UNIT – I

RELATIONAL DATABASE-MANAGEMENT SYSTEM

File Management System

A **file system** (or **file system**) is an abstraction to store, retrieve and update a set of files. The term also identifies the data structures specified by some of those abstractions, which are designed to organize multiple files as a single stream of bytes, and the network protocols specified by some other of those abstractions, which are designed to allow files on a remote machine to be accessed.

The file system manages access to the data and the metadata of the files, and manages the available space of the device(s) which contain it. Ensuring reliability is a major responsibility of a file system. A file system organizes data in an efficient manner, and may be tuned to the characteristics of the backing device.

FILENAMES

A **filename** (or **file name**) is used to identify a storage location in the file system. Most file systems have restrictions on the length of filenames. In some file systems, filenames are case-insensitive (i.e., filenames such as FOO and foo refer to the same file); in others, filenames are case-sensitive (i.e., the names FOO and foo refer to two separate files).

Most modern file systems allow filenames to contain a wide range of characters from the Unicode character set. Most file system interface utilities, however, have restrictions on the use of certain special characters, disallowing them within filenames (the file system may use these special characters to indicate a device, device type, directory prefix, or file type).

DIRECTORIES

File systems typically have **directories** (also called **folders**) which allow the user to group files into separate collections. This may be implemented by associating the file name with an index in a table of contents or an inode in a Unix-like file system. Directory structures may be flat (i.e. linear), or allow hierarchies where directories may contain subdirectories. The first file system to support arbitrary hierarchies of directories was used in the Multics operating system.

METADATA

The length of the data contained in a file may be stored as the number of blocks allocated for the file or as a byte count. The time that the file was last modified may be stored as the file's timestamp. File systems might store the file creation time, the time it was last accessed, the time the file's meta-data was changed, or the time the file was last backed up. Other information can include the file's device type (e.g. block, character, socket, subdirectory, etc.), its owner user ID and group ID, its access permissions and other file attributes (e.g. whether the file is read-only, executable, etc.).

TYPES OF FILE SYSTEMS

File system types can be classified into disk/tape file systems, network file systems and special-purpose file systems.

Disk file systems

A *disk file system* takes advantages of the ability of disk storage media to randomly address data in a short amount of time. Additional considerations include the speed of accessing data following that initially requested and the anticipation that the following data may also be requested. This permits multiple users (or processes) access to various data on the disk without regard to the sequential location of the data.

Examples include

FAT – File Allocation Table (FAT12, FAT16, FAT32), exFAT,

NTFS (NT **file system**; sometimes New Technology **File System**),

Optical discs

ISO 9660 and Universal Disk Format (UDF) are two common formats that target Compact Discs, DVDs and Blu-ray discs. Mount Rainier is an extension to UDF supported by Linux 2.6 series and Windows Vista that facilitates rewriting to DVDs.

Flash file systems

A *flash file system* considers the special abilities, performance and restrictions of flash memory devices. Frequently a disk file system can use a flash memory device as the underlying storage media but it is much better to use a file system specifically designed for a flash device.

Tape file systems

A *tape file system* is a file system and tape format designed to store files on tape in a self-describing form. Magnetic tapes are sequential storage media with significantly longer random data access times than disks, posing challenges to the creation and efficient management of a general-purpose file system.

In a disk file system there is typically a master file directory, and a map of used and free data regions. Any file additions, changes, or removals require updating the directory and the used/free maps. Random access to data regions is measured in milliseconds so this system works well for disks.

Tape requires linear motion to wind and unwind potentially very long reels of media. This tape motion may take several seconds to several minutes to move the read/write head from one end of the tape to the other.

Consequently, a master file directory and usage map can be extremely slow and inefficient with tape. Writing typically involves reading the block usage map to find free blocks for writing, updating the usage map and directory to add the data, and then advancing the tape to write the data in the correct spot. Each additional file write requires updating the map and directory and writing the data, which may take several seconds to occur for each file.

Tape file systems instead typically allow for the file directory to be spread across the tape intermixed with the data, referred to as *streaming*, so that time-consuming and repeated tape motions are not required to write new data.

HIERARCHY OF DATA

Data are the principal resources of an organization. Data stored in computer systems form a hierarchy extending from a single bit to a database, the major record-keeping entity of a firm. Each higher rung of this hierarchy is organized from the components below it.

Data are logically organized into:

Bit (Character) - a bit is the smallest unit of data representation (value of a bit may be a 0 or 1). Eight bits make a byte which can represent a character or a special symbol in a character code.

Field - a field consists of a grouping of characters. A data field represents an attribute (a characteristic or quality) of some entity (object, person, place, or event).

Record - a record represents a collection of attributes that describe a real-world entity. A record consists of fields, with each field describing an attribute of the entity.

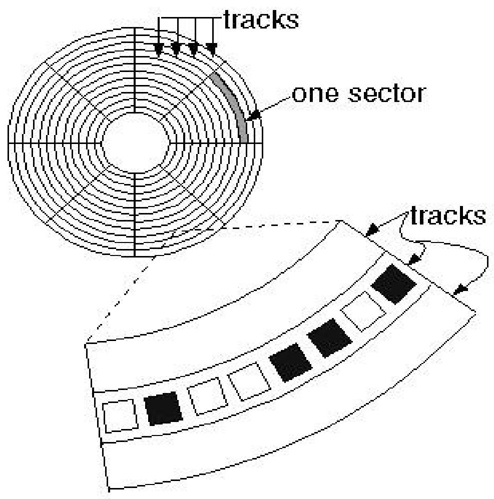
File - a group of related records. Files are frequently classified by the application for which they are primarily used (employee file). A ***primary key*** in a file is the field (or fields) whose value identifies a record among others in a data file.

Magnetic disk

The primary computer storage device. Like tape, it is magnetically recorded and can be re-recorded over and over. Disks are rotating platters with a mechanical arm that moves a read/write head between the outer and inner edges of the platter's surface. It can take as long as one second to find a location on a floppy disk to as little as a couple of milliseconds on a fast hard disk. See hard disk for more details.

Tracks and Spots

The disk surface is divided into concentric tracks (circles within circles). The thinner the tracks, the more storage. The data bits are recorded as tiny magnetic spots on the tracks. The smaller the spot, the more bits per inch and the greater the storage.



Sectors

Tracks are further divided into sectors, which hold a block of data that is read or written at one time; for example, READ SECTOR 782, WRITE SECTOR 5448. In order to update the disk, one or more sectors are read into the computer, changed and written back to disk. The operating system figures out how to fit data into these fixed spaces.Modern disks have more sectors in the outer tracks than the inner ones because the outer radius of the platter is greater than the inner radius

Magnetic tape

A sequential storage medium used for data collection, backup and archiving. Like videotape, computer tape is made of flexible plastic with one side coated with a ferromagnetic material. Tapes were originally open reels, but were superseded by cartridges and cassettes of many sizes and shapes.

Tape has been more economical than disks for archival data, but that is changing as disk capacities have increased enormously. If tapes are stored for the duration, they must be periodically recopied or the tightly coiled magnetic surfaces may contaminate each other.

Sequential Medium

The major drawback of tape is its sequential format. Locating a specific record requires reading every record in front of it or searching for markers that identify predefined partitions. Although most tapes are used for archiving rather than routine updating, some drives allow rewriting in place if the byte count does not change. Otherwise, updating requires copying files from the original tape to a blank tape (scratch tape) and adding the new data in between.

FILE ORGANIZATION

Data files are organized so as to facilitate access to records and to ensure their efficient storage. A tradeoff between these two requirements generally exists: if rapid access is required, more storage is required to make it possible.

Access to a record for reading it is the essential operation on data. There are two types of access:

Sequential access - is performed when records are accessed in the order they are stored. Sequential access is the main access mode only in batch systems, where files are used and updated at regular intervals.

Direct access - on-line processing requires direct access, whereby a record can be accessed without accessing the records between it and the beginning of the file. The primary key serves to identify the needed record.

There are three methods of file organization:

Sequential organization
Indexed-sequential organization
Direct organization

FILE SYSTEMS VERSUS A DBMS:

To understand the need for a DBMS, let us consider a motivating scenario: A company has a large collection (say, 500 GB) of data on employees, departments, products, sales, and so on. This data is accessed concurrently by several employees. Questions about the data must be answered quickly, changes made to the data by different users must be applied consistently, and access to certain parts of the data (e.g., salaries) must be restricted.

We can try to deal with this data management problem by storing the data in a collection of operating system files. This approach has many drawbacks, including the following:

- We probably do not have 500 GB of main memory to hold all the data. We must therefore store data in a storage device such as a disk or tape and bring relevant parts into main memory for processing as needed. Even if we have 500 GB of main memory, on computer systems with 32-bit addressing, we cannot refer directly to more than about 4 GB of data! We have to program some method of identifying all data items.
- We have to write special programs to answer each question that users may want to ask about the data. These programs are likely to be complex because of the large volume of data to be searched. We must protect the data from inconsistent changes made by different users accessing the data concurrently. If programs that access the data are written with such concurrent access in mind, this adds greatly to their complexity. We must ensure that data is restored to a consistent state if the system crashes while changes are being made. Operating systems provide only a password mechanism for security. This is not sufficiently flexible to enforce security policies in which different users have permission to access different subsets of the data.

A DBMS is a piece of software that is designed to make the preceding tasks easier. By storing data in a DBMS, rather than as a collection of operating system files, we can use the DBMS's features to manage the data in a robust and efficient manner. As the volume of data and the number of users grow hundreds of gigabytes of data and thousands of users are common in current corporate databases DBMS support becomes indispensable.

INTRODUCTION to DBMS

What is Data ?

Data is stored raw facts (or real world facts) that can be processed for any computing machine.

Data is collection of facts, which is in unorganized but they can be organized into useful form. Data may be numerical data which may be integers or floating point numbers and non-numerical data such as characters, date and etc., Data is of two types :

Raw data : it is a data which are collected from different sources and has no meaning.
Derived data : it is a data that are extracted from raw data and used for getting useful information.

Example: The above numbers may be anything: It may be distance in kms or amount in rupees or no of days or marks in each subject etc.,

Information:

Information is data that has been converted into more useful or intelligible form. Example is Student Mark Sheet.

Information is RELATED DATA. The data (information) which is used by an organization – a college, a library, a bank, a manufacturing company – is one of its most valuable resources.

Knowledge:

Human mind purposefully organized the information and evaluate it to produce knowledge.

Example: 238 is a data,
Marks of student is information and
The hard work require to get mark is knowledge.

1. Fact based Knowledge :
   It is knowledge gain from fundamental and through experiment.
   The result is guaranteed.

2. Heuristic based Knowledge:
   It is the knowledge of good practice and good judgment like hypothesis.
   The result is not guaranteed.

Database :

Databases and database systems have become an essential component of everyday life in modern society. In the course of a day, most of us encounter several activities that involve some interaction with a database.

For example, if we go to the bank to deposit or withdraw funds, if we make a hotel or airline reservation, if we access a computerized library catalog to search for a bibliographic item, or if we buy some item-such as a book, toy, or computer-from an Internet vendor through its Web page, chances are that our activities will involve someone or some computer program accessing a database. Even purchasing items from a supermarket nowadays in many cases involves an automatic update of the database that keeps the inventory of supermarket items.

- These interactions are examples of what we may call **traditional database applications**, in which most of the information that is stored and accessed is either textual or numeric.
- In the past few years, advances in technology have been leading to exciting new applications of database systems. **Multimedia** databases can now store pictures, video clips, and sound messages.
- **Geographic information systems** (CIS) can store and analyze maps, weather data, and satellite images.
- **Data warehouses and online analytical processing (OLAP)** systems are used in many companies to extract and analyze useful information from very large databases for decision making.
- **Real-time and active database technology** is used in controlling industrial and manufacturing processes.
- And **database search techniques** are being applied to the World Wide Web to improve the search for information that is needed by users browsing the Internet.

Databases and database technology are having a major impact on the growing use of computers. It is fair to say that databases play a critical role in almost all areas where computers are used, including business, electronic commerce, engineering, medicine, law, education, and library science, to name a few.

Database

A database is a collection of related data. By data, we mean known facts that can be recorded and that have implicit and useful meaning.

For example, consider the names, telephone numbers, and addresses of the people you know. You may have recorded this data in an indexed address book, or you may have stored it on a hard drive, using a personal computer and software such as Microsoft Access, or Excel. This is a collection of related data with an implicit meaning and hence is a database.

Characteristics of the database in the DBMS:

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

Database-Management System:

A **database-management system (DBMS)** is a collection of interrelated data and a set of programs to access those data. The collection of data, usually referred to as the **database**, contains information relevant to an enterprise. The primary goal of a DBMS is to provide a way to store and retrieve database information that is both *convenient* and *efficient*.

Database systems are designed to manage large bodies of information. Management of data involves both defining structures for storage of information and providing mechanisms for the manipulation of information.

In addition, 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.

There are several Database Management Systems (DBMS), such as:

- Microsoft SQL Server
- Oracle
- Sybase
- DBase
- Microsoft Access
- MySQL from Sun Microsystems (Oracle)
- DB2 from IBM etc.

What is the need of DBMS?

Database systems are basically developed for large amount of data. When dealing with huge amount of data, there are two things that require optimization: **Storage of data** and **retrieval of data**.

Storage: According to the principles of database systems, the data is stored in such a way that it acquires lot less space as the redundant data (duplicate data) has been removed before storage. Let's take a layman example to understand this:

In a banking system, suppose a customer is having two accounts, one is saving account and another is salary account. Let's say bank stores saving account data at one place (these places are called tables we will learn them later) and salary account data at another place, in that case if the customer information such as customer name, address etc. are stored at both places then this is just a wastage of storage (redundancy/ duplication of data), to organize the data in a better way the information should be stored at one place and both the accounts should be linked to that information somehow. The same thing we achieve in DBMS.

Fast Retrieval of data: Along with storing the data in an optimized and systematic manner, it is also important that we retrieve the data quickly when needed. Database systems ensure that the data is retrieved as quickly as possible.

Database-System Applications

Databases are widely used. Here are some representative applications:

Enterprise Information

Sales**:** For customer, product, and purchase information.
Accounting**:** For payments, receipts, account balances, assets and other accounting information.
Human resources**:** For information about employees, salaries, payroll taxes, and benefits, and for generation of paychecks.
Manufacturing**:** For management of the supply chain and for tracking production of items in factories, inventories of items in warehouses and stores, and orders for items.
Online retailers: For sales data noted above plus online order tracking, generation of recommendation lists, and maintenance of online product evaluations.

Banking and Finance

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

Credit card transactions: For purchases on credit cards and generation of monthly statements.

Finance: For storing information about holdings, sales, and purchases of financial instruments such as stocks and bonds; also for storing real-time market data to enable online trading by customers and automated trading by the firm.

Universities: For student information, course registrations, and grades (in addition to standard enterprise information such as human resources and accounting).

Airlines: For reservations and schedule information. Airlines were among the first to use databases in a geographically distributed manner.

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

PURPOSE OF DATA BASE SYSTEMS:

Before DBMS was invented, Information was stored using File Processing System. In this System, data is stored in permanent system files (secondary Storage). Different application programs are written to extract data from these files and to add record to these files. But, there are Number of disadvantages in using File Processing System, to store the data.

One way to keep the information on a computer is to store it in permanent system files. To allow users to manipulate the stored information, the system has a number of application programs that manipulate the organized files. These application programs are written by system programmers in response to the needs of the organizations. New application programs are added to the system as the need arises. Thus, as the time goes more files and more application programs are added to the system. A typical file processing system described above is the system used to store information before the advent of DBMS.

Characteristics of Traditional File Processing System:

- It stores data of an organization in group of files.
- **Files carrying data are independent** on each other.
- **COBOL, C, C++** programming languages were used to design the files.
- Each file contains data for some specific area or department like library, student fees, and student examinations.
- It is **less flexible** and has many limitations.
- It is very difficult to maintain file processing system.
- Any change in one file affects all the files that creates burden on the programmer.
- File in Traditional File Processing Systems are called **flat files**.

Overall, **Traditional File Processing Systems** was good in many cases in compare to manual non computer based system but still it had many disadvantages that were overcome by Data Base Management System.

Keeping the information of an organization in a file processing system has a number of disadvantages, namely

FILE MANAGEMENT SYSTEM PROBLEMS

- **Data Redundancy and Inconsistency:** Since the files and applications programs are created by different programmers over a long period, the various files are likely to have different formats and the programs may be written in several programming languages. Moreover, the same information may be duplicated in several places. This redundancy leads too higher storage and access cost. In addition, it may lead to data inconsistency.
- **Difficulty in Accessing Data:** The file processing system does not allow needed data to be retrieved in a convenient and efficient manner.
- **Data Isolation:** I a file processing system, as the data are scattered in various files, and files may be in different formats. It is very difficult to write new application programs to retrieve the appropriate data.
- **Integrity problems:** The data values stored in the database must satisfy certain types of consistency Constraints (Conditions).

  For example, the minimum balance in a bank account may never fall below an amount of Rs. 500. Developers enforce these constraints in the system by adding appropriate code in the application programs. However, when new constraints are added, it is difficult to change the application programs to enforce them.
- **Atomicity problems**. A computer system, like any other 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 Rs.500 from the account balance of department *A* to the account balance of department *B*. If a system failure occurs during the execution of the program, it is possible that the Rs.500 was removed from the balance of department *A* but was not credited to the balance of department *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.
- **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
- **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 various bank employees. They do not need access to information about customer accounts.

  In the file processing systems, as the application programs are added to the system in an adhoc manner, it is difficult to enforce security.

The above disadvantages can be overcome by use of DBMS and it provides the following advantages.

1. Provides for mass storage of relevant data.
2. Make easy access of the data to user.
3. Allows for the modification of data in a consistent manner.
4. Allows multiple users to be active at a time
5. Eliminate or reduce the redundant data.
6. Provide prompt response to the users request for data.
7. Supports Backup and recovery of data.
8. Protect data from physical hardware failure and unauthorized access.

9. Constraints can be set to database to maintain data integrity.

ADVANTAGES AND DISADVANTAGES OF A DBMS

Using a DBMS to manage data has many advantages:

Reduction of Redundancy: This is perhaps the most significant advantage of using DBMS. Redundancy is the problem of storing the same data item in more one place. Redundancy creates several problems like requiring extra storage space, entering same data more than once during data insertion, and deleting data from more than one place during deletion. Anomalies may occur in the database if insertion, deletion etc are not done properly.

Sharing of Data: In a paper-based record keeping, data cannot be shared among many users. But in computerized DBMS, many users can share the same database if they are connected via a network.

Data Integrity: We can maintain data integrity by specifying integrity constrains, which are rules and restrictions about what kind of data may be entered or manipulated within the database. This increases the reliability of the database as it can be guaranteed that no wrong data can exist within the database at any point of time.

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.

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.

Concurrent access and crash recovery: A DBMS schedules concurrent accesses to the data in such a manner that users can think of the data as being accessed by only one user at a time. Further, the DBMS protects users from the effects of system failures.

Reduced application development time: Clearly, the 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 A 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.

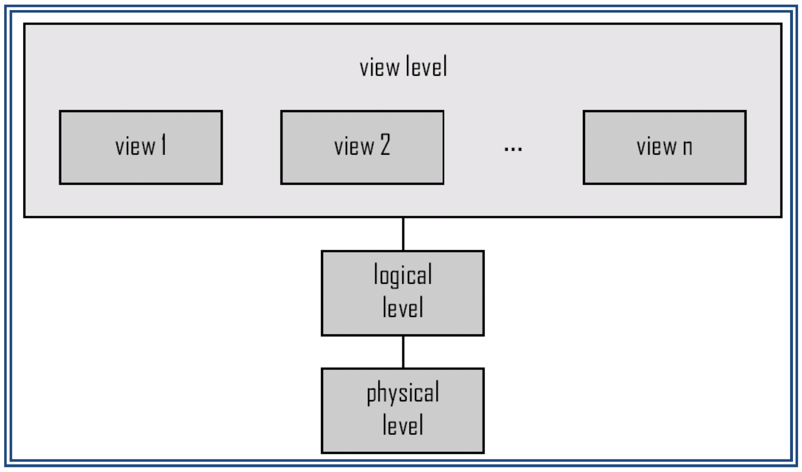
Qualified Personnel: The professional operation of a database system requires appropriately trained staff. Without a qualified database administrator nothing will work for long.

Costs: Through the use of a database system new costs are generated for the system itselfs 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 specialised software which is produced and optimised exactly for one problem.

VIEW OF DATA:

A DBMS is a collection of interrelated files and set of programs which allows the users to access and modify these files.



Data Abstraction

A major purpose of a database system is to provide users with an abstract view of the data. That is, the system hides certain details of how the data are stored and maintained. It is called data abstraction.

Level of Abstraction: basically, Abstraction can be divided in to 3 levels. **They are**

1. **Physical Level :** The lowest of abstraction describes how the data are actually stored. At the physical level, complex low-level data structures are described in detail.

2. **Logical Level (Conceptual Level) :** This next higher level of abstraction describes what data are stored in the database, and what relationship exist among those data. This level of abstraction is

used by Database Administrators (DBA), Who must decide what information is to be kept in the database.

3. **View Level :** This Highest level of abstraction describes only part of the entire database. The use of simpler structures at the logical level, some complexity remains, because of the large databases. Many users of the database system will not be concerned with all this information. Such users need to access only a Part of the database. So that their interaction with the system is simplified, the view level of abstraction is defined. The system may provide views for the same database.

An analogy to the concept of data types in programming languages may clarify the distinction among levels of abstraction. Many high-level programming languages support the notion of a structured type. For example, we may describe a record as follows:

```
type instructor = record
                        ID : char (5);
                        name : char (20);
                        dept name : char (20);
                        salary : numeric (8,2);
                      end;
```

This code defines a new record type called *instructor* with four fields. Each field has a name and a type associated with it. A university organization may have several such record types, including

department, with fields *dept name*, *building*, and *budget*

course, with fields *course id*, *title*, *dept name*, and *credits*

student, with fields *ID*, *name*, *dept name*, and *tot cred*

At the physical level, an *instructor*, *department*, or *student* record can be de-scribed as a block of consecutive storage locations. The compiler hides this level of detail from programmers. Similarly, the database system hides many of the lowest-level storage details from database programmers. Database administrators, on the other hand, may be aware of certain details of the physical organization of the data.

At the logical level, each such record is described by a type definition, as in the previous code segment, and the interrelationship of these record types is defined as well. Programmers using a programming language work at this level of abstraction. Similarly, database administrators usually work at this level of abstraction.

Finally, at the view level, computer users see a set of application programs that hide details of the data types. At the view level, several views of the database are defined, and a database user sees some or all of these views. In addition to hiding details of the logical level of the database, the views also provide a security mechanism to prevent users from accessing certain parts of the database. For example, clerks in the university registrar office can see only that part of the database that has information about students; they cannot access information about salaries of instructors.

<u>INSTANCES AND SCHEMAS:</u>

Databases change over time as information is inserted and deleted.

Instances

The collection of information stored in the database at a particular moment is called an **instance** of the database. It is also called as *snapshot or set of occurrence or current state of the database*.

Example: Instance of the employee schema

Eno	Ename	Salary	Address
1	A	10000	1st street
2	B	20000	2nd steet
3	C	30000	3rd street

Schemas

The overall design of the database is called the database **schema**. Schemas are changed infrequently, if at all.

The concept of database schemas and instances can be understood by analogy to a program written in a programming language. A database schema corresponds to the variable declarations (along with associated type definitions) in a program. Each variable has a particular value at a given instant. The values of the variables in a program at a point in time correspond to an *instance* of a database schema. In general, database system supports one physical schema, one logical schema and several subschema's.

Database systems have several schemas, partitioned according to the levels of abstraction.

- The **physical schema** describes the database design at the physical level,
- The **logical schema** describes the database design at the logical level.
- A database may also have several schemas at the view level, sometimes called **subschemas**, that describe different views of the database.

Database Schema diagram for a company:

EMPLOYEE

Eno	Ename	Salary	Address

DEPARTMENT

Dno	Dname	Dlocation

DATA INDEPENDENCE

The three-schema architecture can be used to further explain the concept of data independence, which can be defined as the capacity to change the schema at one level of a database system without having to change the schema at the next higher level.

We can define two types of data independence:

1. **Logical data independence** is the capacity to change the conceptual schema without having to change external schemas or application programs. We may change the conceptual schema to expand the database (by adding a record type or data item), to change constraints, or to reduce the database (by removing a record type or data item).

Only the view definition and the mappings need be changed in a DBMS that supports logical data independence. After the conceptual schema undergoes a logical reorganization, application programs that reference the external schema constructs must work as before. Changes to constraints can be applied to the conceptual schema without affecting the external schemas or application programs.

2. **Physical data independence** is the capacity to change the internal schema without having to change the conceptual schema. Hence, the external schemas need not be changed as well. Changes to the internal schema may be needed because some physical files had to be reorganized.

For example, by creating additional access structures to improve the performance of retrieval or update. If the same data as before remains in the database, we should not have to change the conceptual schema.

Whenever we have a multiple-level DBMS, its catalog must be expanded to include information on how to map requests and data among the various levels. The DBMS uses additional software to accomplish these mappings by referring to the mapping information in the catalog.

Data independence occurs because when the schema is changed at some level, the schema at the next higher level remains unchanged; only the mapping between the two levels is changed. Hence, application programs referring to the higher-level schema need not be changed. The three-schema architecture can make it easier to achieve true data independence, both physical and logical. However, the two levels of mappings create an overhead during compilation or execution of a query or program, leading to inefficiencies in the DBMS. Because of this, few DBMSs have implemented the full three-schema architecture.

