DATA BASE MANAGEMENT SYSTEMS

(DMCA106)

(MCA)

ACHARYA NAGARJUNA UNIVERSITY

CENTRE FOR DISTANCE EDUCATION

NAGARJUNA NAGAR,

GUNTUR

ANDHRA PRADESH

UNIT – I

1. INTRODUCTION

Objective

The major objective of this lesson is to provide basics of database concepts and technology.

After reading this chapter, you understand:

- The elementary concepts data, information and database.
- Database management system and structure.
- Differences between conventional data processing and database management system.
- Systematic database design approaches covering conceptual design, logical design and an overview of physical design.
- Various database models.

Structure of Lesson

Introduction

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.

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

In simple words, DBMS is a computerized record keeping system that lets the user to perform various operations over the database.
The main objective of DBMS is to provide an environment that is both convenient and efficient to use in retrieving and storing data.

Data: Data refers to facts, symbols, events, entities, variable, names or any other, which has little meaning. Database that contains the facts like Employee Name, City, College etc, For Ex:

Krishna ECIL Hyderabad

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 is known as meta data. The metadata describe the properties of data, but not include that data.

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

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

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

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.

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.

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 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.

What is 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.

Evaluation of Database Systems

DBMS were first introduced during the 1960's. This was called "proof-off-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.

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 is known as object-relational databases are now available. In the future multidimensional data will become more important.

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.

Advantages of 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:

Data consistency is 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:

To improve productivity the cost and time for developing new business applications are reduced. 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.

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.

Disadvantages of 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 requires 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 renter 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.

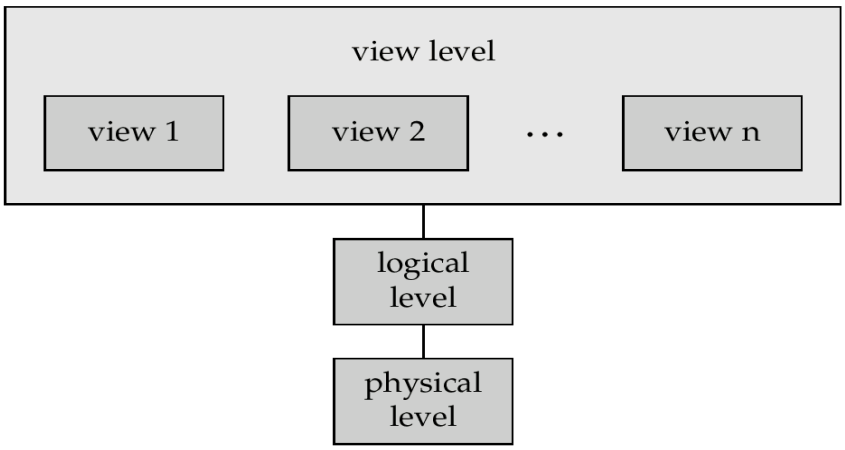
View of Data

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 database architecture allows a clear separation of the 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.

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.

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.
- Semi Structured 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 of in this model:

- Entity-Relationship model.
- Object- oriented data model.

Entity Relationship data model:

The entity – relationship data model is a logical representation of the data for an organization or for a business area or 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. The three widely acceptable record based data models are:

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

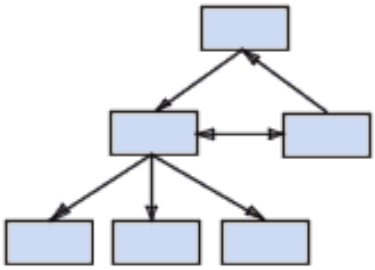
Relational data model:

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 are 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. The relational data model is concerned with three aspects:

1. Data structure.
2. Data integrity.
3. Data manipulation.

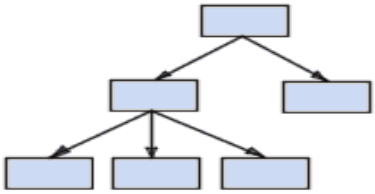
Network data model:

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



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.



Physical Data models:

The physical data model is used to describe data at lowest level. Unlike other data models, physical data models are very less in use. Two widely known ones are:

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

Semi-structured Data Models: The semi-structured data model permits the specification of data where individual data items of the same type may have different sets of attributes. This is quiet against to the above mentioned data models. The XML (Extensible Markup Language) is widely used to represent semi-structured data.

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 fact, the data-definition and data-manipulation are not two separate languages. Instead they simply form a database language, such as the widely used SQL language.

Data Manipulation Language

A data-manipulation language (DML) is a language that enables users to access or manipulate data as organized by the appropriate data model (i.e., relational data model). The types of access are:

- ✓ Retrieval of information stored in the database
- ✓ Insertion of new information into the database
- ✓ Deletion of information from the database
- ✓ Modification of information stored in the database

There are basically two types of DMLs: Procedural & Declarative.

Procedural DMLs require a user to specify what data are needed and how to get those data.

Declarative DMLs (also referred as nonprocedural DMLs) require a user to specify what data are needed without specifying how to get those data. Declarative DMLs are easier to learn than the procedural DMLs. A query is a statement requesting the retrieval of information. Implementing queries is possible through any query language. There are number of query languages in use. One of the best one is SQL (structured query language).

Data Definition language

We define any database schema by a set of definitions expressed by a data definition language (DDL). The data definition language used to define structure of database and it allows maintaining the data integrity and consistency. In order to maintain integrity one should enforce constraints over the database to be designed. Once a constraint is applied, the database checks the data before performing any operation like insertion, deletion and update.

Domain Constraints:

A domain of possible values must be associated with every attribute. Domain constraints are the most elementary form of integrity constraint. They can be tested easily by the system whenever a new data item is entered in to the database.

Referential integrity:

There are certain cases, where a column of values in one relation refer to the column of values in another relation. In such cases one can enforce referential integrity to maintain consistency.

Assertions:

An assertion is any condition that the database must always satisfy. Domain constraints and integrity constraints are special forms of assertions. However, every assertion may not be applied through the above two approaches. For example, “every loan must have a customer who maintains an account with a minimum balance of Rs.3000” , must be expressed as an assertion. If the assertion is valid, then any future modification to the database is allowed only if it does not cause that assertion to be violated.

Authorization:

Sometimes the users may be differentiated while accessing the database. Some of the users must have access to partial database only. In such situations one can go for authorization. The authorization may be read authorization, update authorization, and delete authorization. Once delete authorization is given to a user, he will be allowed to perform delete operation over a relation. Moreover, either all or none of the users may hold authorization to a particular database.

Database Users and Administrators

A primary goal of 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 or database administrators.

Database users

There are four different types of database-system users, differentiated by the way they expect to interact with the system. Different types of user interfaces have been defined for different types of users.

Naive Users:

Persons who are unsophisticated users and who interact with the system by invoking one of the application programs that have been written previously are naïve users. For example a bank teller who transfers Rs. 500 from account A to account B invokes a program called transfer. People, who wish to check their balance over the worldwide web, are the other example for naïve users. Naïve users use the forms as their user interfaces and can read the reports.

Application Programmers:

Persons who are computer professionals and who write application programs are application programmers. Application programmers can choose from many tools to develop user interfaces. RAD (Rapid Application Development) Tools are the tools that enable an application programmer to construct forms and reports with minimal programming effort.

Sophisticated Users:

Persons who interact with the system without writing programs are sophisticated users. Instead, they form their requests in a database query language. They submit each query to the query processor. The job of a query processor is to bread down the DML statements in to instructions so that they can be understood by the storage manager. Analysts who submit queries to explore or extract data in the database will fall in this category.

Specialized Users:

They are sophisticated users who write specialized database applications that do not fit into the traditional data processing framework. CAD systems, knowledge based expert systems are the examples for such specialized database applications.

End Users:

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

DBA

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:

- Schema Definition. The DBA creates the original database schema by executing a set of data definition statements in the DDL.
- Storage Structure and access-method definitions.
- Schema and physical organization modification. The DBA carries out changes to the schema and physical organization to reflect the changing needs of the organization, or to alter the physical organization to improve performance.
- Granting of authorization for data access. By granting different types of authorization, the database administrator can regulate which parts of the database the users can access. The authorization information is kept in a special system structure that the database system consults whenever someone attempts to access the data in the system.
- Routine maintenance. Examples of the database administrator's routine maintenance activities are:
- Periodically backup the database, either on to tapes or on to remote servers, to prevent loss of data incase of natural calamities like flooding.
- Ensuring that enough disk space is available for normal operations, and upgrading disk space as required.

Monitoring jobs running on the database and ensuring that performance is not degraded by very expensive tasks submitted by some users.

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.

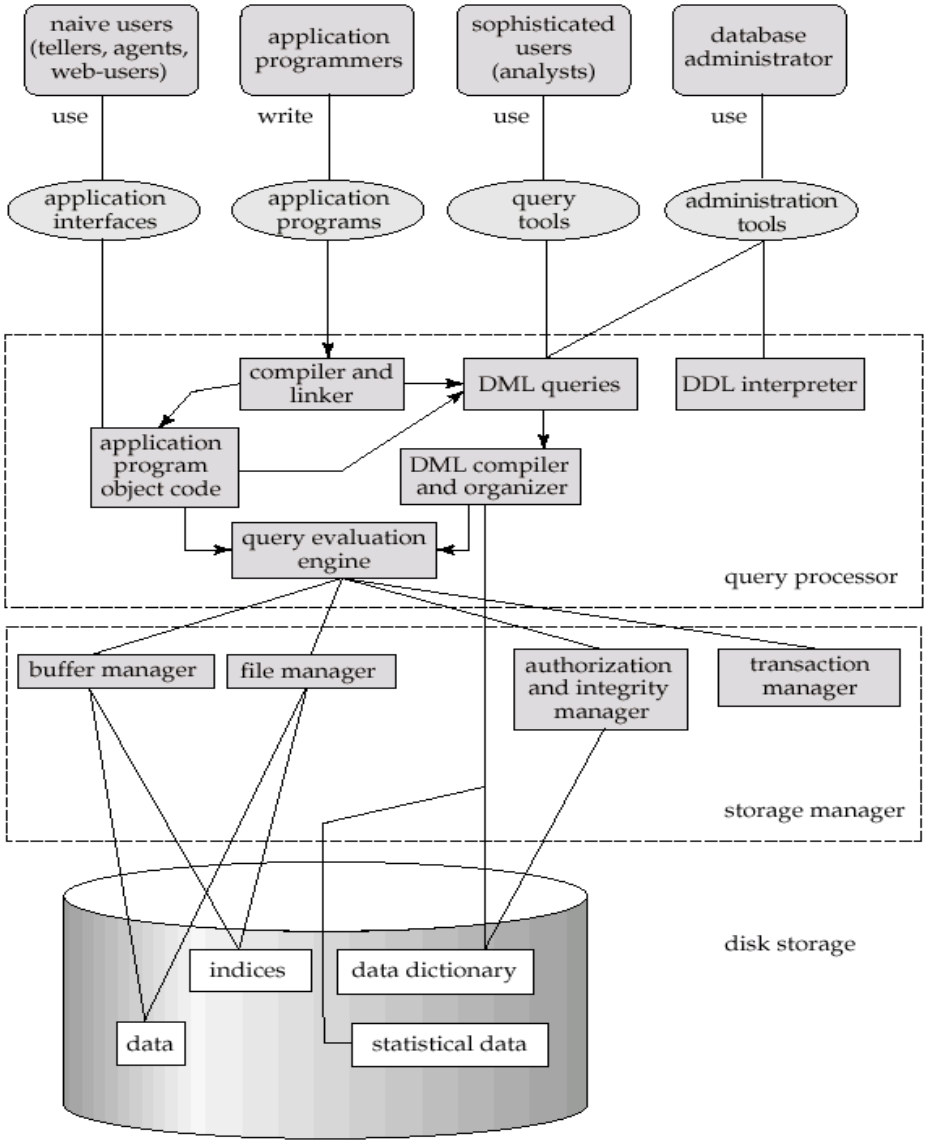
Concurrency control manager controls the interaction among the concurrent transactions, to ensure the consistency of the database

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 ad 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

History of database systems

Data processing drives the growth of computers, as it has from the earliest days of commercial computers. In fact, automation of data processing tasks predates computers. Punched cards are used widely as a means of entering data into computers.

1950s and early 1960s:

Magnetic tapes were developed for data storage. Payrolls and inventory databases were automated, with the data stored on tapes. The tasks like copying data from one tape to another, read data from the tape and sending to printer were performed. In fact, data was read in sequential manner only from the tapes and data sizes were much larger than the main memory.

Late 1960s and 1970s:

Wide spread use of hard disks in the late 1960s changed the scenario for data processing greatly, since hard disks allowed random access to data. The data over the hard disk can be anywhere; the position of the data was immaterial. With disks, network and hierarchical databases could become easy to maintain.

1980s:

Initially relational data models were tough to manage when compared to network and hierarchical databases. By early 1980s, relational database have become so easy to manage; eventually they replaced network and hierarchical data bases. Relational databases were introduced by E.F. Codd.

The first relational product SQL was introduced in this time. The other relational products are IBM's DB2, Sybase and Ingres. The 1980s also saw much research on parallel and distributed databases as well as object based databases.

Early 1990s:

The SQL language was designed primarily for decision support applications, which are query intensive. Many database vendors have introduced parallel database products.

Late 1990s:

The introduction of worldwide web has changed the scenario rapidly. Databases were deployed much more extensively than ever before. Databases have reliability and have 24x7 availability. Web interfaces were introduced.

Early 2000s:

In early 200s, we have seen the emerging of XML and the associated query language X Query as a new database technology.

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.

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.

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.

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. Give the architecture of data base management system and explain each block ?
4. What are five main functions of Database Administrator (DBA)?
5. Explain clearly about DDL and DML?

References

Database System Concepts
Silberschatz, Korth, and Sudarshan

An Introduction to Database Systems
Bipin Desai

An Introduction to Database Systems
C. J. Date

DATABASE MANAGEMENT SYSTEMS

B.Tech(COMPUTER SCIENCE)
III YEAR

Lesson Writer

P.Ammi Reddy , M.Tech.,
Vasireddy Venkatadri Institute of Technology
Nambur (P.O.), GUNTUR – Dt.

Editor & Advisor for the Course

Prof. E.SREENIVASA REDDY, M.Tech., Ph.D.
Principal
Vasireddy Venkatadri Institute of Technology
Nambur (P.O.), GUNTUR – Dt.

Director
Prof.V.CHANDRASEKHARA RAO, M.Com., Ph.D.

CENTRE FOR DISTANCE EDUCATION
ACHARAYA NAGARJUNA UNIVERSITY
NAGARJUNA NAGAR – 522 510

Ph: 0863-2293299,2293356,08645-211023,Cell:98482 85518
08645-21102 4 (Study Material)
Website: www.anucde.com, e-mail:anucde@yahoo.com

2. ENTITY-RELATIONSHIP MODEL

Objective

The main objective of this lesson is to develop the skills necessary for the design and evaluation of database management systems through the Entity - Relationship data 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

Introduction

The entity –relationship data model is a logical representation of the data for an organization or for a business area or 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.

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, “Krishna” with Emp-ID V053 is an entity, as he can be uniquely identified as one particular employee in the organization.

Person: EMPLOYEE, STUDENT, and PATIENT
Concept: ACCOUNT, COURSE, and WORK CENTRE

The overall logical structure of database can be expressed graphically by an E-R diagram, which is built up from the following components:

Rectangles	- To represent entities.
Ellipses	- To represent attributes.
Diamonds	- To represent relationships among entities.
Lines	- To link attributes to entities.

Basic E-R notations

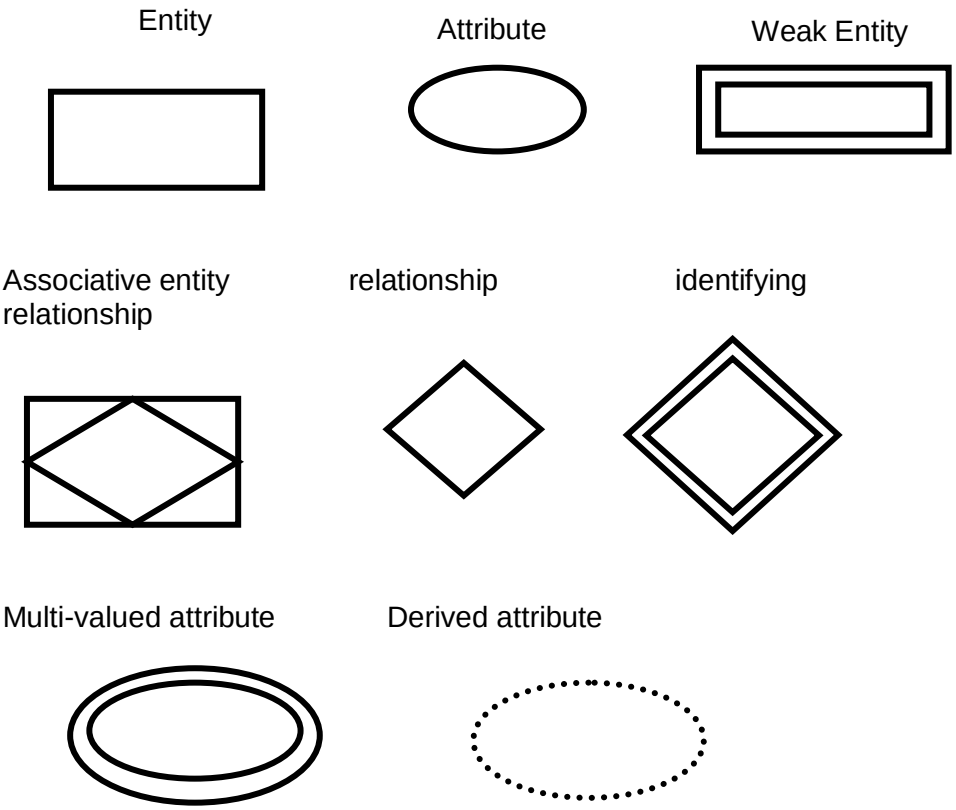


Figure 2.1 Basic E-R notations

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: Customer Number, Order Number, Order Date, Product Number.

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.

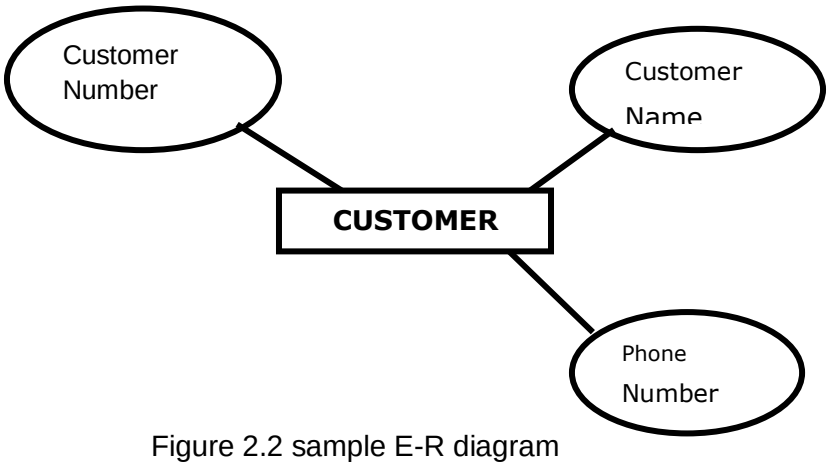


Figure 2.2 sample E-R diagram

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.

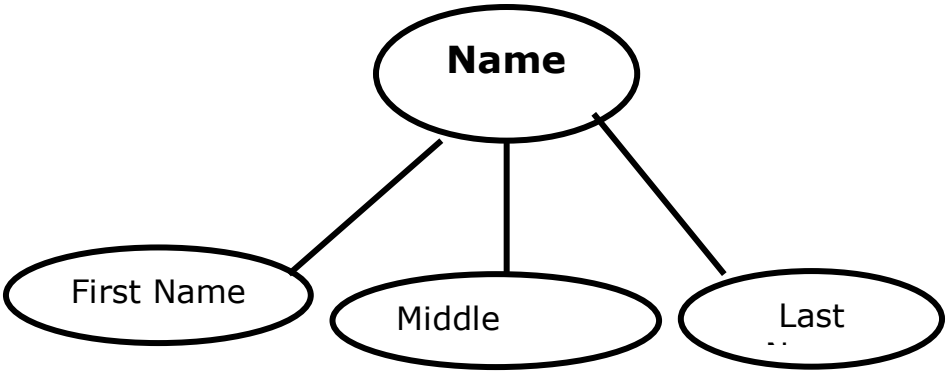


Figure 2.3 composite attribute

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

Single Valued Attributes:

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

Multi-valued Attributes:

A multi-valued 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.

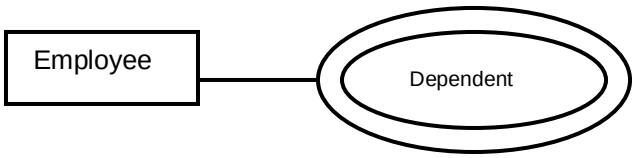


Figure 2.4 multi-valued attribute

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

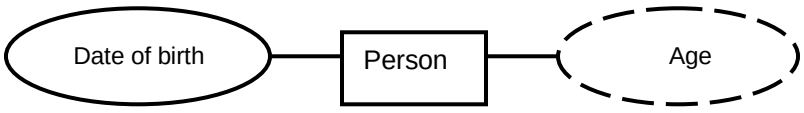


Figure 2.5 derived attribute

Relationships

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.

Figure 2.6 ER diagram notation for relationship

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 are entity sets, then a relationship set R is a subset of

$$\{(e_1, e_2, \ldots, e_n) \mid e_1 \in E_1, e_2 \in E_2, \ldots, e_n \in E_n\},$$

Where $(e_1, e_2, \ldots, e_n)$ 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.

Cardinality Constraint

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 entity in A. observe fig

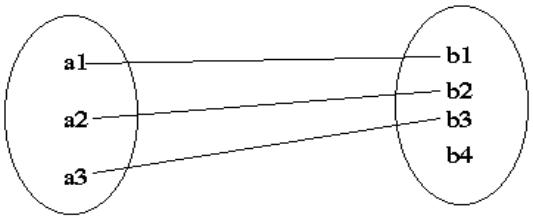


Figure 2.4.1

One-to-Many:

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

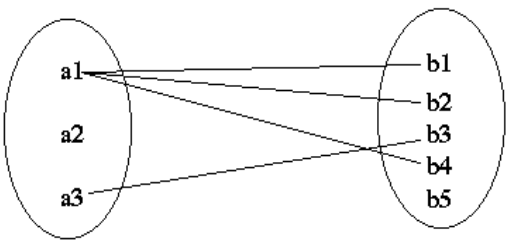


Figure 2.4.2

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.

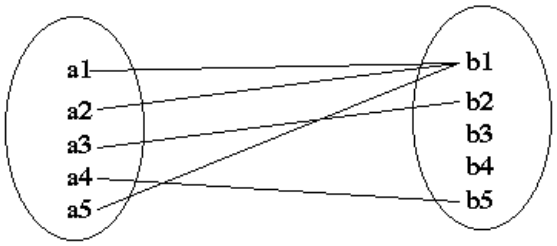


Figure 2.4.3

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.

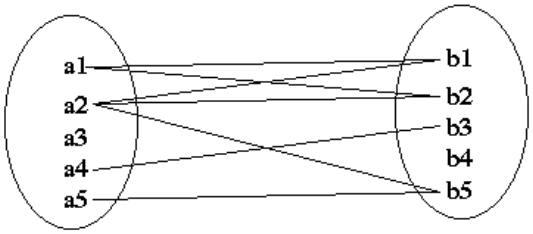


Figure 2.4.4

E-R Diagrams

The following are different types of E-R diagrams:

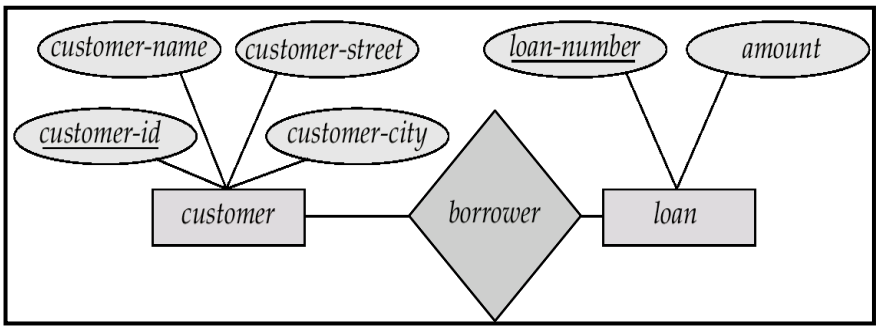


Figure 2.5.1- E-R Diagram corresponding to customer and loan

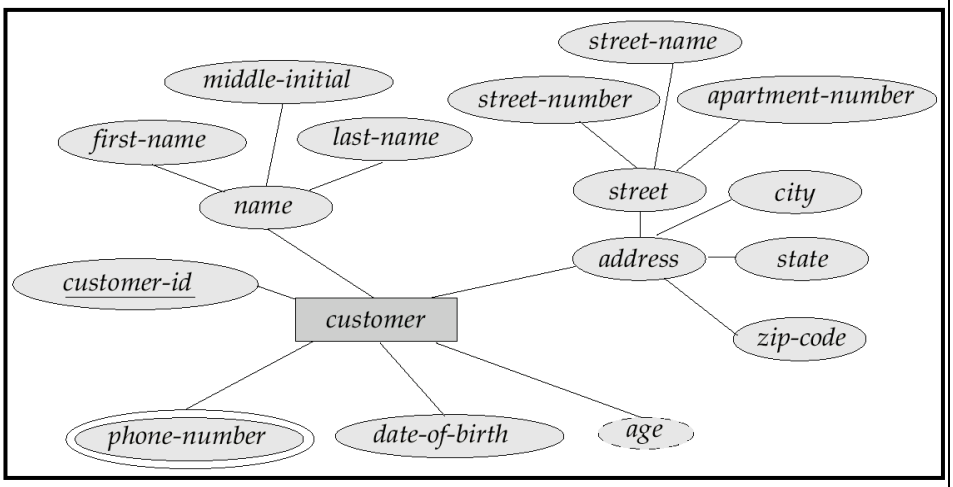


Figure 2.5.2 - E-R Diagram with all types of attributes

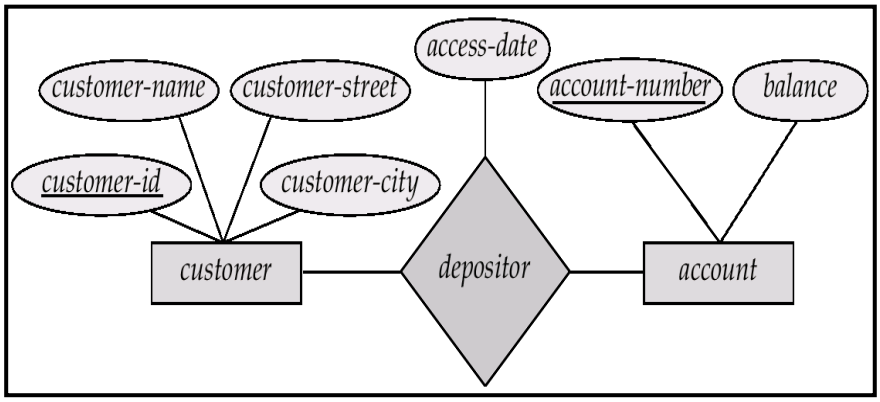


Figure 2.5.3 - Relationship with Attributes

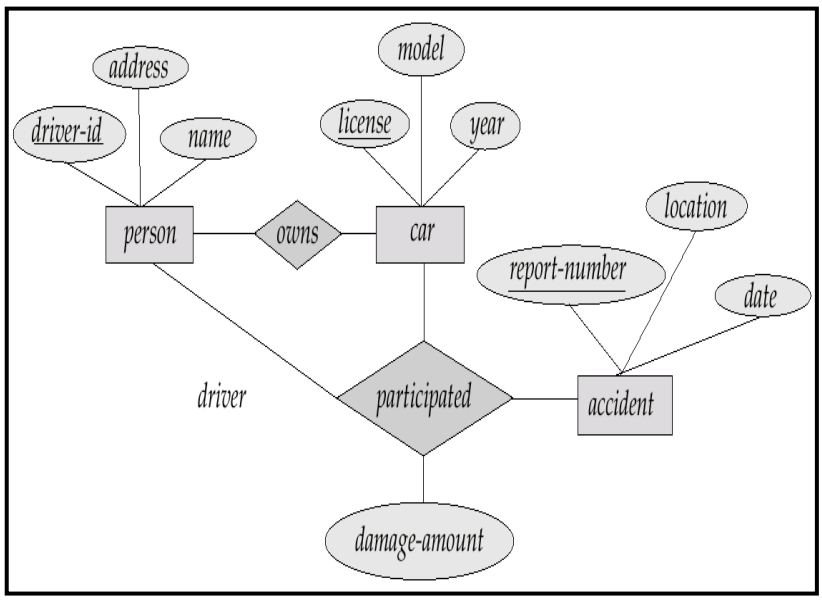


Figure 2.5.4 – E-R diagram with multiple relationships

The other category of relationship follows:

- Unary Relationships
- Binary Relationships
- Ternary Relationships

Unary Relationship: An entity that holds a relation with the entity of same set.

