UNIT -1
The Worlds of Database Systems

INTRODUCTION TO BASIC CONCEPTS OF DATABASE SYSTEMS:

What is Data?

The raw facts are called as data. The word "raw" indicates that they have not been processed.

Ex: For example 89 is the data.

What is information?

The processed data is known as information.

Ex: Marks: 89; then it becomes information.

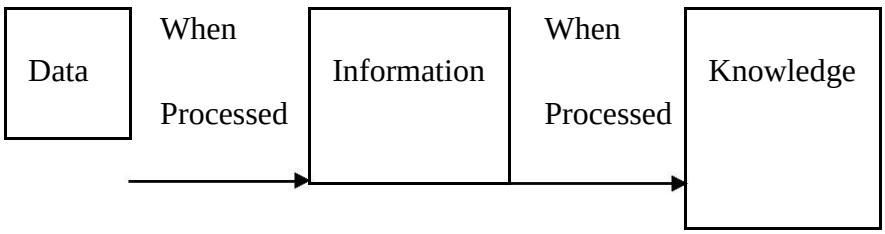
What is Knowledge?

1. Knowledge refers to the practical use of information.

2. Knowledge necessarily involves a personal experience.

DATA/INFORMATION PROCESSING:

The process of converting the data (raw facts) into meaningful information is called as data/information processing.



Note: In business processing knowledge is more useful to make decisions for any organization.

DIFFERENCE BETWEEN DATA AND INFORMATION:

DATA	INFORMATION
1.Raw facts	1.Processed data
2. It is in unorganized form	2. It is in organized form
3. Data doesn't help in decision making process	3. Information helps in decision making process

FILE ORIENTED APPROACH:

The earliest business computer systems were used to process business records and produce information. They were generally faster and more accurate than equivalent manual systems. These systems stored groups of records in separate files, and so they were called **file processing systems.**

1. File system is a collection of data. Any management with the file system, user has to write the procedures

2. File system gives the details of the data representation and Storage of data.

3. In File system storing and retrieving of data cannot be done efficiently.

4. Concurrent access to the data in the file system has many problems like a Reading the file while other deleting some information, updating some information

5. File system doesn't provide crash recovery mechanism.
Eg. While we are entering some data into the file if System crashes then content of the file is lost.

6. Protecting a file under file system is very difficult.

The typical file-oriented system is supported by a conventional operating system. Permanent records are stored in various files and a number of different application programs are written to extract records from and add records to the appropriate files.

DISADVANTAGES OF FILE-ORIENTED SYSTEM:

The following are the disadvantages of File-Oriented System:

Data Redundancy and Inconsistency:

Since files and application programs are created by different programmers over a long period of time, the files are likely to be having different formats and the programs may be written in several programming languages. Moreover, the same piece of information may be duplicated in several places. This redundancy leads to higher storage and access cost. In addition, it may lead to data inconsistency.

Difficulty in Accessing Data:

The conventional file processing environments do not allow needed data to be retrieved in a convenient and efficient manner. Better data retrieval system must be developed for general use.

Data Isolation:

Since data is scattered in various files, and files may be in different formats, it is difficult to write new application programs to retrieve the appropriate data.

Concurrent Access Anomalies:

In order to improve the overall performance of the system and obtain a faster response time, many systems allow multiple users to update the data simultaneously. In such an environment, interaction of concurrent updates may result in inconsistent data.

Security Problems:

Not every user of the database system should be able to access all the data. For example, in banking system, payroll personnel need only that part of the database that has information about various bank employees. They do not need access to information about customer accounts. It is difficult to enforce such security constraints.

Integrity Problems:

The data values stored in the database must satisfy certain types of consistency constraints. For example, the balance of a bank account may never fall below a prescribed amount. These constraints are enforced in the system by adding appropriate code in the various

application programs. When new constraints are added, it is difficult to change the programs to enforce them. The problem is compounded when constraints involve several data items for different files.

Atomicity Problem:

A computer system like any other mechanical or electrical device is subject to failure. In many applications, it is crucial to ensure that once a failure has occurred and has been detected, the data are restored to the consistent state existed prior to the failure

Example:

Consider part of a savings-bank enterprise that keeps information about all customers and savings accounts. One way to keep the information on a computer is to store it in operating system files. To allow users to manipulate the information, the system has a number of application programs that manipulate the files, including:

- A program to debit or credit an account
- A program to add a new account
- A program to find the balance of an account
- A program to generate monthly statements

Programmers wrote these application programs to meet the needs of the bank. New application programs are added to the system as the need arises. For example, suppose that the savings bank decides to offer checking accounts.

As a result, the bank creates new permanent files that contain information about all the checking accounts maintained in the bank, and it may have to write new application programs to deal with situations that do not arise in savings accounts, such as overdrafts. Thus, as time goes by, the system acquires more files and more application programs. The system stores permanent records in various files, and it needs different

Application programs to extract records from, and add records to, the appropriate files. Before database management systems (DBMS) came along, organizations usually stored information in such systems. Organizational information in a file-processing system has a number of major disadvantages:

1. Data Redundancy and Inconsistency:

The address and telephone number of a particular customer may appear in a file that consists of savings-account records and in a file that consists of checking-account records. This redundancy leads to higher storage and access cost. In, it may lead to data inconsistency; that is, the various copies of the same data may no longer agree. For example, a changed customer address may be reflected in savings-account records but not elsewhere in the system.

2. **Difficulty in Accessing Data:**

Suppose that one of the bank officers needs to find out the names of all customers who live within a particular postal-code area. The officer asks the data-processing department to generate such a list. Because there is no application program to generate that. The bank officer has now two choices: either obtain the list of all customers and extract the needed information manually or ask a system programmer to write the necessary application program. Both alternatives are obviously unsatisfactory.

3. **Data Isolation:**

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

4. **Integrity Problems:**

The balance of a bank account may never fall below a prescribed amount (say, $25). Developers enforce these constraints in the system by adding appropriate code in the various application programs. However, when new constraints are added, it is difficult to change the programs to enforce them. The problem is compounded when constraints involve several data items from different files.

5. **Atomicity Problems:**

A computer system, like any other mechanical or electrical device, is subject to failure. In many applications, it is crucial that, if a failure occurs, the data be restored to the consistent state that existed prior to the failure. Consider a program to transfer $50 from account *A* to account *B*. If a system failure occurs during the execution of the program, it is possible that the $50 was removed from account *A* but was not credited to account *B*, resulting in an inconsistent database state. Clearly, it is essential to database consistency that either both the credit and debit occur, or that neither occur. That is, the funds transfer must be *atomic*—it must happen in its entirety or not at all. It is difficult to ensure atomicity in a conventional file-processing system.

6. **Concurrent-Access Anomalies:**

For the sake of overall performance of the system and faster response, many systems allow multiple users to update the data simultaneously. In such an environment, interaction of concurrent updates may result in inconsistent data. Consider bank account *A*, containing $500. If two customers withdraw funds (say $50 and $100 respectively) from account *A* at about the same time, the result of the concurrent executions may leave the account in an incorrect (or inconsistent) state. Suppose that the programs executing on behalf of each withdrawal read the old balance, reduce that value by the amount being withdrawn, and write the result back. If the two programs run concurrently, they may both read the value $500, and write back $450 and $400, respectively. Depending on which one writes the value last, the account may contain $450 or $400, rather than the correct value of $350. To guard against this possibility, the system must maintain some form of supervision. But supervision is difficult to provide because data may be accessed by many different application programs that have not been coordinated previously.

7. **Security Problems:**

Not every user of the database system should be able to access all the data. For example, in a banking system, payroll personnel need to see only that part of the database that has information about the various bank employees. They do not need access to information about customer accounts. But, since application programs are added to the system in an ad hoc manner, enforcing such security constraints is difficult. These difficulties, among others, prompted the development of database systems.

INTRODUCTION TO DATABASES:

History of Database Systems:

1950s and early 1960s:

- Magnetic tapes were developed for data storage
- Data processing tasks such as payroll were automated, with data stored on tapes.
- Data could also be input from punched card decks, and output to printers.
- Late 1960s and 1970s: The use of hard disks in the late 1960s changed the scenario for data processing greatly, since hard disks allowed direct access to data.
- With disks, network and hierarchical databases could be created that allowed data structures such as lists and trees to be stored on disk. Programmers could construct and manipulate these data structures.
- With disks, network and hierarchical databases could be created that allowed data structures such as lists and trees to be stored on disk. Programmers could construct and manipulate these data structures.
- In the 1970's the EF CODD defined the **Relational Model.**

In the 1980's:

- Initial commercial relational database systems, such as IBM DB2, Oracle, Ingress, and DEC Rdb, played a major role in advancing techniques for efficient processing of declarative queries.
- In the early 1980s, relational databases had become competitive with network and hierarchical database systems even in the area of performance.
- The 1980s also saw much research on parallel and distributed databases, as well as initial work on object-oriented databases.

Early 1990s:

- The SQL language was designed primarily in the 1990's.
- And this is used for the transaction processing applications.
- Decision support and querying re-emerged as a major application area for databases.

- Database vendors also began to add object-relational support to their databases.

Late 1990s:

- The major event was the explosive growth of the World Wide Web.
- Databases were deployed much more extensively than ever before. Database systems now had to support very high transaction processing rates, as well as very high reliability and 24 * 7 availability (availability 24 hours a day, 7 days a week, meaning no downtime for scheduled maintenance activities).
- Database systems also had to support Web interfaces to data.

The Evolution of Database systems:

The Evolution of Database systems are as follows:

1. File Management System
2. Hierarchical database System
3. Network Database System
4. Relational Database System

File Management System:

The file management system also called as FMS in short is one in which all data is stored on a single large file. The main disadvantage in this system is searching a record or data takes a long time. This lead to the introduction of the concept, of indexing in this system. Then also the FMS system had lot of drawbacks to name a few like updating or modifications to the data cannot be handled easily, sorting the records took long time and so on. All these drawbacks led to the introduction of the Hierarchical Database System.

Hierarchical Database System:

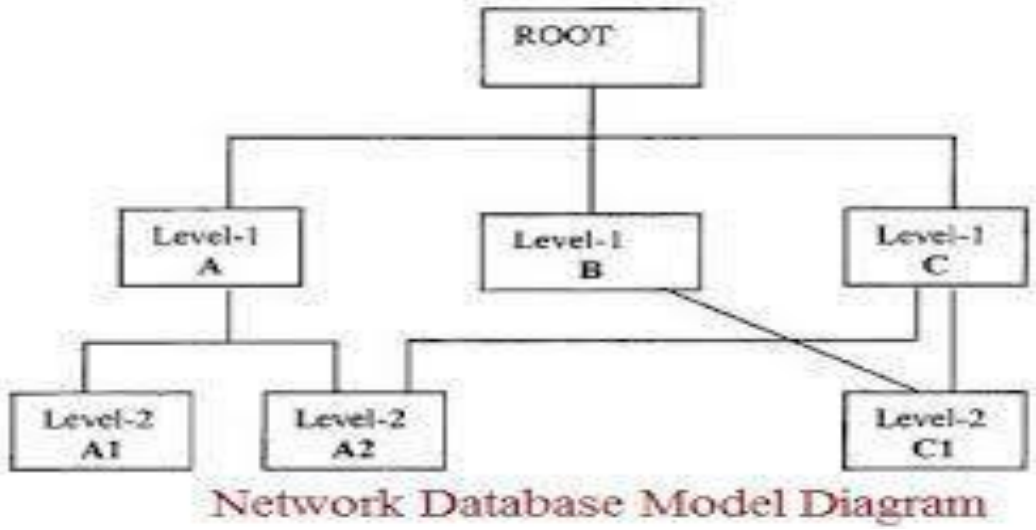
The previous system FMS drawback of accessing records and sorting records which took a long time was removed in this by the introduction of parent-child relationship between records in database. The origin of the data is called the root from which several branches have data at different levels and the last level is called the

leaf. The main drawback in this was if there is any modification or addition made to the structure then the whole structure needed alteration which made the task a tedious one. In order to avoid this next system took its origin which is called as the Network Database System.

Fig: Hierarchical Database System

Network Database System:

In this the main concept of many-many relationships got introduced. But this also followed the same technology of pointers to define relationships with a difference in this made in the introduction if grouping of data items as sets.



Network Database Model Diagram

Relational Database System:

In order to overcome all the drawbacks of the previous systems, the Relational Database System got introduced in which data get organized as tables and each record forms a row with many fields or attributes in it. Relationships between tables are also formed in this system.

Name	FName	City	Age	Salary
Smith	John	3	35	$280
Doe	Jane	1	28	$325
Brown	Scott	3	41	$265
Howard	Shemp	4	48	$359
Taylor	Tom	2	22	$250

DATABASE:

A database is a collection of related data.

(OR)

A database is a collection of information that is organized so that it can be easily accessed, managed and updated.

Examples / Applications of Database Systems:

The following are the various kinds of applications/organizations uses databases for their business processing activities in their day-to-day life. They are:

1. **Banking:** For customer information, accounts, and loans, and banking transactions.

2. **Airlines:** For reservations and schedule information. Airlines were among the first to use

databases in a geographically distributed manner—terminals situated around the world accessed the central database system through phone lines and other data networks.

3. **Universities:** For student information, course registrations, and grades.

4. **Credit Card Transactions:** For purchases on credit cards and generation of monthly statements.

5. **Telecommunication:** For keeping records of calls made, generating monthly bills, maintaining balances on prepaid calling cards, and storing information about the communication networks.

6. **Finance:** For storing information about holdings, sales, and purchases of financial instruments such as stocks and bonds.

7. **Sales:** For customer, product, and purchase information.

8. **Manufacturing:** For management of supply chain and for tracking production of items in factories, inventories of items in warehouses/stores, and orders for items.

9. **Human resources:** For information about employees, salaries, payroll taxes and benefits, and for generation of paychecks.

10. **Railway Reservation Systems:** For reservations and schedule information.

11. **Web:** For access the Back accounts and to get the balance amount.

12. **E –Commerce:** For Buying a book or music CD and browse for things like watches, mobiles from the Internet.

CHARACTERISTICS OF DATABASE:

The database approach has some very characteristic features which are discussed in detail below:

Structured and Described Data:

Fundamental feature of the database approach is that the database system does not only contain the data but also the complete definition and description of these data. These descriptions are basically details about the extent, the structure, the type and the format of all data and, additionally, the relationship between the data. This kind of stored data is called metadata ("data about data").

Separation of Data and Applications:

Application software does not need any knowledge about the physical data storage like encoding, format, storage place, etc. It only communicates with the management system of a database (DBMS) via a standardized interface with the help of a standardized language like SQL. The access to the data and the metadata is entirely done by the DBMS. In this way all the applications can be totally separated from the data.

Data Integrity:

Data integrity is a byword for the quality and the reliability of the data of a database system. In a broader sense data integrity includes also the protection of the database from unauthorized access (confidentiality) and unauthorized changes. Data reflect facts of the real world.

Transactions:

A transaction is a bundle of actions which are done within a database to bring it from one consistent state to a new consistent state. In between the data are inevitable inconsistent. *A* transaction is atomic what

means that it cannot be divided up any further. Within a transaction all or none of the actions need to be carried out. Doing only a part of the actions would lead to an inconsistent database state.

Example: One example of a transaction is the transfer of an amount of money from one bank account to another.

Data Persistence:

Data persistence means that in a DBMS all data is maintained as long as it is not deleted explicitly. The life span of data needs to be determined directly or indirectly be the user and must not be dependent on system features. Additionally data once stored in a database must not be lost. Changes of a database which are done by a transaction are persistent. When a transaction is finished even a system crash cannot put the data in danger

TYPES OF DATABASES:

Database can be classified according to the following factors. They are:

1. Number of Users
2. Database Location
3. Expected type
4. Extent of use

1. Based on number of Users:

According to the number of users the databases can be classified into following types. They are :

a). Single user b). Multiuser

Single user database:

- Single user database supports only one user at a time.
- Desktop or personal computer database is an example for single user database.

Multiuser database:

- Multi user database supports multiple users at the same time.
- Workgroup database and enterprise databases are examples for multiuser database.

Workgroup database:

If the multiuser database supports relatively small number of users (fewer than 50) within an organization is called as Workgroup database.

Enterprise database:

If the database is used by the entire organization and supports multiple users (more than 50) across many departments is called as Enterprise database.

2. Based on Location:

According to the location of database the databases can be classified into following types. They are:

a).CentralizedDatabase
b).Distributed Database

Centralized Database:

It is a database that is located, stored, and maintained in a single location. This location is most often a central computer or database system, for example a desktop or server CPU, or a mainframe computer. In most cases, a centralized database would be used by an organization (e.g. a business company) or an institution (e.g. a university.)

Distributed Database:

A distributed database is a database in which storage devices are not all attached to a common CPU. It may be stored in multiple computers located in the same physical location, or may be dispersed over a network of interconnected computers.

INTRODUCTION TO DATABASE-MANAGEMENT SYSTEM:

Database Management System:

- A **database-management system** (DBMS) is a collection of interrelated data and a set of programs to access those data.

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

Goals of DBMS:

The primary goal of a DBMS is to provide a way to store and retrieve database information that is both *convenient* and *efficient*

1. Manage large bodies of information

2. Provide convenient and efficient ways to store and access information

3. Secure information against system failure or tampering

4. Permit data to be shared among multiple users

Properties of DBMS:

1. A Database represents some aspect of the real world. Changes to the real world reflected in the database.

2. A Database is a logically coherent collection of data with some inherent meaning.

3. A Database is designed and populated with data for a specific purpose.

Need of DBMS:

1. Before the advent of DBMS, organizations typically stored information using a "File Processing Systems".

Example of such systems is File Handling in High Level Languages like C, Basic and COBOL etc., these systems have Major disadvantages to perform the Data Manipulation. So to overcome those drawbacks now we are using the DBMS.

2. Database systems are designed to manage large bodies of information.

3. In addition to that the database system must ensure the safety of the information stored, despite system crashes or attempts at unauthorized access. If data are to be shared among several users, the system must avoid possible anomalous results.

ADVANTAGES OF A DBMS OVER FILE SYSTEM:

Using a DBMS to manage data has many advantages:

Data Independence:

Application programs should be as independent as possible from details of data representation and storage. The DBMS can provide an abstract view of the data to insulate application code from such details.

Efficient Data Access:

A DBMS utilizes a variety of sophisticated techniques to store and retrieve data efficiently. This feature is especially important if the data is stored on external storage devices.

Data Integrity and Security:

If data is always accessed through the DBMS, the DBMS can enforce integrity constraints on the data. For example, before inserting salary information for an employee, the DBMS can check that the department budget is not exceeded. Also, the DBMS can enforce *access controls* that govern what data is visible to different classes of users.

Concurrent Access and Crash Recovery:

A database system allows several users to access the database concurrently. Answering different questions from different users with the same (base) data is a central aspect of an information system. Such concurrent use of data increases the economy of a system.

An example for concurrent use is the travel database of a bigger travel agency. The employees of different branches can access the database concurrently and book journeys for their clients. Each travel agent sees on his interface if there are still seats available for a specific journey or if it is already fully booked.

A DBMS also protects data from failures such as power failures and crashes etc. by the recovery schemes such as backup mechanisms and log files etc.

Data Administration:

When several users share the data, centralizing the administration of data can offer significant improvements. Experienced professionals, who understand the nature of the data being managed, and how different groups of users use it, can be responsible for organizing the data representation to minimize redundancy and fine-tuning the storage of the data to make retrieval efficient.

Reduced Application Development Time:

DBMS supports many important functions that are common to many applications accessing data stored in the DBMS. This, in conjunction with the high-level interface to the data, facilitates quick development of applications. Such applications are also likely to be more robust than applications developed from scratch because many important tasks are handled by the DBMS instead of being implemented by the application.

DISADVANTAGES OF DBMS:

Danger of a Overkill:

For small and simple applications for single users a database system is often not advisable.

Complexity:

A database system creates additional complexity and requirements. The supply and operation of a database management system with several users and databases is quite costly and demanding.

Qualified Personnel:

`The professional operation of a database system requires appropriately trained staff. Without a qualified database administrator nothing will work for long.

Costs:

Through the use of a database system new costs are generated for the system itself but also for additional hardware and the more complex handling of the system.

Lower Efficiency:

A database system is a multi-use software which is often less efficient than specialized software which is produced and optimized exactly for one problem.

DATABASE USERS & DATABASE ADMINISTRATORS:

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.

Naive users:

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

For example, a bank teller who needs to transfer $50 from account *A* to account *B* invokes a program called *transfer*. This program asks the teller for the amount of money to be transferred, the account from which the money is to be transferred, and the account to which the money is to be transferred.

Application programmers:

Application programmers are computer professionals who write application programs. Application programmers can choose from many tools to develop user interfaces. **Rapid application development (RAD)** tools are tools that enable an application programmer to construct forms and reports without writing a program.

Sophisticated users:

Sophisticated users interact with the system without writing programs. Instead, they form their requests in a database query language. They submit each such query to a **query processor**, whose function is to break down DML statements into instructions that the storage manager understands. Analysts who submit queries to explore data in the database fall in this category.

Specialized users:

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

Database Administrator:

One of the main reasons for using DBMSs is to have central control of both the data and the programs that access those data. A person who has such central control over the system is called a **database administrator** (**DBA**).

Database Administrator Functions/Roles:

The functions of a DBA include:

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 definition.

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 various 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:

1. Periodically backing up the database, either onto tapes or onto remote servers, to prevent loss of data in case of disasters such as flooding.

2. Ensuring that enough free disk space is available for normal operations, and upgrading disk space as required.

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

LEVELS OF ABSTRACTION IN A DBMS:

Hiding certain details of how the data are stored and maintained. A major purpose of database system is to provide users with an "Abstract View" of the data. In DBMS there are 3 levels of data abstraction. The goal of the abstraction in the DBMS is to separate the users request and the physical storage of data in the database.

Levels of Abstraction:

Physical Level:

- The lowest Level of Abstraction describes "How" the data are actually stored.
- The physical level describes complex low level data structures in detail.

Logical Level:

- This level of data Abstraction describes "What" data are to be stored in the database and what relationships exist among those data.
- Database Administrators use the logical level of abstraction.

View Level:

- It is the highest level of data Abstracts that describes only part of entire database.
- Different users require different types of data elements from each database.
- The system may provide many views for the some database.

THREE SCHEMA ARCHITECTURE:

Schema:

The overall design of the database is called the "Schema" or "Meta Data". A database schema corresponds to the programming language type definition. The value of a variable in programming language corresponds to an "Instance" of a database Schema.

Three Schema Architecture:

The goal of this architecture is to separate the user applications and the physical database. In this architecture, schemas can be defined at the following three levels:

1. The **internal level** has an **internal schema,** which describes the physical storage structure of the database. The internal schema uses a physical data model and describes the complete details of data storage and access paths for the database.

2. The **conceptual level** has a **conceptual schema,** which describes the structure of the whole database for a community of users. The conceptual schema hides the details of physical storage structures and concentrates on describing entities, data types, relationships, user operations, and constraints. A high-level data model or an implementation data model can be used at this level.

3. The **external** or **view level** includes a number of **external schemas** or **user views.** Each external schema describes the part of the database that a particular user group is interested in and hides the rest of the database from that user group. A high-level data model or an implementation data model can be used at this level.

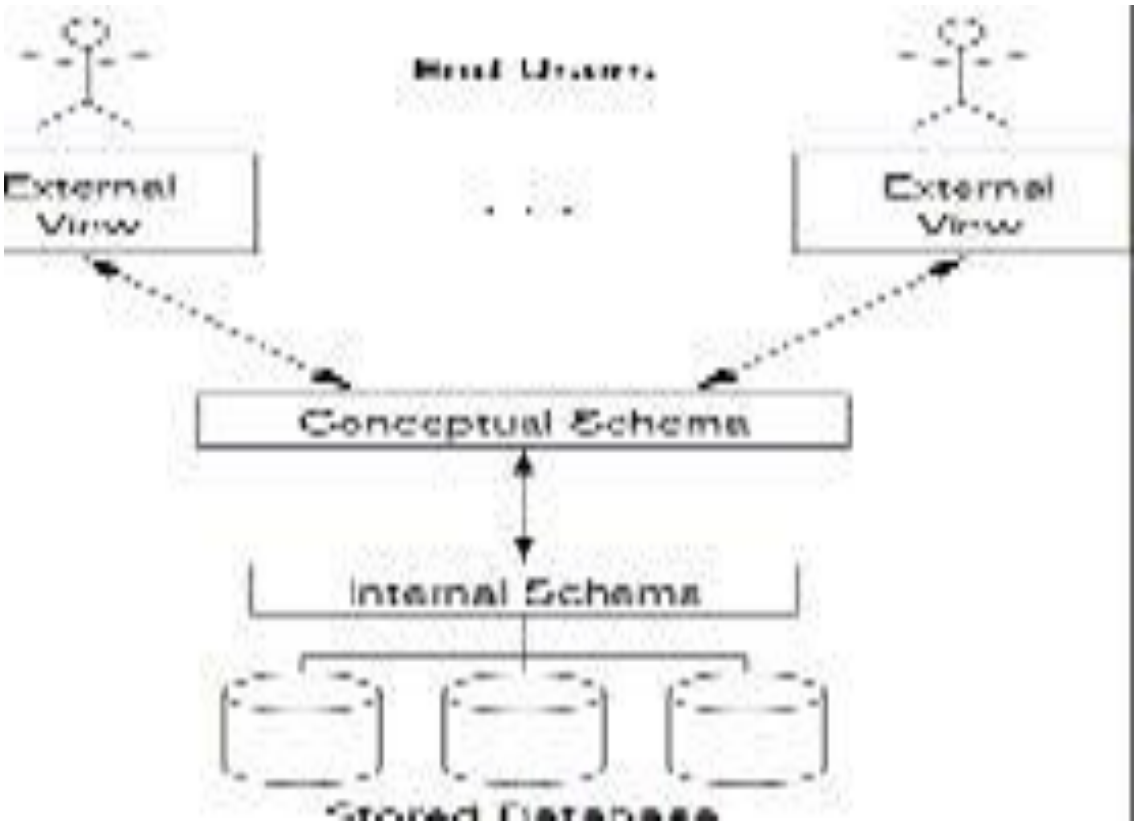


Fig: Three-Schema Architecture

DATA INDEPENDENCE:

- A very important advantage of using DBMS is that it offers Data Independence.
- The ability to modify a scheme definition in one level without affecting a scheme definition in a higher level is called **data independence**.
- There are two kinds:

1. Physical Data Independence
2. Logical Data Independence

Physical Data Independence:

- The ability to modify the physical schema without causing application programs to be rewritten

- Modifications at this level are usually to improve performance.