DATA MODELS:

A collection of tools for describing Data, Data relationships,Data semantics and Data constraints.

The data models can be classified into four different categories:

- Relational model
- Entity-Relationship data model (mainly for database design)
- Object-based data models (Object-oriented and Object-relational)
- Semi-structured data model (XML)
- Other older models:
  - Network model
  - Hierarchical model

Relational Model

The relational model is currently the most popular data model in the database management systems. The popularity is because of simplicity and understandability. This data model is developed by E.F.Codd in 1970 which is based on relation, two dimensional table.

The relational data model uses a collection of tables (also called as relation) to both data and the relationships among those data. Each table has multiple columns and each column has unique name. A relation consists of rows and columns. The row in table (relation) is called as Tuple and column name are known as attribute.

Ex : Customer Table

Customer Name	**UID**	**Address**	**Account No**
Tanveer	A12345	Hyd	A – 101
Ramesh	B23456	Sec'bad	A – 215
Ravi	C34567	Charminar	A – 305
Prasad	A345789	Banglore	A – 201
Smith	Z459087	Delhi	A - 405

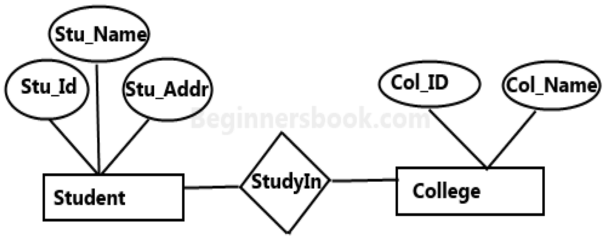
Advantages

1. In this model, data redundancy is controlled to a greater extent
2. The relational data model allows many-to-many relationships.
3. The relational data model structures are very simple and easy to build
4. Faster access of data is possible and storage space required is greatly reduced.

Entity-Relationship Model

The entity-relationship (E-R) data model uses a collection of basic objects, called *entities*, and *relationships* among these objects. An entity is a "thing" or "object" in the real world that is distinguishable from other objects. The entity-relationship model is widely used in database design.

It is a high level conceptual data model that describes the structure of database in terms of entities, relationship among entities & constraints on them.



Sample E-R Diagram

Basic Concepts of E-R Model:

Entity

Entity Set

Attributes

Relationship

Relationship set

Identifying Relationship

which is made up of components. Some of they are

- **Rectangles :** Which represent entity sets.
- **Ellipses :** Which represent attributes
- **Diamonds :** Which represent relationship sets
- **Lines :** Which link attributes to entity sets and entity sets to relationship sets.

Object-Based Data Model

Object-oriented programming (especially in Java, C++, or C#) has become the dominant software-development methodology. This led to the development of an object-oriented data model that can be seen as extending the E-R model with notions of encapsulation, methods (functions), and object identity. The object-relational data model combines features of the object-oriented data model and relational data model.

Semi-structured Data Model

The semi-structured data model permits the specification of data where individual data items of the same type may have different sets of attributes. This is in contrast to the data models mentioned earlier, where every data item of a particular type must have the same set of attributes. The **Extensible Markup Language (XML)** is widely used to represent semi-structured data.

Historically, the **network data model** and the **hierarchical data model** preceded the relational data model. These models were tied closely to the underlying implementation, and complicated the task of modeling data. As a result they are used little now, except in old database code that is still in service in some places.

Network Data Model:

Data in the network model are represented by collections of records and relationships among data are represented by links,which can be viewed as pointers. The records in the database can be organized as a collection of arbitrary graphs. The Network data model is similar to Hierarchical model except that one data can have more than one parent. Any record in the database is allowed to own sets of other type of record.

Advantages

- It can be used to represent many-to-many relationships
- It offers integration of data
- The storage space is reduced considerably due to less redundancy
- It provides faster access of data.

Hierarchical Data Model :

A **hierarchical database model** is a **data model** in which the **data** is organized into a tree-like structure. The **data** is stored as records which are connected to one another through links. A record is a collection of fields, with each field containing only one value.

In this model the relationship among the data is represented by records and links. It consists of records which are connected to another through links. A link can be defined as an association between two records. This hierarchical data model can do considered as an upside –down tree, with the highest level of tree kept as root.

Advantages

- The hierarchical model, allows one-to-one-and one-to-many relationships.
- The model has got the ability to handle large amount of data.

Disadvantages

- The model involves with complicated querying.
- As duplication of data takes place, there is wastage of storage space.
- During updating of data inconsistency exists.
- The model does not allow many-to-many relationships.

DATA BASE LANGUAGES:

A database system provides two different types of Languages, one will specify the schema, and other will express database queries and updates. They are

- Data-Definition Languages (DDL)
- Data-Manipulation Language (DML)
- Data Control language (DCL)

1. **Data-Definition Languages (DDL) :** A database scheme is specified by set of definitions which are expressed by special language called Data Definition Language (DDL). The result of compilation of DDL statements is a set of tables that is stored in a special file called 'Data dictionary' or "data directory.

A data dictionary is a file that contains metadata, i.e. Data about data. This file is consulted before actual data are read or modified in the database system

The storage structure and access methods used by the database system are specified by a set of definitions in a special type of DDL called a 'data storage and data definition language'. The result of consultation of these definitions is a set instruction to specify the implementation details of the database schemas. Which are usually hidden form the users.

The database systems implement integrity constraints that can be tested with minimal overhead:

Domain Constraints: A domain of possible values must be associated with every attribute (for example, integer types, character types, date/time types). Declaring an attribute to be of a particular domain acts as a constraint on the values that it can take. Domain constraints are the most elementary form of integrity constraint. They are tested easily by the system whenever a new data item is entered into the database.

Referential Integrity: There are cases where we wish to ensure that a value that appears in one relation for a given set of attributes also appears in a certain set of attributes in another relation (referential integrity).

Assertions: An assertion is any condition that the database must always satisfy. Domain constraints and referential-integrity constraints are special forms of assertions.

For example, "Every department must have at least five courses offered every semester" must be expressed as an assertion. When an assertion is created, the system tests it for validity. If the assertion is valid, then any future modification to the database is allowed only if it does not cause that assertion to be violated.

Authorization: We may want to differentiate among the users as far as the type of access they are permitted on various data values in the database. These differentiations are expressed in terms of **authorization**, the most common being:

read authorization, which allows reading, but not modification, of data;

insert authorization, which allows insertion of new data, but not modification of existing data;

update authorization, which allows modification, but not deletion, of data; and

delete authorization, which allows deletion of data. We may assign the user all, none, or a combination of these types of authorization.

The DDL commands are

- To create the database instance – <u>CREATE</u>
- To alter the structure of database – **ALTER**
- To drop database instances – <u>DROP</u>
- To delete tables in a database instance – **TRUNCATE**
- To rename database instances – **RENAME**

All these commands specify or update the database schema that's why they come under Data Definition language.

- o Used by the DBA and database designers to specify the conceptual schema of a database.
- o In many DBMSs, the DDL is also used to define internal and external schemas (views).
- o In some DBMSs, separate storage definition language (SDL) and view definition language (VDL) are used to define internal and external schemas.
- o SDL is typically realized via DBMS commands provided to the DBA and database designers
- o DDL compiler generates a set of tables stored in a data dictionary
- o Data dictionary contains metadata (i.e., data about data)

2. **Data-Manipulation Language (DML) :** A DML is language which enables users to access or manipulate data as organized by appropriate data model. The goal is to provide efficient human interaction with the system. The DML allows following
   (a) The retrieval information form the database
   (b) The Insertion of new information in to existing database
   (c) The deletion of existing information from database
   (d) The modification of information stored in the database.

The DML commands are

- To read records from table(s) – <u>SELECT</u>
- To insert record(s) into the table(s) – **INSERT**
- Update the data in table(s) – <u>UPDATE</u>
- Delete all the records from the table – <u>DELETE</u>

  - o Used to specify database retrievals and updates
  - o DML commands (data sublanguage) can be *embedded* in a general-purpose programming language (AKA host language), such as COBOL, C, C++, or Java.
  - o Alternatively, stand-alone DML commands can be applied directly (called a *query language*).
  - o Language for accessing and manipulating the data organized by the appropriate data model
  - o DML also known as query language

A DML is language which enables users to access or manipulate data. There are basically two types.

- **Procedural DML:** This requires a user to specify what data are needed and how to get those data from existing database.
- **Non procedural DML:** Which require a user to specify what data are needed 'without' specifying how to get those data.
-

Non procedural DMLs are usually easier to learn and use than procedural DMLs. A user does not have to specify how to the data, these languages may generate code that is not as that produced by Procedural DML. Hence we can make remedy this difficulty by various optimization techniques.

A Query is a statement, a request for retrieval information. The portion of a DML, that involves information retrieval is called a 'Query Language'.

This query in the SQL language finds the name of the customer whose customer-id is 192-83-7465:

Select *customer.customer-name* **from** *customer* **where***customer.customer-id* = 192-83-7465

The query specifies that those rows *from* the table *customer where* the *customer-id* is 192-83-7465 must be retrieved, and the *customer-name* attribute of these rows must be displayed.

Queries may involve information from more than one table. For instance, the following query finds the balance of all accounts owned by the customer with customerid 192-83-7465.

Select *account.balance* **from** *depositor*, *account* **where** *depositor.customer-id* = 192-83-7465
and *depositor.account-number*= *account.account-number*

There are a number of database query languages in use, either commercially or experimentally.

The levels of abstraction apply not only to defining or structuring data, but also to manipulating data. At the physical level, we must define algorithms that allow efficient access to data. At higher levels of abstraction, we emphasize ease of use. The goal is to allow humans to interact efficiently with the system. The query processor component of the database system translates DML queries into sequences of actions at the physical level of the database system.

- **Data Control language (DCL)**: DCL is used for granting and revoking user access on a database
  - To grant access to user – GRANT
  - To revoke access from user – REVOKE

In practical data definition language, data manipulation language and data control languages are not separate language; rather they are the parts of a single database language such as SQL.

DATA DICTIONARY

We can define a data dictionary as a DBMS component that stores the definition of data characteristics and relationships. You may recall that such "data about data" were labeled metadata. The DBMS data dictionary provides the DBMS with its self describing characteristic. In effect, the data dictionary resembles and X-ray of the company's entire data set, and is a crucial element in the data administration function.

The two main types of data dictionary exist, integrated and stand alone.

An **integrated data dictionary** is included with the DBMS. For example, all relational DBMSs include a built in data dictionary or system catalog that is frequently accessed and updated by the RDBMS.

Other DBMSs **– Stand alone** especially older types, do not have a built in data dictionary instead the DBA may use third party stand alone data dictionary systems.

Data dictionaries can also be classified as active or passive.

An **active data dictionary** is automatically updated by the DBMS with every database access, thereby keeping its access information up-to-date.

A **passive data dictionary** is not updated automatically and usually requires a batch process to be run. Data dictionary access information is normally used by the DBMS for query optimization purpose.

The data dictionary's main function is to store the description of all objects that interact with the database. Integrated data dictionaries tend to limit their metadata to the data managed by the DBMS. Stand alone data dictionary systems are more usually more flexible and allow the DBA to describe and manage all the organization's data, whether or not they are computerized. Whatever the data dictionary's format, its existence provides database designers and end users with a much improved ability to communicate. In addition, the data dictionary is the tool that helps the DBA to resolve data conflicts.

Although, there is no standard format for the information stored in the data dictionary several features are common. For example, the data dictionary typically stores descriptions of all:

- Data elements that are define in all tables of all databases. Specifically the data dictionary stores the name, data types, display formats, internal storage formats, and validation rules. The data dictionary tells where an element is used, by whom it is used and so on.
- Tables define in all databases. For example, the data dictionary is likely to store the name of the table creator, the date of creation access authorizations, the number of columns, and so on.
- Indexes define for each database tables. For each index the DBMS stores at least the index name the attributes used, the location, specific index characteristics and the creation date.
- Define databases: who created each database, the date of creation where the database is located, who the DBA is and so on.
- End users and The Administrators of the data base
- Programs that access the database including screen formats, report formats application formats, SQL queries and so on.
- Access authorization for all users of all databases.
- Relationships among data elements which elements are involved: whether the relationship are mandatory or optional, the connectivity and cardinality and so on.

If the data dictionary can be organized to include data external to the DBMS itself, it becomes an especially flexible to for more general corporate resource management. The management of such an extensive data dictionary, thus, makes it possible to manage the use and allocation of all of the organization information regardless whether it has its roots in the database data.

RELATIONAL DATABASES :

A relational database is based on the relational model and uses a collection of tables to represent both data and the relationships among those data. It also includes a DML and DDL.

The purpose of the relational model is to provide a declarative method for specifying data and queries: users directly state what information the database contains and what information they want from it, and let the database management system software take care of describing data structures for storing the data and retrieval procedures for answering queries.

Relational Database: One of the major advantages of using a relational database is its structural flexibility. It allows the users to retrieve the data in any combination

A relation is a two-dimensional array, consisting of horizontal rows and vertical columns. Each row, column ie a cell contains a unique value and no two rows are identical with respect to one another.

Columns are always self-consistent in the sense that it has the same meaning in every row. This means that the database management system (DBMS) is not concerned with its appearance, either first or next. The table will be processed the same way, regardless of the order of the columns.

Relations are commonly referred as tables.. Every column in a database table acts as attribute since the meaning of the column is same for every row of the database .A row consists of a set of fields and hence commonly referred as a record.

Properties of Relational Database: The important properties of a relational database are listed below:

1. A relational database is a collection of relations.
2. The database tables have a row column format.
3. Operators are available either to join or separate columns of the database table.
4. Relations are formed with respect to data only.
5. The tables can be accessed by using simple non-procedural statements.
6. The data is fully independent, that is it will be the same irrespective of the access path used.

Database Access from Application Programs

SQL is not as powerful as a universal Turing machine; that is, there are some computations that are possible using a general-purpose programming language but are not possible using SQL. SQL also does not support actions such as input from users, output to displays, or communication over the network. Such computations and actions must be written in a *host* language, such as C, C++, or Java, with embedded SQL queries that access the data in the database. **Application programs** are programs that are used to interact with the database in this fashion.

Examples in a university system are programs that allow students to register for courses, generate class rosters, calculate student GPA, generate payroll checks, etc. To access the database, DML statements need to be executed from the host language. There are two ways to do this:

By providing an application program interface (set of procedures) that can be used to send DML and DDL statements to the database and retrieve the results.

The Open Database Connectivity (ODBC) standard for use with the C language is a commonly used application program interface standard. The Java Database Connectivity (JDBC) standard provides corresponding features to the Java language.

By extending the host language syntax to embed DML calls within the host language program. Usually, a special character prefaces DML calls, and a preprocessor, called the **DML precompiler**, converts the DML statements to normal procedure calls in the host language.

DATABASE DESIGN:

Database systems are designed to manage large bodies of information. These large bodies of information do not exist in isolation. They are part of the operation of some enterprise whose end product may be information from the database or may be some device or service for which the database plays only a supporting role.

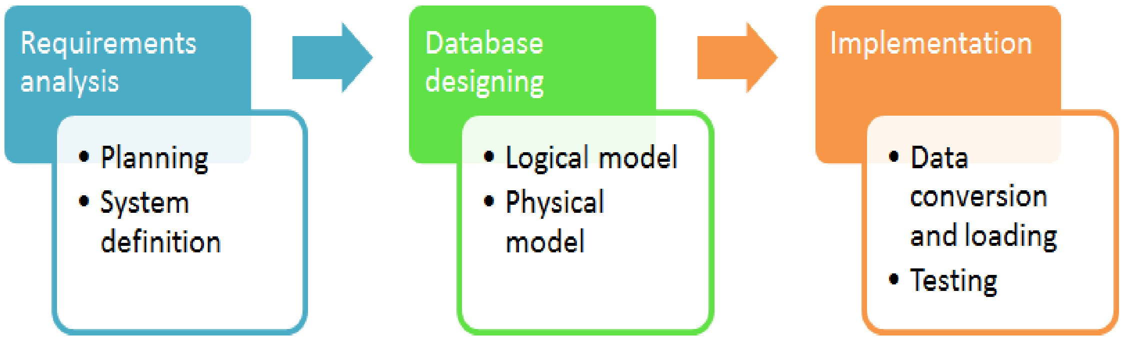
Database Design is a collection of processes that facilitate the designing, development, implementation and maintenance of enterprise data management systems.

It helps produce database systems

1. That meet the requirements of the users
2. Have high performance.

A high-level data model provides the database designer with a conceptual frame-work in which to specify the data requirements of the database users, and how the database will be structured to fulfill these requirements. The initial phase of database design, then, is to characterize fully the data needs of the prospective database users. The database designer needs to interact extensively with domain experts and users to carry out this task. The outcome of this phase is a specification of user requirements.

Design Process:



The database development life cycle has a number of stages that are followed when developing database systems. The steps in the development life cycle do not necessary have to be followed religiously in a sequential manner.

On small database systems, the database system development life cycle is usually very simple and does not involve a lot of steps.

In order to fully appreciate the above diagram, let's look at the individual components listed in each step.

Requirements analysis

- **Planning** - This stages concerns with planning of entire Database Development Life-Cycle. It takes into consideration the Information Systems strategy of the organization.
- **System definition** - This stage defines the scope and boundaries of the proposed database system.

Database designing

The process of moving from an abstract data model to the implementation of the database proceeds in two final design phases.

In the **logical-design phase**, the designer maps the high-level conceptual schema onto the implementation data model of the database system that will be used.

The designer uses the resulting system-specific database schema in the subsequent **physical-design phase**, in which the physical features of the database are specified.

Implementation

A fully developed conceptual schema indicates the functional requirements of the enterprise. In a **specification of functional requirements**, users describe the kinds of operations (or transactions) that will be performed on the data. Example operations include modifying or updating data, searching for and retrieving specific data, and deleting data. At this stage of conceptual design, the designer can review the schema to ensure it meets functional requirements.

- **Data conversion and loading** - this stage is concerned with importing and converting data from the old system into the new database.
- **Testing** - this stage is concerned with the identification of errors in the newly implemented system .It checks the database against requirement specifications.

Database Design for a University Organization

To illustrate the design process, let us examine how a database for a university could be designed. The initial specification of user requirements may be based on interviews with the database users, and on the designer's own analysis of the organization. The description that arises from this design phase serves as the basis for specifying the conceptual structure of the database. Here are the major characteristics of the university.

The university is organized into departments. Each department is identified by a unique name (*dept name*), is located in a particular *building*, and has a *budget*.

Each department has a list of courses it offers. Each course has associated with it a *course id*, *title*, *dept name*, and *credits*, and may also have have associated *prerequisites*.

Instructors are identified by their unique *ID*. Each instructor has *name*, associated department (*dept name*), and *salary*.

Students are identified by their unique *ID*. Each student has a *name*, an associated major department (*dept name*), and *tot cred* (total credit hours the student earned thus far).

The university maintains a list of classrooms, specifying the name of the *building*, *room number*, and room *capacity*.

The university maintains a list of all classes (sections) taught. Each section is identified by a *course id*, *sec id*, *year*, and *semester*, and has associated with it a *semester*, *year*, *building*, *room number*, and *time slot id* (the time slot when the class meets).

The department has a list of teaching assignments specifying, for each instructor, the sections the instructor is teaching.

The university has a list of all student course registrations, specifying, for each student, the courses and the associated sections that the student has taken (registered for).

Two Types of Database Techniques:

1. Normalization
2. ER Modeling

NORMALIZATION :

Another method for designing a relational database is to use a process commonly known as normalization. The goal is to generate a set of relation schemas that allows us to store information without unnecessary redundancy, yet also allows us to retrieve information easily. The approach is to design schemas that are in an appropriate *normal form*. To determine whether a relation schema is in one of the desirable normal forms, we need additional information about the real-world enterprise that we are modeling with the database. The most common approach is to use **functional dependencies**.

To understand the need for normalization, let us look at what can go wrong in a bad database design. Among the undesirable properties that a bad design may have are:

- Repetition of information
- Inability to represent certain information

Normalization is a process of organizing the data in database to avoid data redundancy, insertion anomaly, update anomaly & deletion anomaly. Let's discuss about anomalies first then we will discuss normal forms with examples.

Anomalies in DBMS : There are three types of anomalies that occur when the database is not normalized. These are – Insertion, update and deletion anomaly. Let's take an example to understand this.

Example: Suppose a manufacturing company stores the employee details in a table named employee that has four attributes: emp_id for storing employee's id, emp_name for storing employee's name, emp_address for storing employee's address and emp_dept for storing the department details in which the employee works. At some point of time the table looks like this:

emp_id	emp_name	emp_address	emp_dept
101	Rick	Delhi	D001
101	Rick	Delhi	D002
123	Maggie	Agra	D890
166	Glenn	Chennai	D900
166	Glenn	Chennai	D004

The above table is not normalized. We will see the problems that we face when a table is not normalized.

Update anomaly: In the above table we have two rows for employee Rick as he belongs to two departments of the company. If we want to update the address of Rick then we have to update the same in two rows or the data will become inconsistent. If somehow, the correct address gets updated in one department but not in other then as per the database, Rick would be having two different addresses, which is not correct and would lead to inconsistent data.

Insert anomaly: Suppose a new employee joins the company, who is under training and currently not assigned to any department then we would not be able to insert the data into the table if emp_dept field doesn't allow nulls.

Delete anomaly: Suppose, if at a point of time the company closes the department D890 then deleting the rows that are having emp_dept as D890 would also delete the information of employee Maggie since she is assigned only to this department.

To overcome these anomalies we need to normalize the data.

The Entity-Relationship Model

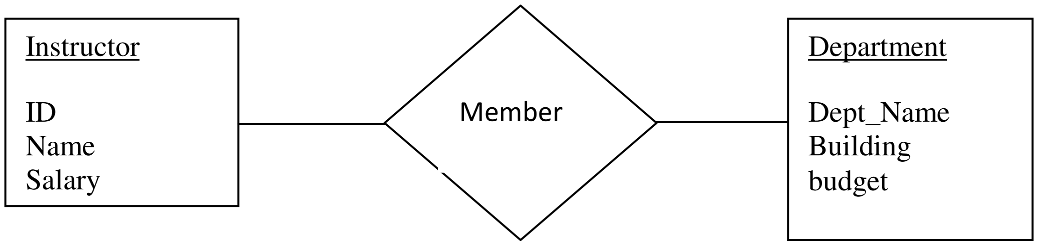
The entity-relationship (E-R) data model uses a collection of basic objects, called *entities*, and *relationships* among these objects. An entity is a "thing" or "object" in the real world that is distinguishable from other objects. For example, each person is an entity, and bank accounts can be considered as entities.

Entities are described in a database by a set of **attributes**. For example, the attributes *dept name*, *building*, and *budget* may describe one particular department in a university, and they form attributes of the *department* entity set. Similarly, attributes *ID*, *name*, and *salary* may describe an *instructor* entity.

The extra attribute *ID* is used to identify an instructor uniquely (since it may be possible to have two instructors with the same name and the same salary). A unique instructor identifier must be assigned to each instructor. In the United States, many organizations use the social-security number of a person (a unique number the U.S. government assigns to every person in the United States) as a unique identifier.

A **relationship** is an association among several entities. For example, a *member* relationship associates an instructor with her department. The set of all entities of the same type and the set of all relationships of the same type are termed an **entity set** and **relationship set**, respectively.

The overall logical structure (schema) of a database can be expressed graph-ically by an *entity-relationship (E-R) diagram*. There are several ways in which to draw these diagrams. One of the most popular is to use the **Unified Modeling Language (UML)**. In the notation we use, which is based on UML, an E-R diagram is represented as follows:



Entity sets are represented by a rectangular box with the entity set name in the header and the attributes listed below it.
Relationship sets are represented by a diamond connecting a pair of related entity sets. The name of the relationship is placed inside the diamond.

As an illustration, consider part of a university database consisting of instruc-tors and the departments with which they are associated. In the above Figure shows the corresponding E-R diagram. The E-R diagram indicates that there are two entity sets, *instructor* and *department*, with attributes as outlined earlier. The diagram also shows a relationship *member* between *instructor* and *department*.

In addition to entities and relationships, the E-R model represents certain constraints to which the contents of a database must conform. One important constraint is **mapping cardinalities**, which express the number of entities to which another entity can be associated via a relationship set. For example, if each instructor must be associated with only a single department, the E-R model can express that constraint.

DATA ARCHITECTURE:

Three important characteristics of the database approach are

(1) Insulation of programs and data (program-data and program-operation independence);

(2) Support of multiple user views; and

(3) Use of a catalog to store the database description (schema).

In this section we specify an architecture for database systems, called the **three-schema architecture**, which was proposed to help achieve and visualize these characteristics.

The goal of the three-schema architecture, illustrated in Figure 1.1, is to separate the user applications and the physical database. In this architecture, schemas can be defined at the following three levels:

The **internal level** has an **internal or Physical 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.

The **conceptual level** has a **conceptual or Logical 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.

The **external** or **view level** includes a number of **external or View 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.

Hence, the DBMS must transform a request specified on an external schema into a request against the conceptual schema, and then into a request on the internal schema for processing over the stored database. If the request is a database retrieval, the data extracted from the stored database must be reformatted to match the user's external view.

The processes of transforming requests and results between levels are called **mappings.** These mappings may be time-consuming, so some DBMSs—especially those that are meant to support small databases—do not support external views. Even in such systems, however, a certain amount of mapping is necessary to transform requests between the conceptual and internal levels.

The DBMS accepts SQL commands generated from a variety of user interfaces, produces query evaluation plans, executes these plans against the database, and returns the answers. (This is a implication: SQL commands can be embedded in host language application programs, e.g., Java or COBOL programs. We ignore these issues to concentrate on the core DBMS functionality.)