Binary Relationship: A relationship that holds between two entities is a binary relationship.

Ternary Relationship: A relationship that can be hold among three entities is a ternary relationship.

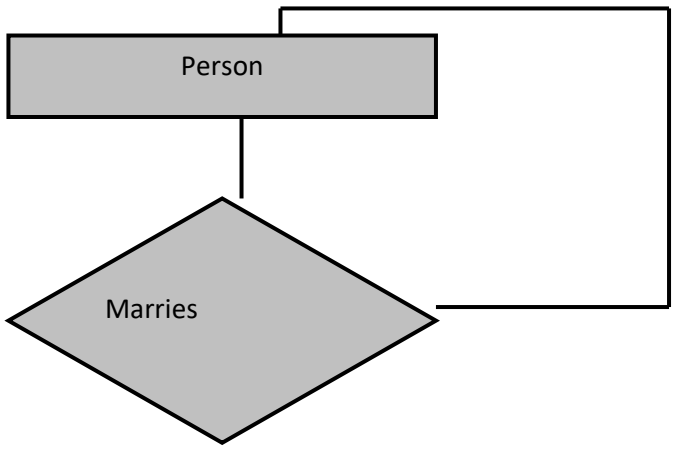


Figure 2.5.5 - Unary Relationship

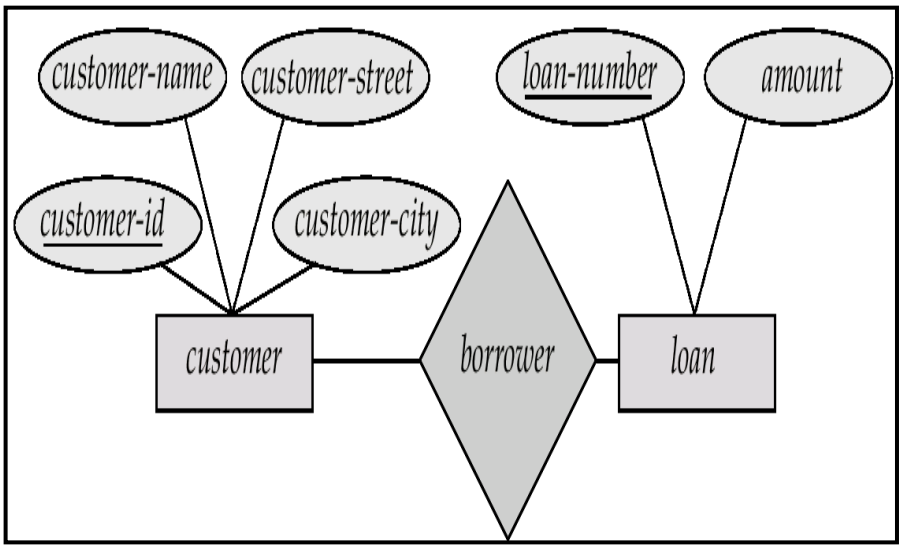


Figure 2.5.6 - Binary Relationship

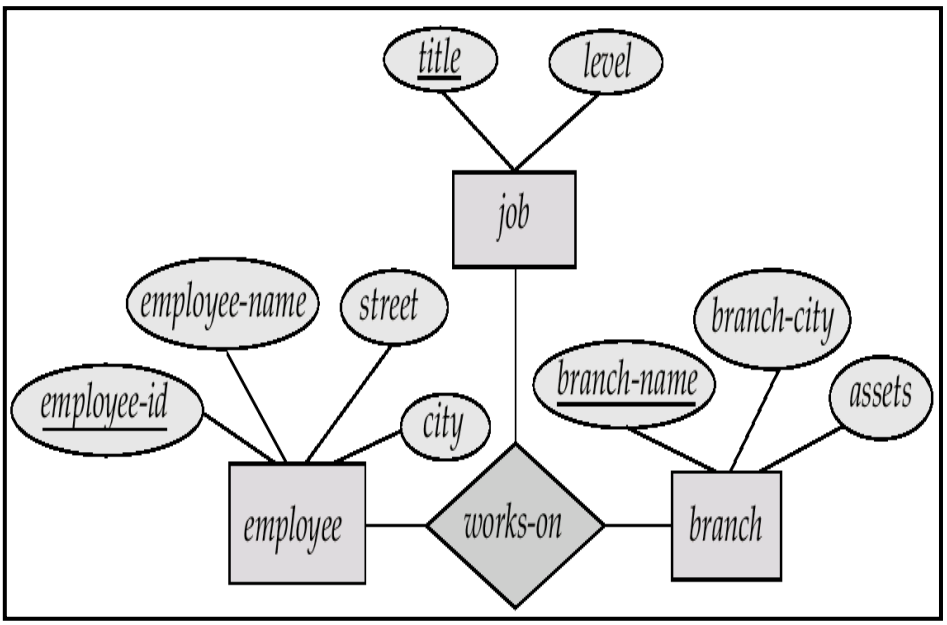


Figure 2.5.7 - One-to-Many

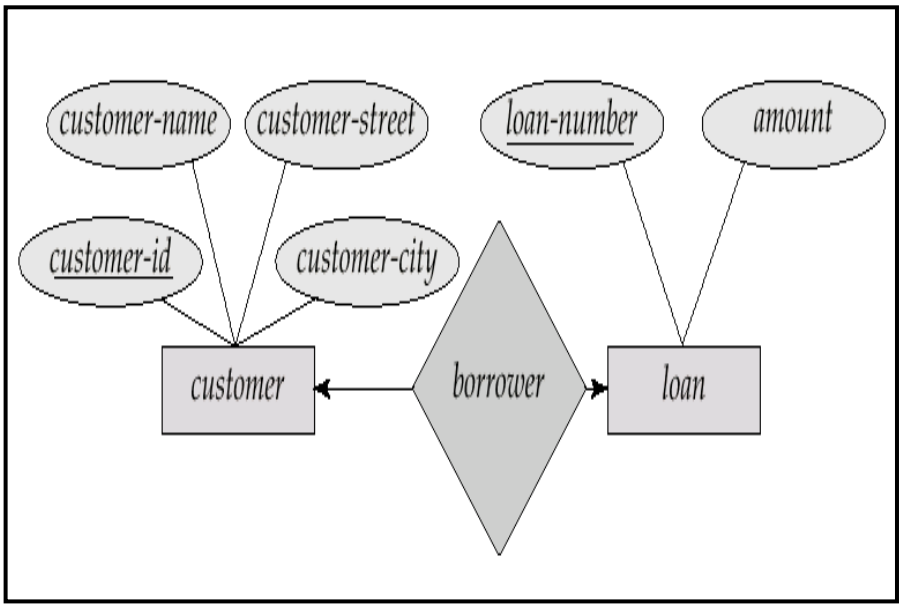


Figure 2.5.8 - Many-to-One

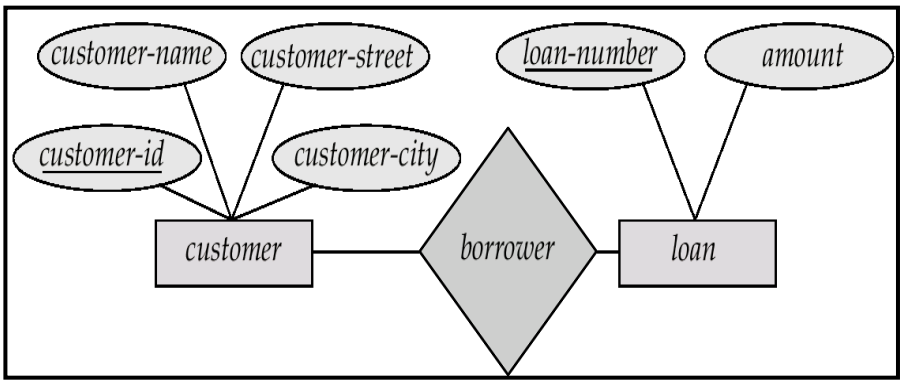


Figure 2.5.9 - Many-to-Many

The other way of representing cardinalities

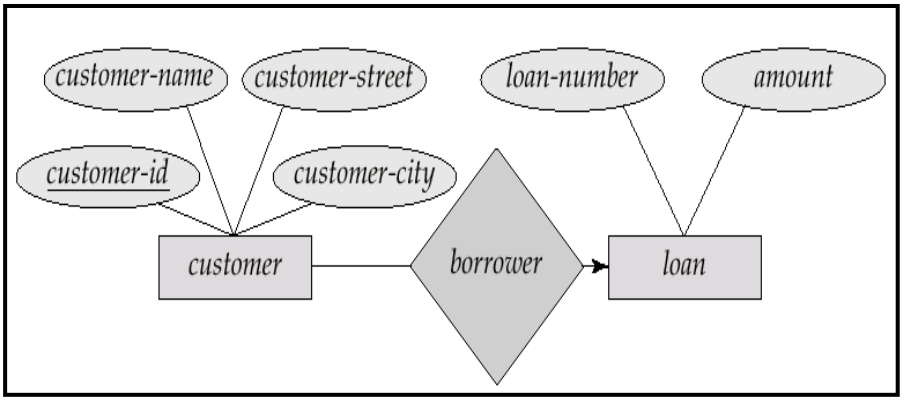


Figure 2.5.9

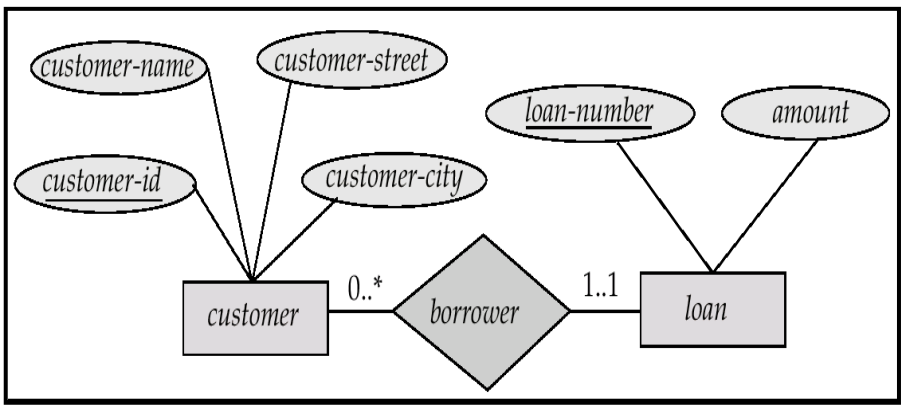


Figure 2.5.10

An alternative Notation for Cardinalities

Weak Entity Sets

An entity set may not have sufficient attributes to form a primary key. Such an entity set is termed a weak entity set. An entity set that has a primary key is termed as Strong Entity Set. For example:

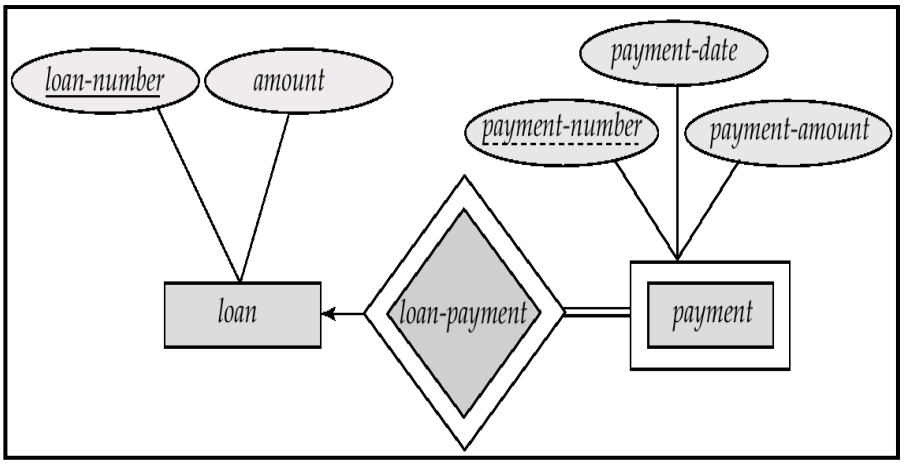


Figure 2.5.11

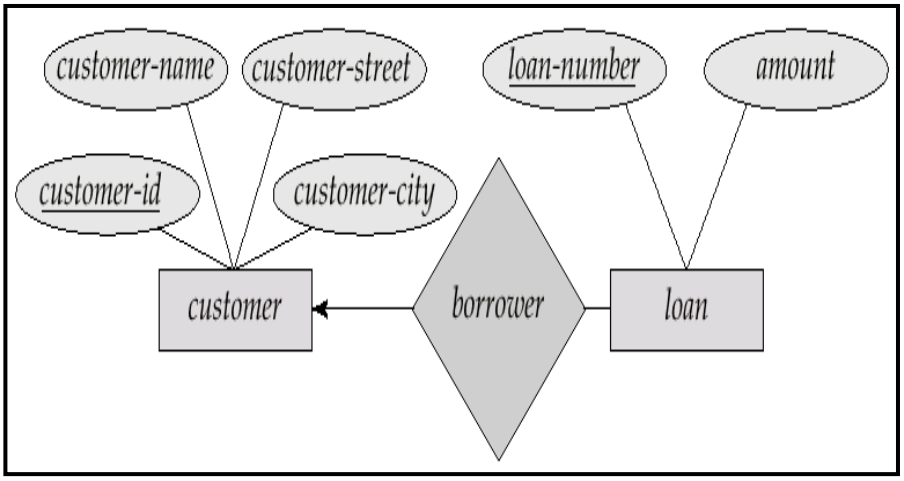


Figure 2.5.12

E-R Diagram for Banking Enterprise

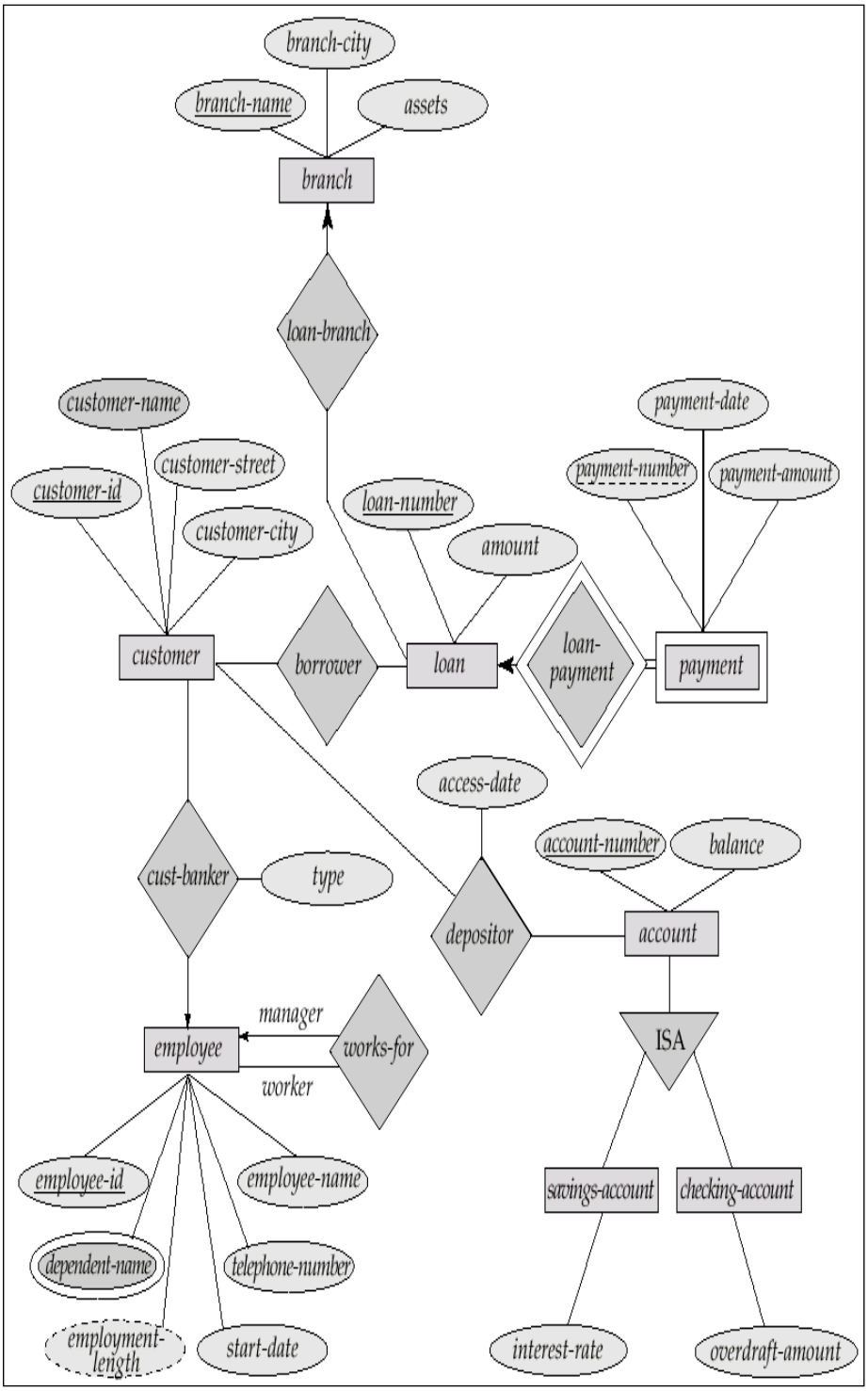


Figure 2.5.13

Summary

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, multi-valued and derived attribute.

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.

Multi-valued Attribute: A multi-valued 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.

Model Questions:

- What is an attribute? Explain various types of attributes?
- Explain Entity set and Relationship set?
- Write about mapping Cardinalities?
- Explain the importance of E-R Data Model?

Reference

Database System Concepts
Silberschatz, Korth, and Sudarshan

Modern Database Management
F. McFadden, J. Hoffer

3. EXTENDED E-R MODEL

Objective

The main objective of this lesson is to analyze the basic E-R diagrams in detail in such a way that the design and evaluation of database management systems becomes even more efficient.

After reading this chapter, you should understand:

- What is an Extended E-R Diagram?
- What is Generalization?
- What is Specialization?

Structure of the Lesson

Basic Concept
Specialization
Generalization
Constraints
Aggregation
UML
Summary
Technical Terms
Model Questions
References

Basic Concept

Although the basic concepts can model most database features, some aspects of a database may be more aptly expressed by certain extensions to the basic E-R model. In this section of lesson, we discuss extended E-R features of specialization, generalization, higher level and lower level entity sets, attribute inheritance and aggregation.

Specialization

An entity set may include subgroupings of entities that are distinct in some way from other entities in the set. For instance, a subset of entities within an entity 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 grouping. For example a person may be classified into the following:

- *Employee*
- *Customer*

Further a bank employee may further classified as the following

- Teller
- Officer
- Secretary

Now, the above entity can be represented as follows:

Specialization may be represented as triangle or circle. Here, 'ISA' is the representation. The ISA relationship may be referred as super class-subclass relationship.

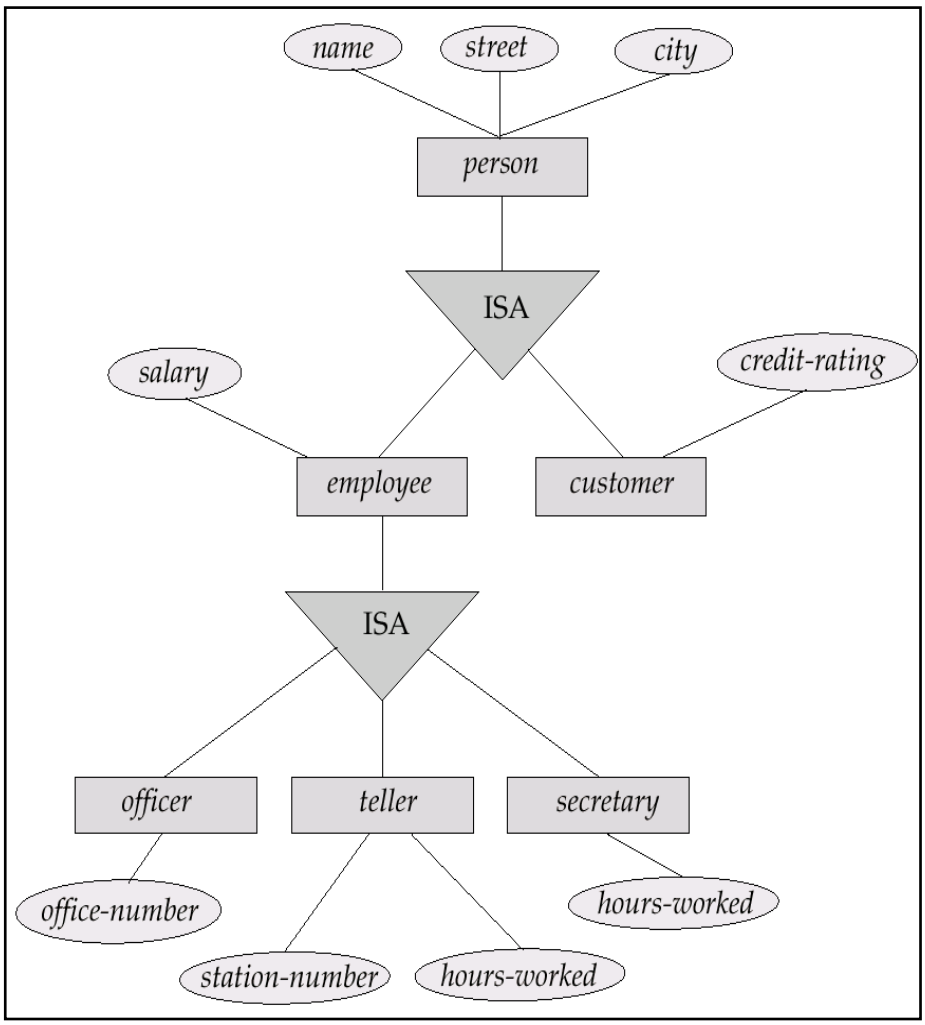


Figure 3.1

Generalization

The refinement from an initial entity set into successive levels of entity subgroupings represents a top-down design process in which distinctions are made explicit. Clearly we can form a generic entity type from sub types. Generalization and Specializations are inverse to one another.

The other examples are:

The entity Employee may be generated from two sub entity types (or sub types) called Part-time Employee and Full-Time Employee.
The entity Patient may be generated from two sub entity types (or sub types) called Inpatient and Outpatient.

The superclass entity type is also referred to as Higher level entity form (Ex. Employee)
The subclass entity type is reffered to as Lower level entity form (Ex. Teller, secretary and officer).

Constraints

Certain Constraints may be imposed on the extended E-R models. They are:

- Completeness Constraint
- Disjointness Constraint

Completeness Constraint is further divided into 2 constraints:

Total Specialization:

It means that each of the super type or higher form entity must any one of the sub type entity sets or lower forms.

For example, an Employee may be either the part-time or full-time, nothing beyond the scope.

Partial Specialization:

It means that each super type or higher form entity may not belong to any lower level entity sets.

For example, suppose a vehicle is defined as super class and car and bus are as the sub types. Now any vehicle may be either the car or bus and even it can be a cab.

Disjointness Constraint also further has two constraints:

Disjoint Rule:

It requires that an entity belong to no more than one lower level entity set. For example, an account type may be either savings or current but not both.

Overlapping Rule:

It requires that an entity may belong to more than one lower level entity set within a single generalization. For example, a Person may be both employee and customer as well.

Attribute Inheritance

A crucial property of the higher level and lower level entities created by specialization and generalization is attibute inheritance. The attributes of higher level or super type are said to be **inherited** by lower level entity sets. For example, customer and employee inherit the attributes of Person.