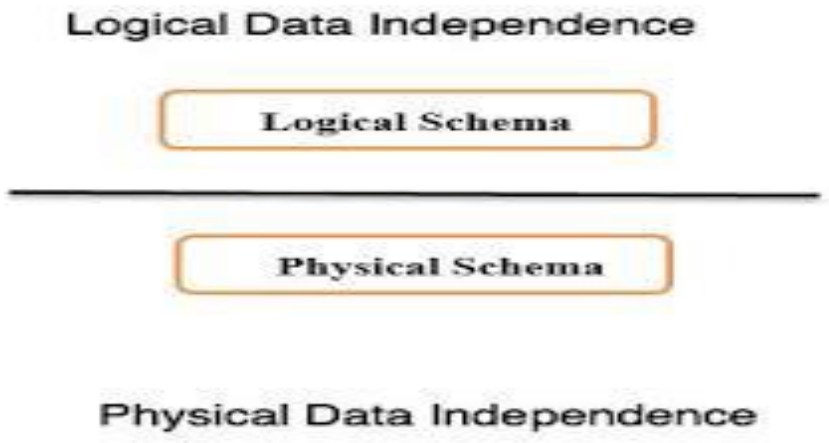


Fig: Data Independence

Logical Data Independence:

- The ability to modify the conceptual schema without causing application programs to be rewritten
- Usually done when logical structure of database is altered

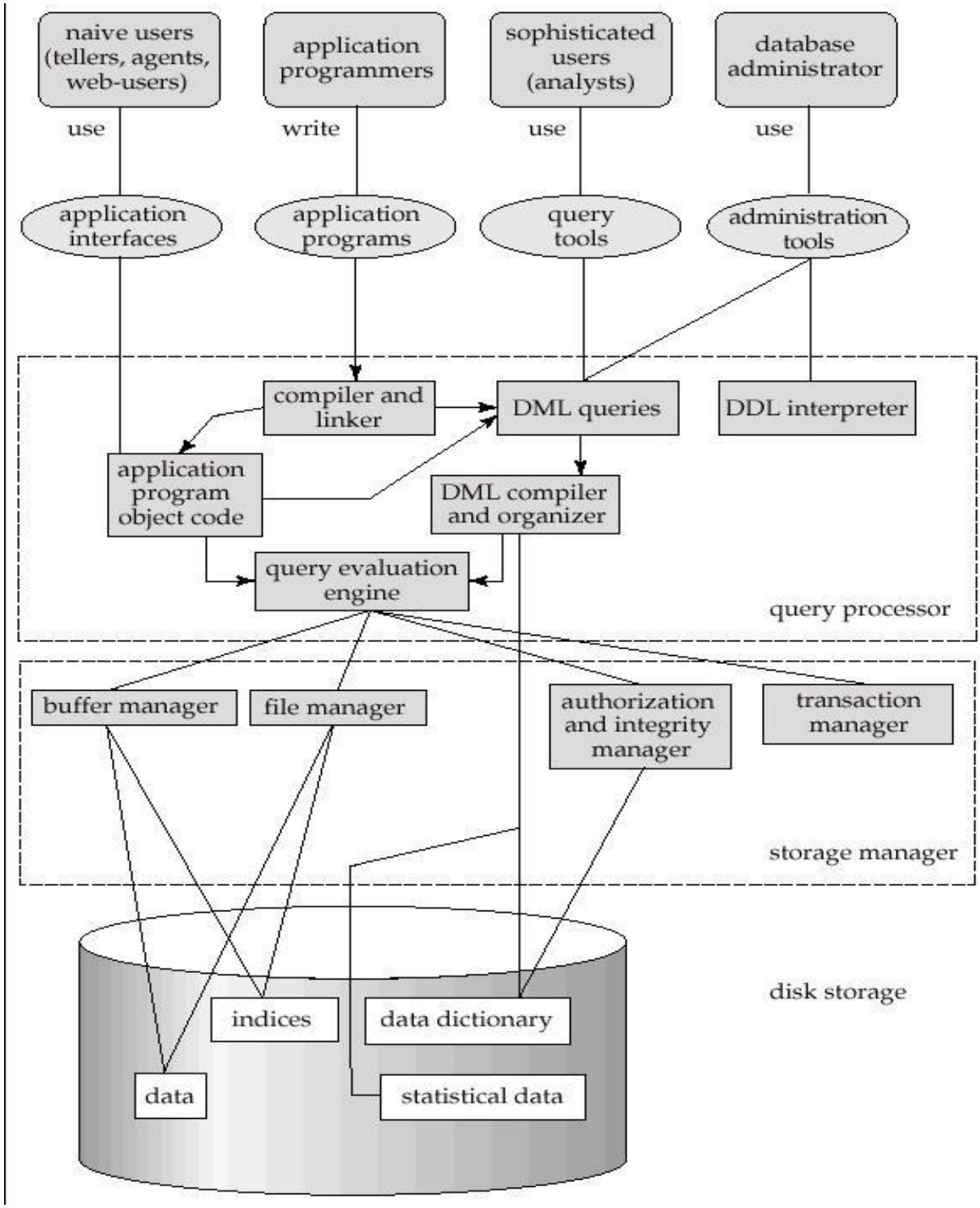
- Logical data independence is harder to achieve as the application programs are usually heavily dependent on the logical structure of the data.

DATABASE SYSTEM STRUCTURE:

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

The storage manager is important because databases typically require a large amount of storage space. Some Big organizations Database ranges from Giga bytes to Tera bytes. So the main memory of computers cannot store this much information, the information is stored on disks. Data are moved between disk storage and main memory as needed.

The query processor also very important because it helps the database system simplify and facilitate access to data. So quick processing of updates and queries is important. It is the job of the database system to translate updates and queries written in a nonprocedural language,


naive users (tellers, agents, web-users)
application programmers
sophisticated users (analysts)
database administrator
use
write
use
use
application interfaces
application programs
query tools
administration tools
compiler and linker
DML queries
DDL interpreter
application program object code
DML compiler and organizer
query evaluation engine
query processor
buffer manager
file manager
authorization and integrity manager
transaction manager
storage manager
disk storage
indices
data dictionary
data
statistical data

StorageManager:

A 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 storage manager is responsible for the interaction with the file manager. The storage manager translates the various DML statements into low-level file-system commands. Thus, the storage manager is responsible for storing, retrieving, and updating data in the database.

Storage Manager Components:

Authorization and integrity manager which tests for the satisfaction of integrity constraints and checks the authority of users to access data.

Transaction manager which ensures that the database itself remains in a consistent state despite system failures, and that concurrent transaction executions proceed without conflicting.

File manager: which manages the allocation of space on disk storage and the data structures used to represent information stored on disk.

Buffer manager which is responsible for fetching data from disk storage into main memory. Storage manager implements several data structures as part of the physical system implementation. Data files are used to store the database itself. Data dictionary is used to stores metadata about the structure of the database, in particular the schema of the database.

Query Processor Components:

DDL interpreter: It interprets DDL statements and records the definitions in the data dictionary.

DML compiler: It translates DML statements in a query language into an evaluation plan consisting of low-level instructions that the query evaluation engine understands.

Query evaluation engine: It executes low-level instructions generated by the DML compiler.

Application Architectures:

Most users of a database system today are not present at the site of the database system, but connect to it through a network. We can therefore differentiate between client machines, on

which remote database users' work, and server machines, on which the database system runs. Database applications are usually partitioned into two or three parts. They are:

1. Two – Tier Architecture

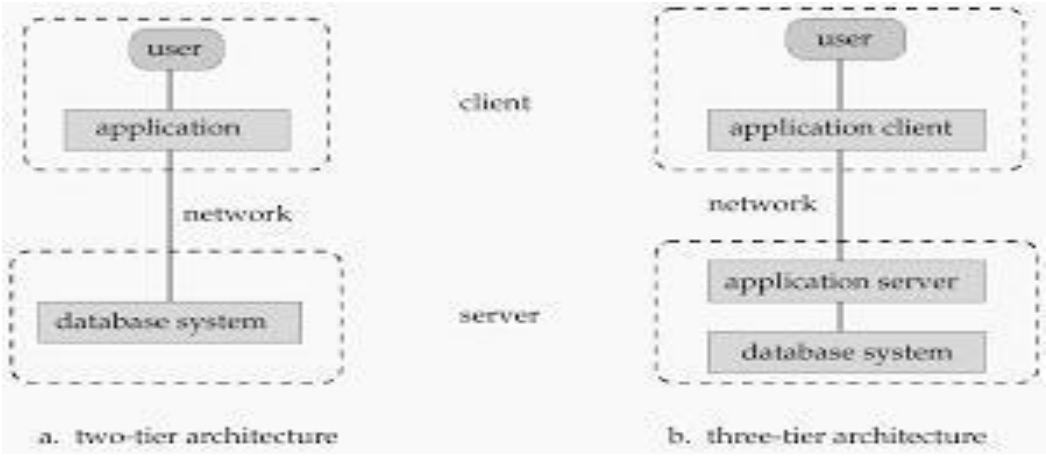
2. Three – Tier Architecture.

Two-Tier Architecture:

The application is partitioned into a component that resides at the client machine, which invokes database system functionality at the server machine through query language statements. Application program interface standards like ODBC and JDBC are used for interaction between the client and the server.

Three-Tier Architecture:

The client machine acts as merely a front end and does not contain any direct database calls. Instead, the client end communicates with an application server, usually through forms interface. The application server in turn communicates with a database system to access data. The business logic of the application, which says what actions to carry out under what conditions, is embedded in the application server, instead of being distributed across multiple clients. Three-tier applications are more appropriate for large applications, and for applications that run on the World Wide Web.



DATABASE DESIGN:

The database design process can be divided into six steps. The ER Model is most relevant to the first three steps. Next three steps are beyond the ER Model.

1. Requirements Analysis:

The very first step in designing a database application is to understand what data is to be stored in the database, what applications must be built on top of it, and what operations are most frequent and subject to performance requirements. The database designers collect information of the organization and analyzer, the information to identify the user's requirements. The database designers must find out what the users want from the database.

2. Conceptual Database Design:

Once the information is gathered in the requirements analysis step a conceptual database design is developed and is used to develop a high level description of the data to be stored in the database, along with the constraints that are known to hold over this data. This step is often carried out using the ER model, or a similar high-level data model.

3. Logical Database Design:

In this step convert the conceptual database design into a database schema (Logical Database Design) in the data model of the chosen DBMS. We will only consider **relational DBMSs**, and therefore, the task in the

logical design step is to convert an ER schema into a relational database schema. The result is a conceptual schema, sometimes called the **logical schema**, in the relational data model.

Beyond the ER Design:

The first three steps are more relevant to the ER Model. Once the logical scheme is defined designer consider the physical level implementation and finally provide certain security measures. The remaining three steps of database design are briefly described below:

4. Schema Refinement:

The fourth step in database design is to analyze the collection of relations in our relational database schema to identify potential problems, and to refine it. In contrast to the requirements analysis and conceptual design steps, which are essentially subjective, schema refinement can be guided by some elegant and powerful theory.

5. **Physical Database Design:**

In this step we must consider typical expected workloads that our database must support and further refine the database design to ensure that it meets desired performance

criteria. This step may simply involve building indexes on some tables and clustering some tables, or it may involve a substantial redesign of parts of the database schema obtained from the earlier design steps.

6. Security Design:

The last step of database design is to include security features. This is required to avoid unauthorized access to database practice after all the six steps. We required Tuning step in which all the steps are interleaved and repeated until the design is satisfactory.

DBMS FUNCTIONS:

- DBMS performs several important functions that guarantee the integrity and consistency of the data in the database.
- Those functions transparent to end users and can be accessed only through the use of DBMS. They include:
  - Data Dictionary Management
  - Data Storage Management
  - Data transformation and Presentation
  - Security Management
  - Multiple Access Control
  - Backup and Recovery Management
  - Data Integrity Management
  - Database Access Languages
  - Databases Communication Interfaces

Data Dictionary Management:

- DBMS stores definitions of database elements and their relationship (Metadata) in the data dictionary.
- The DBMS uses the data dictionary to look up the required data component structures and relationships.
- Any change made in database structure is automatically recorded in the data dictionary.

Data Storage Management:

- Modern DBMS provides storage not only for data but also for related data entities.
- Data Storage Management is also important for database “performance tuning”.
- Performance tuning related to activities that make database more efficiently.

Data Transformation and Presentation:

- DBMS transforms entered data to confirm to required data structures.
- DBMS formats the physically retrieved data to make it confirms to user's logical expectations.
- DBMS also presents the data in the user's expected format.

Security Management:

- DBMS creates a security system that enforces the user security and data privacy.
- Security rules determines which users can access the database, which data items each user can access etc.

- DBA and authenticated user logged to DBMS through username and password or through Biometric authentication such as Finger print and face reorganization etc.

Multiuser Access Control:

- To provide data integrity and data consistency, DBMS uses sophisticated algorithms to ensure that multiple users can access the database concurrently without compromising the integrity of database.

Backup and Recovery Management:

- DBMS provides backup and recovery to ensure data safety and integrity.

- Recovery management deals with the recovery of database after failure such as bad sector in the disk or power failure. Such capability is critical to preserve database integrity.

Data Integrity Management:

- DBMS provides and enforces integrity rules, thus minimizing data redundancy and maximizing data consistency.

- Ensuring data integrity is especially important in transaction- oriented database systems.

Database Access Languages:

- DBMS provides data access through a query language.

- A query language is a non-procedural language i.e. it lets the user specify what must be done without specifying how it is to be done.

- SQL is the default query language for data access.

Databases Communication Interfaces:

- Current DBMS's are accepting end-user requests via different network environments.

- For example, DBMS might provide access to database via Internet through the use of web browsers such as Mozilla Firefox or Microsoft Internet Explorer.

What is Schema?

A database schema is the skeleton structure that represents the logical view of the entire database. (or)

The logical structure of the database is called as Database Schema. (or)

The overall design of the database is the database schema.

- It defines how the data is organized and how the relations among them are associated.

- It formulates all the constraints that are to be applied on the data.

EG:

STUDENT

SID	SNAME	PHNO

What is Instance?

The actual content of the database at a particular point in time. (Or)

The data stored in the database at any given time is an instance of the database

Student

Sid	Name	phno
1201	Venkat	9014901442
1202	teja	9014774422

In the above table 1201, 1202, Venkat etc are said to be instance of student table.

Difference between File system & DBMS:

File system	DBMS
1. File system is a collection of data. Any management with the file system, user has to write the procedures	1. DBMS is a collection of data and user is not required to write the procedures for managing the database.
2. File system gives the details of the data representation and Storage of data.	2. DBMS provides an abstract view of data that hides the details.
3. In File system storing and retrieving of data cannot be done efficiently.	3. DBMS is efficient to use since there are wide varieties of sophisticated techniques to store and retrieve the data.
4. Concurrent access to the data in the file system has many problems like : Reading the file while other deleting some information, updating some information	4. DBMS takes care of Concurrent access using some form of locking.
5. File system doesn't provide crash recovery mechanism. Eg. While we are entering some data into the file if System crashes then content of the file is lost	5. DBMS has crash recovery mechanism, DBMS protects user from the effects of system failures.
6. Protecting a file under file system is very difficult.	6. DBMS has a good protection mechanism.

UNIT-II

Relational Algebra and Calculus

Relational Algebra:

Relational Algebra

- Basic operations:
  - *Selection* ( ) Selects a subset of rows from relation.
  - *Projection* ( ) Deletes unwanted columns from relation.
  - *Cross-product* ( ) Allows us to combine two relations.
  - *Set-difference* ( ) Tuples in reln. 1, but not in reln. 2.
  - *Union* ( ) Tuples ~~in~~ reln. 1 and in reln. 2.
- Additional operations:
  - Intersection, *join*, division, renaming: Not essential, but (very!) useful.
- Since each operation returns a relation, operations can be *composed*! (Algebra is "closed".)

Slide No:L6-4

- Basic operations:
  - *Selection* ( ) Selects a subset of rows from relation.
  - *Projection* ( ) Deletes unwanted columns from relation.
  - *Cross-product* ( ) Allows us to combine two relations.
  - *Set-difference* ( ) Tuples in reln. 1, but not in reln. 2.
  - *Union* ( ) Tuples in reln. 1 and in reln. 2.
- Additional operations:
  - Intersection, *join*, division, renaming: Not essential, but (very!) useful.
- Since each operation returns a relation, operations can be *composed*! (Algebra is "closed".)

Projection:

Projection

- Deletes attributes that are not in *projection list*.
- *Schema* of result contains exactly the fields in the projection list, with the same names that they had in the (only) input relation.
- Projection operator has to eliminate *duplicates*! (Why??)
  - Note: real systems typically don't do duplicate elimination unless the user explicitly asks for it. (Why not?)

sname	rating
yuppy	9
lubber	8
guppy	5
rusty	10

$$\pi_{sname,rating}(S2)$$

age
35.0
55.5

$$\pi_{age}(S2)$$

Slide No:L6-5

- Deletes attributes that are not in *projection list*.
- *Schema* of result contains exactly the fields in the projection list, with the same names that they had in the (only) input relation.
- Projection operator has to eliminate *duplicates*! (Why??)
  - Note: real systems typically don't do duplicate elimination unless the user explicitly asks for it. (Why not?)

Selection

Selection

- Selects rows that satisfy *selection condition*.
- No duplicates in result! (Why?)
- *Schema* of result identical to schema of (only) input relation.
- *Result* relation can be the *input* for another relational algebra operation! (*Operator composition.*)

sid	sname	rating	age
28	yuppy	9	35.0
58	rusty	10	35.0

$$\sigma_{rating>8}(S2)$$

sname	rating
yuppy	9
rusty	10

$$\pi_{sname,rating}(\sigma_{rating>8}(S2))$$

Slide No:L6-6

- Selects rows that satisfy *selection condition*.
- No duplicates in result! (Why?)
- *Schema* of result identical to schema of (only) input relation.
- *Result* relation can be the *input* for another relational algebra operation! (*Operator composition.*)

Set Operations:

Union, Intersection, Set-Difference

- All of these operations take two input relations, which must be *union-compatible*:
  - Same number of fields.
  - `Corresponding' fields have the same type.
- What is the *schema* of result?

Union, Intersection, Set-Difference

- All of these operations take two input relations, which must be *union-compatible*:
  - Same number of fields.
  - `Corresponding' fields have the same type.
- What is the *schema* of result?

sid	sname	rating	age
22	dustin	7	45.0
31	lubber	8	55.5
58	rusty	10	35.0
44	guppy	5	35.0
28	yuppy	9	35.0

$S1 \cup S2$

sid	sname	rating	age
31	lubber	8	55.5
58	rusty	10	35.0

$S1 \cap S2$

sid	sname	rating	age
22	dustin	7	45.0

$S1 - S2$

Slide No:L6-7

Cross-Product

- Each row of S1 is paired with each row of R1.
- *Result schema* has one field per field of S1 and R1, with field names `inherited' if possible.
  - *Conflict*: Both S1 and R1 have a field called *sid*.

Cross-Product

- Each row of S1 is paired with each row of R1.
- *Result schema* has one field per field of S1 and R1, with field names `inherited' if possible.
  - *Conflict*: Both S1 and R1 have a field called *sid*.

(sid)	sname	rating	age	(sid)	bid	day
22	dustin	7	45.0	22	101	10/10/96
22	dustin	7	45.0	58	103	11/12/96
31	lubber	8	55.5	22	101	10/10/96
31	lubber	8	55.5	58	103	11/12/96
58	rusty	10	35.0	22	101	10/10/96
58	rusty	10	35.0	58	103	11/12/96

- *Renaming operator*: $\rho\,(C(1 \to sid1, 5 \to sid2),\ S1 \times R1)$

Joins:

Joins

$$R \bowtie_c S = \sigma_c (R \times S)$$

- *Condition Join*:

(sid)	sname	rating	age	(sid)	bid	day
22	dustin	7	45.0	58	103	11/12/96
31	lubber	8	55.5	58	103	11/12/96

$$S1 \bowtie_{S1.sid < R1.sid} R1$$

- *Result schema* same as that of cross-product.
- Fewer tuples than cross-product, might be able to compute more efficiently
- Sometimes called a *theta-join*.

- *Condition Join*:
- *Result schema* same as that of cross-product.
- Fewer tuples than cross-product, might be able to compute more efficiently
- Sometimes called a *theta-join*.
- *Equi-Join*: A special case of condition join where the condition *c* contains only ***equalities.***
- *Result schema* similar to cross-product, but only one copy of fields for which equality is specified.
- *Natural Join*: Equijoin on *all* common fields.

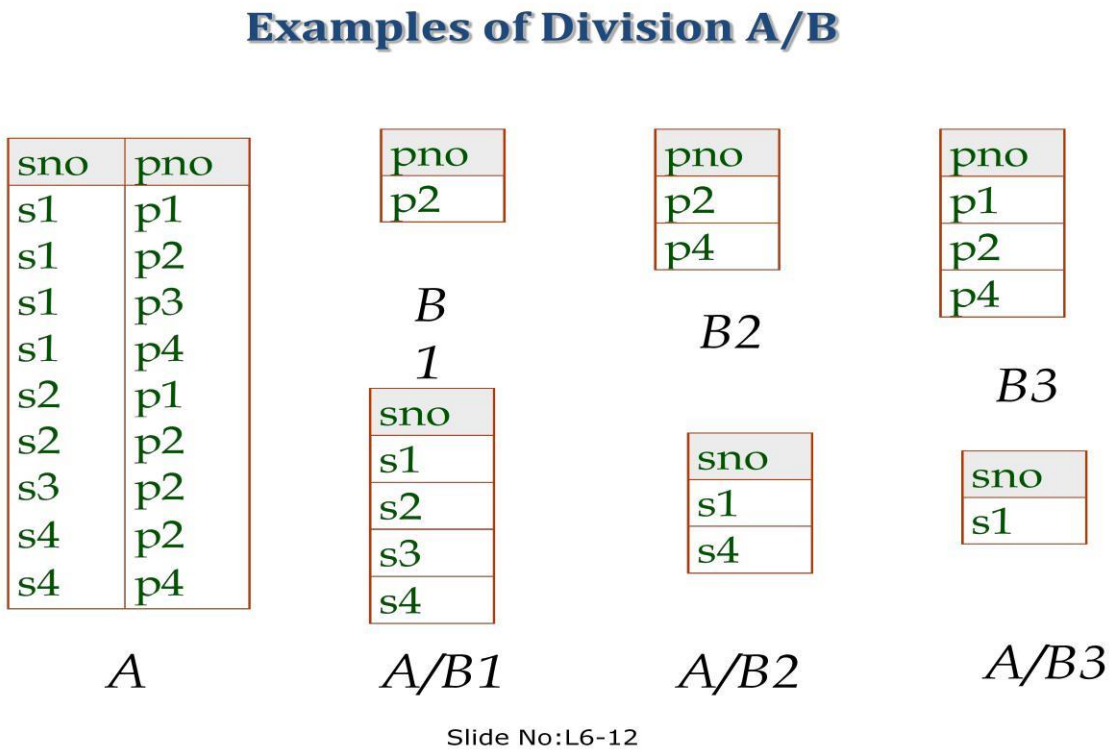
Division

- Not supported as a primitive operator, but useful for expressing queries like:

  *Find sailors who have reserved **all** boats.*

- Let *A* have 2 fields, *x* and *y*; *B* have only field *y*:
  - *A/B* = $\{\langle x \rangle \mid \exists \langle x, y \rangle \in A \ \ \forall \langle y \rangle \in B\}$
  - i.e., ***A/B* contains all *x* tuples (sailors) such that for *every* *y* tuple (boat) in *B*, there is an *xy* tuple in *A*.**
  - *Or*: If the set of *y* values (boats) associated with an *x* value (sailor) in *A* contains all *y* values in *B*, the *x* value is in *A/B*.
- In general, *x* and *y* can be any lists of fields; *y* is the list of fields in *B*, and *x* *y* is the list of fields of *A*.

Examples of Division A/B



Find names of sailors who've reserved boat #103

- Solution 1:
- Find names of sailors who've reserved a red boat
- Information about boat color only available in Boats; so need an extra join:

Find sailors who've reserved a red or a green boat

- Can identify all red or green boats, then find sailors who've reserved one of these boats:

Find sailors who've reserved a red <u>and</u> a green boat

- Previous approach won't work! Must identify sailors who've reserved red boats, sailors who've reserved green boats, then find the intersection (note that *sid* is a key for Sailors):

Relational Calculus:

- Comes in two flavors: *Tuple relational calculus* (TRC) and *Domain relational calculus* (DRC).
- Calculus has *variables, constants, comparison ops, logical connectives* and *quantifiers.*
  - *TRC*: Variables range over (i.e., get bound to) *tuples.*
  - *DRC*: Variables range over *domain elements* (= field values).
  - Both TRC and DRC are simple subsets of first-order logic.
- Expressions in the calculus are called *formulas.* An answer tuple is essentially an assignment of constants to variables that make the formula evaluate to *true*.

Domain Relational Calculus:

- *Query* has the form:

DRC Formulas

- *Atomic formula:*

–, or X *op* Y, or X *op* constant *op* is one of

- *Formula:*
  - an atomic formula, or
  - , where p and q are formulas, or
  - , where variable X is *free* in p(X), or
  - , where variable X is *free* in p(X)
- The use of quantifiers and is said to *bind* X.
  - A variable that is not bound is free.

Free and Bound Variables

- The use of quantifiers and in a formula is said to *bind* X.

– A variable that is not bound is free.

- Let us revisit the definition of a query:

Find all sailors with a rating above 7

- The condition ensures that the domain variables *I, N, T* and *A* are bound to fields of the same Sailors tuple.

- The term to the left of `|' (which should be read as *such that*) says that every tuple that satisfies *T*>7 is in the answer.

- Modify this query to answer:

– Find sailors who are older than 18 or have a rating under 9, and are called 'Joe'.

Find sailors rated > 7 who have reserved boat #103

- We have used as a shorthand for

- Note the use of to find a tuple in Reserves that `joins with' the Sailors tuple under consideration.

Find sailors rated > 7 who've reserved a red boat

- Observe how the parentheses control the scope of each quantifier's binding.

- This may look cumbersome, but with a good user interface, it is very intuitive. (MS Access, QBE)

Find sailors who've reserved all boats

- Find all sailors *I* such that for each 3-tuple either it is not a tuple in Boats or there is a tuple in Reserves showing that sailor *I* has reserved it.

Find sailors who've reserved all boats (again!)

- Simpler notation, same query. (Much clearer!)

- To find sailors who've reserved all red boats:

Expressive Power of Algebra and Calculus:

- It is possible to write syntactically correct calculus queries that have an infinite number of answers! Such queries are called *unsafe*.
  - e.g.,
- It is known that every query that can be expressed in relational algebra can be expressed as a safe query in DRC / TRC; the converse is also true.
- *Relational Completeness*: Query language (e.g., SQL) can express every query that is expressible in relational algebra/calculus.

