The format translation code, along with the code for loading and unloading data from a database, can form a significant percentage of the total code required for an application.



2. The programmer using an embedded query language is responsible for writing explicit code to fetch data from the database into memory. If any updates are performed, the programmer must write code explicitly to store the updated data back in the database.

### 10.5.1. Persistent Of Objects:

Object-oriented programming languages already have a concept of objects, a type system to define object types, and constructs to create objects. However, these objects are transient. They vanish when the program terminates, just as variables in a Pascal or C program vanish when the program terminates. If we wish to use such a language into a database programming language, the first step is to provide a way to make objects persistent. Several approaches have been proposed.

- **Persistent by class:** The simplest, but least convenient, way is to declare that a class is persistent. All objects of the class are then persistent objects by default. Objects of non-persistent classes are all transient.
- **Persistent by creation:** In this approach, new syntax is introduced to create persistent objects, by extending the syntax for creating transient objects. Thus, an object is either persistent or transient, depending on how it was created. Several object-oriented database systems follow this approach.
- **Persistence by marking:** A variant of preceding approach is to mark objects as persistent after they are created. All objects are created as transient objects, but, if an object is to persist beyond the execution of the program, it must be marked explicitly as persistent before the program terminates. This approach, unlike the previous one, postpones the decision on persistence or transience until after the object is created.
- **Persistent by reach ability:** One or more objects are explicitly declared as persistent objects. All other objects are persistent if they are referred to directly, or indirectly, from a root persistent object.

Thus, all objects referred from the root persistent objects are persistent. But also, all objects referred from these objects are persistent, and objects to which they refer are in turn persistent, and so on.

A benefit of this scheme is that it is easy to make entire data structures persistent by merely declaring the root of such structures as persistent. However, the database system has the burden of following chains of references to detect, which objects are persistent, and that can be expensive.

### 10.5.2. Object Identity and Pointers:

In an object-oriented programming language that has not been extended to handle persistence, when an object is created, the system returns a transient object identifier. Transient object identifiers are valid only when the program that created them is executing. After that program terminates, the objects are deleted and the identifier is meaningless. When the persistent object is created, it is assigned a persistent object identifier.

The notion of object identity has an interesting relationship to pointers in programming languages. A simple way to achieve built-in identity is through pointers to physical locations in storage. In particular, many object-oriented languages such as c++ a transient object identifier is actually an in-memory pointer.

However, the association of an object with physical location in storage may change over time. There are several degrees of permanence of identity

- **Intraprocedure:** Identity persists only during the execution of a single procedure. Examples of intraprogram identity are local variables within procedures.
- **Intraprogram:** Identity persists only during the execution of a single program or query. Examples of intraprogram identity are global variables in programming languages. Main-memory or virtual memory pointers offer only intraprogram identity.
- **Interprogram:** Identity persists from one program execution to another. Pointers to file-system data on disk offer interprogram identity, but they may change if the way data is stored in the file system is changed.
- **Persistent:** Identity persists not only among program structural reorganizations of the data. It is the persistent form of identity that is required for object-oriented systems.

In persistent extensions of languages such as C++, object identifiers for persistent objects are implemented as "persistent pointers". A persistent pointer is a type of pointer that, unlike in-memory pointers, remains valid even after the end of a program execution, and across some forms of data reorganization. A programmer may use a persistent pointer in the same way that she may use an in-memory pointer in a programming language. Conceptually, we may think of a persistent pointer as a pointer to an object in the database.

## 10.6. Summary:

The object oriented data model is an adaptation to database systems of the object oriented programming paradigm. It is based on the concept of encapsulating in an object, the data and the code that operated on those data. There are two approaches to creating an object-oriented database. We can add the concept of object orientation to existing database languages, or we can extend existing object oriented languages to deal with the database by adding concepts such as persistence and collections. Extended relational databases take the former approach. Persistent programming languages follow the latter approach.

## 10.7. Technical Terms:

**Object:** An element that combines data and behavior in a single container of code.

**Encapsulation:** In object-oriented programming, the grouping of data and the code that manipulates it into a single object. If a change is made to an object class, all instances of that class (i.e., all objects) are changed. Encapsulation is one of the benefits of object-oriented programming.

**Method:** The software that implements the behavior specified by an operation.

**Inheritance:** An inheritance is a way to form new classes (instances of which will be objects) using pre-defined objects or classes where new ones simply take over old one's implementations and characteristics. It is intended to help reuse of existing code with little or no modification.

**Subclass:** A subclass is a class that inherits some properties from its superclass.

**SuperClass:** A superclass is a class from which other classes are derived. A superclass is also called a parent class or base class

**10.8. Model Questions:**

1. Explain the Object Oriented Data Model?
2. Write About Object Oriented Languages?
3. What are the features of Object Oriented Languages?

**10.9. References:****Database System Concepts**

Silberschatz, Korth, and Sudarshan

**Database Management Systems**

Arun K. Majumdar, Pritimoy Bhattacharyya

**An Introduction to Database Systems**

Bipin Desai

**Modern Database Management**

F. McFadden, J. Hoffer

**An Introduction to Database Systems**

C. J. Date;

**AUTHOR:**

**Y.SURESH BABU.**, M.Com., M.C.A.,  
Lecturer,  
Dept. Of Computer Science,  
JKC College.  
GUNTUR.

**11.10. Model Questions:**

1. What does nested relation mean? Explain with an example?
2. Explain the concept of Complex Types?
3. Write about Inheritance and various types?
4. What are Reference types?
5. Write Short notes on functions and Procedures?

**11.11. References:****Database System Concepts**

Silberschatz, Korth, and Sudarshan

**Database Management Systems**

Arun K. Majumdar, Pritimoy Bhattacharyya

**An Introduction to Database Systems**

Bipin Desai

**Modern Database Management**

F. McFadden, J. Hoffer

**An Introduction to Database Systems**

C. J. Date;

**AUTHOR:**

**Y.SURESH BABU.**, M.Com., M.C.A.,  
Lecturer,  
Dept. Of Computer Science,  
JKC College.  
GUNTUR.

### 11.7.2. External Language Routines:

SQL1999 allows us to define functions in a programming language such as c or c++. Functions defined in this fashion can be more efficient than defined in SQL, and computations that cannot be carried out in SQL can be executed in these functions. An example of the use of such functions would be to perform a complex arithmetic computation on the data in a tuple.

External procedures and functions can be specified in this way:

```
Create procedure author-count-proc( in title varchar(20),out count integer)
```

```
Language c
```

```
External name ' / usr/avi/bin/author-count-proc
```

```
    Create function author-count(title varchar(20))
```

```
        Returns integer
```

```
        Language c
```

```
        External name '/usr/avi/bin/author-count
```

The external language procedures need to deal with null values and exceptions. They must therefore have several extra parameters: an SQL state value to indicate failure/success status, a parameter to store the return value of the function, and indicator variables for each parameter/function result to indicate if the value is null. An extra line parameter style generally added to the declaration above indicates that the external procedures/functions take only the arguments shown and do not deal with null values or exceptions.

### 11.8. Summary:

The object-relational data model extends the relational data model by providing a richer type system including collection types, and object orientation. Object orientation provides inheritance with subtypes and sub tables, as well as object references. Collection types include nested relations, sets, multi sets and arrays. The object relational model permits attributes of a table to be collections.

Object-relational database systems provide a convenient migration path for users of relational databases who wish to use object oriented- features.

### 11.9. Technical Terms:

**Object:** Instance of a Class.

**Arrays:** Collection of elements of the same data type.

**Function:** A function is a sequence of code, which performs a specific task, as part of a larger program.

**Relational database:** A database system in which the database is organized and accessed according to the relationships between data items without the need for any consideration of physical orientation and relationship. Relationships between data items are expressed by means of tables.

**Exceptions:** Exception is an unusual situation in a program.

### 11.7.1. SQL Functions And Procedures:

Suppose that we want a function that, given the title of a book, returns the count of the number of authors, using the 4NF schema. We can define the function this way:

```
Create function author-count(title varchar(20))
    Returns integer
Begin
    Declare a-count integer
        Select count (author) into a-count
        From authors
        Where authors.title=title
    Return a-count
End
```

This function can be used in a query that returns the titles of all books that have more than one author:

```
Select title
From books4
Where author-count(title)>1
```

Functions are particularly useful with specialized data types such as images and geometric objects. For instance a polygon data type used in a map database may have an associated function that checks if two polygons overlap, and an image data type may have associated functions to compare two images for similarity. Functions may be written in an external language such as C. Some database systems also supports functions that return relations, that is multi sets of tuples, although such functions are not supported by SQL 1999

The functions, which we saw in STRUCTURED TYPES can be viewed as functions associated with structured types. They have an implicit first parameter called **self**, which is a set to the structured type value on which the method is invoked. Thus, the body of the method can refer to an attribute **a** of the value by using **self.a**. These attributes can also be updated by the method.

SQL1999 also supports procedures. The author-count function could instead be written as a procedure:

```
Create procedure author-count- proc(in title varchar(20), out a-count integer)
    Begin
        Select count(author) into a-count
        From authors
        Where authors.title=title
    End
```

Procedures can be invoked either from an SQL procedure or from embedded SQL by the call statement:

```
    Declare a-count integer;
    Call author -count-proc('data base systems concepts', a-count);
```

SQL 1999 permits more than one procedure of the same name, so long as the number of arguments, of the procedures with same name is different. The name along with the number of arguments is used to identify the procedure. SQL 1999 also permits more than one function with the same name, so long as the different functions with the same name either have different numbers of arguments, or for functions with the same number of arguments, differ in the type of at least one argument.

<i>title</i>	<i>author</i>	<i>publisher</i>	<i>keyword-set</i>
		<i>(pub-name, pub-branch)</i>	
Compilers	Smith	(McGraw-Hill, New York)	{parsing, analysis}
Compilers	Jones	(McGraw-Hill, New York)	{parsing, analysis}
Networks	Jones	(Oxford, London)	{Internet, Web}
Networks	Frick	(Oxford, London)	{Internet, Web}

### A partially nested version of the flat books relation.

```
Select title,set(author) as author-set,publisher(pub-name,pub-branch) as publisher,
      Set(keyword) as keyword-set
      From flat-books
```

```
      Groupby title,publisher
```

Another approach to creating nested relations is to use subqueries in the select clause. The following query which performs the same task as the previous query:

```
Select title,
      (select author
       from flat-books as M
       where M.title=O.title) as author-set,
      publisher(pub-name, pub-branch) as publisher,

      (select keyword
       from flat-books as N
       where N.title=O.title) as keyword-set,
      from flat-books as O
```

The system executes the nested sub queries in the select clause for each tuple generated by the **from** and **where** clause of the outer query. Observe that the attribute O.title from the outer query is used in the nested queries, to ensure that the only correct sets of authors and keywords are generated for each title. An advantage of this approach is that an **orderby** clause can be used in the nested query, to generate results in a desired order. A array or a list could be constructed from the result of the nested query. Without such an ordering, arrays and lists would not be uniquely determined.

We note that while un-nesting of array-valued attributes can be carried out in SQL 1999. The reverse processing of nesting is not supported by SQL 1999

## 11.7. Functions And Procedures:

SQL 1999 allows the definition of functions, procedures, and methods. These can be defined either by the procedural component of SQL 1999 or by external programming languages such as java, c and c++. We look at definitions of SQL 1999 first, and then see how to use definitions in external languages. Several database systems support their own procedural languages, such as pl/sql in oracle and transact SQL in Microsoft SQL Server. These resemble the procedural part of SQL 1999, but there are differences in syntax and semantics.

```

Select title
From books
Where 'database' in(unnest(key word-set))

```

Note that we have used `unnest(keyword-set)` in a position where SQL without nested relations would have required a `select-from-where` sub-expression.

If we know that a particular book has three authors, we could write:

```

Select author-array[1],author-array[2],author[3]
From books
Where title='Database System Concepts'

```

Now, suppose that we want a relation containing pairs of the form "title, author-name" for each book. We can use this query:

```

Select B.title,A.name
From books as B,unnest(B.author-array) as A

```

Since the `author-array` attribute of `books` is a collection valued field, it can be used in a `from` clause, where a relation is expected

### 11.6.3. Nesting And Un-Nesting:

The transformation of a nested relation into a form with fewer (or no) relation-valued attributes is called un-nesting. The `books` relation has two attributes, `title` and `publisher`, that are not. Suppose, if we want to convert the relation into a single flat relation, with no nested relations or structured types as attributes, we can use the following query to carry out the task:

```

Select title, A as author,publisher.name a name,publisher.branch
As pub-branch ,k as keyword

```

```

From books as B, unnest (B.author-array) as A, unnest(b.key word-set) as k

```

The variable `B` in the **from** clause is declared to range over books. The variable `A` is declared to range over the authors in author array from the book `B`, and `k` is declared to range over the keywords in the keyword-set of the book `B`. Fig 9.1 shows an instance `books` relation, and figure 9.2 shows the 1NF relation that is the result of the preceding query.

The reverse process of transforming 1NF relation into a nested relation is called nesting. Nesting can be carried out by an extension of grouping in SQL, a temporary multiset relation is (logically) created for each group, and an aggregate function is applied on the temporary relation. By returning the multiset instead of applying the aggregate function, we can create a nested relation. Suppose that we are given a 1NF relation `flat-books`, as in figure 9.2, the following query nests the relation on the attribute `keyword`:

```

Select title, author, publisher(pub-name, pub-branch) as publisher,
Set(keyword) as keyword-set
From flat-books
Groupby title,author,publisher

```

If we want to nest the `author` attribute as well, and thereby to convert the 1NF table.



```
Create type person
    (name varchar(20) primary key,
     address varchar(20))
    ref from(name)
create table people of person
    ref is oid derived
```

Note that the table definition must specify that the reference is derived, and must still specify a self-referential attribute name. When inserting a tuple for departments, we can then use:

```
Insert into departments
    Values('cs', 'jhon')
```

## 11.6. Querying With Complex Types:

In this section we represent extensions of the SQL query language to deal with complex types. Let us start with a simple example: Find the title and the name of the publisher of each book. This query carries out the task:

```
Select title,publisher.name
From books
```

Notice that, the field name of the composite attribute publisher is referred to by a dot notation.

### 11.6.1. Path Expression:

References are de referenced in SQL 1999 the symbol  $\rightarrow$ . Consider the departments table defined earlier. We can use this query to find the names and addresses of the heads of all departments:

```
Select head  $\rightarrow$  name,head  $\rightarrow$  address
From departments
```

An expression such as “head  $\rightarrow$  name” is called a path expression.

Since, head is a referenced to a tuple in the people table, the attribute name in the preceding query is the name attribute of the tuple from the people table. References can be used to hide join operations. In the preceding example, without the references, the head field of department would be declared a foreign key of the table people. To find the name and address of the head of a department, we would require an explicit join of the relations departments and people. The use of references simplifies the query considerably.

### 11.6.2. Collection – Values Attributes:

Arrays are the only collection type supported by SQL 1999, but we use the same syntax for the relation-valued attributes also. An expression evaluating to a collection can appear anywhere that a relation name may appear such as in a **from** clause as the following paragraphs illustrate. We use the table, books which we defined earlier.

If we want to find all books that have the word “database” as one of their key words, we can use this query:

```

Insert into departments
  Values('cs',null)
Update departments
  Set head=(select ref(p)
            From people as p
            Where name='john')
  Where name='cs'

```

This syntax for accessing the identifier of a tuple is based on the oracle syntax. SQL 1999 adopts a different approach, one where the referenced table must have an attribute that stores the identifier of a tuple. We declare this attribute called the self-referential attribute, by adding a **ref is** clause to the create table statement:

```

Create table people of person
  Ref is oid system generated

```

Here, oid is an attribute name, not a keyword. The subquery above would then use

```
Select p.oid
```

Instead of select ref(p).

An alternative to system-generated identifier is to allow users to generate identifiers. The type of the self-referential attribute must specify that the reference is user generated:

```

Create type person
  (name varchar(20),
  address varchar(20))
  ref using varchar(20)
create table people of person
  ref is oid user generated

```

When inserting a tuple in people, we must provide a value for the identifier:

```

Insert into people values
  ('01284567','jhon','23 Coyote Run')

```

No other tuple for people or its super tables can have the same identifier. We can then use the identifier value when inserting a tuple into departments, without the need for a separate query to retrieve the identifier:

```

Insert into departments
  Values('cs','01284567')

```

It is even possible to use an existing primary key value as the identifier, by including the **ref from** clause in the type definition:

For each entity to have exactly one most-specific type, we would have to create a sub type for every possible combination of the super types. In the preceding example, we would have sub types such as Foreign Under graduate student, Foreign graduate Student foot ballplayer, and so on. Unfortunately, we would end up with an enormous number of sub types of person.

A better approach in the context of database systems is to allow an object to have multiple types, without having a most-specific type. Object-relational systems can model such a feature by using inheritance at the level of tables, rather than of types, and allowing an entity to exist in more than one table at once.

For example, suppose we again have the type person, with sub types student and teacher, and the corresponding table people, with sub tables teachers and students. We can then have a tuple in teachers and a tuple in students corresponding to the same tuple in people.

There is no need to have a type teaching assistant unless we wish to store extra attributes or redefine methods in a manner specific to people who are both students and teachers.

We note that however SQL 1999 prohibits such a situation, because of consistency requirement. Since SQL 1999 also does not support multiple inheritance, we cannot use inheritance to model a situation where a person can be both a student and a teacher. We can of course create separate tables to represent the information without using inheritance. We would have to add appropriate referential integrity constraints to ensure that students and teachers are also represented in the people table.

### 11.5. Reference Types:

Object oriented languages provide the ability to refer to objects. An attribute of a type can be a reference to an object of a specified type. For example, in SQL 1999 we can define a type department with a field name and a field head, which is a reference to the type person, and a table departments of type department as follows:

```
Create type Department(  
    name varchar(20),  
    head ref(person) scope people  
)  
  
create table departments of department
```

Here the reference is restricted to tuples of the table people. The restriction of the scope of a reference to tuples of a table is mandatory in SQL 1999, and it makes references behave like foreign keys.

We can omit the declaration scope people from the type declaration and instead we make an addition to the create table statement:

```
Create table departments of Department  
    (head with options scope people)
```

In order to initialize a reference attribute, we need to get the identifier of the tuple that is to be referenced. We can get the identifier value of a tuple by means of a query. Thus, to create a tuple with the reference value, we may first create the tuple with a null reference and then set the reference separately:

There are some consistency requirements for sub tables. Before we state the constraints, we need a definition: We say that tuples in a sub table **corresponds** to tuples in a parent table if they have the same values for all inherited attributes. Thus, corresponding tuples represent the same entity.

The consistency requirements for sub tables are:

1. Each tuple of the super table can correspond to at most one tuple in each of its immediate sub tables.
2. SQL 1999 has an additional constraint that all the tuples corresponding to each other must be derived from one tuple.

For example, without the first condition, we could have two tuples in students (or teachers) that correspond to the same person.

The second condition rules out a tuple in people corresponding to both a tuple in students and a tuple in teachers unless all these tuple was inserted in a table, teaching assistants, which is a sub table of both teachers and students.

Since SQL 1999 does not support multiple inheritance, the second condition actually prevents a person from being both a teacher and a student. The same problem would arise if the sub table teaching-assistants is absent, even if multiple inheritance were supported. Obviously it would be useful to model a situation where a person can be a teacher and a student, even if, a common sub table teaching-assistants is not present. Thus, it can be useful to remove the second consistency constraint.

Sub tables can be stored in an efficient manner without replication of all inherited fields, in one of two ways:

- Each table stores the primary key (which may be inherited from a parent table) and the attributes defined locally. Inherited attributes (other than the primary key) do not need to be stored, and can be derived by means of a join with the super table, based on the primary key.
- Each table stores all inherited and locally defined attributes. When a tuple is inserted, it is stored only in the table in which it is inserted, and its presence is inferred in each of the super tables. Access to all attributes of a tuple is faster, since a join is not required. However, in case the second consistency constraint is absent, i.e., an entity can be represented in two sub tables with out being present in a common subtable of both this representation can result in replication of information.

#### 11.4.3. Overlapping Sub tables:

Inheritance of types should be used with care. A university database may have many sub types of person, such as student, teacher, football player, foreign citizen, and so on.

Student may it self have sub types such as undergraduate student, graduate student, and part time student. Clearly, a person can belong to several of these categories at once. Each of these categories is sometimes called a role.

**Create type** teaching assistant  
**Under** student, teacher

Teaching assistant would inherit all the attributes of student and teacher. There is a problem, however, since the attributes name, address, and department are present in student, as well as in teacher.

The attributes name and address are actually inherited from 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 another department. To avoid a conflict between the two occurrences of department, we can rename them by using an **as** clause, as in this definition of the type Teaching Assistant:

**Create type** Teaching Assistant  
**Under** student **with**(department **as** student-dept),  
 Teacher **with**(department **as** teacher-dept)

#### 11.4.2. Table Inheritance:

Sub tables in SQL 1999 correspond to the E-R notion of specialization/generalization. For instance, suppose we define the people table as follows:

**Create table** people **of** person

We can then define tables students and teachers as **subtables** of people, as follows:

**Create table** students **of** student

**Under** people

**Create table** teachers **of** teacher

**Under** people

The types of the sub tables must be subtypes of the type of the table. Thereby, every attribute present in people is also present in the sub tables.

Further, when we declare students and teachers as sub tables of people, every tuple present in students or teachers also implicitly present in people. Thus, if a query uses the table people, it will find not only tuples directly inserted into that table, but also tuples inserted into its sub tables, namely students and teachers. However, only those attributes that are present in people can be accessed.

Multiple inheritance is possible with tables, just as it is possible with types. For example, we can create a table of type Teaching Assistant:

**Create table** teaching-assistants

**Of** Teaching Assistant

**Under** students, teachers

As a result of the declaration, every tuple present in the teaching assistants table is also implicitly present in the teachers and in the students table, and in turn in the people table.

SQL 1999 permits us to find tuples that are in people but not in its sub tables by using "only people" in place of people in a query.

**Insert into** books

**Values**

('compilers',**array**['Smith','jones'],publisher('McGraw-Hill','NewYork'),**set**('parsing', 'analysis'))

## 11.4. Inheritance:

Inheritance can be at the level of types, or at the level of tables. We first consider inheritance of types, then inheritance at the level of tables.

### 11.4.1. Type Inheritance:

Suppose that we have the following type definition for people:

```
Create type person
    (name varchar(20),
    address varchar(20))
```

We may want to store extra information in the database about people:

```
create type person
    (name varchar(20),
    address varchar(20))
```

We may want to store extra information in the database about people who are students, and 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
    (degree varchar(20),
    department varchar(20))

create type teacher
    under person
    (salary integer,
    department varchar(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 super type of student as well as teacher

Methods of structured type are inherited by its subtypes, just as attributes are. However a sub type can redefine the effect of a method by declaring the method again, using **overriding method** in place of **method** in method declaration.

Now, suppose if we want to store information about teaching assistants who are simultaneously students and teachers, perhaps even in different departments. We can do this by using **multiple inheritances**. SQL 1999 standard does not support multiple inheritances.

For instance, if our type system supports multiple inheritances, we can define a type for teaching assistants as follows:

With the above declaration, there is no explicit type for rows of the table.

A structured type can have **methods** defined on it. We declare methods as part of the type definition of a structured type:

```
Create type Employee as (  
    Name varchar(20),  
    Salary integer)  
Method giverraise (percent integer)
```

We create method body separately:

```
Create method giverraise(percent integer) for Employee  
Begin  
Setself.salary=self.salary+(self.salary*percent)/100;  
End
```

The variable **self** refers to the structured type instance on which the method is invoked. The body of the method can contain procedural statements.

### 11.3.3. Creation Of Values Of Complex Types:

In SQL 1999 **constructor functions** are used to create values of structured types. A function with same name as structured type is a constructor function for the structured 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
```

In SQL 1999, unlike in object-oriented databases, a constructor creates a value of the type, not an object of the type, i.e., the value the constructor creates has no object identity. In SQL 1999 objects corresponds to tuples of a relation, and are created by inserting a tuple in a relation.

By default every structured type has a constructor with no arguments, which sets the 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; although they have the same name, they must be distinguishable by the number of arguments and types of their arguments.

We create set-valued attributes, such as keyword-set, by enumerating their elements within parentheses following the keyword **set**. We can create multiset values just like set values, by replacing **set** by **multiset**.

Here we have created a value for the attribute publisher by invoking a constructor function for publisher with appropriate arguments.

If we want to insert the preceding tuple into the relation books, we could execute the statement:

Sets are an instance of **collection types**. Other instances of collection types include **arrays** and **multisets** (i.e., unordered collections, where an element may occur multiple times). The following attribute definitions illustrate the declaration of an array:

Author-array **varchar(20) array[10]**

Here, author-array is an array of 10 author names to the maximum. We can access elements of an array by specifying the array index, for example author-array[1].

Arrays are the only collection type supported by SQL: 1999 the syntax used is as in the preceding declaration. SQL 1999 does not support unordered sets or multisets, although they may appear in future versions of SQL.

### 11.3.2. Structured Types:

Structured types can be declared and used in SQL 1999 as in the following example:

```
Create type publisher as
    (name varchar(20),
    branch varchar(20))
create type book as
    (title varchar(20),
    author-array varchar(20)array[10],
    pub-date date,
    publisher publisher,
    keyword-set setoff(varchar(20))
create table books of book
```

The first statement defines a type called publisher, which has two components: a name and a branch. The second statement defines a structured type Book, which contains a title, an author-array, which is an array of authors, a publication date, a publisher(of type publisher), and a set of keywords. (The declaration of keyword-set as a set uses our extended syntax, and is not supported by the SQL 1999 standard). The types illustrated above are called **structured types** in SQL 1999. Structured types allow composite attributes of E-R diagrams to be represented directly. Unnamed **row types** can also be used in SQL 1999 to define composite attributes. For instance, we could have defined an attribute publisher1 as:

```
Publisher1 row(name varchar(20),
                Branch varchar(20))
```

instead of creating a named type publisher.

We can of course create tables without creating an intermediate type for the table. For example, the table books could also be defined as follows:

```
Create table books
    (title varchar(20),
    author-array varchar(20)array[10],
    pub-date date,
    publisher publisher,
    keyword-set setoff(varchar(20)))
```



Although our example book database can be adequately expressed without using nested relations, the use of nested relations leads to an easier-to-understand model. The typical user of an information-retrieval system thinks of the database in terms of books having sets of authors, as the non-1NF design models. The 4NF design would require users to include joins in their queries, thereby complicating interaction with the system.

We could define a non-nested relational view (whose contents are identical to flat-books) that eliminates the need for users to write joins in their query. In such a view, however, we lose the one-to-one correspondence between tuples and books.

### 11.3. Complex Types:

Nested relations are just one example of extensions to the basic relational model. Other non-atomic data types, such as nested records, have also proved useful. The object-oriented data model has caused a need for features such as inheritance and references to objects. With complex types systems and object orientation, we can represent E-R model concepts, such as identity of entities, multi-valued attributes, and generalization and specialization directly, without a complex translation to the relational model.

<i>title</i>	<i>author</i>
Compilers	Smith
Compilers	Jones
Networks	Jones
Networks	Frick

*authors*

<i>title</i>	<i>keyword</i>
Compilers	parsing
Compilers	analysis
Networks	Internet
Networks	Web

*keywords*

<i>title</i>	<i>pub-name</i>	<i>pub-branch</i>
Compilers	McGraw-Hill	New York
Networks	Oxford	London

*books4*

#### 4NF version of the relation flat-books

In this section, we describe extensions to SQL to allow complex types including, nested relations and object-oriented features. Our presentation is based on the SQL: 1999 standard, but we also outline features that are not currently in the standard but may be introduced in future versions of SQL standards.

#### 11.3.1. Collection And Large Object Types:

Consider this fragment of code.

```

Create table books (
    .....
    keyword-set setoff(varchar(2));

```

This table definition differs from table definitions in ordinary relational databases, since it allows attributes that are sets, thereby permitting multi-valued attributes of E-R diagrams to be represented directly.

We can see that, if we define a relation for the preceding information, several domains will be non-atomic.

- **Authors:** A book may have a set of authors. Nevertheless, we may want to find all books of which Jones was one of the authors. Thus we are interested in a sub-part of the domain element "set of authors".
- **Keywords:** If we store a set of keywords for a book, we expect to be able to retrieve all books whose keywords include one or more keywords as non-atomic.
- **Publisher:** Unlike keywords and authors, publisher does not have a set-valued domain. However, we may view publisher as consisting of the sub fields name and branch. This view makes the domain of publisher non-atomic.

Figure shows an example relation, books. The books relation can be represented in 1NF, as in figure 9.2 since we must have atomic domains in 1NF, yet want access to individual authors and to individual keywords, we need one tuple for each (keyword, author) pair. The publisher attribute is replaced in the 1NF version by two attributes: one for each sub field of publisher.

<i>title</i>	<i>author</i>	<i>pub-name</i>	<i>pub-branch</i>	<i>keyword</i>
Compilers	Smith	McGraw-Hill	New York	parsing
Compilers	Jones	McGraw-Hill	New York	parsing
Compilers	Smith	McGraw-Hill	New York	analysis
Compilers	Jones	McGraw-Hill	New York	analysis
Networks	Jones	Oxford	London	Internet
Networks	Frick	Oxford	London	Internet
Networks	Jones	Oxford	London	Web
Networks	Frick	Oxford	London	Web

#### **Flat-books, a 1NF version of non-1NF relational**

Much of the awkwardness of the flat-books relation in the above figure disappears if we assume that the following multi-valued dependencies hold:

- Title  $\twoheadrightarrow$  author
- Title  $\twoheadrightarrow$  keyword
- Title  $\twoheadrightarrow$  pub-name, pub-branch

Then, we can decompose the relation into 4NF using the schemas:

- Authors (title, author)
- Keywords (title, keyword)
- Books4 (title, pub-name, pub-branch)

### 11.1. Introduction:

Programming languages add persistence and other database features to existing programming languages by using an existing object-oriented type system. In contrast, *object-relational* data model, by providing a richer type system including complex data types and object orientation. Relational query languages, in particular SQL need to be correspondingly extended to deal with the richer type system. Such extensions attempt to preserve the relational foundations in particular, the declarative access to data while extending the modeling power. Object-relational database systems (i.e., database systems based on the object-relational model) provide a convenient migration path for users of relational databases who wish to use object-oriented features.

We first present the motivation for the nested relational model, which allows relations that are not in first normal form, and allows direct representation of hierarchical structures. We then show how to extend SQL by adding a variety of object-relational features.

### 11.2. Nested Relations:

We know that first normal form contains all attributes with atomic domains, i.e., the elements of the domain are indivisible units.

The assumption of 1NF is a natural one in the bank example we have considered. However, not all applications are best modeled by 1NF relations. For example, rather than view a database as a set of records, users of certain applications view it as a set of objects. These objects may require several records for their representation. We shall see that a simple, easy-to-use interface requires a one-to-one correspondence between the user's intuitive notion of an object and the database system's notation of data item.

<i>title</i>	<i>author-set</i>	<i>publisher</i>	<i>keyword-set</i>
		( <i>name, branch</i> )	
Compilers	{Smith, Jones}	(McGraw-Hill, New York)	{parsing, analysis}
Networks	{Jones, Frick}	(Oxford, London)	{Internet, Web}

#### Non-1NF books relation, books

The **nested relational model** is an extension of the relational model in which domains may be either atomic or relation valued. Thus the value of a tuple on an attribute may be a relation, and relations may be contained within relations. A complex object thus can be represented by a single tuple of a nested relation. If we view a tuple of a nested relation as a data item, we have a one-to-one correspondence between data items and objects in the user's view of the database.

We illustrate nested relations by an example from library. Suppose we store for each book the following information:

- Book title
- Set of authors
- Publisher
- Set of keywords

## Lesson 11

# Object Oriented Databases

### 11.0 Objectives:

The Lesson introduces a variety of modern techniques in database. The major topics include, object-oriented database systems.

After reading this chapter, you should understand:

- To define the concept of Nested Relations
- Various Complex Types
- Explain the concept of Inheritance
- Various Reference Types
- What is Querying with complex types?
- What are Functions and Procedures?

### Structure Of the Lesson:

- 11.1. Introduction
- 11.2. Nested Relations
- 11.3. Complex Types
  - 11.3.1. Collection And Larger Object types
  - 11.3.2. Structured Types
  - 11.3.3. Creation of Values of complex types
- 11.4. Inheritance
  - 11.4.1. Type Inheritance
  - 11.4.2. Table Inheritance
  - 11.4.3. Overlapping Sub tables
- 11.5. Reference Types
- 11.6. Querying with complex types
  - 11.6.1. Path Expression
  - 11.6.2. Collection – values attributes
  - 11.6.3. Nesting and Unnesting
- 11.7. Functions and Procedures
  - 11.7.1. SQL functions and Procedures
  - 11.7.2. External Language Routines
- 11.8. Summary
- 11.9. Technical Terms
- 11.10. Model Questions
- 11.11. References.

## Lesson 12

# XML

### 12.0 Objectives:

To:

- Learn about XML
- Understand the structure of XML documents
- Explore the XML Schema
- Discover how to write queries on XML documents
- Learn about the XML APIs and XML applications.

### Structure Of the Lesson:

- 12.1. Introduction
- 12.2. XML Document Schema
- 12.3. Querying and Transformation
- 12.4. Application Program Interfaces to XML
- 12.5. Storage of XML data
- 12.6. XML Applications
- 12.7. Summary
- 12.8. Technical terms
- 12.9. Model questions
- 12.10. References

### 12. 1. Introduction

Markup means a part of the document that is not part of the printed output. A Markup language is a formal description of what part of the document is content, what part is markup, and what the markup means. Markup languages use tags to specify markup. Tags are enclosed in angular brackets, <>, and used in pairs. The beginning portion of a tag is written as <tag> and the ending portion as </tag>. These two delimit the tag. For example: <title> XML NOTES </title>. XML does not prescribe the set of tags allowed. Tags in an XML document, may be chosen as needed by the application. XML provides a standard way of tagging the data. Data can be efficiently exchanged between two organizations using XML provided they agree upon what tags appear in an XML document and what they mean. XML is also useful for storing structured information in files. The XML representation of the data has the following advantages:

1. The XML tags make the message self-documenting.
2. The format of the document is not rigid. The ability to recognize and ignore unexpected tags allows the format of the data to evolve over time without invalidating existing applications.
3. XML allows nested structures.
4. XML is widely accepted and tools to process XML documents are evolving faster.

**Structure of XML data:**

The fundamental construct in an XML document is the element. An element is a pair of matching start and end tags, and the text that appears between them. The XML documents must have a single root element that encompasses all other elements in the document. XML elements must nest properly. An example of XML elements is shown below.

```
<account> ....<balance> ....</balance> .... </account>
<bank>
  <account>
    <account_number>A-101</account_number>
    <branch_name>Downtown</branch_name>
    <balance>500</balance>
  </account>
  <account>
    <account_number>A-102</account_number>
    <branch_name>Perryridge</branch_name>
    <balance>400</balance>
  </account>
  <account>
    <account_number>A-201</account_number>
    <branch_name>Brighton</branch_name>
    <balance>900</balance>
  </account>
  <customer>
    <customer_name>Johnson</customer_name>
    <customer_street>Alma</customer_street>
    <customer_city>Palo Alto</customer_city>
  </customer>
  <customer>
    <customer_name>Hayes</customer_name>
    <customer_street>Main</customer_street>
    <customer_city>Harrison</customer_city>
  </customer>
  <depositor>
    <account_number>A-101</account_number>
    <customer_name>Johnson</customer_name>
  </depositor>
  <depositor>
    <account_number>A-201</account_number>
```

```
        <customer_name>Johnson</customer_name>
    </depositor>
    <depositor>
        <account_number>A-102</account_number>
        <customer_name>Hayes</customer_name>
    </depositor>
</bank>
```

**Fig. 12. 1**

```
<bank-1>
    <customer>
        <customer_name>Johnson</customer_name>
        <customer_street>Alma</customer_street>
        <customer_city>Palo Alto</customer_city>
        <account>
            <account_number>A-101</account_number>
            <branch_name>Downtown</branch_name>
            <balance>500</balance>
        </account>
        <account>
            <account_number>A-201</account_number>
            <branch_name>Brighton</branch_name>
            <balance>900</balance>
        </account>
    </customer>
    <customer>
        <customer_name>Hayes</customer_name>
        <customer_street>Main</customer_street>
        <customer_city>Harrison</customer_city>
        <account>
            <account_number>A-102</account_number>
            <branch_name>Perryridge</branch_name>
            <balance>400</balance>
        </account>
    </customer>
</bank-1>
```

**Fig. 12. 2**

XML allows us to give attributes to an element. For example the type of an account can be represented as an attribute. The attributes appear as name=value pairs before the closing '>' of a tag. The attributes of an element can appear only once. Since XML documents are designed to be exchanged between applications, a namespace mechanism has been introduced to allow organizations to specify globally unique names to be used as element tags in documents. The idea of a namespace is to prepend each tag or attribute with a universal resource identifier. Organizations may prefer their web URL as namespace, as they are unique. XML not only allows long namespace identifiers but also abbreviations for them. A document can have more than one namespace, declared as part of the root element. However a default namespace can be defined by using the xmlns attribute. The following example shows how a namespace can be defined.

```
<bank xmlns:FB=http://www.FirstBank.com>
...
  <FB:branch>
    <FB:branch_name> Downtown </FB:branch_name>
    <FB:branch_city> Brooklyn </FB:branch_city>
  </FB:branch>
...
</bank>
```

Sometimes we need to store values containing tags without the tags interpreted as XML tags. This can be done as <![CDATA[<account> ... </account>]]>. Here CDATA stands for character data. Because <account> is enclosed in CDATA it is interpreted as normal text, not as a tag.

## 12. 2. XML Document Schema:

Like databases, XML document data can be constrained by a schema. A schema constrains the type and value of data to be stored in a document or a database. Document Type Definition (DTD) is included as a schema definition language in XML standard. DTD constrains the appearance of sub-elements and attributes within an element. It is a list of rules that specifies the sub-elements and attributes. An example of DTD is given in the following picture.

```
<!DOCTYPE bank [
  <!ELEMENT bank ((account | customer | 60
depositor)+)>
  <!ELEMENT account (account_number branch_name balance)>
  <!ELEMENT customer (customer_name customer_street customer_city)>
  <!ELEMENT depositor (customer_name account_number)>
  <!ELEMENT account_number (#PCDATA)>
  <!ELEMENT branch_name (#PCDATA)>
  <!ELEMENT balance (#PCDATA)>
  <!ELEMENT customer_name (#PCDATA)>
  <!ELEMENT customer_street (#PCDATA)>
  <!ELEMENT customer_city (#PCDATA)>
]>
```

**Fig. 12. 3**



The above picture defines a schema for an XML document prepared as part of a bank application. In that document a bank element consists of one or more account, customer, or depositor elements. The | operator specifies “or”. The + operator specifies “one or more”. The \* operator may be used to specify “zero or more”. One of the types used in DTD is #PCDATA, which stands for “parsed character data”. The above picture shows the allowable attributes for each element, and imposes no order for them. Attributes may be specified to be of type CDATA, ID, IDREF, or IDREFS. ID type attribute provides a unique identifier for the element. An element of type IDREF is a reference to an element, and it contains a value, which is given in ID attribute of some element. IDREFS type attribute allows references to multiple elements. These references are separated by spaces. The ID and IDREF attributes serve the same purpose as reference mechanism in object-oriented and object-relational databases. They allow construction of complex data relationships. An example of XML data with ID and IDREF attributes is shown in the following picture.

```
<bank-2>
  <account account_number="A-401" owners="C100 C102">
    <branch_name>Downtown</branch_name>
    <balance>500</balance>
  </account>
  <account account_number="A-402" owners="C102 C101">
    <branch_name>Perryridge</branch_name>
    <balance>900</balance>
  </account>
  <customer customer_id="C100" accounts="A-401">
    <customer_name>Joe</customer_name>
    <customer_street>Monroe</customer_street>
    <customer_city>Madison</customer_city>
  </customer>
  <customer customer_id="C101" accounts="A-402">
    <customer_name>Lisa</customer_name>
    <customer_street>Mountain</customer_street>
    <customer_city>Murray Hill</customer_city>
  </customer>
  <customer customer_id="C102" accounts="A-401 A-402">
    <customer_name>Mary</customer_name>
    <customer_street>Erin</customer_street>
    <customer_city>Newark</customer_city>
  </customer>
</bank-2>
```

**Fig. 12. 4**

Despite its flexibility and ease of use, the DTD has some deficiencies. To overcome those deficiencies, XML Schema language was introduced. This language defines a number of built-in types such as **string**, **integer**, **decimal**, **date**, and **Boolean**. It allows user-defined types using constructors such as **complexType** and **sequence**. The following figure gives the XML schema

version of DTD, which is given in the previous example

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xs:element name="bank" type="BankType"/>
<xs:element name="account">
<xs:complexType>
  <xs:sequence>
    <xs:element name="account_number" type="xs:string"/>
    <xs:element name="branch_name" type="xs:string"/>
    <xs:element name="balance" type="xs:decimal"/>
  </xs:sequence>
</xs:complexType>
</xs:element>
<xs:element name="customer">
  <xs:element name="customer_number" type="xs:string"/>
  <xs:element name="customer_street" type="xs:string"/>
  <xs:element name="customer_city" type="xs:string"/>
</xs:element>
<xs:element name="depositor">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="customer_name" type="xs:string"/>
      <xs:element name="account_number" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:complexType name="BankType">
  <xs:sequence>
    <xs:element ref="account" minOccurs="0" maxOccurs="unbounded"/>
    <xs:element ref="customer" minOccurs="0" maxOccurs="unbounded"/>
    <xs:element ref="depositor" minOccurs="0" maxOccurs="unbounded"/>
  </xs:sequence>
</xs:complexType>
</xs:schema>
```

**Fig. 12. 5**

The first element is the root element. Other elements definitions follow the root element. Any element type that has either attributes or nested subelements must be specified to be a complex type. An XML schema can define minimum and maximum number of occurrence of subelements by using **minOccurs** and **maxOccurs**.