When a user issues a query, the parsed query is presented to a query optimizer, which uses information about how the data is stored to produce an efficient execution plan for evaluating the query. An execution plan is a blueprint for evaluating a query, and is usually represented as a tree of relational operators

The files and access methods layer code sits on top of the buffer manager, which brings pages in from disk to main memory as needed in response to read requests.

The lowest layer of the DBMS software deals with management of space on disk, where the data is stored. Higher layers allocate, de-allocate, read, and write pages through (routines provided by) this layer, called the disk space manager.

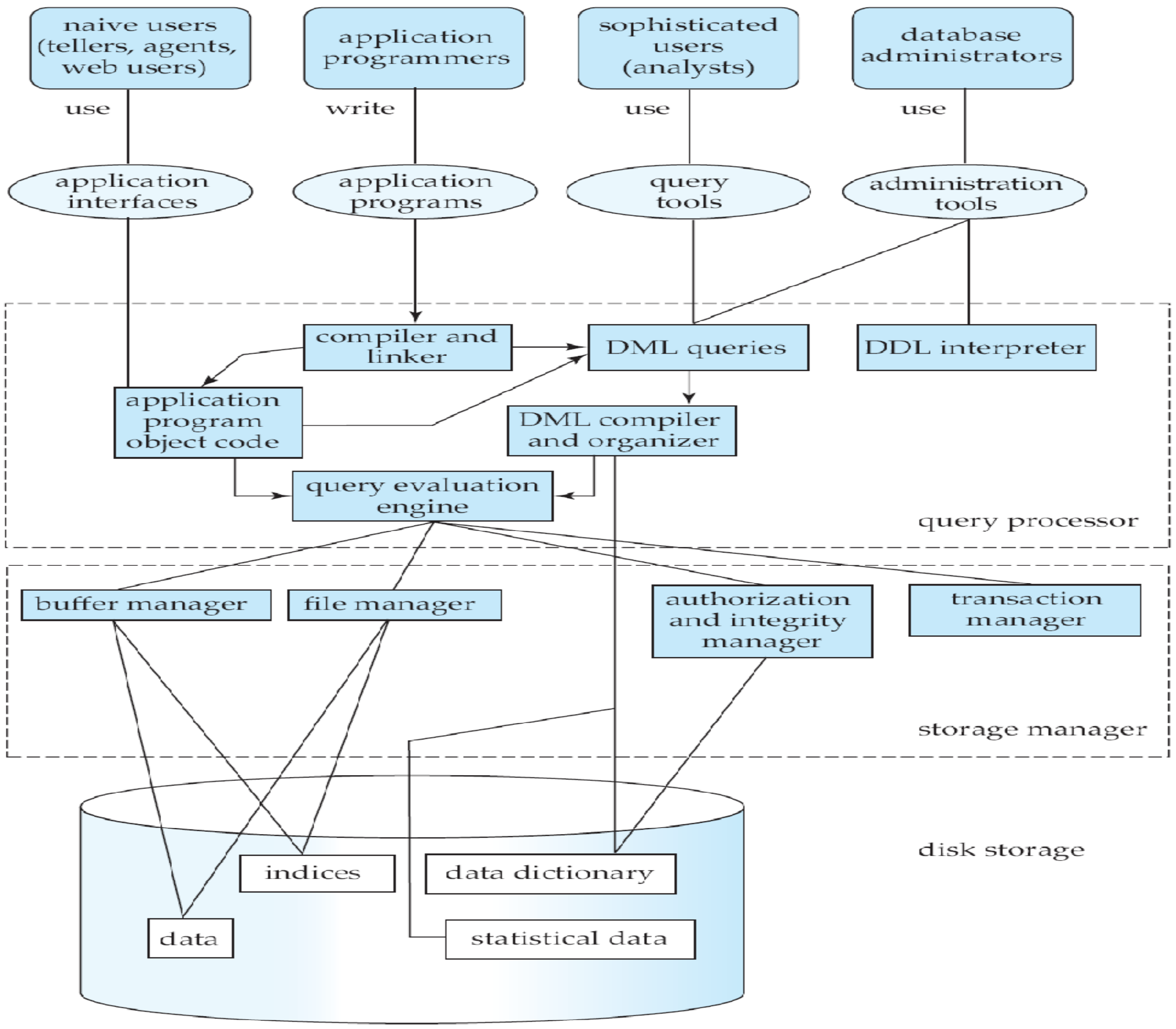
The DBMS supports concurrency and crash recovery by carefully scheduling user requests and maintaining a log of all changes to the database. DBMS components associated with concurrency control and recovery include the transaction manager, which ensures that transactions request and release locks according to a suitable locking protocol and schedules the execution transactions; the lock manager, which keeps track of requests for locks and grants locks on database objects when they become available; and the recovery manager, which is responsible for maintaining a log, and restoring the system to a consistent state after a crash. The disk space manager, buffer manager, and file and access method layers must interact with these components.

The architecture of a database systems is greatly influenced by the underlying computer system on which the database is running:

- Centralized
- Client-server
- Parallel (multi-processor)
- Distributed

The database and the DBMS catalog are usually stored on disk. Access to the disk is controlled primarily by the operating system (OS), which schedules disk read/write. Many DBMSs have their own buffer management module to schedule disk read/write, because this has a considerable effect on performance. Reducing disk read/write improves performance considerably. A higher-level stored data manager module of the DBMS controls access to DBMS information that is stored on disk, whether it is part of the database or the catalog.

In the following Figure, in a simplified form, the typical DBMS components. The figure is divided into two parts. The top part of the figure refers to the various users of the database environment and their interfaces. The lower part shows the internals of the DBMS responsible for storage of data and processing of transactions.



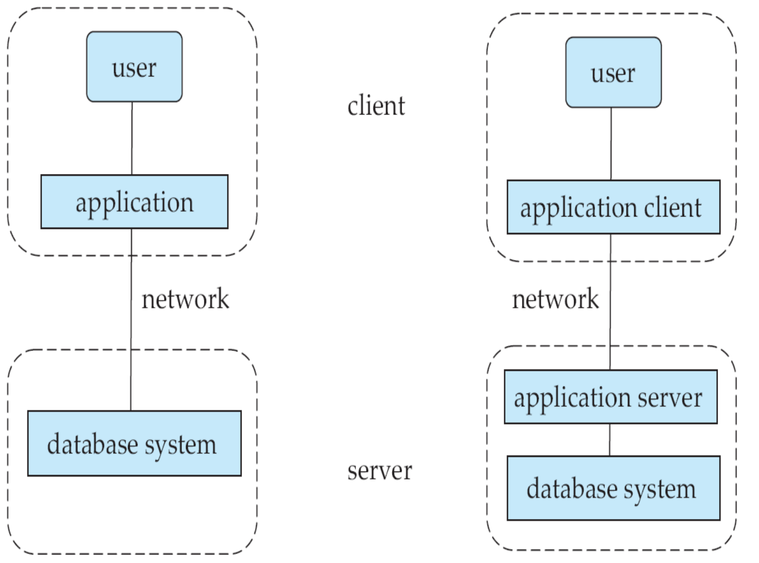
The architecture of a database system is greatly influenced by the underlying computer system on which the database system runs. Database systems can be centralized, or client-server, where one server machine executes work on behalf of multiple client machines. Database systems can also be designed to exploit parallel computer architectures. Distributed databases span multiple geographically separated machines.

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.

One-tier architecture

Imagine a person on a desktop computer who uses Microsoft Access to load up a list of personal addresses and phone numbers that he or she has saved in MS Windows' “My Documents” folder.

This is an example of a one-tier database architecture. The program (Microsoft Access) runs on the user's local machine, and references a file that is stored on that machine's hard drive, thus using a single physical resource to access and process information.



(a) Two-tier architecture (b) Three-tier architecture

Two-tier architecture

Database applications are usually partitioned into two or three parts, as in Figure (a). In a **Two-tier architecture**, the application resides at the client machine, where it 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.

Advantages:

1. Easy to maintain and modification is bit easy.
2. Communication is faster.

Disadvantages:

1. In two tier architecture application performance will be degrade upon increasing the users.
2. Cost-ineffective.

Three-tier architecture

In contrast (Figure (b)), in a **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 Intermediate layer called **application server**, usually through a forms interface. The application server in turn communicates with a database system to access data. It is commonly used architecture for web applications.

Advantages

1. High performance, lightweight persistent objects.
2. Scalability – Each tier can scale horizontally.
3. Performance – Because the Presentation tier can cache requests, network utilization is minimized, and the load is reduced on the Application and Data tiers.
4. Better Re-usability.
5. Improve Data Integrity.
6. Improved Security – Client is not direct access to database.
7. Easy to maintain, to manage, to scale, loosely coupled etc.

Disadvantages

1. Increase Complexity/Effort

DATA STORAGE AND QUERYING:

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. Corporate databases range in size from hundreds of gigabytes to, for the largest databases, terabytes of data. A gigabyte is approximately 1000 megabytes (actually 1024) (1 billion bytes), and a terabyte is 1 million megabytes (1 trillion bytes).

Since 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. Since the movement of data to and from disk is slow relative to the speed of the central processing unit, it is imperative that the database system structure the data so as to minimize the need to move data between disk and main memory.

The query processor is important because it helps the database system to simplify and facilitate access to data. The query processor allows database users to obtain good performance while being able to work at the view level and not be burdened with understanding the physical-level details of the implementation of the system. It is the job of the database system to translate updates and queries written in a nonprocedural language, at the logical level, into an efficient sequence of operations at the physical level.

Storage Manager

The *storage manager* is the component of a database system 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 raw data are stored on the disk using the file system provided by the operating system. 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.

The storage manager components include:

- **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 remains in a consistent (correct) 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, and deciding what data to cache in main memory. The buffer manager is a critical part of the database system, since it enables the database to handle data sizes that are much larger than the size of main memory.
-

The storage manager implements several data structures as part of the physical system implementation:

- **Data files**, which store the database itself.
- **Data dictionary**, which stores metadata about the structure of the database, in particular the schema of the database.
- **Indices**, which can provide fast access to data items. Like the index in this textbook, a database index provides pointers to those data items that hold a particular value. For example, we could use an index to find the *instructor* record with a particular *ID*, or all *instructor* records with a particular *name*. Hashing is an alternative to indexing that is faster in some but not all cases.

The Query Processor

The query processor components include:

- **DDL interpreter**,which interprets DDL statements and records the definitions in the data dictionary.
- **DML compiler**,which translates DML statements in a query language into an evaluation plan consisting of low-level instructions that the query evaluation engine understands.
  A query can usually be translated into any of a number of alternative evaluation plans that all give the same result. The DML compiler also performs **query optimization**; that is, it picks the lowest cost evaluation plan from among the alternatives.
- **Query evaluation engine**, which executes low-level instructions generated by the DML compiler.

TRANSACTION MANAGEMENT:

A transaction is a collection of operations that performs a single logical function in a database application. Each transaction is a unit of both atomicity and consistency. A transaction can be defined as a group of tasks. A single task is the minimum processing unit which cannot be divided further.

Let's take an example of a simple transaction. Suppose a bank employee transfers Rs 500 from A's account to B's account. This very simple and small transaction involves several low-level tasks.

A's Account **B's Account**

Open_Account(A)	Open_Account(B)
Old_Balance = A.balance	Old_Balance = B.balance
New_Balance = Old_Balance – 500	New_Balance = Old_Balance + 500
A.balance = New_Balance	B.balance = New_Balance
Close_Account(A)	Close_Account(B)

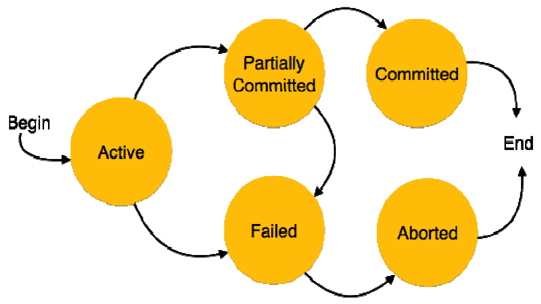
ACID Properties

A transaction is a very small unit of a program and it may contain several lowlevel tasks. A transaction in a database system must maintain **A**tomicity, **C**onsistency, **I**solation, and **D**urability – commonly known as ACID properties – in order to ensure accuracy, completeness, and data integrity.

- **Atomicity** – This property states that a transaction must be treated as an atomic unit, that is, either all of its operations are executed or none. There must be no state in a database where a transaction is left partially completed. States should be defined either before the execution of the transaction or after the execution/abortion/failure of the transaction.
- **Consistency** – The database must remain in a consistent state after any transaction. No transaction should have any adverse effect on the data residing in the database. If the database was in a consistent state before the execution of a transaction, it must remain consistent after the execution of the transaction as well.
- **Durability** – The database should be durable enough to hold all its latest updates even if the system fails or restarts. If a transaction updates a chunk of data in a database and commits, then the database will hold the modified data. If a transaction commits but the system fails before the data could be written on to the disk, then that data will be updated once the system springs back into action.
- **Isolation** – In a database system where more than one transaction are being executed simultaneously and in parallel, the property of isolation states that all the transactions will be carried out and executed as if it is the only transaction in the system. No transaction will affect the existence of any other transaction.

States of Transactions

A transaction in a database can be in one of the following states –



Active – In this state, the transaction is being executed. This is the initial state of every transaction.

Partially Committed – When a transaction executes its final operation, it is said to be in a partially committed state.

Failed – A transaction is said to be in a failed state if any of the checks made by the database recovery system fails. A failed transaction can no longer proceed further.

Aborted − If any of the checks fails and the transaction has reached a failed state, then the recovery manager rolls back all its write operations on the database to bring the database back to its original state where it was prior to the execution of the transaction. Transactions in this state are called aborted. The database recovery module can select one of the two operations after a transaction aborts −

- Re-start the transaction
- Kill the transaction

Committed − If a transaction executes all its operations successfully, it is said to be committed. All its effects are now permanently established on the database system.

A transaction is a collection of operations that performs a single logical function in a database application. Each transaction is a unit of both atomicity and consistency. Thus, we require that transactions do not violate any database consistency constraints. That is, if the database was consistent when a transaction started, the database must be consistent when the transaction successfully terminates. However, during the execution of a transaction, it may be necessary temporarily to allow inconsistency, since either the debit of A or the credit of B must be done before the other. This temporary inconsistency, although necessary, may lead to difficulty if a failure occurs.

It is the programmer's responsibility to define properly the various transactions, so that each preserves the consistency of the database. For example, the transaction to transfer funds from the account of department A to the account of department B could be defined to be composed of two separate programs: one that debits account A, and another that credits account B. The execution of these two programs one after the other will indeed preserve consistency. However, each program by itself does not transform the database from a consistent state to a new consistent state. Thus, those programs are not transactions. Ensuring the atomicity and durability properties are the responsibility of the database system itself specifically, of the recovery manager. In the absence of failures, all transactions complete successfully, and atomicity is achieved easily.

However, because of various types of failure, a transaction may not always complete its execution successfully. If we are to ensure the atomicity property, a failed transaction must have no effect on the state of the database. Thus, the database must be restored to the state in which it was before the transaction in question started executing. The database system must therefore perform failure recovery, that is, detect system failures and restore the database to the state that existed prior to the occurrence of the failure.

Finally, when several transactions update the database concurrently, the consistency of data may no longer be preserved, even though each individual transaction is correct. It is the responsibility of the concurrency-control manager to control the interaction among the concurrent transactions, to ensure the consistency of the database. The transaction manager consists of the **concurrency-control manager and the recovery manager.**

DATA MINING AND INFORMATION RETRIEVAL:

The term **data mining** refers loosely to the process of semi-automatically analyzing large databases to find useful patterns. Like knowledge discovery in artificial intelligence (also called **machine learning**) or statistical analysis, data mining attempts to discover rules and patterns from data.

However, data mining differs from machine learning and statistics in that it deals with large volumes of data, stored primarily on disk. That is, data mining deals with "knowledge discovery in databases."

The practice of examining large pre-existing database in order to generate new information or pattern.

Data mining applications that analyze large amounts of data searching for the occurrences of specific patterns or relationships, and for identifying unusual patterns in areas such as credit card usage. It was quickly apparent that basic relational systems were not very suitable for many of these applications, usually for one or more of the following reasons:

- More complex data structures were needed for modeling the application than the simple relational representation.
- New data types were needed in addition to the basic numeric and character string types.
- New operations and query language constructs were necessary to manipulate the new data types.
- New storage and indexing structures were needed for efficient searching on the new data types.

This led DBMS developers to add functionality to their systems. Some functionality was general purpose, such as incorporating concepts from object-oriented databases into relational systems. Other functionality was special purpose, in the form of optional modules that could be used for specific applications.

Data mining is a process used by companies to turn raw data into useful information. By using software, to look for pattern in large batch of data. Business can learn more about their customers and develop more effective marketing strategies as well as increase sales and decrease costs.

The major steps involved in a data mining process are:

- Extract, transform and load data into a data warehouse
- Store and manage data in a multidimensional databases
- Provide data access to business analysts using application software
- Present analyzed data in easily understandable forms, such as graphs.

Data mining process depends on effective data collection and warehousing as well as computer processing. When companies centralize their data into one database or program, It is called data warehousing. Such as data warehouses, for efficient analysis, data mining algorithms, facilitating business decision making and other information requirements to kindly cut costs and increase the sales.

Databases versus Information Retrieval

Textual data, too, has grown explosively. Textual data is unstructured, unlike the rigidly structured data in relational databases. Querying of unstructured textual data is referred to as ***information retrieval.***

Traditionally, database technology applies to structured and formatted data that arises in routine applications in government, business and industry. Database technology is heavily used in manufacturing, retail, banking, insurance, finance, and health care industries, where structured data is collected through forms, such as invoices or patient registration documents. An area related to database technology is Information Retrieval (IR), which deals with books, manuscripts, and various forms of library-based articles. Data is indexed, cataloged and annotated using keywords.

Information retrieval, as the name implies, concerns the retrieving of relevant information from databases. It is basically concerned with facilitating the user's access to large amounts of (predominantly textual) information.

The process of information retrieval involves the following stages:

1. Representing Collections of Documents
   - how to represent, identify and process the collection of documents.
2. User-initiated querying
   - understanding and processing of the queries.
3. Retrieval of the appropriate documents
   - the searching mechanism used to obtain and retrieve the relevant documents

Applications of Information retrieval:

1. Text Information Retrieval

   Terabytes of data are being cumulated on the internet which includes Facebook and Twitter data as well as Instagrams and other social networking sites. This vast repository may be mined, and controlled to some extent, to swerve public opinion in a candidate's favor (election strategy) or evaluate a product's performance (marketing and sales strategy)

2. Multimedia Information Retrieval

   Storage, indexing, search, and delivery of **multimedia** data such as images, videos, sounds, 3D graphics or their combination. By definition, it includes works on, for example, extracting descriptive features from images, reducing high-dimensional indexes into low-dimensional ones, defining new similarity metrics, efficient delivery of the retrieved data, and so forth. **System**s that provide all or part of the above functionalities are **multimedia retrieval system**s.

   The Google image search engine is a typical example of such a **system**. A video-on-demand site that allows people to search movies by their titles is another example

DATABASE USERS AND ADMINISTRATORS:

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

Database Users and User Interfaces

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

- **Naive users** are unsophisticated users who interact with the system by invoking one of the application programs that have been written previously.
  - Bank tellers check account balances and post withdrawals and deposits.
  - Reservation agents for airlines, hotels, and car rental companies check availability for a given request and make reservations.
  - Employees at receiving stations for shipping companies enter package identifications via bar codes and descriptive information through buttons to update a central database of received and in-transit packages.

- As another example, consider a student, who during class registration period, wishes to register for a class by using a Web interface. Such a user connects to a Web application program that runs at a Web server. The application first verifies the identity of the user, and allows her to access a form where she enters the desired information. The form information is sent back to the Web application at the server, which then determines if there is room in the class (by retrieving information from the database) and if so adds the student information to the class master in the database.

- **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 with minimal programming effort.
- **Sophisticated users** interact with the system without writing programs. Instead, they form their requests either using a database query language or by using tools such as data analysis software. Analysts who submit queries to explore data in the database fall in this category.
- **Specialized users** are sophisticated users who write specialized database applications that do not fit into the traditional data-processing framework. Among these applications are computer-aided design systems, knowledge-base and expert systems, systems that store data with complex data types (for example, graphics data and audio data), and environment-modeling systems.

- **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). 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:
    - Periodically backing up the database, either onto tapes or onto remote servers, to prevent loss of data in case of disasters such as flooding.
    - Ensuring that enough free disk space is available for normal operations, and upgrading disk space as required.
    - Monitoring jobs running on the database and ensuring that performance is not degraded by very expensive tasks submitted by some users.

As a whole, the DBA jobs are

- Creating primary database storage structures
- Modifying the structure of the database
- Monitoring database performance and efficiently
- Transferring data between the database and external file
- Monitoring and reestablishing database consistency
- Controlling and monitoring user access to the database
- Manipulating the physical location of the database.

HISTORY OF DATABASE SYSTEMS:

1950s and early 1960s:	
	Data processing using magnetic tapes for storage
	-Tapes provide only sequential access
	Punched cards for input
Late 1960s and 1970s:	
	Hard disks allow direct access to data
	Network and hierarchical data models in widespread use
	Ted Codd defines the relational data model
	-Would win the ACM Turing Award for this work
	-IBM Research begins System R prototype
	-UC Berkeley begins Ingres prototype
	High-performance (for the era) transaction processing
1980s:	
	Research relational prototypes evolve into commercial systems
	-SQL becomes industrial standard
	Parallel and distributed database systems
	Object-oriented database systems
1990s:	
	Large decision support and data-mining applications
	Large multi-terabyte data warehouses
	Emergence of Web commerce
2000s:	
	XML and XQuery standards
	Automated database administration

Information processing drives the growth of computers, as it has from the earliest days of commercial computers. In fact, automation of data processing tasks predates computers. Punched cards, invented by Herman Hollerith, were used at the very beginning of the twentieth century to record U.S. census data, And Mechanical systems were used to process the cards and tabulate results. Punched cards were later widely used as a means of entering data into computers. Techniques for data storage and processing have evolved over the years:

• 1950s and early 1960s:

Magnetic tapes were developed for data storage. Data processing tasks such as payroll were automated, with data stored on tapes. Processing of data consisted of reading data from one or more tapes and writing data to a new tape. Data could also be input from punched card decks, and output to printers. For example, salary raises were processed by entering the raises on punched cards and reading the punched card deck in synchronization with a tape containing the master salary details. The records had to be in the same sorted order. The salary raises would be added to the salary read from the master tape, and written to a new tape; the new tape would become the new master tape. Tapes (and card decks) could be read only sequentially, and data sizes were much larger than main memory; thus, data processing programs were forced to process data in a particular order, by reading and merging data from tapes and card decks.

• Late 1960s and 1970s:

Wide spread use of hard disks in the late1960s changed the scenario for data processing greatly, since hard disks allowed direct access to data. The position of data on disk was immaterial, since any location on disk could be accessed in just tens of milliseconds. Data were thus freed from the tyranny of sequentially. 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.

A landmark paper by Codd [1970] defined the relational model and nonprocedural ways of querying data in the relational model, and relational databases were born. The simplicity of the relational model and the possibility of hiding implementation details completely from the programmer were enticing indeed. Codd later won the prestigious Association of Computing Machinery Turing Award for his work.

•1980s:

Although academically interesting, the relational model was not used in practice initially, because of its perceived performance disadvantages; relational databases could not match the performance of existing network and hierarchical databases. That changed with System R, a ground breaking project at IBM Research that developed techniques for the construction of an efficient relational database system. Excellent overviews of System R are provided by Astrahan et al. [1976] and Chamberlin et al. [1981]. The fully functional System R prototype led to IBM's first relational database product, SQL/DS. At the same time, the Ingres system was being developed at the University of California at Berkeley. It led to a commercial product of the same name. Initial commercial relational database systems, such as IBM DB2, Oracle, Ingres, and DEC Rdb, played a major role in advancing techniques for efficient processing of declarative queries.

By the early 1980s, relational databases had become competitive with network and hierarchical database systems even in the area of performance. Relational databases were so easy to use that they eventually replaced network and hierarchical databases; programmers using such databases were forced to deal with many low-level implementation details, and had to code their queries in a procedural fashion. Most importantly, they had to keep efficiency in mind when designing their programs, which involved a lot of effort.

In contrast, in a relational database, almost all these low-level tasks are carried out automatically by the database, leaving the programmer free to work at a logical level. Since attaining dominance in the 1980s, the relational model has reigned supreme among data models. 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 for decision support applications, which are query-intensive, yet the main stay of databases in the 1980s was transaction-processing applications, which are update-intensive. Decision support and querying re-emerged as a major application area for databases. Tools for analyzing large amounts of data saw large growths in usage. Many database vendors introduced parallel database products in this period. Database vendors also began to add object-relational support to their databases.

• 1990s:

The major event of the 1990s 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.

• 2000s:

The first half of the 2000s saw the emerging of XML and the associated query language XQuery as a new database technology. Although XML is widely used for data exchange, as well as for storing certain complex data types, relational databases still form the core of a vast majority of large-scale database applications. In this time period we have also witnessed the growth in "autonomic-computing/auto-admin" techniques for minimizing system administration effort. This period also saw a significant growth in use of open-source database systems, particularly PostgreSQL and MySQL. The latter part of the decade has seen growth in specialized databases for data analysis, in particular column-stores, which in effect store each column of a table as a separate array, and highly parallel database systems designed for analysis of very large data sets. Several novel distributed data-storage systems have been built to handle the data management requirements of very large Web sites such as Amazon, Facebook, Google, Microsoft and Yahoo!, and some of these are now offered as Web services that can be used by application developers. There has also been substantial work on management and analysis of streaming data, such as stock-market ticker data or computer network monitoring data. Data-mining techniques are now widely deployed; example applications include Web-based product-recommendation systems and automatic placement of relevant advertisements on Web pages.