The Form of a Basic SQL Queries:

History

- IBM Sequel language developed as part of System R project at the IBM San Jose Research Laboratory
- Renamed Structured Query Language (SQL)
- ANSI and ISO standard SQL:
  - SQL-86
  - SQL-89
  - SQL-92
  - SQL:1999 (language name became Y2K compliant!)
  - SQL:2003
- Commercial systems offer most, if not all, SQL-92 features, plus varying feature sets from later standards and special proprietary features.
  - Not all examples here may work on your particular system.
- Data Definition Language
- The schema for each relation, including attribute types.
- Integrity constraints
- Authorization information for each relation.
- Non-standard SQL extensions also allow specification of
  - The set of indices to be maintained for each relations.
  - The physical storage structure of each relation on disk.
- Create Table Construct
- An SQL relation is defined using the **create table** command:

create table r (A_1 D_1, A_2 D_2, ..., A_n D_n,

(integrity-constraint$_1$),

...,

(integrity-constraint$_k$))

– r is the name of the relation

– each A_i is an attribute name in the schema of relation r

– D_i is the data type of attribute A_i

Example:

create table *branch*

(*branch_name* **char**(15)**,**

branch_city **char(**30),

assets **integer**)

- Domain Types in SQL
- **char(n).** Fixed length character string, with user-specified length *n.*
- **varchar(n).** Variable length character strings, with user-specified maximum length *n.*
- **int.** Integer (a finite subset of the integers that is machine-dependent).
- **smallint.** Small integer (a machine-dependent subset of the integer domain type).
- **numeric(p,d).** Fixed point number, with user-specified precision of *p* digits, with *n* digits to the right of decimal point.
- **real, double precision.** Floating point and double-precision floating point numbers, with machine-dependent precision.
- **float(n).** Floating point number, with user-specified precision of at least *n* digits.
- More are covered in Chapter 4.
- Integrity Constraints on Tables
- **not null**

- **primary key** (A_1, ..., A_n)
- Basic Insertion and Deletion of Tuples
- Newly created table is empty
- Add a new tuple to *account*

  insert into *account* **values** ('A-9732', 'Perryridge', 1200)

– Insertion fails if any integrity constraint is violated

- Delete *all* tuples from *account*

  delete from *account*

  Note: Will see later how to delete selected tuples
- Drop and Alter Table Constructs
- The **drop table** command deletes all information about the dropped relation from the database.
- The **alter table** command is used to add attributes to an existing relation:

  alter table *r* **add** *A D*

  where *A* is the name of the attribute to be added to relation *r* and *D* is the domain of *A*.

– All tuples in the relation are assigned *null* as the value for the new attribute.

- The **alter table** command can also be used to drop attributes of a relation:

  alter table *r* **drop** *A*

  where *A* is the name of an attribute of relation *r*

– Dropping of attributes not supported by many databases

Basic Query Structure

- A typical SQL query has the form:
  select A_1, A_2, ..., A_n
  from r_1, r_2, ..., r_m
  where P
  - A_i represents an attribute
  - R_i represents a relation
  - P is a predicate.

- This query is equivalent to the relational algebra expression.
  The result of an SQL query is a relation.
- The select Clause
- The **select** clause list the attributes desired in the result of a query
  - corresponds to the projection operation of the relational algebra
- Example: find the names of all branches in the *loan* relation:
  select *branch_name*
  from *loan*
- In the relational algebra, the query would be:
  $\tilde{O}_{branch\_name}$ (*loan*)
- NOTE: SQL names are case insensitive (i.e., you may use upper- or lower-case letters.)
  - E.g. *Branch_Name* ≡ *BRANCH_NAME* ≡ *branch_name*
  - Some people use upper case wherever we use bold font.
- SQL allows duplicates in relations as well as in query results.
- To force the elimination of duplicates, insert the keyword **distinct** after select**.**

- Find the names of all branches in the *loan* relations, and remove duplicates
  select distinct *branch_name* **from** *loan*

- The keyword **all** specifies that duplicates not be removed.

  select all *branch_name* **from** *loan*

- The select Clause (Cont.)

- An asterisk in the select clause denotes "all attributes"
  select * **from** *loan*

- The **select** clause can contain arithmetic expressions involving the operation, +, –, *, and /, and operating on constants or attributes of tuples.

E.g.:

select *loan_number, branch_name, amount* * 100 **from** *loan*

- The where Clause

- The **where** clause specifies conditions that the result must satisfy

– Corresponds to the selection predicate of the relational algebra.

- To find all loan number for loans made at the Perryridge branch with loan amounts greater than $1200.

select *loan_number* **from** *loan* **where** *branch_name* = 'Perryridge' **and** *amount* > 1200

- Comparison results can be combined using the logical connectives **and, or,** and **not.**

- The from Clause

- The **from** clause lists the relations involved in the query

– Corresponds to the Cartesian product operation of the relational algebra.

- Find the Cartesian product *borrower X loan*

Select *from *borrower, loan*

- The Rename Operation

- SQL allows renaming relations and attributes using the **as** clause:
  old-name **as** *new-name*

- E.g. Find the name, loan number and loan amount of all customers; rename the column name *loan_number* as *loan_id.*

- Tuple Variables

- Tuple variables are defined in the **from** clause via the use of the **as** clause.

- Find the customer names and their loan numbers and amount for all customers having a loan at some branch.

11.Example Basic Sql Queries:

Example Instances

- We will use these instances of the Sailors and Reserves relations in our examples.
- If the key for the Reserves relation contained only the attributes *sid* and *bid*, how would the semantics differ?

R1

sid	bid	day
22	101	10/10/96
58	103	11/12/96

S1

sid	sname	rating	age
22	dustin	7	45.0
31	lubber	8	55.5
58	rusty	10	35.0

S2

sid	sname	rating	age
28	yuppy	9	35.0
31	lubber	8	55.5
44	guppy	5	35.0
58	rusty	10	35.0

Slide No:L2-1

- We will use these instances of the Sailors and Reserves relations in our examples.

- If the key for the Reserves relation contained only the attributes *sid* and *bid*, how would the semantics differ?

Basic SQL Query

```
SELECT    [DISTINCT] target-list
FROM      relation-list
WHERE     qualification
```

- *relation-list* A list of relation names (possibly with a *range-variable* after each name).
- *target-list* A list of attributes of relations in *relation-list*
- *qualification* Comparisons (Attr *op* const or Attr1 *op* Attr2, where *op* is one of) combined using AND, OR and NOT.
- DISTINCT is an optional keyword indicating that the answer should not contain duplicates. Default is that duplicates are *not* eliminated!

Conceptual Evaluation Strategy

- Semantics of an SQL query defined in terms of the following conceptual evaluation strategy:
  - Compute the cross-product of *relation-list*.
  - Discard resulting tuples if they fail *qualifications*.
  - Delete attributes that are not in *target-list*.
  - If DISTINCT is specified, eliminate duplicate rows.
- This strategy is probably the least efficient way to compute a query! An optimizer will find more efficient strategies to compute *the same answers*.

Example of Conceptual Evaluation

```
SELECT S.sname
FROM   Sailors S, Reserves R
WHERE  S.sid=R.sid AND R.bid=103
```

(sid)	sname	rating	age	(sid)	bid	day
22	dustin	7	45.0	22	101	10/10/96
22	dustin	7	45.0	58	103	11/12/96
31	lubber	8	55.5	22	101	10/10/96
31	lubber	8	55.5	58	103	11/12/96
58	rusty	10	35.0	22	101	10/10/96
58	rusty	10	35.0	58	103	11/12/96

A Note on Range Variables

- Really needed only if the same relation appears twice in the FROM clause. The previous query can also be written as:

```
SELECT S.sname
FROM   Sailors S, Reserves R
WHERE  S.sid=R.sid AND bid=103
```

```
SELECT sname
FROM   Sailors, Reserves
WHERE  Sailors.sid=Reserves.sid
       AND bid=103
```

- Find sailors who've reserved at least one boat

ould adding DISTINCT to this query make a difference?

- What is the effect of replacing *S.sid* by *S.sname* in the SELECT clause? Would adding DISTINCT to this variant of the query make a difference?

Expressions and Strings

- Illustrates use of arithmetic expressions and string pattern matching: *Find triples (of ages of sailors and two fields defined by expressions) for sailors whose names begin and end with B and contain at least three characters.*

- AS and = are two ways to name fields in result.

- LIKE is used for string matching. `_' stands for any one character and `%' stands for 0

or more arbitrary characters.

String Operations

- SQL includes a string-matching operator for comparisons on character strings. The

operator "like" uses patterns that are described using two special characters:

  – percent (%). The % character matches any substring.

  – underscore (_). The _ character matches any character.

- Find the names of all customers whose street includes the substring "Main".

  select *customer_name*

  from *customer*

  where *customer_street* **like** '% Main%'

- Match the name "Main%" **like** 'Main\%' **escape** '\'

- SQL supports a variety of string operations such as

– concatenation (using "||")

– converting from upper to lower case (and vice versa)

– finding string length, extracting substrings, etc.

Ordering the Display of Tuples

• List in alphabetic order the names of all customers having a loan in Perryridge branch

select distinct *customer_name*

from *borrower, loan*

where *borrower loan_number = loan.loan_number* **and**
branch_name = 'Perryridge'

order by *customer_name*

• We may specify **desc** for descending order or **asc** for ascending order, for each attribute; ascending order is the default.

–Example: **order by** *customer_name* **desc**

Duplicates

• In relations with duplicates, SQL can define how many copies of tuples appear in the result.

• **Multiset** versions of some of the relational algebra operators – given multiset relations r_1 and r_2:

1. $\sigma_\theta (r_1)$**:** If there are c_1 copies of tuple t_1 in r_1, and t_1 satisfies selections σ_θ, then there are c_1 copies of t_1 in $\sigma_\theta (r_1)$.

2. $\Pi_A (r)$**:** For each copy of tuple t_1 in r_1, there is a copy of tuple $\Pi_A (t_1)$ in $\Pi_A (r_1)$ where $\Pi_A (t_1)$ denotes the projection of the single tuple t_1.

3. $r_1 \times r_2$ **:** 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 are $c_1 \times c_2$ copies of the tuple $t_1 . t_2$ in $r_1 \times r_2$

• Example: Suppose multiset relations r_1 (A, B) and r_2 (C) are as follows:

$r_1 = \{(1, a) (2, a)\}$ $r_2 = \{(2), (3), (3)\}$

- Then $\Pi_B(r_1)$ would be {(a), (a)}, while $\Pi_B(r_1)$ x r_2 would be {(*a*,2), (*a*,2), (*a*,3), (*a*,3), (*a*,3), (*a*,3)}
- SQL duplicate semantics:

select A_1,, A_2, ..., A_n

from r_1, r_2, ..., r_m

where P

is equivalent to the *multiset* version of the expression:

Nested Queries:

- A very powerful feature of SQL: a WHERE clause can itself contain an SQL query! (Actually, so can FROM and HAVING clauses.)

```
SELECT S.sname
FROM Sailors S
WHERE S.sid IN (SELECT R.sid
                FROM Reserves R
                WHERE R.bid=103)
```

- To find sailors who've *not* reserved #103, use NOT IN.

- To understand semantics of nested queries, think of a *nested loops* evaluation: *For each Sailors tuple, check the qualification by computing the subquery.*

Correlated Nested Queries:

Nested Queries with Correlation

```
SELECT S.sname
FROM Sailors S
WHERE EXISTS (SELECT *
              FROM Reserves R
              WHERE R.bid=103 AND S.sid=R.sid)
```

- EXISTS is another set comparison operator, like IN.
- If UNIQUE is used, and * is replaced by *R.bid*, finds sailors with at most one reservation for boat #103. (UNIQUE checks for duplicate tuples; * denotes all attributes. Why do we have to replace * by *R.bid*?)
- Illustrates why, in general, subquery must be re-computed for each Sailors tuple.

Set comparison Operators:

Nested Sub queries

- SQL provides a mechanism for the nesting of subqueries.
- A **subquery** is a **select-from-where** expression that is nested within another query.
- A common use of subqueries is to perform tests for set membership, set comparisons, and set cardinality.
- The set operations **union, intersect,** and **except** operate on relations and correspond to the relational algebra operations $\cup$, $\cap$, $-$.
- Each of the above operations automatically eliminates duplicates; to retain all duplicates use the corresponding multiset versions **union all, intersect all** and **except all.**

Suppose a tuple occurs *m* times in *r* and *n* times in *s*, then, it occurs:

- *m* + *n* times in *r* **union all** *s*
- min(*m*,*n)* times in *r* **intersect all** *s*
- max(0, *m* – *n)* times in *r* **except all** *s*

- Find all customers who have a loan, an account, or both:

Find sid's of sailors who've reserved a red and a green boat

- INTERSECT: Can be used to compute the intersection of any two *union-compatible* sets of tuples.
- Included in the SQL/92 standard, but some systems don't support it.
- Contrast symmetry of the UNION and INTERSECT queries with how much the other versions differ.

```
SELECT  S.sid
FROM  Sailors S, Boats B1, Reserves R1,
      Boats B2, Reserves R2
WHERE  S.sid=R1.sid AND R1.bid=B1.bid
 AND  S.sid=R2.sid AND R2.bid=B2.bid
AND (B1.color='red' AND B2.color='green')
```

Key field!

```
SELECT  S.sid
FROM  Sailors S, Boats B, Reserves R
WHERE  S.sid=R.sid AND R.bid=B.bid
       AND B.color='red'
INTERSECT
SELECT  S.sid
FROM  Sailors S, Boats B, Reserves R
WHERE  S.sid=R.sid AND R.bid=B.bid
       AND B.color='green'
```

Slide No:L2-15

More on Set-Comparison Operators

- We've already seen IN, EXISTS and UNIQUE. Can also use NOT IN, NOT EXISTS and NOT UNIQUE.
- Also available: *op* ANY, *op* ALL, *op* IN
- Find sailors whose rating is greater than that of some sailor called Horatio:

Rewriting INTERSECT Queries Using IN

- Similarly, EXCEPT queries re-written using NOT IN.
- To find *names* (not *sid*'s) of Sailors who've reserved both red and green boats, just replace *S.sid* by *S.sname* in SELECT clause. (What about INTERSECT query?)

Division in SQL

(1)

Division in SQL

```
SELECT S.sname
FROM Sailors S
WHERE NOT EXISTS
      ((SELECT B.bid
        FROM Boats B)
       EXCEPT
       (SELECT R.bid
        FROM Reserves R
        WHERE R.sid=S.sid))
```

Find sailors who've reserved all boats.

- Let's do it the hard way, without EXCEPT:

(2)
```
SELECT S.sname
FROM Sailors S
WHERE NOT EXISTS (SELECT B.bid
                  FROM Boats B
                  WHERE NOT EXISTS (SELECT R.bid
                                    FROM Reserves R
                                    WHERE R.bid=B.bid
                                    AND R.sid=S.sid))
```

Sailors S such that ...

there is no boat B without ...

a Reserves tuple showing S reserved B

Slide No:L5-5

Aggregate Operators:

- These functions operate on the multiset of values of a column of a relation, and return a value

avg: average value

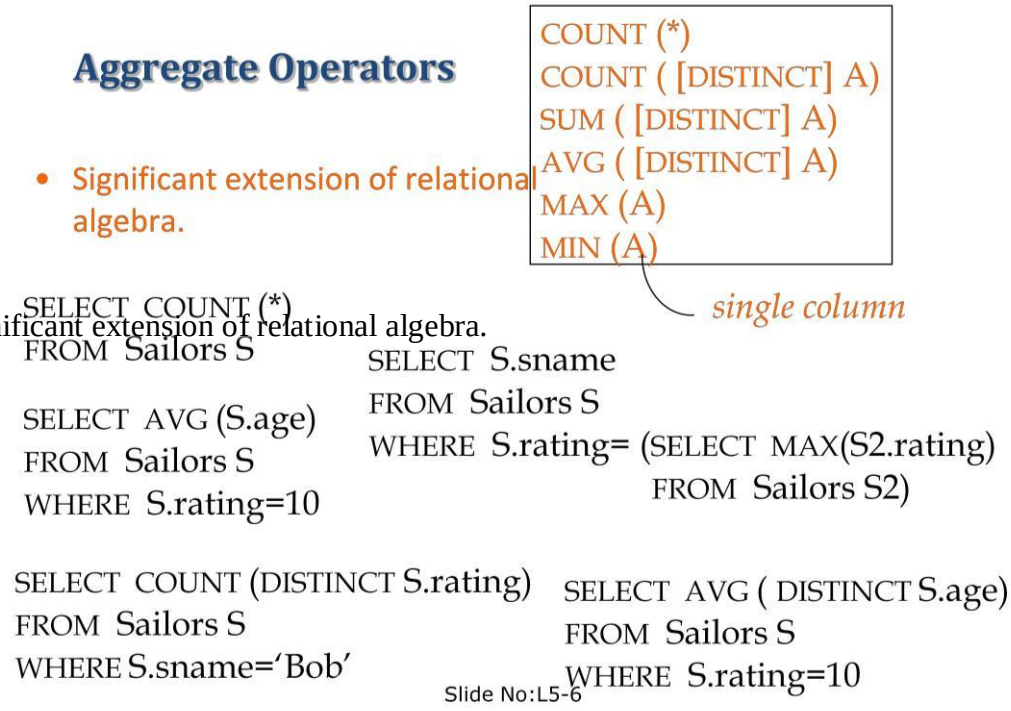
min: minimum value

max: maximum value

sum: sum of values

count: number of values

Aggregate Operators examples



- Significant extension of relational algebra.

Motivation for Grouping

- So far, we've applied aggregate operators to all (qualifying) tuples. Sometimes, we want to apply them to each of several *groups* of tuples.
- Consider: *Find the age of the youngest sailor for each rating level.*
  - In general, we don't know how many rating levels exist, and what the rating values for these levels are!
  - Suppose we know that rating values go from 1 to 10; we can write 10 queries that look like this (!):

Queries With GROUP BY and HAVING

- The *target-list* contains (i) attribute names (ii) terms with aggregate operations (e.g., MIN (*S.age*)).

Find name and age of the oldest sailor(s)

- The first query is illegal! (We'll look into the reason a bit later, when we discuss GROUP BY.)
- The third query is equivalent to the second query, and is allowed in the SQL/92 standard, but is not supported in some systems.

```
SELECT S.sname, MAX (S.age)
FROM Sailors S
```

```
SELECT S.sname, S.age
FROM Sailors S
WHERE S.age =
      (SELECT MAX (S2.age)
       FROM Sailors S2)
```

```
SELECT S.sname, S.age
FROM Sailors S
WHERE (SELECT MAX (S2.age)
       FROM Sailors S2)
       = S.age
```



```
SELECT      [DISTINCT] target-list
FROM        relation-list
WHERE       qualification
GROUP BY    grouping-list
HAVING      group-qualification
```

The attribute list (i) must be a subset of *grouping-list*. Intuitively, each answer tuple corresponds to a *group*, and these attributes must have a single value per group. (A *group* is a set of tuples that have the same value for all attributes in *grouping-list*.)

Conceptual Evaluation

- The cross-product of *relation-list* is computed, tuples that fail *qualification* are discarded, `*unnecessary*' fields are deleted, and the remaining tuples are partitioned into groups by the value of attributes in *grouping-list*.
- The *group-qualification* is then applied to eliminate some groups. Expressions in *group-qualification* must have a *single value per group*!
  - In effect, an attribute in *group-qualification* that is not an argument of an aggregate op also appears in *grouping-list*. (SQL does not exploit primary key semantics here!)
- One answer tuple is generated per qualifying group.

Find age of the youngest sailor with age 18, for each rating with at least 2 such sailors

Find age of the youngest sailor with age > 18, for each rating with at least 2 such sailors

```
SELECT S.rating, MIN (S.age)
                 AS minage
FROM Sailors S
WHERE S.age >= 18
GROUP BY S.rating
HAVING COUNT (*) > 1
```

Answer relation:

rating	minage
3	25.5
7	35.0
8	25.5

Sailors instance:

sid	sname	rating	age
22	dustin	7	45.0
29	brutus	1	33.0
31	lubber	8	55.5
32	andy	8	25.5
58	rusty	10	35.0
64	horatio	7	35.0
71	zorba	10	16.0
74	horatio	9	35.0
85	art	3	25.5
95	bob	3	63.5
96	frodo	3	25.5

- Find age of the youngest sailor with age 18, for each rating with at least 2 such sailors and with every sailor under 60.
- Find age of the youngest sailor with age 18, for each rating with at least 2 sailors between 18 and 60.

For each red boat, find the number of reservations for this boat Grouping over a join of three relations.

- What do we get if we remove *B.color='red'* from the WHERE clause and add a HAVING clause with this condition?
- What if we drop Sailors and the condition involving S.sid?
- Find age of the youngest sailor with age > 18, for each rating with at least 2 sailors (of any age)
- Shows HAVING clause can also contain a subquery.
- Compare this with the query where we considered only ratings with 2 sailors over 18!

- What if HAVING clause is replaced by:

– HAVING COUNT(*) >1

- Find those ratings for which the average age is the minimum over all ratings
- Aggregate operations cannot be nested! WRONG:
- Find the average account balance at the Perryridge branch.

Aggregate Functions – Group By

- Find the number of depositors for each branch.

```
select branch_name, count (distinct customer_name)
        from depositor, account
        where depositor.account_number = account.account_number
        group by branch_name
```

Note: Attributes in **select** clause outside of aggregate functions must appear in **group by** list

Aggregate Functions – Having Clause

- Find the names of all branches where the average account balance is more than $1,200.

```
select branch_name, avg (balance)
        from account
        group by branch_name
        having avg (balance) > 1200
```

Note: predicates in the **having** clause are applied after the formation of groups whereas predicates in the **where** clause are applied before forming groups

Null Values:

- Field values in a tuple are sometimes *unknown* (e.g., a rating has not been assigned) or *inapplicable* (e.g., no spouse's name).
  - SQL provides a special value *null* for such situations.
- The presence of *null* complicates many issues. E.g.:
  - Special operators needed to check if value is/is not *null*.
  - Is *rating>8* true or false when *rating* is equal to *null*? What about AND, OR and NOT connectives?
  - We need a 3-valued logic (true, false and *unknown*).
  - Meaning of constructs must be defined carefully. (e.g., WHERE clause eliminates rows that don't evaluate to true.)
  - New operators (in particular, *outer joins*) possible/needed.

Comparision Using Null Values:

- It is possible for tuples to have a null value, denoted by *null*, for some of their attributes
- *null* signifies an unknown value or that a value does not exist.
- The predicate **is null** can be used to check for null values.
  - Example: Find all loan number which appear in the *loan* relation with null values for *amount.*

select *loan_number*

from *loan*

where *amount* **is null**

- The result of any arithmetic expression involving *null* is *null*
  - Example: 5 + *null* returns null

- However, aggregate functions simply ignore nulls
  - More on next slide
- Null Values and Three Valued Logic
- Any comparison with *null* returns *unknown*
  - Example*: 5 < null or null <> null or null = null*

Logical Connectives:AND,OR,NOT

- Three-valued logic using the truth value *unknown*:
  - OR: (*unknown* **or** *true*) = *true,*

    (*unknown* **or** *false*) = *unknown*

    (*unknown* **or** *unknown) = unknown*
  - AND: *(true* **and** *unknown) = unknown,*

    (false **and** *unknown) = false,*

    (unknown **and** *unknown) = unknown*
  - NOT*: (***not** *unknown) = unknown*
  - "*P* **is unknown**" evaluates to true if predicate *P* evaluates to *unknown*
- Result of **where** clause predicate is treated as *false* if it evaluates to *unknown*
- Null Values and Aggregates
- Total all loan amounts

  select sum (*amount*)

from *loan*

  - Above statement ignores null amounts
  - Result is *null* if there is no non-null amount
- All aggregate operations except **count(*)** ignore tuples with null values on the aggregated attributes.

Impact on SQL Constructs:

"In" Construct

"In" Construct

- Find all customers who have both an account and a loan at the bank.

```
select distinct customer_name
        from borrower
        where customer_name in (select customer_name
                                from depositor )
```

- Find all customers who have a loan at the bank but do not have an account at the bank

```
select distinct customer_name
        from borrower
        where customer_name not in (select customer_name
                                    from depositor )
```

Slide No:L3-8

Example Query

- Find all customers who have both an account and a loan at the Perryridge branch

```
select distinct customer_name
        from borrower, loan
        where borrower.loan_number = loan.loan_number and
          branch_name = 'Perryridge'  and
          (branch_name, customer_name ) in
                (select branch_name, customer_name
                  from depositor, account
                  where depositor.account_number =
                      account.account_number )
```

- Note: Above query can be written in a much simpler manner. The formulation above is simply to illustrate SQL features.

"Some" Construct

"Some" Construct

- Find all branches that have greater assets than some branch located in Brooklyn.

```
select distinct  T.branch_name
        from branch as T, branch as S
        where  T.assets > S.assets and
                S.branch_city = 'Brooklyn'
```

- Same query using > **some** clause

```
select branch_name
        from branch
        where assets > some
                (select assets
                 from branch
                 where branch_city = 'Brooklyn')
```

"All" Construct

- Find the names of all branches that have greater assets than all branches located in Brooklyn.

```
select branch_name
        from branch
        where assets > all
                (select assets
                from branch
                where branch_city = 'Brooklyn')
```

"Exists" Construct

- Find all customers who have an account at all branches located in Brooklyn.
- Absence of Duplicate Tuples
- The **unique** construct tests whether a subquery has any duplicate tuples in its result.
- Find all customers who have at most one account at the Perryridge branch.

```
select distinct S.customer_name
        from depositor as S
        where not exists (
                (select branch_name
                from branch
                where branch_city = 'Brooklyn')
                except
                (select R.branch_name
                from depositor as T, account as R
                where T.account_number = R.account_number and
                        S.customer_name = T.customer_name ))
```

```
select T.customer_name
  from depositor as T
  where unique (
    select R.customer_name
from account, depositor as R
where T.customer_name = R.customer_name and
      R.account_number = account.account_number and
      account.branch_name = 'Perryridge')
```

Example Query

- Find all customers who have at least two accounts at the Perryridge branch.

```
select distinct T.customer_name
from depositor as T
where not unique (
    select R.customer_name
    from account, depositor as R
    where T.customer_name = R.customer_name and
          R.account_number = account.account_number and
          account.branch_name = 'Perryridge')
```

Modification of the Database – Deletion

- Delete all account tuples at the Perryridge branch

delete from *account*

where *branch_name* = 'Perryridge'

- Delete all accounts at every branch located in the city ‘Needham’.

delete from *account*

where *branch_name* **in** (**select** *branch_name*

from *branch*

where *branch_city* = 'Needham')

- Example Query
- Delete the record of all accounts with balances below the average at the bank.

Modification of the Database – Insertion

- Add a new tuple to *account*

  insert into *account*

values ('A-9732', 'Perryridge', 1200) or equivalently

insert into *account* (*branch_name, balance, account_number*)

values ('Perryridge', 1200, 'A-9732')

- Add a new tuple to *account* with *balance* set to null

  insert into *account* **values** ('A-777','Perryridge', *null*)

Modification of the Database – Insertion

- Provide as a gift for all loan customers of the Perryridge branch, a $200 savings account. Let the loan number serve as the account number for the new savings account

  insert into *account*

select *loan_number, branch_name,* 200

from *loan*

where *branch_name* = 'Perryridge'

insert into *depositor*

select *customer_name, loan_number*

from *loan, borrower*

where branch_name = 'Perryridge'

and *loan.account_number = borrower.account_number*

- The **select from where** statement is evaluated fully before any of its results are inserted into the relation

–Motivation: **insert into** *table*1 **select** * **from** *table*1

Modification of the Database – Updates

- Increase all accounts with balances over $10,000 by 6%, all other accounts receive 5%.