The attributes of elements can also be specified using `xs:attribute` tag, where `xs` is a namespace. An attribute can be made a compulsory attribute for an element by adding `use="require"` to its definition. The default value of `use` is optional. Such definitions would appear directly under the enclosing `complexType` specification.

In addition to defining types, an XML schema allows the specification of keys and references. The XML schema has several advantages over DTD. They are listed below.

1. It allows user-defined types.
2. It allows the text that appears in elements to be constrained to specific types.
3. It allows uniqueness and foreign-key constraints.
4. It is integrated with namespaces to allow different parts of a document to conform to different schemas.
5. It allows types to be restricted to create specialized types.
6. It allows complex types to be extended to by using a form of inheritance.

### 12. 3. Querying and Transformation

The need for efficient XML data management tools increased with the increase in the number of applications using it. Several languages provide querying and transformation capabilities:

1. XPath: It is a language for path expressions.
2. XQuery: It is the standard language for querying XML data.
3. XSLT: This language was designed to be a transformation language, as part of the style sheet system, XSL, which is used to control the formatting of XML data into HTML or other print or display languages.

#### **XPath**

This language addresses parts of an XML document by means of path expressions. Its current version is 2.0. A path expression in XPath is a sequence of location steps separated by `/`. The result of a path expression is a set of nodes. It is evaluated from left to right. The initial `/` indicates the root of the document. The result of a path at any point consists of an ordered set of nodes from the document. Attribute values may also be accessed using `@` symbol. The `text ()` function is used to extract just the values between the tags in the final set of nodes. XPath has lot of features. Some of them are listed below.

1. This path expression results three nodes from the previous example.  
`/bank-2/customer/customer.name`
2. The expression `/bank-2/customer/customer.name/text()` would return the same names, but without the enclosing tags.
3. Selection predicates may follow any step, and are contained in square brackets. For example: `/bank-2/account [balance > 400]` returns account elements with balance greater than 400.
4. XPath provides several functions that can be used as part of predicates. For example `count()`, `not(...)`, etc.
5. The function `id("foo")` returns the node with an attribute of type ID and value "foo".
6. The `|` operator allows expression results to be unioned. For example  
`/bank-2/account/id(@owner)|/bank-2/loan/id(@borrower)` gives customers with either accounts or loans.

7. An XPath expression can skip multiple levels of nodes by using “//”.
8. A step in the path need not just select from the children of the nodes in the current node set.
9. The built-in function doc(name) returns the root of a named document. For example, if “bank.xml” contains the bank example data, then the following path expression would return all accounts at the bank.

```
doc("bank.xml")/ bank/account
```

## XQuery

XQuery is the standard query language for XML. This language derives from an XML query language called Quilt. Quilt includes features from XPath and other XML query languages, XQL and XML-QL. XQuery queries are modeled after SQL queries. They are organized into five sections: **for**, **let**, **where**, **order by**, and **return**. They are referred to as “**FLWOR**” expressions.

**for** clause: It is like the from clause of SQL. It specifies variables that range over the results of XPath expressions. When more than one variable is specified, the results include the Cartesian product of the possible values the variables can take.

**let** clause: It allows the results of XPath expressions to be assigned to variable names for simplicity of representation.

**where** clause: It performs additional tests on the joined tuples from the **for** clause.

**order by** clause: It allows sorting of the output of the query.

**return** clause: This clause allows the construction of results in XML.

The following is an example of a FLWOR expression.

This XML query returns the account numbers for checking accounts.

```
for $x in /bank-2/account
let $acctno := $x/@account_number
where $x/balance > 400
return <account<number> { $acctno } </account_number>
```

In the above XML query, the text in curly brackets (“{ }”) is treated as expression to be evaluated. Another example of XQuery that gives a way of constructing elements is given below.

This XML query returns the account numbers for checking accounts.

```
for $x in /bank-2/account
let $acctno := $x/@account_number
where $x/balance > 400
return element account {
  attribute account_number { $x/@account_number },
  attribute branch_name { $x/branch_name },
  element balance { $x/balance }
```

## Joins

Joins can be specified in XQuery. The join of depositor, account, and customer elements in fig.1, can be written in XQuery this way

```
for $a in /bank/account,
   $c in /bank/customer,
```

```
$d in /bank/depositor
```

```
Where $a/account_number = $d/account_number
```

```
and $c/customer_name = $d/customer_name
```

```
return <cust_acct> {$c $a } </cust_acct>
```

Path expressions in XQuery are the same as path expressions in XPath. Path expressions may return a single value or element, or a sequence of values or elements. In the absence of schema information it may not be possible to infer whether a path expression returns a single value or a sequence of values. Such path expressions may participate in comparison operations such as =, <, and >. These operations are specially defined in XQuery. For example in the expression \$x/balance > 400, if the \$x/balance returns sequence containing multiple values, then the expression evaluates to true if at least one of the values is greater than 400. This behavior of the comparison operators is opposite to that of the following operators: **eq**, **ne**, **lt**, **gt**, **le**, **ge**.

### Nested Queries

XQuery FLWOR expressions can be nested in the **return** clause, in order to generate element nestings that do not appear in the source document. This feature is similar to nested subqueries in the “**from**” clause of SQL. XQuery provides a variety of aggregate functions such as sum() and count() that can be applied on sequences of elements or values. The function distinct-values() applied on a sequence returns a sequence without duplicate values. XQuery does not provide a **group by** construct. The following are two examples on nested queries.

A query on bank XML schema to find total balance on all accounts owned by each customer:

```
for $c in /bank/customer
```

```
return
```

```
  <customer-total-balance>
```

```
    <customer_name> {$c/customer_name} </customer_name>
```

```
    <total_balance> {fn:sum(
```

```
      for $d in /bank/depositor[customer_name = $c/customer_name],
```

```
      $a in /bank/account[account_number = $d/account_number]
```

```
      return $a/balance)}
```

```
    </total_balance>
```

```
  </customer-total-balance>
```

An XML query to generate bank XML structure so that account elements are nested within customer elements:

```
<bank-1> {
```

```
  for $c in /bank/customer
```

```
  return
```

```
    <customer>
```

```
      {$c/*}
```

```
      { for $d in /bank/depositor[customer_name = $c/customer_name],
```

```
        $a in /bank/account[account_number = $d/account_number]
```

```
        Return $a}
```

```
    </customer>
```

```
}</bank-1>
```

## Sorting of Results

Results can be sorted in XQuery by using the **order by** clause. The default order is ascending order. To sort the result in descending order we can use “**order by** attribute-name **descending**”. Sorting can be done at multiple levels of nesting. The following example gives a nested representation of bank information sorted in customer name order, with accounts of each customer sorted by account number.

```
<bank-1>{
  for $c in /bank/customer
  order by $c/customer_name
  return
    <customer>
      {$c/*}
      {for $d in /bank/depositor[customer_name = $c/customer_name],
        $a in /bank/account[account_number = $d/account_number]
        order by $a/account_number
        return <account> {$a/*} </account>}
    </customer>
} </bank-1>
```

**Note:** \$c/\* refers to all the children of the node (or sequence of nodes) bound to the variable \$c.

## Functions and Types

XQuery provides a variety of built-in functions, such as numeric functions and string matching and manipulation functions. In addition, XQuery supports user-defined functions. The following user-defined function returns a list of all balances of a customer with a specified name:

```
define function balances(xs:string $c) as xs:decimal* {
  for $d in /bank/depositor[customer_name = $c],
    $a in /bank/account[account_number = $d/account_number]
  return $a/balance
}
```

The type specifications for function arguments and return values are optional, and may be omitted. XQuery uses the type system of XML schema. The namespace prefix xs: used in the above example is predefined by XQuery. Types can be suffixed with a \* to indicate a sequence of values of that type. Types can also be partially specified. XQuery performs conversion automatically whenever required. XQuery also provides functions to convert between types. For example number(x) converts a string x, to a number.

## XSLT

A style sheet is a representation of formatting options for a document. The XML Stylesheet Language (XSL) is a logical extension of HTML style sheets. The language includes a general-purpose mechanism, called **XSL Transformations (XSLT)**. This mechanism is used to transform one XML document into another XML document, or to other formats such as HTML.

XSLT transformations are expressed as a series of recursive rules, called **templates**. The templates allow selection of nodes in an XML tree by an XPath expression. Templates can also generate new XML content. With this selection and content generation can be mixed. A simple template for XSLT consists of a **match** part and a **select** part. The following XSLT example can be used to wrap results in new XML elements.

```
<xsl:template match="/bank-2/customer">
  <xsl:value-of select="customer_name"/>
</xsl:template>
<xsl:template match="*" />
<xsl:template match="/bank-2/customer">
  <customer>
    <xsl:value-of select="customer_name"/>
  </customer>
</xsl:template>
<xsl:template match="*" />
```

The match statement in the above example contains an XPath expression that selects one or more nodes. The first template matches customer elements that occur as children of the bank-2 root element. The xsl:value-of statement enclosed in the match statement outputs values from the nodes in the result of the XPath expression. The first template outputs the value of the customer\_name subelement.

**Structural recursion** is a key part of XSLT. This means, when a template matches an element in the tree structure, XSLT can use structural recursion to apply template rules recursively on subtrees, instead of just outputting a value. It applies rules recursively by the xsl:apply-templates directive, which appears inside other templates. The following example shows recursive application of rules.

```
<xsl:template match="/bank">
  <customers>
    <xsl:apply-templates/>
  </customers>
</xsl:template>
<xsl:template match="/customer">
  <customer>
    <xsl:value-of select="customer_name"/>
  </customer>
</xsl:template>
<xsl:template match="*" />
```

XSLT provides a feature called **keys** that permit lookup of elements by using values of subelements

or attributes. This feature permits, attributes other than the ID attributes, to be used. Keys are defined by an `xsl:key` directive. The directive has three parts as shown in the following example.

```
<xsl:key name="acctno" match="account" use="account_number"/>
```

The name attribute is used to distinguish different keys. The match attribute specifies which nodes the key applies to. The use attribute specifies the expression to be used as the value of the key. The keys can be used in templates as part of any pattern through the key function. For example: `key("acctno", "A-401")` references to the XML node for account "A-401". Keys can be used to implement some types of joins, as in the following figure. In this example the key function joins the depositor elements with matching customer and account elements

```
<xsl:key name="acctno" match="account" use="account_number"/>
<xsl:key name="custno" match="customer" use="customer_name"/>
<xsl:template match="depositor">
  <cust_acct>
    <xsl:value-of select=key("custno", "customer_name")/>
    <xsl:value-of select=key("acctno", "account_number")/>
  </cust_acct>
</xsl:template>
<xsl:template match="*" />
```

**Fig. 12. 6**

## 12. 4. Application program Interfaces to XML

XML got wide acceptance as data representation and exchange format. Software tools are widely available for manipulation of XML data. There are two standard models for programmatic manipulation of XML. Both these APIs can be used to parse an XML document and create an in-memory representation of the document. One of the standard APIs for manipulating XML is based on the document object model (DOM), which treats XML content as a tree, with each element represented by a node, called a DOM node.

DOM libraries are available for most common programming languages and are even present in web browsers. DOM provides a variety of functions for updating the document by adding and deleting attribute and element children of a node, setting node values, and so on. DOM can be used to access XML data stored in databases, and an XML database can be built with DOM. The DOM interface does not support any form of declarative querying. Some of the interfaces and methods in Java API for DOM are given in the following list.

- The java DOM API provides an interface called Node, and interfaces Element and Attribute, which inherit from the Node interface.
- The Node interface provides methods such as `getParentNode()`, `getFirstChild()`, and `getNextSibling()` to navigate the DOM tree, starting with the root node.
- Subelements of an element can be accessed by name using `getElementsByTagName(name)`.
- Attribute values of an element can be accessed by name, using the method `getAttribute(name)`.
- The text value of an element is modeled as a Text node, which is a child of the element node. An element node with no subelements has only one such child node. The method `getData()` on the Text node returns the text contents.



The second commonly used programming interface is the Simple API for XML (SAX). It is an event model, designed to provide a common interface between parsers and applications. This API is built on the concept of event handlers. Parsing events correspond to the recognition of parts of a document. The SAX application developer creates handler function for each event, and registers them. When the SAX parser reads a document in, as each event occurs, the handler function is called with parameters describing the event. The handler functions then carry out their task. SAX generally requires more programming effort than DOM. It does not need the creation of DOM tree, where the application needs to create its own data representation.

### CH. 5. Storage of XML data

Many applications need to store XML data. The different ways of XML data storage are given in the following list.

#### Non-relational Data Stores

- Store in flat files.
- Create an XML database.

#### Relational Databases

- Store as String
- Tree Representation.
- XML-enabled database.
- Native Storage within a Relational Database.

XML data may be stored in a file. This approach has drawbacks. This approach lacks data isolation, atomicity, concurrent access, and security. For small applications this storage is sufficient, if proper tools are used. An alternative is to create an XML database. Early XML databases implemented the Document Object Model on a C++ based object-oriented database. The addition of XQuery or other XML query languages provides declarative querying. However building a full-featured XML data storage and querying but also other database features like security is a very complex task. So it is better to use an existing database system.

Relational databases are widely used in existing applications. So it is wise to convert XML data to relational form, which is a straightforward task if the data were generated from a relational database. When the XML data is not generated from a relational schema, the task of converting it to relational form is not straightforward. In such cases alternative approaches are taken. One alternative is to store small XML documents as string (**clob**) values in tuples in a relational database. Large XML documents with the top-level element having many children can be handled by storing each child element as a string in a separate tuple. A refined alternative is to store different types of elements in different relations, and also store the values of some critical elements as attributes of the relation to enable indexing. Some database systems such as Oracle, support **function indices**, which can help avoid replication of attributes between the XML string and relation attributes.

#### Tree Representation

Arbitrary XML data can be modeled as a tree and stored using a pair of relations:

nodes (id, type, label, value)

child (child\_id, parent\_id)

Each element and attribute in the XML data is given a unique identifier. A tuple is inserted in the nodes relation for each element and attribute with its identifier (id), its type, the name of the element or attribute (label), and the text value of the element or attribute (value). The relation child is used to record the parent element of each element and attribute. To preserve order information an extra

attribute “position” can be added to the child relation to indicate the relative position among the children of the parent.

### Map to Relations

In this approach, XML elements whose schema is known are mapped to relations and attributes. Elements whose schema is unknown are stored as strings or as a tree. A relation is created for each element type, whose schema is known. The attributes of the relation are defined as follows.

- All attributes of these elements are stored as string-valued attributes of the relation.
- If a subelement of the element is a simple type, an attribute is added to the relation and its type defaults to string.
- Otherwise, a relation is created corresponding to the subelement.
  - An identifier attribute is added to the relations representing the element.
  - An attribute `parent_id` is added to the relation representing the subelement, with its value set to its parent identifier.
  - If ordering is to be preserved, an attribute `position` is added to relation representing the subelement.

### XML-enabled database

XML is used to exchange data between business applications. Most often this data originates in relational databases. Data in Relational databases is published to XML form and incoming data is shredded. An XML-enabled database supports an automatic mechanism for publishing relational data as XML. This mechanism may use a simple and straightforward mapping or a more complicated mapping. At the simplest, each row of a table might be mapped to an XML element and each column in that row a subelement of the XML element. A more complicated mapping would allow nested structures to be created. SQL extension queries with nested queries in the select clause are used for easy creation of nested XML output.

### Native Storage within a Relational Database

Some Relational databases have begun to support native storage of XML. They store XML data as strings or in binary representations, without converting the data to relational form. A new data type **xml** is provided to represent XML data. A relation with an attribute of type **xml** can be used to store a collection of XML documents. Each document is stored as a value of **xml** in a separate tuple.

### SQL / XML

The SQL / XML is a standard that defines the extensions of SQL, allowing the creation of nested XML output. This standard includes ways to map SQL types to XML schema types, and relational schemas to XML schemas. For example the following picture shows the SQL / XML representation of the bank schema containing the relations account, customer, and depositor.

```
<bank>
<account>
  <row>
    <account_number>A-101</account_number>
    <branch_name>Downtown</branch_name>
    <balance>500</balance>
```

```
</row>
<row>
  <account_number>A-102</account_number>
  <branch_name>Perryridge</branch_name>
  <balance>400</balance>
</row>
<row>
  <account_number>A-201</account_number>
  <branch_name>Brighton</branch_name>
  <balance>900</balance>
</row>
</account>
<customer>
  <row>
    <customer_name>Johnson</customer_name>
    <customer_street>Alma</customer_street>
    <customer_city>Palo Alto</customer_city>
  </row>
  <row>
    <customer_name>Hayes</customer_name>
    <customer_street>Main</customer_street>
    <customer_city>Harrison</customer_city>
  </row>
</customer>
<depositor>
  <row>
    <account_number>A-101</account_number>
    <customer_name>Johnson</customer_name>
  </row>
  <row>
    <account_number>A-201</account_number>
    <customer_name>Johnson</customer_name>
  </row>
  <row>
    <account_number>A-102</account_number>
    <customer_name>Hayes</customer_name>
  </row>
</depositor>
</bank>
```

**Fig. 12.7**

This standard adds several operators and aggregate operations to SQL, to allow the construction of XML output directly from the extended SQL. The following list describes a few such functions and operators.

**xmlelement** function: Used to create XML elements.

**xmlattributes** function: Used to create attributes

**xmlagg** function: Used to create a forest (collection) of XML elements from the collection of values on which it is applied.

**xmlforest** operator: Used to simplify the construction of XML structures. It is similar to xmlattributes function.

**xmlconcat** operator: Used to concatenate elements created by sub-expressions into a forest.

An example query to create an XML element for each account is given below.

```
select xmlelement( name "account",
  xmlattributes( account_number as account_number),
  xmlelement( name "branch_name", branch_name),
  xmlelement( name "balance", balance))
```

```
from account
```

## 12. 6. XML Applications

XML is useful in several applications like storing data with complex structures, exchanging data, web applications, and data mediation. Standard have been developed based on XML for representation of data for various applications. Protocols have been defined for invoking procedures, using XML for representing the procedure input and output. In many web applications XML is becoming the default tool for data mediation. The following four points briefly discuss the uses of XML.

### Standardized Data Exchange Formats

XML based standards, for data representation, have been developed for a variety of specialized applications. Some sample areas are listed below:

- ChemML is a standard for representing information needed by chemical industry. This standard allows representation of information about chemicals, such as their molecular structure, boiling and melting points, solubility in various solvents, calorific values etc.
- In shipping, customs and tax officials need shipment records containing detailed information about the goods being shipped, from whom and to where they were sent, to whom and to where they are being shipped, the monetary value of the goods and so on.
- The RosettaNet standards for e-business applications define XML schemas and semantics for representing data as well standards for message exchange.

### Storing Data with Complex Structure

Many applications need to store data that are structure, but are not easily modeled as relations. Such applications prefer to store the data in XML format. The XML format is simple and can represent any complex structured data. XML-based representations have been proposed as standards for storing documents, spreadsheet data and other data that are part of office application packages. XML is used to represent data with complex structure that must be exchanged between different parts of an application.

## Web Services

A common requirement of organizations is to require data from outside of the organization or from another department of the same organization that uses a different database. In such cases XML is a useful format for exchange of data between organizations or intra-organization data exchange. When the information is to be used directly by a human, organizations provide web-based forms, where users input some values and get the desired information in HTML format. When a software program needs such information, then the results can be provided in XML format.

The Simple Object Access Protocol (SOAP) defines a standard for invoking procedures, using XML for representing the procedure input and output. A site providing such a collection of SOAP procedures is called a Web Service. To invoke a web service, a client must prepare an appropriate SOAP XML message and send it to the service. As a result, the client gets the response in XML format, which the client must process to extract the information contained in it.

## Data Mediation

Data mediation applications extract information from various resources and aggregate the information to present it to a user. For such applications the task will be simple if all the resources provide data in standard XML format. A mediator is an application used to combine the extracted information under a single schema. The mediator defines a single schema that represents all required information, and provides code to transform data between different representations.

## 12. 7. Summary

XML stands for Extensible Markup Language. It is a descendant of the Standard Generalized Markup Language (SGML). It was originally invented for providing functional markup for web documents, but has become the de facto standard for data exchange between applications. XML documents contain matching tags that indicate the beginning and end of an element. Elements may have subelements nested within them, to any level of nesting. Elements may also have attributes. One of the attributes of an element may be of type ID that stores unique identifier for the element. Elements may have attributes that point to other elements.

XML data can be represented as tree structures, with nodes corresponding to elements and attributes. Nesting of elements is reflected by the parent-child structure of the tree representation. The XQuery language is the standard language for querying XML data. It has a structure similar to SQL. It has for, let, where, order by and return clauses. It supports nested queries and userdefined functions.

The DOM and SAX APIs are widely used for programmatic access to XML data. These APIs are available with a variety of programming languages

## 12. 8. Technical Terms

**HTML:** Hyper Text Markup Language.

**SGML:** Standard Generalized Markup Language.

**XML:** Extensible Markup Language.

**XPath:** A standard notation for specifying the path of an element in an XML document.

**XQuery:** A query used to extract desired information from an XML document.

**SQL:** Structured Query Language.

**DOM:** Document Object Model.

**SOAP:** Simple Object Access Protocol.

**API:** Application Program Interface.

**Schema:** A part of the document that describes the structure and content of the document.

**XSL:** XML stylesheet language.

## 12. 9. Model Questions

1. Define XML. Describe XML document structure.
2. Describe the XML Schema.
3. Explain XPath and XQuery with examples.
4. Explain how XML became the de facto standard for data exchange.
5. Discuss the similarities between XML queries and SQL queries.
6. Give a detailed description of different clauses available in XML.
7. Write an XML query on the bank data given in this chapter, to extract all the accounts of a customer at a branch of the bank along with the balance.

## 12. 10. References

Database System Concepts by Henry F. Korth & S. Sudarshan. Fifth Edition

## **Lesson 13**

# **Storage Structure**

### **13.0 Objectives:**

After completion of this lesson the student will be able to know about:

- Classification of storage media.
- Different characteristics of storage media.
- To define data structures that will allow fast access to data.
- Alternative structures to different kinds of access to data.

### **Structure Of the Lesson:**

#### **13.1 Overview of Physical Storage Media**

#### **13.2 Magnetic Disks**

##### **13.2.1 Physical Characteristics of Disks**

##### **13.2.2 Performance Measures of Disks**

##### **13.2.3 Optimization of Disk-Block Access**

#### **13.3 RAID**

##### **13.3.1 Improvement of Reliability and Redundancy**

##### **13.3.2 Improvement in Performance via Parallelism**

##### **13.3.3 RAID Levels**

##### **13.3.4 Choice of RAID level**

##### **13.3.5 Hardware Issues**

#### **13.4 Tertiary Storage**

##### **13.4.1 Optical Disks**

##### **13.4.2 Magnetic Tapes**

#### **13.5 Storage Access**

##### **13.5.1 Buffer Manager**

##### **13.5.2 Buffer-Replacement Policies**

#### **13.6 Summary**

#### **13.7 Technical terms**

#### **13.8 Model questions**

#### **13.9 References**

## 13.1. Overview of Physical Storage Media

The storage media are classified by the speed with which data can be accessed, cost per unit of data, reliability. Among the media typically available are:

**Cache:** Cache is the fastest and most costly form of storage. Cache is small and is managed by the computer system hardware.

**Main Memory:** The storage medium used for data that are available to be operated on is main memory. General-purpose instructions operate on main memory. Even though it contains many megabytes of data, main memory is too small to (or too expensive) store the entire database. The contents of main memory are usually lost if a power failure or system crash occurs.

**Flash Memory:** Differs from main memory, in that data survive power failure. Reading data from flash memory takes less than 100 nano seconds, which is roughly as fast as reading data from main memory. In flash memory data can be written at a location only once, but location can be erased and then ready to be written again. A drawback of flash memory is it can support only a limited number of write/erase cycles and erasing of memory has to be done to an entire bank of memory. Reading data is roughly as fast as main memory but writing is slow, erasing is slower.

Cost per unit of storage is roughly similar to main memory. Widely used in embedded devices such as digital cameras also known as EEPROM (Electrically Erasable Programmable Read-Only Memory).

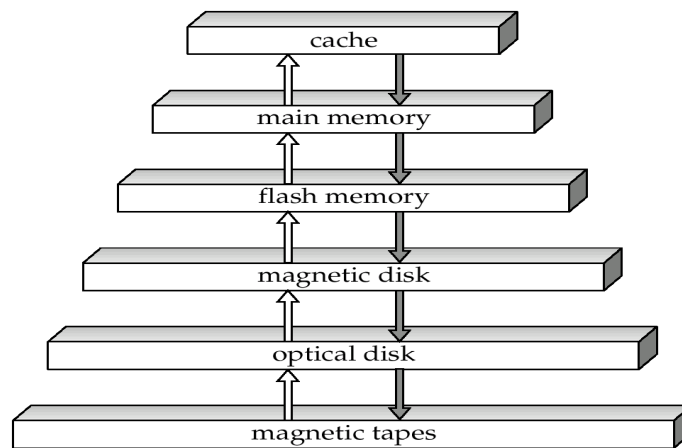
**Magnetic Disk storage:** The primary storage medium for the long-term storage of data is the magnetic disk. The entire database is stored on magnetic disk. Data is stored on spinning disk, read/written magnetically. Data must be moved from disk to main memory for access, and data that have been modified must be written back for storage. Disk storage is referred as direct-access storage because it is possible to read data from any location on disk. Magnetic disk has much slower access than main memory. Capacities range up to roughly 100 GB currently. Much larger capacity and cost/byte than main memory/flash memory. Growing constantly and rapidly with technology improvements (factor of 2 to 3 every 2 years). Survives power failures and system crashes. Disk failure can destroy data, but is very rare.

**Optical Storage:** Non-volatile, data is read optically from a spinning disk using a laser CD-ROM (640 MB) and DVD (4.7 to 17 GB) most popular forms. Write-once, read-many (WORM) optical disks used for storage (CD-R and DVD-R). Multiple write versions are also available (CD-RW, DVD-RW, and DVD-RAM). Reading and writing is slower than with magnetic disk. **Jukebox** systems, with large numbers of removable disks, a few drives, and a mechanism for automatic loading/unloading of disks available for storing large volumes of data.

**Tape Storage:** Non-volatile, used primarily for backup (to recover from disk failure), and for archival data. Tape storage is referred as sequential storage. Much slower than disk. Very high capacity (40 to 300 GB tapes available). Tape can be removed from drive so, storage costs much cheaper than disk, but drives are expensive. Tape jukeboxes available for storing massive amounts of data.

The various storage media can be organized in a hierarchy (**Figure 1**) according to their speed and their cost. The higher levels are expensive, but are fast. As we move down the hierarchy, the cost per bit decreases, where as the access time increases.





**Figure: 1** Storage Device Hierarchy

**Primary Storage:** The fastest storage media but volatile (cache, main memory).

**Secondary Storage:** The next level in hierarchy, non-volatile, moderately fast access time also called **on-line storage**. Secondary storage devices are **flash** memory, magnetic disks etc.

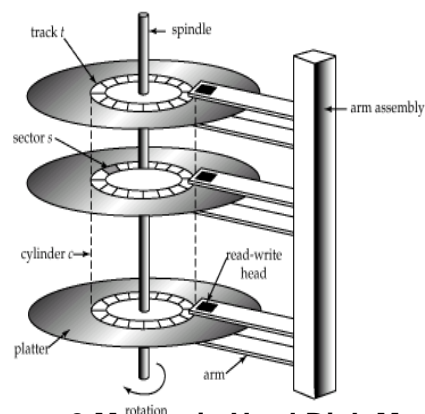
**Tertiary Storage:** Lowest level in hierarchy, non-volatile, slows access time also called **off-line storage**. Tertiary storage devices are magnetic tape, optical storage etc.

## 13.2 Magnetic Disks

Magnetic disks provide the bulk of secondary storage for modern computer systems. Disk capacities have been growing at over 50 percent per year, but the storage requirements of large applications have also been growing very fast.

### 13.2.1 Physical Characteristics of Disks

Physically, disks are relatively simple (**Figure 2**) each platter has a flat circular shape. Its two surfaces are covered with a magnetic material, and information is recorded on the surfaces. Platters are made from rigid metal or glass



**Figure: 2** Magnetic Hard Disk Mechanisms

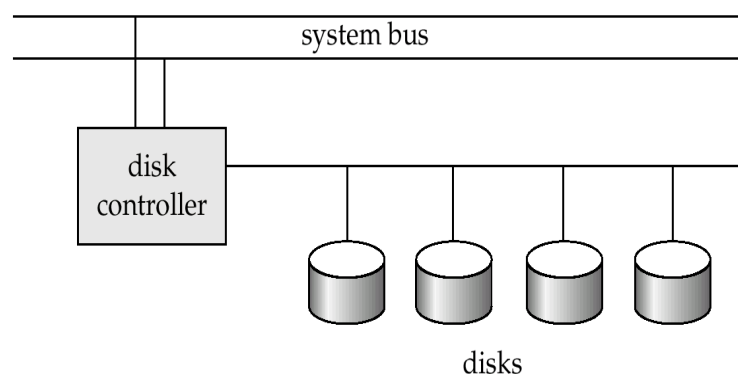
When the disk is in use, a drive motor spins it at a constant high speed. There is a **Read-write head** positioned very close to the platter surface (almost touching it). The disk surface is logically divided into circular **tracks**, which are subdivided into **sectors**.

A sector is the smallest unit of data that can be read from or written to the disk. Sector sizes are typically 512 bytes; there are about 50,000 to 100,000 tracks per platter, and 1 to 5 platters per disk. The inner tracks are of smaller length and outer tracks contain more sectors than the inner tracks. Typical sectors per track: 200 (on inner tracks) to 400 (on outer tracks).

The read-write head stores information on a sector magnetically as reversals of the direction of magnetization of the magnetic material. To read-write a sector, disk arm swings to position head on right track platter spins continually; data is read/written as sector passes under head. The disk platters mounted on a spindle and the heads mounted on a disk arm are together known as head-disk assemblies. Since the heads on all the platters move together, when the head on one platter is on the  $i^{\text{th}}$  track, the tracks on all the other platters are also on the  $i^{\text{th}}$  track. Hence the  $i^{\text{th}}$  tracks of all the platters together are called the  $i^{\text{th}}$  **cylinder**. Earlier generation disks were susceptible to head-crashes. Surface of earlier generation disks had metal-oxide coatings, which would disintegrate on head crash and damage all data on disk. Current generation disks are less susceptible to such disastrous failures, although individual sectors may get corrupted.

A **Disk controller** Interfaces between the computer system and the actual hardware of the disk drive. A disk controller accepts high-level commands to read or write a sector, and initiates actions, such as moving the disk arm to the right track and actually reading or writing the data. Disk controllers also attach **checksums** to each sector to verify whether data is read back correctly. If data is corrupted, with very high probability stored checksum won't match recomputed checksum. If such an error occurs, the controller will retry the read several times; if the error continues to occur, the controller will signal a read failure.

Another task that disk performs is remapping of bad sectors. If the controller detects that a sector is damaged when the disk is initially formatted or when an attempt is made to write the sector, it can logically map the sector to a different physical location. **Figure 3** shows how disks are connected to a computer system. There are a number of common interfaces for connecting disks to personal computers and workstations:



**Figure: 3** Disk Subsystem.

1. The **AT** attachment (**ATA**) interface.
2. The new version of **ATA**, which is **STA** serial **ATA**.
3. The small-computer-system-interconnect (**SCSI**).

### 13.2.2 Performance Measures of Disks

The main measures of the qualities of disk are capacity, access time, data-transfer rate, and reliability.

**Access time** is the time it takes from when a read or write request is issued to when data transfer begins. To access data on a given sector of a disk, the arm must move so that it is positioned over the correct track, and then must wait for the sector to appear under it as the disk rotates.

**Seek time** is the time it takes to reposition the arm over the correct track.

**Average seek time** is the average of seek times. If all tracks have the same number of tracks and we discard the time required for the head to start moving and to stop moving the average seek time is  $1/3$ . Taking these factors into account, the average seek time is  $1/2$  of maximum seek time. Average seek time range between 4 to 10 milliseconds on typical disks.

**Rotational latency** is the time spent waiting for the sector to be accessed to appear under the head. Average latency time is  $1/2$  the time for a full rotation of the disk. Rotational speeds of disks range from 4 to 11 milliseconds on typical disks. The access time is the sum of the seek time and the rotational latency. Ranging from 8 to 20 milliseconds.

**Data-transfer rate** is the rate at which data can be retrieved from or stored to the disk. Current disks support maximum transfer rates of 25 to 100 megabytes per second. Transfer rates are lower than the maximum transfer rates for inner tracks of the disk, since they have fewer sectors.

**Mean Time To Failure (MTTF)**, which is a measure of the reliability of the disk. The mean time to failure of a disk is the amount of time on average; we can expect the system to run continuously without any failure. The mean time to failure of disks ranges from 57 to 136 years. Probability of failure of new disks is quite low, corresponding to a "theoretical MTTF" of 30,000 to 1,200,000 hours for a new disk.

### 13.2.3 Optimization of Disk-Block Access

Requests for disk I/O are generated both by the file system and by the virtual memory manager found in most operating systems. Each request specifies the address on the disk to be referenced; that address is in the form of a block number.

A **Block** is a logical unit consisting of a fixed number of contiguous sectors. Data is transferred between disk and main memory in blocks. Block sizes range from 512 bytes to several kilobytes. Data are transferred between disk and main memory in units of blocks. The lower levels of the file-system manager convert block addresses into the hardware level cylinder, surface, and sector number.

**Scheduling:** If several blocks from a cylinder need to be transferred from disk to main memory, we may be able to save access time by requesting the blocks in the order in which they will pass under the heads. If the desired blocks are on different cylinders, it is advantageous to request the blocks in an order that minimizes disk-arm movement. **Disk-arm-scheduling** algorithms attempt to order accesses to tracks in a fashion that increases the number of accesses that can be processed.

**Elevator algorithm** move disk arm in one direction (from outer to inner tracks or vice versa), processing next request in that direction, till no more requests in that direction, then in reverse direction and repeat.

**File Organization:** Optimize block access time by organizing the blocks to correspond closely to how data will be accessed. For example, store related information on the same or nearby cylinders.

Some operating systems such as UNIX and Windows operating systems hide the disk organization from users, and manage the allocation internally. However over time sequential files may get **fragmented**. For example, if data is inserted/deleted from the file or free blocks on disk are scattered, and newly created file has its blocks scattered over the disk, sequential access to a fragmented file results in increased disk arm movement. Some systems have utilities to that, scan the disk and then move blocks to decrease the fragmentation.

**Nonvolatile Write Buffers:** The contents of main memory are lost in a power failure. Information about database updates has to be recorded on disk to survive possible system crashes. We can use nonvolatile random-access memory to speed up disk writes by writing blocks to a non-volatile RAM buffer immediately. The contents of NV-RAM are not lost in power failure. The common way to implement NV- RAM is to use battery backed up RAM. Even if power fails, the data is safe and will be written to disk when power returns. The controller writes to disk whenever the disk has no other requests or buffer becomes full.

**Log Disk:** A disk devoted to writing a sequential log of block updates. Used exactly like nonvolatile RAM. Writing to log disk is very fast since no seeks are required. No need for special hardware (NV-RAM). File systems typically reorder writes to disk to improve performance. File systems that support log disks are called **Journaling file systems**. **Journaling file systems** can be implemented even without a separate log disk, keeping data and the log on the same disk.

### 13.3. RAID

The data-storage requirements of some applications (Web, database and multimedia applications) have been growing so fast that a large number of disks are needed to store their data, even though disk drive capacities have been growing very fast. A variety of disk-organization technique called **Redundant Arrays of Independent Disks (RAID)**, have been proposed to achieve improved performance and reliability.

#### 13.3.1 Improvement of Reliability and Redundancy

**Reliability:** The chance that some disks out of a set of  $N$  disks will fail is much higher than the chance that a specific single disk will fail. Suppose that the mean time to failure of a disk is 1,00,000 hours, or slightly over 11 years. Then, the mean time to failure of some disk in an array of 100 disks will be  $100,000/100=1000$  hours, or around 42 days, which is not long at all. If we store only one copy of the data, then each disk failure will result in loss of a significant amount of data. Such a high rate of data loss is unacceptable. The solution to the problem of reliability is to introduce **redundancy**.

**Redundancy:** Store extra information that can be used to rebuild information lost in a disk failure.

**Mirroring (or shadowing):** The simplest way to achieve redundancy is duplicate every disk. This technique is called mirroring or shadowing. A logical disk then consists of two physical disks and every write is carried out on both disks. Reads can take place from either disk. If one disk in a pair fails, data is still available in the other. Data loss would occur only if the disk and its mirror disk also fails before the system is repaired. Probability of combined event is very small. Mean time to data

loss depends on mean time to failure, and mean time to repair.

Eg: MTTF (mean time to failure) of 100,000 hours, mean time to repair of 10 hours gives mean time to data loss of  $500 \times 10^6$  hours (or 57,000 years) for a mirrored pair of disks (ignoring dependent failure modes).

### 13.3.2 Improvement in Performance via Parallelism

With disk mirroring, the rate at which read requests could be handled is doubled, since read requests can be sent to either disk. The transfer rate of each read is the same as in a single-disk system, but the number of reads per unit time has doubled.

With multiple disks we can improve the transfer rate as well by striping data across multiple disks. In its simplest form, data striping consists of splitting the bits of each byte across multiple disks; such striping is called bit-level striping.

Block-level striping stripes blocks across multiple disks. It treats the array of disks as a single large disk, and it gives blocks logical numbers. Block-level striping is the most commonly used form of data striping. There are two main goals of parallelism in a disk system:

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

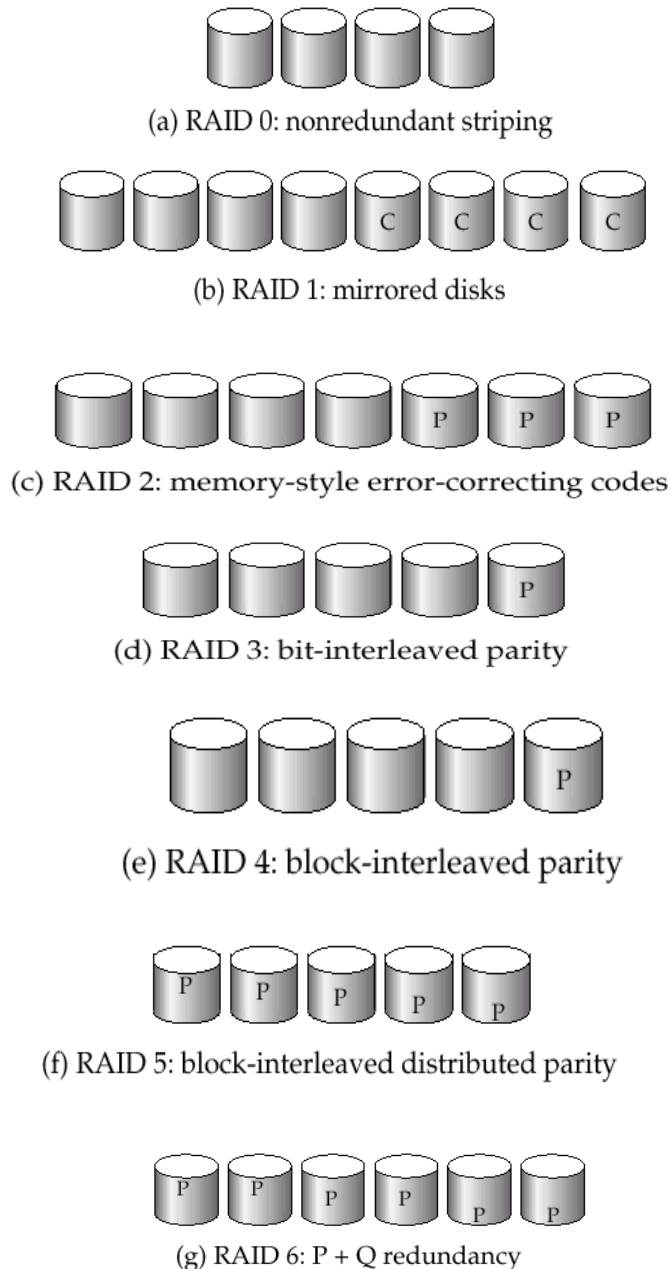
### 13.3.3 RAID Levels

Mirroring provides high reliability, but it is expensive. Striping provides high data-transfer rates, but does not improve reliability. Various alternative schemes aim to provide redundancy at lower cost by combining disk striping. These schemes have different cost-performance trade-offs. These schemes are classified into RAID levels. Different RAID organizations have differing cost, performance and reliability characteristics.

**RAID Level 0:** This level refers to disk arrays with striping at the level of blocks, but without any redundancy. **Figure 4a** shows an array of size 4.

**RAID Level 1:** This level refers to disk mirroring with block striping. **Figure 4b** shows a mirrored organization that holds four disks worth of data.

**RAID Level 2:** This level is known as memory-style error-correcting-code (ECC) organization, employs parity bits. Memory systems have long used parity bits for error detection and correction. Each byte in a memory system may have a parity bit associated with it that records whether the number of bits in the byte that are set to 1 is even (parity=0) or odd (parity=1). If one of the bits in the byte gets damaged, the parity of the byte changes and thus will not match the stored parity. Similarly if the stored parity bit gets damaged, it will not match the computed parity. Thus all 1-bit errors will be detected by the memory system. Error-correcting schemes store two or more extra bits, and can reconstruct the data if a single bit gets damaged. The idea of error correcting codes can be used directly in disk arrays by striping bytes across disks. **Figure 4c** shows the level 2 scheme. For example, the first bit of each byte could be stored in disk 1, the second bit in disk 2, and so on until the eighth bit is stored in disk 8, and the error-correction bits are stored in further disks.