Aggregation

One limitation of E-R model is that it cannot express relationships among relationships. The best solution for this is to use aggregation. Aggregation is abstraction through which relationships are treated as higher-level entities. Observe the following examples:

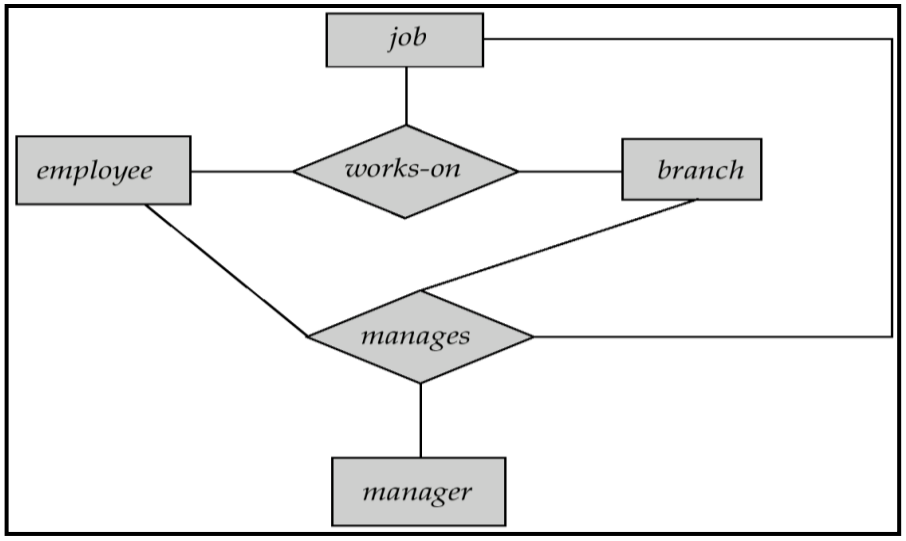
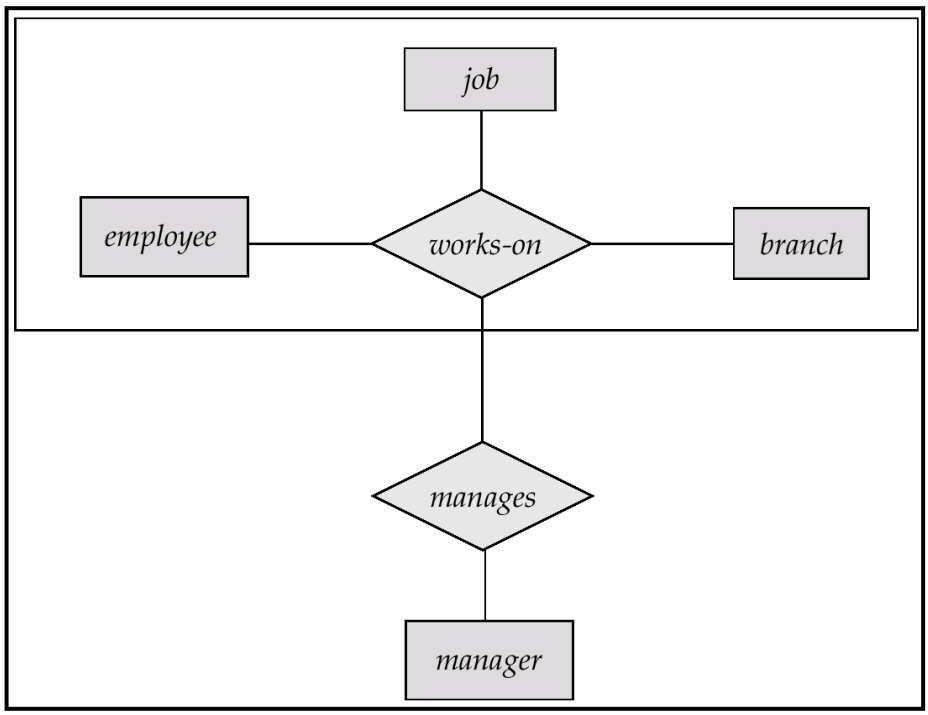


Figure 3.5.1 E-R diagram with redundant Relationships

We can eliminate this redundancy through aggregation. Observe the following diagram:



E-R Diagram with Aggregation

Figure 3.5.2

Summary of Symbols to be used:

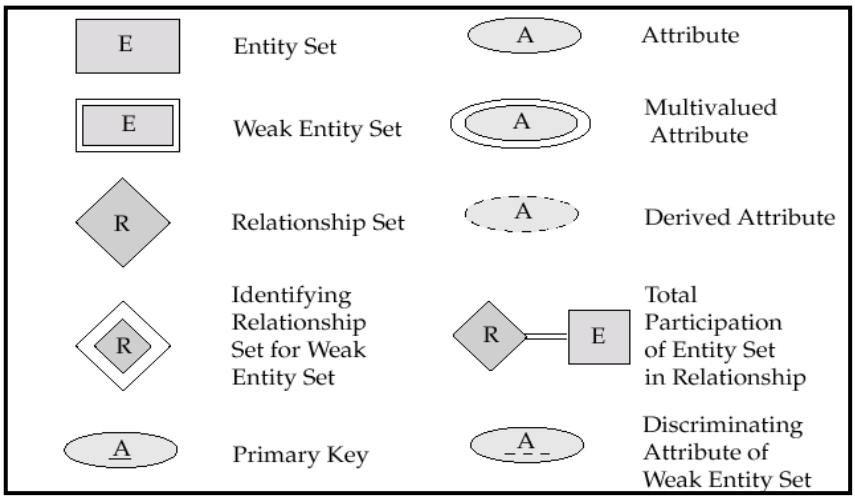


Figure 3.5.3

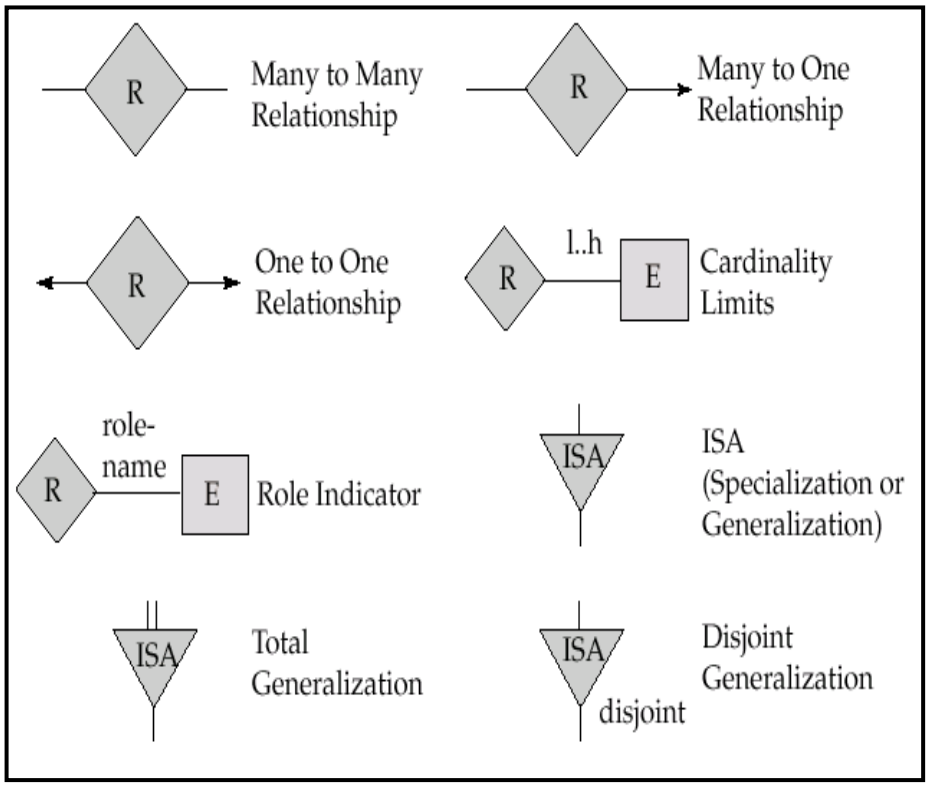


Figure 3.5.4

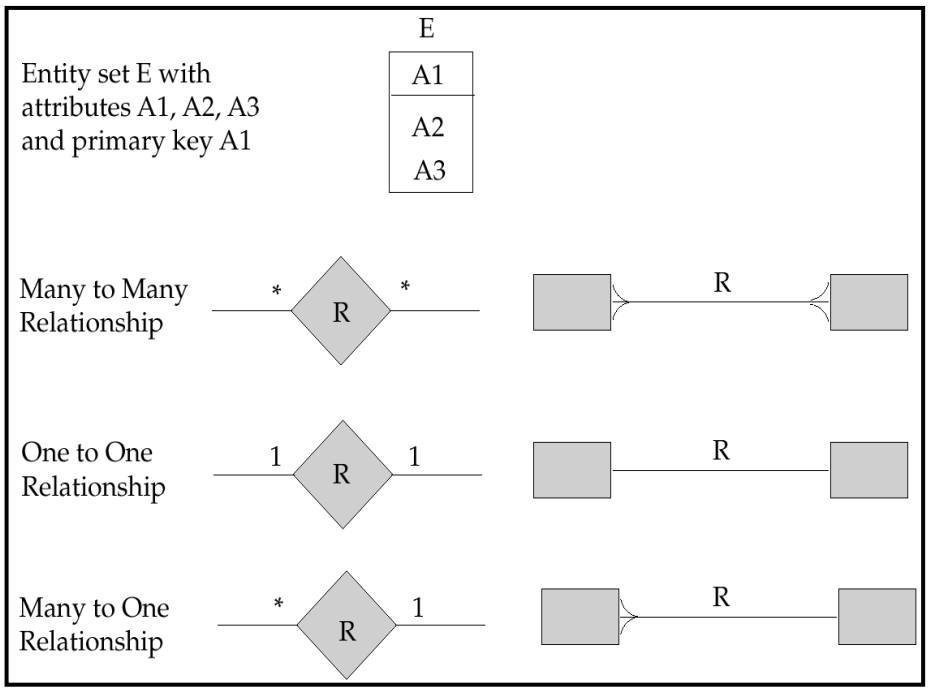


Figure 3.5.5

UML Diagrams (Unified Modeling Language)

The UML is a standard developed by OMG (Object Modeling Group) for creating specifications of various components of a software system. Some of the parts of UML are:

Class Diagram: A class diagram is similar to E-R Diagram.

Use Case Diagram: Use case diagrams show the interaction between users and the system, in particular the steps of tasks that users perform (such as drawing money or registering a course).

Activity Diagram: Activity Diagram depicts the flow of tasks between various components of a system.

Implementation Diagram: It shows the system components and their interconnections both at software level and hardware level.

Summary

Database design mainly involves the design of the database schema. The E-R data model is a widely used data model for database design. It provides a convenient graphical approach to view the data, relationships and constraints.

The specialization and Generalization gives more affective way of representation of entities. Aggregation is an abstraction in which relationship sets are treated as higher form of entities.

The UML provides a graphical means of modeling various components of a software system.

Technical Terms

Specialization: An entity set may include subgroupings of entities that are distinct in some way from other entities in the set.

Generalization: It is a process of generating a higher-level entity type from sub types

UML: The UML provides a graphical means of modeling various components of a software system.

Class Diagram: A class diagram is similar to E-R Diagram.

Use Case Diagram: Use case diagrams show the interaction between users and the system, in particular the steps of tasks that users perform (such as drawing money or registering a course).

Activity Diagram: Activity Diagram depicts the flow of tasks between various components of a system.

Implementation Diagram: It shows the system components and their interconnections both at software level and hardware level.

Model Questions

1. What is an E-E-R model?
2. Differentiate Specialization, Generalization and Aggregation?
3. Explain Attribute Inheritance?

Reference

Database System Concepts
Silberschatz, Korth, and Sudarshan

Modern Database Management
F. McFadden, J. Hoffer

UNIT - II

1. RELATIONAL DATA MODEL

Objective

After completion of this lesson, we can understand,

- What is a Relation?
- What are the various relational algebra operations?

Structure of Lesson

Introduction
- Basic Structure
- Database Schema
- Keys
- Query Languages

Relational Algebra
Additional Relational Algebra Operations
Extended Relational Algebra Operations
Null values

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-dimension 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. Literally a relation is nothing but a table. 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 dom (A_1), dom (A_2)... dom (A_n) which is a subset of the Cartesian product of the domains that define R:

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.

Properties of a Relation

Relations possess certain properties. All of them are immediate consequences of the definition of relation given earlier, and all of them are very essential. The properties of a relation are as follows:

- There are no duplicate tuples
- Tuples are unordered, top to bottom
- Attributes are unordered, left to right
- All attribute values are atomic

The **degree** of a relation is the number of attributes n of its relational schema.

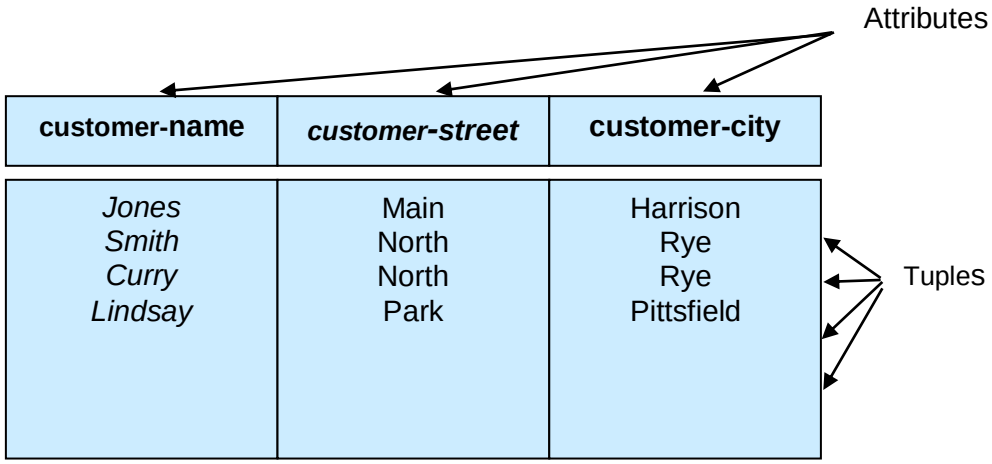
The **domain** D is set of atomic values. Atomic means each value in the domain is an individual item. Examples of a domain:

- USA_Phone_Numbers: The set of 10-digit phone numbers valid in United States.
- Employee_Ages : A set of age values ranges from 18 to 58.

Example of relation:

account-number	*branch-name*	*balance*
A-101	Downtown	500
A-102	Perryridge	400
A-201	Brighton	900
A-215	Mianus	700
A-217	Brighton	750
A-222	Redwood	700
A-305	Round Hill	350

Figure 1.1.1



Customer

Codd Rules

In 1985, E.F Codd proposed an informal set of twelve rules by which a database could be evaluated to see how relational it is. Very few commercial databases exist which meet or satisfy all twelve rules. The rules are:

1. All information in a relational database is represented explicitly at the logical level and in exactly one way – by values in tables.

2. Each and every datum (atomic value) in a relational database is guaranteed to be logically accessible by resorting to a combination of table name, primary key and column value. If a database satisfies rule 2, every atomic value should be easily retrievable.

3. Null values are supported in a fully relational DBMS for representing missing information in a systematic way, independent of data type. A null means an unknown value or not applicable value (or irrelevant).

4. The database description is represented at the logical level in the same way as ordinary data, so that authorized users can apply the same relational language to its interrogation.

5. A relational system may support several languages and various modes of terminal use. However they must support the following: data definition, view definition, data manipulation, authorization, integrity constraints, and transaction management.

6. All views that are theoretically updateable are also updateable by the system.

7. The capability of handling one or more relations with a single operation must be ensured.

8. Physical data Independence must be ensured.

9. Logical data Independence should also be ensured.

10. Integrity constraints specific to a particular relational database must be definable in a relational data sub language and stored in the catalog, not in the application programs.

11. A relational DBMS must have distribution independence.

12. If a relation system has a low-level language, that low-level language cannot be used to subvert or bypass the integrity rules and constraints expressed in the higher-level relational language.

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 ($attribute_1$:dom_1, ..., $attribute_n$:dom_n)

A **relation schema** e.g. employee (name, birthDate, ssn), 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, ssn)
Department (Name, empSSN, employeeName, function)

Keys

We must specify how tuples in the relation must be distinguished. This is expressed in terms of their attributes. That is, the values of the attributes of a tuple must be such that they can uniquely identify the tuple. In other words, no two tuples in a relation are allowed to have exactly the same value for all attributes.

A **Super Key** is a set of one or more attributes that, taken collectively, allow us to uniquely identify a tuple in a relation. For example, customer ID of a customer relation is a super key. Even the combination of customer ID and Customer Name can be the super key.

A key K is a **Candidate key** if K is minimal super key. Example: {*customer-name*} is a candidate key for *Customer*, since it is a super key {assuming no two customers can possibly have the same name), and no subset of it is a super key.

A **primary key** is an attribute that holds uniqueness property and not null property. Every table should possess a primary key

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.

Fundamentals of 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). $R(A_1, ..., A_n)$, $R.A_i$.

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 Operating selects the tuples that satisfy a given condition or predicate.

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

Relation *r*

A	*B*	*C*	*D*
α	α	1	7
α	β	5	7
β	β	12	3
β	β	23	10

$\sigma_{A=B \wedge D > 5}(r)$

A	B	C	D
α	α	1	7
β	β	2	1
		3	0

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. π

Example:

Relation *r*:

A	B	C
α	10	1
α	20	1
β	30	1
β	40	2

- $\prod_{A,C}(r)$

A	C
α	1
α	1
β	1
β	2

=

A	C
α	1
β	1
β	2

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 **U** 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.

Relations *r, s:*

A	B
α	1
α	2
β	1

r

A	B
α	2
β	3

s

r ∪ s:

A	B
α	1
α	2
β	1
β	3

Set Difference:

The set difference operation, denoted by **-**, allows finding 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 ensures 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.

Relations *r, s:*

A	B
α	1
α	2
β	1

r

A	B
α	2
β	3

s

r – s:

A	B
α	1
β	1

Cartesian product:

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

Relations *r, s*:

C	D	E
α	10	a
β	10	a
β	20	b
γ	10	b

s

A	B
α	1
β	2

r

r x *s*:

A	*B*	*C*	*D*	*E*
α	1	α	10	a
α	1	β	19	a
α	1	β	20	b
α	1	γ	10	b
β	2	α	10	a
β	2	β	10	a
β	2	β	20	b
β	2	γ	10	b

Renaming:

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

Additional Relational Operations

We define additional operations that do not add any power to the relational algebra, but that simplify common queries.

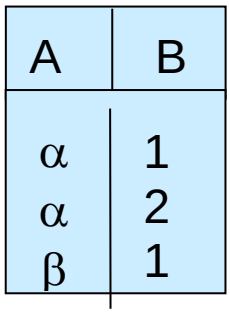
- Set intersection
- Natural join
- Division
- Assignment

Set intersection:

Finds the common tuples in two relations with like attributes.

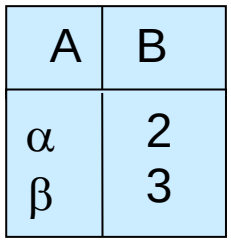
Notation: *r* Ç snDefined as: *r* Ç *s* ={ *t* | *t* Î *r* **and** *t* Î *s* }
Assume: *r*, *s* have the *same arity*
attributes of r and s are compatible
Note: *r* Ç *s* = *r* - (*r* - *s*)

Example



A	B
α	1
α	2
β	1

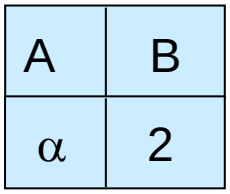
r



A	B
α	2
β	3

s

R ^ S



A	B
α	2

Divide*:*

Takes two relations, with attributes $\{X_1...X_N,Y_1...Y_M\}$ and $\{Y_1...Y_M\}$ respectively, and returns a relation with attributes $\{X_1...X_N\}$ 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|><|S is only those tuples of R×S that agree on some list of attributes.

The natural join may be defined by

1. Compute R×S
2. For each attribute A that names both a column in R and a column in S, select from R×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: employee(id,name), salary(id,salary); the natural join employee-salary(id,name,salary)

The *theta join* of two relations R and S denoted $R|><|_{C}S$ is only those tuples of R×S that satisfy the condition C.

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

Renaming
ρS(A1,...,An)(R) is the same relation as R but its name is S with the attributes named. A1,...,An.

Example:

Relations r, s:

A	B	C	D
α	1	α	a
β	2	γ	a
γ	4	β	b
α	1	γ	a
ς	2	ρ	b

r

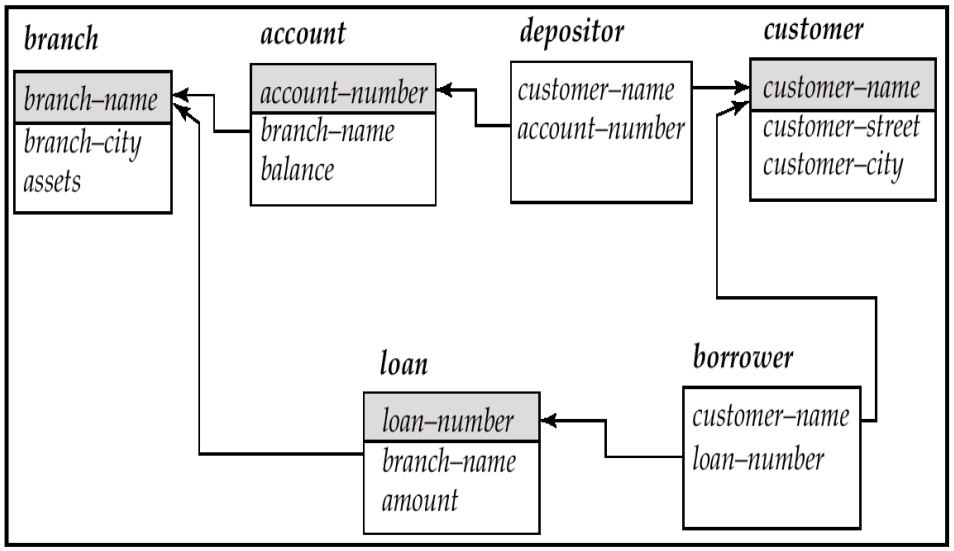
B	D	E
1	a	α
3	a	β
1	a	γ
2	b	δ
3	b	∈

s

$r \bowtie s$

A	B	C	D	E
α	1	α	a	α
α	1	α	a	γ
α	1	γ	a	α
α	1	γ	a	γ
δ	2	β	b	δ

Example Queries on Relational Algebra



$\sigma_{amount} > 1200 (loan)$

2. Find the loan number for each loan of an amount greater than $1200

$\Pi_{loan\text{-}number} (\sigma_{amount} > 1200 (loan))$

3. Find the names of all customers who have a loan, an account, or both, from the bank

$\Pi_{customer\text{-}name} (borrower) \cup \Pi_{customer\text{-}name} (depositor)$

4. Find the names of all customers who have a loan and an account at bank.

$\Pi_{customer\text{-}name} (borrower) \wedge \Pi_{customer\text{-}name} (depositor)$

5. Find the names of all customers who have a loan at the Perryridge branch.

$\Pi_{customer\text{-}name} (\sigma_{branch\text{-}name=\text{“}Perryridge\text{”}} (\sigma_{borrower.loan\text{-}number = loan.loan\text{-}number}(borrower \times loan)))$

6. Find the names of all customers who have a loan at the Perryridge branch but do not have an account at any branch of the bank.

$\Pi_{customer\text{-}name}$ ($\sigma_{branch\text{-}name = \text{“Perryridge”}}$ ($\sigma_{borrower.loan\text{-}number = loan.loan\text{-}number}$(borrower x loan))) – $\Pi_{customer\text{-}name}$(depositor)

SUMMARY

The relational algebra provides the basic set of operations to manipulate one ore more relations. In order to implement operations a set of symbols are kept in use. The operations include all the set operations along with additional set of operations like join and rename.

Model Questions

1. What is Relational Data Model? Explain.
2. List out various Relational Algebra operations with examples?
3. How relational algebra can be mapped on to SQL.

2. RELATIONAL CALCULUS

Objective

After completion of this chapter, you understand:

- Tuple Relational Calculus
- Domain Relational Calculus

Structure of Lesson:

Introduction
Tuple Relational Calculus
Domain Relational Calculus
Summary
Technical terms

Introduction

Relational Calculus combines SELECT and PROJECT commands into one command for listing the required attributes. While doing this, the WHERE clause will specify selection criterion.

Relational Calculus is an alternative to Relational Algebra. Relational Calculus is Non-Procedural where as Relational Algebra is Procedural. In Relational Calculus, the JOIN operation is implicit using WHERE clause which establishes associations between relations. This means that a single retrieval command can join several relations. But, in Relational Algebra, the Join operation is binary.

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) ∣ 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:

{*Query Target* | *Query Condition*}

The *Query Target* is a tuple variable, which ranges over tuples of values. The *Query Condition* is a logical expression such that

- It uses the *Query Target* and possibly some other variables.
- If a concrete tuple of values is substituted for each occurrence of the *Query Target* in *Query Condition*, 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 *Query Condition* is a logical expression of classical first-order logic.

1.INGRES & QUEL uses TUPLE RELATIONAL CALCULUS.

2.The expression of tuple calculus consists of tuple variable, conditions (<,>,<=
and >=) and Well formed formulas.

3.Tuple Relational Calculus is based on no.of tuple variables.

4.A tuple variable is mapped to an individual tuple from the database.

The general form of tuple calculus query is:

{ t | COND(t)}

Where t is a tuple and COND(t) is an expression involved in t.

The result of such query is that the set of all tuples t that satisfy COND(t).

For example

To find all the employees whose salary is above $5000.

{ t | EMPLOYEE(t) and t.SALARY > 5000 }

The other examples follow:

{ t.FNAME, t.LNAME | EMPLOYEE(t) and t.SALARY > 5000 }

Retrieve the birth date and address of employee whose name is Krishna Prasad P.

{ t.BDATE, t.ADDRESS | EMPLOYEE(t) and t.FNAME = 'Krishna' and t.LNAME = 'Prasad' and t.INIT = 'P' }

A General Expression of the tuple relational calculus is of the form:

{ t1.A1, t2.A2…….tn = An | COND(t1,t2……tn, tn+1……tm+n) }

In addition, two special symbols called quantifiers can appear in formulas. They are:

- Existential Quantifier (∃)
- Universal Quantifier (∀)

Example:

1. Retrieve all the employees names and address who work in research dept.

{ t.FNAME, t.LNAME | EMPLOYEE(t) and (∃d) (DEPARTMENT(d) and d.DNAME = 'Research' and d.DNUMBER = t.DNO) }

2. Make a list of project numbers for projects that involve a person 'Smith' as an employee.

{ p.PNUMBER | PROJECT(p) and (((∃e)(∃w) (EMPLOYEE(e) and WORKSON(w) and w.PNO = p.PNUMBER and e.LNAME = 'Smith' and e.SSN = w.ESSN) }

Domain relational calculus

Domain Relational Calculus is the second form of relational calculus. It uses domain values or variables that take on values from an attribute domain, rather than values for an entire tuple. However, the DRC is closely related to TRC.

DRC acts as a theoretical basis for Query by Example.

Queries in the DRC have the form:

{X_1, ..., X_n | *Condition*}

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

The language QBE is based on DOMAIN RELATIONAL CALCULUS. It differs from tuple calculus in the type of variables used. An expression is of the form:

{ X1,X2……..Xn | COND(X1,X2…….Xn, Xn+1,……..Xm+n) }

Example:

Retrieve the birth dates and address of the employee whose name is 'John B.Smith'

{ uv | ((∃q) (∃r) (∃s) (∃t) (∃w) (∃x) (∃y) (∃z) (EMPLOYEE(qrstuvwxyz) and q = 'John' and r = 'B' and s = 'Smith') }

or

{ uv | EMPLOYEE('John', 'Smith', 'B', t,u,v,w,x,y,z) }

Query by Example

QBE is the name of both Data manipulation language and an early database system that included this language. QBE has two versions: text based, graphical based. The following diagram illustrates QBE in Ms Access:

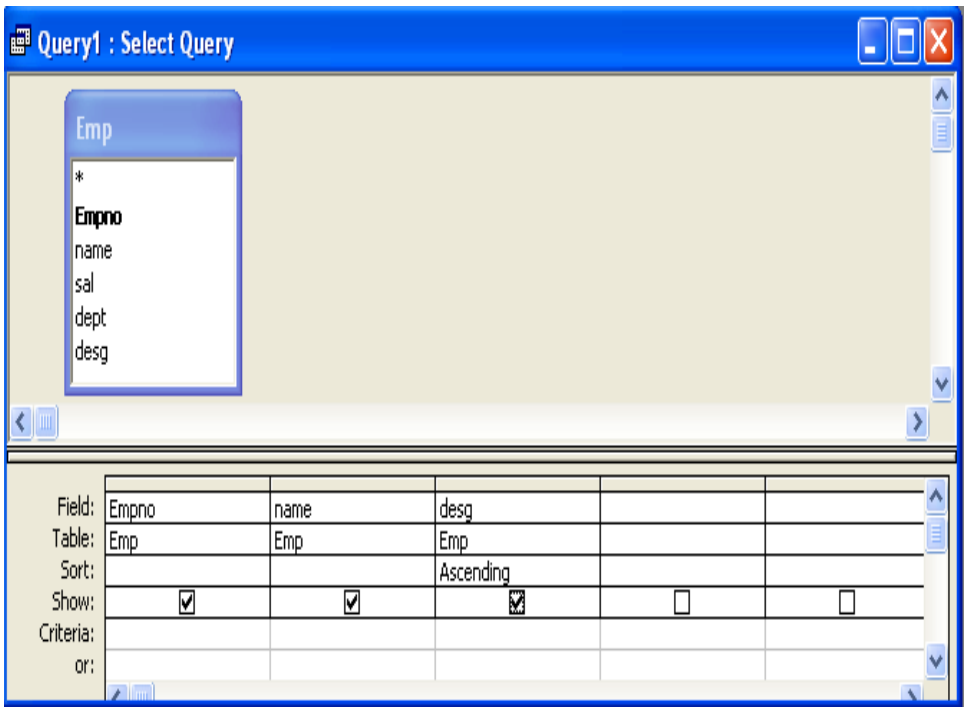


Figure 2.1 Example of QBE in MS-Access

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

The current chapter concentrates on relational calculus which provides an extension to the relational algebra. The relational calculus includes two basic categories: domain relational calculus and tuple relational calculus.

Domain Relational Calculus concentrates on a set of domains in the relations.

Tuple Relational Calculus plays a role in managing the rows in a relation.

Model Questions

1. Explain Relational calculus in detail.
2. Differentiate TRC and DRC with example.
3. Differentiate Relational Algebra and Relational calculus.

3. SQL-I

Objective

After reading this chapter, you will understand:

- What is SQL?
- What are the various components of SQL?
- Basic SQL commands.

Structure of the Lesson

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,

Server Side Tools

RDBMS Kernel	: Database Engine
ORACLE Workgroup or Enterprise Server	:It is the Database Server

Client side Tools

SQL * DBA	: Database Administrator's tool set
SQL * PLUS	: It is a separate Oracle Client side tool
PL / SQL	: Procedural Language SQL, allows Procedural processing of SQL statements.
DEVELOPER 2000	: ORACLE'S GUI tool for Forms. It does the job of front-end development.

Features of SQL

IBM developed the original version of SQL, originally called SEQUEL as a part of System R project in the early 1970s. 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, IBM'S DB2, 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.