– Write two **update** statements:

update *account*

set *balance* = *balance* ∗ 1.06

where *balance* > 10000

update *account*

set *balance* = *balance* ∗ 1.05

where *balance* ≤ 10000

– The order is important

– Can be done better using the **case** statement (next slide)

Case Statement for Conditional Updates

- Same query as before: Increase all accounts with balances over $10,000 by 6%, all other accounts receive 5%.

update *account*

set *balance* = **case**

when *balance* <= 10000 **then** *balance* *1.05

else *balance* * 1.06

end

Outer Joins:

Joined Relations**

- **Join operations** take two relations and return as a result another relation.
- These additional operations are typically used as subquery expressions in the **from** clause

Join types
inner join
left outer join
right outer join
full outer join

Join Conditions
natural
on <predicate>
using $(A_1, A_1, \ldots, A_n)$

- **Join condition** – defines which tuples in the two relations match, and what attributes are present in the result of the join.
- **Join type** – defines how tuples in each relation that do not match any tuple in the other relation (based on the join condition) are treated.
- Joined Relations – Datasets for Examples

Relation *loan*

loan_number	*branch_name*	*amount*
L-170	Downtown	3000
L-230	Redwood	4000
L-260	Perryridge	1700

loan

customer_name	*loan_number*
Jones	L-170
Smith	L-230
Hayes	L-155

borrower

- Joined Relations – Examples

loan_number	*branch_name*	*amount*	*customer_name*	*loan_number*
L-170	Downtown	3000	Jones	L-170
L-230	Redwood	4000	Smith	L-230

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

loan_number	*branch_name*	*amount*	*customer_name*	*loan_number*
L-170	Downtown	3000	Jones	L-170
L-230	Redwood	4000	Smith	L-230
L-260	Perryridge	1700	*null*	*null*

Joined Relations – Examples

loan **natural inner join** *borrower*

loan_number	*branch_name*	*amount*	*customer_name*
L-170	Downtown	3000	Jones
L-230	Redwood	4000	Smith

loan_number	*branch_name*	*amount*	*customer_name*
L-170	Downtown	3000	Jones
L-230	Redwood	4000	Smith
L-155	*null*	*null*	Hayes

Joined Relations – Examples

loan_number	*branch_name*	*amount*	*customer_name*
L-170	Downtown	3000	Jones
L-230	Redwood	4000	Smith
L-260	Perryridge	1700	*null*
L-155	*null*	*null*	Hayes

- Natural join can get into trouble if two relations have an attribute with same name that should not affect the join condition
  - e.g. an attribute such as *remarks* may be present in many tables
- *Solution:*
  - *loan* **full outer join** *borrower* **using** (*loan_number*)
- Derived Relations
- SQL allows a subquery expression to be used in the **from** clause
- Find the average account balance of those branches where the average account balance is greater than $1200.

select *branch_name, avg_balance*

from (**select** *branch_name,* **avg** (*balance*)

from *account*

group by *branch_name*) **as** *branch_avg* (*branch_name, avg_balance*)

where *avg_balance* > 1200

Note that we do not need to use the **having** clause, since we compute the temporary (view) relation *branch_avg* in the **from** clause, and the attributes of *branch_avg* can be used directly in the **where** clause.

Complex Integrity Constraints in SQL:

- Integrity Constraints (Review)
- An IC describes conditions that every *legal instance* of a relation must satisfy.
  - Inserts/deletes/updates that violate IC's are disallowed.
  - Can be used to ensure application semantics (e.g., *sid* is a key), or prevent inconsistencies (e.g., *sname* has to be a string, *age* must be < 200)
- *Types of IC's*: Domain constraints, primary key constraints, foreign key constraints, general constraints.
  - *Domain constraints*: Field values must be of right type. Always enforced.

General Constraints

```
CREATE TABLE Reserves
        ( sname CHAR(10),
        bid INTEGER,
        day DATE,
        PRIMARY KEY (bid,day),
        CONSTRAINT noInterlakeRes
        CHECK (`Interlake' <>
                ( SELECT B.bname
                FROM Boats B
                WHERE B.bid=bid)))
```

- Useful when more general ICs than keys are involved.
- Can use queries to express constraint.
- Constraints can be named.

Constraints Over Multiple Relations

- Awkward and wrong!
- If Sailors is empty, the number of Boats tuples can be anything!
- ASSERTION is the right solution; not associated with either table.

Number of boats plus number of sailors is < 100

```
CREATE TABLE Sailors
( sid INTEGER,
sname CHAR(10),
rating INTEGER,
age REAL,
PRIMARY KEY (sid),
CHECK
( (SELECT COUNT (S.sid) FROM Sailors S)
+ (SELECT COUNT (B.bid) FROM Boats B) < 100 )
```

```
CREATE ASSERTION smallClub
CHECK
( (SELECT COUNT (S.sid) FROM Sailors S)
+ (SELECT COUNT (B.bid) FROM Boats B) < 100 )
```

Triggers and Active Databases:

- Trigger: procedure that starts automatically if specified changes occur to the DBMS
- Three parts:
  - Event (activates the trigger)
  - Condition (tests whether the triggers should run)
  - Action (what happens if the trigger runs)

- Triggers: Example (SQL:1999)

```
CREATE TRIGGER youngSailorUpdate

AFTER INSERT ON SAILORS
REFERENCING NEW TABLE NewSailors

FOR EACH STATEMENT

        INSERT

        INTO YoungSailors(sid, name, age, rating)

        SELECT sid, name, age, rating

        FROM NewSailors N

        WHERE N.age <= 18
```

UNIT-III

Introduction To Schema Refinement:

The Evils of Redundancy

Redundancy is at the root of several problems associated with relational schemas:

– redundant storage, insert/delete/update anomalies

Integrity constraints, in particular *functional dependencies*, can be used to identify schemas with such problems and to suggest refinements.

Main refinement technique: *decomposition* (replacing ABCD with, say, AB and BCD, or ACD and ABD).

Decomposition should be used judiciously:

– Is there reason to decompose a relation?

– What problems (if any) does the decomposition cause?

Problems Caused by Redundancy:

- Storing the same information **redundantly**, that is, in more than one place within a database, can lead to several problems:
- **Redundant storage:** Some information is stored repeatedly.
- **Update anomalies:** If one copy of such repeated data is updated, an inconsistency
- is created unless all copies are similarly updated.
- **Insertion anomalies:** It may not be possible to store some information unless
- some other information is stored as well.
- **Deletion anomalies:** It may not be possible to delete some information without
- losing some other information as well.
- Consider a relation obtained by translating a variant of the Hourly Emps entity set

Ex: Hourly Emps(*ssn, name, lot, rating, hourly wages, hours worked*)

- The key for Hourly Emps is *ssn*. In addition, suppose that the *hourly wages* attribute
- is determined by the *rating* attribute. That is, for a given *rating* value, there is only
- one permissible *hourly wages* value. This IC is an example of a *functional dependency*.
- It leads to possible redundancy in the relation Hourly Emps

Decompositions:

- Intuitively, redundancy arises when a relational schema forces an association between attributes that is not natural.
- Functional dependencies (ICs) can be used to identify such situations and to suggest revetments to the schema.
- The essential idea is that many problems arising from redundancy can be addressed by replacing a relation with a collection of smaller relations.
- Each of the smaller relations contains a subset of the attributes of the original relation.
- We refer to this process as decomposition of the larger relation into the smaller relations
- We can deal with the redundancy in Hourly Emps by decomposing it into two relations:
- Hourly Emps2(*ssn, name, lot, rating, hours worked*)
- Wages(*rating, hourly wages*)

ssn	*name*	*lot*	*rating*	*hours worked*
123-22-3666	Attishoo	48	8	40
231-31-5368	Smiley	22	8	30
131-24-3650	Smethurst	35	5	30
434-26-3751	Guldu	35	5	32
612-67-4134	Madayan	35	8	40

Problems Related to Decomposition:

- Unless we are careful, decomposing a relation schema can create more problems than it solves.
- Two important questions must be asked repeatedly:
- 1. Do we need to decompose a relation?
- 2. What problems (if any) does a given decomposition cause?
- To help with the rst question, several *normal forms* have been proposed for relations.
- If a relation schema is in one of these normal forms, we know that certain kinds of
- problems cannot arise. Considering the n

Functional Dependencies (FDs):

- A functional dependency XY holds over relation R if, for every allowable instance *r* of R:

  t1 *r, t2* *r,* (*t1*) = (*t2*) implies (*t1*) = (*t2*)

  – i.e., given two tuples in *r*, if the X values agree, then the Y values must also agree. (X and Y are *sets* of attributes.)

- An FD is a statement about *all* allowable relations.
  – Must be identified based on semantics of application.
  – Given some allowable instance *r1* of R, we can check if it violates some FD *f*, but we cannot tell if *f* holds over R!
- K is a candidate key for R means that K R
  – However, K R does not require K to be *minimal*!

Example: Constraints on Entity Set

Example (Contd.)

- Problems due to R W
  - *Update anomaly*: Can we change W in just the 1st tuple of SNLRWH?
  - *Insertion anomaly*: What if we want to insert an employee and don't know the hourly wage for his rating?
  - *Deletion anomaly*: If we delete all employees with rating 5, we lose the information about the wage for rating 5!

Wages

R	W
8	10
5	7

Hourly_Emps2

S	N	L	R	H
123-22-3666	Attishoo	48	8	40
231-31-5368	Smiley	22	8	30
131-24-3650	Smethurst	35	5	30
434-26-3751	Guldu	35	5	32
612-67-4134	Madayan	35	8	40

S	N	L	R	W	H
123-22-3666	Attishoo	48	8	10	40
231-31-5368	Smiley	22	8	10	30
131-24-3650	Smethurst	35	5	7	30
434-26-3751	Guldu	35	5	7	32
612-67-4134	Madayan	35	8	10	40

Consider relation obtained from Hourly_Emps:

– Hourly_Emps (*ssn, name, lot, rating, hrly_wages, hrs_worked*)

• *Notation*: We will denote this relation schema by listing the attributes: SNLRWH
– This is really the *set* of attributes {S,N,L,R,W,H}.

– Sometimes, we will refer to all attributes of a relation by using the relation name. (e.g., Hourly_Emps for SNLRWH)

• Some FDs on Hourly_Emps:
– *ssn* is the key: S SNLRWH

– *rating* determines *hrly_wages*: R W

Constraints on a Relationship Set:

- Suppose that we have entity sets Parts, Suppliers, and Departments, as well as a relationship set Contracts that involves all of them. We refer to the schema for Contracts as *CQPSD*. A contract with contract id

• *C* species that a supplier *S* will supply some quantity *Q* of a part *P* to a department *D*.

• We might have a policy that a department purchases at most one part from any given supplier.

• Thus, if there are several contracts between the same supplier and department,

• we know that the same part must be involved in all of them. This constraint is an FD,

DS ! P.

Reasoning about FDs

• Given some FDs, we can usually infer additional FDs:
– *ssn did, did lot* implies *ssn lot*

• An FD *f* is *implied by* a set of FDs *F* if *f* holds whenever all FDs in *F* hold.
– = *closure of F* is the set of all FDs that are implied by *F*.

- Armstrong's Axioms (X, Y, Z are sets of attributes):
  - *Reflexivity*: If X Y, then Y X
  - *Augmentation*: If X Y, then XZ YZ for any Z
  - *Transitivity*: If X Y and Y Z, then X Z
- These are *sound* and *complete* inference rules for FDs!
- Couple of additional rules (that follow from AA):
  - *Union*: If X Y and X Z, then X YZ
  - *Decomposition*: If X YZ, then X Y and X Z
- Example: Contracts(*cid,sid,jid,did,pid,qty,value*), and:

  C is the key: C CSJDPQV
  - Project purchases each part using single contract:
  - JP C
  - Dept purchases at most one part from a supplier: S
  - D P
- JP C, C CSJDPQV imply JP CSJDPQV
- SD P implies SDJ JP
- SDJ JP, JP CSJDPQV imply SDJCSJDPQV

- Computing the closure of a set of FDs can be expensive. (Size of closure is exponential in # attrs!)
- Typically, we just want to check if a given FD *X* *Y* is in the closure of a set of FDs *F*. An efficient check:
  - Compute *attribute closure* of X (denoted) wrt *F:*
    - Set of all attributes A such that X A is in
    - There is a linear time algorithm to compute this.

– Check if Y is in

• Does F = {A B, B C, C D E } imply A E?

– i.e, is AE in the closure ? Equivalently, is E in ?

Closure of a Set of FDs

• The set of all FDs implied by a given set *F* of FDs is called the **closure of F** and is denoted as *F*+.

• An important question is how we can **infer**, or compute, the closure of a given set *F* of FD,S

• The following three rules, called **Armstrong's Axioms**, can be applied repeatedly to infer all FDs implied by a set *F* of FDs.

• We use *X*, *Y*, and *Z* to denote *sets* of attributes over a relation schema *R*:

• **Reflexivity:** If *X Y*, then *X !Y*.

• **Augmentation:** If *X ! Y*, then *XZ ! YZ* for any *Z*.

• **Transitivity:** If *X ! Y* and *Y ! Z*, then *X ! Z*.

• Armstrong's Axioms are **sound** in that they generate only FDs in *F*+ when applied to a set *F* of FDs.

• They are **complete** in that repeated application of these rules will generate all FDs in the closure *F*+.

• It is convenient to use some additional rules while reasoning about *F*+:

• **Union:** If *X ! Y* and *X ! Z*, then *X !YZ*.

• **Decomposition:** If *X ! YZ*, then *X !Y* and *X ! Z*.

- These additional rules are not essential; their soundness can be proved using Armstrong's Axioms.

Attribute Closure

- If we just want to check whether a given dependency, say, $X \rightarrow Y$, is in the closure of a set F of FDs,
- we can do so eciently without computing $F+$. We rst compute the **attribute closure** $X+$ with respect to F,
- which is the set of attributes A such that $X \rightarrow A$ can be inferred using the Armstrong Axioms.
- The algorithm for computing the attribute closure of a set X of attributes is
- *closure* = X;

repeat until there is no change: *{*

if there is an FD $U \rightarrow V$ in F such that *U subset of closure*,

then set *closure = closure union of V}*

Normal Forms:

- The normal forms based on FDs are *rst normal form (1NF)*, *second normal form (2NF)*, *third normal form (3NF)*, and *Boyce-Codd normal form (BCNF)*.
- These forms have increasingly restrictive requirements: Every relation in BCNF is also in 3NF,
- every relation in 3NF is also in 2NF, and every relation in 2NF is in 1NF.
- A relation
- is in **first normal form** if every field contains only atomic values, that is, not lists or sets.
- This requirement is implicit in our defition of the relational model.
- Although some of the newer database systems are relaxing this requirement
- 2NF is mainly of historical interest.
- 3NF and BCNF are important from a database design standpoint.

Normal Forms

- Returning to the issue of schema refinement, the first question to ask is whether any refinement is needed!

- If a relation is in a certain *normal form* (BCNF, 3NF etc.), it is known that certain kinds of problems are avoided/minimized. This can be used to help us decide whether decomposing the relation will help

- Role of FDs in detecting redundancy:

– Consider a relation R with 3 attributes, ABC.

- No FDs hold: There is no redundancy here.

- Given A B: Several tuples could have the same A value, and if so, they'll all have the same B value!

First Normal Form:

- 1NF (First Normal Form)
- a relation R is in 1NF if and only if it has only single-valued attributes (atomic values)
- EMP_PROJ (SSN, PNO, HOURS, ENAME, PNAME, PLOCATION)

  | | |

  PLOCATION is not in 1NF (multi-valued attrib.)
- solution: decompose the relation

  EMP_PROJ2 (SSN, PNO, HOURS, ENAME, PNAME)

  |

  LOC (PNO, PLOCATION)

  | |

  | |

Second Normal Form:

- 2NF (Second Normal Form)
- a relation R in 2NF if and only if it is in 1NF and every nonkey column depends on a key not a subset of a key
- all nonprime attributes of R must be fully functionally dependent on a whole key(s) of the relation, not a part of the key
- no violation: single-attribute key or no nonprime attribute
- 2NF (Second Normal Form)
- violation: part of a key → nonkey

  EMP_PROJ2 (SSN, PNO, HOURS, ENAME, PNAME)

  SSN → ENAME

  PNO → PNAME
- solution: decompose the relation

  EMP_PROJ3 (SSN, PNO, HOURS)

  EMP (SSN, ENAME)

  PROJ (PNO, PNAME)

Third Normal Form:

- 3NF (Third Normal Form)
- a relation R in 3NF if and only if it is in 2NF and every nonkey column does not depend on another nonkey column
- all nonprime attributes of R must be non-transitively functionally dependent on a key of the relation
- violation: nonkey → nonkey
- 3NF (Third Normal Form)
- SUPPLIER (SNAME, STREET, CITY, STATE, TAX)

  SNAME → STREET, CITY, STATE

  STATE → TAX (nonkey → nonkey)

  SNAME → STATE → TAX (transitive FD)
- solution: decompose the relation

  SUPPLIER2 (SNAME, STREET, CITY, STATE)

  TAXINFO (STATE, TAX)
- Boyce-Codd Normal Form (BCNF)
- Reln R with FDs *F* is in BCNF if, for all X A in
  - A X (called a *trivial* FD), or
  - X contains a key for R.

- In other words, R is in BCNF if the only non-trivial FDs that hold over R are key constraints.
  - No dependency in R that can be predicted using FDs alone.
  - If we are shown two tuples that agree upon the X value, we cannot infer the A value in one tuple from the A value in the other.
  - If example relation is in BCNF, the 2 tuples must be identical (since X is a key).

Third Normal Form (3NF)

- Reln R with FDs *F* is in 3NF if, for all XA in
  - AX (called a *trivial* FD), or
  - X contains a key for R, or
  - A is part of some key for R.
- *Minimality* of a key is crucial in third condition above!
  - If R is in BCNF, obviously in 3NF.

BCNF:

- If R is in 3NF, some redundancy is possible. It is a compromise, used when BCNF not achievable (e.g., no ``good'' decomp, or performance considerations).
  - *Lossless-join, dependency-preserving decomposition of R into a collection of 3NF relations always possible.*

Properties of Decompositions :

- Suppose that relation R contains attributes *A1 ... An.* A *decomposition* of R consists of replacing R by two or more relations such that:
  - Each new relation scheme contains a subset of the attributes of R (and no attributes that do not appear in R), and
  - Every attribute of R appears as an attribute of one of the new relations.
- Intuitively, decomposing R means we will store instances of the relation schemes produced by the decomposition, instead of instances of R.
- E.g., Can decompose SNLRWH into SNLRH and RW.

Example Decomposition

- Decompositions should be used only when needed.

– SNLRWH has FDs SSNLRWH and RW

– Second FD causes violation of 3NF; W values repeatedly associated with R values. Easiest way to fix this is to create a relation RW to store these associations , and to remove W from the main schema:

i.e., we decompose SNLRWH into SNLRH and RW

- The information to be stored consists of SNLRWH tuples. If we just store the projections of these tuples onto SNLRH and RW, are there any potential problems that we should be aware of?

Problems with Decompositions

- There are three potential problems to consider:

– Some queries become more expensive.

- e.g., How much did sailor Joe earn? (salary = W*H)

– Given instances of the decomposed relations, we may not be able to reconstruct the corresponding instance of the original relation!

- Fortunately, not in the SNLRWH example.

– Checking some dependencies may require joining the instances of the decomposed relations.

- Fortunately, not in the SNLRWH example.

- *Tradeoff*: Must consider these issues vs. redundancy.

Lossless Join Decompositions:

- Decomposition of R into X and Y is *lossless-join* w.r.t. a set of FDs F if, for every instance *r* that satisfies F:

– (*r*) (*r*) = *r*

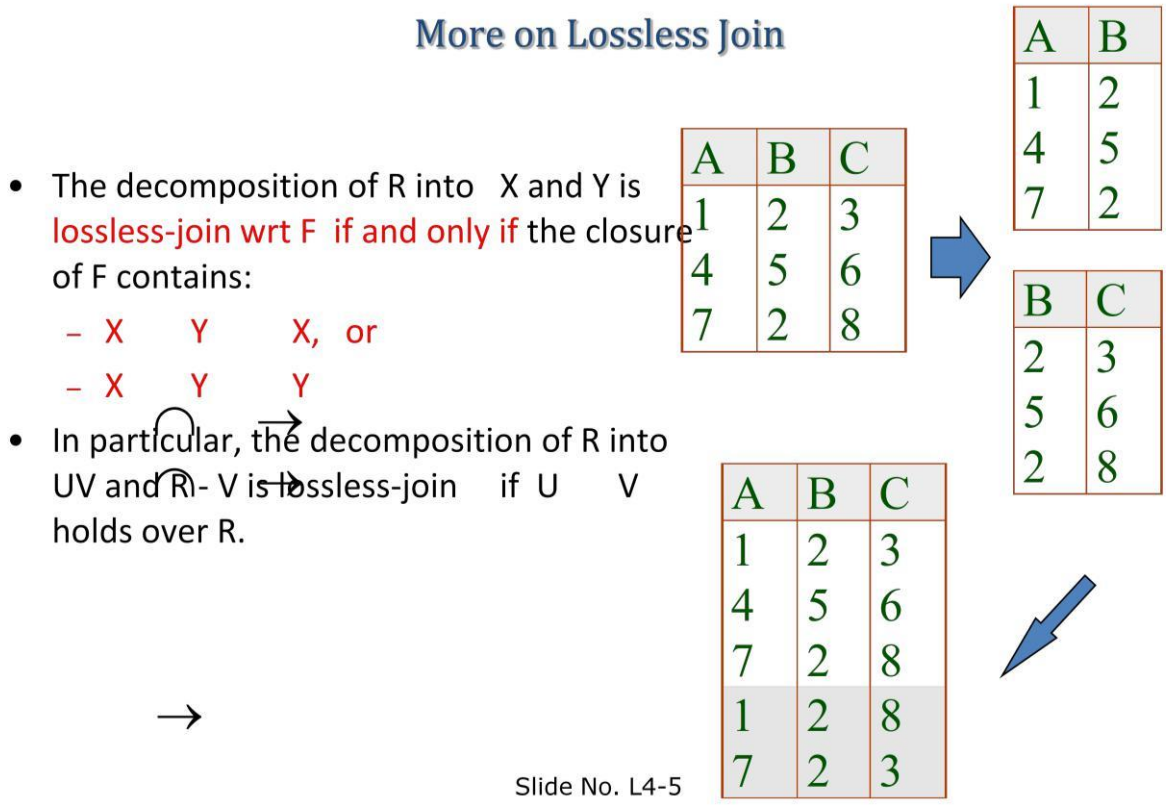
- It is always true that *r* (*r*) (*r*)

– In general, the other direction does not hold! If it does, the decomposition is lossless-join.

- Definition extended to decomposition into 3 or more relations in a straightforward way.
- *It is essential that all decompositions used to deal with redundancy be lossless! (Avoids*

Problem (2).)

More on Lossless Join



- Dependency Preserving Decomposition

- Consider CSJDPQV, C is key, JP C and SDP.
  – BCNF decomposition: CSJDQV and SDP

  – Problem: Checking JP C requires a join!

<u>Dependency</u> preserving decomposition (Intuitive):

  – If R is decomposed into X, Y and Z, and we enforce the FDs that hold on X, on Y and on Z, then all FDs that were given to hold on R must also hold. *(Avoids Problem (3).)*

- *<u>Projection of set of FDs F</u>*: If R is decomposed into X, ... projection of F onto X enoted F_X) is the set of FDs U V in F^+ (*closure of F*) such that U, V are in X.

- Decomposition of R into X and Y is *dependency preserving*

  if $(F_X \text{ union } F_Y)^+ = F^+$

  – i.e., if we consider only dependencies in the closure F^+ that can be checked in X without considering Y, and in Y without considering X, these imply all dependencies in F^+.

- Important to consider F^+, not F, in this definition:

  – ABC, A B, B C, C A, decomposed into AB and BC.

  – Is this dependency preserving? Is C A preserved?????

- Dependency preserving does not imply lossless join:

  – ABC, A B, decomposed into AB and BC.

- And vice-versa! (Example?)

Decomposition into BCNF

- Consider relation R with FDs F. If X Y violates BCNF, decompose R into R - Y and XY.

  – Repeated application of this idea will give us a collection of relations that are in BCNF; lossless join decomposition, and guaranteed to terminate.

  – e.g., CSJDPQV, key C, JP C, SD P, J S

  – To deal with SD P, decompose into SDP, CSJDQV.

  – To deal with J S, decompose CSJDQV into JS and CJDQV

- In general, several dependencies may cause violation of BCNF. The order in which we ``deal with'' them could lead to very different sets of relations!

BCNF and Dependency Preservation

- In general, there may not be a dependency preserving decomposition into BCNF.

– e.g., CSZ, CS Z, Z C

– Can't decompose while preserving 1st FD; not in BCNF.

> Similarly, decomposition of CSJDQV into SDP, JS and CJDQV is not dependency preserving (w.r.t. the FDs JP C, SD P and J S).

– However, it is a lossless join decomposition.

– In this case, adding JPC to the collection of relations gives us a dependency preserving decomposition.

- JPC tuples stored only for checking FD! (*Redundancy!*)

Decomposition into 3NF

- Obviously, the algorithm for lossless join decomp into BCNF can be used to obtain a lossless join decomp into 3NF (typically, can stop earlier).
- To ensure dependency preservation, one idea:

– If X Y is not preserved, add relation XY.

– Problem is that XY may violate 3NF! e.g., consider the addition of CJP to `preserve' JP C. What if we also have J C ?

- Refinement: Instead of the given set of FDs F, use a *minimal cover for F*.

Schema Refinement in Data base Design:

Constraints on an Entity Set

- Consider the Hourly Emps relation again. The constraint that attribute *ssn* is a key can be expressed as an FD:
- *{ ssn }-> { ssn, name, lot, rating, hourly wages, hours worked}*

- For brevity, we will write this FD as *S* -> *SNLRWH*, using a single letter to denote each attribute
- In addition, the constraint that the *hourly wages* attribute is determined by the *rating* attribute is an

FD: *R* -> *W*.

Constraints on a Relationship Set

- The previous example illustrated how FDs can help to rene the subjective decisions made during ER design,
- but one could argue that the best possible ER diagram would have led to the same nal set of relations.
- Our next example shows how FD information can lead to a set of relations that eliminates some redundancy problems and is unlikely to be arrived at solely through ER design.

Identifying Attributes of Entities

- in particular, it shows that attributes can easily be associated with the `wrong' entity set during ER design.
- The ER diagram shows a relationship set called Works In that is similar to the Works In relationship set
- Using the key constraint, we can translate this ER diagram into two relations:
- Workers(*ssn, name, lot, did, since*)

Identifying Entity Sets

- Let Reserves contain attributes *S*, *B*, and *D* as before, indicating that sailor *S* has a reservation for boat *B* on day *D*.
- In addition, let there be an attribute *C* denoting the credit card to which the reservation is charged.
- Suppose that every sailor uses a unique credit card for reservations. This constraint is expressed by the FD

S -> *C*. This constraint indicates that in relation Reserves, we store the credit card number for a sailor as often as we have reservations for that

- sailor, and we have redundancy and potential update anomalies.