**Figure: 4** RAID levels.

**RAID Level 3: Bit-Interleaved Parity**

Organization. Improves on level 2 by exploiting the fact that disk controllers unlike memory systems can detect whether a sector has been read correctly, so a single parity can be used for error correction as well as detection. The idea is if one of the sectors gets damaged, the system knows exactly which sector it is. **Figure 4d** shows the level 3 scheme. For each bit in the sector, the

system can figure out whether it is 1 or 0 by computing the parity of corresponding bits from sectors in the other disks. If the parity of the remaining bits is equal to the stored parity, the missing bit is 0, otherwise, it is 1.

**RAID Level 4:** Block-Interleaved Parity, uses block-level striping, and keeps a parity block on a separate disk for corresponding blocks from  $N$  other disks. This scheme is shown pictorially in **figure 4e**. If one of the disks fails, the parity block can be used with the corresponding blocks from the other disks to restore the blocks of the failed disk.

A block read accesses only one disk, allowing other requests to be processed by other disks. Thus, the transfer rate for each access is slower, but multiple read accesses can proceed in parallel, leading to higher I/O rates for independent block reads than Level 3. The transfer rates for larger reads is high, since all disks can be read in parallel. A write of a block has to access the disk on which the block is stored, as well as the parity disk, since the parity block has to be updated.

**RAID Level 5:** Block-Interleaved Distributed Parity; partitioning data and parity among all  $N + 1$  disks, rather than storing data in  $N$  disks and parity in one disk. **Figure 4f** shows the setup. The P's are distributed across all the disks. For example, with an array of 5 disks, the parity block for  $n^{\text{th}}$  set of blocks is stored on disk  $(n \bmod 5) + 1$ , with the data blocks stored on the other 4 disks. Higher I/O rates than Level 4. Block writes occur in parallel if the blocks and their parity blocks are on different disks. The pattern shown gets repeated on further blocks.

P0	0	1	2	3
4	P1	5	6	7
8	9	P2	10	11
12	13	14	P3	15
16	17	18	19	P4

**RAID Level 6:** P+Q Redundancy scheme: Similar to level 5, but stores extra redundant information to guard against multiple disk failures. Level 6 uses error-correcting codes instead of using parity. In the scheme in **figure 4g**, 2 bits of redundant data are stored for every 4 bits of data.

### 13.3.4 Choice of RAID level

The factors to be taken into account in choosing a RAID levels are:

- Monetary cost of extra disk-storage requirements.
- Performance requirements in terms of number of I/O operations.
- Performance when disk has failed.
- Performance during rebuilds.

RAID 0 is used only when data safety is not important. For example, data can be recovered quickly from other sources. Level 2 and 4 are never used since 3 and 5 subsumes them. Level 3 is not used anymore since bit-striping forces single block reads to access all disks, wasting disk arm movement, which block striping (level 5) avoids. Level 6 is rarely used since levels 1 and 5 offer adequate safety for almost all applications. So competition is between 1 and 5 only.

Level 1 provides much better write performance than level 5. Level 5 requires at least 2 blocks reads and 2 blocks writes to write a single block, whereas Level 1 only requires 2 block writes. Level 1 preferred for high update environments such as log disks.

### 13.3.5 Hardware Issues

Another issue in the choice of RAID implementations is at the level of hardware. RAID can be implemented with no change at the hardware level, using only software modification. Such RAID implementations are called **software RAID**.

However, there are significant benefits to be had by building special purpose hardware to support RAID, which we outline below; systems with special hardware support are called **hardware RAID** systems.

Some hardware RAID implementation permit **hot swapping**; that is, faulty disks can be removed and replaced by new ones without turning power off. Hot swapping reduces the mean time of repair.

## 13.4 Tertiary Storage

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

### 13.4.1 Optical Disks

Compact disks have been a popular medium for distributing software, multimedia data such as audio and images, and other electronically published information. They have a fairly large capacity (640 mega bytes), and they are cheap.

Digital video disks (DVDs) have now replaced compact disks in applications that require larger amounts of data. Disks in the DVD-5 format can store 4.7 Giga Bytes(Gb) of data (in one recording layer), while disks in the DVD-9 format can store 8.5 GB of data (in two recording layers). Recording on both sides of a disk yields even larger capacities; DVD-10 and DVD-18 formats, which are the two-sided versions of DVD-5 and DVD-9, can store 9.4 Gb and 17 Gb, respectively. CD and DVD drives have much longer sought times (100 milliseconds is common) than do magnetic disk drives, since the head assembly is heavier.

Rotational speeds are typically lower than those of magnetic disks, although the faster CD and DVD drives have rotation speeds of about 3000 rotations per minute, which is comparable to speeds of lower-end magnetic-disk drives. Rotational speeds of CD drives originally correspond to the audio CD standards, and the speeds of DVD drives originally corresponded to the DVD video standards, but current-generation drives rotate many times the standard rate.

Data transfer rates are somewhat less than for magnetic disks. Current CD drives read at around 3 to 6 Mbps(Mega bytes per second) and current DVD drives read at 8 to 20 Mbps. Like magnetic-disk drives, optical disks store more data in outside tracks and less data in inner tracks. The transfer rate of optical drives is characterized as  $nx$ , which means the drive supports transfers at  $n$  times the standard rate; rates of around 50x for CD and 16x for DVD are now common.

The record-once version of optical disks (CD-R, and ,DVD-R ) are popular for distribution of data and particularly for archival storage of data because they have a high capacity, have a longer lifetime than magnetic disks, and can be removed and stored at a remote location. Since they cannot be over written, they can be used to store information that should not be modified, such as audit trails. The multiple-write versions (CD-RW, DVD-RW, DVD+RW, and DVD-RAM) are also used for archival purposes. Jukeboxes are devices that store a large number of optical disks (up to several hundred) and load them automatically on demand to one of a small number of drives (usually 1 to 10).



### 13.4.2 Magnetic Tapes:

Although magnetic tapes are relatively permanent, and can hold large volumes of data, they are slow in comparison to magnetic and optical disks. Even more important, magnetic tapes are limited to sequential access. Thus, they cannot provide random access for secondary-storage requirements, although historically, prior to the use of magnetic disks, tapes were used as a secondary-storage medium.

Tapes are used mainly for backup, for storage of infrequently used information, and as an off-line medium for transferring information from one system to another. Tapes are also used for storing large volumes of data, such as video or image data that either, do not need to be accessible quickly or are so voluminous that magnetic-disk storage would be too expensive.

A tape is kept in a spool, and is wound or rewound past a read/write head. Moving to the correct spot on a tape can take seconds or even minutes, rather than milliseconds; once positioned, however, tape drives can write data at densities and speeds those of disk drives. Capacities vary, depending on the length and width of the tape and on the density at which the head can read and write. The market is currently fragmented among a wide variety of tape formats. Currently available tape capacities range from a few gigabytes with the **Digital Audio Tape (DAT)** format, 10 to 40Gb with the **Digital Linear Tape (DLT)** format, 100 gigabytes and higher with the Ultrium format, to 330 Gb (with **Ampex helical scan** tape formats). Data-transfer rates are of the order of a few to tens of Mbps.

## 3.5 Storage Access

A database is mapped into a number of different files that are maintained by the underlying operating system. These files reside permanently on disks, with backups on tapes. Each file is partitioned into fixed-length storage units called blocks, which are the units of both storage allocation and data transfer.

A block may contain several data items. The form of physical data organization being used determines the exact set of data items that a block contains. We shall assume that no data item spans two or more blocks. This assumption is realistic for most data processing applications.

A major goal of the database system is to minimize the number of block transfers between the disk and memory. One way to reduce the number of disk accesses is to keep as many blocks as possible in main memory. The goal is to maximize the chance that, when a block is accessed, it is already in main memory, and thus, no disk access is required. The subsystem responsible for the allocation of buffer space is called the **buffer manager**.

### 13.5.1 Buffer Manager

If you are familiar with operating-system concepts, you will note that the buffer manager appears to be nothing more than a virtual-memory manager, like those found in most operating systems. One difference is that the size of the database may be much more than the hardware address space of machine, so memory addresses are not sufficient to address all disk blocks. Further, to serve the database system well, the buffer manager must use techniques more sophisticated than typical virtual-memory management schemes:

**Buffer Replacement Strategy:** When there is no room left in the buffer, a block must be removed from the buffer before a new one can be read in. Most operating systems use **Least Recently Used (LRU)** scheme, in which the block that was referenced least recently is written back to disk and is removed from the buffer. This simple approach can be improved on for database applications.

**Pinned Blocks:** For the database system to be able to recover from crashes, it is necessary to restrict those times when a block may be written back to disk. For instance, most recovery systems require that a block should not be written to disk while an update on the block is in progress. A block that is not allowed to be written back to disk is said to be pinned. Although many operating systems do not support pinned blocks, such a feature is essential for a database system that is resilient to crashes.

**Forced output of blocks:** There are situations in which it is necessary to write back the block to disk, even though the buffer space that it occupies is not needed. This write is called the **forced output** of a block.

### 13.5.2 Buffer-Replacement Policies

The goal of a replacement strategy for blocks in the buffer is to minimize accesses to the disk. For general-purpose programs, it is not possible to predict accurately which blocks will be referenced. Therefore, operating systems use the past pattern of block references as a predictor of future references. The assumption generally made is that blocks that have been referenced recently are likely to be referenced again. Therefore, if a block must be replaced, the least recently referenced block is replaced. This approach is called the **Least Recently Used (LRU)** block-replacement scheme.

LRU is an acceptable replacement scheme in operating systems. However, a database system is able to predict the pattern of future references more accurately than an operating system. A user request to the database system involves several steps. The database system is often able to determine in advance which blocks will be needed by looking at each of the steps required to perform the user-requested operation. Thus unlike operating systems, which must rely on the past to predict the future, database systems may have information regarding at least the short-term future.

To illustrate how information about future block access allows us to improve the **LRU** strategy, consider the processing of the relational-algebra expression

Borrower X customer

Assume that the pseudocode program shown in **Figure 5** gives the strategy chosen to process this request.

**For each** tuple b of borrower **do**

**For each** tuple c of customer **do**

**If** b[customer\_name]=c[customer\_name]

**Then begin**

Let x be a tuple defined as follows:

x[customer\_name]:=b[customer\_name]

x[loan\_number]:=b[loan\_number]

x[customer\_street]:=c[customer\_street]

$x[customer\_city]:=c[customer\_city]$

Include tuple  $x$  as part of result of borrower X customer

**End**

**End**

**End**

**Figure:5 Procedure for computing join.**

Assume that the two relations of this example are stored in separate files. In this example, we can see that, once a tuple of borrower has been processed, that tuple is not needed again. Therefore, once processing of an entire block of borrower tuples is completed, that block is no longer needed in main memory, even though it has been used recently. The buffer manager should be instructed to free the space occupied by a borrower block as soon as the final tuple has been processed. This buffer-management strategy is called the **toss-immediate** strategy.

Now consider blocks containing *customer* tuples. We need to examine every block of customer tuples once for each tuple of the *borrower* relation. When processing of a customer block is completed, we know, that block will not be accessed again until all other *customer* blocks have been processed. Thus, the most recently used *customer* block will be the final block to be re-referenced, and the least recently used *customer* block is the block that will be referenced next. This assumption set is the exact opposite of the one that forms the basis for the LRU strategy. Indeed, the optimal strategy for block replacement for the above procedure is the **Most Recently Used (MRU)** strategy. If a customer block must be removed from the buffer, the MRU strategy chooses the most recently used block (blocks are not eligible for replacement while they are being used).

## 13.6 SUMMARY

Several types of data storage exist in most computer systems. They are classified by speed with which they can access data, by either cost per unit of data to buy the memory, and by their reliability. Among the media available are cache, main memory, flash memory, magnetic disks, optical disks, and magnetic tapes.

Two factors determine the reliability of storage media: whether a power failure or system crash causes data to be lost, and what the likelihood is of physical failure of the storage device.

We can reduce the likelihood of physical failure by retaining multiple copies of data. For disks, we can use mirroring. Or we can use more sophisticated methods based on redundant arrays of independent disks (RAID).

By striping data across disks, these methods offer high throughput rates on large accesses; by introducing redundancy across disks, they improve reliability greatly. Several different RAID organizations are possible, each with different cost, performance and reliability characteristics. RAID level 1 and RAID level 5 are the most commonly used.

## 13.7 Technical Terms

**Cache Memory:** High-speed memory closely attached to a CPU, containing a copy of the most recently used memory data. When the CPU's request for instructions or data can be satisfied from the cache, the CPU can run at full rated speed. In a multiprocessor or when DMA is allowed, a bus-watching cache is needed.

**Main Memory:** Refers to physical memory that is internal to the computer. The word *main* is used to distinguish it from external mass storage devices such as disk drives. Another term for main memory is RAM.

**Flash Memory:** Flash memory is a non-volatile memory device that retains its data after the power is removed.

**Optical Storage:** the generic name given to a series of optical disks of which CD ROMs, CD-R i.e. CD-Recordable drive which has read and write capacity. Using this device about 650Mb of data can be written in about 15 minutes. Standard CD-R disks can only be written to once (WORM ... Write Once, Read Many) but there is a type of disk called CD-RW. With suitable drives these disks can be written, erased and rewritten. DVD (Digital Versatile Disks) are also examples of Optical Disks

**Platter:** The actual disk inside of a disk drive. Its surface is coated with a magnetic material that records data. Both sides of the platter are used, and a typical disk drive has several platters, stacked like pancakes

**Seek Time:** The length of time required to move a disk drive's read/write head to a particular location on the disk. The major part of a hard disk's access time is actually seek time.

**RAID:** Redundant Array of Independent (or inexpensive) Disks; a collection of storage disks with a controller (or controllers) to manage the storage of data on the disks.

**Access Time:** The amount of time it takes a computer to locate an area of memory for data storage or retrieval.

**Disk Controller:** The hardware that controls the writing and reading data to and from and to a disk drive. It can be used with floppy disks or hard drives. It can be hard-wired or built into a plug-in interface board.

**Checksums:** A checksum is a form of redundancy check, a very simple measure for protecting the integrity of data by detecting errors in data that is sent through space (telecommunications) or time (storage). It works by adding up the basic components of a message, typically the bytes, and storing the resulting value.

### 13.8 Model Questions

1. List the physical storage media available on the computers you use routinely. Give the speed with which data can be accessed on each media.
2. How does the remapping of bad sectors by disk controllers affect data retrieval rates?
3. Define RAID. Explain all the RAID levels in brief.
4. Give an example of a database application in which the reserved space method of representing variable length records is preferable to the pointer method.
5. Give an example of a database application in which the pointer method of representing variable length records is preferable to the reserved space method.

### 13.9 References

Database System Concepts", 4<sup>th</sup> edition by Abraham Silberschatz, Henry F.Korth and S. Sudarshan.

Fundamentals of database systems, 4<sup>th</sup> edition by R. Elmasri and B. Navathe.

**AUTHOR:**

**M.V.BHUJANGA RAO. M.C.A.,  
Lecturer,  
Dept. Of Computer Science,  
JKC College,  
GUNTUR**

## Lesson 14

# File Structure

### 14.0 Objectives:

After completion of this lesson the student will be able to know about:

- Structure our files to accommodate multiple lengths of records.
- How records are organized in files.
- Storage for object-oriented databases.

### Structure Of the Lesson:

- 14.1 File Organization
- 14.2 Organization of Records in Files
- 14.3 Data-Dictionary Storage
- 14.4 Storage for Object Oriented databases
- 14.5 Summary
- 14.6 Technical terms
- 14.7 Model questions
- 14.8 References

### 14.1 File Organization

A **file** is organized logically as a sequence of records. These records are mapped onto disk blocks. Files are provided as a basic construct in operating system, so we shall assume the existence of an underlying *file system*. We need to consider ways of representing logical data models in terms of files.

Although blocks are of fixed size determined by the physical properties of the disk and by the operating system, record sizes vary. In a relational database, tuples of distinct relations are generally of different sizes.

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

#### 14.1.1 Fixed-Length Records

As an example, let us consider a file of *account* records for our bank database. Each record of this file is defined (in pseudo code) as:

**Type** *deposit* = **record**

*Account\_number*: char(10);

*Branch\_name*: char(22);

*Balance*: real;

End

If we assume that each character occupied 1 byte and that numeric(12,2) occupies 8 bytes, our account record is 40 bytes long. A simple approach is to use the first 40 bytes for the first record, the next 40 bytes for the second record, and so on (**Fig.1**).

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

**Figure: 1** File containing *account* records.

However, there are two problems with this simple approach:

1. It is difficult to delete a record from this structure. The space occupied by the record to be deleted must be filled with some other record of the file, or we must have a way of marking deleted records so that they can be ignored.
2. Unless the block size happens to be a multiple of 40 (which is unlikely), some records will cross, block boundaries, i.e., part of the record will be stored in one block and remaining part in another. It would thus require two block accesses to read or write such a record.

When a record is deleted, we could move the records that came after it into the space formerly occupied by the deleted record, and so on, until every record following the deleted record has been moved ahead(**figure 2**).

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

**Figure: 2** File of Figure 1, with record 2 deleted and all records moved.

Such an approach requires moving a large number of records. It might be easier simply to move the final record of the file into the space occupied by the deleted record (**Fig.3**).

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

**Figure:3** File of Figure 1, with record 2 deleted and final record moved.

It is undesirable to move records to occupy the space freed by a deleted record, since doing so requires additional block accesses. Since insertions tend to be more frequent than deletions, it is acceptable to leave open the space occupied by the deleted record, and to wait for a subsequent insertion before reusing the space. A simple marker on a deleted record is not sufficient, since it is hard to find this available space when an insertion is being done. Thus, we need to introduce an additional structure.

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

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

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

**Fig.4** shows the file of **Fig.1**, with the free list, after records 1,4, and 6 have been deleted.



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

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

### 14.1.2 Variable-Length Records

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

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

Different techniques for implementing variable-length records exist. For purposes of illustration, we shall use one example to demonstrate the various implementation techniques. We shall consider a different representation of the *account* information stored in the file of Fig.11.6, in which we use one variable length record for each branch name and for all the account information for that branch. The format of the record is:

**Type** account-list =**record**

*Branch-name*: char(22);

*Account-info*: **array** [1..∞] **of record**;

*Account-number* :char(10);

*Balance*: real;

**End**

**End**

We define *account-info* as an array with an arbitrary number of elements, i.e., the type definition does not limit the number of elements in the array.

#### 14.1.2.1 Byte-string Representation

A simple method for implementing variable-length records is to attach a special end-of-record ( $\perp$ ) symbol to the end of each record. We can then store each record as a string of consecutive bytes. **Fig.5** shows such an organization to represent the file of fixed-length records of **fig.1** as variable-length records

1.	Perryridge	A-102 400	A201 900	A218 700	$\perp$
2.	Round Hill	A-305 350	$\perp$		
3.	Mianus	A-215 700	$\perp$		
4.	Downtown	A-101 500	A110 600	$\perp$	
5.	Redwood	A-222 700	$\perp$		
6.	Brighton	A-217 750	$\perp$		

**Figure: 5** Byte-string representation of variable-length records.

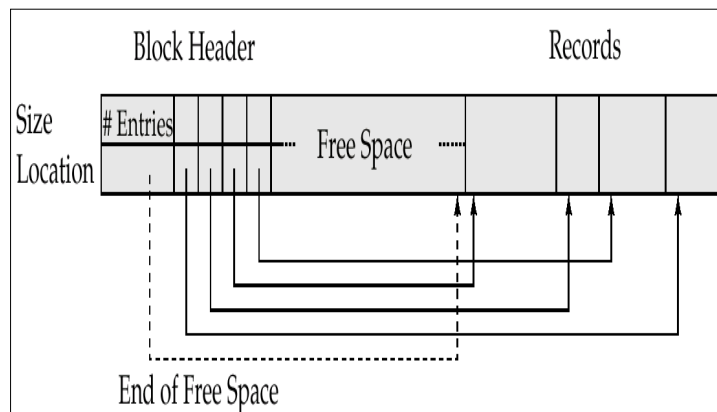
An alternative version of the byte-string representation stores the record length at the beginning of each record, instead of using end-of-record symbols. The byte-string representation as described in **Fig.5** has some disadvantages:

It is not easy to reuse space occupied formerly by a deleted record. Although techniques exist to manage insertion and deletion, they lead to a large number of small fragments of disk storage that are wasted.

There is no space, in general, for records to grow longer. If a variable-length record becomes longer, it must be moved—movement is costly if pointers to the record are stored elsewhere in the database (e.g., in indices, or in other records), since the pointers must be located and updated.

Thus, the basic byte-string representation described here not usually used for implementing variable-length records. However, a modified form of the byte-string representation, called the slotted-page structure is commonly used for organizing records within a single block.

**Fig.5** Byte-string representation of variable-length records. The **slotted-page structure** appears in **Fig.6**



**Figure: 6 Slotted-page structure.**

There is a header at the beginning of each block containing the following information:

1. The number of record entries in the header
2. The end of free space in the block
3. An array whose entries contain the location and size of each record

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

#### 14.1.2.2 Fixed-Length Representation

Another way to implement variable-length records efficiently in a file system is to use one or more fixed-length records to represent one variable-length record. There are two ways of doing this:

1. Reserved space. If there is a maximum record length that is never exceeded, we can use fixed-length records of that length. Unused space (for records shorter than the maximum space) is filled with a special null, or end-of-record, symbol.
2. List representation. We can represent variable-length records by list of fixed-length records, chained together by pointers.

If we choose to apply the reserved-space method to our account example, we need to select a maximum record length. **Fig.7** shows how the file of **Fig.5** would be represented if we allowed a maximum of three accounts per branch.

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

**Figure: 7** File of **Fig.5**, using the reserved-space method.

A record in this file is of the account-list type, but with the array containing exactly three elements. Those branches with fewer than three accounts (for example, Round Hill) have records with null fields. We use the symbol  $\perp$  to represent this situation in **Fig.7**. In practice, a particular value that can never represent real data is used (for example, an account number that is blank, or a name beginning with “\*”).

The reserved-space method is useful when most records have a length close to the maximum. Otherwise, a significant amount of space may be wasted. In our bank example, some branches may have many more accounts than others. This situation leads us to consider the linked list method. To represent the file by the linked list method, we add a pointer field as we did in **Fig.4**. The resulting structure appears in **Fig.8**.

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

**Figure: 8** File of **Figure 5** using linked lists.

The file structures of **Figures 4** and **8** both use pointers; the difference is that, in **Fig.4**, we use pointers to chain together only deleted records, whereas in **Fig.8**, we chain together all records pertaining to the same branch.

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

A disadvantage to the structure of **Fig.8** is that we waste space in all records except the first in a chain. The first record needs to have the branch-name value, but subsequent records do not. Nevertheless, we need to include a field for branch-name in all records, lest the records not be of fixed length. This wasted space is significant, since we expect, in practice, that each branch has a large number of accounts. To deal with this problem we allow two kinds of blocks in our file:

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

Thus, all records within a block have the same length, even though not all records in the file have the same length. **Fig.9** shows this file structure.



**Figure: 9 Anchor-block and overflow-block structures**

## 14.2 Organization of Records in Files

So far, we have studied how records are represented in a file structure. A relation is a set of records. Given a set of records, the next question is how to organize them in a file. Several of the possible ways of organizing records in files are:

**Heap file organization:** Any record can be placed anywhere in the file where there is space for the record. There is no ordering of records. Typically, there is a single file for each relation.

**Sequential file organization:** Records are stored in sequential order, according to the value of a “search key” of each record.

**Hashing file organization:** A hash function is computed on some attribute of each record. The result of the hash function specifies in which block of the file the record should be placed.

### 14.2.1 Sequential File Organization

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

**Fig.10** shows a sequential file of account records taken from our banking example.

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

**Figure: 10 Sequential file for account records.**

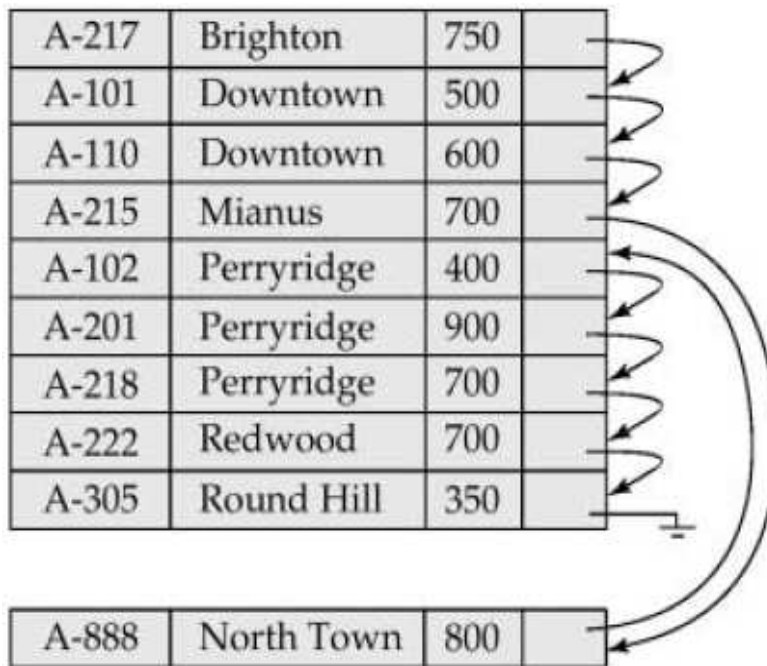
In that example, the records are stored in search-key order, using *branch\_name* as the search key.

The sequential file organization allows records to be read in sorted order; that can be useful for display purposes, as well as for certain query-processing algorithms.

It is difficult, however, to maintain physical sequential order as records are inserted and deleted, since it is costly to move many records as a result of a single insertion or deletion. We can manage deletion by using pointer chains, as we saw previously. For insertion, we apply the following rules:

1. Locate the record in the file that comes before the record to be inserted in search-key order.
2. If there is a free record (that is, space left after a deletion) within the same block as this record, insert the new record there. Otherwise, insert the new record in an overflow block. In either case, adjust the pointers so as to chain together the records in search-key order.

**Fig.11** shows the file of **Fig.10** after the insertion of the record (north Town, A-888, 800).



The structure in **Fig.11** allows fast insertion of new records, but forces sequential file-processing applications to process records in an order that does not match the physical order of the records.

#### 14.2.2 Clustering File Organization

Many relational database systems store each relation in a separate file, so that they can take full advantage of the file system that the operating system provides. Usually, tuples of a relation can be represented as fixed-length records. Thus, relations can be mapped to a simple file structure. This simple implementation of a relational database system is well suited to low-cost database implementations, for example, embedded systems or portable devices. In such systems, the size of the database is small, so little is gained from a sophisticated file structure. Furthermore, in such environments, it is essential that the overall size of the object code for the database system be small. A simple file structure reduces the amount of code needed to implement the system.

This simple approach to relational database implementation becomes less satisfactory as the size of the database increases. We have seen that there are performance advantages to be gained from careful assignment of records to blocks, and from careful organization of the blocks themselves. Clearly, a more complicated file structure may be beneficial, even if we retain the strategy of storing each relation in a separate file.

However, many large-scale database systems do not rely directly on the underlying operating system for file management. Instead, one large operating-system file is allocated to the database system. The database system stores all relations in this one file, and manages the file itself. To see the advantage of storing many relations in one file, consider the following SQL query for the bank database:

```
Select account-number, customer-name, customer-street, from depositor, customer
where depositor.customer_name=
customer.customer_name
```

This query computes a join of the *depositor* and *customer* relations. Thus, for each tuple of *depositor*, the system must locate the *customer* tuples with the same value for *customer\_name*. Ideally, these records will be located with the help of indices. However, they need to be transferred from disk into main memory. In the worst case, each record will reside on a different block, forcing us to do one block read for each record required by the query

Customer_name	Account_number
Hayes	A-102
Hayes	A-220
Hayes	A-503
Turner	A-305

**Figure: 12** The *depositor* relation.

Customer_name	Customer_street	Customer_city
Hayes	Main	Brooklyn
Turner	Putnam	Stamford

**Figure: 13** The *customer* relation.

As a concrete example, consider the *depositor* and *customer* relations of **Fig.12** and **13**, respectively. In **Fig.14**, we show a file structure designed for efficient executing of queries involving *depositor* X *customer*.

Hayes	Main	Brooklyn
Hayes	A-102	
Hayes	A-220	
Hayes	A-503	
Turner	Putnam	Stamford
Turner	A-305	

**Figure: 14** Clustering file structure.

The *depositor* tuples for each *customer\_name* are stored near the *customer* tuple for the corresponding *customer\_name*. This structure mixes together tuples of two relations, but allows for efficient processing of the join. When a tuple of the *customer* relation is read, the entire block containing that tuple is copied from disk into main memory. Since the corresponding *depositor* tuples are stored on the disk near the *customer* tuple, the block containing the *customer* tuple

contains tuples of the *depositor* relation needed to process the query. If a *customer* has so many accounts that the *depositor* records do not fit in one block, the remaining records appear on nearby blocks.

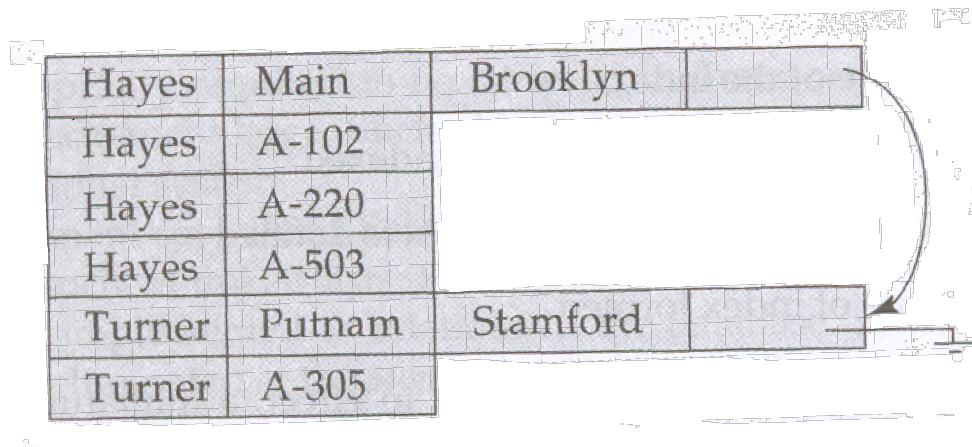
A **clustering file organization** is a file organization, such as that illustrated in **fig.14**, which stores related records of two or more relations in each block. Such a file organization allows us to read records that would satisfy the join condition by using one block read. Thus, we are able to process this particular query more efficiently.

Our use of clustering of multiple tables into a single file has enhanced processing of a particular join (*depositor* X *customer*), but it results in slowing processing of other types of query. For example,

**Select \***

**from** *customer*

Requires more block accesses than it did in the scheme under which we stored each relation in a separate file. Instead of several *customer* records appearing in one block, each record is located in a distinct block. Indeed, simply finding all the *customer* records is not possible without some additional structure. To locate all tuples of the *customer* relation in the structure of **fig.14**, we need to chain together all the records of that relation using pointers, as in **fig.15**



**Figure: 15 Clustering file structure with pointer chains.**

### 14.3 Data-Dictionary Storage

So far, we have considered only the representation of the relations themselves. A relational database system needs to maintain data about the relations, such as the schema of the relations. This information is called the data dictionary or system catalog. Among the types of information that the system must store are these:

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

In addition, many systems keep the following data on users of the system:

- Names of authorized users



- Authorization and accounting information about users
- Passwords or other information used to authenticate users

Further, the database may store statistical and descriptive data about the relations, such as:

- Number of tuples in each relation
- Method of storage for each relation (for example, clustered or nonclustered)

The data dictionary may also note the storage organization (sequential, hash, or heap) of relations, and the location where each relation is stored:

- If relations are stored in operating system files, the dictionary would note the names of the file (or files) containing each relation.
- If the database stores all relations in a single file, the dictionary may note the blocks containing records of each relation in a data structure such as a linked list.

## 14.4 Storage for Object-Oriented

The file-organization techniques such as the heap, sequential, hashing and clustering organizations can also be used for storing objects in an object-oriented database. However, some extra features are needed to support object-oriented database features, such as set-valued fields and persistent pointers.

### 14.4.1 Mapping of Objects to Files

The mapping of objects to files is in many ways like the mapping of tuples to files in a relational system. At the lowest level of data representation, both tuples and the data parts of objects are simply sequences of bytes. We can therefore store object data in the file structures described in this chapter, with some modifications, which we note next.

Objects in object-oriented databases may lack the uniformity of tuples in relational databases. For example, fields of records may be sets; in relational databases, in contrast, data are typically required to be (at least) in first normal form. Furthermore, objects may be extremely large. Such objects have to be managed differently from records in a relational system.

We can implement set-valued fields that have a small number of elements using data structures such as linked lists. Set-valued fields that have a larger number of elements can be implemented as relations in the database. Set-valued fields of objects can also be eliminated at the storage level by normalization: A relation is created containing one tuple for each value of a set-valued field of an object. Each tuple also contains the object identifier of the object. However, this relation is not made visible to the upper levels of the database system. The storage system gives the upper levels of the database system the view of a set-valued field, even though the set-valued field has actually been normalized by creating a new relation.

### 14.4.2 Implementation of Object Identifiers

Since objects are identified by object identifiers (OIDs), an object-storage system needs a mechanism to locate an object, given an OID. If the OIDs are **logical OIDs**, i.e., they do not specify the location of the object—then the storage system must maintain an index that maps OIDs to the actual location of the object. If the OIDs are **physical OIDs**, i.e., they encode the location of the object—then the object can be found directly. Physical OIDs typically have the following three parts:

1. A volume or file identifier
2. A block identifier within the volume or file
3. An offset within the block

A volume is a logical unit of storage that usually corresponds to a disk.

In addition, physical OIDs may contain a **unique identifier**, which is an integer that distinguishes the OID from the identifiers of other objects that happened to be stored at the same location earlier, and were deleted or moved elsewhere. The unique identifier is also stored with the object, and the identifiers in an OID and the corresponding object should match. If the unique identifier in a physical OID does not match the unique identifier in the object to which that OID points, the system detects that the pointer is a **dangling pointer**, and signals an error. (A **dangling pointer** is a pointer that does not point to a valid object.) **Fig.16** illustrates this scheme.

Physical Object Identifier

Volume. Block. Offset	Unique-Id
Object	

Unique-Id	Date
-----------	------

#### a) General structure

Location	Unique-Id	Date
519.56850.1200	51	-----
Good OID	519.56850.1200	51
Bad OID	519.56850.1200	50

#### (b) example of use

**Figure: 16 Unique identifiers in an OID**

Such pointer errors occur when physical OIDs corresponding to old objects that have been deleted are used accidentally. If the space occupied by the object had been reallocated, there may be a new object in the location, and it may get incorrectly addressed by the identifier of the old object. If a dangling pointer is not detected, it could cause corruption of a new object stored at the same location. The unique identifier helps to detect such errors, since the unique identifiers of the old physical OID and the new object will not match.

Suppose that an object has to be moved to a new block, perhaps because the size of the object has increased, and the old block has no extra space. Then, the physical OID will point to the old block, which no longer contains the object. Rather than change the OID of the object (which involves changing every object that points to this one), we leave the object, it finds the forwarding address instead of the object; it then uses the forwarding address to locate the object.

### 14.4.3 Management of Persistent Pointers

We implement persistent pointers in a persistent programming language by using OIDs. In some implementations, persistent pointers are physical OIDs; in others, they are logical OIDs. An important difference between persistent pointers and in-memory pointers is the size of the pointer. In-memory pointers need to be only big enough to address all virtual memory. On most current computers, in-memory pointers are usually 4 bytes long, which is sufficient to address 4 gigabytes of memory. The most recent computer architectures have pointers that are 8 bytes long.

Persistent pointers need to address all the data in a database. Since database systems are often bigger than 4 gigabytes, persistent pointers are usually at least 8 bytes long. Many object-oriented databases also provide unique identifiers in persistent pointers, to catch dangling references.

The action of looking up an object, given its identifier, is called **dereferencing**. Given an in-memory pointer (as in C++), looking up the object is merely a memory reference. Given a persistent pointer, dereferencing an object has an extra step—finding the actual location of the object in memory by looking up the persistent pointer in a table. If the object is not already in memory, it has to be loaded from disk. We can implement the table lookup fairly efficiently by using a hash table data structure, but the lookup is still slow compared to a pointer dereference, even if the object is already in memory.

#### 14.4.4 Hardware Swizzling

A simple way to merge persistent and in-memory pointer types is just to extend the length of in-memory pointers to the same size as persistent pointers, and to use 1 bit of the identifier to distinguish between persistent and in-memory pointers. However the storage cost of longer persistent pointers will have to be borne by in-memory pointers as well; understandably, this scheme is unpopular.

We shall describe a technique called **hardware swizzling**, which uses the virtual-memory-management hardware present in most current computer systems to address this problem. When data in a virtual memory page are accessed, and the operating system detects that the page does not have real storage allocated for it, or has been access protected, then a **segmentation violation** is said to occur. Many operating systems provide a mechanism to specify a function to be called when a segmentation violation occurs, and a mechanism to allocate storage for a page in virtual address space, and to set that page's access permissions.

Hardware swizzling has two major advantages over software swizzling:

1. It is able to store persistent pointers in objects in the same amount of space as memory pointers require (along with extra storage external to the object).
2. It transparently converts between persistent pointers and in-memory pointers in a clever and efficient way. Software written to deal with in-memory pointers can thereby deal with persistent pointers as well, without any changes.

##### 14.4.4.1 Pointer Representation

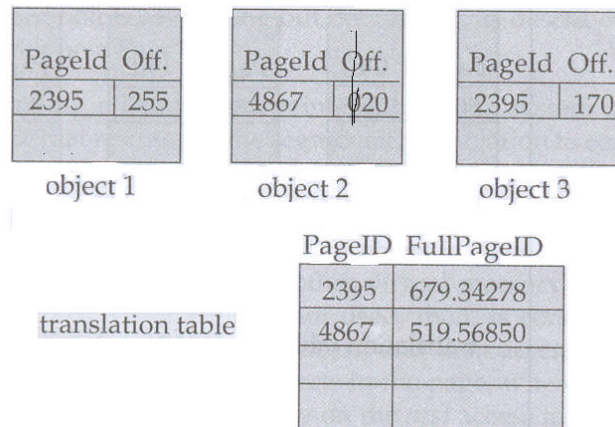
Hardware swizzling uses the following representation of persistent pointers contained in objects that are on disk. A persistent pointer is conceptually split into two parts: a page identifier in the database, and an offset within the page.

The page identifier in a persistent pointer is actually a small indirect pointer, which we call the short page identifier. Each page (or other unit of storage) has a translation table that provides a mapping from the short page identifiers to full database page identifiers. The system has to look up the short page identifier in a persistent pointer, in the translation table to find the full-page identifier.

The translation table, in the worst case, will be only as big as the maximum number of pointers that can be contained in objects in a page; with a page size of 4096, and a pointer size of 4 bytes, the maximum number of pointers is 1024. In practice, the translation table is likely to contain much less than the maximum number of elements (1024 in our example) and will not consume excessive space. The short page identifier needs to have only enough bits to identify a row in the table; with a maximum table size of 1024, only 10 bits are required. Hence, a small number of bits are enough to store the short page identifier. Thus, the translation table permits an entire persistent pointer to fit into the same space as an in-memory pointer. Even though only a few bits are needed for the short

page identifier, all the bits of an in-memory pointer other than the page-offset bits are used as the short page identifier. This architecture facilitates swizzling, as we shall see.

The persistent-pointer representation scheme appears in **Figure 17**, where there are three objects in the page, each containing a persistent pointer.



**Figure: 17 Page image before swizzling.**

The translation table gives the mapping between short page identifiers and the full database page identifiers for each of the short page identifiers in these persistent pointers. The database page identifiers are shown in the format volume.page.offset.

Each page maintains extra information so that all persistent pointers in the page can be found. The system updates the information when an object is created or deleted in the page. The need to locate all the persistent pointers in a page will become clear later.

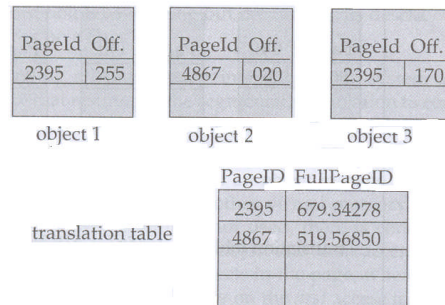
#### 14.4.4.2 Swizzling Pointers on a Page

Initially no page of the database has been allocated a page in virtual memory. Virtual memory pages may be allocated to database pages even before they are actually loaded, as we will see shortly. Database pages get loaded into virtual memory when the database system needs to access data on the page. Before a database page is loaded, the system allocates a virtual memory page to the database page if one has not already been allocated. The system then loads the database page into the virtual-memory page it has allocated to it.

When the system loads a database page  $P$  into virtual memory, it does pointer swizzling on the page: it locates all persistent pointers contained in objects in page  $P$ , using the extra information stored in the page. It takes the following actions for each persistent pointer in the page. (Let the value of the persistent pointer be  $\langle p_i, o_i \rangle$ , where  $p_i$  is the short page identifier and  $o_i$  is the offset within the page. Let  $P_i$  be the full-page identifier of  $p_i$ , found in the translation table in page  $P$ .)