Features:

- SQL is English like language.
- SQL is a non-procedural language.
- SQL is a 4GL (4th Generation 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.

The SQL language has several parts:

Data Definition Language (DDL): The SQL DDL provides commands for creating relational schemas (or tables), deletion of relations and restructuring the existing relations.

Data Manipulation Language (DML): The SQL DML includes a query language based on both relational algebra and tuple relational calculus (TRC). It includes all the commands to perform insertion, deletion and updation of rows in a relation.

Integrity: The SQL DDL provides all the integrity constraints to maintain correctness of data.

View Definition: The SQL DDL includes all the commands to generate views.

Transaction Control: SQL includes commands for specifying the “begin and end” transactions.

Embedded and Dynamic SQL: SQL defines how SQL statements can be embedded within general purpose programming languages like C, C++, Java, COBOL.

Authorization: The SQL DDL includes commands for specifying access rights to relations and views, as security to the relations is much important to maintain.

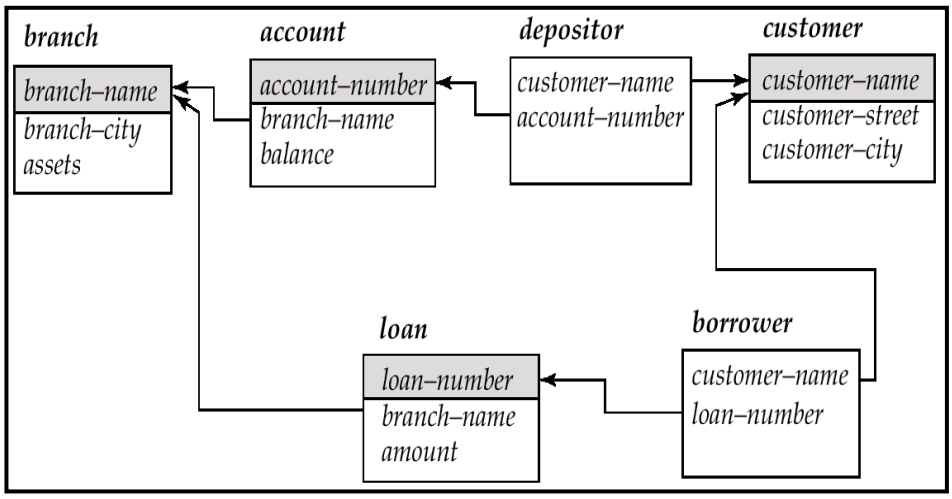
Database Objects:

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
- Synonyms
- Clusters
- Sequences
- Database triggers
- Stored functions and procedures
- Packages.

The basic relational schema used here in this lesson follows:



Note: The highlighted columns in the above relations are the primary key columns used to uniquely identify rows.

Data Definition

The set of relations in a database must be specified to the system by means of a data definition language. The DDL allows specification of not only a set of relations, but also information about each relation, including,

- The schema for each relation

- The domain of values associated with each attribute
- The integrity constraint
- Set of indices to be maintained for each relation
- The security and authorization information for each relation
- The Physical storage structure of each relation on disk

Basic Domain Types

The SQL standard supports a variety of built-in domain types (or SQL data types).

Char (n): a fixed length character string with user specified length n.

Varchar (n): a variable length character string with user specified length n.

Numeric (p, d): a fixed-point number with user specified precision. The decimal number consists of p digits and d of the digits after the decimal points.

Float (n): a floating point number, with precision of atleast n digits.
Date: it is of date format. It is of the form day-MON-year.

Basic Structure

Schema Definition follows:

Create table r (A1 D1, A2 D2…….An Dn,
{Integrity-constraint1},…..{integrity constraint k});

where r is the relation name, A1……An are the attribute names, D1…..Dn are the domain types. Integrity constraints are meant for validating the data to be inserted.

One of the Integrity Constraints that is essential is primary key constraint. It is for uniquely identifying a row or record in a relation.

To remove a relation from the existing database, use the following form:

Drop table r;

To alter the existing structure of a relation, use another instruction that is given below:

Alter table r {add/Modify} A D;

A is the name of attribute to be added/or modified.

We can drop attribute from the relation r as follows:

Alter table r drop A;

Basic Structure of SQL Queries

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

SELECT Statement

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 col1, col2, col3….col n from <relation name> where <condition>;

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

$$\Pi_{A_1, A_2, \ldots, A_n}(\sigma_P(r_1 \times r_2 \times \ldots \times r_m))$$

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.*

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"*

Tuple Variable

A tuple variable in SQL must be associated with a particular relation. Tuple variables are defined in the **form** 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 clasue by placing it after the name of the relations with which it is associated, with the keyword as inbetween .

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

String Operator

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.

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.

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 σ_θ, 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 $A_1, A_2, \ldots, A_n$

from $r_1, r_2, \ldots, r_m$

where *P*

is equivalent to the algebra expression

$$\Pi_{A_1, A_2, \ldots, A_n}(\sigma p(r_1 \times r_2 \times \ldots \times r_m))$$

using the multiset versions of the relational operators σ, Π, and $\times$.

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.

Aggregate Functions

The aggregate functions are the functions that take a collection (a set or multi set) 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.

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**

Nested Sub queries

SQL provides a mechanism for nesting subqueries. A subquery is a **select-from-where** expression that is nested with in 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 f*rom *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.

Example: To compare set elements in terms of inequalities, we can write

```
select distinct T.bname
                from branch T,branch S
                where T.assets > S.assets
        and S.bcity=``Burnaby''
```

or

```
select bname
        from branch
        where assets > some
                (select assets
                        from branch
                where bcity=``Burnaby'')
```

We can use any of the equality or inequality operators with **some**. If we change > **some** to > **all**, we find branches whose assets are greater than all branches in Burnaby

TEST FOR EMPTY RELATIONS: (**EXISTS**)

This operator tests whether a value is present in the list or not. If the value exists it returns True, otherwise it returns False.

Example : To Find all customers who have a loan and an account at the bank.

```
select cname from borrower where exists
        (select * from depositor
       where depositor.cname = borrower.cname)
```

TEST FOR THE ABSENCE OF DUPLICATE TUPLES: (**unique**)

The unique construct returns the value true if the argument subquery contains no duplicate rows.

Example : To Find all customers who have only one account at the SFU branch.

```
select T.cname  from depositor as T
                where unique (select R.cname
                                from account, depositor as R
                where T.cname = R.cname and
                        R.account# = account.account# and
                                account.bname = ``SFU'')
```

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'.* 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.

Model Questions:

1. Explain the features of SQL.
2. Explain the Structure of SQL Statements in detail.
3. What is a Sub query? Explain with Examples?
4. What is an aggregate function? List out various aggregate functions supported by SQL.

4. SQL- II

Objective

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

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#)
```

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;
```

Modification 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.

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

loan-number	***branch-name***	***Amount***
L-170	Downtown	3000
L-230	Redwood	4000
L-260	Perry ridge	1700

Borrower

customer-name	***loan-number***
Jones	L-170
Smith	L-230
Hayes	L-155

Inner join:

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

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

Result of *loan* inner join *borrower*.

loan-no	brnch-name	amount	cust-name	Loan-no
L-170 L-230	Downtown Redwood	3000 4000	Jones Smith	L-170 L-230

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

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*

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.

Join conditions: natural, **on** predicate**, using** $(A_1, A_2, \ldots, A_n)$.

The use of join condition is mandatory for outer joins, but is optional for inner joins (if it is omitted, a Cartesian product results).

Embedded SQL

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

- 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.
- 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

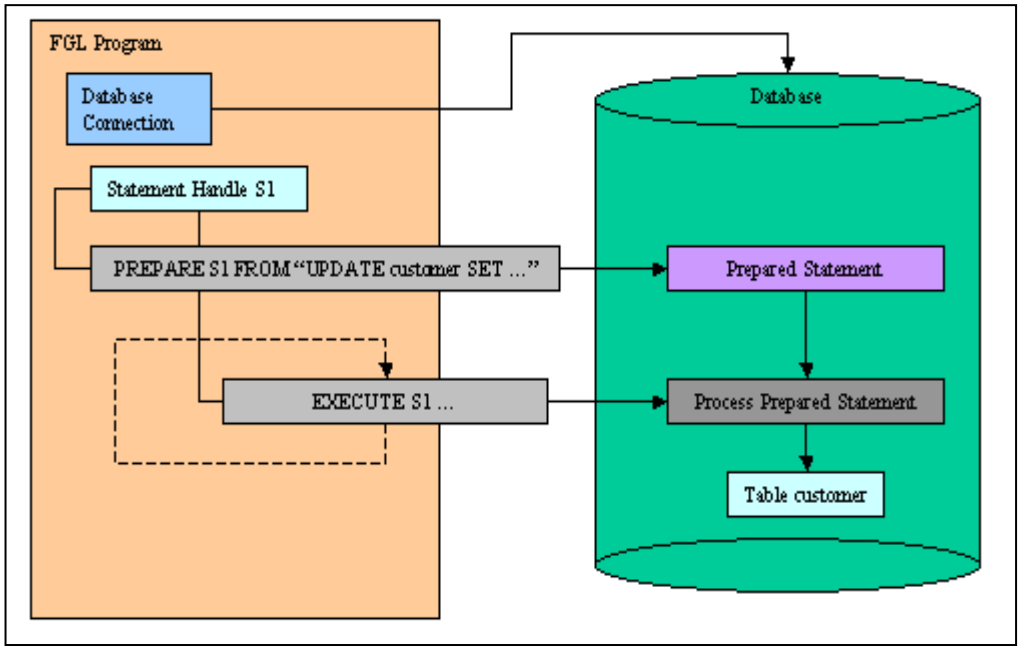
Embedded SQL can execute any valid **update**, **insert**, or **delete** statements.

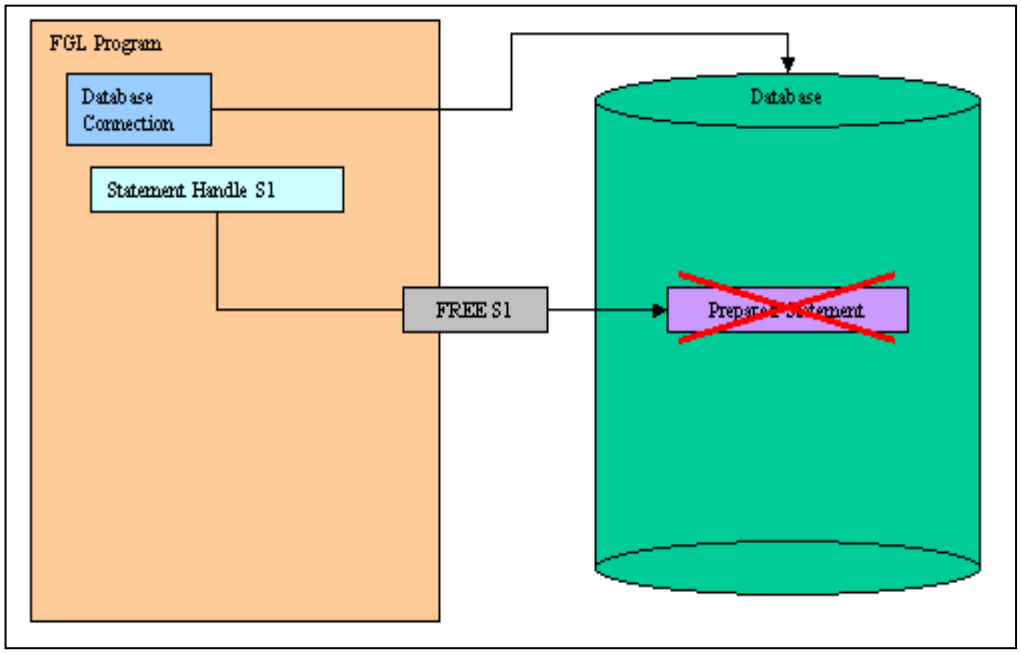
Dynamic SQL

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 you execute the prepared statement one or more times:

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


FGL Program
Database Connection
Statement Handle S1
PREPARE S1 FROM "UPDATE customer SET ..."
EXECUTE S1 ...
Database
Prepared Statement
Process Prepared Statement
Table customer


FGL Program
Database Connection
Statement Handle S1
FREE S1
Database
Prepared Statement

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.

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

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.

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. Storage & File Structure

Objective

- Different types of Physical Storage Media
- RAID

Structure of the Lesson

Overview of Physical Storage Media
Magnetic Disks
- Physical Characteristics of Disks
- Performance Measures of Disks
- Optimization of Disk-Block Access

RAID
- Improvement of Reliability and Redundancy
- RAID Levels
- Choice of RAID level
- Hardware Issues

Tertiary Storage
- Optical Disks
- Magnetic Tapes

Storage Access
- Buffer Manager
- Buffer-Replacement Policies

Technical terms
Model questions

Overview of Physical Storage Media:

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

- Cache
- Main Memory
- Flash Memories
- Magnetic Disk Storage

- Optical Storage
- Tape Storage

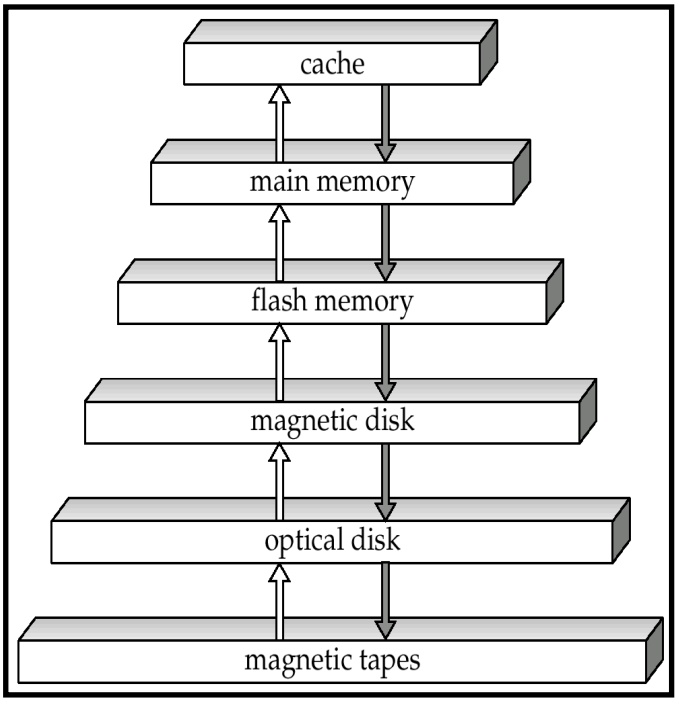


Figure 6.1 Memory Hierarchy

Cache:

The cache is the fastest and most expensive among all the other mediums. Cache memory is very small; its usage is managed by the computer hardware. It improves the efficiency of the hardware. However, the database system is nowhere concern about the management of cache.

Main Memory:

The storage medium used for data that are available to be operated is main memory. The general-purpose machine instructions operate on main memory. It is smaller in size though it holds mega bytes and gigabytes of storage. The content of main memory is usually lost as the power supply is lost.

Flash Memory:

Flash memory differs from main memory in that data survive power failure. Reading data from the flash memory takes less than 100 nanoseconds, which is roughly as fast as reading data from

main memory. However, writing data to flash memory is more complicated- data can be written once, which takes about 4 to 10 milliseconds, but cannot be overwritten directly. Flash memory is a form of electrically erasable programmable read-only memory (EEPROM).

Flash memories are more popular as a replacement for magnetic disks for storing small volumes of data. Flash memories are portable in size. Universal Serial Bus (USB) acts as an interface for flash memories. They can be used in digital cameras, video cameras and other devices.

Magnetic disk storage:

The physical medium for the long term on line storage of data is the magnetic disk. Usually, the entire database is stored on magnetic disk. The data is moved from disk to main memory so that it is accessed properly.

The size of the magnetic disks currently ranges from a few gigabytes to 400 gigabytes. Disk storage survives power failures and system crashes.

Optical Storage:

The most popular forms of optical storage are the compact disk (CD), which can hold about 700 megabytes of data and has a playtime of about 80 minutes, and the digital video disk (DVD), which can hold 4.7 or 8.5 gigabytes of data per side of the disk. The optical disks are available in read only forms like CD-ROM and DVD-ROM.

Tape Storage:

Non-volatile, used primarily for backup (to recover from disk failure), and for archival data. Tape storage is referred as sequential storage. It is much slower than disk. Very high capacity (40 to 300 GB tapes available). Tape can be removed from drive, storage costs much cheaper than disk, but drives are expensive. Tape jukeboxes are available for storing massive amounts of data.

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.

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.

Physical Characteristics of Disk

Physically, disks are relatively simple 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.

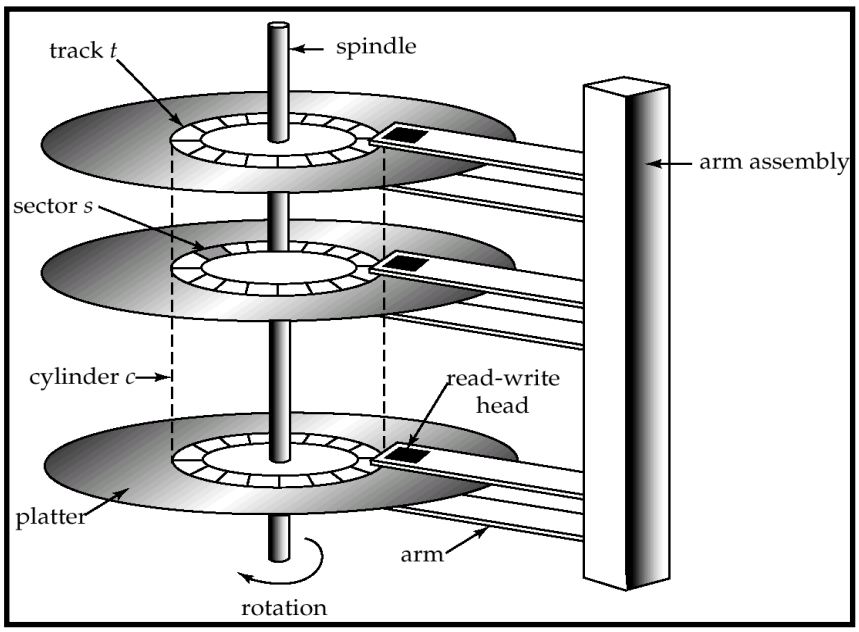


Figure 6.2 Magnetic Disk

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 $\mathbf{i^{th}}$ track, the tracks on all the other platters are also on the $\mathbf{i^{th}}$ track. Hence the ith tracks of all the platters together are called the ith **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 number of common interfaces for connecting disks to personal computers and workstations:

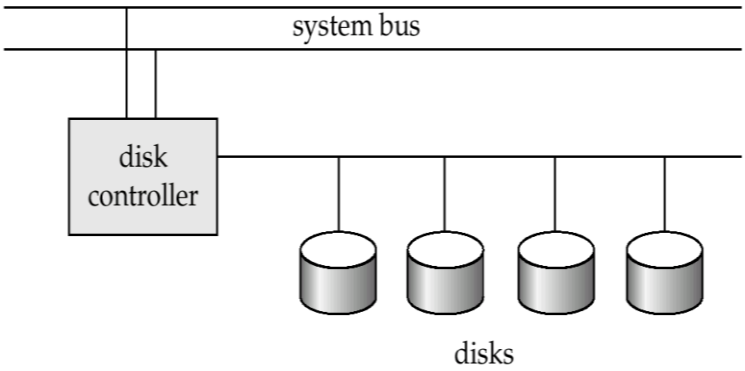


Figure 6.3 Disk Subsystem

- The **AT** attachment (**ATA**) interface.
- The new version of **ATA**, which is **STA** serial **ATA**.
- The small-computer-system-interconnect (**SCSI**).

Performance of Measured Disks

The main measures of the qualities of disk are capacity, access time, data-transfer rate, and reliability.

Access time is the time 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 average seek time the average seek time is 1/3. Taking these factors into account, the average seek time is the 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 seeks time and the rotational latency, and ranges 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 that, 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.

Optimization of Disk-Block Access

Requests for disk I/O are generated both by the file system and by the virtual memory manger 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 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 reverse direction and repeat.

RAID (Redundant Array of Inexpensive Disks)

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.

Improvement of Reliability and Redundancy

Reliability:

The chance that some disk 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 100,000 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 still available in the other. Data loss would occur only if a disk fails, 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.

E.g. MTTF (mean time to failure) of 100,000 hours, mean time to repair of 10 hours gives mean time to data loss of $500*10^6$ hours (or 57,000 years) for a mirrored pair of disks (ignoring dependent failure modes).

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 (a)** shows an array of size 4.

RAID Level 1:

This level refers to disk mirroring with block striping. **Figure (b)** shows a mirrored organization that holds four disks worth of data.

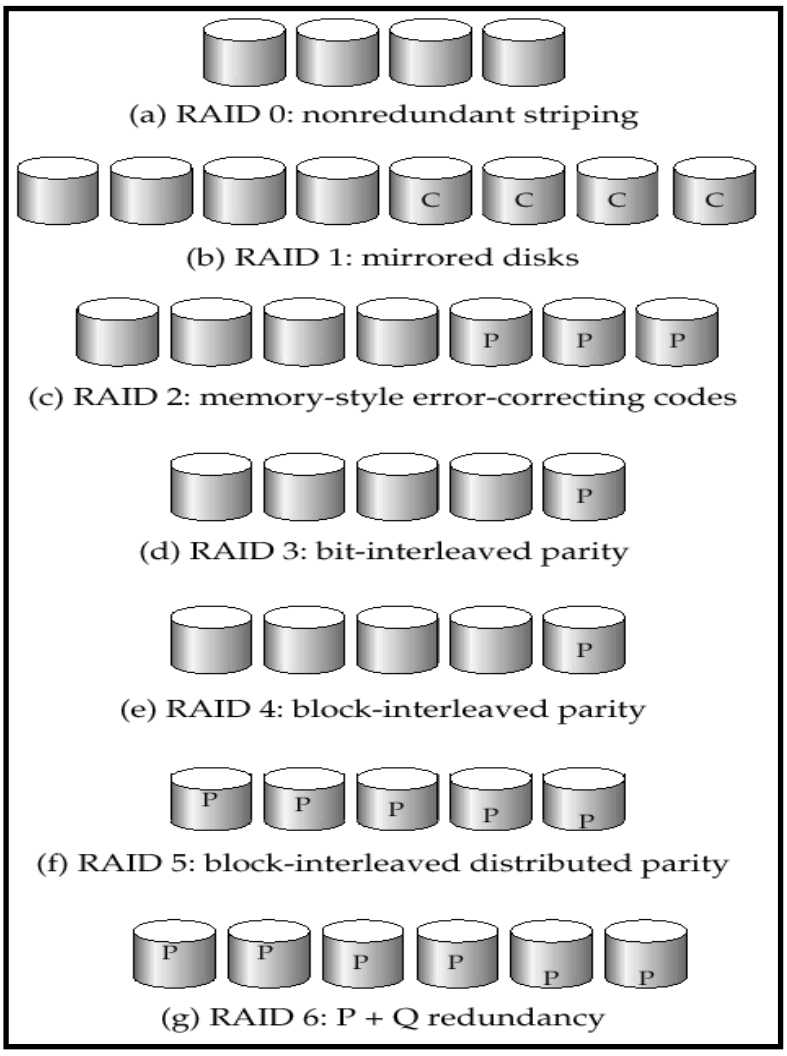


Figure 6.4 RAID levels

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(c)** 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.

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 (d)** 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 (e)**. 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 are 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 (f)** 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^{th} set of blocks is stored on disk (*n mod* 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 (g)**, 2 bits of redundant data are stored for every 4 bits of data.

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 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.

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.

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.

Tertiary Storage

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 megabytes), and they are cheap to mass-produce.

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 gigabytes of data (in one recording layer), while disks in the DVD-9 format can store 8.5 gigabytes 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 gigabytes and 17 gigabytes, 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 derives. Rotational speeds of CD drives originally corresponded to the audio CD standards, and the speeds of DVD drives originally corresponded to the DVD video standards, but current-generation drives rotate at many times the standard rate.

Data-transfer rates are somewhat less than for magnetic disks. Current CD derives read at around 3 to 6 megabytes per second, and current DVD drives read at 8 to 20 megabytes per second. Like magnetic-disk drives, optical disks store more data in outside tracks and less data in inner tracks. The transfer rate of optical derives is characterized as n x, which means the derive 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).

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 approaching 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 40 gigabytes with the **Digital Linear Tape** (DLT)

format, 100 gigabytes and higher with the Ultrium format, to 330 gigabytes(with **Ampex helical scan** tape formats). Data-transfer rates are of the order of a few to tens of megabytes per second.

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**.

Storage Access

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.

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.

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 devise.

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.

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 moving 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.

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. INTEGRITY & SECURITY

Objective

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

Structure of the Lesson

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'))

Reference 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
$t_1[K_1] = t_2[a]$.

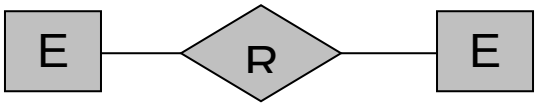
Referential integrity constraint also called **subset dependency** since its can be written as

$$\Pi_a (r_2) \text{ Í } \Pi_{K1} (r_1)$$

Referential Integrity in E-R model

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.

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.

$$\prod_{\alpha}(r_2) \subseteq \prod_{K}(r_1)$$

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 $t_1[K] = t_2[\alpha]$ that is

$$t_2[\alpha] \in \prod_{K}(r_1)$$

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:

$$\sigma_{\alpha = t1[K]}(r_2)$$

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 t2 is updated in relation r2 and the update modifies values for foreign key **a**, then a test similar to the insert case is made: Let t2' denote the new value of tuple t2.

The system must ensure that

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

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

$$\sigma_{\alpha = t1[K]}(r_2)$$

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.

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.

```
create table customer
(customer-name char(20),
customer-street char(30),
customer-city char(30),
primary key (customer-name))

create table branch
(branch-name char(15),
branch-city char(30),
assets integer,
primary key (branch-name))
```

create table *account*
(account-number char(10)**,**
branch-name char(15),
balance integer,
primary key (*account-number),*
foreign key (*branch-name)* **references** *branch)*

create table *depositor*
(customer-name char(20)**,**
account-number char(10)**,**
primary key *(customer-name, account-number),*
foreign key *(account-number)* **references** *account,*
foreign key *(customer-name)* **references** *customer)*

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.

Assertions

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.

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.

Need of 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.

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**

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.

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.

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.

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 or 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*

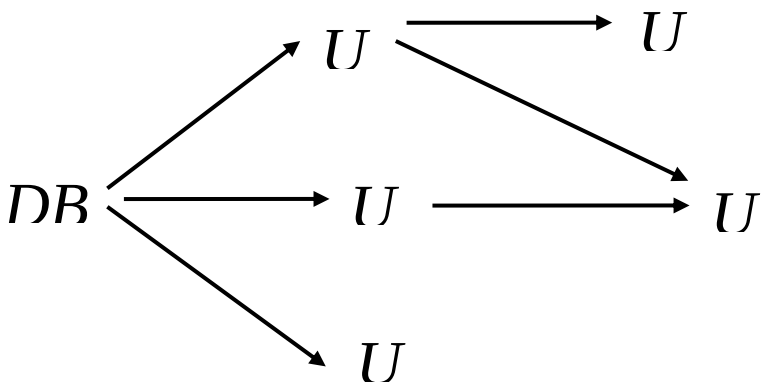
When the query processor translates the result into a query on the actual relations in the database, we obtain a query on *borrower* and *loan*. Authorization must be checked on the clerk's query before query processing replaces a view by the definition of the view.

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.*

Granting Previleges

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 $U_i \rightarrow U_j$ indicates that user U_i has granted update authorization on loan to U_j.



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.

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

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.

Limitations of SQL authorizations

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 allover an application
- Checking for absence of authorization loopholes becomes very difficult since it requires reading large amounts of application code

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 **encrypt**ed 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 from 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.

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.

Model Questions

1. What is a Constraint? Explain the need of the constraints in database management.
2. Why triggers are so important in databases?
3. Discuss about privileges in SQL.
4. Discuss about Security and Authorization.