When Not to Use a DBMS

In spite of the advantages of using a DBMS, there are a few situations in which a DBMS may involve unnecessary overhead costs that would not be incurred in traditional file processing. The overhead costs of using a DBMS are due to the following:

- High initial investment in hardware, software, and training
- The generality that a DBMS provides for defining and processing data
- Overhead for providing security, concurrency control, recovery, and integrity functions

Therefore, it may be more desirable to use regular files under the following circumstances:

- Simple, well-defined database applications that are not expected to change at all
- Stringent, real-time requirements for some application programs that may not be met because of DBMS overhead.
- Embedded systems with limited storage capacity, where a general-purpose DBMS would not fit.
- No multiple-user access to data

Certain industries and applications have elected not to use general-purpose DBMSs.

For example, many computer-aided design (CAD) tools used by mechanical and civil engineers have proprietary file and data management software that is geared for the internal manipulations of drawings and 3D objects. Similarly, communication and switching systems designed by companies like AT&T were early manifestations of database software that was made to run very fast with hierarchically organized data for quick access and routing of calls. Similarly, GIS implementations often implement their own data organization schemes for efficiently implementing functions related to processing maps, physical contours, lines, polygons, and so on. General-purpose DBMSs are inadequate for their purpose.

<u>UNIT -II</u>

The Relational Data Model

- The relational model was first proposed by E.F. Codd in 1970 and the first such system (notably INGRES and System/R) was developed in 1970s. The relational model is now the dominant model for commercial data processing applications. By far the most likely data model in which you'll implement a database application today.

- Of historical interest: the relational model is not the first implementation data model. Prior to that were the network data model, exemplified by CODASYL, and the hierarchical model, implemented by IBM's IMS.

- Salient features of the relational model:
  – Conceptually simple; the fundamentals are intuitive and easy to pick up.

  – Powerful underlying theory: the relational model is the only database model that is powered by formal mathematics, which results in excellent dividends when developing database algorithms and techniques.

  – Easy-to-use database language: though not formally part of the relational model, part of its success is due to SQL, the default language for working with relational databases.

Relational database systems are the most common DBMS today. These relational DBMSs organize data into separate structures called **tables,** which can be **linked** via common information to make data storage more efficient. A DBMS is like a traditional filing system in that it stores individual groups and pieces of information. Like a filing system, a DBMS consists of separate components, like the cabinet, drawers and folders. A relational DBMS has the following basic components:

· **database** - the complete collection of information

· **tables** - a group of data items with a common theme

· **records** - an individual data item

· **fields** - a separate piece of information which describe the data item

Basic Structure

- A relational database is a collection of tables.
  – Each table has a unique name.

– Each table consists of multiple rows.

– Each row is a set of values that by definition are related to each other in some way; these values conform to the attributes or columns of the table.

– Each attribute of a table defines a set of permitted values for that attribute; this set of permitted set is the domain of that attribute.

- This definition of a database table originates from the pure mathematical concept of a relation, from which the term "relational data model" originates.
  – Formally, for a table r with n attributes $a_1 \ldots a_n$, each attribute a_k has a domain D_k, and any given row of r is an n-tuple $(v_1, \ldots, v_n)$ such that $v_k \in D_k$.

  – Thus, any instance of table r is a subset of the Cartesian product $D_1 \times \cdot \cdot \cdot \times D_n$.

  – We require that a domain D_k be atomic - that is, we do not consider the elements of D_k to be breakable into subcomponents.

[We require that, for all relations *r*, the domains of all attributes of *r* be atomic. A domain is **atomic** if elements of the domain are considered to be indivisible units. For example, suppose the table *instructor* had an attribute *phone number*, which can store a set of phone numbers corresponding to the instructor. Then the domain of *phone number* would not be atomic, since an element of the domain is a set of phone numbers, and it has subparts, namely the individual phone numbers in the set.]

– A possible member of any domain is null - that is, an unknown or non-existent value; in practice, we try to avoid the inclusion of null in our databases because they can cause a number of practical issues.

[The **null** value is a special value that signifies that the value is unknown or does not exist. For example, suppose as before that we include the attribute *phone number* in the *instructor* relation. It may be that an instructor does not have a phone number at all, or that the telephone number is unlisted. We would then have to use the null value to signify that the value is unknown or does not exist.]

- More notation:
  – Given a tuple t that belongs to a relation r, we can say that $t \in R$ since after all r is a set of tuples.

    * By "set of tuples" we do mean the mathematical concept of a set; thus order doesn't matter.

  – To talk about a specific attribute a_k of t, we write $t[a_k]$.

  – This notation also applies to a set of attributes A — the notation t[A] refers to the "sub-tuple" of t consisting only of the attributes in A.

  – Alternatively, t[k] can refer to that same attribute of t, as long as we are consistent about how attributes are ordered in the relation.

In general, a row in a table represents a ***relationship*** among a set of values. Since a table is a collection of such relationships, there is a close correspondence between the concept of ***table*** and the mathematical concept of ***relation*,** from which the relational data model takes its name.

In mathematical terminology, **a *tuple*** is simply a sequence (or list) of values. A relationship between *n* values is represented mathematically by **an *n-tuple*** of values, i.e., a tuple with *n* values, which corresponds to a **row** in a table.

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

Example of Employee table of information in relational database

Eno	**Ename**	**Desgn**	**Salary**	**Dno**	**Dname**	**Joindate**
20	Naresh	Clerk	3000	3	Sales	20-jul-12
30	Suresh	Cashier	5000	5	Servicing	10-aug-12
50	Kumar	Supdt	7000	1	Marketing	03-oct-10
25	Amar	Programmer	5000	2	Computer	16-mar-10

The table gives the information about a list of employees working in an organization. Every employee number (ENO) corresponds to a particular employee in the organization. ENO, ENAME, DESGN, SDALARY, DNO, DNAME and JOINDATE are called the columns (attributes) of the table EMPLOYEE. For each of the above attributes, there is a set of values, which is known as the DOMAIN of the attribute.

For example, for the DESGN attributes, the domain is the set of all designations of employees and for DNAME attribute, the domain is the set of all department names and so on.

In the above relation EMPLOYEE, there are four tuples (Rows), each row identifying the details of a particular employee with respect to attributes.

For example, the first row specifies that employee number is 20, named Naresh, who is clerk in the Sales department earning 3000 and joined the organization on 20th July 1995.

Terminologies:

Relation : A relation is defined as a table with columns and rows. According to E.F. Codd data can be stored in form of a two- dimensional (2D) table.

Row – will be the record and column – will be the attributes.

Attributes : It is defined as a named columns of a relation. Attributes are nothing else but column of the relation. So, the relation hold the information about the objects to represents the entire database. It can appear in any order it cannot change the meaning of the database.

Each entity has certain characteristics knows as attributes.

Tuple : "Row of relation (Table) is referred as tuple. Tuple having a set of 'n' numbers of attributes are called as n-tuple.

Domain: The values for an attribute or a column are drawn from a set of permitted values known as a Domain. The domain of an attribute contains the set of values that the attribute may assume.

In the relation model, no two rows of relation are identical and the ordering of rows is not significant, the domain of Die is 1, 2,3,4,5 and 6. Similarly, domain of coin is Head or Tail.

Degree of a table:

A relationship's degree indicates the number of associated entities or attributes.

A **unary relationship** exists when an association is maintained within a single entity. A **binary relationship** exists when two entities are associated. A **ternary relationship** exists when three entities are associated. Although higher degrees exist, they are rare and are specifically named.

Unary relationship: Course ————— participant

Binary Relationship: Lecturer ——— Batches ——— Class

Ternary relationship: Contributor — CFR —— recipient Fund

Extension (or state) of a relation: it is defined as the set of row that appear in that relation at any given instant of time. It various with time ie, it changes as the rows are created, deleted and updated

Intension of a relation: it is a permanent part of the table and is independent of time. So, it corresponds to what is specified in the relational schema.

Cardinality: it is defined as the number of rows or tuples it contains. So it changes as we add new rows or delete them. It is a property of the extension of the table or relation.

NULL :

The NULL value indicates that the value does not exist or is not known. An unknown value may be either missing or not known.

Missing – the value does exist, but we do not have that information

Not known – we do not known whether or not the value actually exist

An attribute takes a **null** value when an entity does not have a value for it. The null value may indicate "not applicable" – that is, that the value does not exist for the entity.

Relational database: It is defined as the collection of normalized or structured relations with distinct relation names.

Attributes - types of attributes

It is defined as named columns of a relation. Attributes are nothing else but column of the relation. So, the relation hold the information about the objects to represents the entire database. It can appear in any order it cannot change the meaning of the database.

Each entity has certain characteristics knows as attributes. For instance the student entity might include the following attributes, Student name, Roll Number etc.

For each attribute, there is a set of permitted values, called the **domain,** or **value set**, of that attribute. An attribute of an entity set is a function that maps from the entity set into a domain. Since an entity set may have several attributes, each entity can be described by a set of (attribute, data value) pairs, one pair for each attribute of the entity set.

The attributes can be classified in to

1. Simple attributes
2. Complex/ composite attributes
3. Single – valued attributes
4. Multi - valued attributes
5. Derived attribute
6. Null Attribute

Mapping Cardinalities:

Mapping cardinalities, or cardinality ratios, express the number of entities to which another entity can be associated via a relationship set. Mapping cardinalities are most useful in describing binary relationship sets, although they can contribute to the description of relationship sets that involve more than two entity sets.

One to one. An entity in A is associated with at most one entity in B, and an entity in B is associated with at most one entity in A.

One to many. An entity in A is associated with any number (zero or more) of entities in B. An entity in B, however, can be associated with at most one entity in A.

Many to one. An entity in A is associated with at most one entity in B. An entity in B, however, can be associated with any number (zero or more) of entities in A.

Many to many. An entity in A is associated with any number (zero or more) of entities in B, and an entity in B is associated with any number (zero or more) of entities in A.

Keys:

A relation always has a unique identifier, a field or group of fields (attributes) whose values are unique throughout all of the rows of the table. Each row is distinct and can be identified by the values of one or more of its attributes called **key.**

A key allows us to identify a set of attributes that suffice to distinguish entities from each other. Keys also help uniquely identify relationships, and thus distinguish relationships from each other.

The keys can be categorized in to

Superkey: A superkey is a set of one or more attributes that, taken collectively, allow us to identify uniquely a tow in a table.

For example, the customer-id attribute of the entity set customer is sufficient to distinguish one customer entity from another. Thus, customer-id is a superkey. Similarly, the combination of customer-name and customer-id is a superkey for the entity set customer. The customer-name attribute of customer is not a superkey, because several people might have the same name.

For example, the *ID* attribute of the relation *instructor* is sufficient to distinguish one instructor tuple from another. Thus, *ID* is a superkey. The *name* attribute of *instructor*, on the other hand, is not a superkey, because several instructors might have the same name.

Formally, let *R* denote the set of attributes in the schema of relation *r* . If we say that a subset *K* of *R* is a *superkey* for *r* , we are restricting consideration to instances of relations *r* in which no two distinct tuples have the same values on all attributes in *K*. That is, if t_1 and t_2 are in *r* and $t_1 \ = t_2$, then $t_1.K \ = t_2.K$.

Candidate key: Minimal superkeys are called candidate keys.

If K is a superkey, then so is any superset of K. We are often interested in superkeys for which no proper subset is a superkey. It is possible that several distinct sets of attributes could serve as a candidate key.

Suppose that a combination of customer-name and customer-street is sufficient to distinguish among members of the customer entity set. Then, both {customer-id} and {customer-name, customer-street} are candidate keys. Although the attributes customerid and customer-name together can distinguish customer entities, their combination does not form a candidate key, since the attribute customer-id alone is a candidate key.

A superkey may contain extraneous attributes. For example, the combination of *ID* and *name* is a superkey for the relation *instructor*. If *K* is a superkey, then so is any superset of *K*. We are often interested in superkeys for which no proper subset is a superkey. Such minimal superkeys are called **candidate keys**.

It is possible that several distinct sets of attributes could serve as a candidate key. Suppose that a combination of *name* and *dept name* is sufficient to distinguish among members of the *instructor* relation. Then, both *{ID}* and *{name*, *dept name}* are candidate keys. Although the attributes *ID* and *name* together can distinguish *instructor* tuples, their combination, *{ID*, *name}*, does not form a candidate key, since the attribute *ID* alone is a candidate key.

Primary key:

when one or more number of attributes uniquely identify the row is called primary key. It cannot contain any null value. It has minimal number of attributes

We shall use the term **primary key** to denote a candidate key that is chosen by the database designer as the principal means of identifying tuples within a relation. A key (whether primary, candidate, or super) is a property of the entire relation, rather than of the individual tuples. Any two individual tuples in the relation are prohibited from having the same value on the key attributes at the same time. The designation of a key represents a constraint in the real-world enterprise being modeled.

The primary key should be chosen such that its attribute values are never, or very rarely, changed.

For instance, the address field of a person should not be part of the primary key, since it is likely to change. Social-security numbers, on the other hand, are guaranteed never to change. Unique identifiers generated by enterprises generally do not change, except if two enterprises merge; in such a case the same identifier may have been issued by both enterprises, and a reallocation of identifiers may be required to make sure they are unique.

t is customary to list the primary key attributes of a relation schema before the other attributes; for example, the *dept name* attribute of *department* is listed first, since it is the primary key. Primary key attributes are also underlined.

Alternate key :

The alternate key of any table are those candidate keys which are not currently selected as the primary key.

Example : The combination of Rollno and name as candidate key. Rollno alone is primary key and name alone is alternate key.

Composite key :

A primary key that is made up of more than one attributes known as a composite key.

Ex; Project (P_id, E_id, E_name, Hours_works)

Secondary key :

Other than primary key, candidate key and alternate key attributes are called secondary key. An attribute (or) Combination of attributes used strictly for data retrieval purposes.

Foreign key :

The attribute of one table that references a primary key of the another table is called s a foreign key. Foreign key provide a method for maintaining integrity and for navigation between different instances of an tables. Foreign key values must be matched by the corresponding primary key values.

A relation, say r_1, may include among its attributes the primary key of an-other relation, say r_2. This attribute is called a **foreign key** from r_1, referencing r_2.
The relation r_1 is also called the **referencing relation** of the foreign key dependency, and
The relation r_2 is called the **referenced relation** of the foreign key.

For example, the attribute *dept name* in *instructor* is a foreign key from *instructor*, referencing *depart-ment*, since *dept name* is the primary key of *department*. In any database instance, given any tuple, say t_a , from the *instructor* relation, there must be some tuple, say t_b , in the *department* relation such that the value of the *dept name* attribute of t_a is the same as the value of the primary key, *dept name*, of t_b .

The constraint from *section* to *teaches* is an example of a **referential integrity constraint**; a referential integrity constraint requires that the values appearing in specified attributes of any tuple in the referencing relation also appear in specified attributes of at least one tuple in the referenced relation.

Database Schema

When we talk about a database, we must differentiate between the **database schema**, which is the logical design of the database, and the **database instance**, which is a snapshot of the data in the database at a given instant in time.

The concept of a relation corresponds to the programming-language notion of a variable, while the concept of a **relation schema** corresponds to the programming-language notion of type definition.

Dept Name	Building	Budget
Biology	Watson	90000
Comp. Sci.	Taylor	100000
Elec. Eng.	Taylor	85000

In general, a relation schema consists of a list of attributes and their corresponding domains. We shall not be concerned about the precise definition of the domain of each attribute

The concept of a relation instance corresponds to the programming-language notion of a value of a variable. The value of a given variable may change with time similarly the contents of a relation instance may change with time as the relation is updated. In contrast, the schema of a relation does not generally change.

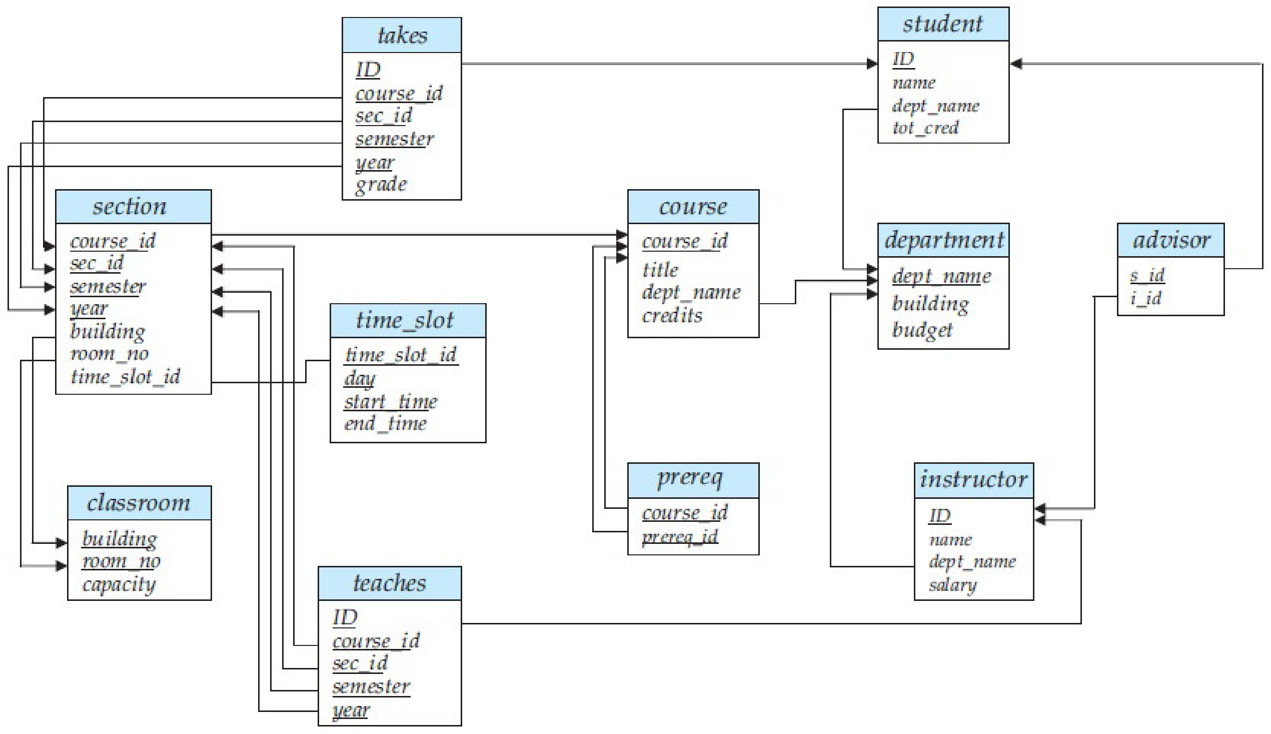
Although it is important to know the difference between a relation schema and a relation instance, we often use the same name, such as *instructor*, to refer to both the schema and the instance. Where required, we explicitly refer to the schema or to the instance, for example "the *instructor* schema," or "an instance of the *instructor* relation." However, where it is clear whether we mean the schema or the instance, we simply use the relation name.

Consider the *department* relation of above Figure. The schema for that relation is

department (*dept name*, *building*, *budget*)

Note that the attribute *dept name* appears in both the *instructor* schema and the *department* schema. This duplication is not a coincidence. Rather, using common attributes in relation schemas is one way of relating tuples of distinct relations. For example, suppose we wish to find the information about all the instructors who work in the Watson building. We look first at the *department* relation to find the *dept name* of all the departments housed in Watson. Then, for each such department, we look in the *instructor* relation to find the information about the instructor associated with the corresponding *dept name*.

Relational Schema Diagram Notation



Schema diagram for the university database.

Figure 1: Sample relational schema diagram for our sample university database.

- A relational database schema can be depicted pictorially through a relational schema diagram. Yes, it’s yet another notation like E-R and UML, but this notation is very focused and specific to the relational data model:

  – Relations are drawn as boxes with the relation name above the box.

  – A relation’s attributes are listed within its box.

  – The attributes that belong to the relation’s primary key (if it has one) are listed first, with a line separating this primary key from the other attributes and their background shaded in gray.

  – Foreign key dependencies are illustrated as arrows from the foreign key attributes of the referencing relation to the primary key attributes of the referenced relation.

RELATIONAL QUERY LANGUAGES

A query language is a language in which a user requests information from the database. These languages are usually on a level higher than that of a standard programming language.

Query languages can be categorized as either

Procedural or
Nonprocedural.

In a procedural language, the user instructs the system to perform a sequence of operations on the database to compute the desired result.

In an Nonprocedural language, the user describes the desired information without giving a specific procedure for obtaining that information.

Query languages used in practice include elements of both the procedural and the non procedural approaches.

There are a number of "pure" query languages:

The relational algebra is procedural,
whereas the tuple relational calculus and domain relational calculus are non procedural.

These query languages are terse and formal, lacking the "syntactic sugar" of commercial languages, but they illustrate the fundamental techniques for extracting data from the database

The relational algebra consists of a set of operations that take one or two relations as input and produce a new relation as their result. The relational calculus uses predicate logic to define the result desired without giving any specific algebraic procedure for obtaining that result.

RELATIONAL OPERATIONS

All procedural relational query languages provide a set of operations that can be applied to either a single relation or a pair of relations.

These operations have the nice and desired property that their result is always a single relation. This property allows one to combine several of these operations in a modular way. Specifically, since the result of a relational query is itself a relation, relational operations can be applied to the results of queries as well as to the given set of relations.

The *join* operation allows the combining of two relations by merging pairs of tuples, one from each relation, into a single tuple. There are a number of different ways to join relations

In general, the natural join operation on two relations matches tuples whose values are the same on all attribute names that are common to both relations.

The *Cartesian product* operation combines tuples from two relations, but unlike the join operation, its result contains *all* pairs of tuples from the two relations, regardless of whether their attribute values match.

Because relations are sets, we can perform normal set operations on relations.

The *union* operation performs a set union of two "similarly structured" tables (say a table of all graduate students and a table of all undergraduate students). For example, one can obtain the set of all students in a department. Other set operations, such as *intersection* and *set difference* can be performed as well.

Given this simple and restricted data structure, it is possible to define some very powerful relational operators which, from the users' point of view, act in parallel' on all entries in a table simultaneously, although their implementation may require conventional processing.

Codd originally defined eight relational operators.

1. SELECT originally called RESTRICT
2. PROJECT
3. JOIN
4. PRODUCT
5. UNION
6. INTERSECT
7. DIFFERENCE
8. DIVIDE

The most important of these are (1), (2), (3) and (8), which, together with some other aggregate functions, are powerful enough to answer a wide range of queries. The eight operators will be described as general procedures - i.e. not in the syntax of SQL or any other relational language. The important point is that they define the result required rather than the detailed process of obtaining it - what but not how.

SELECT
RESTRICTS the rows chosen from a table to those entries with specified attribute values.

```
SELECT item
FROM stock_level
WHERE quantity > 100
```

constructs a new, logical table - an unnamed relation - with one column per row (i.e. item) containing all rows from stock_level that satisfy the WHERE clause.

PRODUCT
Builds a relation from two specified relations consisting of all possible combinations of rows, one from each of the two relations.

For example, consider two relations, A and B, consisting of rows:

```
A: a    B: d    =>   A product B: a   d
   b       e                      a   e
   c                              b   d
                                  b   e
                                  c   d
```

PROJECT
Selects rows made up of a sub-set of columns from a table.

```
PROJECT stock_item
OVER item AND description
```

produces a new logical table where each row contains only two columns - item and description. The new table will only contain distinct rows from stock_item; i.e. any duplicate rows so formed will be eliminated.

JOIN
Associates entries from two tables on the basis of matching column values.

```
JOIN stock_item
WITH stock_level
OVER item
```

It is not necessary for there to be a one-to-one relationship between entries in two tables to be joined - entries which do not match anything will be eliminated from the result, and entries from one table which match several entries in the other will be duplicated the required number of times.

UNION
Builds a relation consisting of all rows appearing in either or both of the two relations.

For example, consider two relations, A and B, consisting of rows:

```
A: a    B: a    =>    A union B: a
   b       e                     b
   c                             c
                                 e
```

INTERSECT
Builds a relation consisting of all rows appearing in both of the two relations.

For example, consider two relations, A and B, consisting of rows:

```
A: a    B: a    =>    A intersect B: a
   b       e
   c
```

DIFFERENCE
Builds a relation consisting of all rows appearing in the first and not in the second of the two relations.

For example, consider two relations, A and B, consisting of rows:

```
A: a    B: a    =>  A - B: b   and   B - A: e
   b       e               c
   c
```

DIVIDE
Takes two relations, one binary and one unary, and builds a relation consisting of all values of one column of the binary relation that match, in the other column, all values in the unary relation.

```
A: a  x    B: x    =>    A divide B: a
   a  y       y
   a  z
   b  x
   c  y
```

Of the relational operators 3.2.4. to 3.2.8.defined by Codd, the most important is DIVISION. For example, suppose table A contains a list of suppliers and commodities, table B a list of all commodities bought by a company. Dividing A by B produces a table listing suppliers who sell all commodities.

FORMAL RELATIONAL QUERY LANGUAGES

Two mathematical Query Languages form the basis for "real" languages (e.g. SQL), and for implementation:

Relational Algebra: More operational, very useful for representing execution plans.

Relational algebra is one of the two formal query languages associated with the relational model. Queries in algebra are composed using a collection of operators. A fundamental property is that every operator in the algebra accepts (one or two) relation instances as arguments and returns a relation instance as the result.