Multivalued Dependencies:

- Suppose that we have a relation with attributes *course*, *teacher*, and *book*, which we denote as *CTB*.
- The meaning of a tuple is that teacher *T* can teach course *C*, and book *B* is a recommended text for the course.
- There are no FDs; the key is *CTB*. However, the recommended texts for a course are independent of the instructor.

There are three points to note here:

- The relation schema *CTB* is in BCNF; thus we would not consider decomposing it further if we looked only at the FDs that hold over *CTB*.
- There is redundancy. The fact that Green can teach Physics101 is recorded once per recommended text for the course. Similarly, the fact that Optics is a text for Physics101 is recorded once per potential teacher.
- The redundancy can be eliminated by decomposing *CTB* into *CT* and *CB*.
- Let *R* be a relation schema and let *X* and *Y* be subsets of the attributes of *R*. Intuitively,
- the **multivalued dependency** *X !! Y* is said to hold over *R* if, in every legal
- The redundancy in this example is due to the constraint that the texts for a course are independent of the instructors, which cannot be epressed in terms of FDs.
- This constraint is an example of a *multivalued dependency*, or MVD. Ideally, we should model this situation using two binary relationship sets, Instructors with attributes *CT* and Text with attributes *CB*.
- Because these are two essentially independent relationships, modeling them with a single ternary relationship set with attributes *CTB* is inappropriate.
- Three of the additional rules involve only MVDs:

MVD Complementation: If $X \rightarrow\rightarrow Y$, then $X \rightarrow\rightarrow R - XY$

MVD Augmentation: If $X \rightarrow\rightarrow Y$ and $W > Z$, then

$WX \rightarrow\rightarrow YZ$.

- **MVD Transitivity:** If $X \rightarrow\rightarrow Y$ and $Y \rightarrow\rightarrow Z$, then

$X \rightarrow\rightarrow (Z - Y)$.

Fourth Normal Form:

- R is said to be in **fourth normal form (4NF)** if for every MVD $X \rightarrow\rightarrow Y$ that holds over R, one of the following statements is true:
- Y *subset of* X or $XY = R$, or
- X is a superkey.

Join Dependencies:

- A join dependency is a further generalization of MVDs. A **join dependency** (JD) $\infty\{ R1,\ldots.. Rn \}$ is said to hold over a relation R if $R1,\ldots. Rn$ is a lossless-join decomposition of R.
- An MVD X ->-> Y over a relation R can be expressed as the join dependency ∞ { XY,X(R−Y)*}*
- As an example, in the *CTB* relation, the MVD C ->->T can be expressed as the join dependency $\infty\{$ CT, CB*}*
- Unlike FDs and MVDs, there is no set of sound and complete inference rules for JDs.

Fifth Normal Form:

- A relation schema R is said to be in **fth normal form (5NF)** if for every JD $\infty\{ R1,\ldots. Rn \}$ that holds over R, one of the following statements is true:
- $Ri = R$ for some i, or

- The JD is implied by the set of those FDs over *R* in which the left side is a key for *R*.
- The following result, also due to Date and Fagin, identies conditions|again, detected using only FD information|under which we can safely ignore JD information.
- If a relation schema is in 3NF and each of its keys consists of a single attribute,it is also in 5NF.

Inclusion Dependencies:

2. MVDs and JDs can be used to guide database design, as we have seen, although they are less common than FDs and harder to recognize and reason about.
3. In contrast, inclusion dependencies are very intuitive and quite common. However, they typically have little influence on database design
4. The main point to bear in mind is that we should not split groups of attributes that participate in an inclusion dependency.
5. Most inclusion dependencies in practice are *key-based*, that is, involve only keys.

UNIT-IV

Transaction Concept:

1. A **transaction** is a *unit* of program execution that accesses and possibly updates various data items.
2. E.g. transaction to transfer $50 from account A to account B:

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

4. Two main issues to deal with:

– Failures of various kinds, such as hardware failures and system crashes

– Concurrent execution of multiple transactions

Example of Fund Transfer

- Transaction to transfer $50 from account A to account B:
- **read**(*A*)
- *A* := *A* – 50
- **write**(*A*)
- **read**(*B*)
- *B* := *B* + 50
- **write**(*B)*

Atomicity requirement

– if the transaction fails after step 3 and before step 6, money will be "lost" leading to an inconsistent database state

- Failure could be due to software or hardware

– the system should ensure that updates of a partially executed transaction are not reflected in the database

Durability requirement — once the user has been notified that the transaction has completed (i.e., the transfer of the $50 has taken place), the updates to the database by the transaction must persist even if there are software or hardware failures.

Example of Fund Transfer (Cont.)

- Transaction to transfer $50 from account A to account B:
- **read**(*A*)
- *A* := *A* – 50
- **write**(*A*)
- **read**(*B*)
- *B* := *B* + 50
- **write**(*B)*

Consistency requirement in above example:

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

- In general, consistency requirements include
- Explicitly specified integrity constraints such as primary keys and foreign keys
- Implicit integrity constraints

– e.g. sum of balances of all accounts, minus sum of loan amounts must equal value of cash-in-hand

– A transaction must see a consistent database.

– During transaction execution the database may be temporarily inconsistent.

– When the transaction completes successfully the database must be consistent

• Erroneous transaction logic can lead to inconsistency

Example of Fund Transfer (Cont.)

Isolation requirement — if between steps 3 and 6, another transaction T2 is allowed to access the partially updated database, it will see an inconsistent database (the sum $A + B$ will be less than it should be).

T1 **T2**

- **read**(A)
- $A := A - 50$
- **write**(A)

read(A), read(B), print(A+B)

- **read**(B)
- $B := B + 50$
- **write**(B

- Isolation can be ensured trivially by running transactions **serially**

– that is, one after the other.

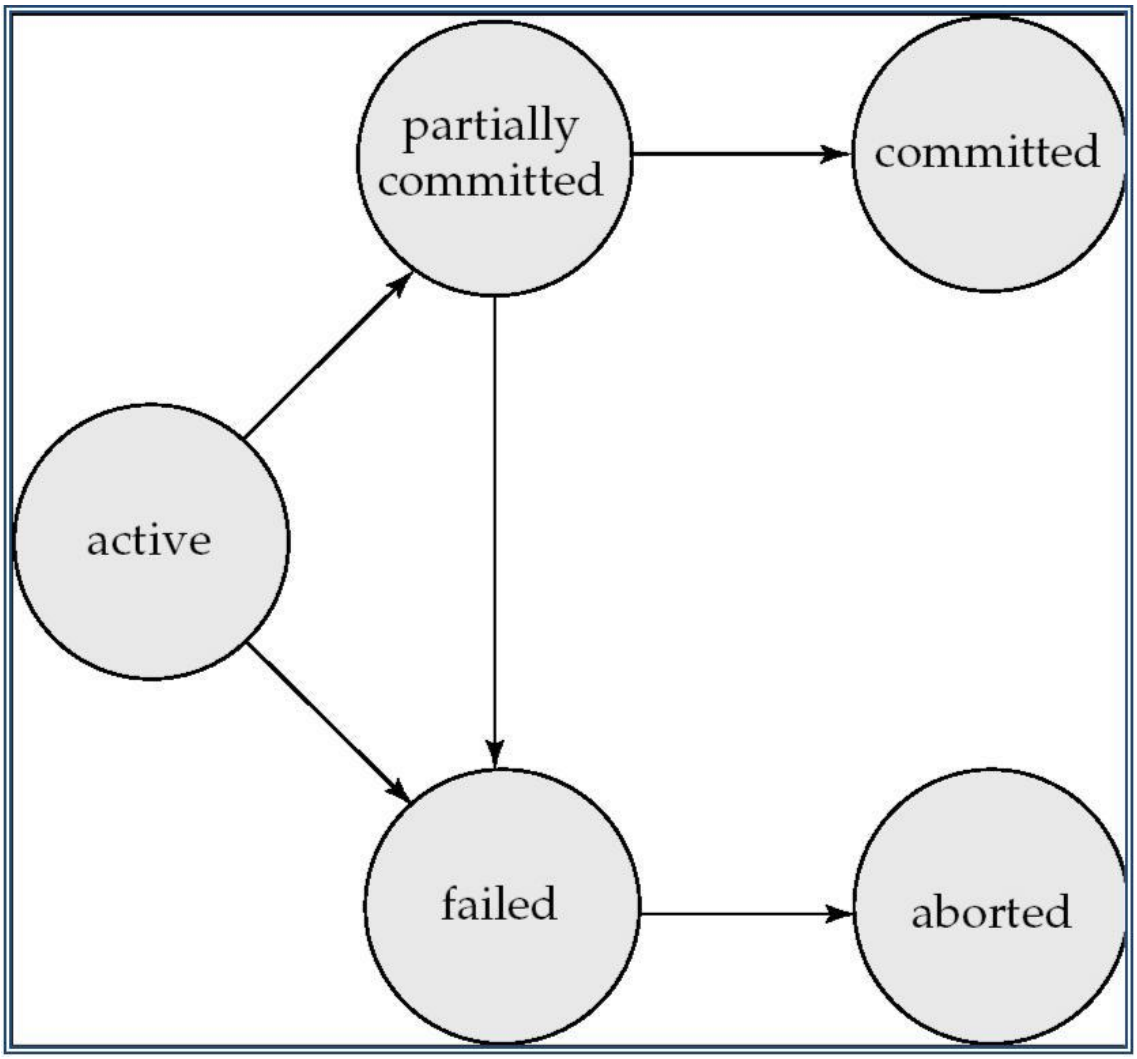
- However, executing multiple transactions concurrently has significant benefits, as we will see later.

ACID Properties

- **Atomicity.** Either all operations of the transaction are properly reflected in the database or none are.
- **Consistency.** Execution of a transaction in isolation preserves the consistency of the database.
- **Isolation.** Although multiple transactions may execute concurrently, each transaction must be unaware of other concurrently executing transactions. Intermediate transaction results must be hidden from other concurrently executed transactions.
  - That is, for every pair of transactions T_i and T_j, it appears to T_i that either T_j, finished execution before T_i started, or T_j started execution after T_i finished.
- **Durability.** After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures.

Transaction State:

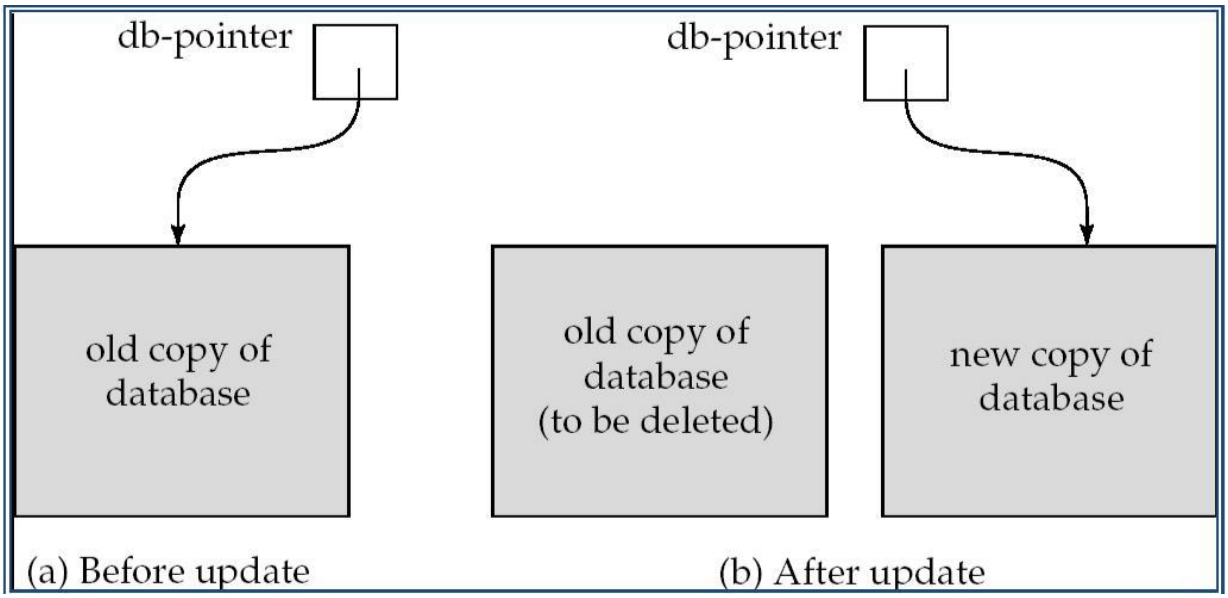
- **Active** – the initial state; the transaction stays in this state while it is executing
- **Partially committed** – after the final statement has been executed.
- **Failed --** after the discovery that normal execution can no longer proceed.
- **Aborted** – after the transaction has been rolled back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



Implementation of Atomicity and Durability:

- The **recovery-management** component of a database system implements the support for atomicity and durability.
- E.g. the ***shadow-database*** scheme:
  - all updates are made on a *shadow copy* of the database
  - **db_pointer** is made to point to the updated shadow copy after
    - the transaction reaches partial commit and
    - all updated pages have been flushed to disk.

- **db_pointer** always points to the current consistent copy of the database.
  - In case transaction fails, old consistent copy pointed to by **db_pointer** can be used, and the shadow copy can be deleted.



- The shadow-database scheme:
  - Assumes that only one transaction is active at a time.
  - Assumes disks do not fail
  - Useful for text editors, but
    - extremely inefficient for large databases (why?)
      - Variant called shadow paging reduces copying of data, but is still not practical for large databases
  - Does not handle concurrent transactions
- Will study better schemes in Chapter 17.

4.Concurrent Executions:

- Multiple transactions are allowed to run concurrently in the system. Advantages are:
  - **increased processor and disk utilization**, leading to better transaction *throughput*

- E.g. one transaction can be using the CPU while another is reading from or writing to the disk
  - **reduced average response time** for transactions: short transactions need not wait behind long ones.
- **Concurrency control schemes** – mechanisms to achieve isolation
  - that is, to control the interaction among the concurrent transactions in order to prevent them from destroying the consistency of the database
- Will study in Chapter 16, after studying notion of correctness of concurrent executions.

Schedules

- **Schedule** – a sequences of instructions that specify the chronological order in which instructions of concurrent transactions are executed
  - a schedule for a set of transactions must consist of all instructions of those transactions
  - must preserve the order in which the instructions appear in each individual transaction.
- A transaction that successfully completes its execution will have a commit instructions as the last statement
  - by default transaction assumed to execute commit instruction as its last step
- A transaction that fails to successfully complete its execution will have an abort instruction as the last statement

Schedule 1

- Let T_1 transfer \$50 from *A* to *B*, and T_2 transfer 10% of the balance from *A* to *B*.
- A serial schedule in which T_1 is followed by T_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)

Schedule 2

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

Schedule 3

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

- Let T_1 and T_2 be the transactions defined previously. The following schedule is not a serial schedule, but it is *equivalent* to Schedule 1.

  Schedule 4

- The following concurrent schedule does not preserve the value of $(A + B)$.

Serializability:

- **Basic Assumption** – Each transaction preserves database consistency.
- Thus serial execution of a set of transactions preserves database consistency.
- A (possibly concurrent) schedule is serializable if it is equivalent to a serial schedule.

  Different forms of schedule equivalence give rise to the notions of:

- **conflict serializability**

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

- **view serializability**

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

Simplified view of transactions

– We ignore operations other than **read** and **write** instructions

– We assume that transactions may perform arbitrary computations on data in local buffers in between reads and writes.

– Our simplified schedules consist of only **read** and **write** instructions.

Conflicting Instructions

- Instructions l_i and l_j of transactions T_i and T_j respectively, **conflict** if and only if there exists some item Q accessed by both l_i and l_j, and at least one of these instructions wrote Q.

1. l_i = **read**(Q), l_j = **read**(Q). l_i and l_j don't conflict.
2. l_i = **read**(Q), l_j = **write**(Q). They conflict.
3. l_i = **write**(Q), l_j = **read**(Q). They conflict
4. l_i = **write**(Q), l_j = **write**(Q). They conflict

- Intuitively, a conflict between l_i and l_j forces a (logical) temporal order between them.

– If l_i and l_j are consecutive in a schedule and they do not conflict, their results would remain the same even if they had been interchanged in the schedule.

Conflict Serializability

- 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' are **conflict equivalent**.
- We say that a schedule S is **conflict serializable** if it is conflict equivalent to a serial schedule

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

- Schedule 3 can be transformed into Schedule 6, a serial schedule where T_2 follows T_1, by series of swaps of non-conflicting instructions.
  - Therefore Schedule 3 is conflict serializable.
- Example of a schedule that is not conflict serializable:
- We are unable to swap instructions in the above schedule to obtain either the serial schedule < T_3, T_4 >, or the serial schedule < T_4, T_3 >.
- View Serializability
- Let S and S' be two schedules with the same set of transactions. S and S' are **view equivalent** if the following three conditions are met, for each data item Q,
  - If in schedule S, transaction T_i reads the initial value of Q, then in schedule S' also transaction T_i must read the initial value of Q.
  - If in schedule S transaction T_i executes **read**(Q), and that value was produced by transaction T_j (if any), then in schedule S' also transaction T_i must read the value of Q that was produced by the same **write**(Q) operation of transaction T_j .
  - The transaction (if any) that performs the final **write**(Q) operation in schedule S must also perform the final **write**(Q) operation in schedule S'.

  As can be seen, view equivalence is also based purely on **reads** and **writes** alone.

T_3	T_4	T_6
read(Q)		
	write(Q)	
write(Q)		
		write(Q)

- A schedule S is **view serializable** if it is view equivalent to a serial schedule.
- Every conflict serializable schedule is also view serializable.
- Below is a schedule which is view-serializable but *not* conflict serializable.
- What serial schedule is above equivalent to?
- Every view serializable schedule that is not conflict serializable has **blind writes.**
- Other Notions of Serializability

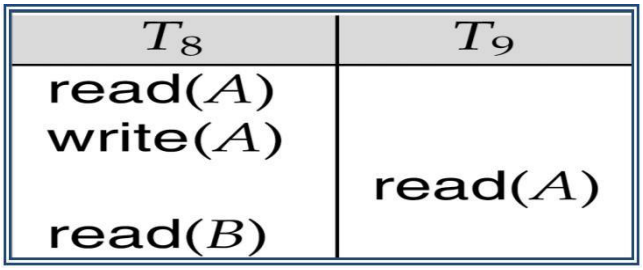
T_1	T_5
read(A)	
$A := A - 50$	
write(A)	
	read(B)
	$B := B - 10$
	write(B)
read(B)	
$B := B + 50$	
write(B)	
	read(A)
	$A := A + 10$
	write(A)

- The schedule below produces same outcome as the serial schedule < T_1, T_5 >, yet is not conflict equivalent or view equivalent to it.

Determining such equivalence requires analysis of operations other than read and write.

Recoverability:

- **Recoverable schedule** — if a transaction T_j reads a data item previously written by a transaction T_i , then the commit operation of T_i appears before the commit operation of T_j.
- The following schedule (Schedule 11) is not recoverable if T_9 commits immediately after the read



T_8	T_9
read(A)	
write(A)	
	read(A)
read(B)	

- If T_8 should abort, T_9 would have read (and possibly shown to the user) an inconsistent database state. Hence, database must ensure that schedules are recoverable.

Cascading Rollbacks

- **Cascading rollback** – a single transaction failure leads to a series of transaction rollbacks. Consider the following schedule where none of the transactions has yet committed (so the schedule is recoverable)

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

If T_{10} fails, T_{11} and T_{12} must also be rolled back.

- Can lead to the undoing of a significant amount of work
- **Cascadeless schedules** — cascading rollbacks cannot occur; 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 read operation of T_j.
- Every cascadeless schedule is also recoverable
- It is desirable to restrict the schedules to those that are cascadeless

Concurrency Control

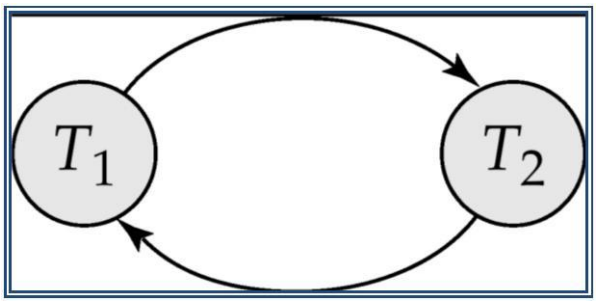
- A database must provide a mechanism that will ensure that all possible schedules are
  - either conflict or view serializable, and
  - are recoverable and preferably cascadeless
- A policy in which only one transaction can execute at a time generates serial schedules, but provides a poor degree of concurrency
  - Are serial schedules recoverable/cascadeless?
- Testing a schedule for serializability *after* it has executed is a little too late!

- **Goal** – to develop concurrency control protocols that will assure serializability.

Implementation of Isolation:

- Schedules must be conflict or view serializable, and recoverable, for the sake of database consistency, and preferably cascadeless.
- A policy in which only one transaction can execute at a time generates serial schedules, but provides a poor degree of concurrency.
- Concurrency-control schemes tradeoff between the amount of concurrency they allow and the amount of overhead that they incur.
- Some schemes allow only conflict-serializable schedules to be generated, while others allow view-serializable schedules that are not conflict-serializable.

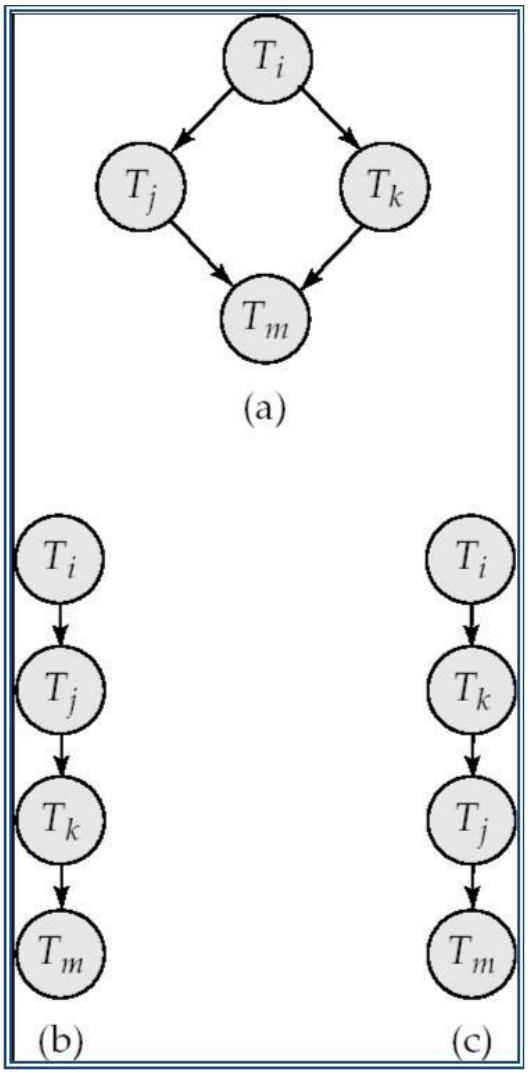
Testing for Serializability:



- Consider some schedule of a set of transactions T_1, T_2, ..., T_n
- **Precedence graph** — a direct graph where the vertices are the transactions (names).
- We draw an arc from T_i to T_j if the two transaction conflict, and T_i accessed the data item on which the conflict arose earlier.
- We may label the arc by the item that was accessed.

Test for Conflict Serializability

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



- If precedence graph is acyclic, the serializability order can be obtained by a *topological sorting* of the graph.

– This is a linear order consistent with the partial order of the graph.

– For example, a serializability order for Schedule A would be $T_5 \rightarrow T_1 \rightarrow T_3 \rightarrow T_2 \rightarrow T_4$

- Are there others?

Test for View Serializability

> The precedence graph test for conflict serializability cannot be used directly to test for view serializability.

– Extension to test for view serializability has cost exponential in the size of the precedence graph.

- The problem of checking if a schedule is view serializable falls in the class of *NP*-complete problems.
  - Thus existence of an efficient algorithm is *extremely* unlikely.
- However practical algorithms that just check some **sufficient conditions** for view serializability can still be used.

Concurrency Control:

Concurrency Control vs. Serializability Tests

- Concurrency-control protocols allow concurrent schedules, but ensure that the schedules are conflict/view serializable, and are recoverable and cascadeless .
- Concurrency control protocols generally do not examine the precedence graph as it is being created
  - Instead a protocol imposes a discipline that avoids nonseralizable schedules.
  - We study such protocols in Chapter 16.
- Different concurrency control protocols provide different tradeoffs between the amount of concurrency they allow and the amount of overhead that they incur.
- Tests for serializability help us understand why a concurrency control protocol is correct.

Weak Levels of Consistency

- Some applications are willing to live with weak levels of consistency, allowing schedules that are not serializable
  - E.g. a read-only transaction that wants to get an approximate total balance of all

– E.g. database statistics computed for query optimization can be approximate (why?)

– Such transactions need not be serializable with respect to other transactions

- Tradeoff accuracy for performance
- Levels of Consistency in SQL-92

Serializable — default

- **Repeatable read** — only committed records to be read, repeated reads of same record must return same value. However, a transaction may not be serializable – it may find some records inserted by a transaction but not find others.
- **Read committed** — only committed records can be read, but successive reads of record may return different (but committed) values.
- **Read uncommitted** — even uncommitted records may be read.
- Transaction Definition in SQL
- Data manipulation language must include a construct for specifying the set of actions that comprise a transaction.
- In SQL, a transaction begins implicitly.
- A transaction in SQL ends by:

– **Commit work** commits current transaction and begins a new one.

– **Rollback work** causes current transaction to abort.

- In almost all database systems, by default, every SQL statement also commits implicitly if it executes successfully

– Implicit commit can be turned off by a database directive

- E.g. in JDBC, connection.setAutoCommit(false);

Lock-Based Protocols:

- A lock is a mechanism to control concurrent access to a data item

	S	X
S	true	false
X	false	false

- Data items can be locked in two modes :

1. *exclusive (X) mode*. Data item can be both read as well as written. X-lock is requested using **lock-X** instruction.
2. *shared (S) mode*. Data item can only be read. S-lock is requested using **lock-S** instruction.

- Lock requests are made to concurrency-control manager. Transaction can proceed only after request is granted.
- Lock-compatibility matrix
- A transaction may be granted a lock on an item if the requested lock is compatible with locks already held on the item by other transactions
- Any number of transactions can hold shared locks on an item,

– but if any transaction holds an exclusive on the item no other transaction may hold any lock on the item.

- If a lock cannot be granted, the requesting transaction is made to wait till all incompatible locks held by other transactions have been released. The lock is then granted.
- Example of a transaction performing locking:

T_2: **lock-S**(*A*);

read *(A)*;

unlock*(A)*;

lock-S*(B)*;

read *(B)*;

unlock*(B)*;

display*(A+B)*

- Locking as above is not sufficient to guarantee serializability — if *A* and *B* get updated in-between the read of *A* and *B*, the displayed sum would be wrong.
- A **locking protocol** is a set of rules followed by all transactions while requesting and releasing locks. Locking protocols restrict the set of possible schedules.
- Pitfalls of Lock-Based Protocols
- Consider the partial schedule
- Neither T_3 nor T_4 can make progress — executing **lock-S***(B)* causes T_4 to wait for T_3 to release its lock on *B*, while executing **lock-X***(A)* causes T_3 to wait for T_4 to release its lock on *A*.
- Such a situation is called a **deadlock**.