7. INDEXING AND HASHING

Objective

- Index-sequential file organization.
- Algorithms for updating indices.
- How B^{+}-tree is more advantageous than index-sequential file organization.
- Updations on B^{+} -trees.

Structure of the Lesson

Basic Concepts
Ordered Indices
    Primary Index
        Dense and Sparse Indices
        Multilevel Indices
        Index Update
Secondary Indices
B^{+} -Tree Files
Structure of a B^{+} -Tree
Updates on B^{+} -Trees
B^{+} -Tree File Organization
B^{+} -Tree Index Files
Multiple Key Access
Static Hashing
Dynamic hashing
Comparison of Order Indexing and Hashing
Index Definition in SQL
Summary
Technical terms

Model questions

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 worth- while 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.

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.

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.

Figure 1 shows a sequential file of account records taken from our banking example.

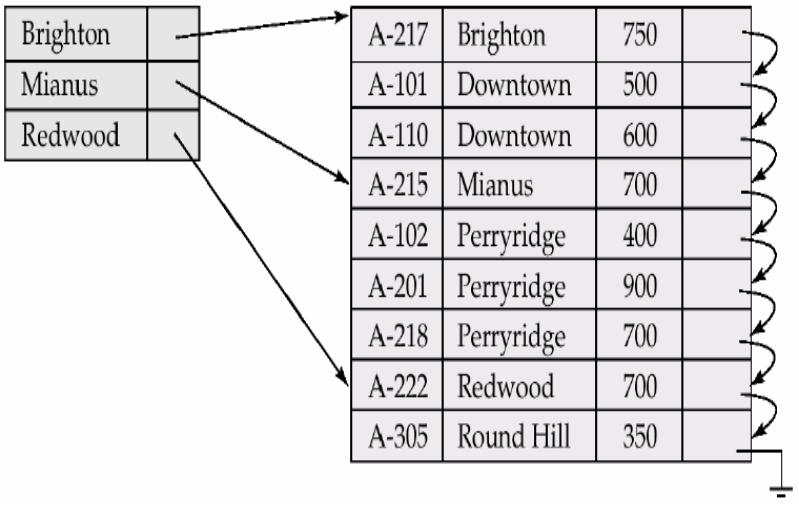


Figure 1 Sequential file for account records.

In the example of Figure 1, the records are stored in search-key order, with branch name used as the search key.

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:

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, since, 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.

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.

Figure 2 and Figure 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 Figure 2, we follow the pointer directly to the first Perryridge record.

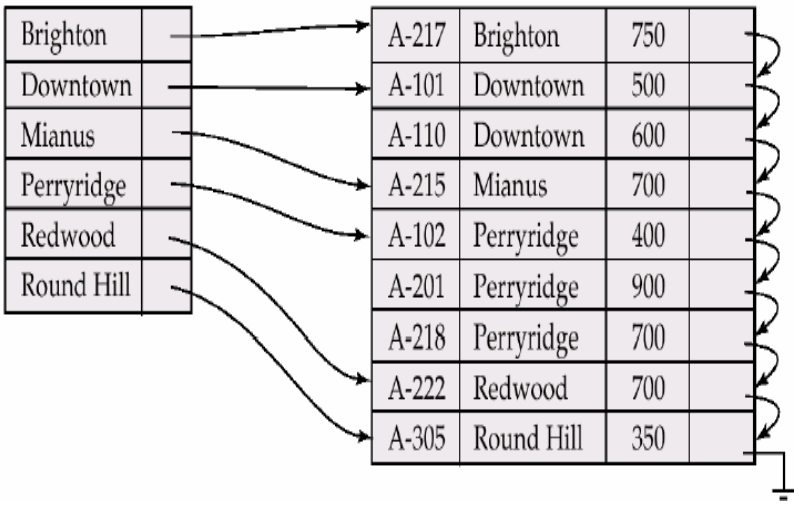


Figure 2 Dense Index

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.

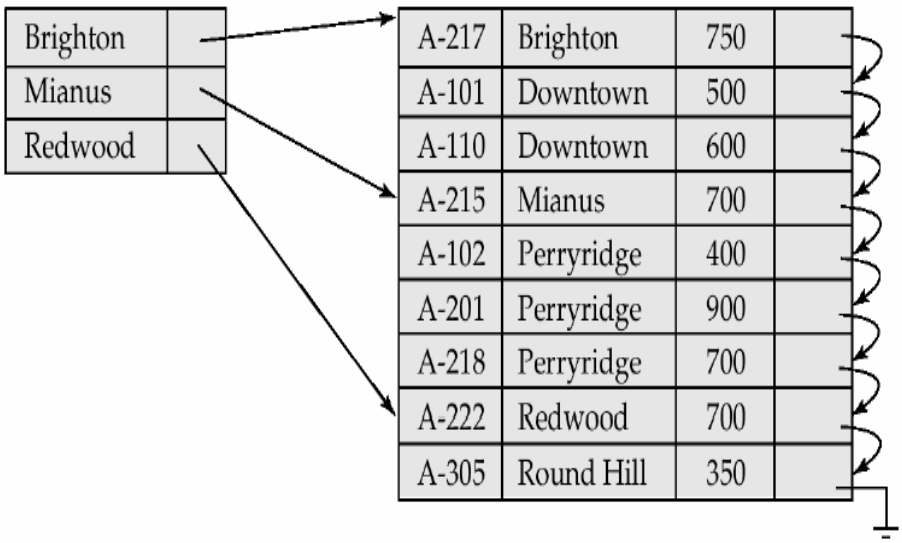


Figure 3 Sparse Index

If we are using the sparse index (Figure 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 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.

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 $+\log_2(b)+$ blocks to read ($+ x +$ 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 read takes 30milliseconds, the search will take 210milliseconds, 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 Figure 4.

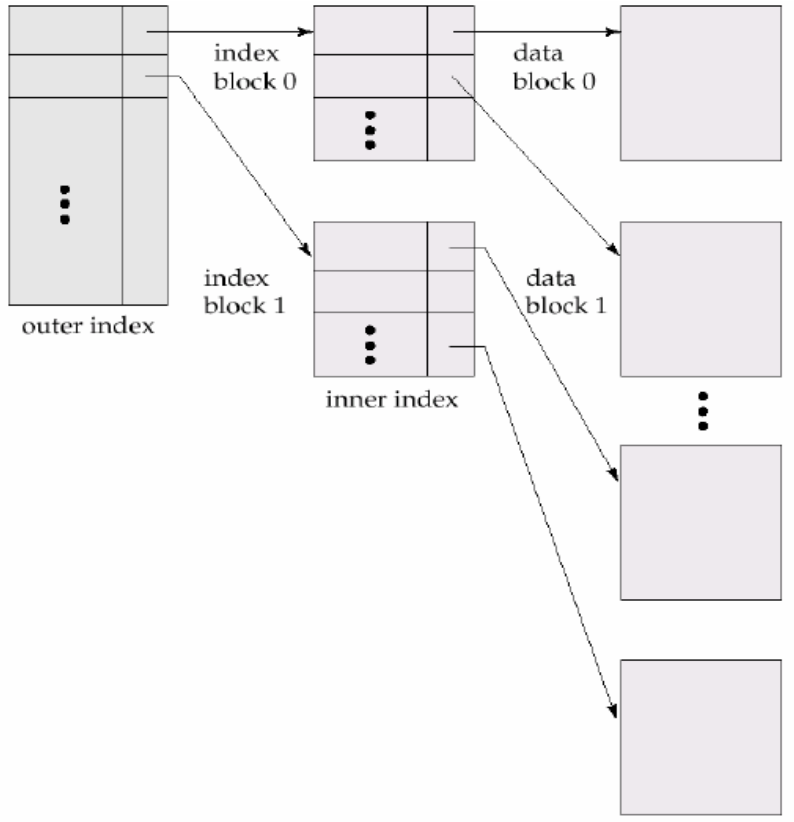


Figure 4. Two-level sparse index (Multilevel Index)

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.

Index Update

Regardless of what form of index is used, every index must be updated 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:

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.

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:

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.

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. Other wise the system takes the following actions:

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.

Other wise, 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 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.

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 any where in the file and, in general, we cannot find them with out 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, if 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 any where 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 on account file, on non-candidate key balance.

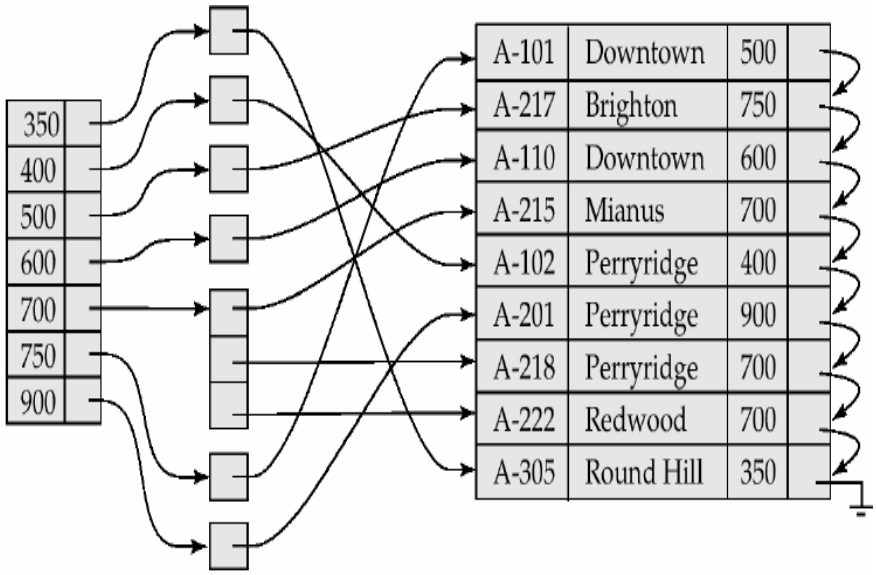


Figure 5. Secondary index an *account* file, on non-candidate key *balance*

Figure 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. How ever, 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.

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 [n/2] and n children, where n is fixed for a particular tree.

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.

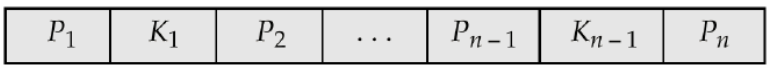


Figure 6 Typical node of a B^+ -tree

Figure 6 shows a typical node of a B^+ -tree .It contains up to n-1 search key values k_1, $K_2 \ldots k_{n-1}$, and n pointers p_1, $p_2 \ldots p_n$. 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,…., 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.

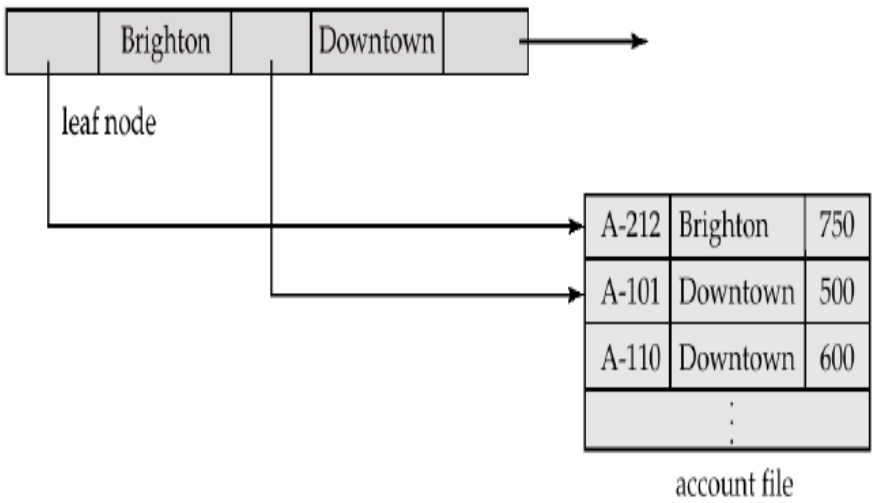


Figure 7 A leaf node for account B^+ - tree index (n=3)

Figure 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 [(n-1)/2] 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 [n/2] 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,…,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 Pm 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 [n/2] 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.

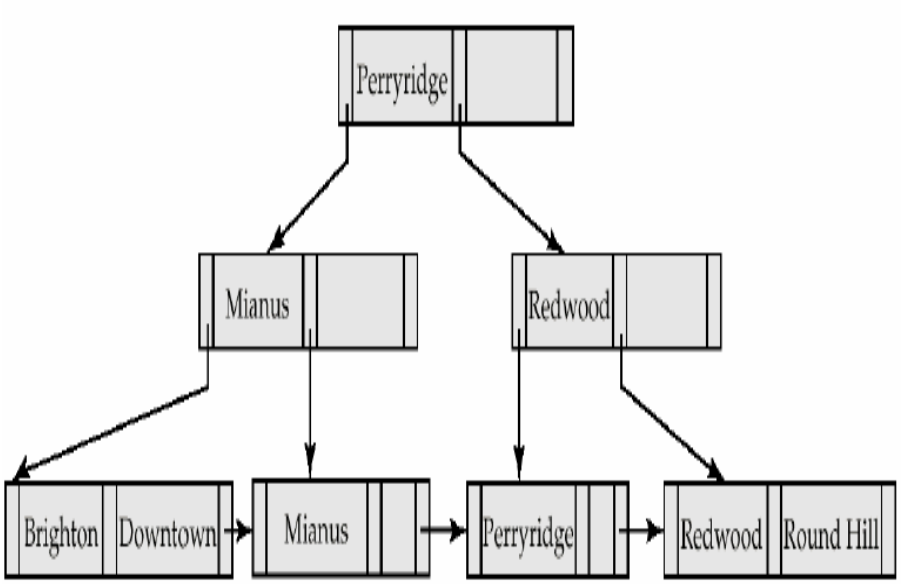


Figure 8 B^+ -tree for account file (n=3)

Figure 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. Figure 9 shows a B^+ -tree for the *account* file with n=5.

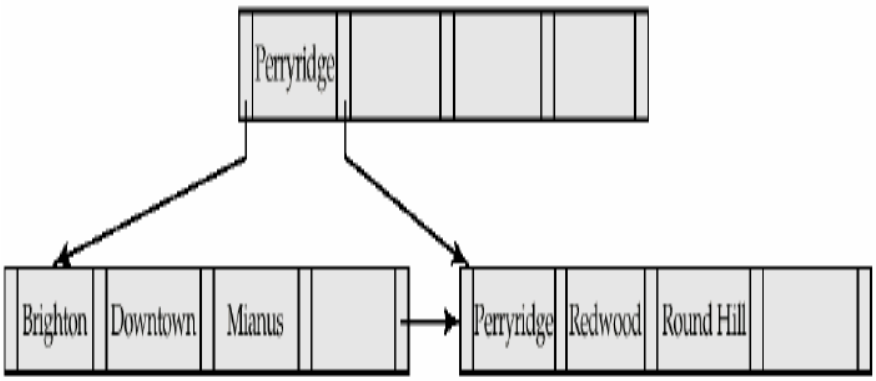


Figure 9 B^+-tree for *account* file with n=5.

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 [n/2] 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 bucker 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. Figure 10 shows the two leaf nodes that result from inserting “Clearview” and splitting the node containing “Brighton” and “Downtown”.

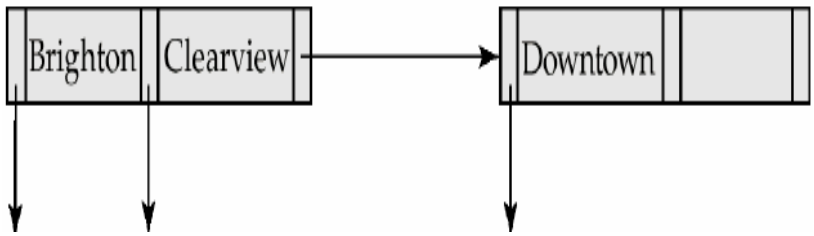


Figure 10 Split of leaf node on insertion of “Clearview”

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 [n/2] in the existing node and the remaining values in a new node.

Having spilt 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 t he parent of the leaf node that was split. The B^+ -tree of figure 11 shows the result of the insertion.

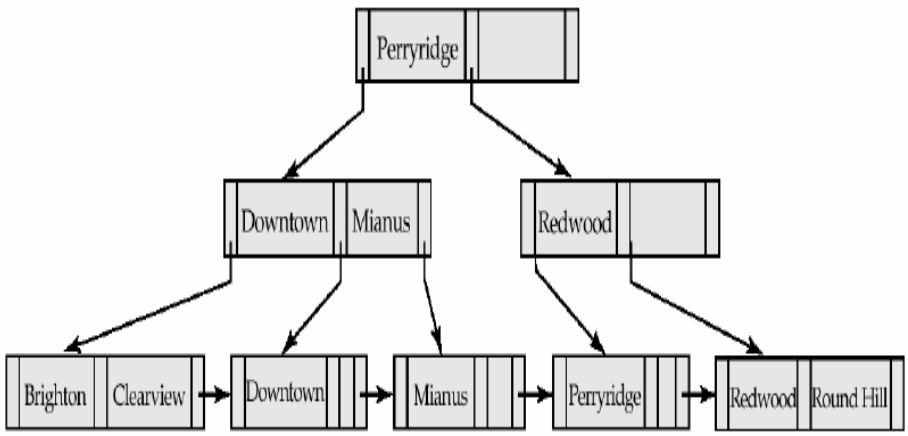


Figure 11 Insertion of “Clearview” into the B^+ -tree of figure 8.

The search-key value “Downtown” was inserted into the parent. It was possible to perform this insertion because there was room for and added search-key value .If there were no room, the parent would have had to be spilt. 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 node s to contain too few pointers, first, let us delete “Downtown” from the B^+ -tree of figure 11. We locate the entry for “Downtown” by using our lookup algorithm. When we delete the entry for “Down town” from its lea node, the leaf becomes empty. Since, in our example n = 3 and 0< [“(nm-1)/2], 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≥ [n/2], the node is still sufficiently large, and the deletion operation is complete. The resulting B^+ -tree appears in figure 12.

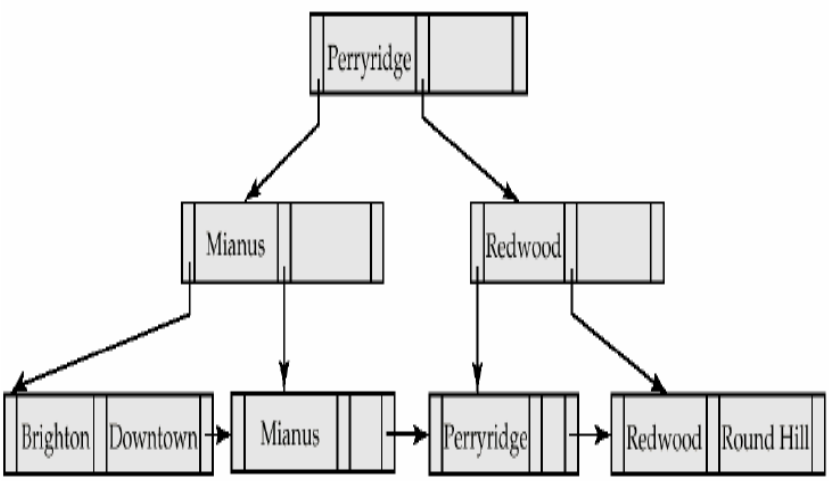


Figure 12 Deletion of “Downtown” from the B^+ -tree of fig-11

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 figure 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, [n/2] = 2, and thus only one pointer is too few. However, since the parent node contains useful information, we cannot simply delete if. 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); figure 13 shows the result.

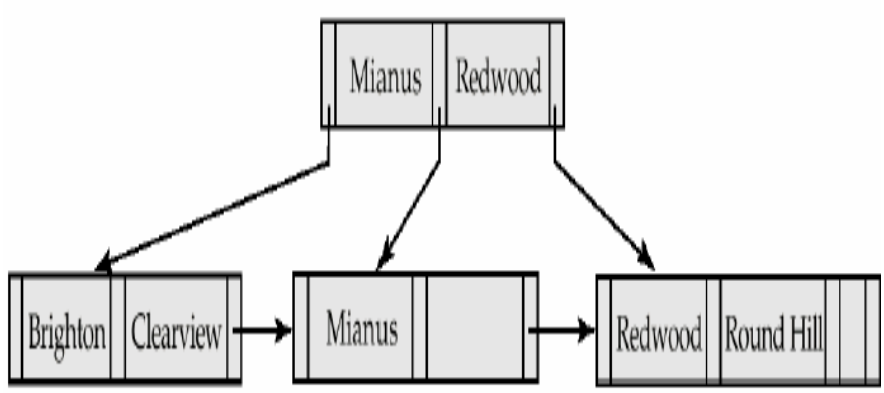


Figure 13 Deletion of “Perryridge” from the B^+ -tree of figure 12

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 by1.

It is not always possible to coalesce nodes. As an illustration, delete “Perryridge” from the B^+ -tree of figure 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 if 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 figure 14.

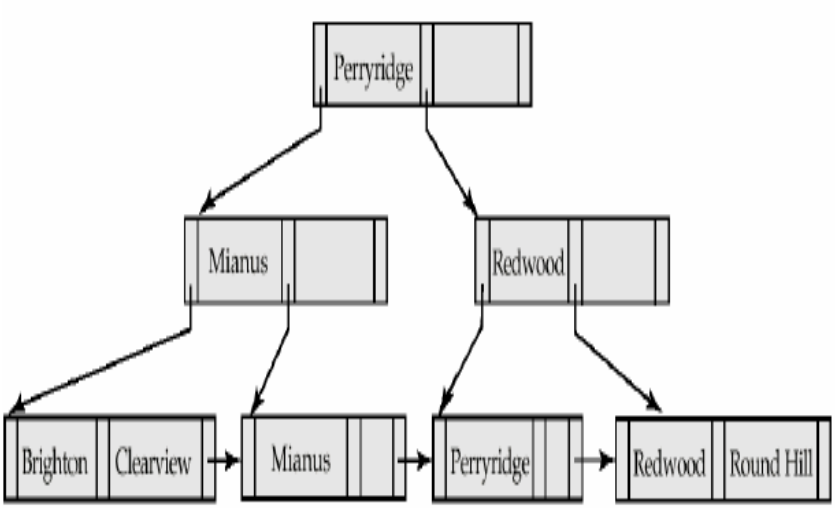


Figure 14 Deletion of "Perryridge" from the B^+ -tree of figure 11

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^+ -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 of 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^+ -tree, and is therefore low. It is the speed of operation on B^+ -tree that makes them a frequently used index structure in database implementations.

B^+ -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^+ -tree indices on the file. We solve the degradation problem for storing the actual records by using the leaf level of the B^+ -tree to organize the blocks containing the actual records. We use the B^+ -tree structure not only as an index, but also as an organizer for records in a file. In a B^+ -tree file organization, the leaf nodes of the tree store records, instead of

storing pointers to records. Figure 15 shows an example of a B^+ -tree file organization.

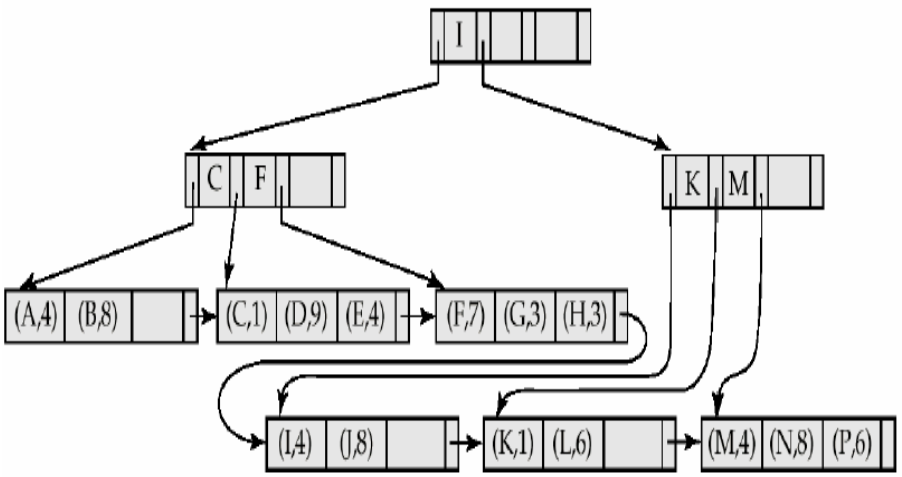


Figure 15 B^+ -tree file organization

Since records are usually larger than pointers, the maximum number of records that can be stored in a leaf nod e is less than the number of pointers in a nonleaf 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 locaters the block that should contain the record by searching the B^+ -tree insertion, the system splits the block in two, and redistributed 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 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 utili9zatrioni of space in a B^+ -

tree by invoking more sibling nodes in redistribution during splits merges. The technique is applicable to both le4af 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 [2n/3] entries, where n is the maximum number of entries that the node can hold. ([x] 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 [2n/3], the system attempts to borrow an entry form one of the sibling nodes. If both sibling nodes have [2n/3] 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 becauswe3 the total number of entries is 3[2n/3]-1, which is less than 2n. With three adjacent nodes used for redistribution, each node can be guaranteed to have [2n/4] entries. In general, if m nodes (m-1siblings) are involved in redistribution, each node can be guaranteed to contain at least [(m-1) n/m] entries. However, the cost of update becomes higher as more sibling nodes are involved in the redistribution.

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 figure given, 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. Figure 16 shows a B -tree that represents the same keys as the B^+ -tree.

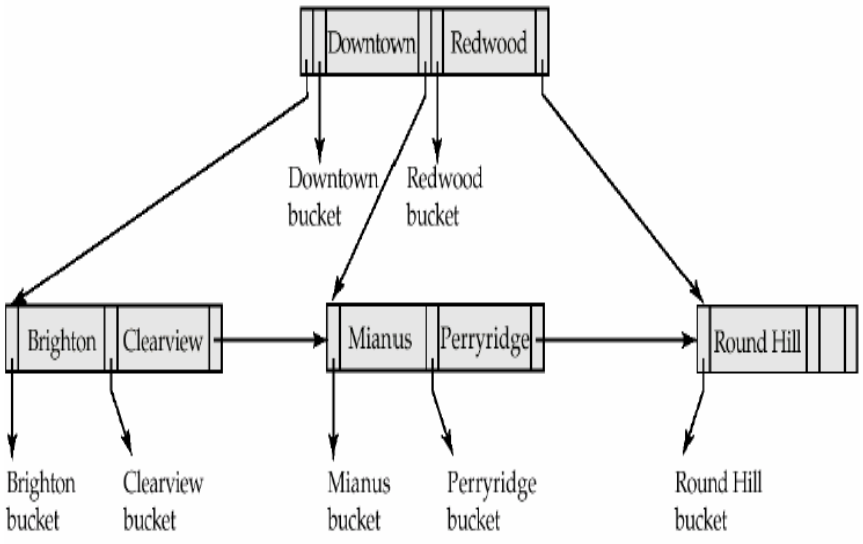


Figure 16 B -tree equivalent of B^+ -tree in figure 10.12

Since search keys are not repeated in the B^+ -tree, we may be able to store the index in fewer tree nodes than in the corresponding B^+ -tree index.

However, since search keys that appear in non-leaf nodes appear nowhere else in the B^+ -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 figure 17a; a non-leaf node appears in figure 17b; Leaf nodes are the same as in B^+ -trees.

(a)

(b)

Figure 17 Typical nodes of a B^+ -tree. (a) Leaf node. (b) Non-leaf node

In non-leaf nodes, the pointers P_i are the tree pointers that we used also for B^+ -tree, while the pointers B_i are bucket or file–record pointers.

In the generalized B -tree in the figure, there 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.

Multiple Key Access

We can use multiple indices for certain types of queries.

Example: **select** *account_number* **from** *account*
where *branch_name* = "Perryridge" **and**
balance = 1000;

The Possible strategies for processing query using indices on single attributes:

- Use index on *branch_name* to find accounts with branch name Perryridge; test *balance = 1000*
- Use index on *balance* to find accounts with balances of $1000; test *branch_name* = "Perryridge".
- Use *branch_name* index to find pointers to all records pertaining to the Perryridge branch. Similarly use index on *balance*. Take intersection of both sets of pointers obtained.

Composite search keys are search keys containing more than one attribute.