1. If page  $P$ , does not already have a virtual-memory page allocated to it, the system now allocates a free page in virtual memory to it. The page  $p_i$  will reside at this virtual-memory location if and when it is brought in. At this point, the page in virtual address space does not have any storage allocated for it, either in memory or on disk; it is merely a range of addresses reserved for the database page. The system allocates actual space when it actually loads the database page  $P_i$  into virtual memory.
2. Let the virtual-memory page allocated (either earlier or in the preceding step) for  $P_i$  be  $v_i$ . The system updates the persistent pointer being considered, whose value is  $\langle p_i, o_i \rangle$ , by replacing  $p_i$  with  $v_i$ .

**Fig.18** shows the state of the page from **Fig.17** after the system has brought that page into memory and swizzled the pointers in it.



**Figure: 18 Page image after swizzling.**

Here, we assume that the page whose database page identifier is 679.34278 has been mapped to page 5001 in memory, whereas the page whose identifier is 519.56850 has been mapped to page 4867 (which is the same as the short page identifier). All the pointers in objects have been updated to reflect the new mapping, and can now be used as in-memory pointers.

#### 14.4.4.3 Pointer Dereference

Consider the first time that an in-memory pointer to a virtual memory page  $v_i$  is dereferenced, when storage has not yet been allocated for the page. As we described, a segmentation violation will occur, and will result in a function call on the database system. The database system takes the following actions:

It first determines which database page was allocated to virtual memory page  $v_i$ ; let the full-page identifier of the database page be  $P_i$ . (If no database page has been allocated to  $v_i$ , the pointer is incorrect, and the system flags an error.)

It allocates storage space for page  $v_i$  and loads the database page  $P_i$  into virtual memory page  $v_i$ .

It carries out pointer swizzling out on page  $P_i$ , as described earlier in "Swizzling Pointer on a Page".

After swizzling all persistent pointers in  $P_i$ , the system allows the pointer dereference that resulted in the segmentation violation to continue. The pointer dereference will find the object for which it was looking loaded in memory.

#### 14.4.4.4 Optimizations

Several optimizations can be carried out on the basic scheme described here. When the system swizzles page  $P$ , for each page  $P'$  referred to by any persistent pointer in  $P$ , it attempts to allocate  $P'$  to the virtual address location indicated by the short page identifier of  $P'$  on page  $P$ . If the system can allocate the page in this attempt, pointers to it do not need to be updated.

#### 14.4.5 Large Objects

Large objects containing binary data are called binary large objects (blobs), while large objects containing character data, are called character large objects (clobs). Most relational databases restrict the size of a record to be no longer than the size of a page, to simplify buffer management and free-space management. Large objects and long fields are often stored in a special file reserved for long-field storage. For practical reasons, we may manipulate large objects by using application programs, instead of doing so within the database:

**Text data:** Text is usually treated as a byte string manipulated by editors and formatters.

**Image/graphical data:** Graphical data may be represented as a bitmap or as a set of lines, boxes, and other geometric objects. Although some graphical data often are managed within the database system itself, special application software is used for many cases, such as integrated circuit design.

**Audio and video data:** Audio and video data are typically a digitized, compressed representation created and display by separate application software. Data are usually modified with special purpose editing software, outside the database system.

## 14.5 Summary

We can organize a file logically as a sequence of records mapped on to disk blocks. One approach to mapping the database to files is to use several files, and to store records of only one fixed length in any given file. An alternative is to structure files so that they can accommodate multiple lengths for records. There are different techniques for implementing variable length records, including the slotted page method, the pointer method, and the reserved space method.

Since data are transferred between disk storage and main memory in units of a block, it is worthwhile to assign file records to blocks in such a way that a single block contains related records. If we can access several of the records we want with only one block access, we can save disk accesses. Since disk accesses are usually the bottleneck in the performance of a database system, careful assignment of records to blocks can pay significant performance dividends.

One way to reduce the number of disk accesses is to keep as many blocks as possible in main memory. Since it is not possible to keep all blocks in main memory, we need to manage the collection of the space available in main memory for the storage of blocks. The duffer is that part of main memory available for storage of copies of disk blocks. The subsystem responsible for the allocation of buffer space is called the buffer manager.

Storage systems for object-oriented database are somewhat different from storage systems for relational databases. They must deal with large objects, for example, and must support persistent pointers. There are schemes to detect dangling persistent pointers.

Software and hardware base swizzling schemes permit efficient dereferencing of persistent pointers. The hardware based schemes use the virtual memory management support implemented in hardware, and made accessible to user programs by many current generation operating systems.

## 14.6 Technical Terms

**File organization:** A technique for physically arranging the records of a file.

**Sequential file organization:** The rows in the file are stored in sequence according to a primary key value.

**Hashed file Organization:** The address for each row is determined using an algorithm.

**Free list:** A linked list of blocks in a table, cluster, or index that will be examined when space is required for new or relocated data.

**Search key:** A piece of data that the File Manager uses when searching through a B\*-tree to locate the information it needs.

**Data Dictionary:** A file that defines the basic organization of a database. It will contain a list of all files in the database, the number of records in each file, and the names and types of each field.

**Object identifier:** (OID) A unique specially formatted number that is composed of a most significant part assigned by an internationally recognized standards organization to a specific owner and a least significant part assigned by the owner of the most significant part. For example, the unique alphanumeric/numeric identifier registered under the ISO registration standard to reference a specific object or object class.

**Unique identifier:** Data will appear only in a single record in the table; no two fields will be assigned the same data (Primary Key).

**Dangling Pointer:** A pointer referring to an area of memory that has been deallocated. Dereferencing such a pointer usually produces garbage.

**Page fault:** A page fault is an exception, which is raised by the memory management unit when a needed page is not mapped in physical memory. This exception is passed on to the operating system, which will bring the required page into physical memory

## 14.7 Model Questions

1. Explain why the allocation of records to blocks affects database system performance significantly.
2. In the sequential file organization, why is an overflow block used even if there is, at the moment, only one overflow record?
3. Write a short notes on clustering file organization.
4. Briefly explain data-dictionary storage.
5. Define object identifier (OID). How to implement the object identifiers?

## 14.8 References

Database System Concepts”, 4<sup>th</sup> edition by Abraham Silberschatz, Henry F.Korth and S.Sudarshan

Fundamentals of database systems, 4<sup>th</sup> edition by R. Elmasri and B. Navathe

### AUTHOR:

**M.V.BHUVANGA RAO. M.C.A.,**  
**Lecturer,**  
**Dept. Of Computer Science,**  
**JKC College,**  
**GUNTUR**

## Lesson 15

# Indexing

### 15.0 Objectives:

After completion of this chapter the student will be able to know about:

- Index-sequential file organization.
- Algorithms for updating indices.
- How **B**<sup>+</sup>-tree is more advantageous than index-sequential file organization.
- Updates on **B**<sup>+</sup>-trees.

### Structure Of the Lesson:

#### 15.1 Basic Concepts

#### 15.2 Ordered Indices

##### 15.2.1 Primary Index

###### 15.2.1.1 Dense and Sparse Indices

###### 15.2.1.2 Multilevel Indices

###### 15.2.1.3 Index Update

##### 15.2.2 Secondary Indices

#### 15.3 B<sup>+</sup> -Tree Files

##### 15.3.1 Structure of a B<sup>+</sup> -Tree

##### 15.3.2 Updates on B<sup>+</sup> -Trees

##### 15.3.3 B<sup>+</sup> -Tree File Organization

#### 15.4 B<sup>+</sup> -Tree Index Files

#### 15.5 Summary

#### 15.6 Technical terms

#### 15.7 Model questions

#### 15.8 References



## 15.1 Basic Concepts

An index for a file in a database system works in much the same way as the index in the textbook. If you want to learn about a particular topic in the textbook, we can search for the topic in the index at the back of the book, find the pages where it occurs, and then read the pages to find the information we are looking for. The words in the index are in stored order, making it easy to find the word we are looking for. Moreover, the index is much smaller than the book, further reducing the effort needed to find the words we are looking for.

Database system indices play the same role as book indices in libraries. For example, to retrieve an *account* record given the account number, the database system would look up an index to find on which disk block the corresponding record resides, and then fetch the disk block, to get the *account* record.

Keeping a stored list of account numbers would not work well on very large databases with millions of accounts, since the index would itself be very big: further, even though keeping the index stored reduces the search time, finding an account can still be rather time-consuming. Instead, more sophisticated indexing techniques may be used. We shall discuss several of these techniques in this chapter.

There are two basic kinds of indices:

**Ordered indices:** Based on a sorted ordering of the values.

**Hash indices:** Based on a uniform distribution of values across a range of buckets. The bucket to which a value is assigned is determined by a function, called a hash function. We shall consider several techniques for both ordered indexing and hashing. No one technique is the best. Rather, each technique is best suited to particular database applications. Each technique must be evaluated on the basis of these factors:

**Access types:** The types of access that are supported efficiently. Access types can include finding records with a specified attribute value and finding records whose attribute values fall in a specified range.

**Access time:** The time taken to find a particular data item, or set of items, using the technique in question.

**Insertion time:** The time it takes to insert a new data item. This value includes the time it takes to find the correct place to insert the new data item, as well as the time it takes to update the index structure.

**Deletion time:** The time it takes to delete a data item. This value includes the time it takes to find the item to be deleted, as well as the time it takes to update the index structure.

**Space overhead:** The additional space occupied by an index structure. Provided that the amount of additional space is moderate, it is usually worthwhile to sacrifice the space to achieve improved performance.

We often want to have more than one index for a file. For example, we may wish to search for a book by author, by subject or by title. An attribute or set of attributes used to look up records in a file is called a search key. Note that this definition of *key* differs from that used in *primary key*, *candidate key* and *super key*. This duplicate meaning for **key** is (unfortunately) well established in practice. Using our notion of a search key, we see that if there are several indices on a file, there are several search keys.

## 15.2 Ordered Indices

To gain fast random access to records in a file, we can use an index structure. Each index structure is associated with a particular search key. Just like the index of a book or a library catalog, an ordered index stores the values of the search keys in sorted order, and associates with each search key the records that contain it.

The records in the indexed file may themselves be stored in some sorted order, just as books in a library are stored according to some attribute such as the Dewey decimal number. A file may have several indices, on different search keys. If the file containing the records is sequentially ordered, a **primary index** is an index whose Search key also defines the sequential order of the file. **Primary indices** are also called **clustering indices**. The search key of a primary index is usually the primary key, although that is not necessarily so. Indices whose search key specifies an order different from the sequential order of the file are called **secondary indices**, or **non-clustering indices**.

### 15.2.1 Primary Index

In sections, we assume that all files are ordered sequentially on some search key. Such files, with a clustering index on the search key, are called **index-sequential files**. They represent one of the oldest index schemes used in database systems. They are designed for applications that require both sequential processing of the entire file and random access to individual records.

**Fig.1** shows a sequential file of account records taken from our banking example

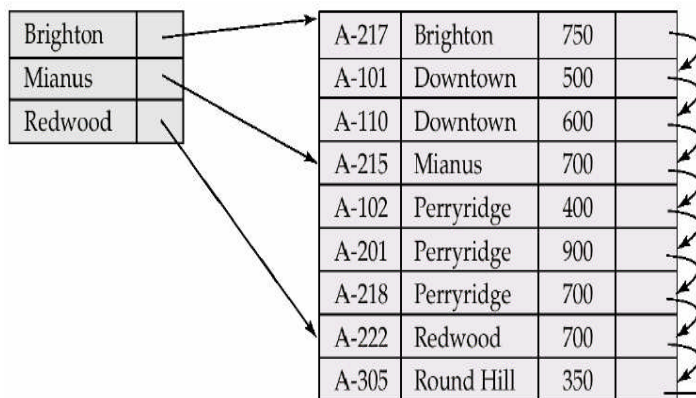


Fig.1 Sequential file of account records.

In the example of **Fig.1**, the records are stored in search-key order, with branch name used as the search key.

#### 15.2.1.1 Dense and Sparse Indices

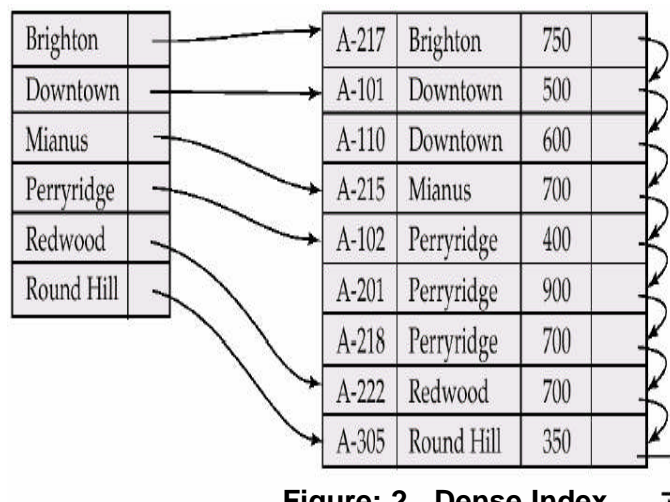
An **index record**, or **index entry** consists of a search-key value, and pointers to one or more records with that value as their search-key value. The pointer to a record consists of the identifier of a disk block and an offset within the disk block to identify the record within the block. There are *two types* of ordered indices that we can use:

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

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

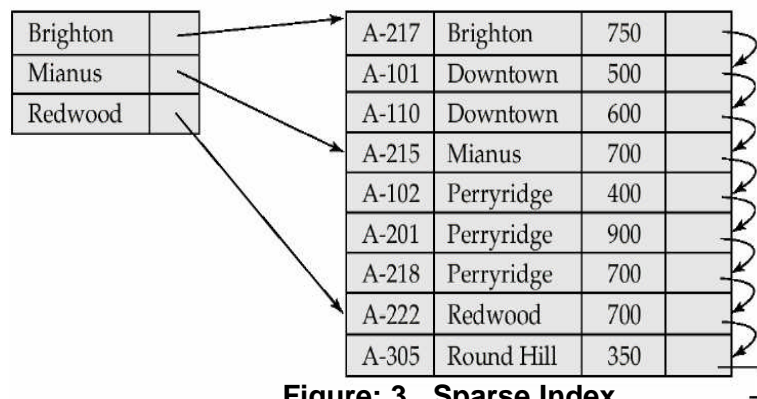
2. **Sparse Index:** An index record appears for only some of the search key values. As it is true in dense indices, each index record contains a search key value and a pointer to the first data record with that search-key value. To locate a record, we find the index entry with the largest search-key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

**Fig.2** and **Fig.3** show dense and sparse indices, respectively, for the account file. Suppose that we are looking up records for the Perryridge branch. Using the dense index of **Fig.2**, we follow the pointer directly to the first Perryridge record.



**Figure: 2 Dense Index.**

We process this record, and follow the pointer in that record to locate the next record in search-key (branch name) order. We continue processing records until we encounter a record for a branch other than Perryridge



**Figure: 3 Sparse Index.**

If we are using the sparse index (**Fig.3**), we do not find an index entry for “Perryridge”. Since the last entry (in alphabetical order) before “Perryridge” is “Mianus”, we follow that pointer. We then read the account file in sequential order until we find the first Perryridge record, and begin processing at that point.

As we have seen, it is generally faster to locate a record if we have a dense index rather than a sparse index. However, sparse indices have advantages over dense indices in that they require less space and they impose less maintenance overhead for insertions and deletions.

There is a trade-off that the system designer must make between access time and sparse overhead. Although the decision regarding this trade-off depends on the specific application, a good compromise is to have a sparse index with one index entry per block. The reason that this design is a good trade-off is that the dominant cost in processing a database request is the time that it takes to bring a block from disk into main memory. Once we have brought in the block, the time to scan the entire block is negligible. Using this sparse index, we locate the block containing the record that we are seeking.

### 15.2.1.2 Multilevel Indices

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

If an index is sufficiently small to be kept in main memory, the search time to find an entry is low. However, if the index is so large that it must be kept on disk, a search for an entry requires several disk-block reads. Binary search can be used on the index file to locate an entry, but the search still has a large cost. If the index occupies  $b$  blocks, binary search requires as many as  $\lceil \log_2(b) \rceil$  blocks to read ( $\lceil x \rceil$  denotes the least integer that is greater than or equal to  $x$ ; i.e., we round upward.) For our 100-block index, binary search requires seven blocks reads. On a disk system where a block

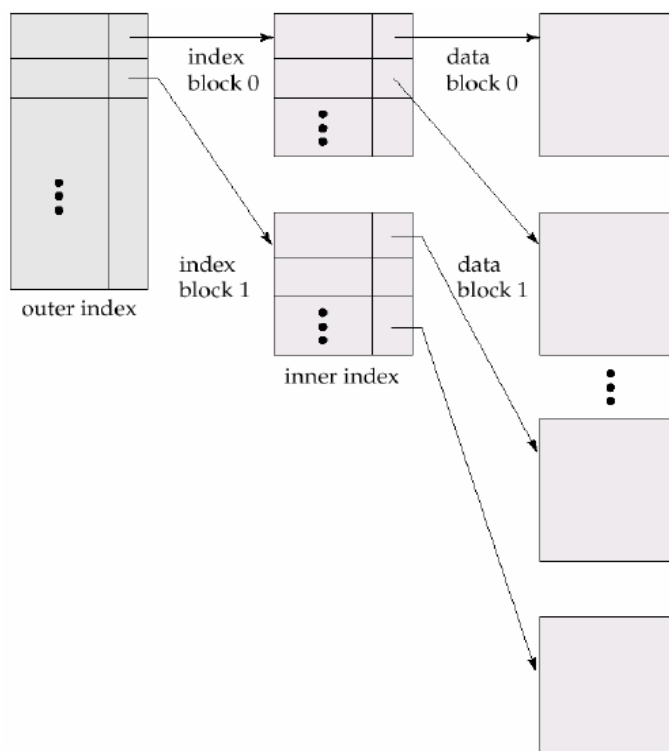


Figure: 4 Two-level sparse indexes.

read takes 30 milliseconds, the search will take 210 milliseconds, which is long. Note that, if overflow blocks have been used, binary search will not be possible. In that case, a sequential search is typically used, and that requires  $b$  block reads, which will take even longer. Thus, the process of searching a large index may be costly.

To deal with this problem, we treat the index just as we would treat any other sequential file, and construct a sparse index on the clustering index, as in

To locate a record, we first use binary search on the outer index to find the record for the largest search-key value less than or equal to the one that we desire. The pointer points to a block of the inner index. We scan this block until we find the record that has the largest search-key value less than or equal to the one that we desire. The pointer in this record points to the block of the file that contains the record for which we are looking.

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

A typical dictionary is an example of a multilevel index in the non-database world. The header of each page lists the first word alphabetically on that page. Such a book index is a multilevel index: the words are at the top of each page of the book index from a sparse index on the contents of dictionary pages.

Multilevel indices are closely related to tree structures, such as the binary trees used for in-memory indexing.

### 15.2.1.3 Index Update

Regardless of what form of index is used, every index must be updated, and a record is either inserted into or deleted from the file. We first describe algorithms for updating single level indices.

**Insertion.** First, the system performs a lookup using the search key value that appears in the record to be inserted. The action the system takes next depends on whether the index is **dense** or **sparse**:

#### Dense Indices:

1. If the search-key value does not appear in the index, the system inserts an index record with the search-key value in the index at the appropriate position.
2. Otherwise the following actions are taken:
  - a. If the index record stores pointers to all records with the same search-key value, the system adds a pointer to the new record to the index record.
  - b. Otherwise, the index record stores a pointer to only the first record with the search-key value. The system then places the record being inserted after the other records with the same search-key values.

**Sparse Indices:** We assume that the index stores an entry for each block. If the system creates a new block, it inserts the first search-key value (in search-key order) appearing in the new block into

the index. On the other hand, if the new record has the least search-key value in its block, the system updates the index entry pointing to the block; if not, the system makes no change to the index.

**Deletion:** To delete a record, the system first looks up the record to be deleted. The actions the system takes next depend on whether the index is **dense** or **sparse**:

### Dense Indices:

1. If the deleted record was the only record with its particular search-key value, then the system deletes the corresponding index record from the index.
2. Otherwise the following actions are taken:
  - a. If the index record stores pointers to all records with the same search-key value, the system deletes the pointer to the deleted record from the index record.
  - b. Otherwise, the index record stores a pointer to only the first record with the search-key value. In this case, if the deleted record was the first record with the search-key value, the system updates the record to point to the next record.

### Sparse Indices:

1. If the index does not contain an index record with the search-key value of the deleted record, nothing needs to be done to the index.
2. Otherwise the system takes the following actions:
  - a. If the deleted record was the only record with its search-key, the system replaces the corresponding index record with an index record for the next search-key value (in search-key order). If the next search-key value already has an index entry, the entry is deleted instead of being replaced.
  - b. Otherwise, if the index record for the search-key value points to the record being deleted, the system updates the index record to point to the next record with the same search-key value.

Insertion and deletion algorithms for multilevel indices are simple extension of the scheme just described. On deletion or insertion, the system updates the lowest-level index as described. As far as the second level is concerned, the lowest level index merely is a file containing records, thus, if there is any change in the lowest -level index, the system updates the second-level index as described. The same technique applies to further levels of the index, if there are any.

## 15.2.2 Secondary Indices

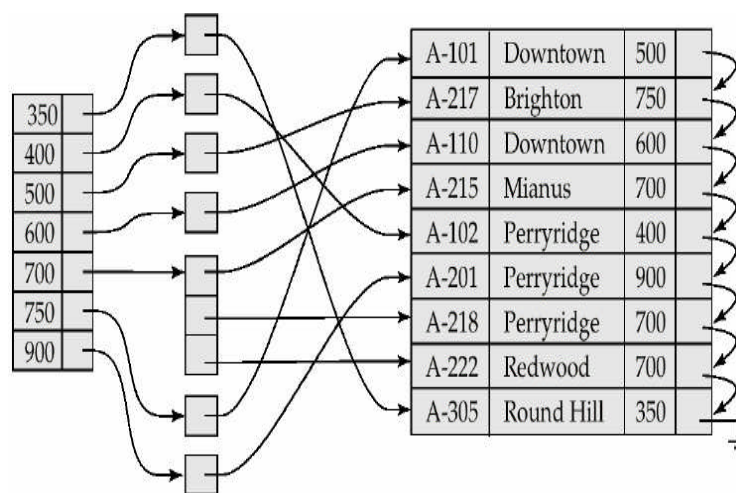
Secondary indices must be dense, with an index entry for every search-key value, and a pointer to every record in the file. A clustering index may be sparse, storing only some of the search-key values, since it is always possible to find records with intermediate search-key values by a sequential access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

A secondary index on a candidate key looks just like a dense clustering index, except that the records pointed to by successive values in the index are not stored sequentially. In general, however, secondary indices may have a different structure from clustering indices. If the search-key of the

clustering index is not a candidate key, it suffices if the index points to the first record with a particular value for the search key, since the other records can be fetched by a sequential scan of the file.

In contrast, if the search key of a secondary index is not a candidate key, it is not enough to point to just the first record with each search-key value. The remaining records with same search-key value could be anywhere in the file, since the records are ordered by the search key of the clustering index, rather than by the search key of the secondary index. Therefore, a secondary index must contain pointers to all the records.

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



**Figure: 5 Secondary index an *account* file, on Non-candidate key *balance*.**

**Fig.5** shows the structure of a secondary index that uses an extra level of indirection on the account file, on the search key *balance*.

A sequential scan in clustering index order is efficient because records in the file are stored physically in the same order as the index order. However, we cannot (except in rare special cases) store a file physically ordered by both the search key of the clustering index and the search key of a secondary index. Because secondary key order and physical key order differ, if we attempt to scan the file sequentially in secondary-key order, the reading of each record is likely to require the reading of a new block from disk, which is very slow.

The procedure described earlier for deletion and insertion can also be applied to secondary indices; the actions taken are those described for dense indices storing a pointer to every record in the file. If a file has multiple indices, whenever the file is modified, every index must be updated.

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

However, they impose a significant overhead on modification of the database. The designer of a database decides which secondary indices are desirable on the basis of an estimate of the relative frequency of queries and modifications.

### 15.3 B+ -Tree Index Files

The main disadvantage of the index-sequential file organization is that performance degrades as the file grows, both for index lookups and for sequential scans through the data.

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

#### 15.3.1 Structure of a B+ - tree structure

A B+ -tree index is a multilevel index, but it has a structure that differs from that of the multilevel index sequential file

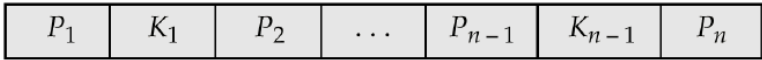


Figure: 6 Typical node of a B+ -tree.

Fig.6 shows a typical node of a B+ -tree. It contains up to  $n-1$  search key values  $k_1, k_2, \dots, k_{n-1}$ , and  $n$  pointers  $p_1, p_2, \dots, p_n$ . The search key values within a node are kept in sorted order; thus, if  $i < j$ , then  $k_i < k_j$ .

We consider first the structure of the leaf nodes. For  $i=1, 2, \dots, n-1$ , pointer  $p_i$  points to either a file record with search-key value  $k_i$  or bucket of pointers, each of which points to a file record with search key value  $k_i$ . The bucket structure is used only if the search key does not form a candidate key, and if the file is not sorted in the search key value order.

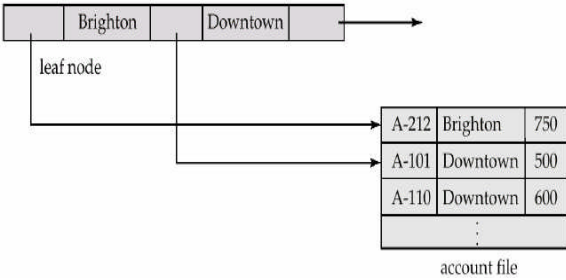


Figure: 7 A leaf node for account B+ - tree index(n=3).

Fig.7 shows one leaf node of a B+ - tree for the account file, in which we have chosen  $n$  to be 3, and the search key is branch-name. Note that, since the account file is ordered by branch-name, the pointers in the leaf node point directly to the file.

Now that we have seen the structure of a leaf node, let us consider how search key values are assigned to a particular node. Each leaf can hold up to  $n-1$  values. We allow leaf nodes to contain as few as  $\lceil (n-1)/2 \rceil$  values. The range of values in each leaf does not overlap. Thus, if  $L_i$  and  $L_j$  are leaf nodes and  $i < j$ , then every search -key value in  $L_i$  is less than every search-key value in  $L_j$ . If the B+ -tree index is to be a dense index, every search key value must appear in some leaf node.



Now, we can explain the tree of the pointer  $P_n$ . Since there is a linear order on the leaves based on the search key values that they contain, we use  $P_n$  to chain together the leaf nodes in search-key order. This ordering allows for efficient sequential processing of the file.

The non-leaf nodes of the  $B^+$ -tree form a multilevel (sparse) index on the leaf nodes. The structure of the non-leaf nodes is the same as that for leaf nodes, except that all pointers are pointers to tree nodes. A leaf node may hold upto  $n$  pointers, and must hold at least  $\lceil n/2 \rceil$  pointers. The number of pointers in a node is called the fan out of the node.

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

The root node can hold fewer than  $\lceil n/2 \rceil$  pointers; however, it must hold at least two pointers, unless the tree consists of only one node. It is always possible to construct a  $B^+$ -tree, for any  $n$ , that satisfies the proceeding requirements

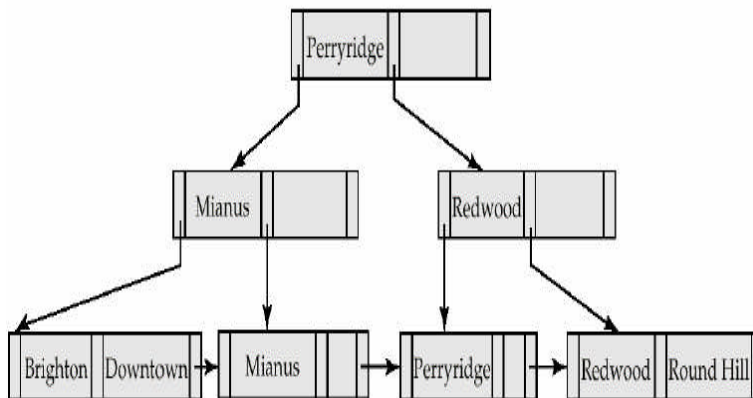


Figure: 8  $B^+$ -Tree for account file ( $n=3$ )

Fig.8 shows a complete  $B^+$ -tree for the *account* file( $n=3$ ). For simplicity, we have omitted both the pointers to the file itself and the null pointers. Fig.9 shows a  $B^+$ -tree for the *account* file with  $n=5$ .

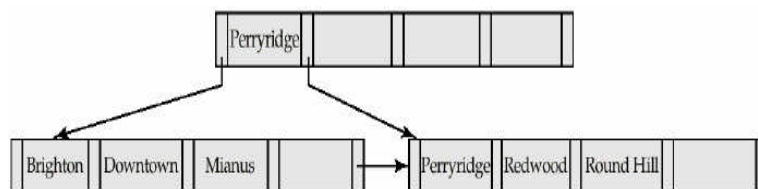


Figure: 9  $B^+$ -tree for *account* file with  $n=5$ .

### 15.3.2 Updates on $B^+$ -Trees

Insertion and deletion are more complicated, since it may be necessary to split a node that becomes too large as the result of an insertion, or to coalesce node (that is, combine nodes) if node becomes too small (fewer than  $\lceil n/2 \rceil$  pointers). Furthermore, when a node is split or a pair of nodes is combined, we must ensure that balance is preserved. To introduce the idea behind insertion and

deletion in a  $B^+$ -tree, we shall assume temporarily that nodes never become too large or too small. Under this assumption, insertion and deletion are performed as defined next.

- **Insertion-** First we find the leaf node in which the search-key value would appear. If the search-key value already appears in the leaf node, we add the new record to the file and, if necessary, add to the bucket a pointer to the record. If the search-key value does not appear, we insert the value in the leaf node and position it such that the search keys are still in order. We then insert the new record in the file and, if necessary, create a new bucket with the appropriate pointer.
- **Deletion-** First we find the record to be deleted, and remove it from the file. We remove the search-key value from the leaf node if there is no bucket associated with that search-key value or if the bucket becomes empty as a result of the deletion.

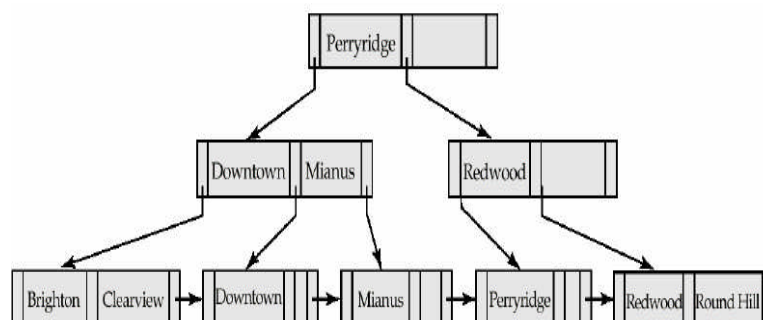
We now consider an example in which a node must be split. Assume that we wish to insert a record with a branch-name value of “Clearview” into the  $B^+$ -tree. We find that “Clearview” should appear in the containing “Brighton” and “Downtown”. There is no room to insert the search-key value “Clearview”. Therefore, the node is split into two nodes. **Fig.10** shows the two leaf nodes that result from inserting “Clearview” and splitting the node containing “Brighton” and “Downtown”.



**Figure: 10 Split of leaf node on insertion of “Clearview”.**

In general, we take the  $n$  search-key values (the  $n-1$  values in the leaf node plus the value being inserted), and put the first  $\lceil n/2 \rceil$  in the existing node and the remaining values in a new node.

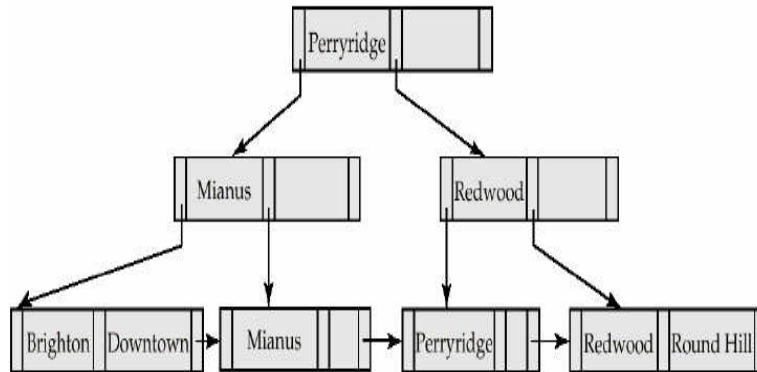
Having split a leaf node, we must insert the new leaf node into the  $B^+$ -tree structure. In our example, the new node has “Downtown” as its smallest search-key value. We need to insert this search-key value into the parent of the leaf node that was split. The  $B^+$ -tree of **fig.11** shows the result of the insertion



**Figure: 11 Insertion of “Clearview” into the  $B^+$ -tree of figure 8.**

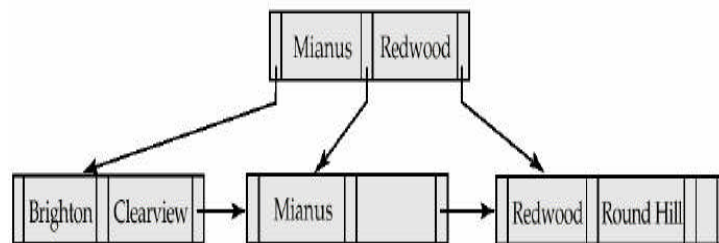
The search-key value “Downtown” was inserted into the parent. It was possible to perform this insertion because there was room for added search-key value. If there were no room, the parent would have had to be split. If the root itself is split, proceed recursively up the tree until either an insertion does not cause a split or a new root is created.

We now consider deletions that cause tree nodes to contain too few pointers, first, let us delete "Downtown" from the  $B^+$ -tree of **fig.11**. We locate the entry for "Downtown" by using our lookup algorithm. When we delete the entry for "Downtown" from its leaf node, the leaf becomes empty. Since, in our example  $n = 3$  and  $0 < \lfloor (nm-1)/2 \rfloor$ , this node must be eliminated from the  $B^+$ -tree. To delete a leaf node, we must delete the pointer to it from its parent. In our example, this deletion leaves the parent node, which formerly contained three pointers, with only two pointers. Since  $2 \geq \lfloor n/2 \rfloor$ , the node is still sufficiently large, and the deletion operation is complete. The resulting  $B^+$ -tree appears in **fi.12**.



**Figure: 12 Deletion of "Downtown" from the  $B^+$ -tree of Fig.11**

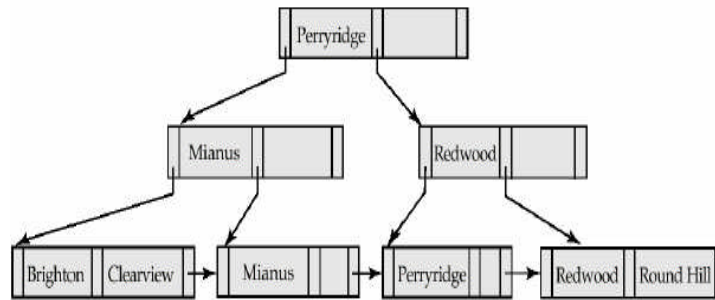
When we make a deletion from a parent of a leaf node, the parent node itself may become too small. That is exactly what happens if we delete "Perryridge" from the  $B^+$ -tree of **fig.12**. Deletion of the Perryridge entry causes a leaf node to become empty. When we delete the pointer to this node in the latter's parent. The parent is left with only one pointer. Since  $n=3$ ,  $\lfloor n/2 \rfloor = 2$ , only one pointer is too few. However, since the parent node contains useful information, we cannot simply delete it. Instead, we look at the sibling node (the nonleaf node containing the one search key, Mianus). This sibling node has room to accommodate the information contained in our now-too-small node, so we coalesce these nodes, such that the sibling node now contains the keys "Mianus" and "Redwood". The other node (the node containing only the search key "Redwood") now contains redundant information and can be deleted from its parent (which happens to be the root in our example); **fig.13** shows the result.



**Figure: 13 Deletion of "Perryridge" from the  $B^+$ -tree of Fig.12**

Notice that the root has only one child pointer after the deletion, so it is deleted and its sole child becomes the root. So the depth of the  $B^+$ -tree has been decreased by 1.

It is not always possible to coalesce nodes. As an illustration, delete “Perryridge” from the **B**<sup>+</sup>-tree of **fig.11**. In this example, the “Down town” entry is still part of the tree. Once again, the leaf node containing “Perryridge” becomes empty. The parent of the leaf node becomes too small (only one pointer). However, in this example, the sibling node already contains the maximum number of pointers, three. Thus it cannot accommodate an additional pointer. The solution in this case is to redistribute the pointers such that each sibling has two pointers. The result appears in **fig.14**.



**Figure: 14 Deletion of “Perryridge” from the **B**<sup>+</sup>-tree of Fig.11.**

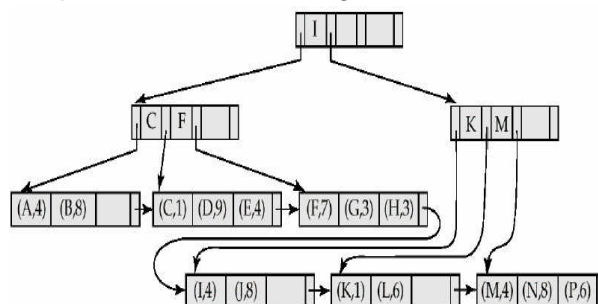
Note that the redistribution of values necessitates a change of a search-key value in the parent of the two siblings.

Although insertion and deletion operations on **B**<sup>+</sup>-trees are complicated, they require relatively few I/O operations, which is an important benefit since I/O operations are expensive. It can be shown that the number of I/O operations needed for a worst-case insertion or deletion is proportional to  $\log[n/2](K)$ , where  $n$  is the maximum number of pointers in anode, and  $K$  is the number of search-key values. In other words, the cost of insertion and deletion operations is proportional to the height of the **B**<sup>+</sup>-tree, and is therefore low. It is the speed of operation on **B**<sup>+</sup>-tree that makes them a frequently used index structure in database implementations.

### 15.3.3 **B**<sup>+</sup>-Tree File Organization

The main drawback of index sequential files organization is the degradation of performance as the file grows: with growth, an increasing percentage of index records and actual records become out of order, and are stored in overflow blocks. We solve the degradation of index lookups by using **B**<sup>+</sup>-tree indices on the file. We solve the degradation problem for storing the actual records by using the leaf level of the **B**<sup>+</sup>-tree to organize the blocks containing the actual records. We use the **B**<sup>+</sup>-tree structure not only as an index, but also as an organizer for records in a file. In a **B**<sup>+</sup>-tree file organization, the leaf nodes of the tree store records, instead of storing pointers to records.

Fig.15 shows an example of a **B**<sup>+</sup>-tree file organization.



**Figure: 15 **B**<sup>+</sup>-Tree file organization.**

Since records are usually larger than pointers, the maximum number of records that can be stored in a leaf node is less than the number of pointers in a non-leaf node. However, the leaf nodes are still required to be at least half full.

Insertion and deletion of records from a  $B^+$ -tree file organization are handled in the same way as insertion and deletion of entries in a  $B^+$ -tree index. When a record with a given key value  $v$  is inserted, the system locates the block that should contain the record by searching the  $B^+$ -tree. Upon insertion, the system splits the block in two, and redistributes the records in it (in the  $B^+$ -tree-key order) to create space for the new record. The split propagates up the  $B^+$ -tree in the normal fashion. When we delete a record, the system first removes it from the block containing it. If a block  $B$  becomes less than half full as a result, the records in  $B$  are redistributed with the records in an adjacent block  $B'$ . Assuming fixed-sized records, each block will hold at least one-half as many records as the maximum that it can hold. The system updates the nonleaf nodes of the  $B^+$ -tree in the usual fashion.

When we use a  $B^+$ -tree for file organization, space utilization is particularly important since the space occupied by the records is likely to be much more than the space occupied by keys and pointers. We can improve the utilization of space in a  $B^+$ -tree by invoking more sibling nodes in redistribution during splits and merges. The technique is applicable to both leaf nodes and internal nodes, and works as follows:

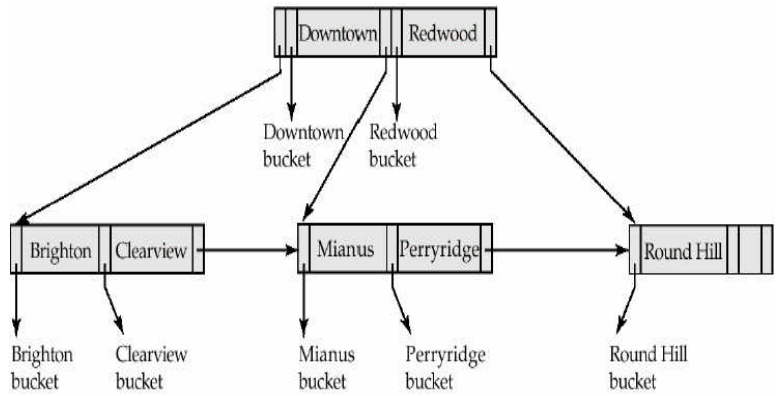
During insertion, if a node is full the system attempts to redistribute some of its entries to one of the adjacent nodes, to make space for a new entry. If this attempt fails because the adjacent nodes are themselves full, the system splits the node, and splits the entries evenly among one of the adjacent nodes and the two nodes that it obtained by splitting the original node. Since the three nodes together contain one more record than can fit in two nodes, each node will be about two-thirds full. More precisely, each node will have at least  $\lfloor 2n/3 \rfloor$  entries, where  $n$  is the maximum number of entries that the node can hold. ( $\lfloor x \rfloor$  denotes the greatest integer that is less than or equal to  $x$ ; that is, we drop the fractional part, if any)

During deletion of a record, if the occupancy of a node falls below  $\lfloor 2n/3 \rfloor$ , the system attempts to borrow an entry from one of the sibling nodes. If both sibling nodes have  $\lfloor 2n/3 \rfloor$  records, instead of borrowing an entry, the system redistributes the entries in the node and in the two siblings evenly between two of the nodes, and deletes the third node. We can use this approach because the total number of entries is  $3\lfloor 2n/3 \rfloor - 1$ , which is less than  $2n$ . With three adjacent nodes used for redistribution, each node can be guaranteed to have  $\lfloor 2n/4 \rfloor$  entries. In general, if  $m$  nodes ( $m-1$  siblings) are involved in redistribution, each node can be guaranteed to contain at least  $\lfloor (m-1)n/m \rfloor$  entries. However, the cost of update becomes higher as more sibling nodes are involved in the redistribution.