– To handle a deadlock one of T_3 or T_4 must be rolled back and its locks released.

- The potential for deadlock exists in most locking protocols. Deadlocks are a necessary evil.
- **Starvation** is also possible if concurrency control manager is badly designed. For example:

– A transaction may be waiting for an X-lock on an item, while a sequence of other transactions request and are granted an S-lock on the same item.

- The same transaction is repeatedly rolled back due to deadlocks.

- Concurrency control manager can be designed to prevent starvation.
- The Two-Phase Locking Protocol
- This is a protocol which ensures conflict-serializable schedules.
- Phase 1: Growing Phase
  - transaction may obtain locks
  - transaction may not release locks
- Phase 2: Shrinking Phase
  - transaction may release locks
  - transaction may not obtain locks
- The protocol assures serializability. It can be proved that the transactions can be serialized in the order of their **lock points** (i.e. the point where a transaction acquired its final lock).
- Two-phase locking *does not* ensure freedom from deadlocks
- Cascading roll-back is possible under two-phase locking. To avoid this, follow a modified protocol called **strict two-phase locking**. Here a transaction must hold all its exclusive locks till it commits/aborts.
- **Rigorous two-phase locking** is even stricter: here *all* locks are held till commit/abort. In this protocol transactions can be serialized in the order in which they commit.
- The Two-Phase Locking Protocol (Cont.)
- 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.

- Lock Conversions
- Two-phase locking with lock conversions:
  - First Phase:
    - can acquire a lock-S on item
    - can acquire a lock-X on item
    - can convert a lock-S to a lock-X (upgrade)
  - Second Phase:
    - can release a lock-S
    - can release a lock-X
    - 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.

Automatic Acquisition of Locks

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

7. 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

ii **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;

xi All locks are released after commit or abort

xii Implementation of Locking

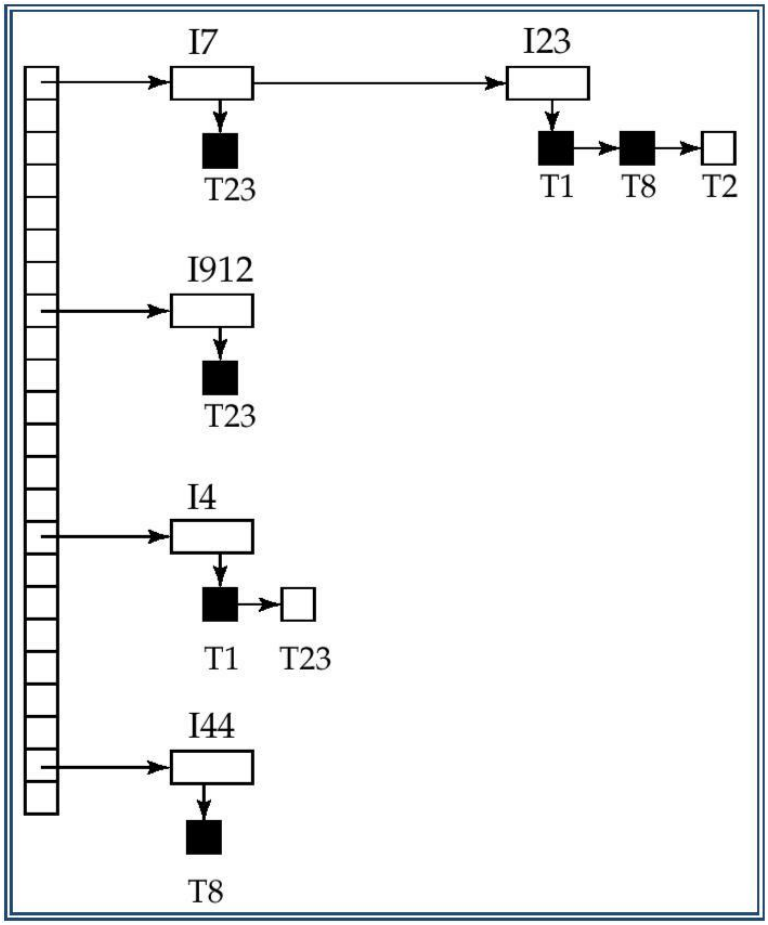
xiii A **lock manager** can be implemented as a separate process to which transactions send lock and unlock requests

xiv 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)

xv The requesting transaction waits until its request is answered

- The lock manager maintains a data-structure called a **lock table** to record granted locks and pending requests
- The lock table is usually implemented as an in-memory hash table indexed on the name of the data item being locked

Lock Table



- Black rectangles indicate granted locks, white ones indicate waiting requests
- Lock table also records the type of lock granted or requested
- 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
- Unlock requests result in the request being deleted, and later requests are checked to see if they can now be granted

- If transaction aborts, all waiting or granted requests of the transaction are deleted

– lock manager may keep a list of locks held by each transaction, to implement this efficiently

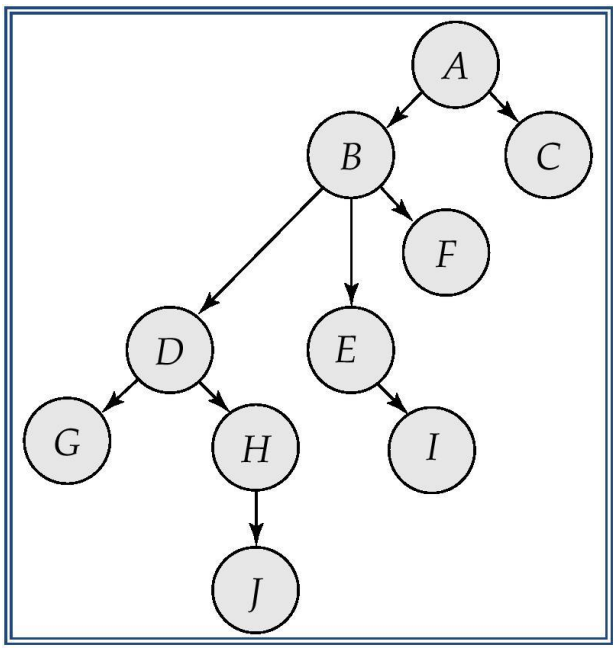
- Graph-Based Protocols
- Graph-based protocols are an alternative to two-phase locking
- Impose a partial ordering $\rightarrow$ on the set **D** = {d_1, d_2 ,..., d_h} of all data items.

– If $d_i \rightarrow d_j$ then any transaction accessing both d_i and d_j must access d_i before accessing d_j.

– 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.

Tree 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

Timestamp-Based Protocols:

- Each transaction is issued a timestamp when it enters the system. If an old transaction T_i has time-stamp TS(T_i), a new transaction T_j is assigned time-stamp TS(T_j) such that TS(T_i) <TS(T_j).
- The protocol manages concurrent execution such that the time-stamps determine the serializability order.
- In order to assure such behavior, the protocol maintains for each data Q two timestamp values:
  - **W-timestamp**(Q) is the largest time-stamp of any transaction that executed **write**(Q) successfully.
  - **R-timestamp**(Q) is the largest time-stamp of any transaction that executed **read**(Q) successfully.
- Timestamp-Based Protocols (Cont.)
- The timestamp ordering protocol ensures that any conflicting **read** and **write** operations are executed in timestamp order.
- Suppose a transaction T_i issues a **read**(Q)
  - If TS(T_i) ≤ **W**-timestamp(Q), then T_i needs to read a value of Q that was already overwritten.

    Hence, the **read** operation is rejected, and T_i is rolled back.

– If TS(T_i)≥ **W**-timestamp(Q), then the **read** operation is executed, and R-timestamp(Q) is set to **max**(R-timestamp(Q), TS(T_i)).

• Suppose that transaction T_i issues **write**(Q).

– If TS(T_i) < R-timestamp(Q), then the value of Q that T_i is producing was needed previously, and the system assumed that that value would never be produced.

Hence, the **write** operation is rejected, and T_i is rolled back.

– If TS(T_i) < W-timestamp(Q), then T_i is attempting to write an obsolete value of Q.

Hence, this **write** operation is rejected, and T_i is rolled back.

– Otherwise, the **write** operation is executed, and W-timestamp(Q) is set to TS(T_i).

Example Use of the Protocol

Example Use of the Protocol

A partial schedule for several data items for transactions with timestamps 1, 2, 3, 4, 5

T_1	T_2	T_3	T_4	T_5
				read(X)
read(Y)	read(Y)			
		write(Y)		
		write(Z)		
				read(Z)
	read(X)			
	abort			
read(X)				
		write(Z)		
		abort		
				write(Y)
				write(Z)

Slide No. L8-4

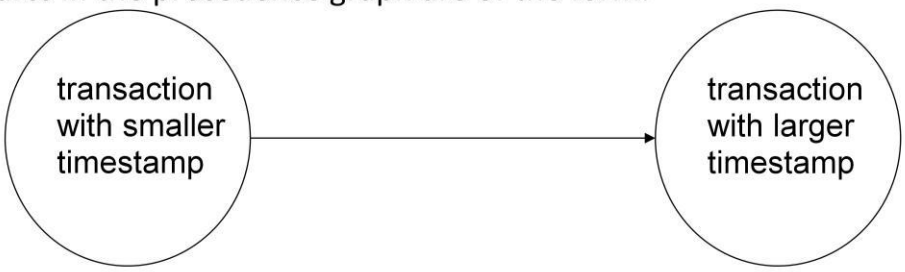
A partial schedule for several data items for transactions with

timestamps 1, 2, 3, 4, 5

Correctness of Timestamp-Ordering Protocol

Correctness of Timestamp-Ordering Protocol

- The timestamp-ordering protocol guarantees serializability since all the arcs in the precedence graph are of the form:



Thus, there will be no cycles in the precedence graph

- Timestamp protocol ensures freedom from deadlock as no transaction ever waits.
- But the schedule may not be cascade-free, and may not even be recoverable.

Slide No. L8-5

- The timestamp-ordering protocol guarantees serializability since all the arcs in the precedence graph are of the form:

  Thus, there will be no cycles in the precedence graph
- Timestamp protocol ensures freedom from deadlock as no transaction ever waits.
- But the schedule may not be cascade-free, and may not even be recoverable.
- Thomas' Write Rule
- Modified version of the timestamp-ordering protocol in which obsolete **write** operations may be ignored under certain circumstances.
- When T_i attempts to write data item Q, if TS(T_i) < W-timestamp(Q), then T_i is attempting to write an obsolete value of $\{Q\}$.

– Rather than rolling back T_i as the timestamp ordering protocol would have done, this {**write**} operation can be ignored.

- Otherwise this protocol is the same as the timestamp ordering protocol.
- Thomas' Write Rule allows greater potential concurrency.

– Allows some view-serializable schedules that are not conflict-serializable.

Validation-Based Protocols:

- Execution of transaction T_i is done in three phases.
- **Read and execution phase**: Transaction T_i writes only to temporary local variables
- **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.

- 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

- Start(T_i) : the time when T_i started its execution
- Validation(T_i): the time when T_i entered its validation phase
- 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.
- Validation Test for Transaction T_j
- If for all T_i with TS (T_i) < TS (T_j) either one of the following condition holds:
  - **finish**(T_i) < **start**(T_j)
  - **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

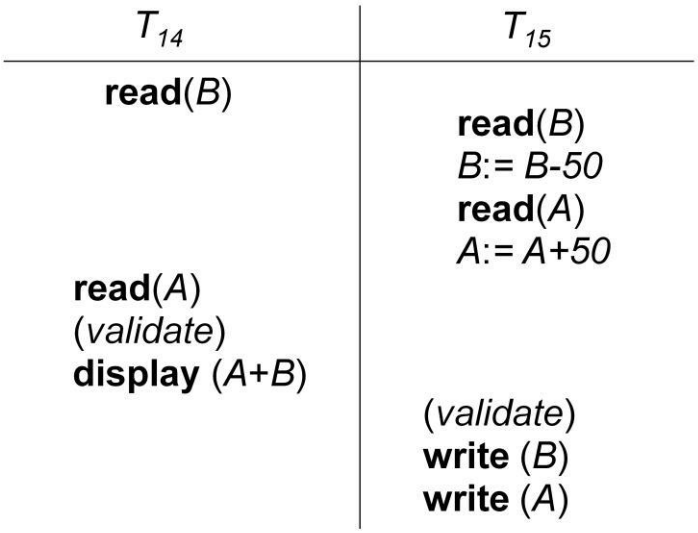
  the writes of T_j do not affect reads of T_i since they occur after T_i has finished its reads.

  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 by Validation

Schedule Produced by Validation

- Example of schedule produced using validation



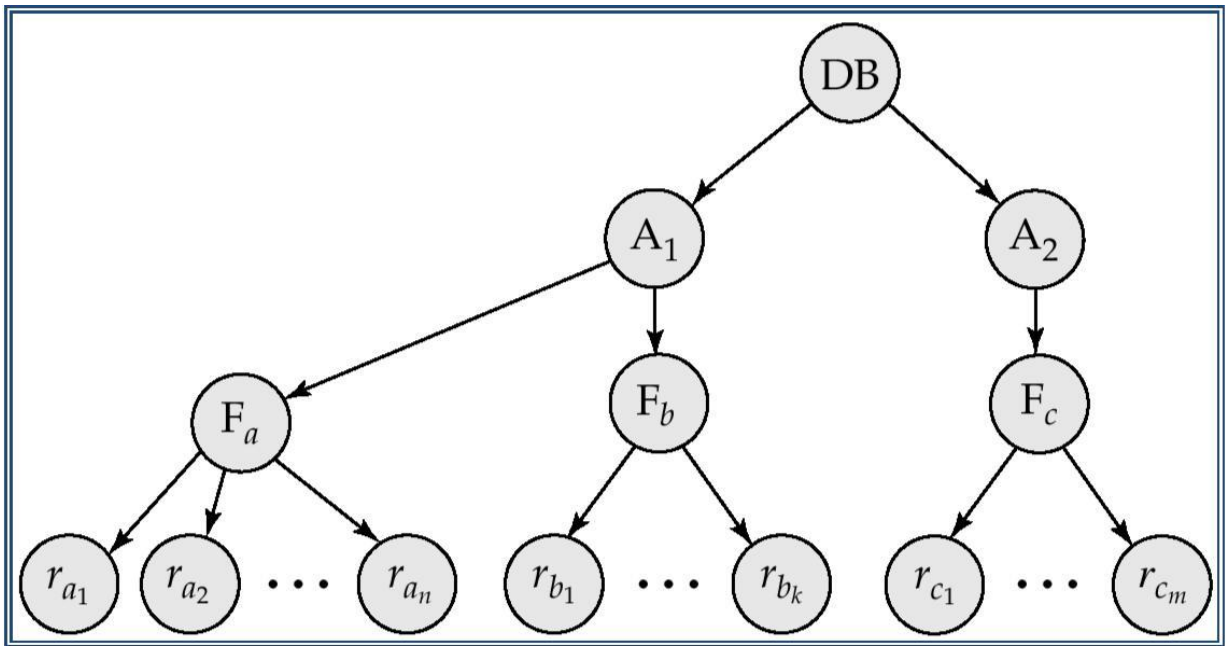
13.Multiple Granularity:

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

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

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

Example of Granularity Hierarchy



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

– *database*

– *area*

– *file*

– *record*

• Intention Lock 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 with Intention Lock Modes

- The compatibility matrix for all lock modes is:

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

Slide No. L9-8

- The compatibility matrix for all lock modes is:

- Multiple Granularity Locking Scheme
- Transaction T_i can lock a node Q, using the following rules:

– The lock compatibility matrix must be observed.
– The root of the tree must be locked first, and may be locked in any mode.
– 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.
– 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.

– T_i can lock a node only if it has not previously unlocked any node (that is, T_i is two-phase).

– 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.

Recovry System-Failure Classification:

To see where the problem has occurred we generalize the failure into various categories, as follows:

Transaction failure

When a transaction is failed to execute or it reaches a point after which it cannot be completed successfully it has to abort. This is called transaction failure. Where only few transaction or process are hurt.

Reason for transaction failure could be:

- **Logical errors:** where a transaction cannot complete because of it has some code error or any internal error condition
- **System errors:** where the database system itself terminates an active transaction because DBMS is not able to execute it or it has to stop because of some system condition. For example, in case of deadlock or resource unavailability systems aborts an active transaction.

System crash

There are problems, which are external to the system, which may cause the system to stop abruptly and cause the system to crash. For example interruption in power supply, failure of underlying hardware or software failure.

Examples may include operating system errors.

Disk failure:

In early days of technology evolution, it was a common problem where hard disk drives or storage drives used to fail frequently.

Disk failures include formation of bad sectors, unreachability to the disk, disk head crash or any other failure, which destroys all or part of disk storage

Storage Structure:

We have already described storage system here. In brief, the storage structure can be divided in various categories:

- **Volatile storage:** As name suggests, this storage does not survive system crashes and mostly placed very closed to CPU by embedding them onto the chipset itself for examples: main memory, cache memory. They are fast but can store a small amount of information.
- **Nonvolatile storage:** These memories are made to survive system crashes. They are huge in data storage capacity but slower in accessibility. Examples may include, hard disks, magnetic tapes, flash memory, non-volatile (battery backed up) RAM.

Recovery and Atomicity:

Modifying the database without ensuring that the transaction will commit may leave the database in an inconsistent state.Consider transaction T_i that transfers \$50 from account *A* to account *B*; goal is either to perform all database modifications made by T_i or none at all. Several output operations may be required for T_i (to output *A* and *B*). A failure may occur after one of these modifications have been made but before all of them are made.To ensure atomicity despite failures, we first output information describing the modifications to stable storage without modifying the database itself.We study two approaches:

log-based recovery, and **shadow-paging**

Recovery Algorithms

- Recovery algorithms are techniques to ensure database consistency and transaction atomicity and durability despite failures
- Recovery algorithms have two parts
  - Actions taken during normal transaction processing to ensure enough information exists to recover from failures
  - Actions taken after a failure to recover the database contents to a state that ensures atomicity, consistency and durability.

17.Log-Based Recovery:

- A **log** is kept on stable storage.
  - The log is a sequence of **log records**, and maintains a record of update activities on the database.
- When transaction T_i starts, it registers itself by writing a <T_i **start**>log record
- *Before* T_i executes **write**(X), a log record <T_i, X, V_1, V_2> is written, where V_1 is the value of X before the write, and V_2 is the value to be written to X.
  - Log record notes that T_i has performed a write on data item X_j X_j had value V_1 before the write, and will have value V_2 after the write.
- When T_i finishes it last statement, the log record <T_i **commi**t> is written.

- We assume for now that log records are written directly to stable storage (that is, they are not buffered)
- Two approaches using logs
  - Deferred database modification
  - Immediate database modification
- Deferred Database Modification
- The **deferred database modification** scheme records all modifications to the log, but defers all the **write**s to after partial commit.
- Assume that transactions execute serially
- Transaction starts by writing $<T_i$ ***start***$>$ record to log.
- A **write**(X) operation results in a log record $<T_i, X, V>$ being written, where V is the new value for X
  - Note: old value is not needed for this scheme
- The write is not performed on X at this time, but is deferred.
- When T_i partially commits, $<T_i$ **commit**$>$ is written to the log
- Finally, the log records are read and used to actually execute the previously deferred writes.
- During recovery after a crash, a transaction needs to be redone if and only if both $<T_i$ **start**$>$ and$<T_i$ **commit**$>$ are there in the log.
- Redoing a transaction T_i (**redo**T_i) sets the value of all data items updated by the transaction to the new values.
- Crashes can occur while
  - the transaction is executing the original updates, or

– while recovery action is being taken

- example transactions T_0 and T_1 (T_0 executes before T_1):T_0: **read** (A) T_1 : **read** (C)

  A: - A - 50 *C:- C- 100*
  Write (A) **write** (C)

  read (B)

  B:- B + 50

  write (B)

- Below we show the log as it appears at three instances of time.
- If log on stable storage at time of crash is as in case:
  - No redo actions need to be taken
  - redo(T_0) must be performed since <T_0 **commi**t> is present
  - **redo**(T_0) must be performed followed by redo(T_1) since

    <T_0 **commit**> and <T_i commit> are present

- Immediate Database Modification
- The **immediate database modification** scheme allows database updates of an uncommitted transaction to be made as the writes are issued

– since undoing may be needed, update logs must have both old value and new value

- Update log record must be written *before* database item is written

– We assume that the log record is output directly to stable storage
– Can be extended to postpone log record output, so long as prior to execution of an **output**(B) operation for a data block B, all log records corresponding toitems B must be flushed to stable storage

Immediate Database Modification

- Output of updated blocks can take place at any time before or after transaction commit
- Order in which blocks are output can be different from the order in which they are written.
- Recovery procedure has two operations instead of one:
  - **undo**(T_i) restores the value of all data items updated by T_i to their old values, going backwards from the last log record for T_i
  - **redo**(T_i) sets the value of all data items updated by T_i to the new values, going forward from the first log record for T_i
- Both operations must be **idempotent**
  - That is, even if the operation is executed multiple times the effect is the same as if it is executed once
- Needed since operations may get re-executed during recovery
- When recovering after failure:
  - Transaction T_i needs to be undone if the log contains the record <T_i **start**>, but does not contain the record <T_i **commit**>.
  - Transaction T_i needs to be redone if the log contains both the record <T_i **start**> and the record <T_i **commit**>.
- Undo operations are performed first, then redo operations.
- Immediate Database Modification Example

Log	Write	Output
<T_0 **start**>		
<T_0, A, 1000, 950>		
T_0, B, 2000, 2050		
	A = 950	
	B = 2050	
<T_0 **commit**>		
<T_1 **start**>		
<T_1, C, 700, 600>		
	C = 600	
		B_B, B_C
<T_1 **commit**>		
		B_A

- Note: B_X denotes block containing X.
- Immediate DB Modification Recovery Example

Below we show the log as it appears at three instances of time.

Recovery actions in each case above are:

- undo (T_0): B is restored to 2000 and A to 1000.
- undo (T_1) and redo (T_0): C is restored to 700, and then A and B are set to 950 and 2050 respectively.
- redo (T_0) and redo (T_1): A and B are set to 950 and 2050 respectively. Then C is set to 600

Checkpoints

- Problems in recovery procedure as discussed earlier :

1. searching the entire log is time-consuming
2. we might unnecessarily redo transactions which have already
3. output their updates to the database.

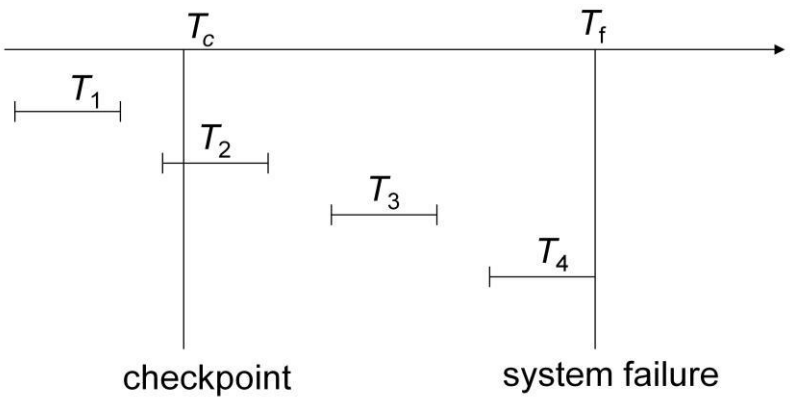
- Streamline recovery procedure by periodically performing **checkpointing**

1. Output all log records currently residing in main memory onto stable storage.
2. Output all modified buffer blocks to the disk.
3. Write a log record < **checkpoint**> onto stable storage.

- During recovery we need to consider only the most recent transaction T_i that started before the checkpoint, and transactions that started after T_i.

1. Scan backwards from end of log to find the most recent <**checkpoint**> record
2. Continue scanning backwards till a record <T_i **start**> is found.
3. Need only consider the part of log following above **star**t record. Earlier part of log can be ignored during recovery, and can be erased whenever desired.
4. For all transactions (starting from T_i or later) with no <T_i **commit**>, execute **undo(***T_i***)**. (Done only in case of immediate modification.)
5. Scanning forward in the log, for all transactions starting from T_i or later with a <T_i **commit**>, execute **redo(***T_i***)**.

Example of Checkpoints



- T_1 can be ignored (updates already output to disk due to checkpoint)
- T_2 and T_3 redone.
- T_4 undone

- Recovery With Concurrent Transactions
- We modify the log-based recovery schemes to allow multiple transactions to execute concurrently.

1. All transactions share a single disk buffer and a single log
2. A buffer block can have data items updated by one or more transactions

- We assume concurrency control using strict two-phase locking;

1. i.e. the updates of uncommitted transactions should not be visible to other transactions

- Otherwise how to perform undo if T1 updates A, then T2 updates A and commits, and finally T1 has to abort?

- Logging is done as described earlier.

1. Log records of different transactions may be interspersed in the log.

22. Recovery with Concurrent Transactions:

- The checkpointing technique and actions taken on recovery have to be changed

1. since several transactions may be active when a checkpoint is performed.

- Checkpoints are performed as before, except that the checkpoint log record is now of the form

  checkpoint *L*>

  where *L* is the list of transactions active at the time of the checkpoint

1. We assume no updates are in progress while the checkpoint is carried out (will relax this later)

- When the system recovers from a crash, it first does the following:

1. Initialize *undo-list* and *redo-list* to empty
2. Scan the log backwards from the end, stopping when the first <**checkpoint** *L*> record is found.

For each record found during the backward scan:

- if the record is <T_i **commit**>, add T_i to *redo-list*
- if the record is <T_i **start**>, then if T_i is not in *redo-list*, add T_i to *undo-list*

3. For every T_i in *L*, if T_i is not in *redo-list*, add T_i to *undo-list*

- At this point *undo-list* consists of incomplete transactions which must be undone, and *redo-list* consists of finished transactions that must be redone.

Recovery now continues as follows:

1. Scan log backwards from most recent record, stopping when <T_i **start**> records have been encountered for every T_i in *undo-list*.

- During the scan, perform **undo** for each log record that belongs to a transaction in *undo-list*.