This property makes it easy to compose operators to form a complex query—a relational algebra expression is recursively defined to be a relation,

a unary algebra operator applied to a single expression, or

a binary algebra operator applied to two expressions.

We describe the basic operators of the algebra (selection, projection, union, cross-product, and difference), as well as some additional operators that can be defined in terms of the basic operators but arise frequently enough to warrant special attention, in the following sections.

Each relational query describes a step-by-step procedure for computing the desired answer, based on the order in which operators are applied in the query. The procedural nature of the algebra allows us to think of an algebra expression as a recipe, or a plan, for evaluating a query, and relational systems in fact use algebra expressions to represent query evaluation plans.

Relational Calculus: Lets users describe what they want, rather than how to compute it.
(Non-operational, declarative.)

Relational calculus is an alternative to relational algebra. In contrast to the algebra, which is procedural, the calculus is nonprocedural, or declarative, in that it allows us to describe the set of answers without being explicit about how they should be computed.

Relational calculus has had a big influence on the design of commercial query languages such as SQL and, especially, Query-by-Example (QBE).

The variant of the calculus that we present in detail is called the
tuple relational calculus (TRC). Variables in TRC take on tuples as values.

In another variant, called the domain relational calculus (DRC), the variables range over field values.

TRC has had more of an influence on SQL, while DRC has strongly influence QBE.

Relational Algebra

The relational algebra is a procedural query language. It consists of a set of operations that take one or two relations as input and produce a new relation as their result. The fundamental operations in the relational algebra are select, project, union, set difference, Cartesian product, and rename.

Fundamental Operations

The select, project, and rename operations are called **unary operations**, because they operate on one relation.

The other three operations operate on pairs of relations and are, therefore, called **binary operations.**

The primary operations of relational algebra are as follows:

- Select
- Project
- Union
- Set different
- Cartesian product
- Rename
    - In **addition** to the fundamental operations, there are several other **operations**—namely, set intersection, natural join and assignment.

Formal Definition of the Relational Algebra

The operations in Section 6.1.1 allow us to give a complete definition of an expres-sion in the relational algebra. A basic expression in the relational algebra consists of either one of the following:

- A relation in the database
- A constant relation

A constant relation is written by listing its tuples within { }, for example

{ (22222, Einstein, Physics, 95000), (76543, Singh, Finance, 80000) }.

A general expression in the relational algebra is constructed out of smaller sub expressions. Let E_1 and E_2 be relational-algebra expressions. Then, the following are all relational-algebra expressions:

- $E1 \cup E2$
- $E1 - E2$
- $E1 \times E2$
- $s_P (E_1)$, where P is a predicate on attributes in E_1
- $_S(E_1)$, where S is a list consisting of some of the attributes in E_1
- $r_x (E_1)$, where x is the new name for the result of E_1

The SELECT Operation (σ)

The SELECT operator is used to choose a subset of the tuples (rows) from a relation that satisfies a selection condition, acting as a filter to retain only tuples that fulfills a qualifying requirement.

The SELECT operator is relational algebra is denoted by the symbol σ (sigma).

The syntax for the SELECT statement is then as follows:

$$\sigma_{\text{<Selection condition>}}(R)$$

The σ would represent the SELECT command

The <selection condition> would represent the condition for selection.

The (R) would represent the Relation or the Table from which we are making a selection of the tuples.

To implement the SELECT statement in SQL, we take a look at an example in which we would like to select the EMPLOYEE tuples whose employee number is 7, or those whose date of birth is before 1980…

$$\sigma_{empno=7}(\text{EMPLOYEE})$$

$$\sigma_{dob<\text{'01-Jan-1980'}}(\text{EMPLOYEE})$$

The SQL implementation would translate into:

SELECT empno FROM EMPLOYEE WHERE empno=7

SELECT dob FROM EMPLOYEE WHERE DOB < ’01-Jan-1980′

The PROJECT Operation (∏)

This operator is used to reorder, select and get rid of attributes from a table. At some point we might want only certain attributes in a relation and eliminate others from our query result. Therefore the PROJECT operator would be used in such operations.

The symbol used for the PROJECT operation is ∏ (pi).

The general syntax for the PROJECT operator is:

$$\prod_{\text{<attribute list>}}(R\ )$$

∏ would represent the PROJECT.

<attribute list> would represent the attributes(columns) we want from a relational.

(R) would represent the relation or table we want to choose the attributes from.

To implement the PROJECT statement in SQL, we take a look at an example in which we would like to choose the Date of Birth (dob) and Employee Number (empno) from the relation EMPLOYE…

$$\prod_{\textbf{dob, empno}}(\textbf{EMPLOYEE})$$

In SQL this would translate to:

SELECT dob, empno FROM EMPLOYEE

The RENAME Operator ρ (rho):

The RENAME operator is used to give a name to results or output of queries, returns of selection statements, and views of queries that we would like to view at some other point in time:

The RENAME operator is symbolized by ρ (rho).

The general syntax for RENAME operator is: $\rho_{s(B1, B2, B3, \ldots Bn)}\ (R)$

ρ is the RENAME operation.

S is the new relation name.

- $B_1, B_2, B_3, \ldots B_n$ are the new renamed attributes (columns).

R is the relation or table from which the attributes are chosen.

To implement the RENAME statement in SQL, we take a look at an example in which we would like to choose the Date of Birth and Employee Number attributes and RENAME them as 'Birth_Date' and 'Employee_Number' from the EMPLOYE E relation…

$$_{\textbf{s(Birth\_Date, Employee\_Number)}}(\textbf{EMPLOYEE}) \leftarrow \prod_{\textbf{dob, empno}}(\textbf{EMPLOYEE})$$

The arrow symbol ← means that we first get the PROJECT operation results on the right side of the arrow then apply the RENAME operation on the results on the left side of the arrow.

In SQL we would translate the RENAME operator using the SQL 'AS' statement:

SELECT dob AS 'Birth_Date', empno AS 'Employee_Numb er' FROM EMPLOYEE

UNION:

The UNION operation on relation A UNION relation B designated as **A ∪ B**, joins or includes all tuples that are in A or in B, eliminating duplicate tuples. The SQL implementation of the UNION operations would be as follows:

- A and B must have the same quantity of attributes.
- Attribute domains must be compatible.
- Duplicate tuples gets automatically eliminated.

UNION

RESULT ← **A ∪ B**

SQL Statement: SELECT * From A UNION SELECT * From B

The Set-Difference Operation

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

Ex: $\prod_{\textbf{customer name}}$**(depositor) -** $\prod_{\textbf{customername}}$**(borrower)**

As with the union operation, we must ensure that set differences are taken between *compatible* relations. Therefore, for a set-difference operation $r - s$ to be valid, we require that the relations r and s be of the same arity, and that the domains of the ith attribute of r and the ith attribute of s be the same, for all i.

CARTESIAN PRODUCT Operator

The CARTERSIAN PRODUCT operator, also referred to as the cross product or cross join, creates a relation that has all the attributes of A and B, allowing all the attainable combinations of tuples from A and B in the result. The CARTERSIAN PRODUCT A and B is symbolized by X as in A X B.

Let there be Relation $A(A_1, A_2)$ and Relation $B(B_1, B_2)$

The CARTERSIAN PRODUCT C of A and B which is A X B is

C = A X B

$C = (A_1B_1, A_1B_2, A_2B_1, A_2B_2)$

The SQL implementation would be something like:

SELECT A.dob, B.empno from A, B

INTERSECTION:

The INTERSECTION operation on a relation A INTERSECTION relation B, designated by **A ∩ B**, includes tuples that are only in A and B. In other words only tuples belonging to A and B, or shared by both A and B are included in the result. The SQL implementation of the INTERSECTION operations would be as follows:

INTERSECTION

RESULT ← **A ∩ B**

SQL Statement:

SELECT dob From A INTERSECT SELECT dob from B

Additional Relational – Algebra Operations:

The Set-Intersection Operation

The first additional relational-algebra operation that we shall define is **set inter-section** (∩). Suppose that we wish to find the set of all courses taught in both the Fall 2009 and the Spring 2010 semesters. Using set intersection, we can write

$$_{course\ id}(\ {}_{Ssemester = \text{“Fall”} \wedge year = 2009}(section)) \cap {}_{course\ id}(\ {}_{Ssemester = \text{“Spring”} \wedge year = 2010}(section))$$

Note that we can rewrite any relational-algebra expression that uses set in-tersection by replacing the intersection operation with a pair of set-difference operations as:

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

Thus, set intersection is not a fundamental operation and does not add any power to the relational algebra. It is simply more convenient to write $r \cap s$ than to write $- (r - s)$.

NATURAL JOIN Operator

The NATURAL JOIN operation returns results that does not include the JOIN attributes of the second Relation B. It is not required that attributes with the same name be mentioned. The NATURAL JOIN operator is symbolized by:

A $\bowtie$ B,

SQL translation example where attribute dob is Date of Birth and empno is Employee Number:

SELECT A.dob, B.empno FROM A NATURAL JOIN B //where depno =5

We can always use the ‘where’ clause to further res trict our output and stop a Cartesian product output.

The Assignment Operation

It is convenient at times to write a relational-algebra expression by assigning parts of it to temporary relation variables. The **assignment** operation, denoted by ←, works like assignment in a programming language. To illustrate this operation, consider the definition of the natural-join operation. We could write *r s* as:

$$temp1 \leftarrow R \times S$$

$$temp2 \leftarrow \mathrm{s}r.A_1 = s.A_1 \wedge r.A_2 = s.A_2 \wedge \ldots \wedge r.A_n = s.A_n\ {}^{(temp1)}$$

$$result =_{R \cup S} (temp2)$$

The evaluation of an assignment does not result in any relation being displayed to the user. Rather, the result of the expression to the right of the ← is assigned to the relation variable on the left of the ←. This relation variable may be used in subsequent expressions.

With the assignment operation, a query can be written as a sequential program consisting of a series of assignments followed by an expression whose value is displayed as the result of the query. For relational-algebra queries, assignment must always be made to a temporary relation variable. Assignments to permanent relations constitute a database modification. Note that the assignment operation does not provide any additional power to the algebra. It is, however, a convenient way to express complex queries.

Extended Relational-Algebra Operations

We now describe relational-algebra operations that provide the ability to write queries that cannot be expressed using the basic relational-algebra operations. These operations are called **extended relational-algebra** operations.

Generalized Projection

The first operation is the **generalized-projection** operation, which extends the projection operation by allowing operations such as arithmetic and string functions to be used in the projection list. The generalized-projection operation has the form:

$$\prod_{F_1, F_2, \ldots, F_n}(E)$$

where E is any relational-algebra expression, and each of F_1, F_2, . . . , F_n is an arithmetic expression involving constants and attributes in the schema of E. As a base case, the expression may be simply an attribute or a constant. In general, an expression can use arithmetic operations such as $+$, $-$, $*$, and $\div$ on numeric valued attributes, numeric constants, and on expressions that generate a numeric result. Generalized projection also permits operations on other data types, such as concatenation of strings.

For example, the expression:

$$\prod_{ID, name, dept\_name, salary \div 12}(instructor)$$

gives the *ID*, *name*, *dept name*, and the monthly salary of each instructor.

Aggregation

The second extended relational-algebra operation is the aggregate operation $\mathcal{G}$, which permits the use of aggregate functions such as min or average, on sets of values.

Aggregate functions take a collection of values and return a single value as a result. For example, the aggregate function **sum** takes a collection of values and returns the sum of the values. Thus, the function **sum** applied on the collection:

$$\{1, 1, 3, 4, 4, 11\}$$

returns the value 24. The aggregate function **avg** returns the average of the values. When applied to the preceding collection, it returns the value 4. The aggregate function **count** returns the number of the elements in the collection, and returns 6 on the preceding collection. Other common aggregate functions include **min** and **max**, which return the minimum and maximum values in a collection; they return 1 and 11, respectively, on the preceding collection.

The collections on which aggregate functions operate can have multiple occurrences of a value; the order in which the values appear is not relevant. Such collections are called **multisets**. Sets are a special case of multisets where there is only one copy of each element.

To illustrate the concept of aggregation, we shall use the *instructor* relation. Suppose that we want to find out the sum of salaries of all instructors; the relational-algebra expression for this query is:

$$\mathcal{G}_{\mathbf{sum}(salary)}(instructor)$$

The symbol *G* is the letter G in calligraphic font; read it as “calligraphic G.” The relational-algebra operation *G* signifies that aggregation is to be applied, and its subscript specifies the aggregate operation to be applied. The result of the expression above is a relation with a single attribute, containing a single row with a numerical value corresponding to the sum of the salaries of all instructors.

Outer join Operations

The **outer-join** operation is an extension of the join operation to deal with missing information.

More generally, some tuples in either or both of the relations being joined may be “lost” in this way. The **outer join** operation works in a manner similar to the natural join operation we have already studied, but preserves those tuples that would be lost in an join by creating tuples in the result containing null values. We can use the *outer-join* operation to avoid this loss of information. There are actually three forms of the operation:

left outer join, denoted _;
right outerjoin, denoted _ ; and
full outer join, denoted _ .

All three forms of outer join compute the join, and add extra tuples to the result of the join.

The **left outer join** (_) takes all tuples in the left relation that did not match with any tuple in the right relation, pads the tuples with null values for all other attributes from the right relation, and adds them to the result of the natural join.

The **right outer join** (_) is symmetric with the left outer join: It pads tuples from the right relation that did not match any from the left relation with nulls and adds them to the result of the natural join. Thus, all information from the right relation is present in the result of the right outer join.

The **full outer join** (_) does both the left and right outer join operations, padding tuples from the left relation that did not match any from the right relation, as well as tuples from the right relation that did not match any from the left relation, and adding them to the result of the join.

RELATIONAL CALCULUS

In relational calculus, a query is expressed as a formula consisting of a number of variables and an expression involving these variables. It is up to the DBMS to transform these nonprocedural queries into equivalent, efficient, procedural queries. The concept of relational calculus was first proposed by Codd. The relational calculus is used to measure the selective power of relational languages. A language that can be used to produce any relation that can be derived using the relational calculus is said to be relationally complete.

Relation calculus, which in effect means calculating with relations, is based on predicate calculus, which is calculating with predicates. It is a formal language used to symbolize logical arguments in mathematics. Propositions specifying a property consist of an expression that names an individual object, and another expression, called the predicate, that stands for the property that the individual object possesses. If for instance, *p* and *q* are propositions, we can build other propositions *"not p", "p or q", "p and q"* and so on. In predicate calculus, propositions may be built not only out of other propositions but also out of elements that are not themselves propositions. In this manner we can build a proposition that specifies a certain property or characteristic of an object.

The Tuple Relational Calculus

When we write a relational-algebra expression, we provide a sequence of procedures that generates the answer to our query. The tuple relational calculus, by contrast, is a **nonprocedural** query language. It describes the desired information without giving a specific procedure for obtaining that information.

A query in the tuple relational calculus is expressed as:

$$\{t \mid P(t)\}$$

That is, it is the set of all tuples t such that predicate P is true for t. Following our earlier notation, we use $t[A]$ to denote the value of tuple t on attribute A, and we use $t \in r$ to denote that tuple t is in relation r.

Before we give a formal definition of the tuple relational calculus, we re-turn to some of the queries for which we wrote relational-algebra expressions in Section 6.1.1.

Formal Definition

We are now ready for a formal definition. A tuple-relational-calculus expression is of the form:

$$\{t \mid P\ (t\ )\}$$

where P is a *formula*. Several tuple variables may appear in a formula. A tuple variable is said to be a *free variable* unless it is quantified by a $\exists$ or $\forall$. Thus, in:

$$t \in\ instructor \wedge\ \exists\ s \in\ department(t[dept\_name] =\ s[dept\_name])$$

t is a free variable. Tuple variable *s* is said to be a *bound* variable.

A tuple-relational-calculus formula is built up out of *atoms*. An atom has one of the following forms:

$s \in r$, where *s* is a tuple variable and *r* is a relation (we do not allow use of the $\notin$ operator).

$s[x]$ $u[y]$, where *s* and *u* are tuple variables, *x* is an attribute on which *s* is defined, *y* is an attribute on which *u* is defined, and is a comparison operator ($<$, $\leq$, $=$, $=$, $>$, $\geq$); we require that attributes *x* and *y* have domains whose members can be compared by .

$s[x]$ c, where *s* is a tuple variable, *x* is an attribute on which *s* is defined, is a comparison operator, and *c* is a constant in the domain of attribute *x*.

We build up formulae from atoms by using the following rules:

An atom is a formula.

If P_1 is a formula, then so are $\neg P_1$ and (P_1).

• If P_1 and P_2 are formulae, then so are $P_1 \vee P_2$, $P_1 \wedge P_2$, and $P_1 \Rightarrow P_2$.

If $P_1(s)$ is a formula containing a free tuple variable *s*, and *r* is a relation, then

$$\exists\, s \in r\ (P_1(s)) \text{ and } \forall\, s \in r\ (P_1(s))$$

are also formulae.

As we could for the relational algebra, we can write equivalent expressions that are not identical in appearance. In the tuple relational calculus, these equiv-alences include the following three rules:

$P_1 \wedge P_2$ is equivalent to $\neg(\neg(P_1) \vee \neg(P_2))$.
$\forall\, t \in r\ (P_1(t))$ is equivalent to $\neg \exists\, t \in r\ (\neg P_1(t))$.
$P_1 \Rightarrow P_2$ is equivalent to $\neg(P_1) \vee P_2$.

Example Queries

Find the *ID*, *name*, *dept name*, *salary* for instructors whose salary is greater than $80,000:

$$\{t \mid t \in instructor \wedge t[salary] > 80000\}$$

Suppose that we want only the *ID* attribute, rather than all attributes of the *instructor* relation. To write this query in the tuple relational calculus, we need to write an expression for a relation on the schema (*ID*). We need those tuples on (*ID*) such that there is a tuple in *instructor* with the *salary* attribute > 80000. To express this request, we need the construct "there exists" from mathematical logic. The notation:

$$t \in r\ (Q(t))$$

means “there exists a tuple *t* in relation *r* such that predicate *Q*(*t*) is true.”

Using this notation, we can write the query “Find the instructor *ID* for each instructor with a salary greater than \$80,000” as:

$$\{t \mid \exists\ s \in instructor\ (t[ID] = s[ID]$$
$$\wedge\ s[salary] > 80000)\}$$

In English, we read the preceding expression as “The set of all tuples *t* such that there exists a tuple *s* in relation *instructor* for which the values of *t* and *s* for the *ID* attribute are equal, and the value of *s* for the *salary* attribute is greater than \$80,000.”

Consider the query that “Find all students who have taken all courses offered in the Biology department.” To write this query in the tuple relational calculus, we introduce the “for all” construct, denoted by ∀. The notation:

$$t \in r\ (Q(t))$$

means “*Q* is true for all tuples *t* in relation *r*.”

We write the expression for our query as follows:

$$\{t \mid \exists\ r \in student\ (r[ID] = t[ID])\ \wedge$$
$$(\forall\ u \in cour\ se\ (u[dept\ name] = \text{“ Biology”} \Rightarrow$$
$$s \in takes\ (t[ID] = s[ID]$$
$$s[course\ id] = u[course\ id]))\}$$

In English, we interpret this expression as “The set of all students (that is, (*ID*) tuples *t*) such that, for *all* tuples *u* in the *course* relation, if the value of *u* on attribute *dept name* is ’Biology’, then there exists a tuple in the *takes* relation that includes the student *ID* and the *course id*.”

Note that there is a subtlety in the above query: If there is no course offered in the Biology department, all student *ID*s satisfy the condition. The first line of the query expression is critical in this case —without the condition

$$r \in student\ (r[ID] = t[ID])$$

if there is no course offered in the Biology department, any value of *t* (including values that are not student *ID*s in the *student* relation) would qualif

Safety of Expressions

There is one final issue to be addressed. A tuple-relational-calculus expression may generate an infinite relation. Suppose that we write the expression:

$$\{t \mid \neg (t \in \textit{instructor})\}$$

There are infinitely many tuples that are not in *instructor*. Most of these tuples contain values that do not even appear in the database! Clearly, we do not wish to allow such expressions.

To help us define a restriction of the tuple relational calculus, we introduce the concept of the **domain** of a tuple relational formula, *P*.

Intuitively, the domain of *P*, denoted *dom*(*P*), is the set of all values referenced by *P*. They include values mentioned in *P* itself, as well as values that appear in a tuple of a relation mentioned in *P*. Thus, the domain of *P* is the set of all values that appear explicitly in *P* or that appear in one or more relations whose names appear in *P*. For example, $dom(t \in instructor \land t[salary] > 80000)$ is the set containing 80000 as well as the set of all values appearing in any attribute of any tuple in the *instructor* relation. Similarly, $dom(\neg (t \in i\ nstr\ uctor))$ is also the set of all values appearing in *instructor*, since the relation *i nstr uctor* is mentioned in the expression.

We say that an expression $\{t \mid P(t)\}$ is *safe* if all values that appear in the result are values from *dom*(*P*). The expression $\{t \mid \neg (t \in instructor)\}$ is not safe. Note that $dom(\neg (t \in instructor))$ is the set of all values appearing in *instructor*. However, it is possible to have a tuple *t* not in *instructor* that contains values that do not appear in *instructor*. The other examples of tuple-relational-calculus expressions that we have written in this section are safe

The number of tuples that satisfy an unsafe expression, such as $\{t \mid \neg (t \in instructor)\}$, could be infinite, whereas safe expressions are guaranteed to have finite results. The class of tuple-relational-calculus expressions that are allowed is therefore restricted to those that are safe.

Expressive Power of Languages

The tuple relational calculus restricted to safe expressions is equivalent in expressive power to the basic relational algebra (with the operators ∪, −, ×, s, and r, but without the extended relational operations such as generalized projection and aggregation (*G*)).

Thus, for every relational-algebra expression using only the basic operations, there is an equivalent expression in the tuple relational calculus, and for every tuple-relational-calculus expression, there is an equivalent relational-algebra expression.

We shall not prove this assertion here; the bibliographic notes contain references to the proof. Some parts of the proof are included in the exercises. We note that the tuple relational calculus does not have any equivalent of the aggregate operation, but it can be extended to support aggregation. Extending the tuple relational calculus to handle arithmetic expressions is straightforward.

The Domain Relational Calculus

A second form of relational calculus, called **domain relational calculus**, uses *domain* variables that take on values from an attributes domain, rather than values for an entire tuple. The domain relational calculus, however, is closely related to the tuple relational calculus.

Domain relational calculus serves as the theoretical basis of the widely used QBE language (see Appendix B.1), just as relational algebra serves as the basis for the SQL language.

Formal Definition

An expression in the domain relational calculus is of the form

$$\{< x_1,\ x_2, \ldots, x_n > \mid P(x_1,\ x_2, \ldots, x_n)\}$$

where $x_1, x_2, \ldots, x_n$ represent domain variables. P represents a formula composed of atoms, as was the case in the tuple relational calculus. An atom in the domain relational calculus has one of the following forms:

$< x_1, x_2, \ldots, x_n > \in r$, where r is a relation on n attributes and $x_1, x_2, \ldots, x_n$ are domain variables or domain constants.

$x\ y$, where x and y are domain variables and is a comparison operator ($<$, $\leq$, $=$, $=$,$>$, $\geq$). We require that attributes x and y have domains that can be compared by .

$x\ c$, where x is a domain variable, is a comparison operator, and c is a constant in the domain of the attribute for which x is a domain variable.

We build up formulae from atoms by using the following rules:

An atom is a formula.

If P_1 is a formula, then so are $\neg\ P_1$ and (P_1).

• If P_1 and P_2 are formulae, then so are $P_1 \lor P_2$, $P_1 \land P_2$, and $P_1 \Rightarrow P_2$.

If $P_1(x)$ is a formula in x, where x is a free domain variable, then

$$x\ (P_1(x)) \text{ and } \forall\ x\ (P_1(x))$$

are also formulae.

As a notational shorthand, we write

$\exists\ a, b, c\ (P(a, b, c))$ for $\exists\ a\ (\exists\ b\ (\exists\ c\ (P(a, b, c))))$.

Example Queries

We now give domain-relational-calculus queries for the examples that we considered earlier. Note the similarity of these expressions and the corresponding tuple-relational-calculus expressions.

Find the instructor *ID*, *name*, *dept name*, and *salary* for instructors whose salary is greater than $80,000:

$$\{< i, n, d, s > \mid < i, n, d, s > \in \quad instructor \wedge s > 80000\}$$

Find all instructor *ID* for instructors whose salary is greater than $80,000:

$$\{< n > \mid \exists\, i, d, s\, (< i, n, d, s > \in \quad instructor \wedge s > 80000)\}$$

Although the second query appears similar to the one that we wrote for the tuple relational calculus, there is an important difference. In the tuple calculus, when we write $\exists\ s$ for some tuple variable *s*, we bind it immediately to a relation by writing $\exists\ s \in r$. However, when we write $\exists\ n$ in the domain calculus, *n* refers not to a tuple, but rather to a domain value. Thus, the domain of variable *n* is unconstrained until the subformula $< i, n, d, s > \in instructor$ constrains *n* to instructor names that appear in the *instructor* relation.

We now give several examples of queries in the domain relational calculus.