## 15.4 B -Tree Index Files

$B$ -tree indices are similar to  $B^+$ -tree indices. The primary distinction between the two approaches is that a  $B^+$ -tree eliminates the redundant storage of search-key values. In the  $B^+$ -tree of a **fig.11**, the search keys "downtown" "mianus" "Redwood," and "perryridge" appear twice. Every search key value appears in some leaf node; several are repeated in the non-leaf node.

A  $B$ -tree allows search-key values to appear only once. **Fig.16** shows a  $B$ -tree that represents the same keys as the  $B^+$ -tree of **fig.11**.



**Figure: 16 B -tree equivalent of B+ -tree in fig.12.**

Since search keys are not repeated in the **B**<sup>+</sup>-tree, we may be able to store the index in fewer tree nodes than in the corresponding **B**<sup>+</sup>-tree index.

However, since search keys that appear in non-leaf nodes appear nowhere else in the **B**<sup>+</sup>-tree, we are forced to include an additional pointer field for each search key in a non-leaf node. These additional pointers point to either file records or buckets for the associated search key.

A generalized **B**-tree leaf node appears in **fig.17a**; a non-leaf node appears in **fig.17b**. Leaf nodes are the same as in **B**<sup>+</sup>-trees.



(a)



(b)

**Figure: 17 Typical nodes of a B<sup>+</sup>-tree. (a) Leaf node. (b) Nonleaf node**

In non-leaf nodes, the pointers  $P_i$  are the tree pointers that we used also for **B**<sup>+</sup>-tree, while the pointers  $B_i$  are bucket or file-record pointers.

In the generalized **B**-tree in the figure, there are discrepancy keys in the leaf node, but there are **m-1** keys in the non-leaf node. This discrepancy occurs because non-leaf nodes must include pointers  $B_i$ , thus reducing the number of search keys that can be held in these nodes. Clearly,  $m < n$ , but the exact relationship between **m** and **n** depends on the relative size of search keys and pointers.

**15.5 Summary :**

Many queries reference only a small portion of the records in a file. To reduce the overhead in searching for these records, we can construct indices for the files that store the database.

Index-sequential files are one of the oldest index schemes used in database systems. To permit fast retrieval of records in search-key order, records are stored sequentially, and out-of-order records are chained together. To allow fast random access, we use an index structure.

There are two types of indices that we can use: dense indices and sparse indices. Dense indices contain entries for every search-key value, whereas sparse indices contain entries only for some search-key values.

If the sort order of a search key matches the sort order of a relation, an index on the search key is called a primary index. The other indices are called secondary indices. Secondary indices improve the performance of queries that use search keys other than the primary one. However, they impose an overhead on modification of the database.

The primary disadvantage of the index-sequential file organization is that performance degrades as the file grows. To overcome this deficiency, we can use a **B<sup>+</sup>**-tree index.

We can use **B<sup>+</sup>**-trees for indexing a file containing records, as well as to organize records into a file.

## 15.6 Technical Terms:

**Access Time:** The average time interval between a storage peripheral (usually a disk drive or semiconductor memory) receiving a request to read or write a certain location and returning the value read or completing the write.

**Primary Index:** An index used to improve performance on the combination of columns most frequently used to access rows in a table.

**Secondary Index:** An index that is maintained for a data file, but not used to control the current processing order of the file. For example, a secondary index could be maintained for customer name, while the primary index is set up for customer account number.

**Clustering Index:** The index that determines how rows are physically ordered in a tablespace.

**Tree:** A hierarchical structure like an organization chart.

**B -tree:** Also called a multiway tree, a B-tree is a fast data-indexing method that organizes the index into a multi-level set of nodes. Each node contains a sorted array of key values (the indexed data).

**B -tree index:** A type of index that uses a balanced tree structure for efficient record retrieval. B-tree indexes store key data in ascending or descending order.

## 15.7 Model Questions

1. When it is preferable to use a dense index rather than a Sparse index? Explain your answer.
2. What is the difference between a primary index and a Secondary index?
3. Construct a **B<sup>+</sup>**-tree for the following set of key values:  
(2,3,5,7,11,17,19,23,29,31)
4. Since indices speed query processing, why might they not be kept on several search keys? List as many reasons as possible.
5. It is possible in general to have two primary indices on the same relation for different search keys? Explain your answer.

## 15.8 References

“Database System Concepts”, 4<sup>th</sup> edition by Abraham Silberschatz, Henry F.Korth and S.Sudarshan

Fundamentals of database systems, 4<sup>th</sup> edition by R. Elmasri and B. Navathe

**AUTHOR:**

**M.V.BHUVANGA RAO. M.C.A.,  
Lecturer,  
Dept.Of Computer Science,  
JKC College,  
GUNTUR**



## Lesson 16

# Hashing

### 16.0 Objectives:

After completion of this lesson the student will be able to know about:

- Implementation of static hashing
- Implementation of dynamic hashing
- How to compare ordered index and hashing
- Usefulness of multiple-key access

### Structure Of the Lesson:

- 16.1 Static Hashing
- 16.2 Dynamic Hashing
- 16.3 Comparison Of Ordered Indexing And Hashing
- 16.4 Index Definition In SQL
- 16.5 Multiple-Key Access
- 16.6 Summary
- 16.7 Technical Terms
- 16.8 Model Questions
- 16.9 References

### 16.1 Static Hashing

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

#### 16.1.1 Hash File Organization

In a *hash file organization*, we obtain the address of the disk block containing a desired record directly by computing a function on the search key value of the record. In our description of hashing, we shall use the term bucket to denote a unit of storage that can store one or more records. A bucket is typically a disk block, but could be function.

To insert a record with search key  $K_i$ , we compute  $h(K_i)$ , which gives the address of the bucket for that record. Assume for now that there is space in the bucket to store the record. Then, the record is stored in that bucket.

To perform a lookup on a search key value  $K_i$ , we simply compute  $h(K_i)$ , and then search the bucket with that address. Suppose that two search keys,  $K_5$  and  $K_7$ , have the same hash value, i.e.,  $h(K_5) = h(K_7)$ . If we perform a lookup on  $K_5$  the bucket, to verify that the record is one that we want.

Deletion is equally straightforward. If the search key value of the record to be deleted is  $K_i$ , we compute  $h(K_i)$ , then search the corresponding bucket for that record, and delete the record from the bucket.

### 16.1.1.1 Hash Functions

The worst possible function maps all search key values to the same bucket. Such a function is undesirable because all the records have to be kept in the same bucket. A lookup has to examine every such record to find the one desired. An ideal hash function distributes the stored keys uniformly across all the buckets, so that every bucket has the same number of records.

Since we do not know at design time precisely which search key values will be stored in the file, we want to choose a hash function that assigns search key values to buckets in such a way that the distribution has these qualities.

- The distribution is **uniform**. That is, the hash function assigns each bucket the same number of search key values from the set of all possible search key values.
- The distribution is **random**. That is, in the average case, each bucket will have nearly the same number of values assigned to it, regardless of the actual distribution of search key values. More precisely, the hash value will not be correlated to any externally visible ordering on the search key values, such as alphabetic ordering or by the length of the search keys. The hash function will appear to be random.

As an illustration of these principles, let us choose a hash function for the account file using the search key branch–name. The hash function that we choose must have the desirable properties not only on the example account file that we have been using, but also an account file of realistic size for a large bank with many branches.

Assume that we decided to have 26 buckets, and we define a hash function that maps names beginning with the  $i^{\text{th}}$  letter of the alphabet to the  $i^{\text{th}}$  bucket. This hash function has the virtue of simplicity, but it fails to provide a uniform distribution. Since we expect more branch names to begin with such letters as **B** and **R** than **Q** and **X**, for example.

Now suppose that we want a hash function on the search key balance. Suppose that the minimum balance is 1 and the maximum balance is 100,000 and we use a hash function that divides the values in to 10 ranges, 1-10,000, 10,001-20,000 and so on. The distribution of search key values is uniform (since each bucket has the same number of different balance values), but it is not random. But records with balance between 1 and 10,000 are far more common than are records with balances between 90,001 and 100,000. As a result the distribution of records is not uniform, some buckets receive more records than others do. If the function has a random distribution, even if there are such correlations in the search keys, the randomness of the distribution will make it very likely that all buckets will have roughly the same number of records, as long as each search key occurs in only a small fraction of the records.

Typically hash functions perform computation on the internal binary machine representation of characters in the search key. A simple hash function of this type first computes the sum of the binary representations of the characters of a key, and then returns the sum module on the number of buckets. **Fig.1** shows the application of such a scheme, with 10 buckets, to the account file, under the assumption that the  $i^{\text{th}}$  letter in the alphabet is represented by the integer  $i$ .

bucket 0			bucket 5		
			A-102	Perryridge	400
			A-201	Perryridge	900
			A-218	Perryridge	700
bucket 1			bucket 6		
bucket 2			bucket 7		
			A-215	Mianus	700
bucket 3			bucket 8		
A-217	Brighton	750	A-101	Downtown	500
A-305	Round Hill	350	A-110	Downtown	600
bucket 4			bucket 9		
A-222	Redwood	700			

Figure: 1 Hash organization of account file, with branch–name as the key.

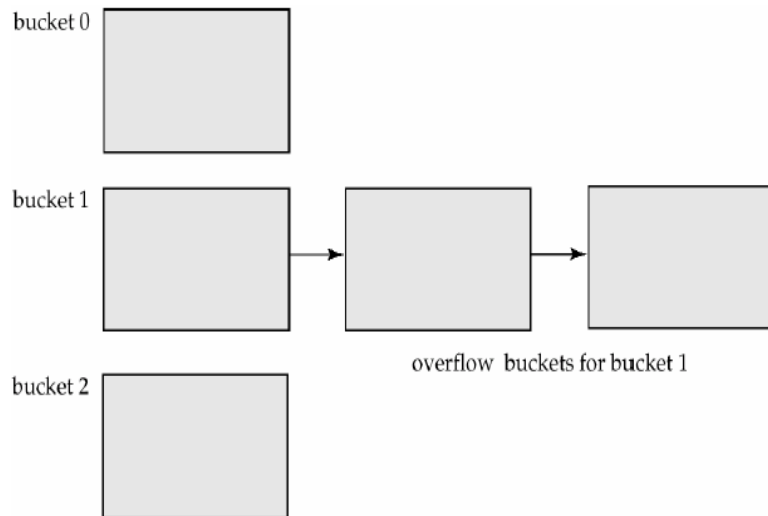
### 16.1.1.2 Handling of Bucket Overflows

So far, we have assumed that, when a record is inserted, the bucket to which it is mapped has space to store the record. If the bucket does not have enough space, a bucket overflow is said to occur. Bucket overflow can occur for several reasons:

- **Insufficient buckets-** The number of buckets, which we denote  $n_B$ , must be chosen such that  $n_B > n_r / f_r$ , where  $n_r$  denotes the total number of records that will be stored, and  $f_r$  denotes the number of records that will fit in a bucket.
- **Skew-** Some buckets are assigned more records than are others, so a bucket may overflow even when other buckets still have space. This situation is called bucket skew. Skew can occur for two reasons:
  1. Multiple records may have the same search key.
  2. The chosen hash function may result in non uniform distribution of search keys.

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

Despite allocation of a few more buckets than required, bucket overflow can still occur. We handle bucket overflow by using overflow buckets. If a record must be inserted into a bucket  $b$ , and  $b$  is already full, the system provides an overflow bucket for  $b$ , and inserts the record into the overflow bucket. If the overflow bucket is also full, the system provides another overflow bucket, and so on. All the overflow buckets of a given bucket are chained together in a linked list, as in **fig.2**. Overflow handling using such a linked list is called **overflow chaining**.



**Figure: 2 Overflow chaining in a hash structure.**

We must change the lookup algorithm slightly to handle overflow chaining. As before, the system uses the hash function on the search key to identify a bucket **b**. The system must examine all the records in the buckets **b** to see whether they match the search key, as before. In addition, if a bucket **b** has overflow buckets, the system must examine the records in all the overflow buckets also.

The form of a hash structure that we have just described is sometimes referred to as closed hashing. Under an alternative approach, called open hashing, the set of buckets is fixed, and there are no overflow chains. Instead, if a bucket is full, the system inserts records in some other bucket in the initial set of buckets **B**. One policy is to use the next bucket (in cyclic order) that has space; this policy is called linear probing. Other policies, such as computing further hash functions, are also used. Open hashing has been used to construct symbol tables for compilers and assemblers, but closed hashing is preferable for database systems. The reason is that deletion under open hashing is troublesome. Usually, compilers and assemblers perform only lookup and insertion operations on their symbol tables. However, in a database system, it is important to be able to handle deletion as well as insertion. Thus, open hashing is of only minor importance in database implementation.

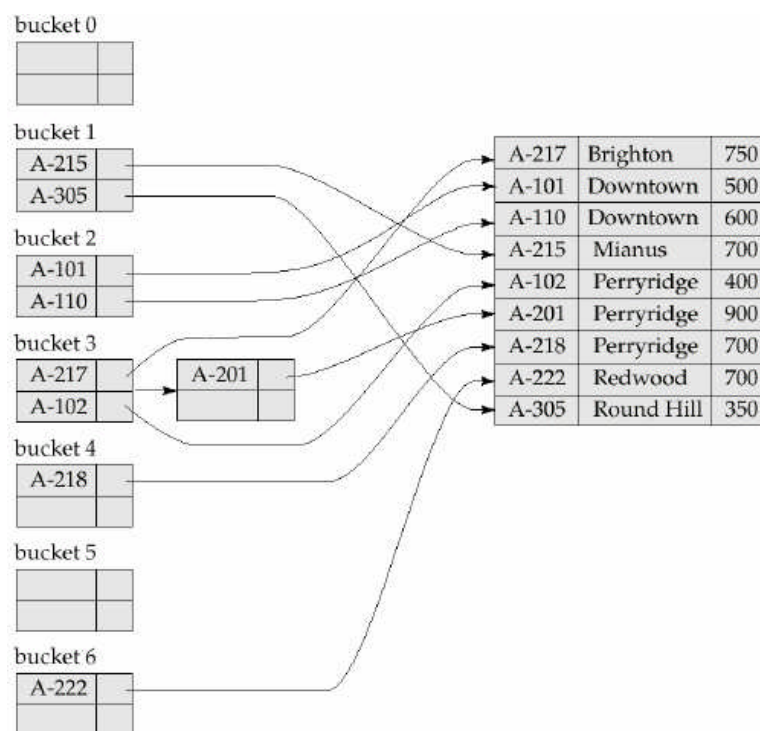
An important drawback to the form of hashing that we have described is that we must choose the hash function when we implement the system, and it cannot be changed easily thereafter if the file being indexed grows or shrinks. Since the function  $H$  maps search-key values to a fixed set **B** of bucket addresses, we want space if **B** is made large to handle future growth of the file. If **B** is too small, the buckets contain records of many different search-key values, and bucket overflows can occur. As the file grows, performance suffers.

### 16.1.2 Hash Indices

Hashing can be used not only for file organization, but also for index structure creation. A hash index organizes the search keys, with their associated pointers, into a hash file structure. We construct a hash index as follows. We apply a hash function on a search key to identify a bucket, and store the key and its associated pointers in the bucket (or in overflow buckets). **Fig.3** shows a

secondary hash index on the account file, for the search key *account-number*.

The hash function in the **fig.3** computes the sum of the digits of the account number modulo 7. The hash index has seven buckets, each of size 2 (realistic indices would, of course, have much larger bucket sizes). **Fig.3** hash index on search key *account-number* of account file. One of the buckets has three keys mapped to it, so it has an overflow bucket. In this example, *account number* is a primary key for account, and so each search key has only one associated pointer. In general, multiple pointers can be associated with each key.



**Figure: 3 Hash index on search key account-number of account file.**

## 16.2 Dynamic Hashing

As we have seen, the need to fix the set **B** of bucket addresses present a serious problem with the static hashing technique of the previous section. Most databases grow larger overtime. If we are to use static hashing for such a database; we have three classes of options:

- 1) Choose a hash function based on the current file size. This option will result in performance degradation as the database grows.
- 2) Choose a hash function based on the anticipant size of the file at some point in the future. Although performance degradation is avoided, a significant amount of space may be wasted initially.
- 3) Periodically recognize the hash structure in response to file growth. Such reorganization involves choosing a new hash function, recomputing the hash function on every record in the file, and generating new bucket assignments. This reorganization is a massive, time consuming operation. Furthermore, it is necessary to forbid access to the file during reorganization.

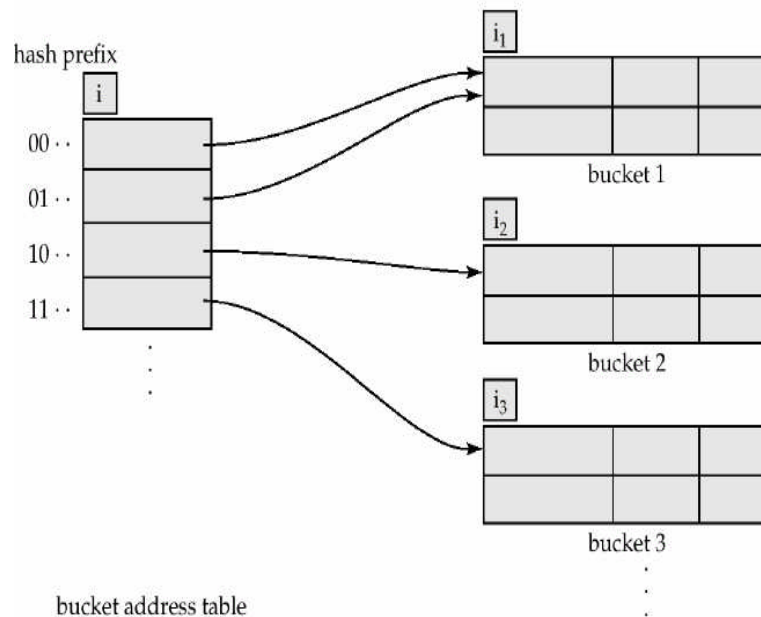
Several *dynamic hashing* techniques allow the hash function to be modified dynamically to accommodate the growth or shrinkage of the database. In this section we describe one form of dynamic hashing called *extendable Hashing*.

### 16.2.1 Data structure

Extendable hashing copes changes in the database size by splitting and coalescing buckets as the database grows and shrinks. As a result, space efficiency is retained. Moreover, since the reorganization is performed on only one bucket at a time, the resulting performance overhead is acceptably low.

With extendable hashing, we choose a hash function  $h$  with the desirable properties of uniform and randomness. However, this hash function generates values over a relatively large range namely,  $b$ -bit binary integers. A typical value for  $b$  is 32.

We do not create a bucket for each for hash value. Indeed,  $2^{23}$  is over 4 billion, and that many buckets is unreasonable for all but the largest databases. Instead, we create buckets on the demand, as records as inserted into the file. We do not use the entire  $b$  bits of the hash value initially. At any point, we use  $i$  bits, where  $0 \leq i \leq b$ . These  $i$  grow and shrink with the size of the database. **Fig.4** shows a general extendable hash structure.



**Figure: 4 General extendable hash structure.**

The  $i$  appearing above the bucket address table in the figure indicates that  $i$  bits of the hash value  $h(K)$  are required to determine the correct bucket for  $K$ . This number will, of course, change as the file grows. Although  $i$  bits are required to find the correct entry in the bucket address table, several consecutive table entries may point to the same bucket. All such entries will have a common hash prefix, but the length of this prefix may be less than  $i$ .

Therefore, we associate with each bucket an integer giving the length of the common hash prefix. In **fig.4** the integer associated with bucket  $j$  is shown as  $i_j$ . The number of bucket-address-table entries that point to bucket  $j$  is  $2^{(i-i_j)}$

### 16.2.2 Queries and Updates

We now see how to perform lookup, insertion, and deletion on an extendable hash structure.

To locate the bucket-containing search-key value  $K_j$ , the system takes the first high order bits of  $h(K_j)$ , looks at the corresponding table  $i$  entry for this bit string, and follows the bucket pointer in the table entry.

To insert a record with search key value  $K_j$ , the system follows the same procedure for lookup as before, ending up in some bucket say,  $j$ , if there is room in the bucket, the system inserts the record in the bucket. If on the other hand, the bucket is full, it must split the bucket and redistribute the current records, plus the new one. To split the bucket, the system must first determine from the hash value whether it needs to increase the number of bits that it uses.

- If  $i=j$ , only one entry in the bucket address table points to bucket  $j$ . Therefore, the system needs to increase the size of the bucket address table so that it can include pointers to the two buckets that result from splitting bucket  $j$ . It does so by considering an additional bit of hash value. It increments the value of  $i$  by 1. Thus doubling the size of the bucket address table. It replaces each entry by two entries, both of which contain the same pointer as the original entry. Now two entries in the bucket address table point to bucket  $j$ . The system allocates a new bucket (bucket  $z$ ), and sets the second entry to point new bucket. It sets  $i_z$  and  $i_z$  to  $i$ . next, it rehashes each record in the bucket either keeps in the bucket  $j$  or allocates it to the newly created bucket.

The system now re-attempts the insertion of the new record. Usually, the attempt will succeed. However, if all the records in the bucket  $j$ , as well as the new record, have the same hash value prefix, it will be necessary to split a bucket again, since all the record in the bucket  $j$  and the new record are assigned to the same bucket. If the hash function has been chosen carefully, it is unlikely that a single insertion will require that a bucket be split more than once, unless there are a large number of records with the same search key. If all the records in the bucket  $j$  have the same search key value, no amount of splitting will help. In such cases, overflow buckets are used to store the records as  $I_j$  static hashing.

- If  $i > j$ , then more than one entry in the bucket address table points to bucket  $j$ . Thus the system can split bucket  $j$  without increasing the size of the bucket address table. Observe that all the entries that point to bucket  $j$  correspond to hash prefixes that have the same value on the leftmost  $i_j$  bits. The system allocates a new bucket (bucket  $z$ ), and set  $i_z$  and  $i_z$  to the value that results from adding 1 to the original  $i_j$  value. Next, the system needs to adjust the entries in the bucket addressee table that previously pointed to bucket  $j$ . The system leaves the first half of the entries as they were (pointing to bucket  $j$ ), and sets all the previous case, the system rehashes each record in bucket  $j$ , and allocates it either to bucket  $j$  or to the newly created bucket  $z$ .

The system then re-attempts the insert. In the unlikely case that it again fails, it applies one of the two cases,  $i=j$  or  $i > j$  as appropriate

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

**Figure: 5 Sample account file.**

Our example *account* file in **fig.5** illustrates the operation of insertion. The 32 –bit hash values on branch name appear in **fig.6**.

<i>branch-name</i>	$h(\text{branch-name})$
Brighton	0010 1101 1111 1011 0010 1100 0011 0000
Downtown	1010 0011 1010 0000 1100 0110 1001 1111
Mianus	1100 0111 1110 1101 1011 1111 0011 1010
Perryridge	1111 0001 0010 0100 1001 0011 0110 1101
Redwood	0011 0101 1010 0110 1100 1001 1110 1011
Round Hill	1101 1000 0011 1111 1001 1100 0000 0001

**Figure: 6 Hash functions for branch-name.**

Assume that, initially the file is empty, as empty as in **fig.7**.

We insert the records one by one. To illustrate all the features of extendable hashing in a small structure, we shall make the unrealistic assumption that a bucket can hold only two records

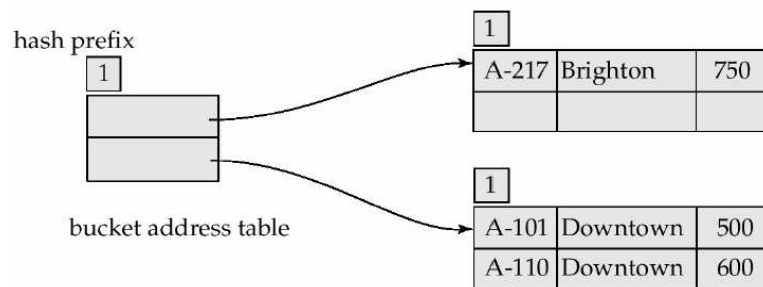


**Figure: 7 Initial extendable hash structures.**



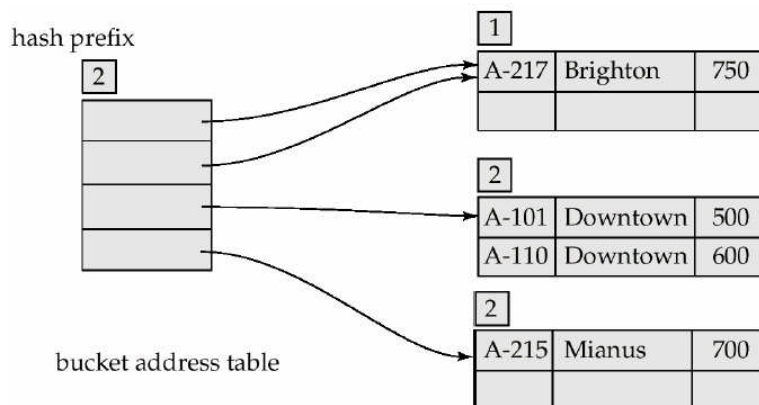
We insert the record (A-217 Brighton, 750). The bucket address table contains a pointer to the one bucket, and the system inserts the record. Next, we insert the record (A-101 Downtown, 500). The system also places this record in the one bucket of our structure.

When we attempt to insert the next record (Downtown, A-110, 600), we find that the bucket is full. Since  $i=i_0$ , we need to increase the number of bits that we use from the hash value. We now use 1 bit, allowing us  $2^1=2$  buckets. This increase in the number of bits necessitates doubling the size of the bucket address table to two entries. The system splits the bucket, placing in the new bucket those records whose search-key has a hash value beginning with 1, and leaving in the original bucket the other records



**Figure: 8 Hash structure after three insertions.**

**Fig.8** shows the state of our structure after the split. Next, we insert (A-215, Mianus, 700). Since the first bit of  $h(\text{Mianus})$  is 1, we must insert this record into the bucket pointed to by the "1" entry in the bucket address table. Once again, we find the bucket full and  $i=i_1$ . We increase the number of bits that we use from the hash to 2.

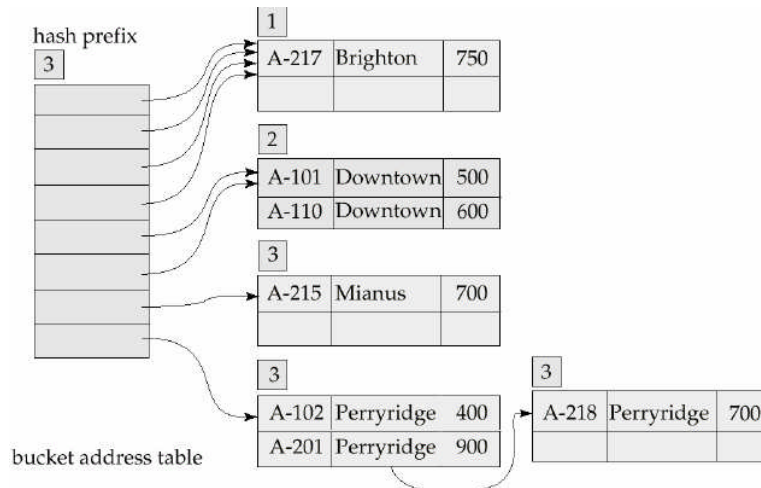


**Figure: 9 Hash structure after four instructions.**

This increase in the number of bits necessities doubling the size of the bucket address table to four entries, as in **Figure 9**. Since the bucket of **figure 8** for hash prefix 0 was not split, the two entries of the bucket address table of 00 and 01 both point to this bucket.

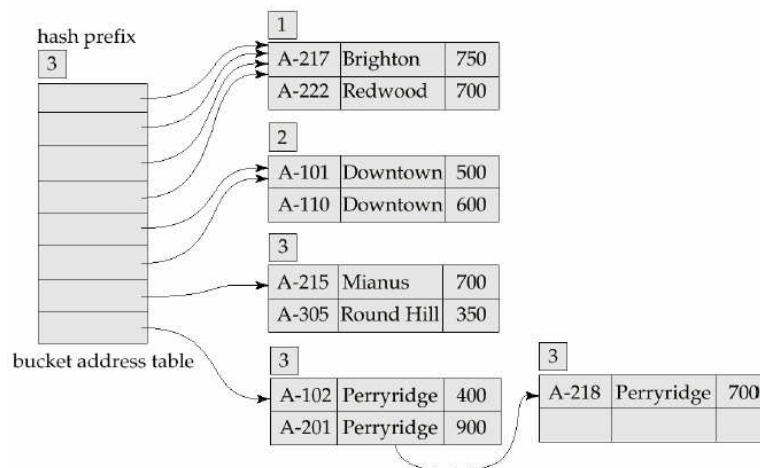
For each record in the bucket of **fig.8** for hash prefix 1 (the bucket being split), the system examines the first 2 bits of the hash value to determine which bucket of the new structure should hold it.

Next, we insert (A-102, Perryridge, 400), which goes in the same bucket as Mianus. The following insertion, of (A-201, Perryridge, 900), results in a bucket overflow, leading to an increase in the number of bits, and a doubling of the size of the bucket address table. The insertion of the third Perryridge record, (A-218, Perryridge, 700), leads to another overflow. However, this overflow cannot be handled by increasing the number of bits, since there are three records with exactly the same hash value. Hence the system uses an overflow bucket, as in **Fig.10**.



**Figure: 10 Hash structure after seven insertions.**

We continue in this manner until we have inserted all the *account* records of **Fig.5**. The resulting structure appears in **Fig.11**.



**Figure: 11 Extendable hash structure for the account File.**

### 16.2.3 Comparison With Other Schemes

We now examine the advantages and disadvantages of extendable hashing, compared with other schemes that we have discussed. The main advantage of extendable hashing is that performance does not degrade as the file grows. Furthermore, there is minimal space overhead. Although the bucket address table incurs additional overhead, it contains one pointer for each hash value for the

current prefix length. This table is thus small. The main space saving of extendable hashing over other forms of hashing is that no buckets need to be reserved for future growth; rather, buckets can be allocated dynamically.

A disadvantage of extendable hashing is that lookup involves an additional level of indirection, since the system must access the bucket address table before accessing the bucket itself. This extra reference has only a minor effect on performance.

Thus, extendable hashing appears to be highly attractive technique, provided that we are willing to accept the added complexity involved in its implementation. The bibliographical notes reference more detailed descriptions of extendable hashing implementation.

### 16.3 Comparison Of Ordered Indexing And Hashing

We have seen several ordered-indexing schemes and several hashing schemes. We can organize files of records as ordered files by using index-sequential organization or **B**<sup>+</sup>-tree organizations. Alternatively, we can organize the files by using hashing. Finally, we can organize them as heap files, where the records are not ordered in any particular way.

Each scheme has advantage in certain situations. A database-system implementer could provide many schemes, leaving the final decision of which schemes to use to the database designer. However, such an approach requires the implementer to write more code, adding both to the cost of the system and to the space that the system occupies. Most database systems support **B**<sup>+</sup>-trees and may additionally support some form of hash file organization or hash indices.

To make a wise choice of file organization and indexing techniques for a relation, the implementer or the database designer must consider the following issues:

- Is the cost of periodic organization of the index or the hash organization acceptable?
- What is the relative frequency of insertion and deletion?
- Is it desirable to optimize average access time at the expense of increasing the worst-case access time?
- What types of queries are users likely to pose?

We have already examined the first three of these issues, first in our review of the relative merits of specific indexing techniques, and again in our discussion of hashing techniques. The fourth issue, the expected type of query, is critical to the choice of ordered indexing or hashing.

If most queries are of the form

```
select  $A_1, A_2, \dots, A_n$ 
from  $r$ 
where  $A_i = c$ 
```

Then, to process this query, the system will perform a lookup on an ordered index or hash structure for attribute  $A_i$ , for value  $c$ . For queries of this form, a hashing scheme is preferable. An ordered-index lookup requires time proportional to the log of the number of values in  $r$  for  $A_i$ . In a hash structure, however, the average lookup time is a constant independent of the size of the database. The only advantage to an index over hash structure for this form of query is that the worst-case lookup time is proportional to the log of the number of values in  $r$  for  $A_i$ . By contrast, for hashing, the worst-case lookup time is proportional to the number of values in  $r$  for  $A_i$ . However, the worst-case lookup time is unlikely to occur with hashing and hashing is preferable in this case.

Ordered-index techniques are preferable to hashing in cases where the query specifies a range of values. Such a query takes the following form:

```

select  $A_1, A_2, \dots, A_n$ 
from  $r$ 
where  $A_i = c_2$  and  $A_i = c_1$ 

```

In other words, the preceding query finds all the records with  $A_i$  values between  $c_1$  and  $c_2$ .

Let us consider how we process this query using an ordered index, we have a hash structure, we can perform a lookup on  $c_1$ . Once we have found the bucket for value  $c_1$ , we follow the pointer chain in the index to read the next bucket in order, and we continue in this manner until we reach  $c_2$

If, instead of an ordered index, we have a hash structure, we can perform a lookup on  $c_1$  and locate the corresponding bucket-but it is not easy, in general, to determine the next bucket that must be examined. The difficulty arises because a good hash function assigns values randomly to buckets. Thus, there is no simple notion of "next bucket in sorted order". The reason we cannot chain buckets together in sorted order on  $A_i$  is that each bucket is assigned many search-key values. Since values are scattered randomly by the hash function, the values in the specified range are likely to be scattered across many or all of the buckets. Therefore, we have to read all the buckets to find the required search keys.

## 16.4 Index Definition In SQL

The SQL standard does not provide any way for the database user or administrator to control what indices are created and maintained in the database system.

We create an index by the **create index** command, which takes the form:

```
Create index <index-name> on <relation-name> (<attribute-list>)
```

The *attribute-list* is the list of attributes that form the search key for the index.

To define an index name *b-index* or *branch* relation with *branch-name* as the search key, we write

```
create index b-index on branch (branch-name)
```

If we wish to declare that the search key is a candidate key, we add the attribute **unique** to the index definition. Thus, the command

```
create unique index b-index on branch (branch -name)
```

declares *branch-name* to be a candidate key for *branch*. If, at the time we enter the **create unique index command**, *branch-name* is not a candidate key, the system will display an error message, and the attempt to create the index will fail. If the index creation attempt succeeds, any sub-sequent attempt to insert a tuple that violates the key declaration will fail.

The index name we specified for an index is required to drop an index. The **drop index** command takes the form:

```
drop index <index name>
```

## 16.5 Multiple-Key Access

Until now, we have assumed implicitly that only one index (or hash table) is used to process a query on a relation. However, for certain types of queries, it is advantageous to use multiple indices if they exist.

### 16.5.1 Using Multiple Single Key Indices

Assume that the account file has two indices: one for branch-name and one for balance. Consider the following query: "Find all account numbers at the Perryridge branch with balances equal to \$1000". We write

```
Select loan_number
from account
where branch-name="Perryridge"
and balance=1000
```

There are three strategies possible for processing this query:

1. Use the index on branch-name to find all records pertaining to the Perryridge branch. Examine each such record to see whether balance=1000.
2. Use the index on branch-name to find all records pertaining to accounts with balance of \$1000. Examine each such record to see whether branch-name="Perryridge".
3. Use the index on branch-name to find pointers to all records pertaining to the perryridge branch. Also, use the index on balance to find pointers to all records pertaining to accounts with a balance of \$1000. Take the intersection of these two sets of pointers. Those pointers that are in the intersection point to records pertaining to both Perryridge and accounts with balance of \$1000.

The third strategy is the only one of the three that takes advantage of the existence of multiple indices. However, even this strategy may be a poor choice if all of the following hold:

- There are many records pertaining to the Perryridge branch.
- There are many records pertaining to accounts with a balance of \$1000.
- There are only a few records pertaining to both the Perryridge branch and accounts with a balance of \$1000.

If these conditions hold, we must scan a large number of pointers to produce a small result. An index structure called a "bitmap index" greatly speeds up the intersection operation used in the third strategy.

### 16.5.2 Indices on Multiple Keys

An alternative strategy for this case is to create and use an index on a search key (branch-name, balance), i.e., the search key consisting of the branch-name concatenated with the account balance. The structure of the index is the same as that of any other index, the only difference being that the search key is not a single attribute, but rather is a list of attributes. The search key can be represented as a tuple of values, of the form  $(a_1, \dots, a_n)$ , where the indexed attributes are  $A_1, \dots, A_n$ . The ordering of search key values is the lexicographic ordering. For example, for the case of two attribute search keys,  $(a_1, a_2) < (b_1, b_2)$  if either  $a_1 < b_1$  or  $a_1 = b_1$  and  $a_2 < b_2$ . Lexicographic ordering is basically the same as alphabetic ordering of words.

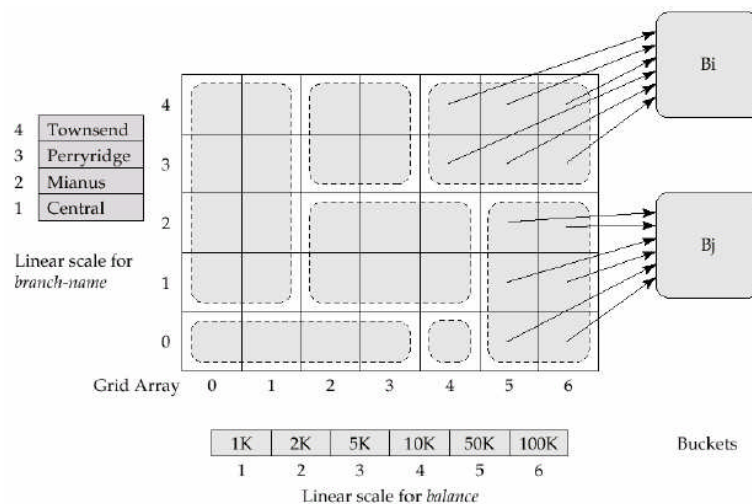
The use of an ordered-index structure on multiple attributes has a few shortcomings. As an illustration, consider the query

```
select loan-number
from account
where branch-name="Perryridge" and balance=1000
```

We can answer this query by using an ordered index on the search key (branch-name, balance): for each value of branch-name that is less than “Perryridge” in alphabetic order, the system locates records with a balance value of 1000. However, each record is likely to be in a different disk block, because of the ordering of records in the file, leading to many I/O operations. The difference between this query and the previous one is that the condition on branch-name is a comparison condition, rather than an equality condition.

### 16.5.3 Grid Files

**Fig.12** shows part of a grid file for the search keys branch-name and balance on the account file. The two dimensional array in the figure is called the grid array, and the one-dimensional arrays are called linear scales. The grid file has a single grid array, and one linear scale for each search-key attribute.



**Figure: 12 Grid file on keys branch-name and balance of the account file.**

Search keys are mapped to cells in this way. Each cell in the grid array has a pointer to a bucket that contains the search-key values and pointers to records. Only some of the buckets and pointers from the cells are shown in the figure. To conserve space, multiple elements of the array can point to the same bucket. The dotted boxes in the figure indicate which cells point to the same bucket.

Suppose that we want to insert in the grid-file index a record whose search-key value is (“Brighton”, 500000). To find the cell to which the key is mapped, we independently locate the row and column to which the cell belongs.

We first use the linear scales on branch-name to locate the row of the cell to which the search key maps. To do so, we search the array to find the least element that is greater than “Brighton”. In this case, it is the first element, so the search key maps to the row marked 0. If it were the  $i$ th element, the search key would map to row  $i-1$ . If the search key is greater than or equal to all elements in the linear scale, it maps to the final row. Next, we use that linear scale on balance to find out similarly to which column the search key maps. In this case, the balance 500000 maps to column 6.

Thus, the search-key value (“Brighton”, 500000) maps to the cell in the row 0 column 6. Similarly, (“Downtown”, 600000) would map to the cell in row 1 column 5. Cells point to the same bucket (as indicated by the dotted box), so, in both cases, the system store the search-key values and the pointer to the record in the bucket labeled  $B_j$  in the figure.

To perform a lookup to answer our example query, with the search condition of

*branch-name* < "Perryridge" **and** *balance* = 1000

We find all rows that can contain branch names less than "Perryridge", using the linear scale on *branch-name*. In this case, these rows are 0, 1 and 2. Rows 3 and beyond contain branch names greater than or equal to "Perryridge". Similarly, we find that only column 1 contain a balance value of 1000. In this case, only column 1 satisfies this condition. Thus, only the cells in column 1, rows 0, 1 and 2, can contain entries that satisfy the search condition. The grid structure is suitable also for queries involving one search key. Consider this query:

```
select *
from account
where branch-name= "Perryridge"
```

The linear scale on *branch-name* tells us that only cells in row 3 can satisfy this condition. Since there is no condition on *balance*, we examine all buckets pointed to by cells in row 3 to find entries pertaining to Perryridge. Thus, we can use a grid-file index on two search keys to answer queries on either search key by itself, as well as to answer queries on both search keys. Thus, a single grid-file index can serve the role of three separate indices. If each index were maintained separately, the three together would occupy more space, and the cost of updating them would be high. Grid files provide a significant decrease in processing time for multiple-key queries.