2. Locate the most recent <**checkpoint** *L*> record.

3. Scan log forwards from the <**checkpoint** *L*> record till the end of the log.

- During the scan, perform **redo** for each log record that belongs to a transaction on *redo-list*

Example of Recovery

- Go over the steps of the recovery algorithm on the following log:

<T_0 **star**t>

<T_0, *A*, 0, 10>

<T_0 **commit**>

<T_1 **start**> /* Scan at step 1 comes up to here */

<T_1, *B*, 0, 10>

<T_2 **start**>

<T_2, *C*, 0, 10>

<T_2, *C*, 10, 20>

<checkpoint {T_1, T_2}>

<T_3 **start**>

<T_3, *A*, 10, 20>

<T_3, *D*, 0, 10>

<T_3 **commit**>

Log Record Buffering

- **Log record buffering**: log records are buffered in main memory, instead of of being output directly to stable storage.
  - Log records are output to stable storage when a block of log records in the buffer is full, or a **log force** operation is executed.
- Log force is performed to commit a transaction by forcing all its log records (including the commit record) to stable storage.
- Several log records can thus be output using a single output operation, reducing the I/O cost.
- The rules below must be followed if log records are buffered:
  - Log records are output to stable storage in the order in which they are created.
  - Transaction T_i enters the commit state only when the log record <T_i **commit**> has been output to stable storage.
  - Before a block of data in main memory is output to the database, all log records pertaining to data in that block must have been output to stable storage.
    - This rule is called the **write-ahead logging** or **WAL** rule
      - Strictly speaking WAL only requires undo information to be output

Buffer Management:

- Database maintains an in-memory buffer of data blocks
  - When a new block is needed, if buffer is full an existing block needs to be removed from buffer
  - If the block chosen for removal has been updated, it must be output to disk
- If a block with uncommitted updates is output to disk, log records with undo information for the updates are output to the log on stable storage first
  - (Write ahead logging)
- No updates should be in progress on a block when it is output to disk. Can be ensured as follows.
  - Before writing a data item, transaction acquires exclusive lock on block containing the data item
  - Lock can be released once the write is completed.
    - Such locks held for short duration are called **latches**.
  - Before a block is output to disk, the system acquires an exclusive latch on the block
    - Ensures no update can be in progress on the block
- Database buffer can be implemented either
  - in an area of real main-memory reserved for the database, or
  - in virtual memory
- Implementing buffer in reserved main-memory has drawbacks:
  - Memory is partitioned before-hand between database buffer and applications, limiting flexibility.

Needs may change, and although operating system knows best how memory should be divided up at any time, it cannot change the partitioning of memory.

- Database buffers are generally implemented in virtual memory in spite of some drawbacks:
  - When operating system needs to evict a page that has been modified, the page is written to swap space on disk.
  - When database decides to write buffer page to disk, buffer page may be in swap space, and may have to be read from swap space on disk and output to the database on disk, resulting in extra I/O!
    - Known as **dual paging** problem.
  - Ideally when OS needs to evict a page from the buffer, it should pass control to database, which in turn should
    - Output the page to database instead of to swap space (making sure to output log records first), if it is modified
    - Release the page from the buffer, for the OS to use

    Dual paging can thus be avoided, but common operating systems do not support such functionality.

Failure with Loss of Nonvolatile Storage:

- So far we assumed no loss of non-volatile storage
- Technique similar to checkpointing used to deal with loss of non-volatile storage
  - Periodically **dump** the entire content of the database to stable storage
  - No transaction may be active during the dump procedure; a procedure similar to checkpointing must take place

    Output all log records currently residing in main memory onto stable storage.

ARIES

- ARIES is a state of the art recovery method
  - Incorporates numerous optimizations to reduce overheads during normal processing and to speed up recovery
  - The "advanced recovery algorithm" we studied earlier is modeled after ARIES, but greatly simplified by removing optimizations
- Unlike the advanced recovery algorithm, ARIES
  - Uses **log sequence number (LSN)** to identify log records
    - Stores LSNs in pages to identify what updates have already been applied to a database page

Physiological redo

- Dirty page table to avoid unnecessary redos during recovery
- Fuzzy checkpointing that only records information about dirty pages, and does not require dirty pages to be written out at checkpoint time

More coming up on each of the above …

Remote Backup Systems

- Remote backup systems provide high availability by allowing transaction processing to continue even if the primary site is destroyed.
- **Detection of failure**: Backup site must detect when primary site has failed

FILE ORGANIZATIONS AND INDEXING

5.1.0 FILE ORGANIZATIONS AND INDEXING: The file of records is an important abstraction in a DBMS, and is implemented by the ***files.*** A file can be created, destroyed, and have records inserted into and deleted from it.

A relation is typically stored in a file of ***records***. The file layer stores the records in a file in a collection of disk ***pages***. It keeps track of pages allocated to each file, and as ***records*** are inserted into and deleted from the file, it also tracks available space within pages allocated to the file.

The simplest file structure is an ***unordered file***, or ***heap file***. Records in a heap file are stored in ***random order*** across the pages of the file. A heap file organization supports retrieval of all records, or retrieval of a particular record specified by its rid; the file manager must keep track of the pages allocated for the file.

An ***index*** is a data structure that organizes data records on disk to optimize certain kinds of retrieval operations. An index allows us to *efficiently retrieve* all records that satisfy search conditions on the search key fields of the index. We can also create additional indexes on a given collection of data records, each with a different search key, to speed up search operations that are not efficiently supported by the file organization used to store the data records.

There are ***three*** main alternatives for what to store as a data entry in an index:

3. A data entry K* is an actual data record (with search key value k).

4. A data entry is a (k, rid) pair, where rid is the record id of a data record with search key value k.

5. A data entry is a (k, rid-list) pair, where rid-list is a list of record ids of data records with search key value k.

5.1.1 Clustered Indexes: When a file is organized so that the ordering of data records is the same as or close to the ordering of data entries in some index, we say that the index is ***clustered***; otherwise, it is an ***unclustered index***. An index that can be a clustered only if the data records are ***sorted on the search key field***. Otherwise, the order of the data records is random, defined purely by their physical order, and there is no reasonable way to arrange the data entries in the index in the same order.

The cost of using an index to answer a range search query can vary tremendously based on whether the index is clustered. If the index is clustered, i.e., we are using the search key of a clustered file, the rids in qualifying data entries point to a contiguous collection of records, and we need to retrieve only a few data pages.

Two data entries are said to be duplicates if they have the same value for the search key field associated with the index. A ***primary index*** is guaranteed ***not to contain duplicates***, but an index on other (collections of) fields can contain duplicates. In general, a ***secondary index contains duplicates***. If we know that no duplicates exist, that is, we know that the search key contains some candidate key, we call the index a unique index.

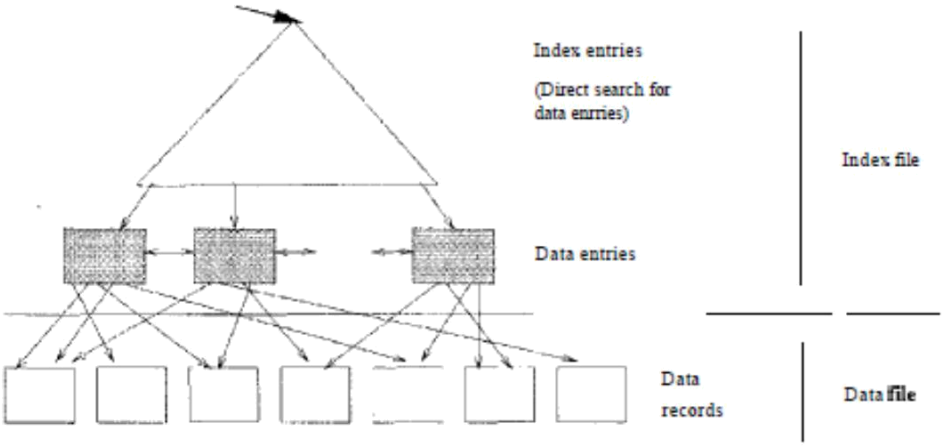


Figure 5.1 Clustered Index

5.2 INDEX DATA STRUCTURES

5.2.1 Hash-Based Indexing: We can organize records using a technique called hashing to quickly find records that have a given search ***key value***. In this approach, the records in a file are grouped in ***buckets***, where a bucket consists of a *primary page* and, possibly, *additional pages* linked in a chain. The bucket to which a record belongs can be determined by applying a special function, called ***a hash function***, to the search key. Given a ***bucket number***, a hash-based index structure allows us to retrieve the primary page for the bucket in one or two disk l/Os.

On inserts, the record is inserted into the appropriate bucket, with 'overflow' pages allocated as necessary. To search for a record with a given search key value, we apply the hash function to identify the bucket to which such records belong and look at all pages in that bucket. If we do not have the search key value for the record.

Hash indexing is illustrated in Figure 5.2, where the data is stored in a file that is hashed on age; the data entries in this first index file are the actual data records. Applying the hash function to the age field identifies the page that the record belongs to. The hash function h for this example is quite simple; it converts the search key value to its binary representation and uses the two least significant bits as the bucket identifier. Figure 5.2 also shows an index with search key sal that contains (sal, rid) pairs as data entries.

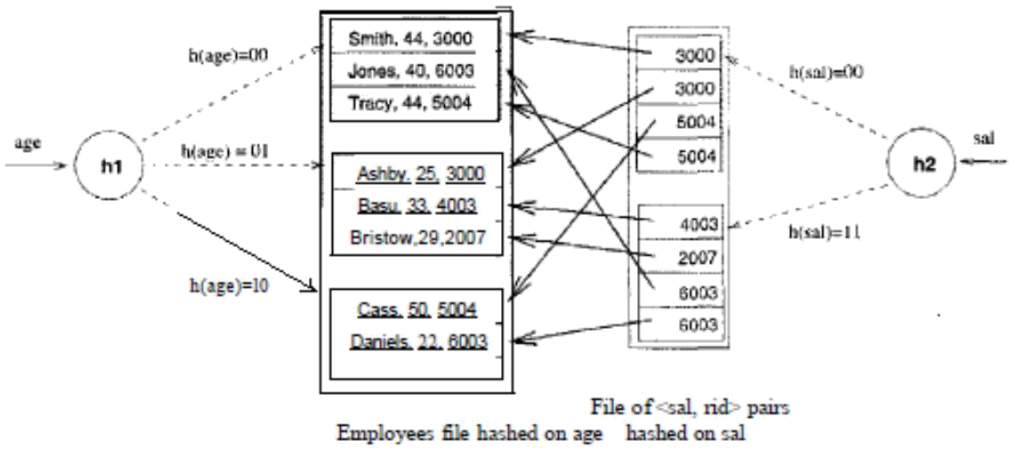
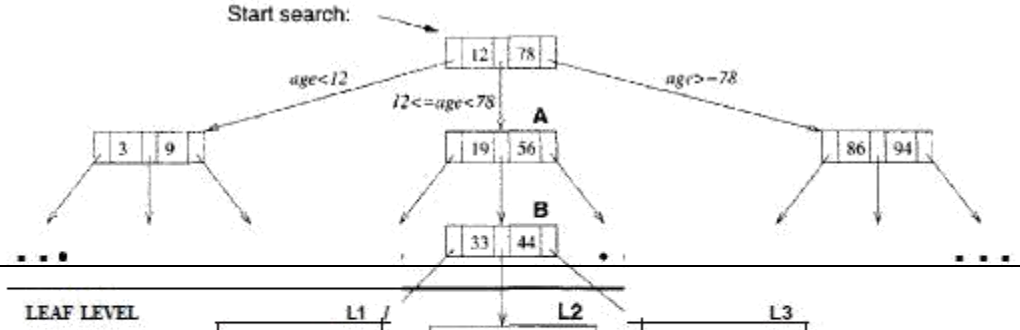


Figure 5.2 Index-Organized File Hashed on *age*, with Auxiliary Index on *sal*

5.2.2 Tree-Based Indexing: An alternative to hash-based indexing is to organize records using a tree like data structure. The data entries are arranged in sorted order by search key value, and a hierarchical search data structure is maintained that directs searches to the correct page of data entries. Figure 5.3 shows the employee records organized in a tree-structured index with search key age

The lowest level of the tree, called the leaf level, contains the data entries; there were additional employee records, some with age less than 22 and some with age greater than 50 (the lowest and highest age values that appear in Figure 5.3). Additional records with age less than 22 would appear in leaf pages to the left page L1, and records with age greater than 50 would appear in leaf pages to the right of page.



The **B+ tree** is an index structure that ensures that all paths from the root to a leaf in a given tree are of the same length, that is, the structure is always balanced in height. Finding the correct leaf page is faster than binary search of the pages in a sorted file because each non-leaf node can accommodate a very large number of node-pointers, and the height of the tree

is rarely more than three or four in practice. The **height** of a balanced tree is the length of a path from root to leaf. The average number of children for a non-leaf node is called the **fan-out** of the tree. If every non-leaf node has n children, a tree of height h has $\mathbf{n^h}$ leaf pages.

5.3 COMPARISON OF FILE ORGANIZATIONS

We now compare the costs of some simple operations for several basic file organizations on a collection of employee records. We assume that the files and indexes are organized according to the composite search key (age,sal) and that all selection operations are specified on these fields.

Our goal is to emphasize the importance of the choice of an appropriate file organization, and the above list includes the main alternatives to consider in practice. Obviously, we can keep the records unsorted or sort them. We can also choose to build an index on the data file. Note that even if the data file is sorted, an index whose search key differs from the sort order behaves like an index on a heap file.

The operations we consider are these:

7. Scan: Fetch all records in the file. The pages in the file must be fetched from disk into the buffer pool.
8. Search with Equality Selection: Fetch all records that satisfy an equality selection.
9. **Search with Range Selection**: Fetch all records that satisfy a range selection

10. **Insert a Record**: Insert a given record into the file. We must identify the page in the file into which the new record must be inserted, fetch that page from disk, modify it to include the new record, and then write back the modified page.

11. **Delete a Record**: Delete a record that is specified using its rid. We must identify the page that contains the record, fetch it from disk, modify it, and write it back.

5.3.1 Cost Model: In comparison of file organizations, we use a simple cost model that allows us to estimate the cost (in terms of execution time) of different database operations. We use **B** to denote the *number of data pages* when records are packed onto pages with no wasted space, and **R** to denote the *number of records per page*. The *average time* to read or write a disk page is **D**, and the *average time to process a record* (e.g., to compare a field value to a selection constant) is **C**.

In the hashed file organization, we use a function, called a hash function, to map a record into a range of numbers; the time required to apply the hash *function to a record* is H. For tree indexes, we will use F to denote the *fan-out*, which typically is at least 100 as mentioned. Typical values today are D = 15 milliseconds, C and H = 100 nanoseconds; we therefore expect the cost of I/O to dominate. I/O is often (even typically) the dominant component of the cost of database operations, and so considering I/O costs gives us a good first approximation to the true costs. Further, CPU speeds are steadily rising, whereas disk speeds are not increasing at a similar pace.

We have chosen to concentrate on the I/O component of the cost model, and we assume the simple constant C for in-memory per-record processing cost. Bear the following observations in mind:

5. Real systems must consider other aspects of cost, such as CPU costs (and network transmission costs in a distributed database).

6. Even with our decision to focus on I/O costs, an accurate model would be too complex for our purposes of conveying the essential ideas in a simple way.

We therefore use a simplistic model in which we just count the number of pages read from or written to disk as a measure of I/O. The cost is equal to the time required to seek the first page in the block and transfer all pages in the block. Such blocked access can be much cheaper than issuing one I/O request per page in the block, especially if these requests do not follow consecutively, because we would have an additional seek cost for each page in the block.

File Type	*Scan*	*Equality Search*	*Range Search*	*Insert*	*Delete*
Heap	*BD*	0.5*BD*	*BD*	2*D*	*Search*+ D
Sorted	*BD*	*Dlog2B*	*Dlog2B* +# *matching pages*	*Search* + *BD*	*Search*+ *BD*
Clustered	*1.5BD*	*DlogF1.5B*	*Dlo9F1.5B*+# *matching pages*	*Search* + D	*Search*+ D
Unclustered tree index	*BD*(*R* + 0.15)	D(1 + *logFO.15B*)	*D*(*lo9FO.15B*+# *matching records*)	*D*(3 + *logFO.15B*)	*Search*+ 2D
Unclustered hash index	*BD*(*R* + 0.125)	2D	*BD*	4D	*Search*+ 2D

Figure **5.4** A Comparison of I/O Costs

5.4 TREE-STRUCTURED INDEXING

5.4.0 Introduction: We now consider two index data structures, called **ISAM** and **B+ trees**, based on tree organizations. These structures provide efficient support for range searches, including sorted file scans as a special case.

5.4.1 INDEXED SEQUENTIAL ACCESS METHOD (ISAM): The ISAM data structure is illustrated in Figure 5.5. The data entries of the ISAM index are in the ***leaf pages*** of the tree and additional overflow pages chained to some leaf page. Database systems carefully organize the layout of pages so that page boundaries correspond closely to the physical characteristics of the underlying storage device. The ISAM structure is *completely static* and facilitates such low-level optimizations. Each tree node is a disk page, and all the data resides in the leaf pages.

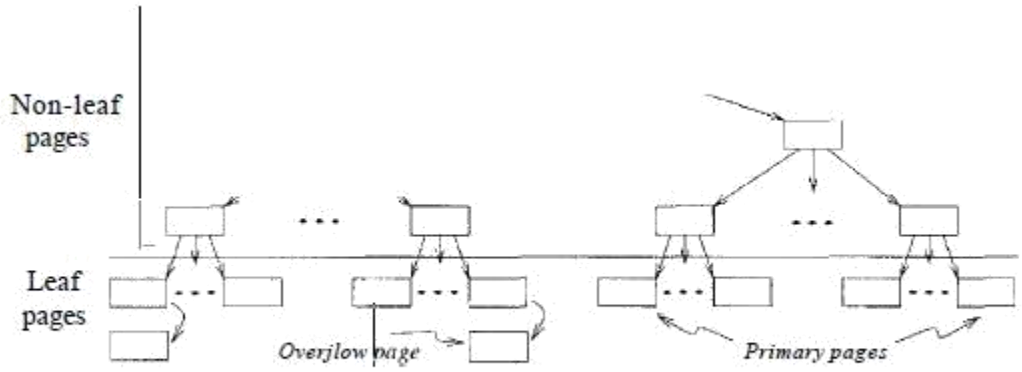


Figure 5.5 ISAM Index Structure

If there are several inserts to the file subsequently, so that more entries are inserted into a leaf than will fit onto a single page, additional pages are needed because the index structure is static. These additional pages are allocated from an ***overflow area***.

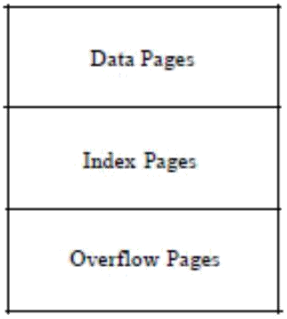


Figure 5.6 Page Allocation in ISAM

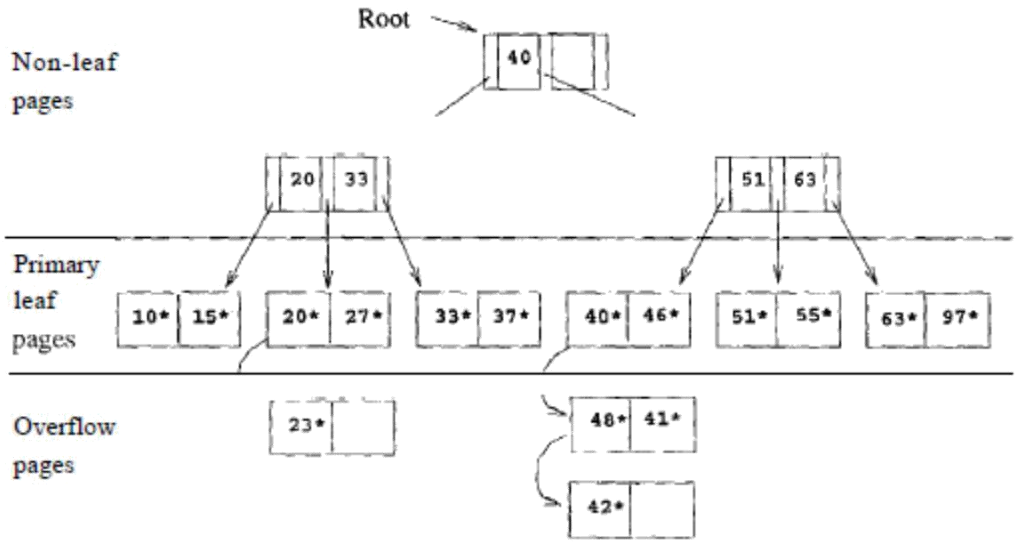


Figure 5.7 ISAM Tree after Inserts

5.4.2 Overflow Pages, Locking Considerations: The fact that only leaf pages are modified also has an important advantage with respect to concurrent access. When a page is accessed, it is typically ***'locked'*** by the requestor to ensure that it is not concurrently modified by other users of the page. To modify a page, it must be locked in ***'exclusive'*** mode, which is permitted only when no one else holds a lock on the page. Locking can lead to ***queues*** of users waiting to get access to a page.

5.5 B+ TREES: A DYNAMIC INDEX STRUCTURE

5.5.0 B+ Tree: A static structure such as the ISAM index suffers from the problem that long overflow chains can develop as the file grows, leading to *poor performance*. This problem motivated the development of *more flexible, dynamic structures* that adjust gracefully to inserts and deletes.

The **B+ tree** search structure, which is widely used, is a balanced tree in which the internal nodes direct the search and the leaf nodes contain the data entries. Since the tree structure *grows and shrinks dynamically.* To retrieve all leaf pages efficiently, we have to link them using *page pointers*. By organizing them into a *doubly linked list*, we can easily traverse the sequence

of leaf pages (sometimes called the *sequence set*) in either direction. This structure is illustrated in

Figure 5.8

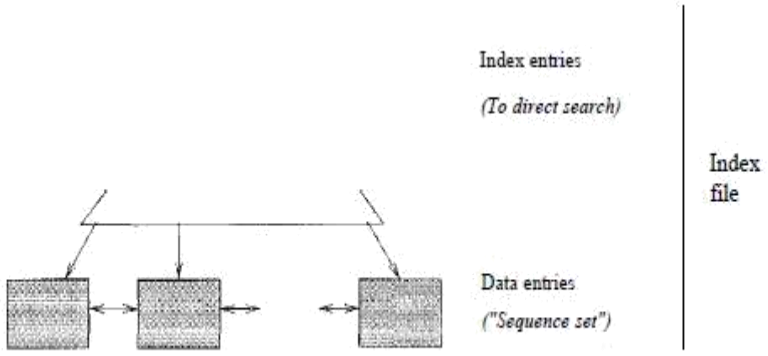


Figure 5.8 Structure of a B+ Tree

The following are some of the main characteristics of a B+ tree:

- Operations (insert, delete) on the tree keep it balanced.
- A minimum occupancy of 50 percent is guaranteed for each node except the root.
- Searching for a record requires just a traversal from the root to the appropriate leaf.

For B+ trees every node contains **m** entries, where d <=m<=2d. The value **d** is a parameter of the B+ tree, called the order of the tree, and is a measure of the capacity of a tree node. The root node is the only exception.

B+ trees are usually also preferable to ISAM indexing because inserts are handled gracefully without overflow chains. Two factors favor ISAM: the *leaf pages are allocated in sequence*, and the locking overhead of ISAM is lower than that for B+ trees. As a general rule, however, B+ trees are likely to perform better than ISAM.

Format of node: Every node contains P pointer and K key value which will be pointing to the actual data. Non-leaf nodes with **m** 'index entries contain **m+1** pointers to children. Pointer Pi points to a subtree in which all key values K are such that Ki < K < Ki+1.

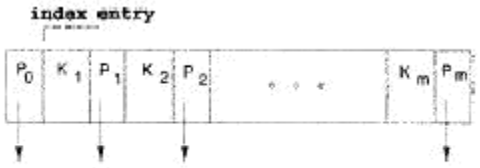


Figure 5.9 Format of an Index Page

5.5.1 SEARCH: The algorithm for search finds the leaf node in which a given data entry belongs. We use the notation *ptr to denote the value pointed to by a pointer variable ptr and & (value) to denote the address of value. Note that finding i in *tree_search* requires us to search within the node, which can be done with either a linear search or a binary search (e.g., depending on the number of entries in the node). In discussing the search, insertion, and deletion algorithms for B+ trees, we assume that there are no duplicates. Refer figure 5.10 for searching algorithm.

```
fune find (search key value K) returns nodepointer
// Given a search key value, finds its leaf node
return tree_search(root, K);                    // searches from root
endfune
```

```
fune tree_search (nodepointer, search key value K) returns nodepointer
// Searches tree for entry
if *nodepointer is a leaf, return nodepointer;
else,
    if K < K_1 then return tree_search(P_0, K);
    else,
        if K ≥ K_m then return tree_search(P_m, K);    // m = # entries
        else,
            find i such that K_i ≤ K < K_{i+1};
            return tree_search(P_i, K)
endfune
```

Figure 5.10 Algorithm for Search

Consider the sample B+ tree shown in Figure 5.11. This B+ tree is of order d=2. That is, each node contains between 2 and 4 entries. Each non--leaf entry is a <*key value,' Nodepointer*> pair; at the leaf level, the entries are data records that we denote by k*.

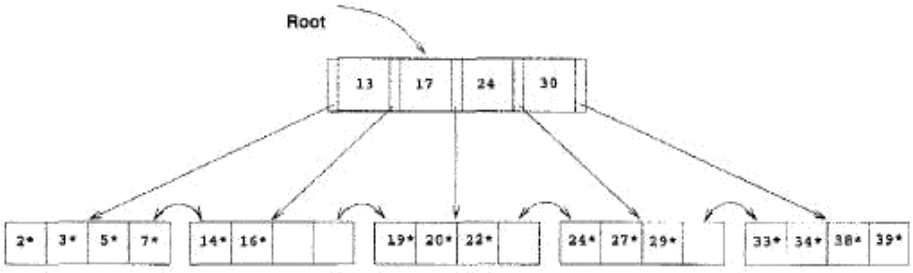


Figure 5.11 Example of a B+ Tree, Order d=2

5.5.2 INSERT: The algorithm

for insertion takes an entry, finds the leaf node where it belongs, and inserts it there. Pseudocode for the B+ tree insertion algorithm is given in Figure 5.12. The basic idea behind the algorithm is that we recursively insert the entry by calling the insert algorithm on the appropriate child node. Usually, this procedure results in going down to the leaf node where the entry belongs, placing the entry there, and returning all the way back to the root node. Occasionally a node is full and it must be split. When the node is split, an entry pointing to the node created by the split must be inserted into its parent; this entry is pointed to by the pointer variable *newchildentry*. If the (old) root is split, a new root node is created and the height of the tree increases by 1.

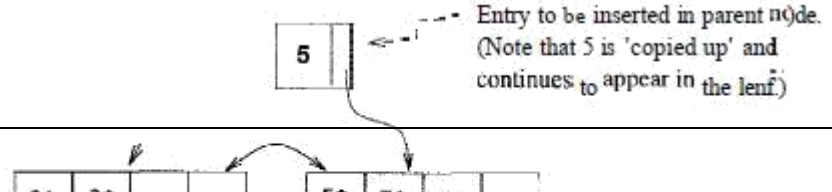
```
proc insert (nodepointel', entry, newchildentry)
// Inselts entry into subtree with TOot '*nodepointer'; degree is d;
/'/'newchildentry' null initially, and null on return unless child is split

if *nodepointer is a non-leaf node, say N,
    find'i such that K_i ≤ entry's key value < J(i+1;    // choose subtree
    insert(P_i, entry, newchildentry);          // recurs'ively, insert entry
    if newchildentry is null, return;        // usual case; didn't split child
    else,                // we split child, must insert *newchildentry in N
        if N has space,                                        // usual case
            put *newchildentry on it, set newchildentry to null, return;
        else,                  // note difference wrt splitting of leaf page!
            split N:        // 2d + 1 key values and 2d + 2 nodepointers
            first d key values and d + 1 nodepointers stay,
            last d keys and d + 1 pointers move to new node, N2;
            // *newchildentry set to guide searches between Nand N2
            newchildentry = & ((smallest key value on N2,
                                pointer to N2));
            if N is the root,                  // root node was just split
                create new node with (pointer to N, *newchildentry);
                make the tree's root-node pointer point to the new node;
            return;

if *nodepointer is a leaf node, say L,
    if L has space,                                            // usual case
    put entry on it, set newchildentry to null, and return;
    else,                                    // once in a while, the leaf is full
        split L: first d entries stay, rest move to brand new node L2;
        newchildentry = & ((smallest key value on L2, pointer to L2));
        set sibling pointers in Land L2;
        return;
endproc
```

Figure 5.12 Algorithm for Insertion into B+ Tree of Order *d*

In the following example when a value 8 is entered into the leaf node the value 5 is pushed up to its root node.