E.g. (*branch_name, balance*)

Lexicographic ordering: $(a_1, a_2) < (b_1, b_2)$ if either
$a_1 < b_1$, or
$a_1=b_1$ and $a_2 < b_2$

Suppose we have an index on combined search-key
(*branch_name, balance*).

With the **where** clause

where *branch_name* = “Perryridge” **and** *balance* = 1000

the index on (*branch_name, balance*) can be used to fetch only records that satisfy both conditions.

Using separate indices in less efficient — we may fetch many records (or pointers) that satisfy only one of the conditions.

Can also efficiently handle

where *branch_name* = “Perryridge” **and** *balance* < 1000

But cannot efficiently handle

where *branch_name* < “Perryridge” **and** *balance* = 1000

It may fetch many records that satisfy the first but not the second condition.

Static Hashing

A **bucket** is a unit of storage containing one or more records (a bucket is typically a disk block).

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

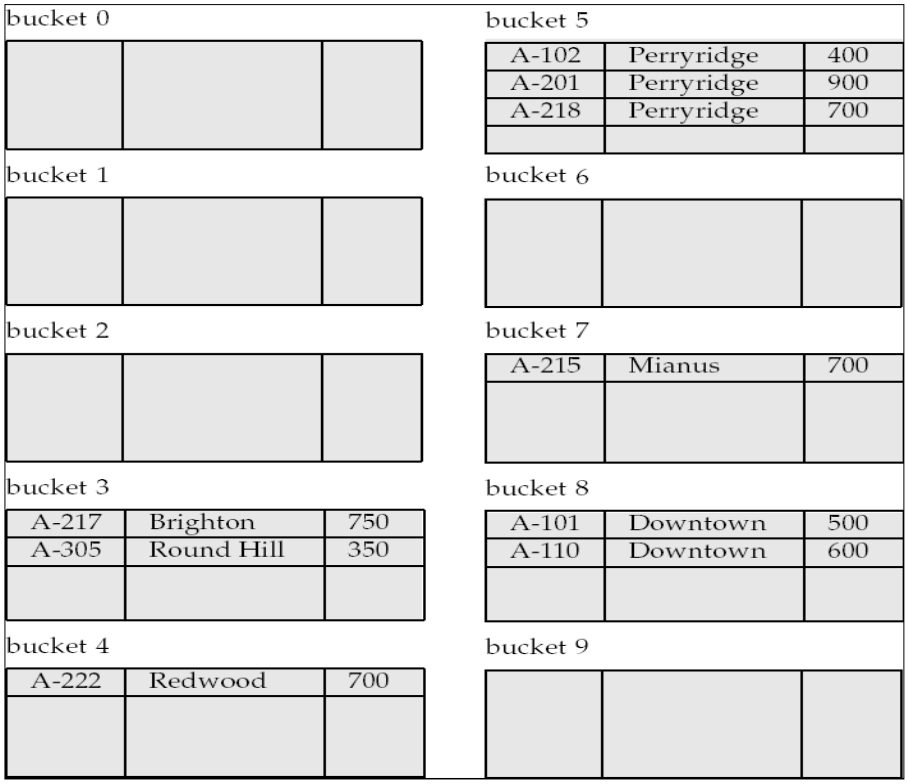


Figure 18. Hash file organization (buckets)

Hash file organization of *account* file, using *branch_name* as key

- There are 10 buckets,
- The binary representation of the ith character is assumed to be the integer i The hash function returns the sum of the binary representations of the characters modulo 10

E.g. h (Perryridge) = 5 h (Round Hill) = 3 h(Brighton) = 3

- Worst hash function maps all search-key values to the same bucket; this makes access time proportional to the number of search-key values in the file.

- An ideal hash function is **uniform**, i.e., each bucket is assigned the same number of search-key values from the set of *all* possible values.

- Ideal hash function is **random**, so each bucket will have the same number of records assigned to it irrespective of the *actual distribution* of search-key values in the file.

- Typical hash functions perform computation on the internal binary representation of the search-key.

- For example, for a string search-key, the binary representations of all the characters in the string could be added and the sum modulo the number of buckets could be returned. .

Handling of Bucket Overflows:

Bucket overflow can occur because of Insufficient buckets.It maintains skew in distribution of records. This can occur due to two reasons:

- multiple records have same search-key value
- chosen hash function produces non-uniform distribution of key values

Although the probability of bucket overflow can be reduced, it cannot be eliminated; it is handled by using *overflow buckets*

Overflow chaining – the overflow buckets of a given bucket are chained together in a linked list. It is also scheme is called closed hashing**.**

An alternative, called open hashing, which does not use overflow buckets, is not suitable for database applications.

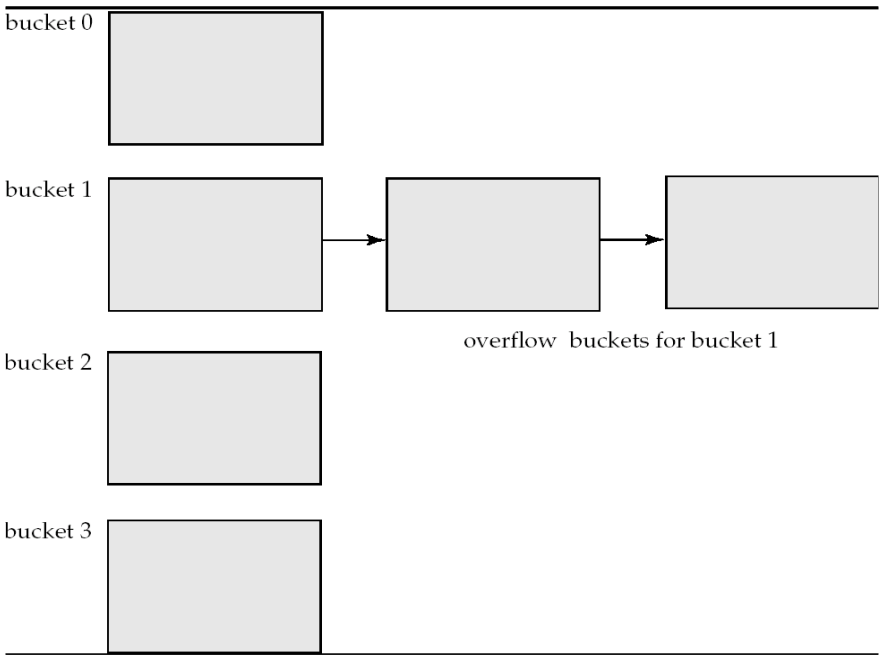


Figure 19. overflow buckets

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 record pointers, into a hash file structure. Strictly speaking, hash indices are always secondary indices if the file itself is organized using hashing; a separate primary hash index on it using the same search-key is unnecessary.

However, we use the term hash index to refer to both secondary index structures and hash-organized files.

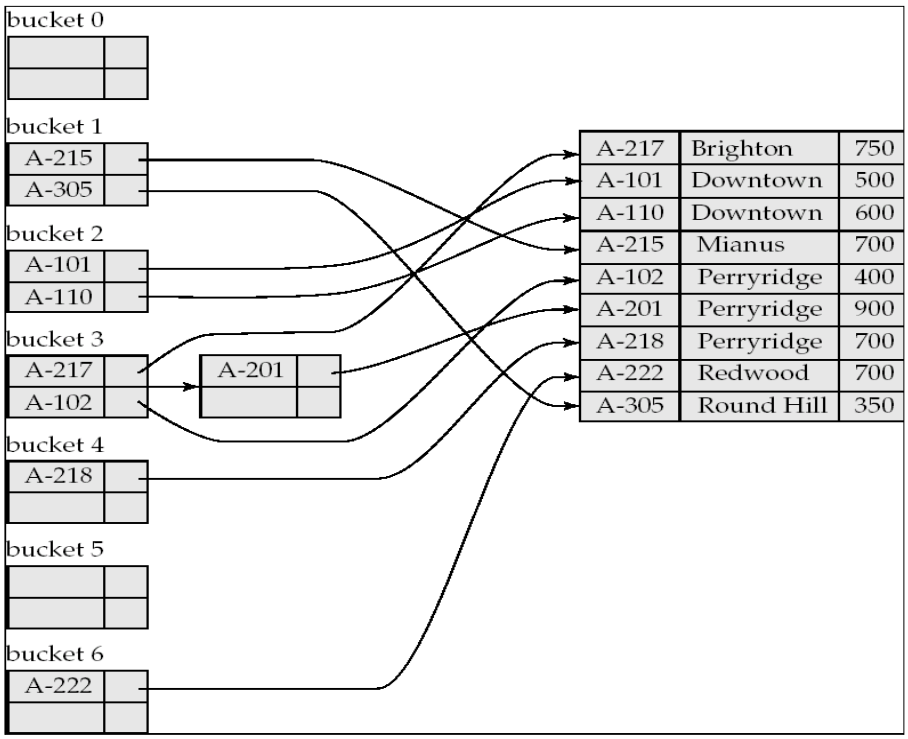


Figure 20. Example of hash index

Deficiencies in Static Hashing:

- In static hashing, function *h* maps search-key values to a fixed set of *B* of bucket addresses.
- Databases grow or shrink with time.
- If initial number of buckets is too small, and file grows, performance will degrade due to too much overflows.
- If space is allocated for anticipated growth, a significant amount of space will be wasted initially (and buckets will be under full).
- If database shrinks, again space will be wasted.

One solution for this is periodic re-organization of the file with a new hash function. The other better solution is to allow the number of buckets to be modified dynamically.

Dynamic Hashing

Dynamic Hashing is good for database that grows and shrinks in size. It allows the hash function to be modified dynamically. The **Extendable hashing** is one form of dynamic hashing

Hash function generates values over a large range — typically *b*-bit integers, with *b* = 32.

At any time use only a prefix of the hash function to index into a table of bucket addresses.

Let the length of the prefix be *i* bits, 0 £ *i* £ 32.

Bucket address table size = 2^i. Initially *i* = 0

Value of *i* grows and shrinks as the size of the database grows and shrinks.

Multiple entries in the bucket address table may point to a bucket (why?)

Thus, actual number of buckets is < 2^i The number of buckets also changes dynamically due to coalescing and splitting of buckets.

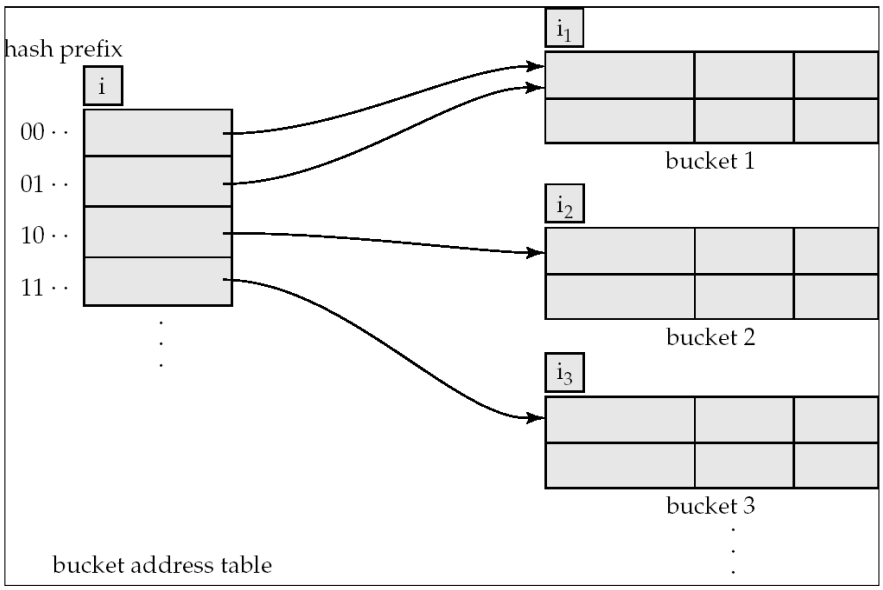


Figure 21. General Structure For Extendible Hashing

Use of Extendible Hashing:

Each bucket *j* stores a value i_j

*A*ll the entries that point to the same bucket have the same values on the first i_j bits.

To locate the bucket containing search-key K_j:

1. Compute $h(K_j) = X$.
2. Use the first *i* high order bits of *X* as a displacement into bucket address table, and follow the pointer to appropriate bucket.

To insert a record with search-key value K_j

- follow same procedure as look-up and locate the bucket, say *j*.
- If there is room in the bucket *j* insert record in the bucket.
- Else the bucket must be split and insertion re-attempted.
- Overflow buckets used instead in some cases

Disadvantages of extendable hashing:

- Extra level of indirection to find desired record
- Bucket address table may itself become very big (larger than memory)
- It cannot allocate very large contiguous areas on disk either.
- Changing size of bucket address table is an expensive operation.

One solution to this is use B^+-tree structure to locate desired record in bucket address table

Comparison of Ordered Indexing & hashing

- Cost of periodic re-organization
- Relative frequency of insertions and deletions
- Is it desirable to optimize average access time at the expense of worst-case access time?

- Expected type of queries:
    - Hashing is generally better at retrieving records having a specified value of the key.
    - If range queries are common, ordered indices are to be preferred

In fact, we can watch the following in practice:

- PostgreSQL supports hash indices, but discourages use due to poor performance.
- Oracle supports static hash organization, but not hash indices.
- SQLServer supports only B^+-trees.

Index Definition SQL

Syntax:

Create an index
create index <index-name> **on** <relation-name> (<attribute-list>);

E.g.: **create index** *b-index* **on** *branch(branch_name)*

Use **create unique index** to indirectly specify and enforce the condition that the search key is a candidate key is a candidate key.

Not really required if SQL **unique** integrity constraint is supported

To drop an index **drop index** <index-name>

Most database systems allow specification of type of index, and clustering.

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^+ -tree index.

We can use B^+ -trees for indexing a file containing records, as well as to organize records into a file.

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 indexes 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.

Model Questions

1. Differentiate Dense Index and Sparse Index in detail

2. What is the difference between a primary index and a Secondary index?

3. Construct a B^+ -tree for the following set of key values:

   (12,13,56,77,1,17,9,94,28,10)

4. Discuss B – trees and B+ tree index files with examples.

UNIT – III

1. RELATIONAL DATABASE DESIGN

Objective

- 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

Structure of the Lesson

Introduction

In general, the main goal of relational database design is to generate a set of relation schemas that allows us to retrieve information easily. This is accomplished by designing schemas that are in appropriate normal forms. To determine whether a relational schema is in one of the desirable normal forms, we need information about the real world enterprise that we are modeling with the database.

Here we introduce a formal approach to relational database design based on the notation of functional dependencies. We then define normal forms in terms of functional dependencies and other types of data dependencies.

In some other way, whenever we design databases we are faced with a number of problems relating to things like data integrity, security, and 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.

Features of Good Relational Designs

Databases have a reputation for being difficult to construct and hard to maintain. In fact, Database design has nothing to do with using computers. It has everything to do with research and

planning. The design process should be completely independent of software choices. The basic elements of the design process are:

- Defining the problem or objective
- Researching the current database
- Designing the data structures
- Constructing database relationships
- Implementing rules and constraints
- Creating database views and reports
- Implementing the design

Among the above, most important step in database design is the first one: defining the problem the database will address or the objective of the database. It is important however, to draw a distinction between:

- How the database will be used and
- What information needs to be stored in it

A database is essentially a collection of data tables, so the next step in the design process is to identify and describe those data structures. Each table in a database should represent some distinct subject or physical object.

Once the data structures are in place, the next step is to establish the relationships between the databases. First you must ensure that each table has a unique key that can identify the individual records in each table. Any field in the database that contains unique values is an acceptable field to use as a key. Usually we can generate relational schemas directly from the E-R diagrams.

In order to achieve the relational database design, one can implement a new concept called normalization and it is an essential part of database design.

Normalization is a process of decomposing a relation in to several sub relations so that they can be managed properly.

Normalization is the process of efficiently organizing data in a database. There are two goals of the normalization process: eliminating redundant data (for example, storing the same data in more than one table) and ensuring data dependencies make sense (only storing related data in a table). Both of these are worthy goals

as they reduce the amount of space a database consumes and ensure that data is logically stored.

Normalization draws heavily on the theory of functional dependencies.

Normalization theory defines six normal forms (NFs). Each normal form involves a set of dependency properties that a schema must satisfy and each gives guarantees about presence/absence of update anomalies. That means higher normal forms have less redundancy so that less update problems.

- A brief history of normal forms:
- First, Second, Third Normal Forms (1NF,2NF, 3NF)
- Boyce-Codd Normal Form (BCNF)
- Fourth Normal Form (4NF)
- Fifth Normal Form (5NF)

NF hierarachy: 5NF ->4NF ->BCNF ->3NF ->2NF ->1NF

1NF allows most redundancy; 5NF allows least redundancy.

1NF	All attributes have atomic values we assume this as part of relational model
2NF	All non-key attributes fully depend on key (i.e. no partial dependencies) avoids much redundancy
3NF, BCNF	No attributes dependent on non-key attributes (i.e. no transitive dependencies) avoids remaining redundancy
4NF	Removes problems due to multi-valued dependencies
5NF	Removes problems due to join dependencies

In practice, BCNF and 3NF are the most important.

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.

11.1.2 First Normal Forms

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 I 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.

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)

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, may 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*

Branch-name-→ assets

Holds on *Lending-schema*, but we do not except the functional dependency *branch-name→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.

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 of *key*.

branch-name	*branch-city*	*assets*	*customer-name*	*loan-number*	*amount*
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

Basic relation in use

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 t1 and t2 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 → R. That is, K is a super key if, whenever $t_1[k]=t_2[k]$, it id 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→amount
loan-number→branch0-name

We would not, however, expect the functional dependency

loan-number→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 —>; 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.

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

Closure of Attribute Sets

To test whether a set of attributes α is a superkey, we need to find the set of attributes functionally determined by α.

1. Let α be a set of attributes. We call the set of attributes determined by α under a set $F$of functional dependencies the **closure** of α under F, denoted α^+.
2. The following algorithm computes α^+:

```
result := α
while (changes to result) do
  for each functional dependency β → γ
        in F do
    begin
      if β ⊆ result
      then result := result ∪γ;
    end
```

3. If we use this algorithm on our example to calculate (AG^+) then we find:
   - We start with *result* = AG.
   - $A \rightarrow B$ causes us to include B in *result*.
   - $A \rightarrow C$ causes *result* to become ABCG.

- *CG* $\rightarrow$*H* causes *result* to become ABCGH.
- *CG* $\rightarrow$*I* causes *result* to become ABCGHI.
- The next time we execute the while loop, no new attributes are added, and the algorithm terminates.

Canonical Forms

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_c.

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

F_c must also have the following properties:

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

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

Logically implies F_c.

Every functional dependency $\alpha \rightarrow \beta$ in F_c contains no **extraneous** attributes in β(ones that can be removed from βwithout changing F_c^+). So A is extraneous in β if $A \in \beta$ and

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

Logically implies F_c.

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

To compute a canonical cover F_c for F,

- Use the union rule to replace dependencies of the form $\alpha_1 \rightarrow \beta_1$ and $\alpha_1 \rightarrow \beta_2$ with $\alpha_1 \rightarrow \beta_1\beta_2$.
- Test each functional dependency $\alpha \rightarrow \beta$ to see if there is an extraneous attribute in α.
- Test each functional dependency $\alpha \rightarrow \beta$ to see if there is an extraneous attribute in β.

- 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_c.

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 $A \rightarrow BC$.

- So we end up with

$$A \rightarrow B$$
$$B \rightarrow C$$

Desirable Properties of Decomposition

Consider a schema

Lending-schema = (bname, assets, bcity, loan#, cname, amount)

which we saw was a bad design.

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

bname → assets bcity
loan# → amount bname

If we decompose it into

Branch-schema = (bname, assets, bcity)

Loan-info-schema = (bname, loan#, amount)
Borrow-schema = (cname, loan#)

we claim this decomposition has several desirable properties.

Lossless Decomposition

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

- Let R be a relation scheme.
- Let F be a set of functional dependencies on R.
- Let R_1 and R_2 form a decomposition of R.
- The decomposition is a lossless-join decomposition of R if at least one of the following functional dependencies are in 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

  $bname \rightarrow bname\ assets\ bcity$

- Since *Branch-scheme* ∩*Borrow-scheme* = *bname*, our decomposition is lossless join.
- Next we decompose *Borrow-scheme* into

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

- As *loan#* is the common attribute, and

  $$loan\# \rightarrow amount\ bname$$

  This is also a loss less-join decomposition.

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 $\{R_1, R_2, \ldots, R_n\}$ 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 set of restrictions $F_1, F_2, \ldots, F_n$ is the set of dependencies that can be checked efficiently.

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

Let $F' = F_1, F_2, \ldots, F_n$.

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

$F' = \emptyset$

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

bname → *assets bcity*

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

The functional dependency

loan# → *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?

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.

Second Normal Form

A relation schema R is in 2NF if every non-prime attribute is fully functionally dependent on any key of R

Prime attribute: An attribute that is part of some key non-prime attribute: An attribute that is not part of any key.

Third normal Form

A relation R is in third normal form (3NF) with respect to a set F of functional dependencies id, for all functional dependencies in F^+ of the form $\alpha \rightarrow \beta$, where $\alpha \subseteq$ R <R and $\beta \subseteq$ R, at least one of the following holds:

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

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 $(R_1, R_2, ..., R_i)$

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 no 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.

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→b and b→c and consequently a→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.

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?

2. ADVANCED NORMAL FORMS

Objective

- 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

Advanced Normal Forms

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.

Comparison between 3NF and BCNF

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-→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*.

customer-name	*banker-name*	*branch-name*
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(α).** 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.

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→ 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→ 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.

Multi-valued Dependency

Functional dependencies rule out certain tuples from being in a relation. If A→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* $\alpha \rightarrow\rightarrow \beta$.

α 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:

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

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

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

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

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

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

Tabular representation of $\alpha \rightarrow\rightarrow \beta$

Basic table

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 $\alpha \rightarrow\rightarrow \beta$ says that the relationship between α and β is independent of the relationship between α and R-β. If the multivalued dependency $\alpha \rightarrow\rightarrow \beta$ is a *trivial* multivalued dependency on schema R then, $\alpha \rightarrow\rightarrow \beta$ 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 -→→ customer -street customer -city

To hold (The multivalued dependency customer-name→→ 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.

loan-number	*customer-name*	*customer-street*	*customer-city*
L-23	Smith	North	Rye
L-23	Smith	Main	Manchester
L-93	Curry	Lake	Horseneck

Relation bc: Am example of redundancy in a BCNF relation.

loan-number	*customer-name*	*customer-street*	*customer-city*
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^{l} 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 α --→ β then, $\alpha \rightarrow\rightarrow \beta$.
In other words, every functional dependency is also a multivalued dependency.

Definitions of Fourth Normal Form

Consider again our BC-schema example in which the multi-valued dependency customer-name→→ customer-street customer-city hold, but no nontrivial functional dependencies hold, although BC- schema is in BCNF, the deign is not ideal, since we must repeat a customers 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 $\alpha \rightarrow\rightarrow \beta$ where $\alpha \subseteq$ R and $\beta \subseteq$ R , at least one of the following holds

$\alpha \rightarrow\rightarrow \beta$ is a trivial dependency.
α 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 4 NF.

Note that the definition of 4NF differs from the definition of BCNF in only the use of multi-valued 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 $\alpha \rightarrow \beta$ be a nontrivial functional dependency that holds on R_i such that $\alpha \rightarrow R_i$ is not in F^+, and $\alpha \cap \beta = \varnothing$;

result := (*result* – R_i) $\cup$ (R_i – β) $\cup$ (α, β);

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 α is not a super key. Since $\alpha \rightarrow \beta$ implies $\alpha \rightarrow\rightarrow \beta$, R cannot be in 4NF.

Let R be a relation schema, and let $R_1, R_2, R_3, \ldots\ldots\ldots\ldots, R_n$ 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

$$\alpha \rightarrow\rightarrow \beta \cap R_i$$

where $\alpha \subseteq R_i$ and $\alpha \rightarrow\rightarrow \beta$ is in D^+.

More Normal Forms

The fourth normal form is by no means the "ultimate" formal 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 r reasoning about the constraints. Hence PJNF and domain-key normal form are used quite rarely. Appendix C provides more details about these normal forms.

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.

branch-name	*loan-number*
Round Hill	L-58

loan-number	*amount*

loan-number	*customer-name*
L-58	Johnson

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.

In brief, observe the following:

3rd Normal Form (3NF)

A table is in 3NF if it is in 2NF and if it has no transitive dependencies.

Boyce-Codd Normal Form (BCNF)

A table is in BCNF if it is in 3NF and if every determinant is a candidate key.

4th Normal Form (4NF)

A table is in 4NF if it is in BCNF and if it has no multi-valued dependencies.

5th Normal Form (5NF)

A table is in 5NF, also called “Projection-join Normal Form” (PJNF), if it is in 4NF and if every join dependency in the table is a consequence of the candidate keys of the table.

Domain-Key Normal Form (DKNF)

A table is in DKNF if every constraint on the table is a logical consequence of the definition of keys and domains.

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-->> 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.

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?

3. DATABASE SYSTEM ARCHITECTURES AND THE SYSTEM CATALOG

Objectives

The major objective of this lesson is to provide basics of database concepts and technology.

After reading this chapter, you understand:

- The elementary concepts data of System Architectures for DBMSs
- Discuss about Catalogs for Relational DBMSs.
- Discuss about System Catalog Information in ORACLE
- Discuss about Catalog Information Accessed by DBMS Software Modules
- Distinguish between Data Dictionary and Data Repository Systems

Structure

System Architectures for DBMSs

Catalogs for Relational DBMSs

System Catalog Information in ORACLE

Other Catalog Information Accessed by DBMS Software Modules

Data Dictionary and Data Repository Systems

System Architectures for DBMSs

Architectures for DBMSs followed trends similar to those of general computer systems architectures. Earlier architectures used mainframe computers to provide the main processing for all functions of the system, including user application programs, user interface programs, as well as all the DBMS functionality. The reason was that most users accessed such systems via computer terminals that did not have processing power and only provided display capabilities. So, all processing was performed remotely, and only display information and controls were sent from the

computer to the display terminals, which were connected to the central computer via various types of communications networks.

As the prices of hardware declined, most users replaced their terminals with personal computers (PCs) and workstations. At first, database systems used these computers in the same way as they had used display terminals, so that the DBMS itself was still a centralized DBMS where all the DBMS functionality, application program execution, and user interface processing were carried out in one machine. Gradually DBMS systems started to exploit the available processing power at the user side, which led to client-server DBMS architectures.

Client-Server Architecture

We first discuss client-server architecture in general, then see how it is applied to DBMSs. The client-server architecture was developed to deal with computing environments where a large number of PCs, workstations, file servers, printers, database servers, Web servers, and other equipment are connected together via a network. The idea is to define specialized servers with specific functionalities. For example, it is possible to connect a number of PCs or small workstations as clients to a file server that maintains the files of the client machines. Another machine could be designated as a printer server by being connected to various printers; thereafter, all print requests by the clients are forwarded to this machine. Web servers or E-mail servers also fall into the specialized server category. In this way, the resources provided by specialized servers can be accessed by many client machines. The client machines provide the user with the appropriate interfaces to utilize these servers as well as with local processing power to run local applications. This concept can be carried over to software, with specialized software such as a DBMS or a CAD (computer-aided design) package being stored on specific server machines and being made accessible to multiple clients. Some machines would be only client sites (for example, diskless workstations or workstations/PCs with disks that have only client software installed). Other machines would be dedicated servers. Still other machines would have both client and server functionality.

The concept of client-server architecture assumes an underlying framework that consists of many PCs and workstations as well as a smaller number of mainframe machines, connected via local area networks and other types of computer networks. A client in this framework is typically a user machine that provides user interface capabilities and local processing. When a client requires access to additional functionality such as database access that does not exist at that machine, it connects to a server that provides the needed functionality. A server is a machine that can provide services to the

client machines, such as printing, archiving, or database access. Two main types of basic DBMS architectures were created on this underlying client-server framework (Note 1). We discuss those next.