#### 16.5.4 Bitmap Indices

Bitmap indices are specialized type of index designed for easy querying on multiple keys, although each bitmap index is built on a single key.

For bitmap indices to be used, records in a relation must be numbered sequentially, starting from, say 0. Given a number *n*, it must be easy to retrieve the record numbered *n*. This is particularly easy to achieve if records are fixed in size, and allocated on consecutive blocks of a file. The record number can then be translated easily into a block number and a number that identifies the record within the block. Consider the relation *r*, with an attribute **A** that can take on only one of a small number (for example, 2 to 20) values. For instance, a relation *customer-info* may have an attribute *gender*, which can take only values **m** (male) or **f** (female).

##### 16.5.4.1 Bitmap Index Structure

A bitmap is simply an array of bits. In its simplest form, a bitmap index on the attribute **A** of relation *r* consists of one bitmap for each value that **A** can take. Each bitmap has as many bits as the number of records in the relation. The *i*<sup>th</sup> bit of the bitmap for value **v<sub>j</sub>** is set to 1 if the record numbered *i* has the value **v<sub>j</sub>** for attribute **A**. all other bits of the bitmap are set to 0.

In our example, there is one bitmap for the value **m** and one for **f**. The *i*th bit of bitmap for **m** is set to 1 if *gender* value of the record numbered *i* is **m**. All other bits of the bitmap for **m** are set to 0. Similarly, the bitmap for **f** has the value 1 for bits corresponding to records with the value **f** for the *gender* attribute; all other bits have the value 0. **Fig.13** shows an example of bitmap indices on a relation *customer-info*.

record number	name	gender	address	income-level
0	John	m	Perryridge	L1
1	Diana	f	Brooklyn	L2
2	Mary	f	Jonestown	L1
3	Peter	m	Brooklyn	L4
4	Kathy	f	Perryridge	L3

Bitmaps for gender		Bitmaps for income-level			
m	<table border="1"><tr><td>10010</td></tr></table>	10010	L1	<table border="1"><tr><td>10100</td></tr></table>	10100
10010					
10100					
f	<table border="1"><tr><td>01101</td></tr></table>	01101	L2	<table border="1"><tr><td>01000</td></tr></table>	01000
01101					
01000					
		L3	<table border="1"><tr><td>00001</td></tr></table>	00001	
00001					
		L4	<table border="1"><tr><td>00010</td></tr></table>	00010	
00010					
		L5	<table border="1"><tr><td>00000</td></tr></table>	00000	
00000					

**Figure: 13 Bitmap indices on relation *customer-info*.**

We now consider when bitmaps are useful. The simplest way of retrieving all records with value **m** (or value **f**) would be to simply read all records of the relation and select those records with value **m** (or **f** respectively). The bitmap index doesn't really help to speed up such a selection.

In fact, bitmap indices are useful for selections mainly when there are selections on multiple keys. Suppose we create a bitmap index on attribute *income-level*, which we described earlier, in addition to the bitmap index on *gender*.

#### 16.5.4.2 Efficient implementation of bitmap operations

We can compute the intersection of two bitmaps easily by using a **for** loop: the  $i^{\text{th}}$  iteration of the loop computes the **and** of the  $i^{\text{th}}$  bits of the two bitmaps. We can speed up computation of the intersection greatly by using bit-wise **and** instructions supported by most computer instruction sets. A *word* usually consists of 32 or 64 bits, depending on the architecture of computer. A bit-wise **and** instruction takes two words as input and outputs a word where each bit is the logical **and** of the bits in the corresponding positions of the input words. What is important to note is that single bit-wise **and** instruction can compute the intersection of 32 or 64 bits at *once*.

#### 16.5.4.3 Bitmaps and B<sup>+</sup>-Trees

Bitmaps can be combined with regular B<sup>+</sup>-tree indices for relations where a few attribute values are extremely common, and other values also occur, but much less frequently. In a B<sup>+</sup>-tree index leaf, for each value would normally maintain a list of all records with that value for the indexed attribute. Each element of the list would be a record identifier, consisting of at least 32 bits, and usually more. For a value that occurs in many records, we store a bitmap instead of a list of records.

### 16.6 Summary

Static hashing uses hash functions in which the set of bucket addresses is fixed. Such hash functions cannot easily accommodate databases that grow significantly larger over time. There are several dynamic hashing techniques that allow the hash function to be modified. One example is extendable hashing, which copes with changes in database size by splitting and coalescing buckets as the database grows and shrinks.

We can also use hashing to create secondary indices; such indices are called hash indices. For notational convenience, we assume hash file organizations have an implicit hash index on the search key used for hashing.



Ordered indices such as **B<sup>+</sup>**-trees and hash indices can be used for selections based on equality conditions involving single attributes. When multiple attributes are involved in a selection condition, we can intersect record identifiers retrieved from multiple indices. Grid files provide a general means of indexing on multiple attributes.

Bitmap indices provide a very compact representation for indexing attributes with very few distinct values. Insertion operations are extremely fast on bitmaps, making them ideal for supporting queries on multiple attributes.

## 16.7 Technical Terms

**Hashing:** The conversion of a column's primary key value to a database page number on which the row will be stored. Retrieval operations that specify the key column value use the same hashing algorithm and can locate the row directly. Hashing provides fast retrieval for data that contains a unique key value.

**Directory hashing:** Directory hashing is a method of indexing file locations on a disk so the time needed to locate a file is reduced.

**Bucket:** A bucket is most commonly a type of data buffer or a type of document.

**Hash Function:** A formula that is applied to each value of a table column or a combination of several columns, called the index key, to get the address of the area in which the row should be stored. When locating data, the database uses the hash function again to get the data's location.

**Skew:** When the superstructure is not perpendicular to the substructure, a skew angle is created. The skew angle is the acute angle between the alignment of the superstructure and the alignment of the substructure.

**Bitmap index:** An index that uses a string of bits that corresponds to rows in a table to indicate whether the indexed value is stored in the row.

**Primary key:** A column in a table whose values uniquely identify the rows in the table. A primary key value cannot be NULL.

## 16.8 Model Questions

1. Explain the distinction between closed and open hashing. Discuss the relative merits of each technique in database applications?
2. What are the causes of bucket overflow in a hash file organization? What can be done to reduce the occurrence of bucket overflows?
3. Why is a hash structure not the best choice for a search key on which range queries are likely?
4. Show how to compute existence bitmaps from other bitmaps.
5. How does data encryption affect index schemes?

## 16.9 References

“Database System Concepts”, 4<sup>th</sup> edition by Abraham

Silberschatz, Henry F.Korth and S.Sudarshan.

Fundamentals of database systems, 4<sup>th</sup> edition by R. Elmasri and B. Navathe.

**AUTHOR:**

**M.V.BHUVANGA RAO. M.C.A.,  
Lecturer,  
Dept. Of Computer Science,  
JKC College,  
GUNTUR**

## Lesson 17

# Transactions

### 17.0 Objectives:

After completion of this lesson the student will be able to know about:

- Different transaction concepts.
- Different states of a transaction.
- Atomicity and durability of transactions.
- Concurrent executions.
- Serializability and how conflicts can be eliminated in serializability.
- Recoverability.

### Structure Of the Lesson:

- 17.1 Transaction concepts
- 17.2 Transaction state
- 17.3 Implementation of Atomicity and Durability
- 17.4 Concurrent Executions
- 17.5 Serializability
- 17.6 Recoverability
- 17.7 Implementation of Isolation
- 17.8 Transaction Definition in SQL
- 17.9 Testing for Serializability
- 17.10 Summary
- 17.11 Technical terms
- 17.12 Model questions
- 17.13 References

## 17.1 Transaction concepts

A transaction is a **unit** of program execution that accesses and possibly updates various data items. Usually, a user program written in a high-level data manipulated languages initiates a transaction. The transaction consists of all operations executed between the **begin transaction** and **end transaction**.

To ensure integrity of the data, we require that the database system maintain the following properties of the transaction:

**Atomicity**

**Consistency**

**Isolation**

**Durability**

**Atomicity:** Either all operations of the transaction are reflected properly in the database, or none are.

**Consistency:** Execution of a transaction in isolation (that is, with no other transaction executing concurrently) preserves the consistency of the database.

**Isolation:** Even though multiple transactions may execute concurrently, the system guarantees that, for every pair of transactions  $T_i$  and  $T_j$ , it appears to  $T_i$  that  $T_j$  finished execution before  $T_i$  started, or  $T_j$  started execution after  $T_i$  finished. Thus, each transaction is unaware of other transactions executing concurrently in the system.

**Durability:** After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures.

These properties are often called the **ACID** properties; the acronym is derived from the first letter of each of the four properties.

## 17.2 Transaction state

Every transaction must be in one of the following states.

**Active**, the initial state; the transaction stays in this state while it is executing.

**Partially committed**, after the final statement has been executed.

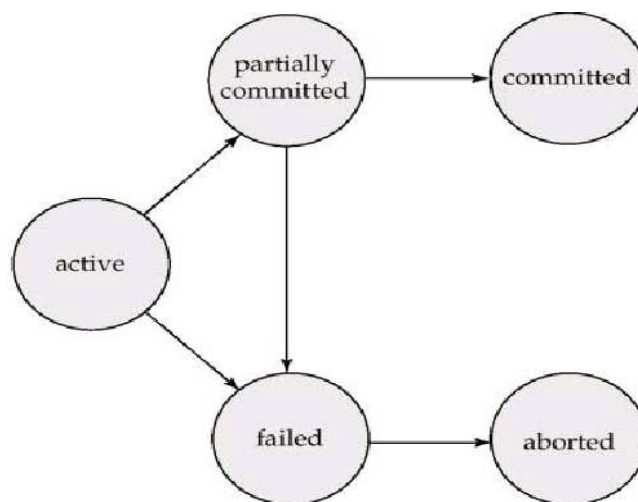
**Failed**, after the discovery that normal execution can no longer proceed.

**Aborted**, after the transaction has been rolled back and the database has been restored to its state prior to the start of the transaction.

**Committed**, after successful completion.

The state diagram corresponding to a transaction appears in **figure 1**. We say that a transaction has committed only if it has entered the committed state. Similarly, we say that a transaction has aborted only if it has entered the aborted state. A transaction is said to have terminated if it has either committed or aborted.

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



**Figure 1** State diagram of a transaction

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

A transaction enters the failed state after the system determines that the transaction can no longer proceed with its normal execution (for example, because of logical or hardware errors). Such a transaction must be rolled back. Then it enters the aborted state. At this point, the system has two opinions.

1. It can restart the transaction, but only if the transaction was aborted as a result of some hardware or software error that was not created through the internal logic of the transaction. A restarted transaction is considered to be a new transaction.
2. It can kill the transaction. It usually does so because of some internal logical error that can be corrected only by rewriting the application program, or because the input was bad, or because the desired data were not found in that database.

We must be cautious when dealing with observable external writes, such as writes to a terminal or printer. Once such a write has occurred, it cannot be erased, since it may have been seen external to the database system. Most systems allow such writes to take place only after the transaction has entered the committed state. One way to implement such a scheme is for the database system to store any value associated with such external writes temporarily in nonvolatile storage, and to perform the actual writes only after the transaction entered the committed state, but before it could complete the external writes, the database system will carry out the external writes (using the nonvolatile storage) when the system is restarted.

Handling external writes can be more complicated in some situations. For example suppose the external action is that of dispensing cash at an automated teller machine, and the system fails just before the cash is actually dispensed (we assume that cash can be dispensed automatically). It makes no sense to dispense cash when the system is restarted, since the user may have left the machine. In such a case a compensating transaction, such as depositing the cash back in the users account, needs to be executed when the system is restarted.

### 17.3 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 copy**, 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 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. After the operating system has written all the pages to the 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. **Figure 2** depicts the scheme, showing the database state before and after the update.

The transaction is said to have been **committed** at the point where the updated db pointer is written to disk.

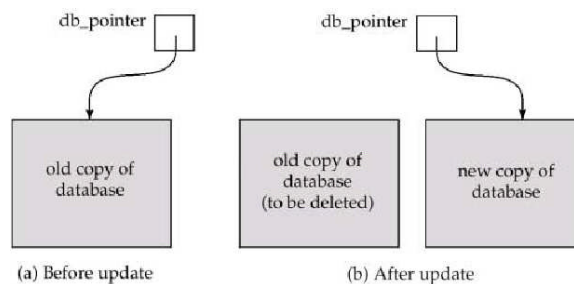


Figure 2 Shadow-copy technique for atomicity and durability

We now consider how the technique handles transaction and system failures. First, consider transaction failure. If the transaction fails at any time before db-pointer is updated, the old contents of the database are not affected. We can abort the transaction by just deleting the new copy of the database. Once the transaction has been committed, all the updates that are performed are in the database pointed to by db-pointer. Thus, either all updates of the transaction are reflected, or none of the effects are reflected, regardless of transaction failure.

Now consider the issue of system failure. Suppose that the system fails at any time before the updated db-pointer is written to disk. Then, when the system restarts, it will read db-pointer and will thus see the original contents of the database, and none of the effects of the transaction will be

visible on the database. Next, suppose that the system fails after db-pointer has been updated on disk. Before the pointer is updated, all pages of the new copy of the database were written to disk. Again, we assume that, once a file is written on disk, its contents will not be damaged even if there is a system failure. Therefore, when the system restarts, it will be read db-pointer and will thus see the contents of the database after all the updates performed by the transaction.

The implementation actually depends on the write to db-pointer being atomic, that is either all its bytes are written or none of its bytes are written. If some of the bytes of the pointer were updated by the write, but others were not, the pointer is meaningless, and neither old nor new versions of the database may be found when the system restarts. Luckily, disk systems provide atomic updates to entire blocks, or at least to a disk sector. In other words, the disk system guarantees that it will be update db-pointer automatically, as long as we make sure that db-pointer at the beginning of a block.

Thus, the atomicity and durability properties of transactions are ensured by the shadow copy implementation of the recovery management component.

## 17.4 Concurrent executions

Transaction-processing systems usually allow multiple transactions to run concurrently. Allowing multiple transactions to update data concurrently causes several complications with consistency of the data, as we saw earlier. Ensuring consistency in spite of concurrent execution of transactions requires extra work; it is far easier to insist that transactions run serially—that is, one at a time, each starting only after the previous one has completed. However, there are two good reasons for allowing concurrency;

**Improved throughput and resource utilization:** A transaction consists of many steps. Some involve I/O activity; others involve CPU activity. The cup and the disks in a computer system can operate in parallel. Therefore, I/O activity can be done in parallel with processing at the CPU. The parallelism of the CPU and the I/O system can therefore be exploited to run multiple transactions in parallel. While a read or write on behalf of one transaction is in progress on one disk, another transaction can be running in the CPU. While another disk may be executing a read or write on behalf of a third transaction. All of this increases the throughput of the system—that is, the no of transactions executed in a given amount of time. Correspondingly, the processor and disk utilization also increase; in other words, the processor and disk spend less time idle, or not performing any usual work.

**Reduced waiting time.** There may be a mix of transactions running on a system, some short and some long. If transactions run serially, a short transaction may have to wait for a preceding long transaction to complete, which can lead to unpredictable delays in running a transaction. If the transactions are operating on different parts of the database, it is better to left them run concurrently, sharing the CPU cycles and disk accesses among them. Concurrent execution reduces the unpredictable delays in running the transactions. Moreover, it also reduces the **average response time**: the average time for a transaction to be completed after it has been submitted.

The motivation for using concurrent execution in a database is essentially the same as the motivation for using multiprogramming in an operating system. When several transactions run concurrently, database consistency can be destroyed despite the correctness of each individual transaction. In this section. We present the concept of schedules to help identify those executions that are guaranteed to ensure consistency.

The database system must control the interaction among the concurrent transactions to prevent them from destroying the consistency of the database. It does so through a variety of mechanisms called concurrency-control schemes. Consider the simplified banking system, which has several accounts, and a set of transactions, that access and update those accounts. Let  $T_1$  and  $T_2$  be two transactions that transfer funds from one account to another. Transaction  $T_1$  transfers \$50 from account **A** to account **B**. It is defined as

```

T1: read (A);
      A: =A-50;
      Write (A);
      Read (B);
      B: =B+50;
      Write(B)

```

Transaction  $T_2$  transfers 10 percent of the balance from account **A** to account **B**. It is defined as

```

T2: read (A);
      Temp: =A*0.1;
      A:=A-temp;
      Write (A);
      Read (B);
      B: =B+temp;
      Write(B)

```

Suppose the current values of accounts **A** and **B** are \$1000 and \$2000, respectively. Suppose also that the two transactions are executed one at a time in the order  $T_1$  followed by  $T_2$ . This execution sequence appears in fig 15.3. In the figure; the sequence of instruction steps in the chronological order from top to bottom. With instructions of  $T_1$  appearing in the left column and instructions of  $T_2$  appearing in the right column. The final values of accounts **A** and **B**, after the execution in **figure 3** takes place, are \$855 and \$2145, respectively. Thus, the total amount of money in accounts **A** and **B** – that is, the sum of **A+B** is preserved after the execution of both transactions.

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

Figure 3 **Schedule 1-a serial schedule in which  $T_1$  is followed by  $T_2$ .**



Similarly, if the transactions are executed one at a time in the order followed by  $T_2$  followed by  $T_1$ , then the corresponding execution sequence is that of **figure 4**. Again, as expected, the **sum A+B** is preserved, and the final values of accounts **A** and **B** are \$850 and \$2150, respectively.

The execution sequences just described are called schedules. They represent the chronological order in which instructions are executed in the system. Clearly, a schedule for a set of transactions must consist of all instructions of those transactions, and must preserve the order in which the instructions appear in each individual transaction. For, example, in transaction  $T_1$ , the instruction write (**A**) must appear before the instruction read (**B**), in any valid schedule. In the following discussion, we shall refer to the first execution sequence ( $T_1$  followed by  $T_2$ ) as schedule 1, and to the second execution sequence ( $T_2$  followed by  $T_1$ ) as schedule 2.

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

**Figure 4** Schedule 2-a serial schedule in which  $T_2$  is followed by  $T_1$ .

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

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

Several execution sequences are possible, since the various instructions from both transactions may now be interleaved. In general, it is not possible to predict exactly how many instructions of a transaction will be executed before the CPU switches to another transaction. Thus, the number of possible schedules for a set of  $n$  transactions is much larger than  $n!$ .

Returning to our previous example, suppose that two transactions are executed concurrently. One possible schedule appears in **figure 5**. After this execution takes place, we arrive at the same state as the one in which the transactions are executed serially in the order  $T_1$  followed by  $T_2$ . The **sum A + B** is indeed preserved.

T <sub>1</sub>	T <sub>2</sub>
read(A) A:=A-50 write(A)	read(A) temp:=A*0.1 A:=A-temp write(A)
read(B) B:=B+50 Write(B)	read(B) B:=B+ temp write(B)

**Figure 5** Schedule 3-a Concurrent Schedule equivalent to schedule 1.

Not all concurrent executions result in a correct state. To illustrate, consider the schedule of **figure 6**.

T <sub>1</sub>	T <sub>2</sub>
read(A) A:=A-50	read(A) temp:=A*0.1 A:=A-temp write(A) read(B)
write(A) read(B) B:=B+50 Write(B)	B:=B+ temp write(B)

**Figure 6** Schedule 4-a concurrent schedule.

After the execution of this schedule, we arrive at a state where the final values of accounts **A** and **B** are \$950 and \$2100, respectively. This final state is inconsistent state, since we have gained \$50 in the process of the concurrent execution. Indeed, the **sum A + B** is not preserved by the execution of the two transactions.

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

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

### 17.5 Serializability

The database system must control concurrent execution of transactions, to ensure that the database state remains consistent. Before we examine how the database system can carry out this task, we must first understand which schedules will ensure consistency, and which schedules will not.

Since transactions are programs, it is computationally difficult to determine exactly what operations a transaction performs and how operations of various transactions interact. For this reason, we shall not interpret the type of operations that a transaction can perform on data item. Instead, we consider only two operations: read and write. We thus assume that, between a read (**R**) instruction and a write (**W**) instruction on a data item **Q**, a transaction may perform an arbitrary sequence of operations on the copy of **Q** that is residing in the local buffer of the transaction. Thus, the only significant operations of a transaction, from a scheduling point of view, are its read and write instructions. We shall therefore usually show only read and write instructions in schedules, as we do in schedule 3 in **figure 7**

T <sub>1</sub>	T <sub>2</sub>
read(A) write(A)	read(A) write(A)
read(B) Write(B)	read(B) write(B)

**Figure 7** Schedule 3-showing only the read and write instructions.

In this section, we discuss different forms of schedule equivalence; they lead to the notions of conflict serializability and view serializability.

### 17.4.1 Conflict serializability:

Let us consider a schedule  $s$  in which there are two consecutive instructions  $I_i$  and  $I_j$  refer to different data items, and then we can swap  $I_i$  and  $I_j$  without affecting the results of any instruction in the schedule. However if  $I_i$  and  $I_j$  refer to the same data item  $Q$ , then the order of the two steps may matter. Since we are dealing with only read and write instructions, there are four cases that we need to consider:

- 1)  $I_i = \text{read}(Q)$ ,  $I_j = \text{read}(Q)$ . The order of  $I_i$  and  $I_j$  does not matter, since  $T_i$  and  $T_j$  read the same value of  $Q$ , regardless of the order.
- 2)  $I_i = \text{read}(Q)$ ,  $I_j = \text{write}(Q)$ . If  $I_i$  comes before  $I_j$ , then  $T_i$  does not read the value of  $Q$  that is written by  $T_j$  in instruction  $I_j$ . If  $I_j$  comes before  $I_i$ , then  $T_i$  reads the value of  $Q$  that is written by  $T_j$ . Thus, the order of  $I_i$  and  $I_j$  matters.
- 3)  $I_i = \text{write}(Q)$ ,  $I_j = \text{write}(Q)$ . The order of  $I_i$  and  $I_j$  matters for reasons similar to those of the previous case.
- 4)  $I_i = \text{write}(Q)$ ,  $I_j = \text{write}(Q)$ . Since both instructions are write operations, the order of these transactions does not affect either  $T_i$  or  $T_j$ . However, the value obtained by the next read( $Q$ ) instruction of  $S$  is affected. Since the result of only the latter of the two write instructions is preserved in the database. If there is no other write( $Q$ ) instruction after  $I_i$  and  $I_j$  in  $s$ , then the order of  $I_i$  and  $I_j$  directly affects the final value of  $Q$  in the database state that results from schedule  $S$ .

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

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

To illustrate the concept of conflicting instructions, we consider schedule 3, in **figure 7**. The write (**A**) instruction of  $T_1$  conflicts with the read (**A**) instruction of  $T_2$ . However, the write (**A**) instruction of  $T_2$  does not conflict with the read (**B**) instruction of  $T_1$ , because the two instructions access same data items.

Let  $I_i$  and  $I_j$  be consecutive instructions of a schedule  $S$ . If  $I_i$  and  $I_j$  are instructions of different transactions and  $I_i$  and  $I_j$  do not conflict, then we can swap the order of  $I_i$  and  $I_j$  to produce a new schedule  $S'$ . We expect  $S$  to be equivalent to  $S'$ , since all instructions appear in the same order in both schedules except for  $I_i$  and  $I_j$ , whose order does not matter.

Since the write (**A**) instruction of  $T_2$  in schedule 3 of **figure 7** does not conflict with the read (**B**) instruction of  $T_1$ , we can swap these instructions to generate an equivalent schedule, schedule 5, in **figure 8**.

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

**Figure 8** Schedule 5-schedule 3 after swapping of a pair of instructions.

Regardless of the initial system state, schedules 3 and 5 both produce the same final system state:

We continue to swap no conflicting instructions:

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

The final result of these swaps, schedule 6 of **figure 9**, is a serial schedule.

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

**Figure 9** Schedule 6 – a serial schedule that is equivalent to schedule 3.

Thus, we have shown that schedule 3 is equivalent to a serial schedule. This equivalence implies that, regardless of the initial system state, schedule 3 will produce the same final state as will some serial schedule.

If a schedule  $S$  can be transformed into a schedule  $S'$  by a series of swaps of non-conflicting instructions, we say that  $S$  and  $S'$  is conflict equivalent.

In our previous examples, schedule 1 is not conflict equivalent to schedule 2. However, schedule 1 is conflict equivalent to schedule 3, because the read (B) and write (B) instruction of  $T_1$  can be swapped with the read (A) and write (A) instruction of  $T_2$ .

The concept of conflict equivalence leads to the concept of conflict serializability. We say that a schedule  $s$  is conflict serializable if it is conflict equivalent to a serial schedule. Thus, schedule 3 is conflict serializable, since it is conflict equivalent to a serial schedule. Thus, schedule 3 is conflict serializable, since it is conflict equivalent to the serial schedule 1.

Finally, consider schedule 7 of **figure 10**; it consists of only the significant operations (that is, the read and write) of transactions  $T_3$  and  $T_4$ . This schedule is not conflict serializable, since it is not equivalent to either the serial schedule  $\langle T_3, T_4 \rangle$  or the serial schedule  $\langle T_4, T_3 \rangle$ .

$T_1$	$T_2$
read(Q)	write(A)
write(QB)	

**Figure 10** Schedule 7.

It is possible to have two schedulers that produce the same outcome, but that are not conflict equivalent. For example, consider transaction  $T_5$ , which transfers \$10 from account B to A.

Let schedule 8 be as defined in **figure 11**. We claim that schedule 8 is not conflict equivalent to the serial schedule  $\langle T_1, T_5 \rangle$ ,

$T_1$	$T_2$
read(A) A:=A-50 write(A)	read(A) B:=B-10 write(B)
read(B) B:=B+50 Write(B)	
	read(B) A:=A+10 write(A)

**Figure 11** Schedule 8.

Since, in schedule 8, the write (**B**) instruction of  $T_5$  conflicts with the read (**B**) instruction of  $T_1$ . Thus, we cannot move all the instructions of  $T_1$  before those of  $T_5$  by swapping consecutive nonconflicting instructions. However, the final values of accounts **A** and **B** after the execution of either schedule 8 or the serial schedule  $\langle T_1, T_5 \rangle$  are the same  $-\$960$  and  $\$2040$  respectively.

We can see from this example that there are less stringent definitions of schedule equivalence than conflict equivalence. For the system to determine that schedule 8 produces the same outcome as the serial schedule  $\langle T_1, T_5 \rangle$ , it must analyze the computation performed by  $T_1$  and  $T_2$ , rather than just the read and write operations. In general, such analysis is hard to implement and is computationally expensive. However, there are other definitions of schedule equivalence based purely on the read and write operations. We will consider one such definition in the next section.

### 17.5.2 view serializability

In this section, we consider a form of equivalence that is less stringent than conflict equivalence, but that, like conflict equivalence, is based on only the read and write operations of transactions. Consider two schedules **S** and **S'**, where the same set of transactions participates in both schedules. The schedules **S** and **S'** are said to be view equivalent if three conditions are met.

1. For each data item **Q**, if transaction **T**, reads the initial value of **Q** in schedule **s**, then transaction **T**, must, in schedule **S'**, also read the initial value of **Q**.
2. For each data item **Q**, if transaction **T**, executes read (**Q**) in schedule **S**, and if that value was produced by a write (**Q**) operation executed by transaction  $T_3$ , then the read (**Q**) operation of transaction  $T_1$  must, in schedule **s**, also read the value of **Q** that was produced by the same write (**Q**) operation of transaction  $T_1$ .
3. For each data item **Q**, the transaction (if any) that performs the final write (**Q**) operation in schedule **s** must perform the final write (**Q**) operation in schedule **s'**.

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

In our previous examples, schedule 1 is not view equivalent to schedule 2, since, in schedule 1, the value of account **a** read by transaction  $T_2$  was produced by  $T_1$ , whereas this case does not hold in schedule 2. However, schedule 1 is view equivalent to schedule 3, because the values of account **A** and **B** read by transaction  $T_2$  were produced by  $T_1$  in both schedules.

The concept of view equivalence leads to the concept of view serializability. We say that a schedule **s** is view serializable if it is view equivalent to a serial schedule.

As an illustration, suppose that we augment schedule 7 with transaction  $T_6$ , and obtain schedule 9 in **figure 12**. Schedule 9 is view serializable. Indeed, it is view equivalent to the serial schedule  $\langle T_3, T_4, \text{ and } T_6 \rangle$ , since the one read (**Q**) instruction reads the initial value of **Q** in both schedules, and to  $T_6$  performs the final value of **Q** in both schedules.

	$T_3$	$T_4$	$T_6$
read (Q)			
write (Q)		write (Q)	
			write(Q)

**Figure 12** Schedule 9-a view-serializable schedule

Every conflict serializable schedule is also view serializable, but there are view serializable schedules that are not conflict serializable, since every pair of consecutive instructions conflicts, and, thus, no swapping of instructions is possible.

Observe that, in schedule 9, transactions  $T_4$  and  $T_6$  perform write (Q) operations without having performed a read (Q) operation. Writes of this sort are called blind writes. Blind writes appear in any view-serializable schedule that is not conflict serializable

## 17.6 Recoverability

So far, we have studied what schedules are acceptable from the viewpoint of consistency of the database, assuming implicitly that there are no transaction failures. We now address the effect of transaction failures during concurrent execution.

If a transaction  $T_i$  fails, for whatever reason, we need to undo the effect of this transaction to ensure the atomic property of the transaction. In a system that allows concurrent execution, it is necessary also to ensure that any transactions  $T_j$  that is dependent on  $T_i$  is also aborted. To achieve this surety, we need to place transactions on the type of schedules permitted in the system

### 17.4.1 Recoverable schedules

Consider schedule 11 in **figure 13** in which  $T_9$  is a transaction that performs only one instruction: read (A). Suppose that the system allows  $T_9$  to commit immediately after executing the read (A) instruction. Thus,  $T_9$  commits before  $T_8$  does. Now, suppose that  $T_8$  fails before it commits. Since  $T_9$  has read the value of data item a written by  $T_8$ , we must abort  $T_9$  to ensure transaction atomicity. However,  $T_9$  has already committed and cannot be aborted. Thus, we have a situation where it is impossible to recover correctly from the failure of  $T_8$ .

$T_8$	$T_9$
read(A)	
write(A)	read(A)
read(B)	

**Figure 13** Schedule 10.

Schedule 10, with the commit happening immediately after the read (A) instruction, is an example of a non-recoverable schedule, which should not be allowed. Most database system requires that all schedules be recoverable. A Recoverable schedule is one where, for each pair of transactions  $T_i$  and  $T_j$  such that  $T_j$  reads a data item previously written by  $T_i$ , the commit operation of  $T_i$  appears before the commit operation of  $T_j$ ;

### 17.4.1 Cascades Schedules

Even if a schedule is recoverable, to recover correctly from the failure of a transaction  $T_i$ , we may have to roll back several transactions. Such situations occur if transactions have read data written by  $T_i$ . As an illustration, consider the partial schedule of **figure 14**.



$T_{10}$	$T_{11}$	$T_{12}$
read(A) read(B) write(A)	read(A) write(A)	read(A)

**Figure: 14** Schedule 11.

Transaction  $T_{10}$  writes a value of **A** that is read by transaction  $T_{11}$ . Transaction  $T_{11}$  writes a value of **A** that is read by transaction  $T_{12}$ . Suppose that, at this point,  $T_{10}$  fails.  $T_{10}$  must be rolled back. Since  $T_{12}$  is dependent on  $T_{11}$ ,  $T_{12}$  must be a rolled back. This phenomenon, in which a single transaction failure leads to a series of transaction rollbacks, is called cascading rollback.

Cascading rollback is undesirable, since it leads to the undoing of a significant amount of work. It is desirable to restrict the schedules to those where cascading rollbacks cannot occur. Such schedules are called cascade less schedules. Formally, a cascade less schedule is one where, for each pair of transactions  $T_i$  and  $T_j$  appear before the read operation of  $T_j$ . It is easy to verify that every cascadeless schedule is also recoverable.

**17.7 Implementation of isolation**

So far, we have seen what properties a schedule must have if it is to leave the database in a consistent state and allow transaction failures to be handled in a safe manner. Specifically, schedules that are conflict or view serializable and cascadeless satisfy these operations.

There are various concurrency- control schemes that we can use to ensure that, even when multiple transactions are executed concurrently, only acceptable schedules are generated, regardless of how the operating-system time –shares resources among the transactions.

As a trivial example of a concurrency control scheme, consider this scheme; a transaction acquires a lock on the entire database before it starts and releases the lock after it has committed. While a transaction holds a lock, no other transaction is allowed to acquire the lock, and all must therefore wait for the lock to be released. As a result of the locking policy, only one transaction can execute at a time. Therefore, only serial schedules are generated. These are trivially serializable, and it is easy to verify that they are cascadeless as well.

A concurrency control scheme such as this one leads to poor performance, since it forces transactions to wait for preceding transactions to finish before they can start. In, other words, it provides a poor degree of concurrency. As explained in sec 15.4 concurrent execution has several performance benefits.

The goal of concurrency control schemes is to provide a high degree of concurrency, while ensuring that all schedules that can be generated are conflict or view serializable, and are cascadeless.

## 17.8 Transaction Definition in SQL

A data-manipulation language must include a construct for specifying the set of actions that constitute a transaction. The SQL standard specifies that a transaction begin implicitly. Transactions ended by one of these SQL statements.

**Commit work** commits the current transaction and begins a new one.

**Rollback work** causes the current transactions to abort.

The keyword work is optional in both the statements. If a program terminates with out either of these commands, the updates are either committed or rolled back which of the two happens is not specified by the standard and depends on the implementation.

The standard also specifies that the system must ensure both serializability and freedom from cascading rollback. The definition of serializability used by the standard is that a schedule must have the same effect, as would some serial schedule. Thus, conflict and view serializability are both acceptable.

This SQL-92 standard also allows a transaction to specify that it may be executed in a manner that causes it to become nonserializable with respect to other transactions

## 17.9 Testing for serializability

When designing concurrency control schemes, we must show that schedules generated by the scheme are serializable. To do that, we must first understand how to determine, given a particular schedule  $s$ , whether the schedule is serializable.

We not present a simple and efficient method for determining conflict serializability of a schedule. Consider a schedule  $S$ . we construct a directed graph, called a precedence graph, from sties graph consists of a pair  $G=(V, E)$ , where  $V$  is a set of vertices and  $E$  is a set of edges. This set of vertices consists of all the transactions participating in the schedule. The set of edges consists of all edges  $T_i \rightarrow T_j$  for which one of three condition blocks.

- 1)  $T_i$  executes write ( $Q$ ) before  $T_j$  executes read ( $Q$ )
- 2)  $T_i$  executes read ( $Q$ ) before  $T_j$  executes write ( $Q$ )
- 3)  $T_i$  executes write ( $Q$ ) before  $T_j$  executes write ( $Q$ )

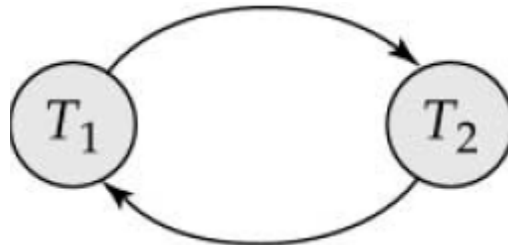
In an edge  $T_i \rightarrow T_j$  exists in the precedence graph, and then in any serial schedule  $s_1$  equivalent to  $s$ ,  $T_i$  must appear before  $T_j$ .

For example, the precedence graph for schedule 1 in **figure 15a** contains the single edge  $T_1 \rightarrow T_2$ , since all the instructions of  $T_1$  are executed before the first instruction of  $T_2$  is executed



**Figure 15** Precedence graph for (a) schedule 1 and (b) schedule 2.

Similarly, **Figure 15b** shows the precedence graph for schedule 2 with the single edge  $T_2 \rightarrow T_1$ , since all transactions of  $T_2$  are executed before the first instruction of  $T_1$  is executed. The precedence of graph for schedule 4 appears in **figure 16**.

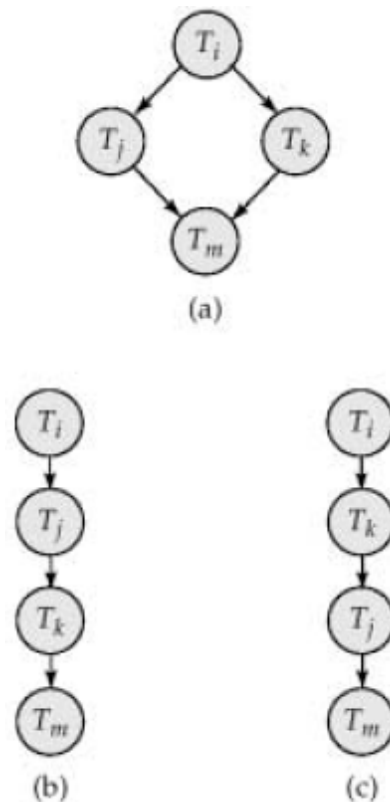


**Figure 16** Precedence graph for schedule 4.

It contains the edge  $T_1 \rightarrow T_2$ , because  $T_1$  executes read (A) before  $T_2$  executes write (A). It also contains the edge  $T_2 \rightarrow T_1$ , because  $T_2$  executes read (B) before  $T_1$  executes Write (B).

If the precedence graph for  $s$  has a cycle, then schedule  $s$  is not conflict serializable. If the graph contains no cycles, then the schedule  $s$  is not serializable.

A serializability order of the transactions can be obtained through topological sorting, which determines a linear order consistent with the partial order of the precedence graph. There are, in general, several possible linear orders that can be obtained through topological sorting. For example, the graph of **figure 17a** has the two acceptable linear orderings shown in **figures 17b** and **17c**.



**Figure 17.** Illustration of topological sorting.

Thus, to test for conflict serializability, we need to construct the precedence graph and to invoke a cycle detection algorithm. Cycle-detection algorithms can be found in standard textbooks on algorithms. Cycle-detection algorithms, such as those based on depth-first search, require on the order of  $n^2$  operations, where  $n$  is the number of vertices in the graph. Thus, we have a practical scheme for determining conflict serializability.

Returning to our previous examples, note that the precedence graphs for schedules 1 and 2 indeed do not contain cycles. The precedence graph for schedule 4, on the other hand, contains a cycle, indicating that this schedule is not conflict serializable.

Testing for view serializability is rather complicated. In fact, it has been shown that the problem of testing, for view serializability is itself NP-complete. Thus, almost certainly there exists no efficient algorithm to test for view serializability. See the bibliographical notes for references on testing for view serializability. However, concurrency-control schemes can still use sufficient conditions for view serializability. That is, if the sufficient conditions are satisfied, the schedule is view serializable, but there may be view-serializable schedules that do not satisfy the sufficient condition.

## 17.10 Summary

A transaction is a unit of program execution that accesses and possibly updates various data items. Understanding the concept of a transaction is a critical for understanding and implementing updates of data in a database in such a way those concurrent executions and failures of various forms do not result in the database becoming inconsistent.

Transactions are required to have the **ACID** properties: atomicity, consistency, isolation, and durability.

Concurrent execution of transaction improves throughput to transactions and system utilization, and also reduces waiting time of transaction. When several transactions execute concurrently in the database, the consistency of data may no longer be preserved. It is therefore necessary for the system to control the interaction among the concurrent transactions.

Serializability of schedules generated by concurrently executing transactions can be ensured through one of a variety of mechanisms called concurrency-control schemes. Schedules must be recoverable, to make sure that if transaction **A** sees the effects of transaction **B**, and **B** then aborts, then **B** also gets aborted.

The concurrency-control-management component of the database is responsible for handling the concurrency-control schemes. The recovery-management component of a database responsible for ensuring the atomicity and durability properties of transactions.

## 17.11 Technical Terms

**1. Transaction:** The term transaction refers to a collection of operations that form a single logical unit of work. For instance, transfer of money from one account to another is a transaction consisting of two updates, one to each account.

**2. Concurrent Execution:** We say that two programs are executed concurrently when they are in effect executed simultaneously. This can be accomplished by actually executing them simultaneously, or by interleaving the actions of one with the actions of the other.

**3. Recoverability:** The measure of ease and time to repair facilities to operational status.

**4. Topological sorting:** In graph theory, a topological sort of a directed acyclic graph (DAG) is a linear ordering of the nodes of the graph such that  $x$  comes before  $y$  if there's a directed path from  $x$  to  $y$  in the DAG. An equivalent definition is that each node comes before all nodes to which it has edges. Any DAG has a topological sort, and in fact most have many.

**5. Precedence graph:** A way of representing the order constraints among a collection of statements. The nodes of the graph represent the statements, and there is a directed edge from node  $A$  to node  $B$  if statement  $A$  must be executed before statement  $B$ . A precedence graph with a cycle represents a collection of statements that cannot be executed without deadlock.

**6. Lock:** In computer science, a lock is a mechanism for enforcing limits on access to a resource in an environment where there are many threads of execution. Locks are one way of enforcing concurrency control policies.

**7. Isolation:** In database systems, isolation is a property that the changes made by an operation are not visible to other simultaneous operations on the system until its completion. This is one of the ACID properties.

**8. Durability:** In computer science, durability is the ACID property that guarantees that transactions that are successfully committed will survive permanently and will not be undone by system failure.

## 17.12 Model Questions

1. List the ACID properties. Explain the usefulness of each.
2. Suppose that there is a database system that never fails. Is a recovery manager required for this system?
3. Explain the distinction between the terms *serial schedule* and *serializable schedule*.
4. During its execution, a transaction passes through several states, until it finally commits or aborts. List all possible sequences of states through which a transaction may pass.
5. What is a recoverable schedule? Why is recoverability of schedules desirable?