.5.3 DELETE: The algorithm for deletion takes an entry, finds the leaf node where it belongs, and deletes it. Pseudocode for the B+ tree deletion algorithm is given in Figure 5.14. The basic idea behind the algorithm is that we recursively delete the entry by calling the delete algorithm on the appropriate child node. We usually go down to the leaf node where the entry belongs, remove the entry from there, and return all the way back to the root node. If entries are redistributed between two nodes, their parent node must be updated to reflect this; the key value in the index entry pointing to the second node must be changed to be the lowest search key in the second node. If two nodes are merged, their parent must be updated to reflect this by deleting the index entry for the second node; this index entry is pointed to by the pointer variable *oldchildentry* when the delete call returns to the parent node. If the last entry in the root node is deleted in this manner because one of its children was deleted, the height of the tree decreases by 1.

```
proc delete (parentpointer, nodepointer, entry, oldchildentry)
// Deletes entry from subtree with root '*nodepointer'; degree is d;
// 'oldchildentry' null initially, and null upon return unless child deleted
if *nodepointer is a non-leaf node, say N,
    find i such that Ki ≤ entry's key value < Ki+1;    // choose subtree
    delete(nodepointer, Pi, entry, oldchildentry);     // recursive delete
    if oldchildentry is null, return;          // usual case: child not deleted
    else,                                  // we discarded child node (see discussion)
        remove *oldchildentry from N,          // next, check for underflow
        if N has entries to spare,                                  // usual case
            set oldchildentry to null, return; // delete doesn't go further
        else,                  // note difference wrt merging of leaf pages!
            get a sibling S of N:      // parentpointer arg used to find S
            if S has extra entries,
                redistribute evenly between N and S through parent;
                set oldchildentry to null, return;
            else, merge N and S                          // call node on rhs M
                oldchildentry = & (current entry in parent for M);
                pull splitting key from parent down into node on left;
                move all entries from M to node on left;
                discard empty node M, return;

if *nodepointer is a leaf node, say L,
    if L has entries to spare,                                      // usual case
        remove entry, set oldchildentry to null, and return;
    else,                      // once in a while, the leaf becomes underfull
        get a sibling S of L;                  // parentpointer used to find S
        if S has extra entries,
            redistribute evenly between L and S;
            find entry in parent for node on right;            // call it M
            replace key value in parent entry by new low-key value in M;
            set oldchildentry to null, return;
        else, merge L and S                              // call node on rhs M
            oldchildentry = & (current entry in parent for M);
            move all entries from M to node on left;
            discard empty node M, adjust sibling pointers, return;
endproc
```

Figure 5.14 Algorithm for Deletion from B+ Tree of Order d

5.5.4 DUPLICATES: The basic search algorithm assumes that all entries with a given key value reside on a single leaf page. One way to satisfy this assumption is to *use overflow pages to deal with duplicates*. Typically, however, we use an alternative approach for duplicates. We handle them just like any other entries and several leaf pages may contain entries with a given key value. To retrieve all data entries with a given key value, we must search for the left-most data entry with the given key value and then possibly retrieve more than one leaf page (using the leaf sequence pointers). Modifying the search algorithm to find the left-most data entry in an index with duplicates is performed.

5.5.5 Bulk-Loading a B+ Tree: When there is a *collection of data records*, systems provide a *bulk-loading utility* for creating a B+ tree index on an existing collection of data records. Thefirst step is to *sort* the data entries k* to be inserted into the (to be created) B+ tree according to the search key k.

In the following example we assume that each data page can hold only two entries, and that each index page can hold two entries and an additional pointer (i.e., the B+ tree is assumed to be of order d = 1). After the data entries have been sorted, we allocate an empty page to serve as the root and insert a pointer to the first page of (sorted) entries into it. We illustrate this process in Figure 5.15.

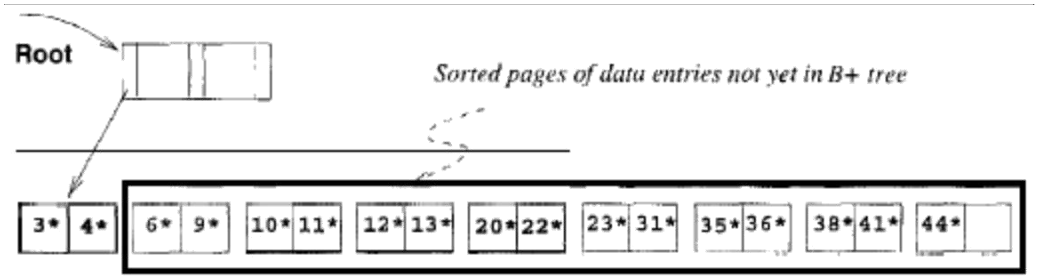


Figure 5.15 Initial Step in B+ Tree Bulk-Loading

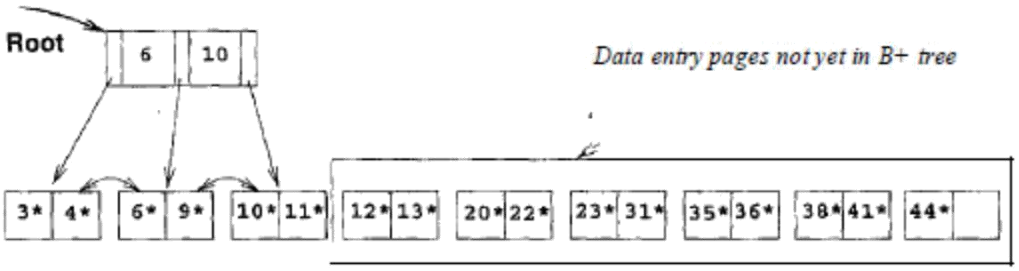
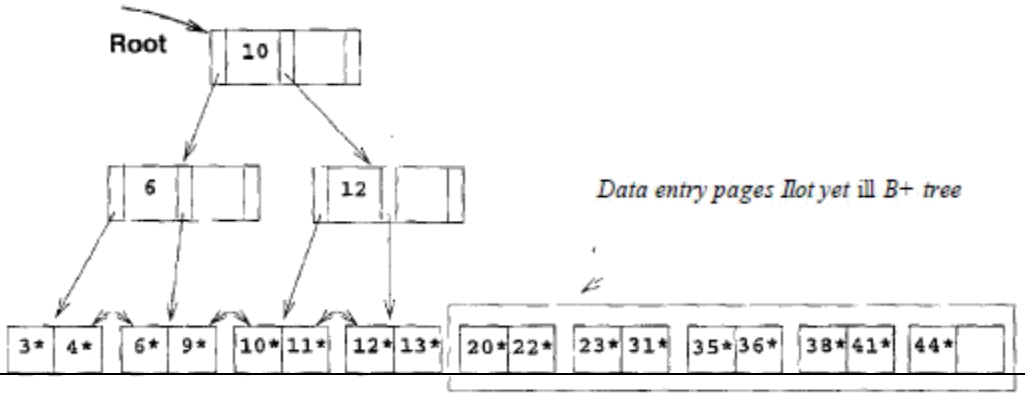


Figure 5.16 Root Page Fills up in B+ Tree Bulk-Loading



5.6 HASH-BASED INDEXING

5.6.0 Hashing Function: The basic idea is to use a *hashing function*, which maps values in a search field into a range of *bucket numbers* to find the page on which a desired data entry belongs. We use a simple scheme called ***Static Hashing*** to introduce the idea. This scheme, like ISAM, suffers from the problem of long overflow chains, which can affect performance. Two solutions to the problem are presented.

- The *Extendible Hashing* scheme uses a *directory* to support inserts and deletes efficiently with no overflow pages.
- The *Linear Hashing* scheme uses a clever policy for creating *new buckets* and supports inserts and deletes efficiently without the use of a directory.

5.6.1 STATIC HASHING: The Static Hashing scheme is illustrated in Figure 5.18. The pages containing the data can be viewed as a collection of buckets, with one primary page and possibly additional overflow pages per bucket. A file consists of buckets a through N - 1, with one primary page per bucket initially. Buckets contain data entries, which can be any of the three alternatives.

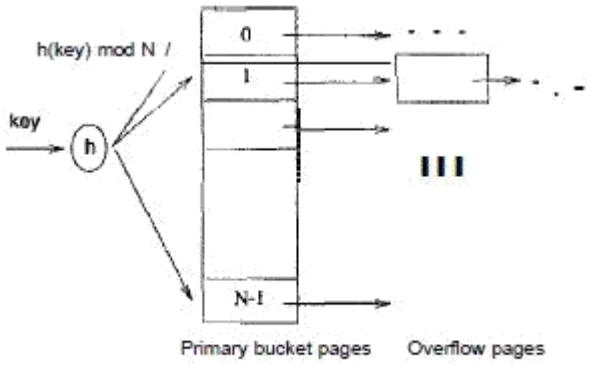


Figure 5.18 Static Hashing

To search for a data entry, we apply a hash function *h* to identify the bucket to which it belongs and then search this bucket. To speed the search of a bucket, we can maintain data entries in sorted order by search key value. To *insert a data entry*, we use the hash function to identify the correct bucket and then put the data entry there. If there is no space for this data entry, we allocate a new *overflow page*, put the data entry on this page, and add the page to the *overflow chain* of the bucket. To delete a data entry, we use the hashing function to identify the correct bucket, locate the data entry by searching the bucket, and then remove it. If this data entry is the last in an overflow page, the overflow page is removed from the overflow chain of the bucket and added to a list of free pages.

The hash function is an important component of the hashing approach. It must distribute values in the domain of the search field uniformly over the collection of buckets. If we have N buckets, numbered a through *N - 1*, a hash function *h* of the form *h(value) = (a * value +b)* works well in practice. (The bucket identified is h(value) mod N.) The constants a and b can be chosen to 'tune' the hash function.

Since the number of buckets in a Static Hashing file is known when the file is created, the primary pages can be stored on successive disk pages. Hence, a search ideally requires just one disk I/O, and insert and delete operations require two I/Os (read and write the page), although

The main problem with Static Hashing is that the number of buckets is fixed. If a file shrinks greatly, a lot of space is wasted; more important, if a file grows a lot, long overflow chains develop, resulting in poor performance.

Another alternative is to use **dynamic hashing techniques** such as *Extendible and Linear Hashing*, which deal with inserts and deletes gracefully.

5.7 Dynamic Hashing Techniques - EXTENDIBLE HASHING

In Static Hashing the performance degrades with overflow pages. This problem, however, can be overcome by a simple idea: Use a directory of pointers to buckets, and double the size of the number of buckets by doubling just the directory and splitting only the bucket that overflowed which the central concept of Extendible is hashing.

To understand the idea, consider the sample file shown in Figure 5.19. The directory consists of an array of size 4, with each element being a pointer to a bucket (global depth). To locate a data entry, we apply a hash function to the search field and take the last 2 bits of its binary representation to get a number between 0 and 3. The pointer in this array position gives us the desired bucket; we assume that each bucket can hold four data entries. Therefore, to locate a data entry with hash value 5 (binary 101), we look at directory element 01 and follow the pointer to the data page (bucket B in the figure). To insert. a dat.a entry, we search to find the appropriate bucket.. For example, to insert a data entry with hash value 13 (denoted as 13*), we examine directory element 01 and go to the page containing data entries 1*, 5*, and 21 *. Since the page has space for an additional data entry, we are done after we insert the entry (Figure 5.20).

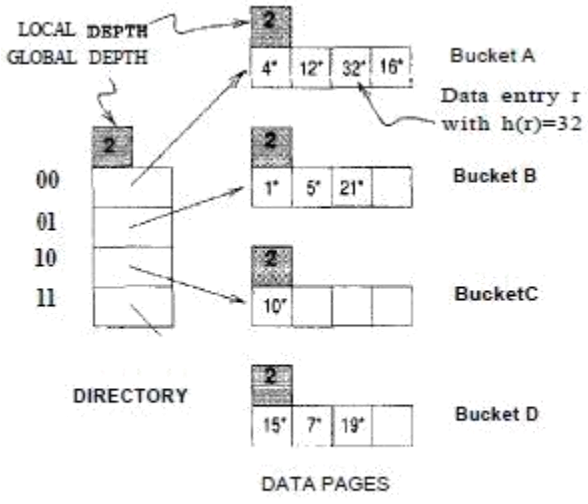


Figure 5.19 Example of an Extendible Hashed File

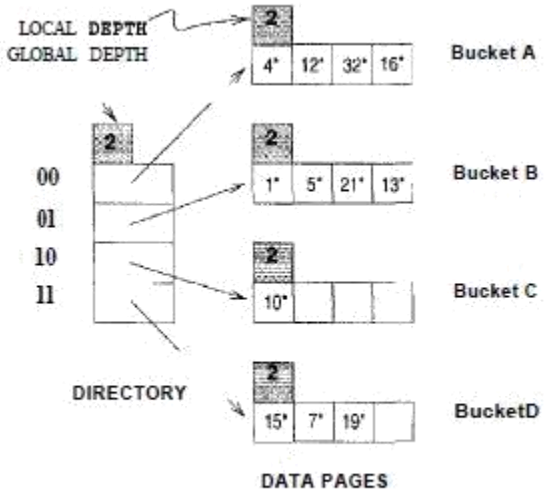


Figure 5.20 After Inserting Entry r with $h(r) = 13$

Next, let us consider insertion of a data entry into a full bucket. The essence of the Extendible Hashing idea lies in how we deal with this case. Consider the insertion of data entry 20* (binary 10100). Looking at directory clement 00, we arc led to bucket A, which is already full. We just first split the bucket by allocating a new bucket and redistributing the contents (including the new entry to be inserted) across the old bucket and its 'split image.' To redistribute entries across the old bucket and its split image, we consider the last three bits of h(T); the last two bits are 00, indicating a data entry that belongs to one of these two buckets, and the third bit discriminates between these buckets. The redistribution of entries is illustrated in Figure 5.21

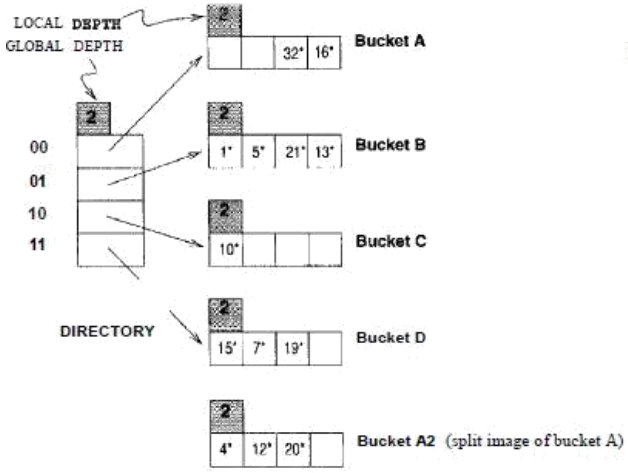


Figure 5.21 While Inserting Entry r with $h(r)=20$

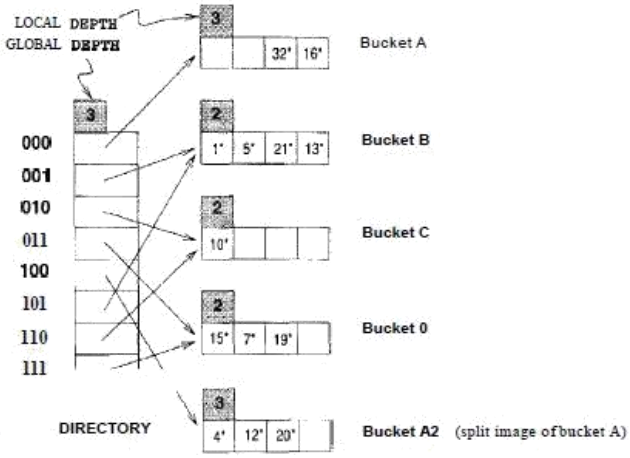


Figure 5.22 After Inserting Entry r with $h(r) = 20$

Therefore, doubling the file requires allocating a new bucket page, writing both this page and the old bucket page that is being split, and doubling the directory array. The directory is likely to be much smaller than the file itself because each element is just a page-id, and can be doubled by simply copying it over The cost of doubling is now quite acceptable.

We observe that the basic technique used in Extendible Hashing is to treat the result of applying a hash function h a" a binary number and interpret the last d bits, where d depends on the size of the directory, as an offset into the directory.

In our example, d is originally 2 because we only have four buckets; after the split, d becomes 3 because we now have eight buckets. A corollary is that, when distributing entries across a bucket and its split image, we should do so on the basis of the dth bit. (Note how entries are redistributed in our example; see Figure 5.22) The number d, called the global depth of the hashed file, is kept as part of the header of the file. It is used every time we need to locate a data entry. An important point that arises is whether *splitting a bucket necessitates a directory doubling*.

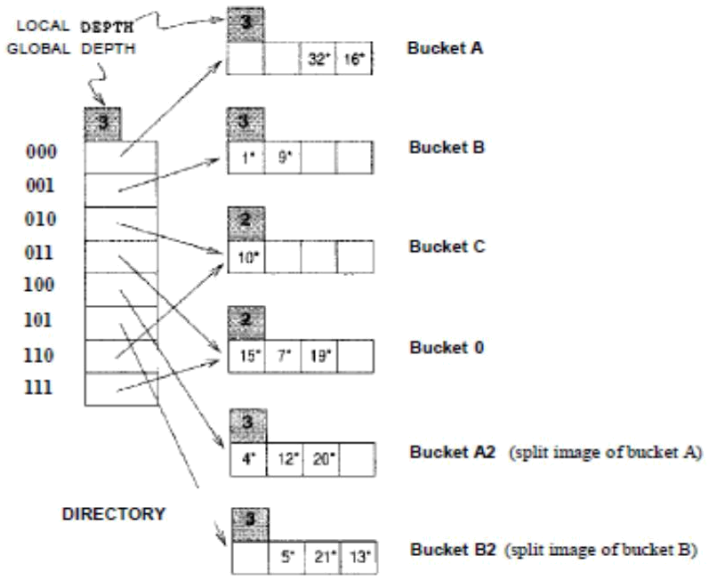


Figure 5.23 After Inserting Entry r with $h(r) = 9$

To differentiate between these cases and determine whether a directory doubling is needed, we maintain a local depth for each bucket. If a bucket whose local depth is equal to the global depth is split, the directory must be doubled. Initially, all local depths are equal to the global depth (which is the number of bits needed to express the total number of buckets). We increment the global depth by 1 each time the directory doubles, of course.

On the other hand, the directory grows in spurts and can become large for skewed data distributions. Even if the distribution of search values is skewed, the choice of a good hashing function typically yields a fairly uniform distribution of hash values; skew is therefore not a problem in practice.

5.8 LINEAR HASHING

Linear Hashing is a dynamic hashing technique, it does not require a directory, deals naturally with collisions, and offers a lot of flexibility with respect to the timing of bucket splits. If the data distribution is very skewed, however, overflow chains could cause Linear Hashing performance to be worse than that of Extendible Hashing.

The scheme utilizes a family of hash functions *h0, h1, h2, ...* , with the property that each function's range is *twice* that of its predecessor. That is, if *hi* maps a data entry into one of M buckets, hi+1 maps a data entry into one of M buckets. Such a family is typically obtained by choosing a hash function hand an initial number N ofbuckets,2 and defining *hi(value) = h(value) mod (2i N)*.

The idea is best understood in terms of rounds of splitting. During round number Level, only hash functions *hLevel* and *hLevel+1* are in use. The buckets in the file at the beginning of the round are split, one by one from the first to the last bucket, thereby doubling the number of buckets. At any given point within a round, therefore, we have buckets that have been split, buckets that are yet to be split, and buckets created by splits in this round.

Consider how we search for a data entry with a given search key value. We apply hash function *h Level* , and if this leads us to one of the unsplit buckets, we simply look there. If it leads us to one of the split buckets, the entry may be there or it may have been moved to the new bucket created earlier in this round by splitting this bucket; to determine which of the two buckets contains the entry, we apply *hLevel+1*.

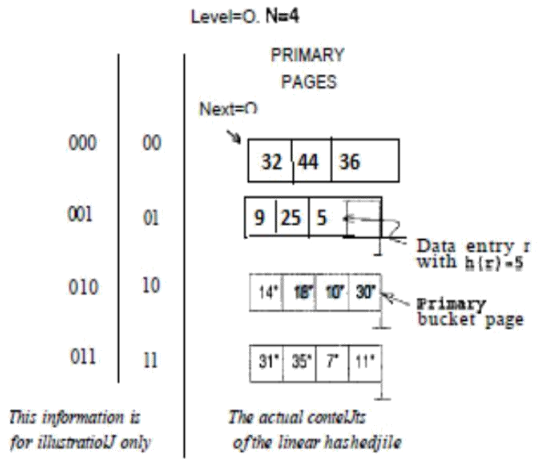


Figure 5.24 Example of a Linear Hashed File

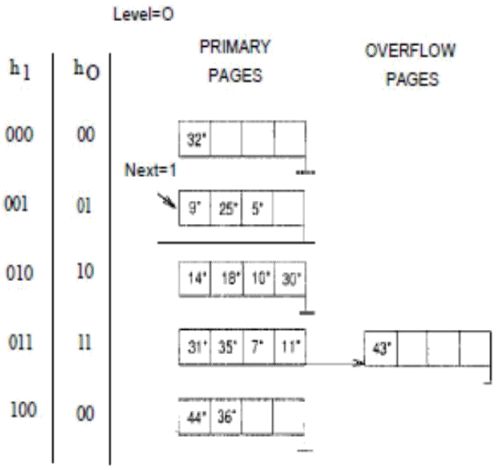


Figure 5.25 After Inserting Record r with $h(T) = 43$

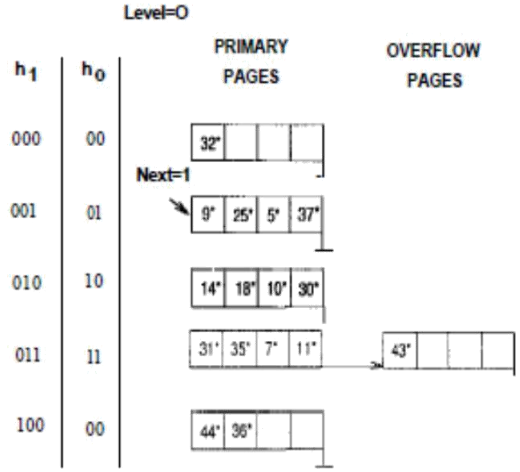


Figure 5.26 After Inserting Record r with $h(r) = 37$

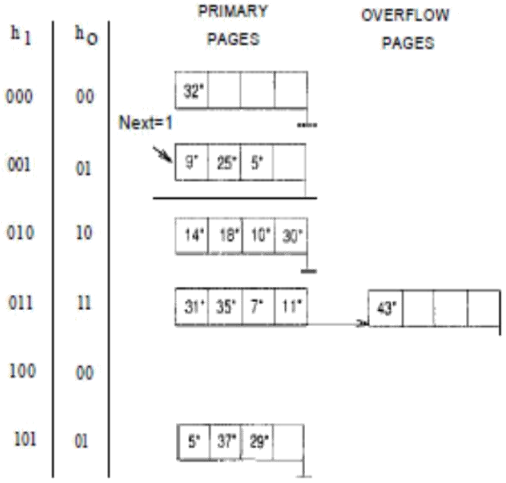


Figure 5.27 After Inserting Record r with $h(r') = 29$

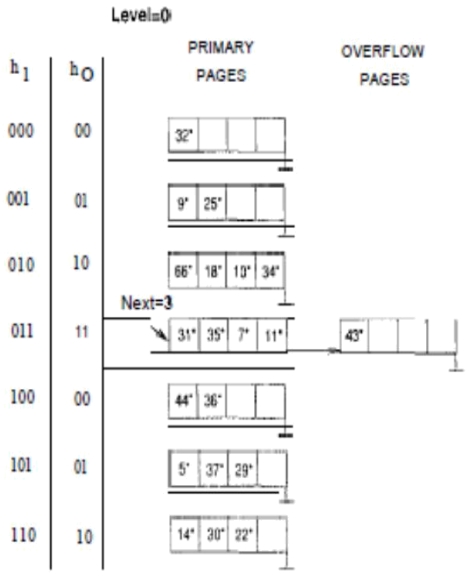


Figure 5.28 After Inserting Records with $h(r) = 22, 66, and 34$

5.8.1 EXTENDIBLE VS. LINEAR HASHING

To understand the relationship between Linear Hashing and Extendible Hashing, imagine that we also have a directory in Linear Hashing with elements 0 to N - 1. The first split is at bucket 0, and so we add directory element N. In principle, we may imagine that the entire directory has been doubled at this point; however, because element 1 is the same as element N + 1, elernent 2 is the same the element N + 2, and so on, we can avoid the actual copying for the rest of the directory. The second split occurs at bucket 1; now directory element N + 1 becomes significant and is added. At the end of the round, all the original N buckets are split, and the directory is doubled in size (because all elements point to distinct buckets).

We observe that the choice of hashing functions is actually very similar to what goes on in Extendible Hashing---in effect, moving from hi to hi+1 in Linear Hashing corresponds to doubling the directory in Extendible Hashing. Both operations double the effective range into which key values are hashed; but whereas the directory is doubled in a single step of Extendible Hashing, moving from hi to hi+l, along with a corresponding doubling in the number of buckets, occurs gradually over the course of a round in Linear Hashing.

The new idea behind Linear Hashing is that a directory can be avoided by a clever choice of the bucket to split. On the other hand, by always splitting the appropriate bucket, Extendible Hashing may lead to a reduced number of splits and higher bucket occupancy.

The directory analogy is useful for understanding the ideas behind Extendible and Linear Hashing. However, the directory structure can be avoided for Linear Hashing (but not for Extendible Hashing) by allocating primary bucket pages consecutively, which would allow us to locate the page for bucket i by a simple offset calculation. For uniform distributions, this implementation of Linear Hashing has a lower average cost for equality selections. For skewed distributions, this implementation could result in any empty or nearly empty buckets, each of which is allocated at least one page, leading to poor performance relative to Extendible Hashing, which is likely to have higher bucket occupancy.A different implementation of Linear Hashing, in which a directory is actually maintained, offers the flexibility of not allocating one page per bucket; null directory elements can be used as in Extendible Hashing. However, this implementation introduces the overhead of a directory level and could prove costly for large, uniformly distributed files.