Find the names of all instructors in the Physics department together with the *course id* of all courses they teach:

$$\{< n, c > \mid \exists\ \ i, a\, (< i, c, a\ , s, y > \in\ teaches$$
$$\exists\, d, s\, (< i, n, d, s > \in\ instructor \wedge\ d = \text{“Physics”}))\}$$

Find the set of all courses taught in the Fall 2009 semester, the Spring 2010 semester, or both:

$$\{< c > \mid \exists\ s\, (< c, a\ , s, y, b, r, t > \in section$$
$$s = \text{“Fall”} \wedge y = \text{“2009”}$$
$$\exists\ u\, (< c, a\ , s, y, b, r, t > \in section$$
$$s = \text{“Spring”} \wedge y = \text{“2010”}$$

Find all students who have taken all courses offered in the Biology depart-ment:

$$\{< i > \mid \exists\ n, d, t\, (< i, n, d, t > \in\ student\ ) \wedge$$
$$x, y, z, w\, (< x, y, z, w > \in\ course \wedge\ z = \text{“Biology”} \Rightarrow$$
$$\exists\ a\ , b\, (< a\ , x, b, r, p, q > \in\ takes \wedge\ < c, a > \in\ depositor\ ))\}$$

Note that as was the case for tuple-relational-calculus, if no courses are offered in the Biology department, all students would be in the result.

Safety of Expressions

We noted that, in the tuple relational calculus (Section 6.2), it is possible to write expressions that may generate an infinite relation. That led us to define *safety* for tuple-relational-calculus expressions. A similar situation arises for the domain relational calculus. An expression such as

$$\{< i, n, d, s > \mid \neg(< i, n, d, s > \in \ instructor\ )\}$$

is unsafe, because it allows values in the result that are not in the domain of the expression.

For the domain relational calculus, we must be concerned also about the form of formulae within "there exists" and "for all" clauses. Consider the expression

$$\{< x > \mid \exists\ y\ (< x, y > \in\ r\ ) \wedge\ \exists\ z\ (\neg(< x, z > \in\ r\ ) \wedge\quad P(x, z))\}$$

where P is some formula involving x and z. We can test the first part of the formula, $\exists\ y\ (< x, y > \in r\ )$, by considering only the values in r. However, to test the second part of the formula, $\exists\ z\ (\neg\ (< x, z > \in r\ ) \wedge P(x, z))$, we must consider values for z that do not appear in r. Since all relations are finite, an infinite number of values do not appear in r. Thus, it is not possible, in general, to test the second part of the formula without considering an infinite number of potential values for z. Instead, we add restrictions to prohibit expressions such as the preceding one.

In the tuple relational calculus, we restricted any existentially quantified vari-able to range over a specific relation. Since we did not do so in the domain calculus, we add rules to the definition of safety to deal with cases like our example. We say that an expression

$$\{< x_1,\ x_2, \ldots , x_n > \mid P\ (x_1,\ x_2, \ldots , x_n)\}$$

is safe if all of the following hold:

All values that appear in tuples of the expression are values from *dom*(P).

For every "there exists" subformula of the form $\exists\ x\ (P_1(x))$, the subformula is true if and only if there is a value x in *dom*(P_1) such that $P_1(x)$ is true.

For every "for all" subformula of the form $\forall x\ (P_1(x))$, the subformula is true if and only if $P_1(x)$ is true for all values x from *dom*(P_1).

The purpose of the additional rules is to ensure that we can test "for all" and "there exists" subformulae without having to test infinitely many possibilities. Consider the second rule in the definition of safety. For $\exists\ x\ (P_1(x))$ to be true,

we need to find only one x for which $P_1(x)$ is true. In general, there would be infinitely many values to test. However, if the expression is safe, we know that we can restrict our attention to values from *dom*(P_1). This restriction reduces to a finite number the tuples we must consider.

The situation for subformulae of the form $\forall x\ (P_1(x))$ is similar. To assert that $\forall x\ (P_1(x))$ is true, we must, in general, test all possible values, so we must examine infinitely many values. As before, if we know that the expression is safe, it is sufficient for us to test $P_1(x)$ for those values taken from $dom(P_1)$.

All the domain-relational-calculus expressions that we have written in the example queries of this section are safe, except for the example unsafe query we saw earlier.

Expressive Power of Languages

When the domain relational calculus is restricted to safe expressions, it is equivalent in expressive power to the tuple relational calculus restricted to safe expressions. Since we noted earlier that the restricted tuple relational calculus is equivalent to the relational algebra, all three of the following are equivalent:

The basic relational algebra (without the extended relational-algebra operations)
The tuple relational calculus restricted to safe expressions
The domain relational calculus restricted to safe expressions

We note that the domain relational calculus also does not have any equivalent of the aggregate operation, but it can be extended to support aggregation, and extending it to handle arithmetic expressions is straightforward.

Overview of the Design Process

The task of creating a database application is a complex one, involving

- design of the database schema
- design of the programs that access and update the data and
- Design of a security scheme to control access to data.

The needs of the users play a central role in the design process. The design of a complete database application environment that meets the needs of the enterprise being modeled requires attention to a broad set of issues. These additional aspects of the expected use of the database influence a variety of design choices at the physical, logical, and view levels.

Design Phases

For small applications, it may be feasible for a database designer who understands the application requirements to decide directly on the relations to be created, their attributes, and constraints on the relations.

However, such a direct design process is difficult for real-world applications, since they are often highly complex. Often no one person understands the complete data needs of an application.

The database designer must interact with users of the application to understand the needs of the application, represent them in a high-level fashion that can be understood by the users, and then translate the requirements into lower levels of the design. A high-level data model serves the database designer by providing a conceptual framework in which to specify, in a systematic fashion, the data requirements of the database users, and a database structure that fulfills these requirements.

- The initial phase of database design is to characterize fully the data needs of the prospective database users. The database designer needs to interact ex-tensively with domain experts and users to carry out this task. The outcome of this phase is a specification of user requirements.
- Next, the designer chooses a data model and, by applying the concepts of the chosen data model, translates these requirements into a conceptual schema of the database. The schema developed at this **conceptual-design** phase pro-vides a detailed overview of the enterprise. Typically, the conceptual-design phase re-sults in the creation of an entity-relationship diagram that provides a graphic representation of the schema.
- A fully developed conceptual schema also indicates the functional require-ments of the enterprise. In a **specification of functional requirements**, users describe the kinds of operations (or transactions) that will be performed on the data. Example operations include modifying or updating data, searching for and retrieving specific data, and deleting data. At this stage of conceptual design, the designer can review the schema to ensure it meets functional requirements.

The process of moving from an abstract data model to the implementation of the database proceeds in two final design phases.

- In the **logical-design phase**, the designer maps the high-level conceptual schema onto the implementation data model of the database system that will be used. The implementation data model is typically the relational data model, and this step typically consists of mapping the conceptual schema defined using the entity-relationship model into a relation schema.

- Finally, the designer uses the resulting system-specific database schema in the subsequent **physical-design phase**, in which the physical features of the database are specified.

Design Alternatives

A major part of the database design process is deciding how to represent in the design the various types of “things” such as people, places, products, and the like. We use the term *entity* to refer to any such distinctly identifiable item.

In a university database, examples of entities would include instructors, students, departments, courses, and course offerings. The various entities are related to each other in a variety of ways, all of which need to be captured in the database design.

For example, a student takes a course offering, while an instructor teaches a course offering; teaches and takes are examples of relationships between entities.

In designing a database schema, we must ensure that we avoid two major pitfalls:

Redundancy: A bad design may repeat information. For example, if we store the course identifier and title of a course with each course offering, the title would be stored redundantly (that is, multiple times, unnecessarily) with each course offering. It would suffice to store only the course identifier with each course offering, and to associate the title with the course identifier only once, in a course entity.

Incompleteness: A bad design may make certain aspects of the enterprise difficult or impossible to model. For example, suppose that, as in case (1) above, we only had entities corresponding to course offering, without having an entity corresponding to courses. Equivalently, in terms of relations, suppose we have a single relation where we repeat all of the course information once for each section that the course is offered. It would then be impossible to represent information about a new course, unless that course is offered. We might try to make do with the problematic design by storing null values for the section information. Such a work-around is not only unattractive, but may be prevented by primary-key constraints.

Avoiding bad designs is not enough. There may be a large number of good designs from which we must choose. As a simple example, consider a customer who buys a product. Is the sale of this product a relationship between the customer and the product? Alternatively, is the sale itself an entity that is related both to the customer and to the product? This choice, though simple, may make an important difference in what aspects of the enterprise can be modeled well.

The Entity-Relationship Model

Introduction

The **Entity – Relationship (E - R)** data model was developed to facilitate database design by allowing specification of an enterprise schema that represents the overall logical by structure of a data base. The E-R model lies in its representation of the several semantic data models; the semantic aspect of the model lies in its pre presentation of the meaning of the data. The E-R model is very useful in mapping the meanings and interactions of real – world enterprises into a conceptual schema. Because of this usefulness, many database design tools draw on concepts from the E – R Model. The E – R data model employs three basic notions: **Entity, Relationship sets and Attributes.**

Entity

An **Entity** is an "object" that exists and is distinguishable from other objects. An Entity is represented by a set of attributes. These attributes are the descriptive properties possessed by each entity. For example, Roll Number of a Student, is an entity, it is uniquely identifies a person in a class. i.e the Roll Number of a student is distinguishable from one other .

Entity Sets

The **Entity Set** is a set of entities of the same type, that share the same properties or Attributes. **Ex :** Persons having an account at bank. or Each student having Admission Number.

The set of all students in a class., can be defined as the entity-set. Who are students of an university, for example, can be defined as the entity set student. Similarly, the entity set **Admission** might represent the set of all admission awarded by a particular university. The individual entities that constitute a set are said to be the **extension** of the entity set. Thus, all the individual students are the extension of the entity set **student.**

Example: Student Table

Student Name	**Address**	**Admission No**
Salman	Hyderabad	27096-12- 101
Muneer	Sec'bad	27096-12- 102
Imran	Charminar	27096-12- 103
Ravi	Banglore	27096-12- 104
Raja	Delhi	27096-12- 105

Bank Account

Account No	Name of the Bank	Branch	Balance
A – 101	HDFC	Malakpet	500
A – 215	SBI	Saidabad	800
A – 305	SBH	YMCA	400
A – 201	CANARABANK	Narayanguda	900
A - 405	HDFC	Malakpet	750

Attributes

Each entity has certain characteristics knows as attributes. For instance the student entity might include the following attributes, Student name, Roll Number etc. For each attribute, there is a set of permitted values, called the **domain,** or **value set**, of that attribute. An attribute of an entity set is a function that maps from the entity set into a domain. Since an entity set may have several attributes, each entity can be described by a set of (attribute, data value) pairs, one pair for each attribute of the entity set.

The attributes can be classified in to

1. Simple attributes
2. Complex/ composite attributes
3. Single – valued attributes
4. Multi - valued attributes
5. Derived attribute
6. Null Attribute

An attribute, as used in the E – R model, can be characterized by the following attribute types.

• **Simple attributes :** The attributes have been simple; that is, they have not been divided into subparts. Example : Student class Roll Number.

• **Composite attributes :** The attributes, which can be sub divided in to sub parts.

Example : Student Name, Which can be divided in parts like First name, Middle name and Last name. Note also that a composite attribute may appear as a hierarchy. In the composite attribute address, its component attribute street can be further divided into street_number, street_name, and Door _ number etc.

• **Single valued attributes :** The attribute which contain/ accept only one value/character.

Example: **Sex :** Male or Female
Marital status : Married or Unmarried

• **Multivalued attributes :** The attributes which has set of values for a specific entity. in our example all have a single value for a particular entity.

Example 1 : Number of dependents in a family may 0,1,2,3,4…..

Example 2 : A student may have several phone numbers, and different students may have different numbers of phones.

• **Derived attribute :** The value of this type of attribute can be derived from the values of other related attributes or entities. The value of a derived attribute is not stored but is computed when required.

• **Null Attributes :** An attribute takes a **null** value when an entity does not have a value for it. The null value may indicate "not applicable" – that is, that the value does not exist for the entity.

Relationship Sets

A **Relationship** is an association among several entities. For example, we can define a relationship between student entity and bank account entity. The student named Rahul having bank account in HDFC ,Malakpet branch with an account number A-101.

The Relationship also can be define as

A **Relationship set** is a set of relationships of the same type. Formally, it is a mathematical relation on n e" 2 (possibly non distinct) entity sets. If E1 ,E2, ……., E_n are entity sets, then a relationship set R is a subset of

{ (e_1, e_2, …………,e_n) le_1 º E_1, e_2 º E_2, ……….. e_n º E_n } Where (e_1, e_2, …….,e_n) is a relationship.

The association between entity sets is referred to as participation; that is, the entity sets

E_1, E_2, …….., E_n **participate** in relationship set R.

A **relationship instance** in an E – R schema represents an association between the named entities in the real – world enterprise that is being modeled.

The function that an entity plays in a relationship is called that entity's **role**. Since entity sets participating in a relationship set are generally distinct, roles are implicit and are not usually specified. However, they are useful when the meaning of a relationship needs clarification. Such is the case when the entity sets of a relationship set are not distinct; the same entity set participates in a relationship

A relationship may also have attributes called **descriptive attributes.** A relationship instance in a given relationship set must be uniquely identifiable from its participating entities, without using the descriptive attributes.

Mapping Constraints

An E – R enterprise schema may define certain constraints to which the contents of a database must conform. In this section, we examine mapping cardinalities, key constraints, and participation constraints.

Cardinalities : Mapping Cardinalities**,** or cardinality ratios, express the number of entities to which another entity can be associated via a relationship set.

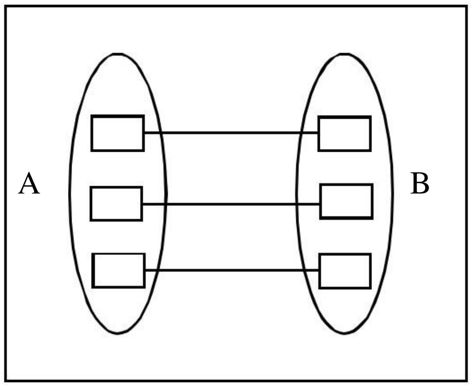
Mapping Cardinalities are most useful in describing binary relationship sets, although they can contribute to the description of relationship sets that involve more than two entity sets. In this section, we shall concentrate on only binary relationship sets.

For a binary relationship set R between entity sets A and B, the mapping cardinality must be one of the following.

There are 4 types of mapping cardinalities.

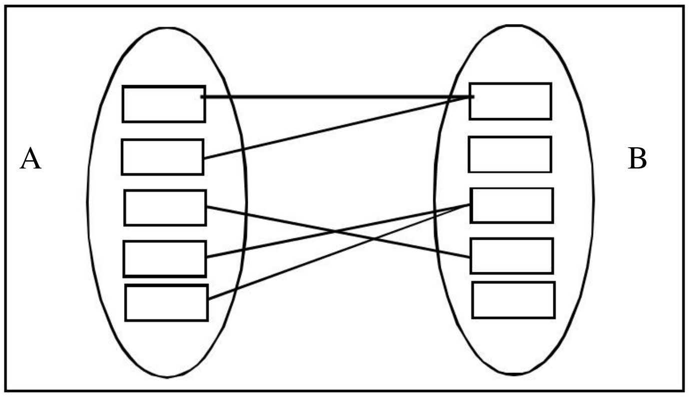
1. ONE – to – ONE Relationship
2. MANY – to – MANY Relationship
3. ONE – to – MANY Relationship
4. MANY – to – MANY Relationship

1. ONE – to – ONE Relationship : An entity in A is associated with at most one entity in B is also associated with at most one entity in A.



Example : Relationship between the entities principal and college. i.e., Principals can lead a single college and a principal can have only one college

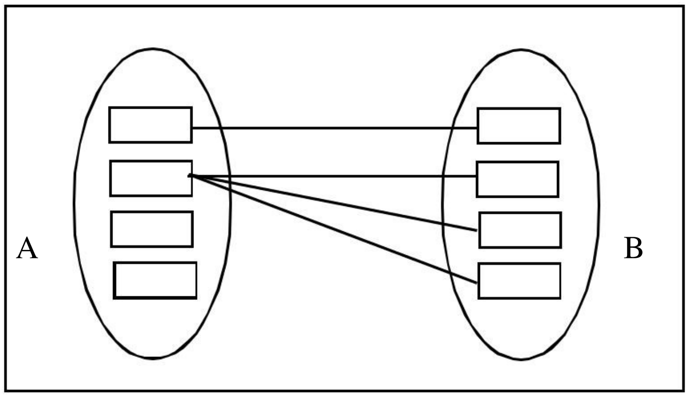
2. Many – to – One Relationship : An entity set in A is associated with at most one entity in B, An entity in B however can be associated with any number of entities in A.



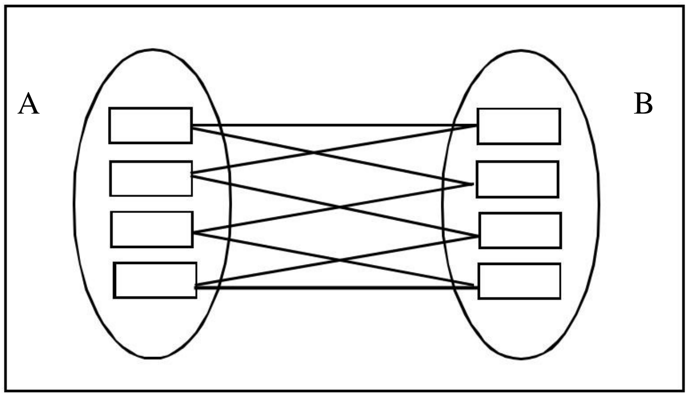
Example : Relationship between the entities Districts and state .i.e. many districts belong to a single state but many states cannot belong to single district.

3. ONE – to - MANY Relationship : An entity set A is associated with any number of entities in B. An entity in B, however can be associated with at most one entity in A.

Example : Relationship between the entities class and student i.e., a class can have many students but a student cannot be in more than one class at a time**.**



4. MANY – to – MANY Relationship : An entity set in A is associated with any number of entities in B and an entity set in B is associated with any number of entities in A.



Example : Relationship between the Entities College and course .i.e. a college can have many courses and course can be offered by many colleges

The appropriate mapping cardinality for a particular relationship set obviously depends on the real – world situation that the relationship set is modeling.

Participation Constraints

The participation of an entity set *E* in a relationship set *R* is said to be **total** if every entity in *E* participates in at least one relationship in *R*. If only some entities in *E* participate in relationships in *R*, the participation of entity set *E* in relationship *R* is said to be **partial**. In the participation of *B* in the relationship set is total while the participation of *A* in the relationship set is partial. In the participation of both *A* and *B* in the relationship set are total.

For example, we expect every *student* entity to be related to at least one instructor through the *advisor* relationship. Therefore the participation of *student* in the relationship set *advisor* is total. In contrast, an *instructor* need not advise any students. Hence, it is possible that only some of the *instructor* entities are related to the *student* entity set through the *advisor* relationship, and the participation of *instructor* in the *advisor* relationship set is therefore partial.

Removing Redundant Attributes in Entity Sets

When we design a database using the E-R model, we usually start by identifying those entity sets that should be included. For example, in the university organization we have discussed thus far, we decided to include such entity sets as *student*, *instructor*, etc. Once the entity sets are decided upon, we must choose the appropriate attributes. These attributes are supposed to represent the various values we want to capture in the database. In the university organization, we decided that for the *instructor* entity set, we will include the attributes *ID*, *name*, *dept name*, and *salary*. We could have added the attributes: *phone number*, *office number*, *home page*, etc. The choice of what attributes to include is up to the designer, who has a good understanding of the structure of the enterprise.

Once the entities and their corresponding attributes are chosen, the relation-ship sets among the various entities are formed. These relationship sets may result in a situation where attributes in the various entity sets are redundant and need to be removed from the original entity sets. To illustrate, consider the entity sets *instructor* and *department*:

The entity set *instructor* includes the attributes *ID*, *name*, *dept name*, and *salary*, with *ID* forming the primary key.

The entity set *department* includes the attributes *dept name*, *building*, and *bud-get*, with *dept name* forming the primary key.

We model the fact that each instructor has an associated department using a relationship set *inst dept* relating *instructor* and *department*.

The attribute *dept name* appears in both entity sets. Since it is the primary key for the entity set *department*, it is redundant in the entity set *instructor* and needs to be removed.

Removing the attribute *dept name* from the *instructor* entity set may appear rather unintuitive, since the relation *instructor* that we used in the earlier chap-ters had an attribute *dept name*. As we shall see later, when we create a relational schema from the E-R diagram, the attribute *dept name* in fact gets added to the relation *instructor*, but only if each instructor has at most one associated depart-ment. If an instructor has more than one associated department, the relationship between instructors and departments is recorded in a separate relation *inst dept*.

Treating the connection between instructors and departments uniformly as a relationship, rather than as an attribute of *instructor*, makes the logical relationship explicit, and helps avoid a premature assumption that each instructor is associated with only one department.

Similarly, the *student* entity set is related to the *department* entity set through the relationship set *student dept* and thus there is no need for a *dept name* attribute in *student*.

As another example, consider course offerings (sections) along with the time slots of the offerings. Each time slot is identified by a *time slot id*, and has associated with it a set of weekly meetings, each identified by a day of the week, start time, and end time. We decide to model the set of weekly meeting times as a multivalued composite attribute. Suppose we model entity sets *section* and *time slot* as follows:

The entity set *section* includes the attributes *course id*, *sec id*, *semester*, *year*, *building*, *room number*, and *time slot id*, with (*course id*, *sec id*, *year*, *semester*) forming the primary key.

The entity set *time slot* includes the attributes *time slot id*, which is the primary key,[4] and a multivalued composite attribute {(*day, start time, end time*)}.[5]

These entities are related through the relationship set *sec time slot*.

The attribute *time slot id* appears in both entity sets. Since it is the primary key for the entity set *time slot*, it is redundant in the entity set *section* and needs to be removed.

As a final example, suppose we have an entity set *classroom*, with attributes *building*, *room number*, and *capacity*, with *building* and *room number* forming the primary key. Suppose also that we have a relationship set *sec class* that relates *section* to *classroom*. Then the attributes *{building*, *room number}* are redundant in the entity set *section*.

good entity-relationship design does not contain redundant attributes. For our university example, we list the entity sets and their attributes below, with primary keys underlined:

classroom: with attributes (*building*, *room number*, *capacity*).
department: with attributes (*dept name*, *building*, *budget*).
course: with attributes (*course id*, *title*, *credits*).
instructor: with attributes (*ID*, *name*, *salary*).
section: with attributes (*course id*, *sec id*, *semester*, *year*).

student: with attributes (<u>*ID*</u>, *name*, *tot cred*).
time slot: with attributes (<u>*time slot id*</u>, {(*day*, *start time*, *end time*) }).

The relationship sets in our design are listed below:

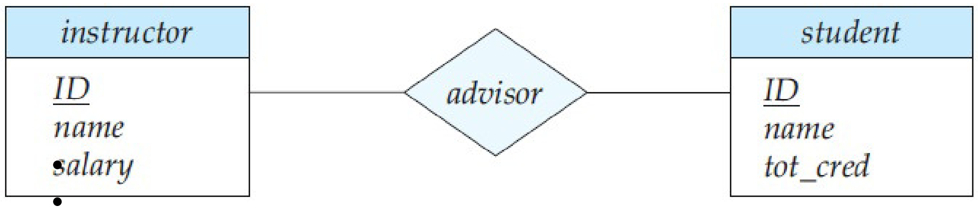
inst dept: relating instructors with departments.
stud dept: relating students with departments.
teaches: relating instructors with sections.
takes: relating students with sections, with a descriptive attribute *grade*.
course dept: relating courses with departments.
sec course: relating sections with courses.
sec class: relating sections with classrooms.
sec time slot: relating sections with time slots.
advisor: relating students with instructors.
prereq: relating courses with prerequisite courses.

You can verify that none of the entity sets has any attribute that is made redundant by one of the relationship sets. Further, you can verify that all the information (other than constraints) in the relational schema for our university database, which we saw earlier in Figure 2.8 in Chapter 2, has been captured by the above design, but with several attributes in the relational design replaced by relationships in the E-R design.

Entity – Relationship Diagrams

An **E-R diagram** can express the overall logical structure of a database graphically. E-R diagrams are simple and clear – qualities that may well account in large part for the widespread use of the E-R model. Such diagram consists of the following major components.

An E-R diagram consists of the following major components:

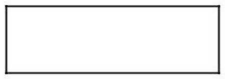
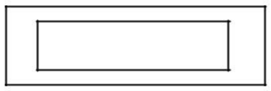
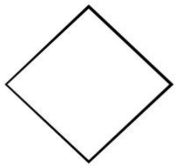
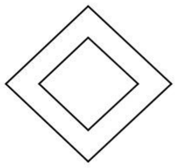
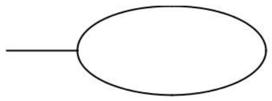


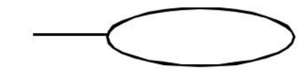
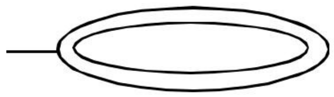
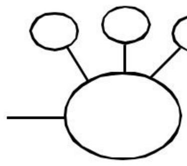
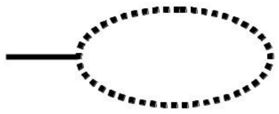
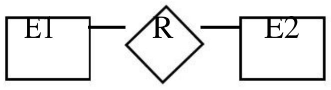
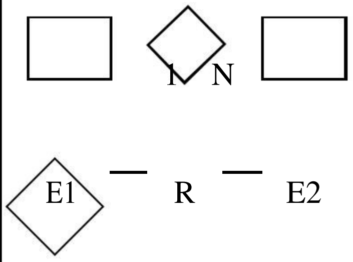
E-R diagram corresponding to instructors and students.