Client-Server Architectures for DBMSs

The client-server architecture is increasingly being incorporated into commercial DBMS packages. In relational DBMSs, many of which started as centralized systems, the system components that were first moved to the client side were the user interface and application programs. Because SQL provided a standard language for RDBMSs, it created a logical dividing point between client and server. Hence, the query and transaction functionality remained at the server side. In such an architecture, the server is often called a query server or transaction server, because it provided these two functionalities. In RDBMSs, the server is also often called an SQL server, since most RDBMS servers are based on the SQL language and standard.

In such client-server architecture, the user interface programs and application programs can run at the client side. When DBMS access is required, the program establishes a connection to the DBMS which is on the server side and once the connection is created, the client program can communicate with the DBMS. A standard called Open Database Connectivity (ODBC) provides an Application Programming Interface (API), which allows client-side programs to call the DBMS, as long as both client and server machines have the necessary software installed. Most DBMS vendors provide ODBC drivers for their systems. Hence, a client program can actually connect to several RDBMSs and send query and transaction requests using the ODBC API, which are processed at the server sites. Any query results are sent back to the client program, which can process or display the results as needed. Another related standard for the JAVA programming language, called JDBC, has also been defined. This allows JAVA client programs to access the DBMS through a standard interface.

The second approach to client-server was taken by some object-oriented DBMSs. Because many of those systems were developed in the era of client-server architecture, the approach taken was to divide the software modules of the DBMS between client and server in a more integrated way. For example, the server level may include the part of the DBMS software responsible for handling data storage on disk pages, local concurrency control and recovery, buffering and caching of disk pages, and other such functions. Meanwhile, the client level may handle the user interface, data dictionary functions, DBMS interaction with programming language compilers, global query optimization/concurrency control/recovery, structuring of complex objects from the data in the buffers, and other such

functions. In this approach, the client-server interaction is more tightly coupled and is done internally by the DBMS modules some of which reside in the client rather than by the users. The exact division of functionality varies from system to system. In such client-server architecture, the server has been called a data server, because it provides data in disk pages to the client, which can then be structured into objects for the client programs by the client-side DBMS software itself.

Catalogs for Relational DBMSs

We now turn our attention to the second topic of this chapter, which is the DBMS catalog, and discuss catalogs for relational DBMSs (Note 2). The information stored in a catalog of an RDBMS includes the relation names, attribute names, and attribute domains (data types), as well as descriptions of constraints (primary keys, secondary keys, foreign keys, NULL/NOT NULL, and other types of constraints), views, and storage structures and indexes. Security and authorization information is also kept in the catalog; this describes each user's privileges to access specific database relations and views, and the creator or owner of each relation.

In relational DBMSs it is common practice to store the catalog itself as relations and to use the DBMS software for querying, updating, and maintaining the catalog. This allows DBMS routines (as well as users) to access the information stored in the catalog—whenever they are authorized to do so using the query language of the DBMS, such as SQL.

The primary key of REL_AND_ATTR_CATALOG is the combination of the attributes {REL_NAME, ATTR_NAME}, because all relation names should be unique and all attribute names within a particular relation should also be unique (Note 3). Another catalog relation can store information such as tuple size, current number of tuples, number of indexes, and creator name for each relation.

To include information on secondary key attributes of a relation, we can simply extend the preceding catalog if we assume that an attribute can be a member of one key only. In this case we can replace the MEMBER_OF_PK attribute of REL_AND_ATTR_CATALOG with an attribute KEY_NUMBER; the value of KEY_NUMBER is 0 if the attribute is not a member of any key, 1 if it is a member of the primary key, and $i > 1$ for the secondary key, where the secondary keys of a relation are numbered 2, 3, ..., n. However, if an attribute can be a member of more than one key, which is the general case, the above representation is not sufficient. One possibility is to store information on key attributes separately in a second catalog relation RELATION_KEYS, with attributes {REL_NAME, KEY_NUMBER, MEMBER_ATTR}, which also together form

the key of RELATION_KEYS. This is shown in Figure 1. 3(a). The DDL compiler assigns the value 1 to KEY_NUMBER for the primary key and values 2, 3, ..., n for the secondary keys, if any. Each key will have a tuple in RELATION_KEYS for each attribute that is part of that key, and the value of MEMBER_ATTRIBUTE gives the name of that attribute. A similar structure can be used to store information involving foreign keys. If constraints are given names so they can be dropped later, then a unique attribute CONSTRAINT_NAME must be added to the catalog tables that describes constraints (including those that describe keys).

Next, let us consider information regarding indexes. In the general case where an attribute can be a member of more than one index, the RELATION_INDEXES catalog relation can be used. The key of RELATION_INDEXES is the combination {INDEX_NAME, MEMBER_ATTR} (assuming that index names are unique). MEMBER_ATTR is the name of an attribute included in the index.

The definitions of views must also be stored in the catalog. A view is specified by a query, with a possible renaming of the values appearing in the query result. We can use the two catalog relations to store view definitions. The first, VIEW_QUERIES, has two attributes {VIEW_NAME, QUERY} and stores the query (as a text string) corresponding to the view. The second, VIEW_ATTRIBUTES, has attributes {VIEW_NAME, ATTR_NAME, ATTR_NUM} to store the names of the attributes of the view, where ATTR_NUM is an integer number greater than zero specifying the correspondence of each view attribute to the attributes in the query result. The key of VIEW_QUERIES is VIEW_NAME, and that of VIEW_ATTRIBUTES is the combination {VIEW_NAME, ATTR_NAME}.

The preceding examples illustrate the types of information stored in a catalog. In a real system, the catalog will typically include many more tables and information. Most relational systems store their catalog files as DBMS relations. However, because the catalog is accessed very frequently by the DBMS modules, it is important to implement catalog access as efficiently as possible. It may be more efficient to use a specialized set of data structures and access routines to implement the catalog, thus trading generality for efficiency. An additional problem is that of system initialization; the catalog tables must be created before the system can function!

In conclusion, we take a conceptual look at the basic information stored in the parts of a relational catalog for describing tables (relations). A high-level EER schema diagram describing the information about schemas, relations, attributes, keys, views, and indexes. The SCHEMA entity type in the figure represents the schemas that have been defined in a RDBMS. The entity

type RELATION is a weak entity type owned by (or identified by) SCHEMA—with partial key RelName—to represent the relations that appear in a particular schema. Two disjoint subclasses, BASE_RELATION and VIEW_RELATION, are created for RELATION. The entity type ATTRIBUTE is a weak entity type owned by BASE_RELATION, and its partial key is AttrName. BASE_RELATIONs also have general key and foreign key constraints, as well as indexes, whereas VIEW_RELATIONs have their defining query, as well as the AttrNum described earlier to specify correspondence of view attributes to query attributes. Notice that an additional unspecified constraint is that all attributes related to a KEY or INDEX entity—via the relationships KEY_ATTRS or INDEX_ATTRS—must be related to the same BASE_RELATION entity to which the KEY or INDEX entity is related. KeyType specifies whether the key is a foreign, primary, or secondary key. FKEY is a subclass for foreign keys and is related to the referenced relation via the REFREL relationship.

System Catalog Information in ORACLE

The various commercial database products adopt different conventions and terminology with regard to their system catalog. However, in general, the catalogs contain similar metadata describing conceptual, internal, and external schemas. In this section, we examine parts of the system catalog for the ORACLE RDBMS as an example of a catalog for a commercial system.

In ORACLE, the collection of metadata is called the data dictionary. The metadata is information about schema objects, such as tables, indexes, views, triggers, and more. Access to the data dictionary is allowed through numerous views, which are divided into three categories: USER, ALL, and DBA. These terms are used as prefixes for the various views. The views that have a prefix of USER contain schema information for objects owned by a given user. Those with a prefix of ALL contain schema information for objects owned by a user as well as objects that the user has been granted access to, and those with a prefix of DBA are for the database administrator and contain information about all database objects.

As already mentioned, the system catalog contains information about all three levels of database schemas: external (view definitions), conceptual (base tables), and internal (storage and index descriptions). To illustrate in ORACLE, we examine some of the catalog views relating to each of the three schema levels. The catalog (metadata) data can be retrieved through SQL statements as can the user (actual) data.

We start with the conceptual schema information. To find the objects owned by a particular user, ‘SMITH’, we can write the following query:

```
SELECT *
FROM ALL_CATALOG
WHERE OWNER = 'SMITH';
```

The result of this query, which indicates that three base tables are owned by SMITH: ACCOUNT, CUSTOMERS, and ORDERS, plus a view CUSTORDER. The meaning of each column in the result should be clear from its name.

To find some of the information describing the columns of the ORDERS table for 'SMITH', the following query could be submitted:

```
SELECT COLUMN_NAME, DATA_TYPE, DATA_LENGTH, NUM_DISTINCT,
LOW_VALUE, HIGH_VALUE
FROM USER_TAB_COLUMNS
WHERE TABLE_NAME = 'ORDERS';
```

The result of this query could be as shown in Figure 1.6. Because the USER_TAB_COLUMNS table of the catalog has the prefix USER_, this query must be submitted by the owner of the ORDERS table. The last three columns specified in the SELECT-clause of the SQL query play an important role in the query optimization process. The NUM_DISTINCT column specifies the number of distinct values for a given column and the LOW_VALUE and HIGH_VALUE specify the lowest and highest value, respectively, for the given column. We should note that these values, called database statistics, are not automatically updated when tuples are inserted/deleted/modified. Rather, the statistics are updated, either by exact computation or by estimation, whenever the ANALYZE SQL statement in ORACLE is executed as follows:

```
ANALYZE TABLE ORDERS
COMPUTE STATISTICS;
```

This SQL statement would update all statistics for the ORDERS relation and its associated indexes. To access information about the internal schema, the USER_TABLES and USER_INDEXES catalog tables can be queried. For example, to find storage information about the ORDERS table, the following query can be submitted:

```
SELECT PCT_FREE, INITIAL_EXTENT, NUM_ROWS, BLOCKS, EMPTY_BLOCKS,

AVG_ROW_LENGTH

  FROM  USER_TABLES

  WHERE TABLE_NAME = 'ORDERS';
```

The result of this query, which contains a subset of the available storage information in the catalog. The information includes from left to right the minimum percentage of free space in a block, the size of the initial storage extent in bytes, the number of rows in the table, the number of used data blocks allocated to the table, the number of free data blocks allocated to the table, and the average length of a row in the table in bytes.

The information from USER_TABLES also plays a useful role in query processing and optimization. The storage information about the indexes is just as important to the query optimizer as the storage information about the relations. For example, the number of index blocks that have to be accessed when searching for a specific key can be computed as the sum of BLEVEL and LEAF_BLOCKS_PER_KEY (Note 5). This information is used by the optimizer in deciding how to execute a query efficiently. For information about the external schema, the USER_VIEWS table can be queried as follows:

```
SELECT *

 FROM  USER_VIEWS;

 SELECT COLUMN_NAME, DATA_TYPE, DATA_LENGTH

 FROM  USER_TAB_COLUMNS

WHERE  TABLE_NAME = 'CUSTORDER';
```

More detailed information about ORACLE's data dictionary facilities can be found in the ORACLE RDBMS Database Administrator's Guide and the ORACLE SQL Language Reference Manual.

Other Catalog Information Accessed by DBMS Software Modules

The DBMS modules use and access a catalog very frequently; that is why it is important to implement access to the catalog as efficiently as possible. In this section we discuss the different ways in which some of the DBMS software modules use and access the catalog. These include the following:

1. DDL (and SDL) compilers: These DBMS modules process and check the specification of a database schema in the data definition language (DDL) and store that description in the catalog. Schema constructs and constraints at all levels conceptual, internal, and external are extracted from the DDL and SDL (storage definition language) specifications and entered into the catalog, as is any mapping information among levels, if necessary. Hence, these software modules actually populate (load) the catalog's minidatabase (or metadatabase) with data, the data being the descriptions of database schemas.

2. Query and DML parser and verifier: These modules parse queries, DML retrieval statements, and database update statements; they also check the catalog to verify whether all the schema names referenced in these statements are valid. For example, in a relational system, a query parser would check that all the relation names specified in the query exist in the catalog and that the attributes specified belong to the appropriate relations and have the appropriate type.

3. Query and DML compilers: These compilers convert high-level queries and DML commands into low-level file access commands. The mapping between the conceptual schema and the internal schema file structures is accessed from the catalog during this process. For example, the catalog must include a description of each file and its fields and the correspondences between fields and conceptual-level attributes.

4. Query and DML optimizer (Note 6): The query optimizer accesses the catalog for access path, implementation information, and data statistics to determine the best way to execute a query or DML command. For example, the optimizer accesses the catalog to check which fields of a relation have hash access or indexes, before deciding how to execute a selection or join condition on the relation.

5. Authorization and security checking: The DBA has privileged commands to update the authorization and security portion of the catalog. All access by a user to a relation is checked by the DBMS for proper authorization by accessing the catalog.

6. External-to-conceptual mapping of queries and DML commands: Queries and DML commands specified with reference to an external view or schema must be transformed to refer to the conceptual schema before they can be processed by the DBMS. This is accomplished by accessing the catalog description of the view in order to perform the transformation.

Data Dictionary and Data Repository Systems

The terms data dictionary and data repository are used to indicate a more general software utility than a catalog. A catalog is closely coupled with the DBMS software; it provides the information stored in it to users and the DBA, but it is mainly accessed by the various software modules of the DBMS itself, such as DDL and DML compilers, the query optimizer, the transaction processor, report generators, and the constraint enforcer. On the other hand, the software package for a stand-alone data dictionary or data repository may interact with the software modules of the DBMS, but it is mainly used by the designers, users, and administrators of a computer system for information resource management. These systems are used to maintain information on system hardware and software configurations, documentation, applications, and users, as well as other information relevant to system administration.

If a data dictionary system is used only by designers, users, and administrators, not by the DBMS software, it is called a passive data dictionary; otherwise, it is called an active data dictionary or data directory. The types of active data dictionary interfaces. Data dictionaries are also used to document the database design process itself, by storing documentation on the results of every design phase and the design decisions. This helps in automating the design process by making the design decisions and changes available to all the database designers. Modifications to the database description are made by changing the data dictionary contents. Using the data dictionary during database design means that, at the conclusion of the design phase, the metadata is already in the data dictionary.

Summary

In this lesson we first gave an overview of the centralized versus client-server system architectures, and described how these architectures are used in the database context. We discussed how earlier database systems were centralized, and how the emergence of the environment of networked workstations, PCs, and mainframes led to client-server computing. We showed how relational systems evolved into SQL servers (also called query servers or transaction servers), and discussed how the newer object databases further divide basic functionality between client and server, leading to data servers.

We then discussed the type of information that is included in a DBMS catalog. We discussed catalog structure for a relational DBMS and showed how it can store the constructs of the relational model, including information

concerning key constraints, indexes, and views. We also gave a conceptual description in the form of an EER schema diagram of the relational model constructs and how they are related to one another. We covered some specifics about the system catalog in the ORACLE RDBMS. We then discussed how different DBMS modules access the information stored in a DBMS catalog, and gave an overview of other types of information stored in a catalog. Finally, we briefly discussed data dictionary/repository systems and how they differ from catalogs.

Review Questions

1. What is the difference between centralized and client-server architectures in general?
2. How did relational DBMSs evolve from the centralized architecture to the client-server architecture? What is ODBC used for in this context?
3. How do object databases differ from relational systems in a client-server system architecture?
4. What is meant by the term metadata?
5. How are relational DBMS catalogs usually implemented?
6. Discuss the types of information included in a relational catalog at the conceptual, internal, and external levels.
7. Discuss how some of the different DBMS modules access a catalog and the type of information each accesses.
8. Why is it important to have efficient access to a DBMS catalog?
9. What are the three different view categories for catalog information in ORACLE and why are they important?

UNIT – IV

1. INTRODUCTION TO TRANSACTION PROCESSING CONCEPTS & THEORY

Structure of Lesson

Transaction concepts

Transaction state

Implementation of Atomicity and Durability

Concurrent Executions

Serializability

Recoverability

Implementation of Isolation

Transaction Definition in SQL

Testing for Serializability

Summary

Technical terms

Model questions

Introduction to Transactions & their properties

The term ***transaction*** refers to a collection of operations that form a single logical unit of work. The concept of transaction provides a mechanism for describing logical units of database processing. Transaction processing systems are the systems with large databases and hundreds of concurrent users executing database transactions. Online Reservation System, Banking

System, Credit Card Processing, Stock markets are some of the examples of such transaction processing systems.

Requirements of a transaction processing system are, ***high availability & fast response time***.

*A **transaction** is an executing program that performs a logical unit of database processing*. A transaction includes one or more database access operations. The operations include insertion, deletion, modification or retrieval operations. For instance, transfer of money from one account to another is a transaction consisting of two updates, one to each account.

A database system must ensure proper execution of transactions despite failures. Such a database system is classified based on the number of users who can use the system concurrently. A DBMS is single-user if at most one user at a time can use the system, and it is multi-user if many users can use the system. Multiple users can access the database because of the concept of multiprogramming. Multiprogramming allows the user to execute multiple programs or processes at the same time.

The concurrency may be achieved either interleaved fashion or by implementing multiple CPUs in parallel. The following example illustrates the same. Consider A, B, C and D as processes or transactions to be executed; here from time t1 to t2, A and B are executing in interleaved fashion where as from t3 to t4, C and D are executing in parallel.

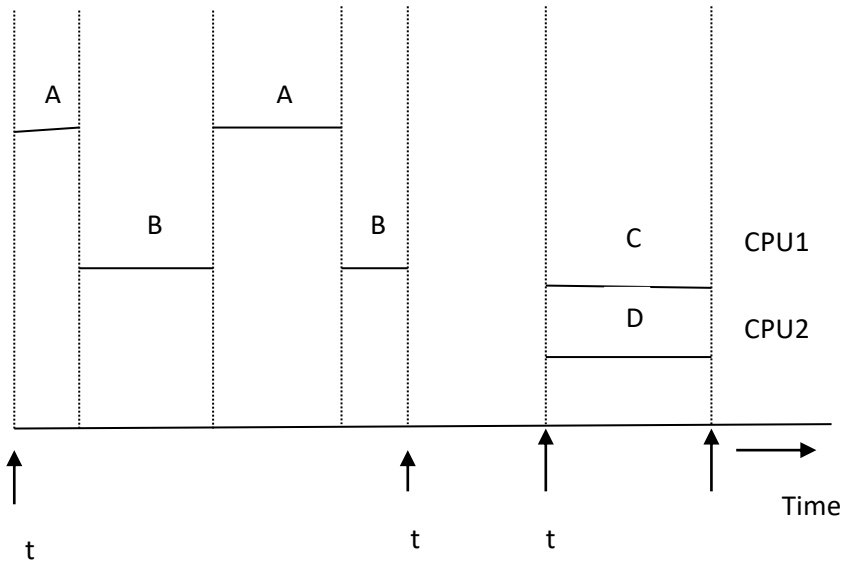


FIG. 1 Interleaved processing Vs Parellel

Desirable properties of a Transaction:

To ensure integrity of the data, a database system that executes a transaction should possess several properties, often called the ACID properties. The following are the ACID properties:

- **Atomicity:** a transaction is an atomic unit of processing; it is performed in its entirety or not performed at all. In some other way, a transaction is either completely done or completely undone.

- **Consistency:** a transaction is constancy preserving if its complete execution take(s) the database from one consistent state to another.

- **Isolation:** a transaction should appear as though it is being executed in isolation from other transactions. That is, the execution of a transaction should not be interfered with by any other transactions executing concurrently.

- **Durability or permanency:** the changes applied to the database by a committed transaction must persist in the database. These changes must not be lost because of any failure.

The acronym ACID is derived from all the first letter of each of the properties mention above.

The atomicity of a transaction requires that we execute a transaction to its completion. It is the responsibility of a transaction recovery sub system of a DBMS to ensure atomicity. If a transaction gets failed during its execution, the affect of such a transaction over the database must be undone.

The preservation of consistency is the responsibility of the programmer who writer the database programs or of the DBMS module that enforces integrity constraints. A consistent state of the database satisfies the constraints specified in the schema as well as any other constraints on the database should hold.

Isolation is enforced by the concurrency control subsystem of the DBMS. Every transaction does not make its updates visible to other transactions until it is committed. There is an approach of specifying isolation, that is, levels. A transaction is said to have Level 0 isolation if it does not overwrite the dirty reads of higher level transactions. Level 1 isolation has no lost updates; and level 2 isolation has no lost updates and no dirty reads. Finally Level 3

isolation (also called true isolation) has, in addition to degree 2 properties, repeatable reads.

Finally durability is the responsibility of recovery subsystem of the DBMS. Certain recovery protocols are to be applied in order to enforce durability and atomicity.

Transaction States

In the absence of failures, a transaction is completed successfully. However, not every time, a transaction is complete its execution successfully. Such a transaction is terms as **aborted**. Any aborted transaction must not show its affect on the database. Hence it requires being undone. Once the changes of the aborted transaction are made undone, we say that the transaction has been **rolled back**. Any successful transaction is said to be **committed**.

Once a transaction is committed, its updates may not be rolled back. The only way of aborting the updates of a committed transaction is to perform a compensating transaction over the database. For example, for a transaction adds Rs.200 to an account, the compensating transaction subtracts Rs.200 from the same account. However, a compensating transaction cannot be generated every time. Hence creation of such compensating transaction is left to the user.

The following is an abstract transaction model, which includes a set of states of a transaction:

Active: It is the initial state; the transaction stays in this state while it is executing.

Partially committed: a transaction is said to be partially committed if it completes execution of its last statement.

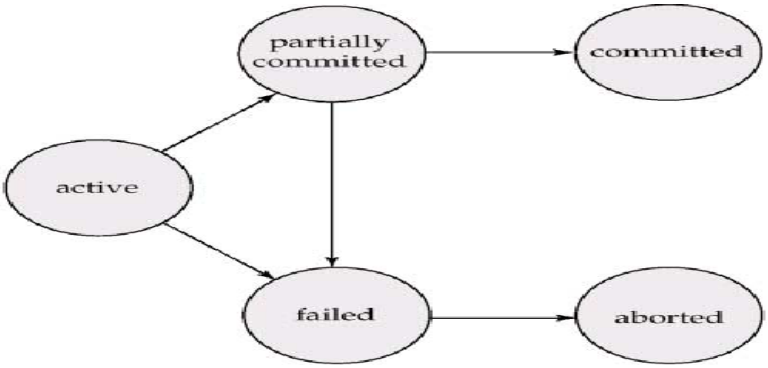


Fig. 2 Transaction states

Failed: A transaction is said to be failed after the discovery that the normal execution can no longer proceed.

Aborted: After the transaction has been rolled back and the database has been restored to its previous state before the transaction started its execution.

Committed: A transaction is said to be committed after successful completion.

The following is the state diagram for a transaction during its execution.

A transaction starts in the active state. When it finishes its final statement, it enters into 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. If at all, no failure is occurred the transaction may be committed.

A transaction that enters in a failure state after the system determines that the transaction can no longer proceed with its normal execution (because of hardware failures or logical errors). Such a transaction must be rolled back. Then, it enters the aborted state. At this point, the system has two options, either the transaction may be **restart**ed or it can be **kill**ed completely by the system.

The transaction may be restarted only if the transaction was aborted as a result of some hardware or software error. A restarted transaction is considered as a new transaction.

The transaction may be killed only if the error is corrected by rewriting the application program that causes the transaction.

Transaction and its Operations:

A transaction includes one or more database access operations. The operations include insertion, deletion, modification or retrieval operations.

The database operations that form a transaction can either be embedded within application program or they can be specified interactively via a high-level query language such as SQL. One way to represent transaction boundaries is by specifying explicit ***Begin Transaction*** and ***End Transaction***.

A transaction that performs only retrieval operation is said to be ***read-only transaction***.

read_item(X): reads a database item named X into a program variable. Assume the program variable is also named as X.

Execution of read_item(X) includes the following operations:

1. Find the address of the disk block that contains item X.
2. Copy that disk block into a buffer in main memory.
3. Copy item X from the buffer to the program variable X.

write_item(X): writes the value of program variable X into the data item named X.

Execution of write_item(X) includes the following operations:

1. Find the address of the disk block that contains item X.
2. Copy that disk block into a buffer in main memory.
3. Copy item X from the program variable name d X into its correct location in the buffer.
4. Store the updated block from the buffer block to disk.

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 is based on making copies of the database and assumes that only one transaction is active at a time. This scheme also assumes that the database is simply a file on the disk. A pointer called db-pointer is maintained on the disk; it points to the current copy of the database.

In the shadow copy scheme, a transaction that wants to update the database first creates a complete copy of the database. All updates are done to the new copy of the database, leaving the original copy, the shadow copy, untouched. If at all at any point of time the transaction has been aborted, the system merely deletes the new copy. The old copy of the database remains unaffected.

If the transaction completes, it is committed as follows. First the operating system is asked to make sure that all pages of the new copy of the database have been written out to disk (UNIX system uses *fsync* command for this purpose.) After the operating system has written all the pages to disk, the database system updates the pointer db-pointer to point to the new copy of the database; the new copy then becomes the current copy of the database. The old copy of the database is then deleted. The following figure depicts the scheme, showing the database state before and after the update.

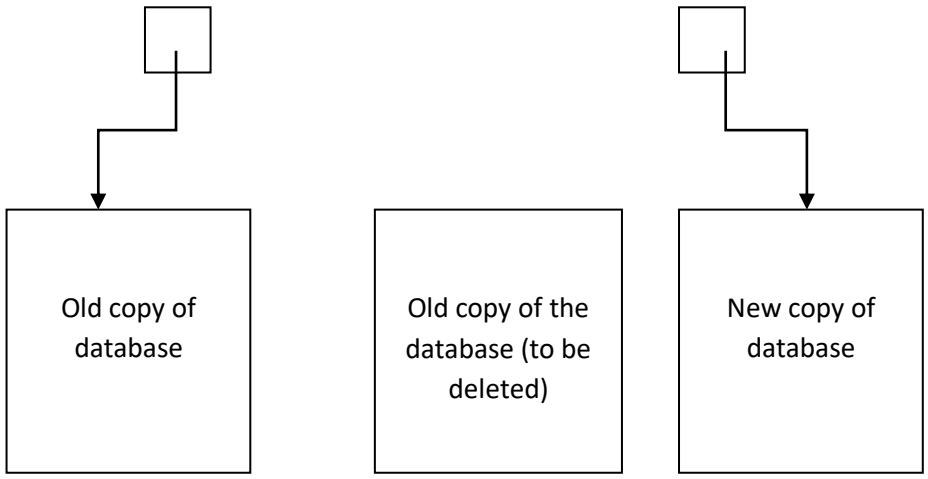


Figure 3 Shadow copy technique for atomicity and durability

Now let us watch how the scheme works in case of transaction and system failures. First consider the 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 is committed, all the updates that it 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 content of the database, and none of the

effects of the transaction will be visible on the database. Next, suppose that the system fails after the db-pointer has been updated on disk. Before the pointer is updated, all the updated pages of the new copy of the database were written to disk. Again, we assume that, once a file is written to disk, its content will not be damaged even if there is a system failure. Therefore, when the system restarts, it will read db-pointer and will thus see the contents of the database after all the updates performed by the transaction.

Thus, the atomicity and durability properties of transactions are ensured by the shadow-copy implementation of the recovery-management component.

Consider an example of text editor away from database concept. Many text editors use essentially implements the approach just described above. An entire editing session can be modeled as a transaction. The actions like reading, writing and updating are related to the transaction operations. A new file is used to store updated file. At the end of the editing session, if the updated file is to be saved, the text editor uses a file rename command to rename the new file to have the actual name of the file. The rename is assumed as an atomic operation and deletes the old file.

Unfortunately this implementation is extremely inefficient in the context of large databases, since executing a single transaction requires copying the entire database.

A Schedule S of n Transactions T_1, T_2, T_3, ……. T_n is an ordering of operations of the transactions subject to the constraint that, for each transaction T_i that participates in S, the operation T_i in S must appear in the same order in which they occur in T_i. However, that operation from other transactions T_j can be interleaved with the operations of T_i in S.