## 17.13 References

"Database System Concepts", 4<sup>th</sup> edition by Abraham Silberschatz, Henry F.Korth and S.Sudarshan.

Fundamentals of database systems, 4<sup>th</sup> edition by R. Elmasri and B. Navathe

### AUTHOR:

**M.V.BHUVANGA RAO M.C.A.,**  
**Lecturer,**  
**Dept. Of Computer Science,**  
**JKC College,**  
**GUNTUR**

## Lesson 18

# Concurrency Control

### 18.0 Objectives:

After completion of this lesson the student will be able to know about:

- Various modes in locking.
- Constraints for granting a lock.
- Using Two-Phase locking protocol for ensuring serializability.
- How locking system is maintained.
- Various models that are used to acquire information for the protocols other than two phase.

### Structure Of the Lesson:

#### 18.1 Lock-Based Protocols

#### 18.2 Summary

#### 18.3 Technical terms

#### 18.4 Model Questions

#### 18.5 References

### 18.1 Lock-Based Protocols

The most common method used to implement this requirement is to allow a transaction to access a data item only if it is currently holding a lock on that item.

#### 18.1.1 Locks

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

- **Shared:** If a transaction  $T_i$  has obtained a **shared-mode lock** (denoted by **S**) on item **Q**, then  $T_i$  can read, but cannot write, **Q**.
- **Exclusive:** If a transaction  $T_i$  has obtained an **exclusive-mode lock** (denoted by **X**) on item **Q**, then  $T_i$  can both read and write **Q**.

We require that every transaction request a lock in an appropriate mode on data item **Q**, depending on the types operations that it will perform on **Q**. The transactions make the request to the concurrency-control manager. The transactions can proceed with the operation only after the concurrency-control manager grants the lock to the transaction.

Given a set of lock modes; we can define a compatibility function on them as follows. Let **A** and **B** represent arbitrary lock modes. Suppose that a transaction  $T_i$  request a lock of mode **A** on item **Q** on which transaction  $T_j$  ( $T_i \neq T_j$ ) currently holds a lock of mode **B**. If transaction  $T_i$  can be granted a lock on **Q** immediately, in spite of the presence of the mode **B** lock, then we say mode **A** is compatible with mode **B**, such a function can be represented conveniently by a matrix.

The compatibility relation between the two modes of locking discussed in this section appears in the matrix comp of **fig.1**. An element **comp (A, B)** of the matrix has the value true if and only if mode **A** is compatible with mode **B**

	<b>S</b>	<b>X</b>
<b>S</b>	<b>True</b>	<b>False</b>
<b>X</b>	<b>False</b>	<b>False</b>

**Figure: 1 Lock- Compatibility Matrix Comp.**

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

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

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

Let us take banking system is an example in that system, **A** and **B** is two accounts that are accessed by transaction  $T_1$  and  $T_2$ . Transaction  $T_1$  transfers \$50 from account **B** to account **A** (**Fig.2**).

```

T1:  lock -X (B);
       Read (B);
       B: =b-50;
       Write (B);
       Unlock (B);
       Lock-X (A);
       Read (A);
       A: =A+50;
       Write (A);
       Unlock (A).

```

**Figure: 2 Transaction  $T_1$ .**

Transaction  $T_2$  displays the total amount of money in accounts **A** and **B**, i.e., the sum  $A+B$  (**Fig.3**).

$T_2$ :  
 lock-S (A);  
 read (A);  
 unlock (A);  
 lock-S (B);  
 read (B);  
 unlock (B);  
 display (A+B);

**Figure: 3 Transaction  $T_2$ .**

Suppose that the values of the accounts **A** and **B** are \$100 and \$200, respectively. If these two transactions are executed in the order  $T_1, T_2$  or the order  $T_2, T_1$  then transaction  $T_2$  will display the value \$300. If, however, these transactions are executed concurrently, then schedule1, in **Fig.4** is possible. In this case, transaction  $T_2$  displays \$250, which is incorrect. The reason for this mistake is that the transaction  $T_1$  unlocked data item **B** too early, as a result of which  $T_2$  saw an inconsistent state.

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

**Figure: 4 Schedule 1**



Suppose now that unlocking is delayed to the end of the transaction. Transaction  $T_3$  corresponds to  $T_1$  with unlocking delayed (**Fig.5**).

```

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

```

**Figure: 5 Transaction  $T_3$ .**

Transaction  $T_4$  corresponds to  $T_2$  with unlocking delayed (**Fig.6**).

```

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

```

**Figure: 6 Transaction  $T_4$ .**

You should verify that the sequence of reads and writes in schedule 1, which leads to an incorrect total of \$250 being displayed, is no longer possible with  $T_3$  and  $T_4$ . Other schedules are possible,  $T_4$  will not print out an inconsistent result in any of them.

$T_3$	$T_4$
lock -X(B) read (B) B: =B-50 write (B)	lock-S(A) read (A) lock-S(B)
lock-X (A)	

**Figure: 7 Schedule 2.**

Unfortunately, locking can lead to an undesirable situation. Consider the partial schedule of **Fig.7** for  $T_3$  and  $T_4$ . Since  $T_3$  is holding an exclusive-mode lock on **B** and  $T_4$  is requesting a shared – mode lock on **B**,  $T_4$  is waiting for  $T_3$  to unlock **B**. Similarly, since  $T_4$  is holding a shared-mode lock on **A** and  $T_3$  is requesting an exclusive –mode lock on **A**,  $T_3$  is waiting for  $T_4$  to unlock **A**. Thus, we have arrived at a state where neither of these transactions can ever proceed with its normal execution. This situation is called **Deadlock**.

When deadlock occurs, the system must roll back one of the two transactions. Once a transaction has been rolled back, the data items that were locked by that transaction those are unlocked. These data items are then available to other transaction, which can continue with its execution.

We shall require that each transaction in the system follow a set of rules, called a **locking protocol**, indicating when a transaction may lock and unlock each of the data items. Locking protocols restrict the number of possible schedules. The set of all such schedules is a proper subset of all possible serializable schedules.

### 18.1.2 Granting Of Locks

When a transaction requests a lock on a data item in a particular mode, and no other transaction has a lock on the same data item in a conflicting mode, the lock can be granted. However, care must be taken to avoid the following scenario. Suppose a transaction  $T_2$  has a shared-mode lock on a data item and another transaction  $T_1$  requests an exclusive-mode lock on the data item. Clearly,  $T_1$  has to wait for  $T_2$  to release the shared-mode lock. Meanwhile, a transaction  $T_3$  may request a shared-mode lock on the same data item. The lock request is compatible with the lock granted to  $T_2$ , so  $T_3$  may be granted the shared-mode lock. At this point  $T_2$  may release the lock, but still  $T_1$  has to wait for  $T_3$  to finish. But again, there may be a new transaction  $T_4$  that requests a shared-mode lock on the same data item, and is granted the lock before  $T_3$  releases it. In fact, it is possible that there is a sequence of transactions that each requests a shared-mode lock on the data item, and each transaction releases the lock a short while after it is granted, but  $T_1$  never gets the exclusive-mode lock on the data item. The transaction  $T_1$  may never progress, and is said to be **starved**.

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

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

Thus, a lock request that is made later will never block a lock request.

### 18.1.3 The Two-Phase Locking Protocol

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

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

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

For example, transactions  $T_3$  and  $T_4$  are two phases. On the other hand, transactions  $T_1$  and  $T_2$  are not two phases. Note that the unlock instructions do not need to appear at the end of the transaction. For example, in the case of transaction  $T_3$ , we could move the **unlock(B)** instruction to just after the **lock-X(A)** instruction, and still retain the two-phase locking property.

We can show that the two-phase locking protocol ensures conflict serializability. Consider any transaction; the point in the schedule where the transaction has obtained its final lock (the end of its growing phase) is called the **lock point** of the transaction.

Two-phase locking does not ensure freedom from deadlock. Observe that transactions  $T_3$  and  $T_4$  are two phases, but, in schedule 2 (**Fig.7**), they are deadlocked.

Cascading rollback may occur under two-phase locking. As an illustration, consider the partial schedule of **fig.8**.

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

**Figure: 8 Partial schedules under two-phase locking.**

Each transaction observes the two-phase locking protocol, but the failure of  $T_5$  after the read (A) step of  $T_7$  leads to cascading rollback of  $T_6$  and  $T_7$ .

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

Another variant of two-phase locking is the **rigorous two-phase locking protocol**, which requires that all locks be held until the transaction commits. We can easily verify that, with **rigorous two-phase locking**, transactions can be serialized in the order in which they commit. Most database systems implement either strict or rigorous two-phase locking.

Consider the following two transactions, for which we have shown only some of the significant, read and write operations:

```

T8: read(a1);
      read (a2);
      ...
      read(an);
      write(a1);

T9: read(a1);
      read(a2);
      display(a1+a2).

```

If we employ the two-phase locking protocol, then  $T_8$  must lock  $a_1$  in exclusive mode. Therefore, any concurrent execution of both transactions amounts to a serial execution. Notice, however, that  $T_8$  needs an exclusive lock on  $a_1$  in only at the end of its execution, when it writes  $a_1$ . Thus, if  $T_8$  could initially lock  $a_1$  in shared mode, and then could later change the lock to exclusive mode, we could get more concurrency, since  $T_8$  and  $T_9$  could access  $a_1$  and  $a_2$  simultaneously.

This observation leads us to a refinement of the basic two-phase locking protocol, in which **lock conversions** are allowed. We shall provide a mechanism for upgrading a shared lock to an exclusive lock, and downgrading an exclusive lock to a shared lock. We denote conversion from shared to exclusive modes by **upgrade**, and from exclusive to shared by **downgrade**. Lock conversion cannot be allowed arbitrarily. Rather, upgrading can take place in only the growing phase, whereas downgrading can take place in only the shrinking phase.

Returning to our example, transactions  $T_8$  and  $T_9$  can run concurrently under the refined two-phase locking protocol, as shown in the incomplete schedule of **Fig.9**, where only some of the locking instructions are shown.

$T_8$	$T_9$
lock-S( $a_1$ )	
lock-S( $a_2$ )	lock-S( $a_1$ )
lock-S( $a_3$ )	lock-S( $a_2$ )
lock-S( $a_4$ )	
	unlock ( $a_1$ )
	unlock ( $a_2$ )
lock-S( $a_n$ )	
upgrade( $a_1$ )	

**Figure: 9 Incomplete Schedules with a Lock conversion.**

Note that a transaction attempting to upgrade a lock on an item  $Q$  may be forced to wait. This enforced wait occurs if  $Q$  is currently locked by *another* transaction in shared mode.

Just like the basic two-phase locking protocol, two phase locking and lock conversion generates only conflict-serializable schedules and their lock points can serialize transactions. Further, if exclusive locks are held until the end of the transaction, the schedules are cascade less.

For a set of transactions, there may be conflict-serializable schedules that cannot be obtained through the two-phase locking protocol. However, to obtain conflict-serializable schedules through non-two-phase locking protocols, we need either to add additional information about the transactions or to impose some structure or ordering on the set of data items in the database. In the absence of such information, two-phase locking is necessary for conflict serializability—if  $T_i$  is a non—two—phase transaction, it is always possible to find another transaction  $T_j$  that is two-phases that there is a schedule possible for  $T_i$  and  $T_j$  that is not conflict serializable.

Strict two-phase locking and rigorous two-phase locking (with lock conversions) are used extensively in commercial database systems.

A simple but widely used scheme automatically generates the appropriate lock and unlocks instructions for a transaction, on the basis of read and writes requests from the transaction:

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

#### 18.1.4 Implementation of Locking

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

The lock manager uses this data structure: For each data item that is currently locked, it maintains a linked list of records, one for each request, in the order in which the requests arrived. It uses a hash table, indexed on the name of data item, to find the linked list (if any) for a data item; this table is called the **lock table**. Each record of the linked list for a data item notes which transaction made the request, and what lock mode it requested. The record also notes if the request has currently been granted.

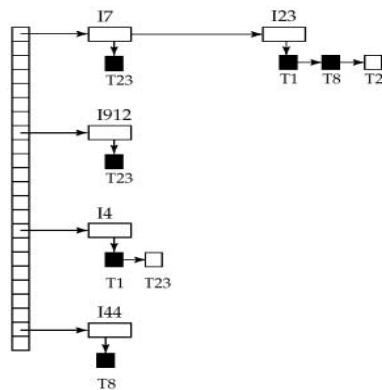


Figure: 10 Lock Table

Fig.10 shows an example of a lock table. The table contains locks for five different data items I4, I7, I23, I912. The lock table uses overflow of chaining, so there is a linked list of data items for each entry in the lock table. There is also a list of transactions that have been granted locks, or are waiting for locks, for each of the data items. Granted locks are the filled-in (black) rectangles, while waiting requests are the empty rectangles. We have omitted the lock mode to keep the figure simple. It can be seen, for example, that T23 has been granted locks on I912 and I7, and is waiting for a lock on I4.

Although the figures does not show it, the lock table should also maintain an index on transaction identifiers, so that it is possible to determine efficiency the set of locks held by a given transaction.

The lock manager processes requests this way:

- When a lock request message arrives, it adds a record to the end of the linked list for the data item, if the linked list is present. Otherwise it creates a new linked list, containing only the record for the request.

It always grants the first lock request on a data item. But if the transaction requests a lock on an item on which a lock has already been granted, the lock manager grants the request only if it is compatible with all earlier requests, and all earlier requests have been granted already. Otherwise the request has to wait.

- When the lock manager receives an unlock message from a transaction, it deletes the record for that data item in the linked list corresponding to that transaction. It tests the record that flows, if any, as described in the previous paragraph, to see if that request can now be granted. If it can, the lock manager grants that request, and processes the record following it, if any, similarly, and so on.
- If a transaction aborts, the lock manager deletes any waiting request made by the transaction. Once the database system has taken appropriate actions to undo the transaction, it releases all locks held by the aborted transaction.

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

### 18.1.5 Graph-Based Protocols

The two-phase locking protocol is both necessary and sufficient for ensuring serializability in the absence of information concerning the manner in which data items are accessed. But, if we wish to develop protocols that are not two-phase, we need additional information on how each transaction will access the database. There are various models that can give us the additional information, each differing in the amount of information provided. The simplest model requires that we have prior knowledge about the order in which the database items will be accessed. Given such information, it is possible to construct locking protocols that are not two-phase, nevertheless, ensure conflict serializability.

To acquire such prior knowledge, we impose a partial ordering  $\rightarrow$  on the set  $D = \{d_1, d_2 \dots d_n\}$  of all data items. If  $d_i \rightarrow d_j$ , then any transaction accessing both  $d_i$  and  $d_j$  must access  $d_i$  before accessing  $d_j$ . This partial ordering may be the result of either the logical or the physical organization of the data, or it may be imposed solely for the purpose of concurrency control.

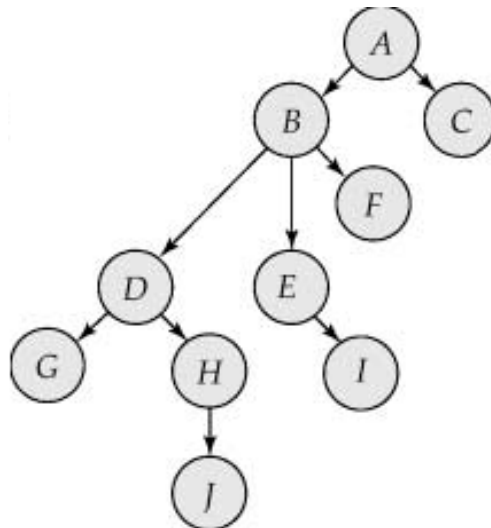
The partial ordering implies that the set  $D$  may now be viewed as a directed acyclic graph, called a **database graph**. We will present a simple protocol, called the tree protocol, which is restricted to employ only exclusive locks. References to other, more complex, graph-based locking protocols are in the bibliographical notes.

In the **tree protocol**, the only lock instruction allowed is lock-X. Each transaction  $T_i$  can lock a data item at most once, and must observe the following rules:

1. The first lock by  $T_i$  may be on any data item.
2. Subsequently, a data item  $Q$  can be locked by  $T_i$  only if the parent of  $Q$  is currently locked by  $T_i$ .
3. Data items may be unlocked at any time.
4. A data item that has been locked and unlocked by  $T_i$  cannot subsequently be relocked by  $T_i$ .

All schedules that are legal under the tree protocol are conflict serializable.

To illustrate this protocol, consider the database graph of **Fig.11**.



**Figure: 11 Tree-structured database graph.**

The following four transactions follow the tree protocol on this graph. We show only the lock and unlock instructions:

$T_{10}$ : lock-X(B);lock-X(D);unlock(B);unlock(E);lock-X(G);

unlock(D);unlock(G)

$T_{11}$ : lock-X (D); lock-X (H); unlock (D); unlock (H).

$T_{12}$ : lock-X (B); lock-X (E); unlock (E); unlock (B).

$T_{13}$ : lock-X (D); lock-X (H); unlock (D); unlock (H).

One possible schedule in which these four transactions participated appears in **Fig.12**. Note that, during its execution, transaction  $T_{10}$  holds locks on two *disjoint* sub trees.

Observe that the schedule of **Fig.12** is conflict serializable. It can be shown not only that the tree protocol ensures conflict serializability also this protocol ensures freedom from deadlock.

$T_{10}$	$T_{11}$	$T_{12}$	$T_{13}$
lock-X(B)	lock-X(D) lock-X(H) unlock-X(D)		
lock-X(E) lock-X(D) unlock(B) unlock(E)		lock-X(B) lock-X(E)	
lock-X(G) unlock(D)	unlock(H)		
	unlock(E) unlock(B)		lock-X(D) lock-X(H) unlock(D) unlock(H)
unlock (G)			

**Figure: 12 Serializable schedule under the tree protocol.**

The tree protocol in **Fig.12** does not ensure recoverability and cascadelessness. To ensure recoverability and cascadelessness, the protocol can be modified not to permit release of exclusive locks until the end of the transaction reduces concurrency. Here is an alternative that improves concurrency but ensures only recoverability: For each data item with an uncommitted write we record which transaction performed the last write to the data item. Whenever a transaction  $T_i$  performs a read of an uncommitted data item, we record a commit dependency of  $T_i$  on the transaction that performed the last write to the data item. Transaction  $T_i$  is then not permitted to commit until the commit of all transactions on which it has a commit dependency. If any of these transactions aborts,  $T_i$  must also be aborted.

The tree-locking protocol has an advantage over the two-phase locking protocol in that, unlike two-phase locking, it is deadlock-free, so no rollbacks are required. The tree-locking protocol has another advantage over the two-phase locking protocol in that unlocking may occur earlier. Earlier unlocking may lead to shorter waiting times, and to an increase in concurrency.

However, the protocol has the disadvantage that, in some cases a transaction may have to lock data items that it does not access. For example, a transaction that needs to access data items **A** and **J** in the database graph of **figure 11** must lock not only **A** and **J**, but also data items **B**, **D**, and



**H.** This additional locking results in increased locking overhead, the possibility of additional waiting time, and a potential decrease in concurrency. Further, without prior knowledge of what data items will need to be locked, transactions will have to lock the root of the tree, and that can reduce concurrency greatly.

For a set of transactions, there may be conflict-serializable schedules that cannot be obtained through the tree protocol. Indeed, there are schedules possible under the two-phase locking protocols that are not possible under the tree protocol, and vice versa.

## 18.2 Summary

When several transactions execute concurrently in the database, the consistency of data may no longer be preserved. It is necessary for the system to control the interaction among the concurrent transactions, and this control is achieved through one way of a variety of mechanisms called concurrency-control schemes.

To ensure serializability, we can use various concurrency control schemes. All these schemes either delay an operation or abort the transaction that issued the operation.

A locking protocol is a set of rules that state when a transaction may lock and unlock each of the data items in the database.

The two-phase locking protocol allows a transaction to lock a new data item only if that transaction has not yet unlocked any data item. The protocol ensures serializability, but not deadlock freedom. In the absence of information concerning the manner in which data items are accessed, the two-phase locking protocol is both necessary and sufficient for ensuring serializability.

## 18.3 Technical Terms

**Concurrency Control:** concurrency control is a method used to ensure that database transactions are executed in a safe manner (i.e., without data loss). Concurrency control is especially applicable to database management systems, which must ensure that transactions are executed safely and that they follow the ACID rules.

**Shared-mode (S) Locks:** No other transactions can modify the data while shared (S) locks exist on the resource. Shared (S) locks on a resource are released as soon as the read operation completes, unless the transaction isolation level is set to repeatable read or higher, or a locking hint is used to retain the shared (S) locks for the duration of the transaction.

**Exclusive (X) Locks:** Exclusive (X) locks prevent access to a resource by concurrent transactions. With an exclusive (X) lock, no other transactions can modify data; read operations can take place only with the use of the NOLOCK hint or read uncommitted isolation level.

**Starvation:** In computer science, starvation is a multitasking-related problem, where a process is perpetually denied necessary resources. Without those resources, the program can never finish its task.

**Two-phase Locking:** A mechanism for preventing deadlock; a process is blocked from allocating resources until it can get all the resources it needs.

**Strict Two-phase Protocol:** In computer science, strict two-phase locking (Strict 2PL) is a locking method used in database management systems.

## 18.4 Model Questions

1. Show that the two-phase locking protocol ensures conflict serializability, and that transactions can be serialized according to their lock point.
2. What benefit does strict two-phase locking provide? What disadvantages result?
3. What benefit does rigorous two-phase locking provide? How does it compare with other forms of two-phase locking?
4. What are the reasons for the popularity of strict two-phase locking?

## 18.5 References

Database System Concepts", 4<sup>th</sup> edition by Abraham Silberschatz, Henry F.Korth and S.Sudarshan

Fundamentals of database systems, 4<sup>th</sup> edition by R. Elmasri and B. Navathe

### **AUTHOR:**

**M.V.BHUVANGA RAO. M.C.A.,  
Lecturer,  
Dept.Of Computer Science,  
JKC College,  
GUNTUR**

## Lesson 19

# Concurrency Control-2

### 19.0 Objectives:

After completion of this lesson the student will be able to know about:

- Timestamp-ordering protocol for ensuring serializability.
- Validation-Based protocols for reducing the overhead.
- Maintaining multiple versions in Timestamp ordering and Two-phase Locking.

### Structure Of the Lesson:

- 19.1 Timestamp-Based Protocols
- 19.2 Validation-Based Protocols
- 19.3 Multiple Granularity
- 19.4 Multiversion Schemes
- 19.5 Summary
- 19.6 Technical Terms
- 19.7 Model Questions
- 19.8 References

### 19.1 Timestamp-Based Protocols

Another method for determining the serializability order is to select an ordering among transactions in advance. The most common method for doing so is to use a timestamp-ordering scheme.

#### 19.1.1 Timestamps

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

- Use the value of the system clock as the timestamp, i.e., a transaction's timestamp is equal to the value of the clock when the transaction enters the system.
- Use a logical counter that is incremented after a new timestamp has been assigned, i.e., a transaction's timestamp is equal to the value of the counter when the transaction enters the system.

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

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

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

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

### 19.1.2 The Timestamp-Ordering Protocol

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

1. Suppose that transaction  $T_i$  issues **read (Q)**.
  - a. If  $TS(T_i) < W\text{-timestamp}(Q)$ , then  $T_i$  needs to read a value of **Q** that was already overwritten. Hence, the read operation is rejected, and  $T_i$  is rolled back.
  - b. If  $TS(T_i) \geq W\text{-timestamp}(Q)$ , then the read operation is executed, and **R-timestamp(Q)** is set to the maximum of **R-timestamp(Q)** and  $TS(T_i)$ .
2. Suppose that transaction  $T_i$  issues **write (Q)**.
  - a. If  $TS(T_i) < R\text{-timestamp}(Q)$ , then the value of **Q** that is producing was needed previously, and the system assumed that value would never be produced. Hence, the system rejects the write operation and rolls  $T_i$  back.
  - b. If  $TS(T_i) < W\text{-timestamp}(Q)$ , then  $T_i$  is attempting to write an obsolete value of **Q**. Hence, the system rejects this write operation and rolls  $T_i$  back.
  - c. Otherwise, the system executes the write operation and sets **W-timestamp (Q)** to  $TS(T_i)$ .

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

To illustrate this protocol, we consider transactions  $T_{14}$  and  $T_{15}$ . Transaction  $T_{14}$  displays the contents of accounts **A** and **B**.

```
T14: read (B);
      read (A);
      display (A+B).
```

Transactions  $T_{15}$  transfer \$50 from account **A** to account **B**, and then display the contents of both:

```
T15: read (B);
      B: = B-50;
      write (B);
      read (A);
      A: = A+50;
      write (A);
      display (A+B).
```

In presenting schedules under the timestamp protocol, we shall assume that a transaction is assigned

a timestamp immediately before its first instruction.

$T_{14}$	$T_{15}$
read (B)	read (B) B:=B-50 write (B)
read (A)	read (A)
display(A+B)	A: = A+50 write (A) display (A+B)

**Figure: 1** schedule 3

Thus, in schedule 3 of **Fig.1**,  $TS(T_{14}) < TS(T_{15})$ , and the schedule is possible under the timestamp protocol.

We note that the preceding execution can also be produced by the two-phase locking protocol. There are, however, schedules that are possible under the two-phase locking protocol, but are not possible under the timestamp protocol, and vice versa.

The timestamp-ordering protocol ensures conflict serializability. This is because conflicting operations are processed in timestamp order.

The protocol ensures freedom from deadlock, since no transaction ever waits. However, there is a possibility of starvation of long transactions if a sequence of conflicting short transactions causes repeated restarting of the long transaction. If a transaction is found to be getting restarted repeatedly, conflicting transactions need to be temporarily blocked to enable the transaction need to be temporarily blocked to enable the transaction to finish.

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

- Performing all writes together at the end of the transaction can ensure recoverability and cascadelessness. The writes must be atomic in the following sense: While the writes are in progress, no transaction is permitted to access any of the data items that have been written.
- Recoverability and cascadelessness can also be guaranteed by using a limited form of locking, whereby reads of uncommitted items are postponed until the transaction that updated the item commits.
- Tracking uncommitted writes can ensure recoverability alone, and allowing a transaction  $T_i$  to commit only after the commit of any transaction that wrote a value that  $T_i$  read.

### 19.1.3 Thomas' Write Rule

We now present a modification to the timestamp-ordering protocol that allows greater potential concurrency than does the protocol of section 19.1.2. Let us consider schedule 4 of **Fig.2**, and apply the timestamp-ordering protocol.

$T_{16}$	$T_{17}$
read(Q)	write(Q)
write(Q)	

**Figure: 2 Schedule 4.**

Since  $T_{16}$  starts before  $T_{17}$ , we shall assume that  $TS(T_{16}) < TS(T_{17})$ . The **read (Q)** operation of  $T_{16}$  succeeds, as does the **write (Q)** operation of  $T_{17}$ . When  $T_{16}$  attempts its **write (Q)** operation, we find that  $TS(T_{16}) < W\text{-timestamp}(Q) = TS(T_{17})$ . Thus, the **write (Q)** by  $T_{16}$  is rejected and transaction  $T_{16}$  must be rolled back.

Although the rollback of  $T_{16}$  is required by the timestamp-ordering protocol, it is unnecessary. Since  $T_{17}$  has already written **Q**, the value that  $T_{16}$  is attempting to write is one that will never need to be read. Any transaction  $T_i$  with  $TS(T_i) < TS(T_{17})$  that attempts a **read (Q)** will be rolled back, since  $TS(T_i) < W\text{-timestamp}(Q)$ . Any transaction  $T_j$  with  $TS(T_j) > TS(T_{17})$  must read the value of **Q** written by  $T_{17}$ , rather than the value written by  $T_{16}$ .

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

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

1. If  $TS(T_i) < R\text{-timestamp}(Q)$ , then the value of **Q** that  $T_i$  is producing was previously needed, and it had been assumed that the value would never be produced. Hence, the system rejects the write operation and rolls  $T_i$  back.
2. If  $TS(T_i) < W\text{-timestamp}(Q)$ , then  $T_i$  is attempting to write an obsolete value of **Q**. Hence, this write operation can be ignored.
3. Otherwise, the system executes the write operation and sets  $W\text{-timestamp}(Q)$  to  $TS(T_i)$ .

The difference between these rules and those of section 16.2.2 lies in the second rule. The timestamp-ordering protocol requires that  $T_i$  be rolled back if  $T_i$  issues write (**Q**) and  $TS(T_i) < W\text{-timestamp}(Q)$  and  $TS(T_i) < W\text{-timestamp}(Q)$ . However, here, in those cases where  $TS(T_i) \geq R\text{-timestamp}(Q)$ , we ignore the obsolete write.

Thomas' write rule makes use of view serializability by, in effect, deleting obsolete write operations from the transactions that issue them. This modification of transactions makes it possible to generate serializable schedules that would not be possible under the other protocols presented in this chapter.

For example, schedule 4 of **Fig.2** is not conflict serializable and, thus, is not possible under any of two-phase rule, the tree protocol, or the timestamp-ordering protocol. Under Thomas' write rule, the write (**Q**) operation of  $T_{16}$  would be ignored. The result is a schedule that is view equivalent to the serial schedule  $\langle T_{16}, T_{17} \rangle$ .

## 19.2 Validation-Based Protocols

In cases where a majority of transactions are read-only transactions, the rate of conflicts among transactions may be low. Thus, many of these transactions, if executed without the supervision of a concurrency-control scheme, would nevertheless leave the system in a consistent state.

A concurrency-control scheme imposes overhead of code execution and possible delay of transactions. It may be better to use an alternative scheme that imposes fewer overheads. A difficulty in reducing the overhead is that we do not know in advance which transactions will be involved in a conflict. To gain that knowledge, we need a scheme for monitoring the system.

We assume that each transaction  $T_i$  executes in two or three different phases in the lifetime, depending on whether it is a read-only or an update transaction. The phases are, in order,

1. **Read Phase:** During this phase, the system executes transaction  $T_i$ . It reads the values of the various data items and stores them in variables local to  $T_i$ . It performs all write operations on temporary local variables, without updates of the actual database.
2. **Validation Phase:** Transaction  $T_i$  performs a validation test to determine whether it can copy to the database the temporary local variables that hold the results of write operations without causing a violation of serializability.
3. **Write Phase:** If transaction  $T_i$  succeeds in validation (step 2), then the system applies the actual updates to the database. Otherwise, the system rolls back  $T_i$ .

Each transaction must go through the three phases in order shown. However, all three phases of concurrently executing transactions can be interleaved.

To perform the validation test, we need to know when the various phases of transactions  $T_i$  took place. We shall, therefore, associate three different timestamps with transaction  $T_i$  :

1. **Start ( $T_i$ )**, the time when  $T_i$  started its execution.
2. **Validation( $T_i$ )**, the time when  $T_i$  finished its read phase and started its validation phase.
3. **Finish( $T_i$ )**, the time when  $T_i$  finished its write phase.

We determine the serializability order by the timestamp-ordering technique, using the value of the timestamp Validation ( $T_i$ ). Thus, the value  $TS(T_i) = \text{validation}(T_i)$  and, if  $TS(T_i) < TS(T_{jk})$ , then any produced schedule must be equivalent to a serial schedule in which transaction  $T_i$  appears before transaction  $T_{jk}$ . The reason we have chosen Validation ( $T_i$ ), rather than Start ( $T_i$ ), as the timestamp of transaction  $T_i$  is that we can expect faster response time provided that conflict rates among transactions are indeed low.

The validation test for transaction  $T_j$  requires that, for all transactions  $T_i$  with  $TS(T_i) < TS(T_j)$ , one of the following two conditions must hold:

1. **Finish ( $T_i$ ), Start( $T_j$ ).** Since  $T_i$  completes its execution before  $T_j$  started, the serializability order is indeed maintained.
2. The set of data items written by  $T_i$  does not intersect with the set of data items read by  $T_j$ , and  $T_i$  completes its write phase before  $T_j$  starts its validation phase (**Start ( $T_j$ ) < Finish ( $T_i$ ) < Validation ( $T_j$ )**). This condition ensures that the writes of  $T_i$  and  $T_j$  do not overlap. Since the writes of  $T_i$  do not affect the read of  $T_j$ , and since  $T_j$  cannot affect the read of  $T_i$ , the serializability order is indeed maintained.

The writes of  $T_i$  and  $T_j$  do not overlap. Since the writes of  $T_i$  do not effect the read of  $T_j$ , and since the  $T_j$  cannot affect the **read** ( $T_i$ ), the serializability order is indeed maintained. As an illustration, consider again transactions  $T_{14}$  and  $T_{15}$ .

$T_{14}$	$T_{15}$
read(B)	read (B) B:=B-50 Read(A) A:=A+50
read(A) <validate> display(A+B)	<validate> write(B) write(A)

**Figure: 3 Schedule 5, a schedule produced by using validation.**

Suppose that  $TS(T_{14}) < TS(T_{15})$ . Then, the validation phase succeeds in the schedule 5 in **fig.3**. Note that the writes to the actual variables are performed only after the validation phase of  $T_{15}$ . Thus,  $T_{14}$  reads the old values of **B** and **A**, and this schedule is serializable.

The validation scheme automatically guards against cascading rollbacks, since the actual writes take place only after the transaction issuing the write has committed. However, there is a possibility of starvation of long transactions, due to a sequence of conflicting short transactions that cause repeated restarts of the long transaction. To avoid starvation, conflicting transactions must be temporarily blocked, to enable the long transaction to finish.

This validation scheme is called the *optimistic concurrency control* scheme since transactions execute optimistically, assuming they will be able to finish execution and validate at the end. In contrast, locking and timestamp ordering are pessimistic in that they force a wait or a rollback whenever a conflict is detected, even though there is a chance that the schedule may be conflict serializable.

### 19.3 Multiple Granularity

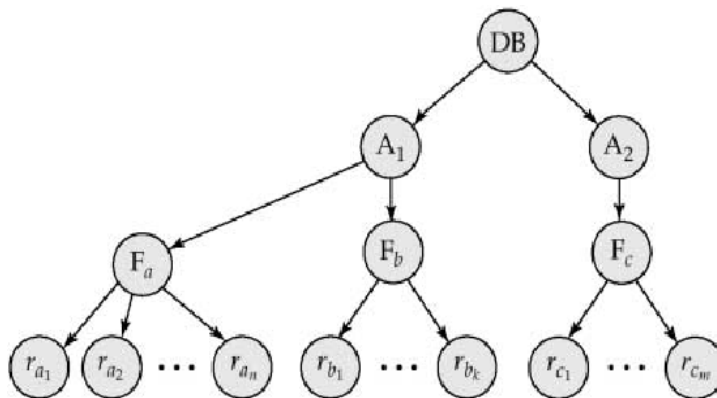
In the concurrency-control schemes described thus far, we have used each individual data item as the unit on which synchronization is performed.

There are circumstances, however, where it would be advantageous to group several data items, and to treat them as one individual synchronization unit. For example, if a transaction  $T_i$  needs to access the entire database, and a locking protocol is used, then  $T_i$  must lock each item in the database. Clearly, executing these locks is time consuming. It would be better if  $T_i$  could issue a *single* lock request to lock the entire database. On the other hand, if transaction  $T_j$  needs to access



only a few data items, it should not be required to lock the entire database, since otherwise concurrency is lost.

What is needed is a mechanism to allow the system to define multiple levels of granularity. We can make one by allowing data items to be of various sizes and defining a hierarchy of data granularities, where the small granularities are nested within larger ones. Such a hierarchy can be represented graphically as a tree. Note that the tree that we describe here is significantly different from that used by the tree protocol (**section 16.1.5**). A non-leaf node of the multiple-granularity tree represents the data associated with its descendants. In the tree protocol, each node is an independent data item.



**Figure: 4 Granularity Hierarchy.**

As an illustration, consider the tree of **Fig.4**, which consists of four levels of nodes. The highest level represents the entire database. Below it are nodes of type area; the database consists of exactly these areas. Each area in turn has nodes of type file as its children. Each area contains exactly those files that are its child nodes. No file consists of exactly those records that are its child nodes, and no record can be present in more than one file.

Each node in the tree can be locked individually. As we did in the two-phase locking protocol, we shall use shared and exclusive lock modes. When a transaction locks a node, in either shared or exclusive mode, the transaction also has implicitly locked all the descendants of that node in the same lock mode. For example, if transaction  $T_i$  gets an explicit lock on file  $F_c$  of **Fig.16.16**, in exclusive mode, then it has an implicit lock in exclusive mode all the records belonging to that file. It does not need to lock the individual records of  $F_c$  explicitly.

Suppose that transaction  $T_j$  wishes to lock record  $r_{b6}$  of file  $F_b$ . Since  $T_i$  has locked  $F_b$  explicitly, it follows that  $r_{b6}$  is also locked (implicitly). But, when  $T_j$  issues a lock request for  $r_{b6}$ ,  $r_{b6}$  is not explicitly locked! How does the system determine whether  $T_j$  can lock  $r_{b6}$ ?  $T_j$  must traverse the tree from the root to record  $r_{b6}$ . If any node in that path is locked in an incompatible mode, then  $T_j$  must be delayed.

Suppose now that transaction  $T_k$  wishes to lock the entire database. To do so, it simply must lock the root of the hierarchy. Note, however that  $T_k$  should not succeed in locking the root node, since  $T_i$  is currently holding a lock on part of the tree (specially, on file  $F_b$ ). But how does the system determine if the root node can be locked? One possibility is for it to search the entire tree. This solution, however, defeats the whole purpose of the multiple-granularity locking scheme. A more efficient way to gain this knowledge is to introduce a new class of lock modes, called intention lock

modes. If a node is locked in an intention mode, explicit locking is being done at a lower level of the tree (that is, at a finer granularity). Intention locks are put on all the ancestors of a node before that node is locked explicitly. Thus, a transaction does not need to search the entire tree to determine whether it can lock a node successfully. A transaction wishing to lock a node—say, **Q**—must traverse a path in the tree from the root to **Q**. While traversing the tree, the transaction locks the various nodes in an intention mode.

There is an intention mode associated with shared mode, and there is one with exclusive mode. If a node is locked in intention-shared (**IS**) mode, explicit locking is being done at a lower level of the tree, but with only shared-mode locks. Similarly, if a node is locked in intention-exclusive (**IX**) mode, then explicit locking is being done at a lower level, with exclusive mode or shared-mode locks. Finally, if a node is locked in shared and intention-exclusive (**SIX**) mode, the sub tree rooted by that node is locked explicitly in shared-mode, and that explicit locking is being done at a lower level with exclusive-mode locks. The compatibility function for these lock modes is in **fig.5**.

	<b>IS</b>	<b>IX</b>	<b>S</b>	<b>SIX</b>	<b>X</b>
<b>IS</b>	true	true	true	true	false
<b>IX</b>	true	true	false	false	false
<b>S</b>	true	false	true	false	false
<b>SIX</b>	true	false	false	false	false
<b>X</b>	false	false	false	false	false

**Figure: 5 Compatibility Matrix.**

The multiple-granularity locking protocol, which ensures serializability, is: each transaction  $T_i$  can lock a node **Q** by following these rules:

1. It must observe the lock-compatibility function of **Fig.5**.
2. It must lock the root of the tree first, and can lock it in any mode.
3. It can lock a node **Q** in **S** or **IS** mode only if it currently has the parent of **Q** locked in either **IX** or **IS** mode.
4. It can lock a node in **X**, **SIX**, or **IX** mode only if it currently has the parent of **Q** locked in either **IX** or **SIX** mode.
5. It can lock a node only if it has not previously unlocked any node (that is,  $T_i$  is two phase).
6. It can unlock a node **Q** only if it currently has none of the children of **Q** locked.

Observe that the multiple-granularity protocol requires that locks be acquired in top-down (root-to-leaf) order, whereas locks must be released in bottom-up (leaf-to-root) order.

As an illustration of the protocol, consider the tree of Fig.4 and these transactions:

- Suppose that transaction  $T_{18}$  reads records  $r_{a2}$  in file  $F_a$ . Then,  $T_{18}$  needs to lock the database, area  $A_1$ , and  $F_a$  in **IS** mode (and in that order), and finally to lock  $r_{a2}$  in **S** mode.

- Suppose that transaction  $T_{19}$  modifies record  $r_{ag}$  in file  $F_a$ . Then,  $T_{19}$  needs to lock the database, area  $A_1$ , and file  $F_a$  in **IX** mode, and finally to lock  $r_{ag}$  in **X** mode.
- Suppose that transaction  $T_{20}$  reads all the records in file  $F_a$ . Then,  $T_{20}$  needs to lock the database and area  $A_1$  (in that order) in **IS** mode, and finally to lock  $F_a$  in **S** mode.
- Suppose that transaction  $T_{21}$  reads the entire database. It can do so after locking the database in **S** mode.

We note that transactions  $T_{18}$ ,  $T_{20}$ ,  $T_{21}$  can access the database concurrently. Transaction  $T_{19}$  can execute concurrently with  $T_{18}$ , but not with either  $T_{20}$  or  $T_{21}$ .

This protocol enhances concurrency and reduces lock overhead. It is particularly useful in applications that include a mix of

- Short transactions that access only a few data items.
- Long transactions that produce reports from an entire file or set of files.

There is a similar locking protocol that is applicable to database systems in which data granularities are organized in the form of a directed acyclic graph. Deadlock is possible in the protocol that we have, as it is in the two-phase locking protocol. There are techniques to reduce deadlock frequency in the multiple-granularity protocol, and also to eliminate deadlock entirely

## 19.4 Multiversion Schemes

In **multiversion concurrency control** schemes, each write(**Q**) operation creates a new version of **Q**. When a transaction issues a read(**Q**) operation, the concurrency-control manager selects one of the versions of **Q** to be read. The concurrency-control scheme must ensure that the version to be read is selected in a manner that ensures serializability. It is also crucial, for performance reasons, that a transaction be able to determine easily and quickly which version of the data item should be read.