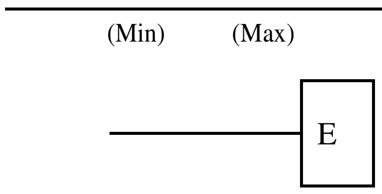
- **Rectangles :** Which represent entity sets.
- **Ellipses :** Which represent attributes
- **Diamonds :** Which represent relationship sets

- **Lines :** Which link attributes to entity sets and entity sets to relationship sets
- **Double ellipses :** Which represents multivalued attributes
- **Dashed ellipses :** Which denote derived attributes.
- **Double Lines :** Which indicate total participation of an entity in a relationship set.
- **Double Rectangles :** Which represent weak entity sets

Symbols used in E-R diagram representation

Symbol	Represented ERD Property
	**Entity**
	**Weak Entity**
	**Relation Ship**
	**Identifying Relationship**
	**Attribute**

	**Key Attribute**
	**Multivalued Attribute**
	**Composite Attribute**
	**Derived Attribute**
	**Total Participation of E 1 in R**
	**Cardinality Ratio 1: N for** **E 1, E2 in R**

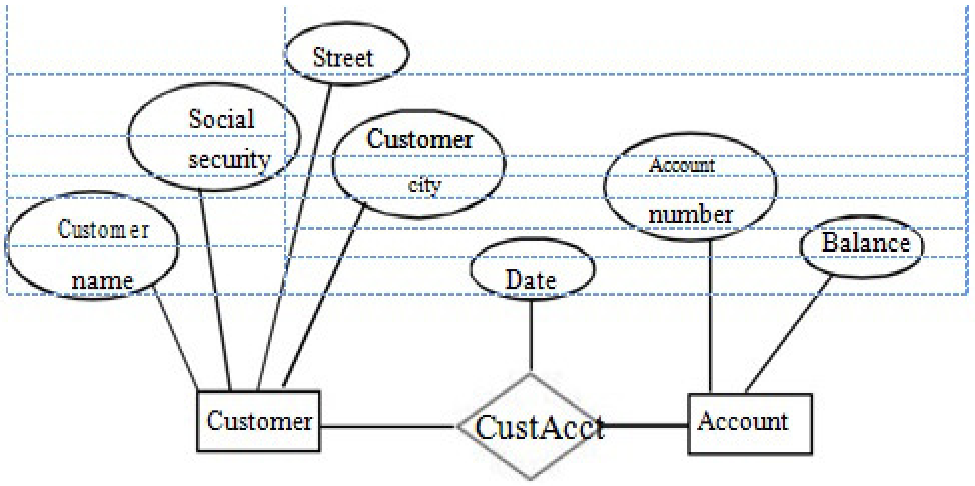


Structural constraint (Min, Max)

on Participation of E in R

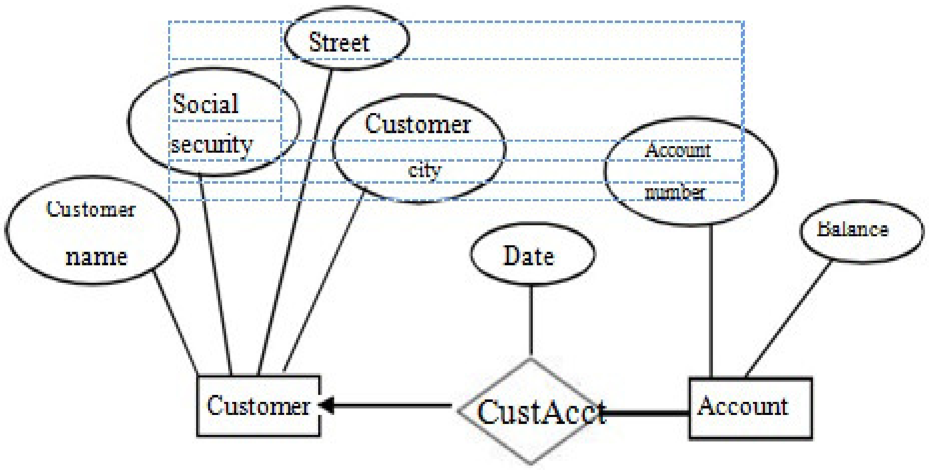
Drawing E-R diagrams

Example I - ER diagram with the entity sets Customer (Customer-name, Social-security, Street, Customer-city), Account (Account-Number, Balance with a relationship Custacct (date), i.e. date is attribute of relationship as shown below.



Example : 2

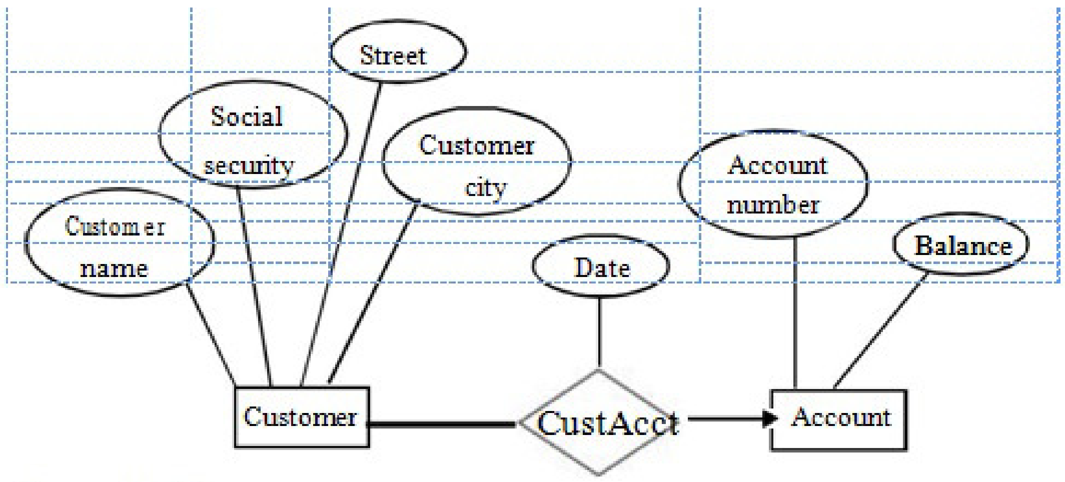
ER diagram showing the cardinality of the customer account relationship as one-to-many is as shown below.



Note : Oserve an arrow mark towards customer on the line joining customer and custacc. i.e, arrow mark indicates the cardinality one, on arrow mark indicates many.

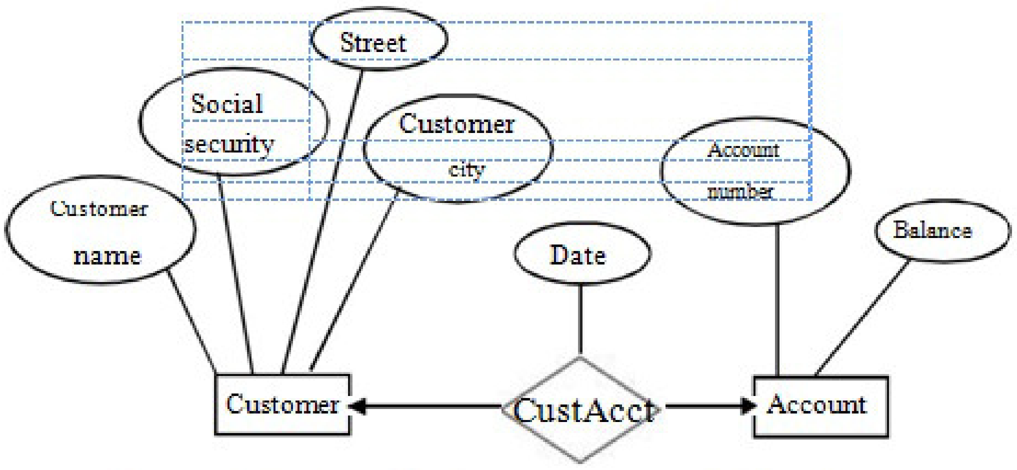
Example : 3

ER diagram showing the many - to one relationship between customer and account is as shown below. (Observe the arrow mark is towards account)



Example : 4

ER diagram showing the one -to one relationship between customer and account is as shown below.(Observe the arrow mark is towards account both customer as well as account).



Note : In the first ER diagram, the cardinality is many - to - many, because there are no arrows towards both the entity sets.

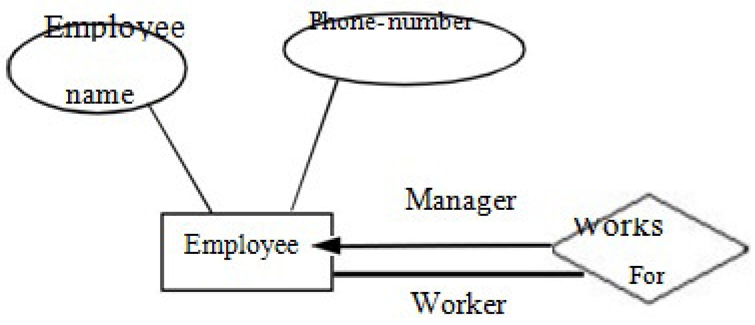
Role in E-R Diagrams

The function that an entity plays in a relationship is called its role. Roles are normally explicit and not specified.

They are useful when the meaning of a relationship set needs clarification.

For example, the entity sets of a relationship may not be distinct. The relationship works - for might be ordered pairs of employee (first is manager, second is worker).

In the E-R diagram, this can be shown by labeling the lines connecting entities (rectangles) to relationships (diamonds).



Weak Entity sets E-R Diagrams

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

Strong Entity set: An entity set that has a primary key is called a strong entity set.

For example

The entity set transaction has attributes transaction - number, date and amount. Different transaction on different accounts could share the same number. These are not sufficent to form a primary key (uniquely identify a transaction). Thus transaction is a weak entity set.

For weak entity set to be meaningful, it must be part of a one - many relationship set. This relationship set should have no descriptive attributes.

The idea of strong and weak entity sets is related to the existence dependencies seen earlier.

Member of a strong entity set a dominant entity. Member of a weak entity set is a subordinate entity.

A weak entity set does not have a primary key, but we need a means os distinguishing among the entities. The discriminator of a weak entity set is a set of attributes that allows this distinction to be made.

The primary key of a weak entity set is formed by taking the primary key of the strong entity set on which its existence depends (see Mapping Constraints) plus its discriminator.

Example : 5

Transaction is a weak entity. It is existence - dependent on account.

The primary key of account is account - number.

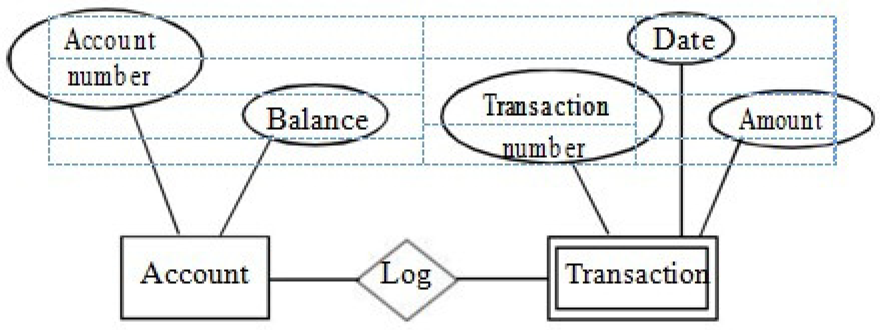
Transaction - number distinguishes transaction entities with in the same account (and is thus the discriminator).

So the primary key for transaction will be (account - number, transaction - number).

Note : The primary key of a weak entity is found by taking the primary key of the strong entity on which it is existence - dependent, plus the discriminator of the weak entity set.

A weak entity set is indicated by a doubly - outlined box. For example, the previously - mentioned weak entity set transaction is dependent on the strong entity set account via the relationship set log.

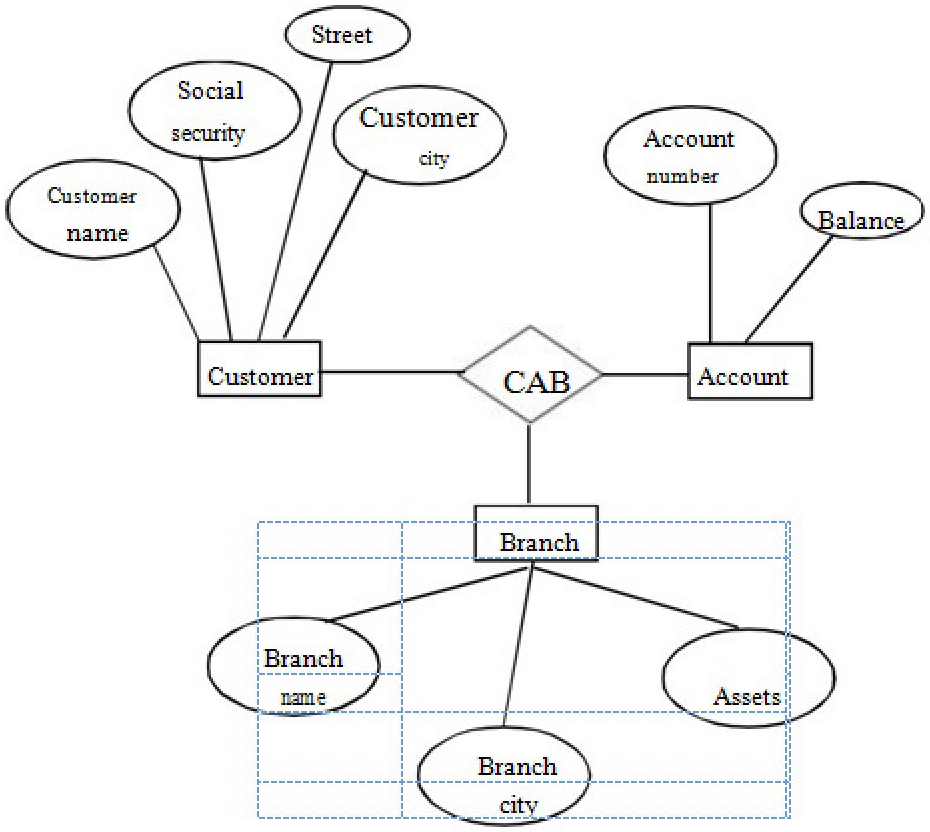
Example : 6



E-R diagram of weak entities. Observe that transaction is placed in double rectangle.

Example : 7

Non - binary ER diagram (ternary) between three entity sets a customer, Account, Branch is as shown below which says that a customer may have several accounts, each located in a specific bank branch, and that an account may belong to several different customers.



Reducing E-R Diagram into tables

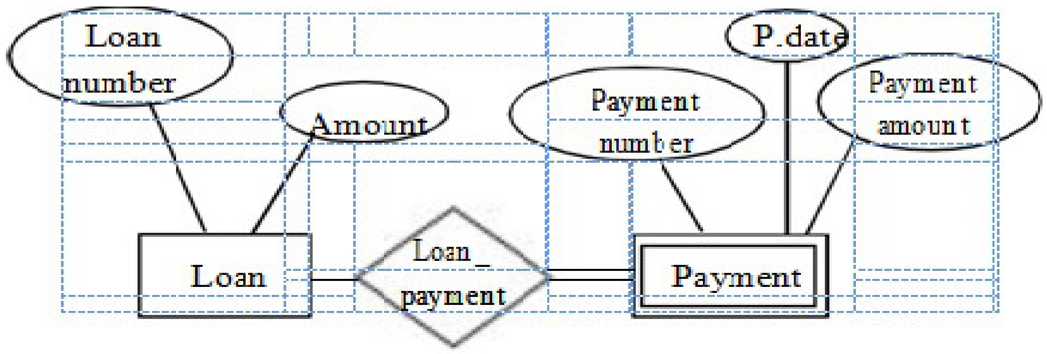
A database conforming to an E-R diagram can be represent by a collection tables. Let us see how it can be done.

- For each entity set and relationship set, there is a unique table which is assigned the name of the corresponding set.
- Each table has a number of columns with unique names.

Rules to followed to reduce ER diagrams in to tables are as given below.

- Primary keys allow entity sets and relationship sets to be expressed uniform as relation TABLES that represent the content of the database.
- Relationship TABLE that is assigned the entity set.
- A strong entity set reduces to a TABLE with the same attributes.
- A weak entity set becomes a table that includes a column for the primary key of the identifying strong entity set.

Consider ER diagram constaining weak entity set as shown below. Here payment is weak entity set.



Example

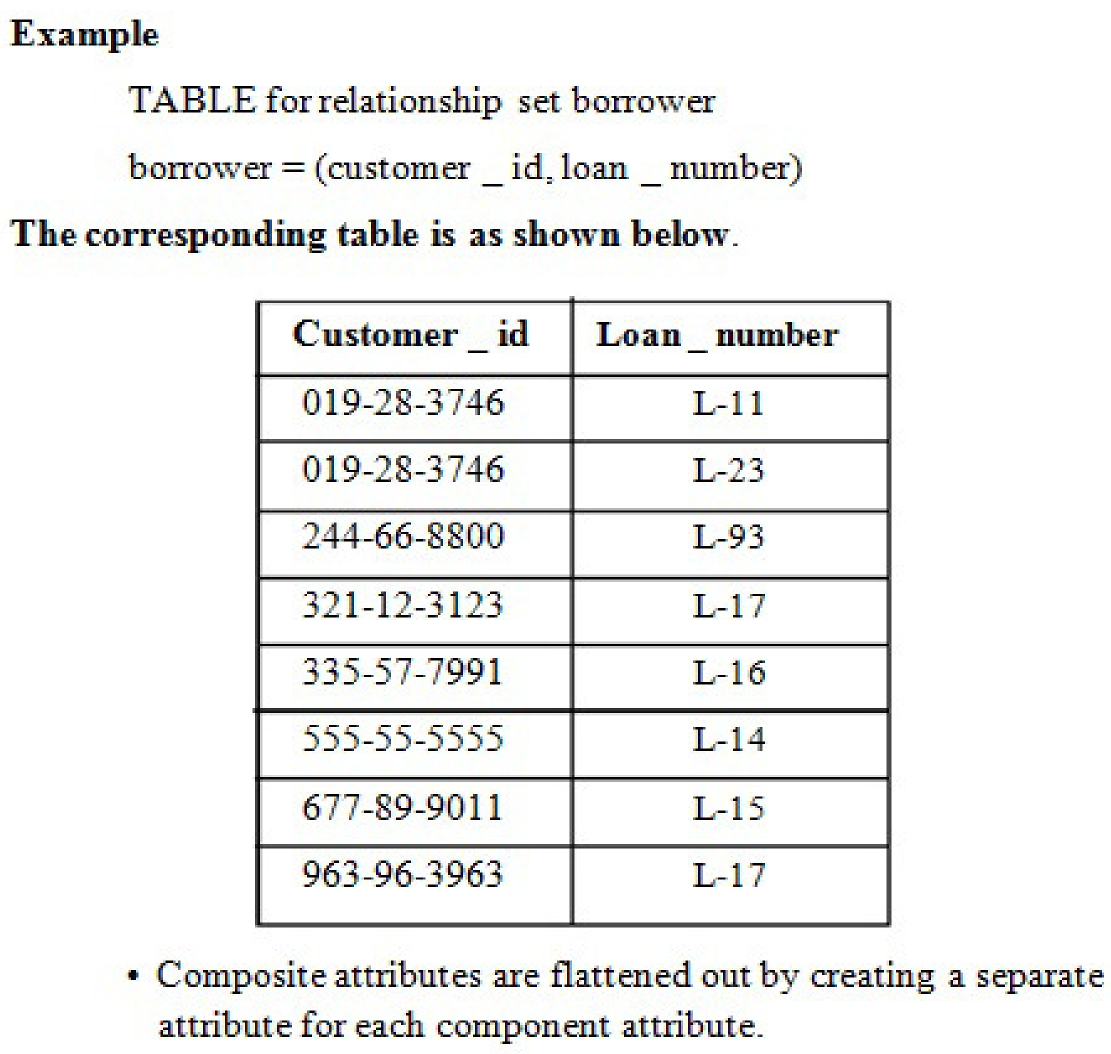
The table for payment = (loan _ number, payment _ number, payment _ date payment _ amount)

The respective table is as shown below.

Loan-number	payment-number	payment-date	payment-amount
L-11	53	7 June 2011	125
L-14	69	28 May 2011	500
L-15	22	23 May 2011	300
L-16	58	18 June 2011	135
L-17	5	10 May 2011	50
L-17	6	7 June 2011	50
L-17	7	17 June 2011	100
L-23	11	17 May 2011	75
L-93	103	3 June 2011	900
L-93	104	13 June 2011	200

- Each TABLE has a number of column (generally corresponding to attributes). which have unique names.
- A many - to - many ralationship set is represented as a TABLE with attributes for the primary keys of the two participating entity sets, and any descriptives attributes of the relationship set.

Example



Example

TABLE for relationship set borrower

borrower = (customer _ id, loan _ number)

The corresponding table is as shown below.

Customer _ id	Loan _ number
019-28-3746	L-11
019-28-3746	L-23
244-66-8800	L-93
321-12-3123	L-17
335-57-7991	L-16
555-55-5555	L-14
677-89-9011	L-15
963-96-3963	L-17

- Composite attributes are flattened out by creating a separate attribute for each component attribute.

Given entity set customer with composite attribute name with component attributes first - name and last - name the table corresponding to the entity set has two attributes.

name. first _ name and name. last _ name

- A multivalued attribute M of an entity E is represented by a separate table EM. Table EM has attributes corresponding to the primary key of E and an attribute corresponding to multivalued attribute M.

Example

Multivalued attribute dependent - names of employee is represented by a table.

employee - dependent - names (employee -id,dname)

- Each value of the multivalued attribute maps to a separate row of the table EM.

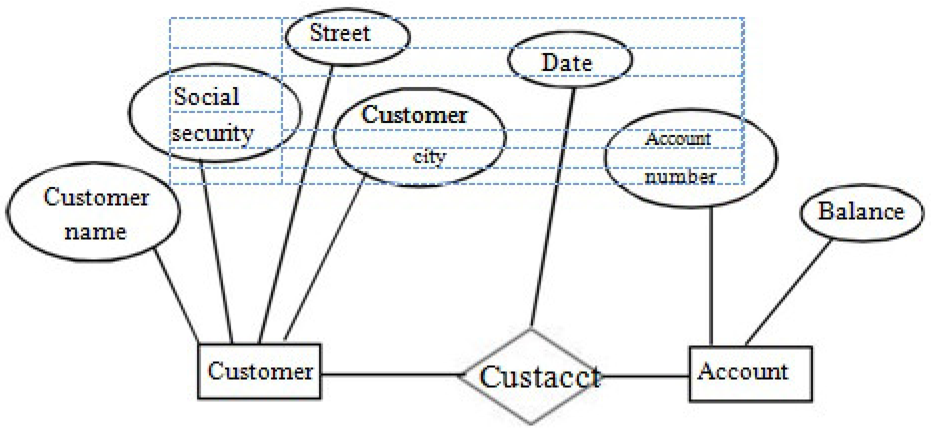
Example

An employee entity with primary key 19444 and dependents John and Maria maps to rows

(19444, John) and (19444,Maria)

Example

Reduce the following ER diagram into tables.



Each entity set customer and account will have a table and the relationship Cust_acct will have a table hence there will be three tables with the columns as given below.

Customer table with columns customer _ name, Social _ security, Street, Customer _ city.

Account table with columns account _ number, balance.

Custacct table with attributes Social _ security, account _ number, date.

Procedure for conversion of ER Diagram into a database table

1. The E – R diagram of any database can be represented by a collection of tables.
2. For each entity set and for each relationship set there is unique table to which is assigned the name of the corresponding entity set or relationship.
3. Each table has a number of columns which again have unique names i.e. attributes.
4. The values of all attributes are called records.
5. The column value which uniquely identifies the record in the table will be defined as primary key.
6. Other keys will be defined according to the relationship with other tables / entities.

Reduction to Relational Schemas

We can represent a database that conforms to an E-R database schema by a col-lection of relation schemas. For each entity set and for each relationship set in the database design, there is a unique relation schema to which we assign the name of the corresponding entity set or relationship set.

Both the E-R model and the relational database model are abstract, logical representations of real-world enterprises. Because the two models employ similar design principles, we can convert an E-R design into a relational design.

In this section, we describe how an E-R schema can be represented by relation schemas and how constraints arising from the E-R design can be mapped to constraints on relation schemas.

Representation of Strong Entity Sets with Simple Attributes

Let E be a strong entity set with only simple descriptive attributes $a_1, a_2, \ldots, a_n$. We represent this entity by a schema called E with n distinct attributes. Each tuple in a relation on this schema corresponds to one entity of the entity set E.

For schemas derived from strong entity sets, the primary key of the entity set serves as the primary key of the resulting schema. This follows directly from the fact that each tuple corresponds to a specific entity in the entity set.

As an illustration, consider the entity set *student* of the E-R diagram in Fig-ure 7.15. This entity set has three attributes: *ID*, *name*, *tot cred*. We represent this entity set by a schema called *student* with three attributes:

student (*ID*, *name*, *tot cred*)

Note that since student *ID* is the primary key of the entity set, it is also the primary key of the relation schema.

Continuing with our example, for the E-R diagram in Figure 7.15, all the strong entity sets, except *time slot*, have only simple attributes. The schemas derived from these strong entity sets are:

classroom (*building*, *room number*, *capacity*)
department (*dept name*, *building*, *budget*)
course (*course id*, *title*, *credits*)
instructor (*ID*, *name*, *salary*)
student (*ID*, *name*, *tot cred*)

As you can see, both the *instructor* and *student* schemas are different from the schemas we have used in the previous chapters (they do not contain the attribute *dept name*). We shall revisit this issue shortly.