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 log. 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 $\mathbf{T_1}$ and $\mathbf{T_2}$ be two transactions that transfer funds from one account to

another. Transactions $\mathbf{T_1}$ transfers $50 from account **A** to account **B**. it is defined as

T_1: read (A);

A: =A-50;
Write (A);
Read (B);

B: =B+50;

Write(B)

Transactions $\mathbf{T_2}$ transfers 10 percent of the balance from account **A** to account **B**. it is defined as

T_2: 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 $\mathbf{T_1}$ followed by $\mathbf{T_2}$. 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 $\mathbf{T_1}$ appearing in the left column and instructions of $\mathbf{T_2}$ appearing in the right column. The final values of accounts **A** and **B**, after the execution in it 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) A:=A-50 Write(A) read(B) B:=B+50 write(B)	read(A) temp:=A*0.1 A:=A-temp write(A) read(B) B:=B+ temp write(B)

Figure 4 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 5** 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 fist 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) A:=A-50 write(A) read(B) B:=B+50 Write(B)	read(A) temp: =A*0.1 A: =A-temp write(A) read(B) B: =B+ temp write(B)

Figure 5 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 $\mathbf{T_1}$ followed by $\mathbf{T_2}$. The **sum A + B** is indeed preserved.

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

Figure 6 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 7**.

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

Figure 7 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.

Serializability

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

Since transactions are programs, it is computationally difficult to determine exactly what operations a transaction performs and how operations of various transactions interact. 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 (**Q**) instruction and a write (**Q**) instruction on a data item **Q**, a transaction may perform an arbitrary sequence of operations on the copy of **Q** that is residing in the local buffer of the transaction. 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 8**.

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

Figure 8 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.

Conflict serializability:

Let us consider a schedule s in which there are two consecutive instructions Ii and Ij refer to different data items, and then we can swap Ii and Ij without affecting the results of any instruction in the schedule. However if Ii and Ij 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=read (**Q**), I_j=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=read (**Q**), I_j=write (**Q**). If Ii 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=write (**Q**), I_j=write (**Q**) .the order of I_i and I_j matters for reasons similar to those of the previous case.

4) I_i=write (**Q**), I_j=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 13.5**. 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 **Ij** 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 $\mathbf{T_2}$ in schedule 3 of **figure 13.5.2** does not conflict with the read (**B**) instruction of $\mathbf{T_1}$, we can swap these instructions to generate an equivalent schedule, schedule 5, in **figure 9**.

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

Figure 9 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 $\mathbf{T_1}$ with the read (**A**) instruction of $\mathbf{T_2}$.
- Swap the write (**B**) instruction of $\mathbf{T_1}$ with the write (A) instruction of $\mathbf{T_2}$.
- Swap the write (B) instruction of $\mathbf{T_1}$ with the read (**A**) instruction of $\mathbf{T_2}$.

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

Figure 10 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, schedules 1 is not conflict equivalent to schedule 2. However, schedule 1 is conflict equivalence to schedule 3,because the read (**B**) and write (**B**) instruction of $\mathbf{T_1}$ can be swapped with the read (**A**) and write (**A**) instruction of $\mathbf{T_2}$.

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 11**; it consists of only the significant operations (that is, the read and write) of transactions $\mathbf{T_3}$ and $\mathbf{T_4}$. This schedule is not conflict serializable, since it is not equivalent to either the serial schedule <$\mathbf{T_3}$, $\mathbf{T_4}$> or the serial schedule <$\mathbf{T_4}$, $\mathbf{T_3}$>

T_3	T_4
read(Q)	
	write(Q)
write(Q)	

Figure 11 Schedule 7.

It is possible to have two schedulers that produce the same outcome, but that are not conflict equivalent. For example, consider transaction $\mathbf{T_5}$, which transfers $10 from account **B** to **A**.

Let schedule 8 be as defined in **figure 12** we claim that schedule 8 is not conflict equivalent to the serial schedule <$\mathbf{T_1, T_5}$>,

T_1	T_5
read(A)	
A:=A-50	
	read(A)
write(A)	B:=B-10
	write(B)
	read(B)
	A:=A+10
read(B)	write(A)
B:=B+50	
Write(B)	

Figure 12 Schedule 8.

Since, in schedule 8, the write (**B**) instruction of $\mathbf{T_5}$ conflicts with the read (**B**) instruction of $\mathbf{T_1}$.Thus, we cannot move all the instructions of $\mathbf{T_1}$ before those of $\mathbf{T_5}$ 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 <$\mathbf{T_1, T_5}$> 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<T_1, T_5>, 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.

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.

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**.

1. 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_i must, in schedule s_1, also read the value of **Q** that was produced by the same write (**Q**) operation of transaction T_j.

2. 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 $\mathbf{T_6}$, and obtain schedule 9 in **figure 13**. Schedule 9 is view serializable. Indeed, it is view equivalent to the serial schedule <$\mathbf{T_3}$, $\mathbf{T_4}$, and $\mathbf{T_6}$>, since the one read (**Q**) instruction reads the initial value of **Q** in both schedules, and to $\mathbf{T_6}$ performs the final value of **Q** in both schedules.

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

Figure 13 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 $\mathbf{T_4}$ and $\mathbf{T_6}$ 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.

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 $\mathbf{T_i}$ 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.

Recoverable schedules

Consider schedule 11 in **figure 14** 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 14 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;

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 15**.

T_{10}	T_{11}	T_{12}
read(A)		
read(B)		
write(A)	read(A) write(A)	
		read(A)

Figure 15 Schedule 11.

Transaction $\mathbf{T_{10}}$ writes a value of **A** that is read by transaction $\mathbf{T_{11}}$.Transaction $\mathbf{T_{11}}$ writes a value of **A** that is read by transaction $\mathbf{T_{12}}$.suppose that, at this point, $\mathbf{T_{10}}$ fails. $\mathbf{T_{10}}$ must be rolled back. Since $\mathbf{T_{12}}$ is dependent on $\mathbf{T_{11}}$, $\mathbf{T_{12}}$ 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 $\mathbf{T_i}$ and $\mathbf{T_j}$ appear before the read operation of $\mathbf{T_j}$. It is easy to verify that every cascadeless schedule is also recoverable.

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 casacdeless 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.

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.

Testing for serializability

When designing concurrency control schemes, we must show that schedules generated by the scheme are serilaizable. 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 16** 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 16 Precedence graph for (a) schedule 1 and (b) schedule 2.

Similarly, **Figure 16** 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 17.**

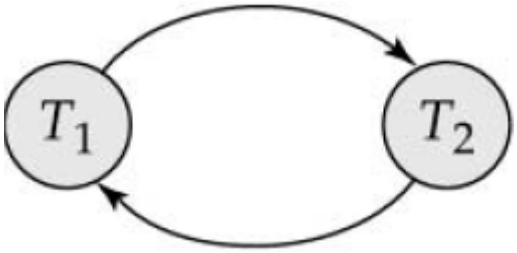


Figure 17 Precedence graph for schedule 4.

It contains the edge $\mathbf{T_1 \rightarrow T_2}$, because $\mathbf{T_1}$ executes read (**A**) before $\mathbf{T_2}$ executes write (**A**). It also contains the edge $\mathbf{T_2 \rightarrow T_1}$, because T_2 executes read (**B**) before $\mathbf{T_1}$ 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 18 a** has the two acceptable linear orderings shown in **figures 18 b** and **c**.

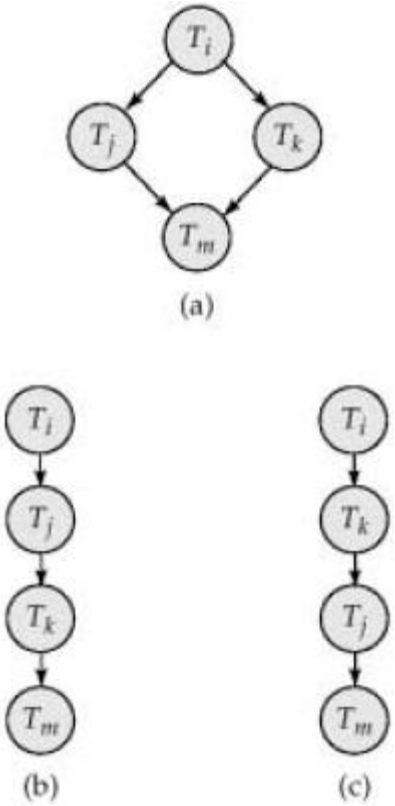


Figure 18 Illustration of topological sorting.

Thus, to test for conflict serializability, we need to construct the preceding 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 $\mathbf{n^2}$ operations, where **n** is the number of vertices in the graph. Thus, we have a practical scheme for determining conflict serializability.

Returning to out 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 serializability.

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 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 serializablity. 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.

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.

Technical Terms

1. Transaction:

The term transaction refers to a collection of operations that forma 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.

Model Questions

1. List the AICD 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?

2. CONCURRENCY CONTROL

Objective

- What is Concurrency Control?
- Lock-Based Protocols
- Timestamp-Based Protocols
- Validation-Based Protocols
- Multiple Granularities
- Multi-version Schemes
- Deadlock Handling

Structure of the Lesson

Lock Based Protocols
Time stamp based Protocols
Validation Based protocols
Multiple granularities
Multi-version schemes
Deadlock based Protocols

When several transactions execute concurrently in the database, however, the isolation property may no longer be preserved. To ensure that it is, the system must control the interactions among the concurrent transactions; this control is achieved through one of the variety of mechanisms called ***concurrency control*** schemes.

The concurrency control schemes that we discuss here are all based on Serializability property.

Lock Based Protocols

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

Locks

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

1. **Shared:** If a transaction T_i has obtained a shared- mode lock (denoted by S) on item Q, then T_i can read, but cannot write, Q.
2. **Exclusive:** If a transaction T_i has obtained an exclusive-mode lock (denoted by X) on item Q, then T_i can both read and write Q.

	S	X
S	True	false
X	False	False

Lock compatibility matrix

T1	**T2**	**T3**	**T4**
Lock –X (B) Read (B) B:=B-50 Write (B) Unlock (B) Lock – X (A) Read (A) A: = A+50 Write (A) Unlock (A)	Lock –S (A) Read (A) Unlock (A) Lock – S (B) Read (B) Unlock (B) Display (A+B)	Lock –X (B) Read (B) B:= B-50 Write (B) Lock –X (A) Read (A) A: =A+50 Write (A) Unlock (B) Unlock (A)	Lock –s (A) Read (A) Lock – s (B) Read (B) Display (A+B) Unlock (A) Unlock (B)

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) Grant – S(A,T_2)	
	Read(A) Unlock(A) Lock-S(B) Grant – S(B, T_2) Read(B) Unlock(B) Display(A+B)	
Lock – X(A)		
	Grant – X(A, T_2)	
Read(A) A: = A+50 Write(A) Unlock(A)		

Schedule – 1

Granting of Locks

When a transaction requests a lock on a data item over 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, transactionT2 has a shared mode lock on a data item, and another transaction T1 requests an exclusive mode lock on the data item. Unless we take care, the transactions may get into situation where the transactions have no progress. Such a situation is known to be starvation.

We can avoid starvation of transactions by granting locks in the following manner: when a transaction Ti requests a lock on a data item Q in a mode that conflicts with M. the concurrency-control manager grants the lock provided that

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

Two-Phase Locking Protocol

One protocol that ensures serializability is that the two-phase protocol. A transaction is said to follow the two-phase locking protocol if all locking operations (read, write locks) precede the first unlock operation in the transaction. Such a transaction requires to be divided into 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 lock.

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 T3 and T4 are two-phase. On the other hand, transactions T1 and T2 are not two-phase. Note that unlock instructions do not need to appear at the end of the transaction. For example, in the case of transaction T3, we could move the unlock (B) instruction to just after the lock-X(A) instruction and still retain the two-phase locking property. The two-phase locking can ensure conflict serializability. Consider any transaction. The point in the schedule where the transaction has obtained its final lock (the end of growing phase) is called the **lock point** of the transaction.

Two-phase locking may limit the amount of concurrency that can occur in a schedule. Cascading rollbacks can be avoided by a modification of two phase locking called, **strict 2PL**. 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 verity that, with rigorous 2PL, transactions can be serialized in the order in which they

commit. Most database systems implement either strict or rigorous 2PL.

- There can be conflict serializable schedules that cannot be obtained if two-phase locking is used.
- However, in the absence of extra information (e.g., ordering of access to data), two-phase locking is needed for conflict serializability in the following sense:

Given a transaction T_i that does not follow two-phase locking, we can find a transaction T_j that uses two-phase locking, and a schedule for T_i and T_j that is not conflict serializable.

Two-phase locking with lock conversions:

– First Phase:

1. can acquire a lock-S on item
2. can acquire a lock-X on item
3. can convert a lock-S to a lock-X (upgrade)
4.

– Second Phase:

–

1. can release a lock-S
2. can release a lock-X
3. can convert a lock-X to a lock-S (downgrade)

This protocol assures serializability. But still relies on the programmer to insert the various locking instructions.

A transaction T_i issues the standard read/write instruction, without explicit locking calls.

The operation read(*D*) is processed as:

if T_i has a lock on *D*
then
read(*D*)
else begin
if necessary wait until no other

transaction has a **lock-X** on *D*
grant T_i a **lock-S** on *D*;
read(*D*)
end

write*(D)* is processed as:

if T_i has a **lock-X** on *D*
then
write(*D*)
else begin
if necessary wait until no other trans. has any lock on *D*,
if T_i has a **lock-S** on *D*
then
upgrade lock on *D* to **lock-X**
else
grant T_i a **lock-X** on *D*
write(*D*)
end;

All locks are released after commit or abort

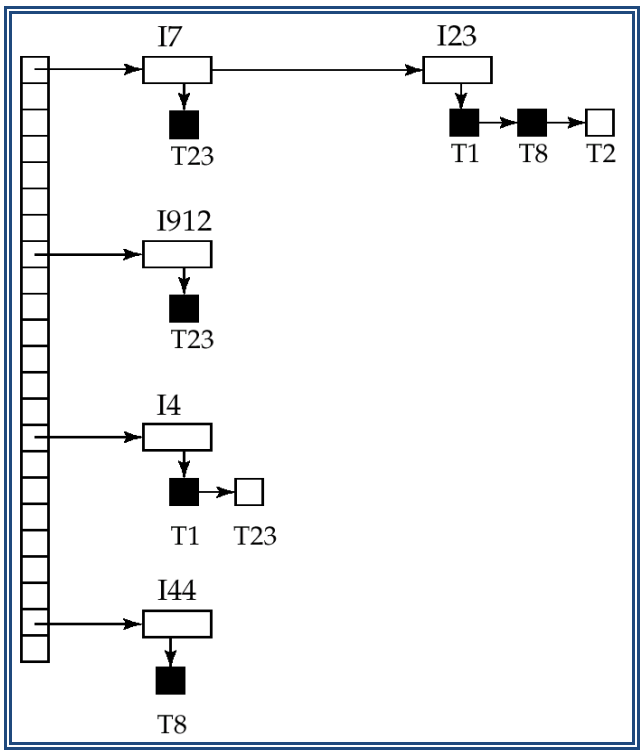
Implementation of Locking:

The main aspects of implementation of locking are:

1. A **lock manager** can be implemented as a separate process to which transactions send lock and unlock requests
2. The lock manager replies to a lock request by sending a lock grant messages (or a message asking the transaction to roll back, in case of a deadlock)
3. The requesting transaction waits until its request is answered
4. The lock manager maintains a data-structure called a **lock table** to record granted locks and pending requests
5. The lock table is usually implemented as an in-memory hash table indexed on the name of the data item being locked**.**

Lock Table

1. Black rectangles indicate granted locks, white ones indicate waiting requests
2. Lock table also records the type of lock granted or requested
3. New request is added to the end of the queue of requests for the data item, and granted if it is compatible with all earlier locks
4. Unlock requests result in the request being deleted, and later requests are checked to see if they can now be granted
5. If transaction aborts, all waiting or granted requests of the transaction are deleted
    a. lock manager may keep a list of locks held by each transaction, to implement this efficiently



Graph based Protocols

Two phase locking protocol is both necessary and sufficient for ensuring serializability where as Graph-based protocols are an alternative to two-phase locking.

→Impose a partial ordering → on the set **D** = {$d_1, d_2, ..., d_h$} of all data items.

1. If $d_i \rightarrow d_j$ then any transaction accessing both d_i and d_j must access d_i before accessing d_j.
2. Implies that the set **D** may now be viewed as a directed acyclic graph, called a *database graph*.

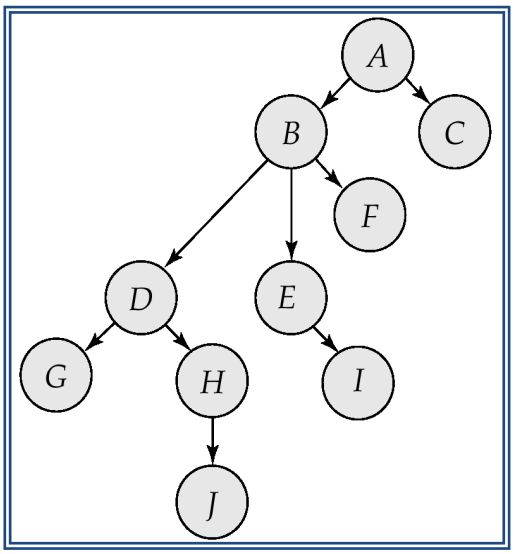
➔ The *tree-protocol* is a simple kind of graph protocol.

➔ Only exclusive locks are allowed.

➔ The first lock by T_i may be on any data item. Subsequently, a data Q can be locked by T_i only if the parent of Q is currently locked by T_i.

➔ Data items may be unlocked at any time.

➔ A data item that has been locked and unlocked by T_i cannot subsequently be relocked by T_i



Tree based protocol

Tree Protocol

1. The tree protocol ensures conflict serializability as well as freedom from deadlock.
2. Unlocking may occur earlier in the tree-locking protocol than in the two-phase locking protocol.
    a. shorter waiting times, and increase in concurrency
    b. protocol is deadlock-free, no rollbacks are required
3. Drawbacks
    a. Protocol does not guarantee recoverability or cascade freedom
        i. Need to introduce commit dependencies to ensure recoverability
    b. Transactions may have to lock data items that they do not access.
        i. increased locking overhead, and additional waiting time
        ii. potential decrease in concurrency
4. Schedules not possible under two-phase locking are possible under tree protocol, and vice versa

Serializable Schedule u Tree Protocol

T_{10}	T_{11}	T_{12}	T_{13}
lock-X(B)			
	lock-X(D)		
	lock-X(H)		
	unlock(D)		
lock-X(E)			
lock-X(D)			
unlock(B)			
unlock(E)			
		lock-X(B)	
		lock-X(E)	
	unlock(H)		
lock-X(G)			
unlock(D)			
			lock-X(D)
			lock-X(H)
			unlock(D)
			unlock(H)
		unlock(E)	
		unlock(B)	
unlock (G)			

Time stamp based Protocols

Validation Based protocols

In validation based protocols, execution of transaction T_i is done in three phases.

1. Read and execution phase: Transaction T_i writes only to temporary local variables.

2. Validation phase: Transaction T_i performs a ``validation test'' to determine if local variables can be written without violating serializability.

3. Write phase: If T_i is validated, the updates are applied to the database; otherwise, T_i is rolled back.

The three phases of concurrently executing transactions can be interleaved, but each transaction must go through the three phases in that order.

Assume for simplicity that the validation and write phase occur together, atomically and serially i.e., only one transaction executes validation/write at a time.

It is also called as **optimistic concurrency control** since transaction executes fully in the hope that all will go well during validation.

Each transaction T_i has 3 timestamps:

1. Start(T_i) : the time when T_i started its execution
2. Validation(T_i): the time when T_i entered its validation phase
3. Finish(T_i) : the time when T_i finished its write phase

Serializability order is determined by timestamp given at validation time, to increase concurrency thus TS (T_i) is given the value of Validation(T_i).

This protocol is useful and gives greater degree of concurrency if probability of conflicts is low because the serializability order is not pre-decided, and relatively few transactions will have to be rolled back.

If for all T_i with TS (T_i) < TS (T_j) either one of the following condition holds:

a. **finish**(T_i) < **start**(T_j)
b. **start**(T_j) < **finish**(T_i) < **validation**(T_j) **and** the set of data items written by T_i does not intersect with the set of data items read by T_j.
   then validation succeeds and T_j can be committed. Otherwise, validation fails and T_j is aborted.

***Justification*:** Either the first condition is satisfied, and there is no overlapped execution, or the second condition is satisfied and

- ➔ the writes of T_j do not affect reads of T_i since they occur after T_i has finished its reads.
- ➔ the writes of T_i do not affect reads of T_j since T_j does not read any item written by T_i.

Schedule produced using Validation

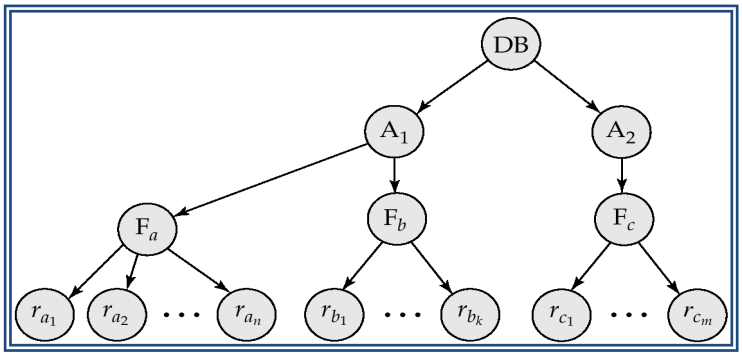
T_{14}	T_{15}
read(B)	
	read(B)
	$B := B - 50$
	read(A)
	$A := A + 50$
read(A)	
⟨*validate*⟩	
display($A + B$)	
	⟨*validate*⟩
	write(B)
	write(A)

Multiple granularities

1. Allow data items to be of various sizes and define a hierarchy of data granularities, where the small granularities are nested within larger ones
2. Can be represented graphically as a tree (but don't confuse with tree-locking protocol)
3. When a transaction locks a node in the tree *explicitly*, it *implicitly* locks all the node's descendents in the same mode.
4. Granularity of locking (level in tree where locking is done):

a. **fine granularity** (lower in tree): high concurrency, high locking overhead

b. **coarse granularity** (higher in tree): low locking overhead, low concurrency



The levels, starting from the coarsest (top) level are:

1. *database*
2. *area*
3. *file*
4. *record*

Intension of Locking modes

In addition to S and X lock modes, there are three additional lock modes with multiple granularity:

➔ ***intention-shared*** (IS): indicates explicit locking at a lower level of the tree but only with shared locks.

➔ ***intention-exclusive*** (IX): indicates explicit locking at a lower level with exclusive or shared locks.

➔ ***shared and intention-exclusive*** (SIX): the subtree rooted by that node is locked explicitly in shared mode and explicit locking is being done at a lower level with exclusive-mode locks.

Intention locks allow a higher level node to be locked in S or X mode without having to check all descendent nodes.

Compatibility Matrix of Lock modes:

	IS	IX	S	S IX	X
IS	✓	✓	✓	✓	×
IX	✓	✓	×	×	×
S	✓	×	✓	×	×
S IX	✓	×	×	×	×
X	×	×	×	×	×

Transaction T_i can lock a node Q, using the following rules:

1. The lock compatibility matrix must be observed.
2. The root of the tree must be locked first, and may be locked in any mode.
3. A node Q can be locked by T_i in S or IS mode only if the parent of Q is currently locked by T_i in either IX or IS mode.
4. A node Q can be locked by T_i in X, SIX, or IX mode only if the parent of Q is currently locked by T_i in either IX or SIX mode.
5. T_i can lock a node only if it has not previously unlocked any node (that is, T_i is two-phase).
6. T_i can unlock a node Q only if none of the children of Q are currently locked by T_i.

Observe that locks are acquired in root-to-leaf order, whereas they are released in leaf-to-root order.

Deadlock based Protocols

System is deadlocked if there is a set of transactions such that every transaction in the set is waiting for another transaction in the set.

Consider the following two transactions:

T_1:	write (X)	T_2:	write(Y)
	write(Y)		write(X)

Schedule with deadlock:

T_1	T_2
lock-X on X write (X)	
	lock-X on Y write (X) wait for **lock-X** on X
wait for **lock-X** on Y	

Deadlock prevention protocols ensure that the system will *never* enter into a deadlock state. Some prevention strategies:

1. They require that each transaction locks all its data items before it begins execution (pre-declaration).
2. They impose partial ordering of all data items and require that a transaction can lock data items only in the order specified by the partial order (graph-based protocol).

Following schemes use transaction timestamps for the sake of deadlock prevention alone.

Wait-die scheme — non-preemptive

1. Older transaction may wait for younger one to release data item. Younger transactions never wait for older ones; they are rolled back instead.

2. a transaction may die several times before acquiring needed data item

Wound-wait scheme — preemptive

a. Older transaction *wounds* (forces rollback) of younger transaction instead of waiting for it. Younger transactions may wait for older ones.
b. May be fewer rollbacks than *wait-die* scheme.

Both in *wait-die* and in *wound-wait* schemes, a rolled back transaction is restarted with its original timestamp. Older transactions thus have precedence over newer ones, and starvation is hence avoided.

Timeout-Based Schemes:

a. A transaction waits for a lock only for a specified amount of time. After that, the wait times out and the transaction is rolled back.
b. thus deadlocks are not possible
c. Simple to implement; but starvation is possible. Also difficult to determine good value of the timeout interval.

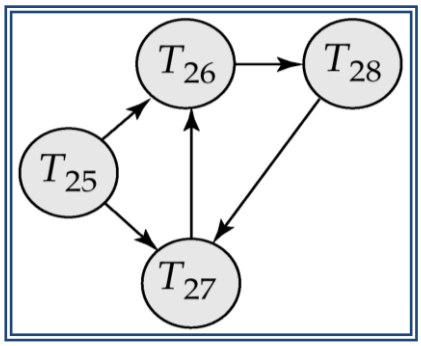
Deadlocks can be described as a *wait-for graph*, which consists of a pair $G = (V,E)$,

- ➔ V is a set of vertices (all the transactions in the system)
- ➔ E is a set of edges; each element is an ordered pair $T_i \rightarrow T_j$.

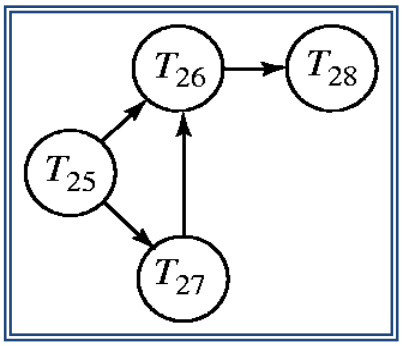
If $T_i \rightarrow T_j$ is in E, then there is a directed edge from T_i to T_j, implying that T_i is waiting for T_j to release a data item.

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

The system is in a deadlock state if and only if the wait-for graph has a cycle. It must invoke a deadlock-detection algorithm periodically to look for cycles.



Wait for graph without a cycle



Wait for graph with a cycle

Deadlock Recovery:

When deadlock is detected:

- Some transaction will have to rolled back (made a victim) to break deadlock.
- Select that transaction as victim that will incur minimum cost.
- Rollback -- determine how far to roll back transaction
- Total rollback: Abort the transaction and then restart it.
- More effective to roll back transaction only as far as necessary to break deadlock.
- Starvation happens if same transaction is always chosen as victim. Include the number of rollbacks in the cost factor to avoid starvation

Summary:

When several transactions execute concurrently in the database, the consistency of the 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 of a variety of mechanisms called concurrency control. To ensure serializability, we introduce various concurrency control schemes.

All these schemes either delay an operation or abort the transaction that issued the operation. The most common schemes are locking protocols, timestamp based, validation based and multiversion schemes.

A two phase locking protocol allows a transaction to lock a new item only if that transaction has not yet unlocked any data item. It ensures serializability.

A time stamp ordering scheme ensures serializability by selecting an ordering in advance between every pair of transactions.

A validation scheme is appropriate concurrency control method in cases where a majority of transactions are read only transactions, and thus the rate of conflicts among these transactions is low.

Model Questions:

1. What is concurrency control? List out various various concurrency control schemes in detail.
2. Explain two phase locking in detail?
3. Explain the need of time stamp ordering and discuss how it works?
4. What is a dead lock protocol? Explain its functionality in detail.