### 19.4.1 Multiversion Timestamp Ordering

The most common transaction ordering technique used by multiversion schemers is timestamping. With each transaction  $T_i$  in the system, we associate a unique static timestamp, denoted by  $TS(T_i)$ . The database system assigns this timestamp before the transaction starts execution, as described in Section 16.2.

With each data item **Q**, a sequence of versions  $\langle Q_1, Q_2 \dots Q_m \rangle$  is associated. Each version  $Q_k$  contains three data fields:

- **Content** is the value of version  $Q_k$ .
- **W-timestamp( $Q_k$ )** is the timestamp of the transaction that created version  $Q_k$ .
- **R-timestamp( $Q_k$ )** is the largest timestamp of any transaction that successfully read version  $Q_k$ .

The multiversion timestamp-ordering scheme presented next ensures serializability. The scheme operates as follows. Suppose that transaction  $T_i$  issues a read(**Q**) or write(**Q**) operation. Let  $Q_k$  denote the version of **Q** whose write timestamp is the largest write timestamp less than or equal to  $TS(T_i)$ ,

- If transaction  $T_i$  issues a **read(Q)**, then the value returned is the content of version  $Q_k$ .
- If transaction  $T_i$  issues **write(Q)**, and if  $TS(T_i) < R\text{-timestamp}(Q_k)$ , then the system rolls back transaction  $T_i$ . On the other hand, if  $TS(T_i) < W\text{-timestamp}(Q_k)$ , the system overwrites the contents of  $Q_k$ ; otherwise it creates a new version of  $Q$ .

The justification for rule 1 is clear. A transaction reads the most recent version that comes before it in time. The second rule forces a transaction to abort if it is “too late” in doing a write. More precisely, if  $T_i$  attempts to write a version that some other transaction would have read, then we cannot allow that write to succeed.

The scheme, however, suffers from two undesirable properties. First, the reading of a data item also requires the updating of the R-timestamp field, resulting in two potential disk accesses, rather than one. Second, the conflicts between transactions are resolved through rollbacks, rather than through waits. This alternative may be expensive. Section 16.5.2 describes an algorithm to alleviate this problem.

### 19.4.2 Multiversion Two-Phase Locking

The multiversion two-phase locking protocol attempts to combine the advantages of multiversion concurrency control with the advantages of two-phase locking. This protocol differentiates between read-only transactions and update transactions.

Update transactions perform rigorous two-phase locking, i.e., they hold all locks up to the end of the transaction. Thus, they can be serialized according to their commit order. Each version of a data item has a single timestamp. The timestamp in this case is not a real clock-based timestamp, but rather is a counter, which we will call the **ts-counter**, that is incremented during commit processing.

Read-only transactions are assigned a timestamp by reading the current value of ts-counter before they start execution; they follow the multiversion timestamp ordering protocol for performing reads. Thus, when a read-only transaction  $T_i$  issues a **read(Q)**, the value returned is the contents of the version whose timestamp is the largest timestamp less than  $TS(T_i)$ .

When an update transaction reads an item, it gets a shared lock on the item, and reads the latest version of that item. When an update transaction wants to write an item, it first gets an exclusive lock on the item, and then creates a new version of the data item. The write is performed on the new version, and the timestamp of the new version is initially set to a value greater than that of any possible timestamp.

When the update transaction  $T_i$  completes its actions, it carries out commit processing: First,  $T_i$  sets the timestamp on every version it has created to 1 more than the value of **ts-counter**; then,  $T_i$  increments **ts-counter** by 1. Only one update transaction is allowed to perform commit processing at a time.

As a result, read-only transactions that start after  $T_i$  increments ts-counter will see the values updated by  $T_i$ , whereas those that start before  $T_i$  increments **ts-counter** will see the value before the updates by  $T_i$ . In either case, read-only transactions never need to wait for locks. Multiversion two-phase locking also ensures that schedules are recoverable and cascadeless.

Versions are deleted in a manner like that of multiversion timestamp ordering. Suppose there are two versions,  $Q_k$  and  $Q_j$ , of data item, and both versions have a timestamp less than the timestamp of the oldest read-only transaction in the system. Then, the older of the two versions  $Q_k$  and  $Q_j$  will not be used again and can be deleted.

Multiversion two-phase locking or variations of it are used in some commercial database systems.

## 19.5 Summary

A timestamp-ordering scheme ensures serializability by selecting an ordering in advance between every pair of transactions. A unique fixed timestamp is associated with each transaction in the system. The timestamps of the transactions determine the serializability order. Thus, if the timestamp of transaction  $T_i$  is smaller than the timestamp of transaction  $T_j$ , then the scheme ensures that the produced schedule is equivalent to a serial schedule in which transaction  $T_i$  appears before transaction  $T_j$ . It does so by rolling back a transaction whenever such an order is violated.

There are circumstances where it would be advantageous to group several data items, and to treat them as one aggregate data item for purposes of working, resulting in multiple levels of granularity. We allow data items of various sizes, and define a hierarchy can be represented graphically as a tree.

Locks are acquired in root-to-leaf order; they are released in leaf-to-root order. The protocol ensures serializability; but not freedom from deadlock.

A multiversion concurrency-control scheme is based on the creation of a new version of a data item for each transaction that writes that item

## 19.6 Technical Terms

**Timestamp:** A timestamp is the current time of an event that is recorded by a computer. Through mechanisms such as the Network Time Protocol (NTP), a computer maintains accurate current time, calibrated to minute fractions of a second. Such precision makes it possible for networked computers and applications to communicate effectively.

**Timestamp-based concurrency control:** In computer science, in the field of databases, timestamp-based concurrency control is a non-lock concurrency control method, used in relational databases to safely handle transactions, using timestamps.

**System clock:** An internal time clock maintained by the operating system. This clock is primarily used to record the time when files were saved to disk.

**Locks:** Memory objects that represent the exclusive right to use a shared resource. A process that wants to use the resource requests the lock that (by agreement) stands for that resource. The process releases the lock when it is finished using the resource.

**Multiple granularity locking:** In computer science, multiple granularity locking (MGL), sometimes called the John Rayner locking method, is a locking method used in database management systems (DBMS) and relational databases.

## 19.7 Model Questions

1. When a transactions is rolled back under timestamp ordering, it is assigned a new timestamp. Why can it not simply keep its old timestamp?
2. Write a short notes on Thomas' write rule.
3. In multiple-granularity locking, what is the difference between implicit and explicit locking?
4. Although **SIX** mode is useful in multiple-granularity locking, an exclusive and intend-shared (**XIS**) mode is of no use. Why is it useless?
5. What are the advantages of multiversion two-phase locking? Justify your answer.

## 19.8 References

"Database System Concepts", 4<sup>th</sup> edition by Abraham Silberschatz, Henry F.Korth and S.Sudarshan

Fundamentals of database systems, 4<sup>th</sup> edition by R. Elmasri and B. Navathe

### AUTHOR:

**M.V.BHUJANGA RAO. M.C.A.,**  
**Lecturer,**  
**Dept.Of Computer Science,**  
**JKC College,**  
**GUNTUR**

**Wound-wait:** The **wound-wait** scheme is a preemptive technique. It is a counterpart to the wait-die scheme. When transaction  $T_i$  requests a data item currently held by  $T_j$ ,  $T_i$  is allowed to wait only if it has a timestamp larger than that of  $T_j$  (that is,  $T_i$  is younger than  $T_j$ ). Otherwise,  $T_j$  is rolled back ( $T_j$  is wounded by  $T_i$ ).

## 20.7 Model Questions

1. Under what conditions is it less expensive to avoid deadlock than to allow deadlocks to occur and then to detect them?
2. If deadlock is avoided by deadlock avoidance schemes, is starvation still possible? Explain your answer.
3. Devise a timestamp-based protocol that avoids the phantom phenomenon.
4. Explain the reasons for the use of degree-two consistencies. What disadvantages does this approach have?

## 20.8 References

Database System Concepts", 4<sup>th</sup> edition by Abraham  
Silberschatz, Henry F.Korth and S.Sudarshan  
Fundamentals of database systems, 4<sup>th</sup> edition by R. Elmasri and B. Navathe

### AUTHOR:

**M.V.BHUJANGA RAO. M.C.A.,**  
**Lecturer,**  
**Dept. Of Computer Science,**  
**JKC College,**  
**GUNTUR**

Lookup and insertion operations cannot lead to deadlock. Coalescing of nodes during deletion can cause inconsistencies, since a lookup may have read a pointer to a deleted node from its parent, before the parent node was updated, and may then try to access the deleted node. The lookup would then have to restart from the root. Leaving nodes uncoalesced avoids such inconsistencies. This solution results in nodes that contain too few search-key values and that violate some properties of  $B^+$ -trees. In most databases, however, insertions are more frequent than deletions, so it is likely that nodes that have too few search-key values will gain additional values relatively quickly.

Instead of locking index leaf nodes in a two-phase manner, some index concurrency control schemes use **key-value locking** on individual key values, allowing other key values to be inserted or deleted from the same leaf. Key-value locking thus provides increased concurrency. Using key-value locking naively, however, would allow the phantom phenomenon to occur; to prevent the phantom phenomenon, the next-key locking technique is used. In this technique, every index lookup must lock not only the keys found within the range (or the single key, in case of a point lookup) but also the next key value- that is, the key value just greater than the last key value that was within the range. Also, every insert must lock not only the value that is inserted, but also the next key value. Thus, if a transaction attempts to insert a value that was within the range of the index lookup of another transaction, the two transactions would conflict on the key value next to the inserted key value. Similarly, deletes must also lock the next key value to the value being deleted, to ensure that conflicts with subsequent range lookups of other queries are detected.

## 20.5 Summary

Various locking protocols do not guard against deadlocks. One way to prevent deadlock is to use an ordering of data items, and to request locks in a sequence consistent with the ordering.

Another way to prevent deadlock is to use preemption and transaction rollbacks. To control the preemption, we assign a unique timestamp to each transaction. The system uses these timestamps to decide whether a transaction should wait or roll back. If a transaction is rolled back, it retains its old timestamp when restarted. The wound-wait scheme is a preemption scheme.

A delete operation may be performed only if the transaction deleting the tuple has an exclusive lock on the tuple to be deleted.

Insertions can lead to the phantom phenomenon, in which an insertion logically conflicts with a query even though the two transactions may access no tuple in common.

Special concurrency-control techniques can be developed for special data structures. Often, special techniques are applied in  $B^+$ -Trees to allow greater concurrency. These techniques are nonserializable access to the  $B^+$ -tree, but they ensure that the  $B^+$ -tree is correct, and ensure that accesses to the database itself are serializable.

## 20.6 Technical Terms

**Deadlock:** A deadlock is a situation in which two computer programs sharing the same resource are effectively preventing each other from accessing the resource, resulting in both programs ceasing to function.

**Wait-die:** The wait-die scheme is a nonpreemptive technique. When transaction  $T_i$  requests a data item currently held by  $T_j$ ,  $T_i$  is allowed to wait only if it has a time stamp smaller than that of  $T_j$  (that is,  $T_i$  is older than  $T_j$ ). Otherwise,  $T_i$  is rolled back (dies).



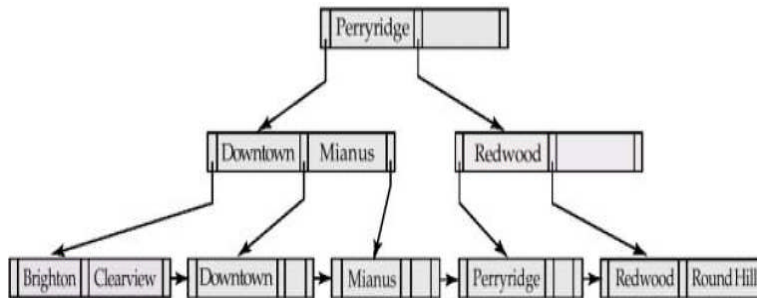


Figure: 4 B+ tree for account file with  $n=3$ .

1. Insert "Clearview"
2. Lookup "downtown"

Let us assume that the insertion operation begins first. It does a lookup on "Clearview", and finds the node into which "Clearview" should be inserted is full. It therefore converts its shared lock on the node to exclusive mode, and creates a new node. The original node now contains the search-key values "Brighton" and "Clearview". The new node contains the search key value "Downtown".

Now assume that a context switch occurs that results in control passing to the lookup operation. This lookup operation accesses the root, and follows the pointer to the left child of the root. It then accesses that node, and obtains a pointer to the left child. This left-child node originally contained the search-key values "Brighton" and "Downtown". Since this node is currently locked by the insertion operation in exclusive mode, the lookup operation must wait. Note that, at this point, the lookup operation holds no locks at all!

The insertion operation now unlocks the leaf node and relocks its parent, this time in exclusive mode. It completes the insertion, leaving the B+ tree as in Fig.16.22.

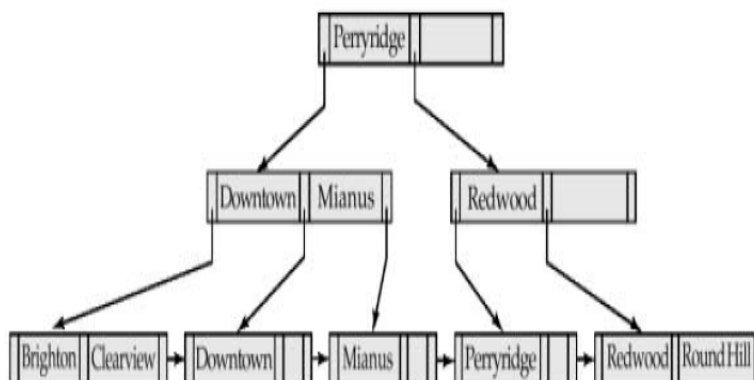


Figure: 5 Insertion of "clear view" into the B+ tree of fig.4

The lookup operation proceeds. However, it is holding a pointer to an incorrect leaf node. It therefore follows the right-sibling pointer to locate the next node. If this node, too, turns out to be incorrect, the lookup follows that node's right-sibling pointer. It can be shown that, if a lookup holds a pointer to an incorrect node, then, by following right-sibling pointers, the lookup must eventually reach the correct node.

Once a particular operation releases a lock on a node, other operations can access that node. There is a possibility of deadlocks between search operations coming down the tree, and splits, coalescing or redistribution propagating up the tree. The system can easily handle such deadlocks by restarting the search operation from the root, after releasing the locks held by the operation.

The second technique achieves even more concurrency, avoiding even holding the lock on one node while acquiring the lock on another node, by using a modified version of  $B^+$ -trees called  $B$ -link trees;  $B$ -link trees require that every node (including internal nodes, not only that node but also that node's right sibling (if one exists)). We shall illustrate this technique with an example later, but we first present the modified procedures of the  $B$ -link-tree locking protocol.

**Lookup:** Each node of the  $B^+$ -tree must be locked in shared mode before it is accessed. A lock on a nonleaf node is released before any lock on any other node in the  $B^+$ -tree is requested. If a split occurs concurrently with a lookup, the desired search-key value may no longer appear within the range of values represented by a node accessed during lookup. In such a case, the search-key value is in the range represented by a sibling node, which the system locates by following the pointer to the right sibling. However, the system locks leaf nodes following the two-phase locking protocol, as Section 16.7.3 describes, to avoid the phantom phenomenon.

**Insertion and Deletion:** The system follows the rules for lookup to locate the leaf node into which it will make the insertion or deletion. It upgrades the shared-mode lock on this node to exclusive mode, and performs the insertion or deletion. It locks leaf nodes affected by insertion or deletion following the two-phase locking protocol, as Section 16.7.3 describes, to avoid the phantom phenomenon.

**Split:** If the transaction splits a node, it creates a new node according to the algorithm of Section 12.3 and makes it the right sibling of the original node. The right-sibling pointers of both the original node and the new node are set. Following this, the transaction releases the exclusive lock on the parent, so that it can insert a pointer to the new node.

**Coalescence:** If a node has too few search-key values after a deletion, the node with which it will be coalesced must be locked in exclusive mode. Once the transaction has coalesced these two nodes, it requests an exclusive lock on the parent so that the deleted node can be removed. At this point, the transaction releases the locks on the coalesced nodes. Unless the parent node must be coalesced also, its lock is released.

Observe this important fact: An insertion or deletion may lock a node, unlock it, and subsequently relock it. Furthermore, a lookup that runs concurrently with a split or coalescence operation may find that the desired search key has been moved to the right-sibling node by the split or coalescence operation.

As an illustration, consider the  $B^+$ -tree in **Figure 4**. Assume that there are two concurrent operations on this  $B^+$ -tree:

transactions. For instance, when it is searching for records satisfying some conditions, a transaction may find some of the records inserted by a committed transaction, but may not find others.

- **Read committed-** allows only committed records to be read, but does not require even repeatable reads. For instance, between two reads of a record by the transaction, the records may have been updated by other committed transactions. This is basically the same as degree-two consistency; most systems supporting this level of consistency would actually implement cursor stability, which is a special case of degree-two consistency.
- **Read uncommitted-** allows even uncommitted records to be read. It is the lowest level of consistency allowed by SQL-92.

## 20.4 Concurrency In Index Structures

It is possible to treat access to index structures like any other database structure, and to apply the concurrency-control techniques discussed earlier. However, since indices are accessed frequently, they would become a point of great lock contention, leading to a low degree of concurrency. Luckily, indices do not have to be treated like other database structures. It is perfectly acceptable for a transaction to perform a lookup on an index twice, and to find that structure of the index has changed in between, as long as the index lookup returns the correct set of tuples. Thus, it is acceptable to have nonserializable concurrent access to an index, as long as the accuracy of the index is maintained.

We outline two techniques for managing concurrent access to **B+**-trees. The techniques that we present for concurrency control on **B+** -trees are based on locking, but neither two-phase locking nor the tree protocol is employed. The first technique is called the **crabbing protocol**:

When searching for a key value, the crabbing protocol first locks the root node in shared mode. When traversing down the tree, it acquires a shared lock on the child node to be traversed further. After acquiring the lock on the child node, it releases the lock on the parent node. It repeats this process until it reaches a leaf node.

When inserting or deleting a key value, the crabbing protocol takes these actions:

- (a). It follows the same protocol as for searching until it reaches the desired leaf node. Up to this point, it obtains only shared locks.
- (b). It locks the leaf node in exclusive mode and inserts or deletes the key value.
- (c). If it needs to split a node or coalesce it with its siblings, or redistribute key values between siblings, the crabbing protocol locks the parent of the node in exclusive mode. After performing these actions, it releases the locks on the node and siblings.

If the parent requires splitting, coalescing, or redistribution of key values, the protocol retains the lock on the parent, and splitting, coalescing, or redistribution propagates further in the same manner. Otherwise, it releases the lock on the parent.

The protocol gets its name from the way in which crabs advance by moving sideways, moving the legs on one side, then the legs on the other, and so on alternately. The progress of locking while the protocol both goes down the tree and goes back up (in case of splits, coalescing, or redistribution) proceeds in a similar crab-like manner.

T <sub>3</sub>	T <sub>4</sub>
lock-S(Q) read(Q) unlock(Q)	
	lock-X(Q) read(Q) write(Q) unlock(Q)
lock-S(Q) read(Q) unlock(Q)	

**Figure: 3** Nonserializable schedule with degree-two consistency.

The potential for inconsistency due to nonserializable schedules under degree-two consistency makes this approach undesirable for many applications.

### 20.3.2 Cursor Stability

Cursor stability is a form of degree-two consistency designed for programs written in host languages, which iterate over tuples of a relation by using cursors. Instead of locking the entire relation, cursor stability ensures that

- The tuple that is currently being processed by the iteration is locked in shared mode.
- Any modified tuples are locked in exclusive mode until the transaction commits.

These rules ensure that degree-two consistency is obtained. Two-phase locking is not required. Serializability is not guaranteed. Cursor stability is used in practice on heavily accessed relations as a means of increasing concurrency and improving system performance. Applications that use cursor stability must be coded in a way that ensures database consistency despite the possibility of nonserializable schedules. Thus, the use of cursor stability is limited to specialized situations with simple consistency constraints.

### 20.3.3 Weak Levels of Consistency in SQL

The SQL standard also allows a transaction to specify that it may be executed in such a way that it becomes nonserializable with respect to other transactions. For instance, a transaction may operate at the level of read uncommitted, which permits the transaction to read records even if they have not been committed. SQL provides such features for long transactions whose results do not need to be precise. For instance, approximate information is usually sufficient for statistics used for query optimization. If these transactions were to execute in a serializable fashion, they could interfere with other transactions, causing the others' execution to be delayed.

The levels of consistency specified by SQL-92 are as follows:

- **Serializable** is the default.
- **Repeatable read-** allows only committed records to be read, and further requires that, between two reads of a record by a transaction, no other transaction is allowed to update the record. However, the transaction may not be serializable with respect to other

- Every relation must have at least one index.
- A transaction  $T_i$  can access tuples of a relation only after finding them through one or more of the indices on the relation.
- A transaction  $T_i$  that performs a lookup (whether a range lookup or a point lookup) must acquire a shared lock on all the index leaf nodes that it accesses.
- A transaction  $T_i$  may not insert, delete, or update a tuple  $t_i$  in a relation  $r$  without updating all indices on  $r$ . The transaction must obtain exclusive locks on all index leaf nodes that are affected by the insertion, deletion, or update. For insertion and deletion, the leaf nodes affected are those that contain (after insertion) or contained (before deletion) the search-key value of the tuple. For updates, the leaf nodes affected are those that (before the modification) contained the old value of the search-key, and nodes that (after modification) contain the new value of the search-key.
- The rules of the two-phase locking protocol must be observed.

Variants of the index-locking technique exist for eliminating the phantom phenomenon under the other concurrency-control protocols presented in this chapter.

## 20.3 Weak Levels Of Consistency

Serializability is a useful concept because it allows programmers to ignore issues related to concurrency when they code transactions. If every transaction has the property that it maintains database consistency if executed alone, then serializability ensures that concurrent executions maintain consistency. However, the protocols required to ensure serializability may allow too little concurrency for certain applications. In these cases, weaker levels of consistency are used. The use of weaker levels of consistency places additional burdens on programmers for ensuring database correctness.

### 20.3.1 Degree-Two Consistency

The purpose of degree-two-consistency is to avoid cascading aborts without necessarily ensuring serializability. The locking protocol for degree-two consistency uses the same two lock modes that we used for the two-phase locking protocol: shared (**S**) and exclusive (**X**). A transaction must hold the appropriate lock mode when it accesses a data item.

In contrast to the situation in two-phase locking, **S**-locks may be released at any time, and locks may be acquired at any time. Exclusive locks cannot be released until the transaction either commits or aborts. Serializability is not ensured by this protocol. Indeed, a transaction may read the same item twice and obtain different results. In **fig.3**,  $T_3$  reads the value of  $Q$  before and after that value is written by  $T_4$ .

- If  $T_{29}$  does not use the tuple newly inserted by  $T_{30}$  in computing  $\text{sum}(\text{balance})$ , then in a serial schedule equivalent to  $S$ ,  $T_{29}$  must come before  $T_{30}$ .

The second of these two cases is curious.  $T_{29}$  and  $T_{30}$  do not access any tuple in common, yet they conflict with each other! In effect,  $T_{29}$  and  $T_{30}$  conflict on a phantom tuple. If concurrency control is performed at the tuple granularity, this conflict would go undetected. This problem is called the **phantom phenomenon**.

To prevent the phantom phenomenon, we allow  $T_{29}$  to prevent other transactions from creating new tuples in the account relation with  $\text{branch-name} = \text{'Perryridge'}$ .

To find *all account* tuples with  $\text{branch-name} = \text{'perryridge'}$ ,  $T_{29}$  must search either the whole account relation, or at least an index on the relation. Up to now, we have assumed implicitly that the only data items accessed by a transaction are tuples. However,  $T_{29}$  is an example of a transaction that reads information about what tuples are in a relation, and  $T_{30}$  is an example of a transaction that updates that information.

Clearly, it is not sufficient merely to lock the tuples that are accessed; the information used to find the tuples that are accessed by the transaction must also be locked.

The simplest solution to this problem is to associate a data item with the relation; the data item represents the information used to find the tuples in the relation. Transactions, such as  $T_{29}$ , that read the information about what tuples are in a relation would then lock the data item corresponding to the relation in shared mode. Transactions, such as  $T_{30}$ , that update the information about what tuples are in a relation would have to lock the data item in exclusive mode. Thus,  $T_{29}$  and  $T_{30}$  would conflict on a real data item, rather than on a phantom.

Do not confuse the locking of an entire relation, as in multiple granularity locking, with the locking of the data item corresponding to the relation. By locking the data item, a transaction only prevents other transactions from updating information about what tuples are in the relation. Locking is still required on the tuples. A transaction that directly accesses a tuple can be granted a lock on the tuples even when another transaction has an exclusive lock on the data item corresponding to the relation itself.

The major disadvantage of locking a data item corresponding to the relation is the low degree of concurrency- two transactions that insert different tuples into a relation are prevented from executing concurrently.

A better solution is the **index-locking** technique. Any transaction that inserts a tuple into a relation must insert information into every index maintained on the relation. We eliminate the phantom phenomenon by imposing a locking protocol for indices. For simplicity we shall only consider **B+** - tree indices.

Every search-key value is associated with an index leaf node. A query will usually use one or more indices to access a relation. An insert must insert the new tuple in all indices on the relation. In our example, we assume that there is an index on account for  $\text{branch-name}$ . Then,  $T_{30}$  must modify the leaf containing the key Perryridge. If  $T_{29}$  reads the same leaf node to locate all tuples pertaining to the Perryridge branch, then  $T_{29}$  and  $T_{30}$  conflict on that leaf node.

The **index-locking protocol** takes advantage of the availability of indices on a relation, by turning instances of the phantom phenomenon into conflicts on locks on index leaf nodes. The protocol operates as follows:

- $I_j = \text{insert}(Q)$ .  $I_i$  and  $I_j$  conflict. Suppose that data item  $Q$  did not exist prior to the execution of  $I_i$  and  $I_j$ . Then, if  $I_i$  comes before  $I_j$ , a logical error results for  $T_i$ . If  $I_j$  comes before  $I_i$ , then no logical error results. Likewise, if  $Q$  existed prior to the execution of  $I_i$  and  $I_j$ , then a logical error results if  $I_j$  comes before  $I_i$ , but not otherwise.

We can conclude the following:

- Under the two-phase locking protocol, an exclusive lock is required on a data item before that item can be deleted.
- Under the timestamp-ordering protocol, a test similar to that for a write must be performed. Suppose that transaction  $T_i$  issues delete ( $Q$ ).
- If  $TS(T_i) < R\text{-timestamp}(Q)$ , then the value of  $Q$  that  $T_i$  was to delete has already been read by a transaction  $T_j$  with  $TS(T_j) > TS(T_i)$ . Hence, the delete operation is rejected, and  $T_i$  is rolled back.
- If  $TS(T_i) < W\text{-timestamp}(Q)$ , then a transaction  $T_j$  with  $TS(T_j) > TS(T_i)$  has written  $Q$ . Hence, this delete operation is rejected, and  $T_i$  is rolled back.
- Otherwise, the delete is executed.

### 20.2.2 Insertion

We have already seen that an insert ( $Q$ ) operation conflicts with a delete ( $Q$ ) operation. Similarly, insert ( $Q$ ) conflicts with a read ( $Q$ ) operation or a write ( $Q$ ) operation; no read or write can be performed on a data item before it exists.

Since an insert ( $Q$ ) assigns a value to data item  $Q$ , an insert is treated similarly to a write for concurrency-control purposes:

- Under the two-phase locking protocol, if  $T_i$  performs an insert ( $Q$ ) operation,  $T_i$  is given an exclusive lock on the newly created data item  $Q$ .
- Under the timestamp-ordering protocol, if  $T_i$  performs an insert ( $Q$ ) operation, the values  $R\text{-timestamp}(Q)$  and  $W\text{-timestamp}(Q)$  are set to  $TS(T_i)$ .

### 20.2.3 The Phantom Phenomenon

Consider transaction  $T_{29}$  that executes the following SQL query on the bank database:

```
select sum(balance)
From account
where branch-name = 'perryridge'
```

Transaction  $T_{29}$  requires access to all tuples of the account relation pertaining to the Perryridge branch.

Let  $T_{30}$  be a transaction that executes the following SQL insertion:

```
insert into account values ( A-201, 'Perryridge', 900)
```

Let  $S$  be a schedule involving  $T_{29}$  and  $T_{30}$ . We expect there to be potential for a conflict for the following reasons:

- If  $T_{29}$  uses the tuple newly inserted by  $T_{30}$  in computing  $\text{sum}(\text{balance})$ , then  $T_{29}$  read a value written by  $T_{30}$ . Thus, in a serial schedule equivalent to  $S$ ,  $T_{30}$  must come before  $T_{29}$ .

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

The simplest solution is a total rollback. Abort the transaction and then restart it. However, it is more effective to roll back the transaction only as far as necessary to break the deadlock. Such partial rollback requires the system to maintain additional information about the state of all the running transactions. Specifically, the sequence of lock requests/grants and updates performed by the transaction needs to be recorded. The deadlock detection mechanism should decide which locks the selected transaction needs to release in order to break the deadlock. The selected transaction must be rolled back to the point where it obtained the first of these locks, undoing all actions it took after that point. The recovery mechanism must be capable of performing such partial rollbacks. Furthermore, the transactions must be capable of resuming execution after a partial rollback. See the bibliographical notes for relevant references.

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

## 20.2 Insert and Delete Operations

Until now, we have restricted our attention to read and write operations. This restriction limits transactions to data items already in the database. Some transactions require not only access to existing data items, but also the ability to create new data items. Others require the ability to delete data items. To examine how such transactions affect concurrency control, we introduce these additional operations:

- **delete (Q)** deletes data item **Q** from the database.
- **insert (Q)** inserts a new data item **Q** into the data base and assigns **Q** an initial value.

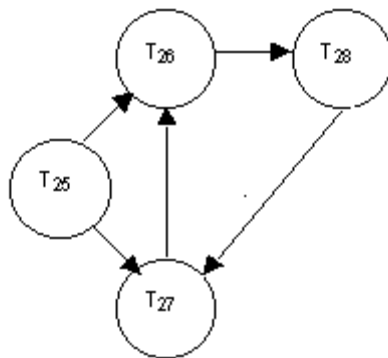
An attempt by a transaction  $T_i$  to perform a read (**Q**) operation after **Q** has been deleted results in a logical error in  $T_i$ . Likewise, an attempt by a transaction  $T_i$  to perform a read (**Q**) operation before **Q** has been inserted results in a logical error in  $T_i$ . It is also a logical error to attempt to delete a nonexistent data item.

### 20.2.1 Deletion

To understand how the presence of delete instructions affects concurrency control, we must decide when a delete instruction conflicts with another instruction. Let  $I_i$  and  $I_j$  be instructions of  $T_i$  and  $T_j$ , respectively that appear in schedule **S** in consecutive order. Let  $I_i = \text{delete (Q)}$ . We consider several instructions  $I_j$ .

- $I_j = \text{read (Q)}$ .  $I_i$  and  $I_j$  conflict. If  $I_i$  comes before  $I_j$ ,  $T_j$  will have a logical error. If  $I_j$  comes before  $I_i$ ,  $T_j$  can execute the read operation successfully.
- $I_j = \text{write (Q)}$ .  $I_i$  and  $I_j$  conflict. If  $I_i$  comes before  $I_j$ ,  $T_j$  will have a logical error. If  $I_j$  comes before  $I_i$ ,  $T_j$  can execute the write operation successfully.
- $I_j = \text{delete (Q)}$ .  $I_i$  and  $I_j$  conflict. If  $I_i$  comes before  $I_j$ ,  $T_i$  will have a logical error. If  $I_j$  comes before  $I_i$ ,  $T_i$  will have a logical error.





**Figure: 2 Wait-for graph with a cycle.**

This time, the graph contains the cycle:

$$T_{26} \rightarrow T_{28} \rightarrow T_{27} \rightarrow T_{26}$$

implying that transactions  $T_{26}$ ,  $T_{27}$ , and  $T_{28}$  are deadlocked consequently, the question arises: When should we invoke the detection algorithm?

The answer depends on two factors:

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

If deadlocks occur frequently, then the detection algorithm should be invoked more frequently than usual. Data items allocated to deadlocked transactions will be unavailable to other transactions until the deadlock can be broken. In addition, the number of cycles in the graph may also grow. In the worst case, we would invoke the detection algorithm every time a request could not be granted immediately.

### 20.1.3.2 Recovery from Deadlock

When a detection algorithm determines that a deadlock exists, the system must recover from the deadlock. The most common solution is to roll back one or more transactions to break the deadlock. Three actions need to be taken:

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

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

- Provide an algorithm that uses this information to determine whether the system has entered a deadlock state.
- Recover from the deadlock when the detection algorithm determines that a deadlock exists.

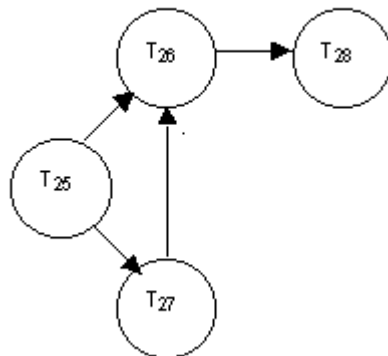
### 20.1.3.1 Deadlock Detection

Deadlocks can be described precisely in terms of a directed graph called a wait-for graph. This graph consists of a pair  $G=(V,E)$ , where  $V$  is a set of vertices and  $E$  is a set of edges. The set of vertices consists of all the transactions in the system. Each element in the set  $E$  of edges is an ordered pair  $T_i \rightarrow T_j$ . If  $T_i \rightarrow T_j$  is in  $E$ , then there is a directed edge from transaction  $T_k$  to  $T_j$ , implying that transaction  $T_i$  is waiting for transaction  $T_j$  to release a data item that it needs.

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

A deadlock exists in the system if and only if the wait-for graph contains a cycle.

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



**Figure: 1** Wait for graph with no cycle.

To illustrate these concepts, consider the wait-for graph in **Fig.1**, which depicts the following situation:

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

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

Suppose now that transaction  $T_{28}$  is requesting an item held by  $T_{27}$ . The edge  $T_{28} \rightarrow T_{27}$  is added to the wait-for graph, resulting in the new system state in **fig.2**.

2. The **wound-wait** scheme is a preemptive technique. It is a counterpart to the wait-die scheme. When transaction  $T_i$  requests a data item currently held by  $T_j$ ,  $T_i$  is allowed to wait only if it has a timestamp larger than that of  $T_j$  (that is,  $T_i$  is younger than  $T_j$ ). Otherwise,  $T_j$  is rolled back ( $T_j$  is wounded by  $T_i$ ).

Returning to our example, with transactions  $T_{22}$ ,  $T_{23}$  and  $T_{24}$ , if  $T_{22}$  requests a data item held by  $T_{23}$ , then the data item will be preempted from  $T_{23}$ , and  $T_{23}$  will be rolled back. If  $T_{24}$  requests a data item held by  $T_{23}$ , then  $T_{24}$  will wait.

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

Both the wound-wait and the wait-die schemes avoid starvation: At anytime, there is a transaction with the smallest timestamp. This transaction cannot be required to roll back in either scheme. Since time stamps always increase, and since transactions are not assigned new time stamps when they are rolled back, a transaction that is rolled back repeatedly will eventually have the smallest timestamp, at which point it will not be rolled back again.

There are, however, significant differences in the way that the two schemes operate.

- In the wait-die scheme, an older transaction must wait for a younger one to release its data item. Thus, the older the transaction gets, the more it tends to wait. By contrasting the wound-wait scheme, an older transaction never waits for a younger transaction.
- In the wait-die scheme, if a transaction  $T_i$  dies and is rolled back because it requested a data item held by transaction  $T_j$ , then  $T_i$  may reissue the same sequence of requests when it is restarted. If the data item is still held by  $T_j$ , then  $T_i$  will die again. Thus,  $T_i$  may die several times before acquiring the needed data item. Contrast this series of events with what happens in the wound-wait scheme. Transaction  $T_i$  is wounded and rolled back because  $T_j$  requested a data item that it holds. When  $T_i$  is restarted and requests the data item now being held by  $T_j$ ,  $T_i$  waits. Thus, there may be fewer rollbacks in the wound-wait scheme.

The major problem with both of these schemes is that unnecessary rollbacks may occur.

### 20.1.2 Timeout-Based Schemes

Another simple approach to deadlock handling is based on lock timeouts. In this approach, a transaction that has requested a lock waits for at most a specified amount of time. If the lock has not been granted within that time, the transaction is said to timeout, and it rolls itself back and restarts. If there was in fact a deadlock, one or more transactions involved in the deadlock will timeout and roll back allowing the others to proceed.

The timeout scheme is particularly easy to implement, and works well if transactions are short and if long waits are likely to be due to deadlocks.

### 20.1.3 Deadlock Detection and Recovery

If a system does not employ some protocol that ensures deadlock freedom, then a detection and recovery scheme must be used. An algorithm that examines the state of the system is invoked periodically to determine whether a dead lock has occurred. If one has, then the system must attempt to recover from the deadlock. To do so, the system must:

- Maintain information about the current allocation of data items to transactions, as well as any outstanding data item requests.

There are two principal methods for dealing with the deadlock problem. We can use a **deadlock prevention** protocol ensure that the system will never enter a dead lock state. Alternatively, we can allow the system to enter a dead lock state, and then try to recover by suing a **deadlock detection** and **deadlock recovery** scheme. So we shall see, both methods may result in transaction rollback. Prevention is commonly used if the probability that the system would enter a dead lock state is relatively high; other wise, detection and recovery are more efficient.

Note that a detection and recovery scheme requires overhead that includes not only the run-time cost of maintaining the necessary information and of executing the detection algorithm, but also the potential losses inherent in recovery from dead lock.

### 20.1.1 Deadlock Prevention

There are two approaches to deadlock prevention. One approach ensures that no cyclic waits can occur by ordering the requests for locks, or requiring all locks to be acquired together. The other approach is closer to deadlock recovery, and performs transaction rollback instead of waiting for a lock, whenever the wait could potentially result in a deadlock.

The simplest scheme under the first approach requires that each transaction lock all its data items before it begins execution. Moreover, either all are locked in one step or none are locked. There are two main disadvantages to this protocol: (1) It is often hard to predict, before the transaction begins, what data items need to be locked. (2) Data-item utilization may be very low, since many of the data items may be locked but unused for al long time.

Another approach for preventing deadlocks is to impose an ordering of all data items, and to require that a transaction lock data items only in a sequence consistent with the ordering. We have seen one such scheme in the tree protocol, which uses a partial ordering of data items.

A variation of this approach is to use a total order of data items, in conjunction with two-phase locking. Once a transaction has locked a particular item, it cannot request locks on items that precede that item in the ordering. This scheme is easy to implement, as long as the set of data items accessed but a transaction is known when the transaction starts execution. There is no need to change the underlying concurrency-control system if two-phase locking is used. All that is needed it to ensure that locks are requested in the right order.

The second approach for preventing deadlocks is to use preemption and transaction rollbacks. In preemption, when a transaction  $T_2$  requests a lock that transaction holds, the lock granted to  $T_1$  may be preempted by rolling back of  $T_1$ , and granting of the lock to  $T_2$ . To control the preemption we assign a unique time stamp to each transaction. The system uses these time stamps only to decide whether a transaction should wait or roll back. Locking is still used for concurrency control. If a transaction is rolled back, it retains its old time stamp when restarted. Two different deadlock prevention schemes using timestamps have been proposed:

1. The **wait-die** scheme is a nonpreemptive technique. When transaction  $T_i$  requests a data item currently held by  $T_j$ ,  $T_i$  is allowed to wait only if it has a time stamp smaller than that of  $T_j$  (that is,  $T_i$  is older than  $T_j$ ). Otherwise,  $T_i$  is rolled back (dies).

For example, suppose that transaction  $T_{22}$ ,  $T_{23}$ , and  $T_{24}$  have timestamps 5, 10, and 15 respectively. If  $T_{22}$  requests a data item held by  $T_{23}$ , then  $T_{22}$  will wait. If  $T_{24}$  requests a data item held by  $T_{23}$ , then  $T_{24}$  will be rolled back.

## Lesson 20

# Concurrency Control-3

### 20.0 Objectives:

After completion of this lesson the student will be able to know about:

- Identifying the deadlock situation and prevention.
- Detecting deadlock and recovery from deadlock.
- Insert and delete operations.
- Different weak levels of consistency for ensuring correctness.
- Concurrency in index structures.

### Structure Of the Lesson:

- 20.1 Deadlock Handling
- 20.2 Insert and Delete Operations
- 20.3 Weak Levels Of Consistency
- 20.4 Concurrency in index Structures
- 20.5 Summary
- 20.6 Technical Terms
- 20.7 Model Questions
- 20.8 References

### 20.1 Deadlock Handling

A system is in deadlock state if there exists a set of transactions such that every transaction in the set is waiting for another transaction in the set. More precisely there exists a set of waiting transactions  $\{T_0, T_1, \dots, T_n\}$  such that  $T_0$  is waiting for a data item that  $T_1$  holds, and  $T_1$  is waiting for a data item that  $T_2$  holds, and ..., and  $T_{n-1}$  is waiting for a data item that  $T_n$  holds, and  $T_n$  is waiting for a data item that  $T_0$  holds. None of the transactions can make progress in such a situation.

The only remedy to this undesirable situation is for the system to invoke some drastic action, such as rolling back some of the transactions involved in the deadlock. Rollback of a transaction may be partial, i.e., a transaction may be rolled back to the point where it obtained a lock whose release resolves the deadlock.