Representation of Strong Entity Sets with Complex Attributes

When a strong entity set has nonsimple attributes, things are a bit more complex. We handle composite attributes by creating a separate attribute for each of the component attributes; we do not create a separate attribute for the composite attribute itself. To illustrate, consider the version of the *instructor* entity set depicted in Figure 7.11. For the composite attribute *name*, the schema generated for *instructor* contains the attributes *first name*, *middle name*, and *last name*; there is no separate attribute or schema for *name*. Similarly, for the composite attribute *address*, the schema generated contains the attributes *street*, *city*, *state*, and *zip code*. Since *street* is a composite attribute it is replaced by *street number*, *street name*, and *apt number*. We revisit this matter

Multivalued attributes are treated differently from other attributes. We have seen that attributes in an E-R diagram generally map directly into attributes for the appropriate relation schemas. Multivalued attributes, however, are an exception; new relation schemas are created for these attributes, as we shall see shortly.

Derived attributes are not explicitly represented in the relational data model. However, they can be represented as "methods" in other data models such as the object-relational data model,

The relational schema derived from the version of entity set *instructor* with complex attributes, without including the multivalued attribute, is thus:

instructor (*ID*, *first name*, *middle name*, *last name*, *street number*, *street name*, *apt number*, *city*, *state*, *zip code*, *date of birth*)

For a multivalued attribute *M*, we create a relation schema *R* with an attribute *A* that corresponds to *M* and attributes corresponding to the primary key of the entity set or relationship set of which *M* is an attribute.

As an illustration, consider the E-R diagram in Figure 7.11 that depicts the entity set *instructor*, which includes the multivalued attribute *phone number*. The primary key of *instructor* is *ID*. For this multivalued attribute, we create a relation schema

instructor phone (*ID*, *phone number*)

Each phone number of an instructor is represented as a unique tuple in the relation on this schema. Thus, if we had an instructor with *ID* 22222, and phone numbers 555-1234 and 555-4321, the relation *instructor phone* would have two tuples (22222, 555-1234) and (22222, 555-4321).

We create a primary key of the relation schema consisting of all attributes of the schema. In the above example, the primary key consists of both attributes of the relation *instructor phone*.

In addition, we create a foreign-key constraint on the relation schema created from the multivalued attribute, with the attribute generated from the primary key of the entity set referencing the relation generated from the entity set. In the above example, the foreign-key constraint on the *instructor phone* relation would be that attribute *ID* references the *instructor* relation.

In the case that an entity set consists of only two attributes — a single primary-key attribute *B* and a single multivalued attribute *M* — the relation schema for the entity set would contain only one attribute, namely the primary-key attribute *B*. We can drop this relation, while retaining the relation schema with the attribute *B* and attribute *A* that corresponds to *M*.

To illustrate, consider the entity set *time slot* depicted in Figure 7.15. Here, *time slot id* is the primary key of the *time slot* entity set and there is a single multivalued attribute that happens also to be composite. The entity set can be represented by just the following schema created from the multivalued composite attribute:

time slot (<u>*time slot id*</u>, *day*, <u>*start time*</u>, *end time*)

Although not represented as a constraint on the E-R diagram, we know that there cannot be two meetings of a class that start at the same time of the same day-of-the-week but end at different times; based on this constraint, *end time* has been omitted from the primary key of the *time slot* schema.

The relation created from the entity set would have only a single attribute *time slot id*; the optimization of dropping this relation has the benefit of simplifying the resultant database schema, although it has a drawback related to foreign keys.

Representation of Weak Entity Sets

Let *A* be a weak entity set with attributes $a_1, a_2, \ldots, a_m$. Let *B* be the strong entity set on which *A* depends. Let the primary key of *B* consist of attributes $b_1, b_2, \ldots, b_n$. We represent the entity set *A* by a relation schema called *A* with one attribute for each member of the set:

$$\{a_1, a_2, \ldots, a_m\} \cup \{b_1, b_2, \ldots, b_n\}$$

For schemas derived from a weak entity set, the combination of the pri-mary key of the strong entity set and the discriminator of the weak entity set serves as the primary key of the schema. In addition to creating a primary key, we also create a foreign-key constraint on the relation *A*, specifying that the attributes $b_1, b_2, \ldots, b_n$ reference the primary key of the relation *B*. The foreign-key constraint ensures that for each tuple representing a weak entity, there is a corresponding tuple representing the corresponding strong entity.

As an illustration, consider the weak entity set *section* in the E-R diagram of Figure 7.15. This entity set has the attributes: *sec id*, *semester*, and *year*. The primary key of the *course* entity set, on which *section* depends, is *course id*. Thus, we represent *section* by a schema with the following attributes:

section (<u>*course id*</u>, <u>*sec id*</u>, <u>*semester*</u>, *year*)

The primary key consists of the primary key of the entity set *course*, along with the discriminator of *section*, which is *sec id*, *semester*, and *year*. We also create a foreign-key constraint on the *section* schema, with the attribute *course id* refer-encing the primary key of the *course* schema, and the integrity constraint "on delete cascade".[7] Because of the "on delete cascade" specification on the foreign key constraint, if a *course* entity is deleted, then so are all the associated *section* entities.

Representation of Relationship Sets

Let *R* be a relationship set, let $a_1, a_2, \ldots, a_m$ be the set of attributes formed by the union of the primary keys of each of the entity sets participating in *R*, and let the descriptive attributes (if any) of *R* be $b_1, b_2, \ldots, b_n$. We represent this relationship set by a relation schema called *R* with one attribute for each member of the set:

$$\{a_1, a_2, \ldots, a_m\} \cup \{b_1, b_2, \ldots, b_n\}$$

We described earlier, in Section 7.3.3, how to choose a primary key for a binary relationship set. As we saw in that section, taking all the primary-key attributes from all the related entity sets serves to identify a particular tuple, but for one-to-one, many-to-one, and one-to-many relationship sets, this turns out to be a larger set of attributes than we need in the primary key. The primary key is instead chosen as follows:

For a binary many-to-many relationship, the union of the primary-key at-tributes from the participating entity sets becomes the primary key.

For a binary one-to-one relationship set, the primary key of either entity set can be chosen as the primary key. The choice can be made arbitrarily.

For a binary many-to-one or one-to-many relationship set, the primary key of the entity set on the "many" side of the relationship set serves as the primary key.

For an *n*-ary relationship set without any arrows on its edges, the union of the primary key-attributes from the participating entity sets becomes the primary key.

For an *n*-ary relationship set with an arrow on one of its edges, the primary keys of the entity sets not on the "arrow" side of the relationship set serve as the primary key for the schema. Recall that we allowed only one arrow out of a relationship set.

We also create foreign-key constraints on the relation schema *R* as follows: For each entity set E_i related to relationship set *R*, we create a foreign-key con-straint from relation schema *R*, with the attributes of *R* that were derived from primary-key attributes of E_i referencing the primary key of the relation schema representing E_i .

As an illustration, consider the relationship set *advisor* in the E-R diagram of

Figure 7.15. This relationship set involves the following two entity sets:

instructor with the primary key *ID*.
student with the primary key *ID*.

Since the relationship set has no attributes, the *advisor* schema has two attributes, the primary keys of *instructor* and *student*. Since both attributes have the same name, we rename them *i ID* and s *ID*. Since the *advisor* relationship set is many-to-one from *student* to *instructor* the primary key for the *advisor* relation schema is *s ID*.

We also create two foreign-key constraints on the *advisor* relation, with at-tribute *i ID* referencing the primary key of *instructor* and attribute *s ID* referencing the primary key of *student*.

Continuing with our example, for the E-R diagram in Figure 7.15, the schemas derived from a relationship set are depicted in Figure 7.16.

Observe that for the case of the relationship set *prereq*, the role indicators associated with the relationship are used as attribute names, since both roles refer to the same relation *course*.

Similar to the case of *advisor*, the primary key for each of the relations *sec course*, *sec time slot*, *sec class*, *inst dept*, *stud dept* and *course dept* consists of the primary key of only one of the two related entity sets, since each of the corresponding relationships is many-to-one.

Foreign keys are not shown in Figure 7.16, but for each of the relations in the figure there are two foreign-key constraints, referencing the two relations created from the two related entity sets. Thus, for example, *sec course* has foreign keys referencing *section* and *classroom*, *teaches* has foreign keys referencing *instructor* and *section*, and *takes* has foreign keys referencing *student* and *section*.

The optimization that allowed us to create only a single relation schema from the entity set *time slot*, which had a multivalued attribute, prevents the creation of a foreign key from the relation schema *sec time slot* to the relation created from entity set *time slot*, since we dropped the relation created from the entity set *time*

teaches (*ID*, *course id*, *sec id*, *semester*, *year*)
takes (*ID*, *course id*, *sec id*, *semester*, *year*, *grade*)
prereq (*course id*, *prereq id*)
advisor (*s ID*, *i ID*)
sec course (*course id*, *sec id*, *semester*, *year*)
sec time slot (*course id*, *sec id*, *semester*, *year*, *time slot id*)
sec class (*course id*, *sec id*, *semester*, *year*, *building*, *room number*)

inst dept (*ID*, *dept name*)

stud dept (*ID*, *dept name*)

course dept (*course id*, *dept name*)

slot. We retained the relation created from the multivalued attribute, and named it *time slot*, but this relation may potentially have no tuples corresponding to a *time slot id*, or may have multiple tuples corresponding to a *time slot id*; thus, *time slot id* in *sec time slot* cannot reference this relation.

The astute reader may wonder why we have not seen the schemas *sec course*, *sec time slot*, *sec class*, *inst dept*, *stud dept*, and *course dept* in the previous chapters. The reason is that the algorithm we have presented thus far results in some schemas that can be either eliminated or combined with other schemas. We ex-plore this issue next.

Entity-Relationship Design Issues

The notions of an entity set and a relationship set are not precise, and it is possible to define a set of entities and the relationships among them in a number of different ways. In this section, we examine basic issues in the design of an E-R database schema. Section 7.10 covers the design process in further detail.

Use of Entity Sets versus Attributes

Consider the entity set *instructor* with the additional attribute *phone number* (Fig-ure 7.17a.) It can easily be argued that a phone is an entity in its own right with attributes *phone number* and *location*; the location may be the office or home where the phone is located, with mobile (cell) phones perhaps represented by the value "mobile." If we take this point of view, we do not add the attribute *phone number* to the *instructor*. Rather, we create:

A *phone* entity set with attributes *phone number* and *location*.

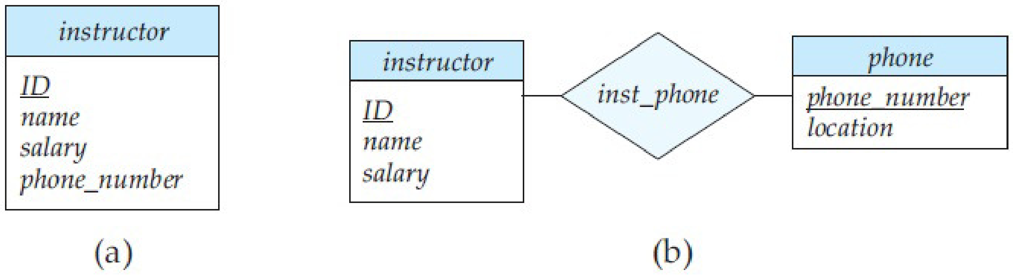
A relationship set *inst phone*, denoting the association between instructors and the phones that they have.

This alternative is shown in Figure 7.17b.

What, then, is the main difference between these two definitions of an instruc-tor? Treating a phone as an attribute *phone number* implies that instructors have precisely one phone number each. Treating a phone as an entity *phone* permits instructors to have several phone numbers (including zero) associated with them. However, we could instead easily define *phone number* as a multivalued attribute to allow multiple phones per instructor.

The main difference then is that treating a phone as an entity better models a situation where one may want to keep extra information about a phone, such as its location, or its type (mobile, IP phone, or plain old phone), or all who share

the



phone. Thus, treating phone as an entity is more general than treating it as an attribute and is appropriate when the generality may be useful.

In contrast, it would not be appropriate to treat the attribute *name* (of an instructor) as an entity; it is difficult to argue that *name* is an entity in its own right (in contrast to the phone). Thus, it is appropriate to have *name* as an attribute of the *instructor* entity set.

Two natural questions thus arise: What constitutes an attribute, and what constitutes an entity set? Unfortunately, there are no simple answers. The distinc-tions mainly depend on the structure of the real-world enterprise being modeled, and on the semantics associated with the attribute in question.

A common mistake is to use the primary key of an entity set as an attribute of another entity set, instead of using a relationship. For example, it is incorrect to model the *ID* of a *student* as an attribute of an *instructor* even if each instructor advises only one student. The relationship *advisor* is the correct way to represent the connection between students and instructors, since it makes their connection explicit, rather than implicit via an attribute.

Another related mistake that people sometimes make is to designate the primary-key attributes of the related entity sets as attributes of the relationship set. For example, *ID* (the primary-key attributes of *student*) and *ID* (the primary key of *instructor*) should not appear as attributes of the relationship *advisor*. This should not be done since the primary-key attributes are already implicit in the relationship set.[8]

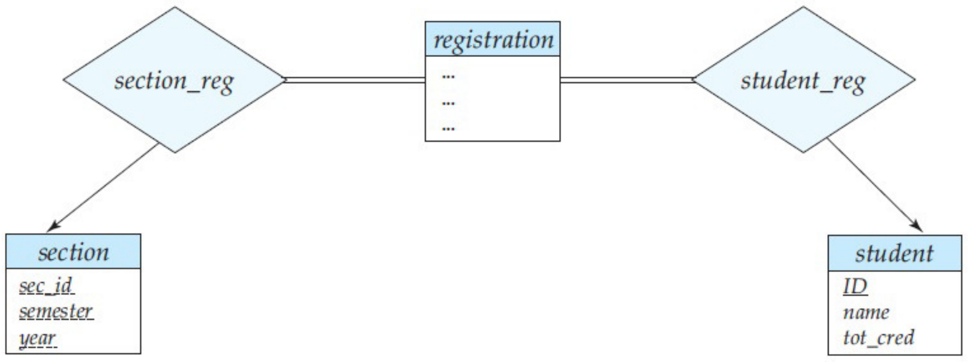
Use of Entity Sets versus Relationship Sets

It is not always clear whether an object is best expressed by an entity set or a relationship set. In Figure 7.15, we used the *takes* relationship set to model the situation where a student takes a (section of a) course. An alternative is to imagine that there is a course-registration record for each course that each student takes. Then, we have an entity set to represent the course-registration record. Let us call that entity set *registration*. Each *registration* entity is related to exactly one student and to exactly one section, so we have two relationship sets, one to relate course-registration records to students and one to relate course-registration records to sections. In Figure 7.18, we show the entity sets *section* and *student* from Figure 7.15 with the *takes* relationship set replaced by one entity set and two relationship sets:

registration, the entity set representing course-registration records.

section reg, the relationship set relating *registration* and *course*.

student reg, the relationship set relating *registration* and *student*.



Replacement of *takes* by *registration* and two relationship sets

One possible guideline in determining whether to use an entity set or a relationship set is to designate a relationship set to describe an action that occurs between entities. This approach can also be useful in deciding whether certain attributes may be more appropriately expressed as relationships.

Extended E-R Features

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

To help with the discussions, we shall use a slightly more elaborate database schema for the university. In particular, we shall model the various people within a university by defining an entity set *person*, with attributes *ID*, *name*, and *address*.

Specialization

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

As an example, the entity set *person* may be further classified as one of the following:

employee.
student.

Each of these person types is described by a set of attributes that includes all the attributes of entity set *person* plus possibly additional attributes. For exam-ple, *employee* entities may be described further by the attribute *salary*, whereas *student* entities may be described further by the attribute *tot cred*. The process of designating sub groupings within an entity set is called **specialization**. The specialization of *person* allows us to distinguish among person entities according to whether they correspond to employees or students: in general, a person could be an employee, a student, both, or neither.

As another example, suppose the university divides students into two categories: graduate and undergraduate. Graduate students have an office assigned to them. Undergraduate students are assigned to a residential college. Each of these student types is described by a set of attributes that includes all the attributes of the entity set *student* plus additional attributes.

The university could create two specializations of *student*, namely *graduate* and *undergraduate*. As we saw earlier, student entities are described by the attributes *ID*, *name*, *address*, and *tot cred*. The entity set *graduate* would have all the attributes of *student* and an additional attribute *office number*. The entity set *undergraduate* would have all the attributes of *student*, and an additional attribute *residential college*.

We can apply specialization repeatedly to refine a design. For instance, university employees may be further classified as one of the following:

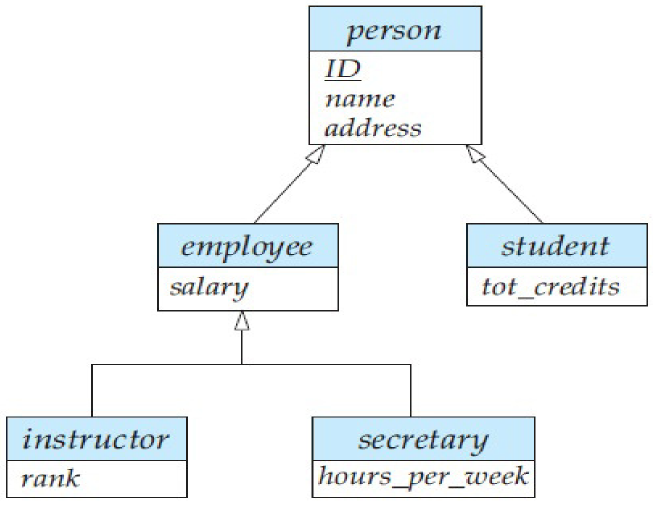
instructor.
secretary.

Each of these employee types is described by a set of attributes that includes all the attributes of entity set *employee* plus additional attributes. For example, *instructor* entities may be described further by the attribute *rank* while *secretary* entities are described by the attribute *hours per week*. Further, *secretary* entities may participate in a relationship *secretary for* between the *secretary* and *employee* entity sets, which identifies the employees who are assisted by a secretary.

An entity set may be specialized by more than one distinguishing feature. In our example, the distinguishing feature among employee entities is the job the employee performs. Another, coexistent, specialization could be based on whether the person is a temporary (limited term) employee or a permanent employee, resulting in the entity sets *temporary employee* and *permanent employee*. When more than one specialization is formed on an entity set, a particular entity may belong to multiple specializations. For instance, a given employee may be a temporary employee who is a secretary.

In terms of an E-R diagram, specialization is depicted by a hollow arrow-head pointing from the specialized entity to the other entity (see Figure 7.21). We refer to this relationship as the ISA relationship, which stands for “is a” and represents, for example, that an instructor “is a” employee.

The way we depict specialization in an E-R diagram depends on whether an entity may belong to multiple specialized entity sets or if it must belong to at most one specialized entity set. The former case (multiple sets permitted) is called **overlapping specialization**, while the latter case (at most one permitted) is called **disjoint specialization**. For an overlapping specialization (as is the case for *student* and *employee* as specializations of *person*), two separate arrows are used. For a disjoint specialization (as is the case for *instructor* and *secretary* as specializations of *employee*), a single arrow is used. The specialization relationship may also be referred to as a **superclass-subclass** relationship. Higher- and lower-level entity



Specialization and generalization.

sets are depicted as regular entity sets — that is, as rectangles containing the name of the entity set.

Generalization

The refinement from an initial entity set into successive levels of entity subgroup-ings represents a **top-down** design process in which distinctions are made explicit. The design process may also proceed in a **bottom-up** manner, in which multiple entity sets are synthesized into a higher-level entity set on the basis of common features. The database designer may have first identified:

instructor entity set with attributes *instructor id*, *instructor name*, *instructor salary*, and *rank*.
secretary entity set with attributes *secretary id*, *secretary name*, *secretary salary*, and *hours per week*.

There are similarities between the *instructor* entity set and the *secretary* entity set in the sense that they have several attributes that are conceptually the same across the two entity sets: namely, the identifier, name, and salary attributes. This commonality can be expressed by **generalization**, which is a containment relationship that exists between a *higher-level* entity set and one or more *lower-level* entity sets. In our example, *employee* is the higher-level entity set and *instructor* and *secretary* are lower-level entity sets. In this case, attributes that are conceptually the same had different names in the two lower-level entity sets. To create a generalization, the attributes must be given a common name and represented with the higher-level entity *person*. We can use the attribute names *ID*, *name*, *address*, as we saw in the example in Section 7.8.1

Higher- and lower-level entity sets also may be designated by the terms **superclass** and **subclass**, respectively. The *person* entity set is the superclass of the *employee* and *student* subclasses.

For all practical purposes, generalization is a simple inversion of specialization. We apply both processes, in combination, in the course of designing the E-R schema for an enterprise. In terms of the E-R diagram itself, we do not distinguish between specialization and generalization. New levels of entity representation are distinguished (specialization) or synthesized (generalization) as the design schema comes to express fully the database application and the user requirements of the database. Differences in the two approaches may be characterized by their starting point and overall goal.

Specialization stems from a single entity set; it emphasizes differences among entities within the set by creating distinct lower-level entity sets. These lower-level entity sets may have attributes, or may participate in relationships, that do not apply to all the entities in the higher-level entity set. Indeed, the reason a designer applies specialization is to represent such distinctive features. If *student* and *employee* have exactly the same attributes as *person* entities, and participate in exactly the same relationships as *person* entities, there would be no need to specialize the *person* entity set.

Generalization proceeds from the recognition that a number of entity sets share some common features (namely, they are described by the same attributes and participate in the same relationship sets). On the basis of their commonalities, generalization synthesizes these entity sets into a single, higher-level entity set. Generalization is used to emphasize the similarities among lower-level entity sets and to hide the differences; it also permits an economy of representation in that shared attributes are not repeated.

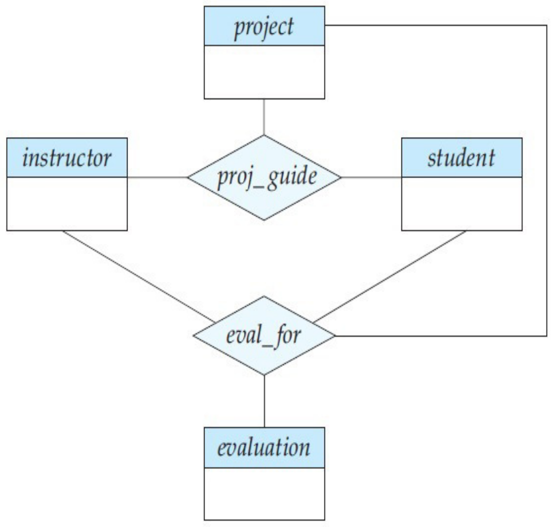
Aggregation

One limitation of the E-R model is that it cannot express relationships among relationships. To illustrate the need for such a construct, consider the ternary relationship *proj guide*, which we saw earlier, between an *instructor*, *student* and *project* (see Figure 7.13).

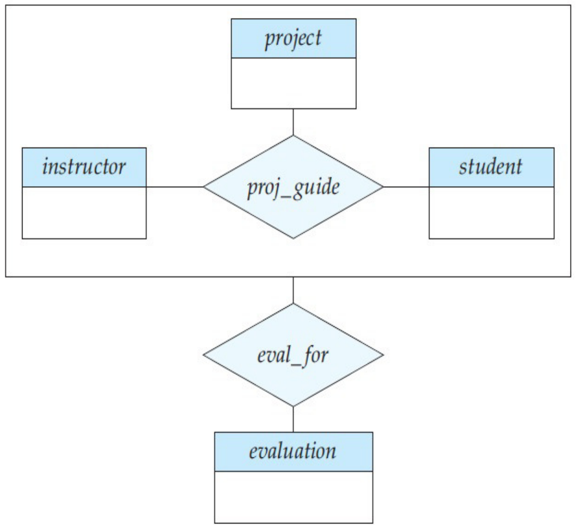
Now suppose that each instructor guiding a student on a project is required tofile a monthly evaluation report.We model the evaluation report as an entity *evaluation*,with a primary key *evaluation id*.

One alternative for recording the (*student*,*project*, *instructor*) combination to which an *evaluation* corresponds is to create a quaternary (4-way) relationship set *eval for* between *instructor*, *student*, *project*, and *evaluation*. (A quaternary relationship is required—a binary relationship between
student and *evaluation*, for example, would not permit us to represent the (*project*, *instructor*) combination to which an *evaluation* corresponds.) Using the basic E-R modeling constructs, we obtain the E-R diagram of Figure 7.22. (We have omitted the attributes of the entity sets, for simplicity.)
It appears that the relationship sets *proj guide* and *eval for* can be combined into one single relationship set. Nevertheless, we should not combine them into a single relationship, since some *instructor*, *student*, *project* combinations may not have an associated *evaluation*. There is redundant information in the resultant figure, however, since every *instructor*, *student*, *project* combination in *eval for* must also be in *proj guide*. If the *evaluation* were a value rather than a entity, we could instead make *evaluation*
a multivalued composite attribute of the relationship set *proj guide*. However, this alternative may not be be an option if an *evaluation* may also be related to other entities; for example, each evaluation report may be associated with a *secretary* who is responsible for further processing of the evaluation report to make scholarship payments. The best way to model a situation such as the one just described is to use aggregation.

Aggregation is an abstraction through which relationships are treated as higher-level entities. Thus, for our example, we regard the relationship set *proj guide* (relating the entity sets *instructor*, *student*, and *project*) as a higher-level entity set called *proj guide*. Such an entity set is treated in the same manner as is any other entity set. We can then create a binary relationship *eval for* between *proj*
guide and *evaluation* to represent which (*student*, *project*, *instructor*) combination an *evaluation* is for. Figure 7.23



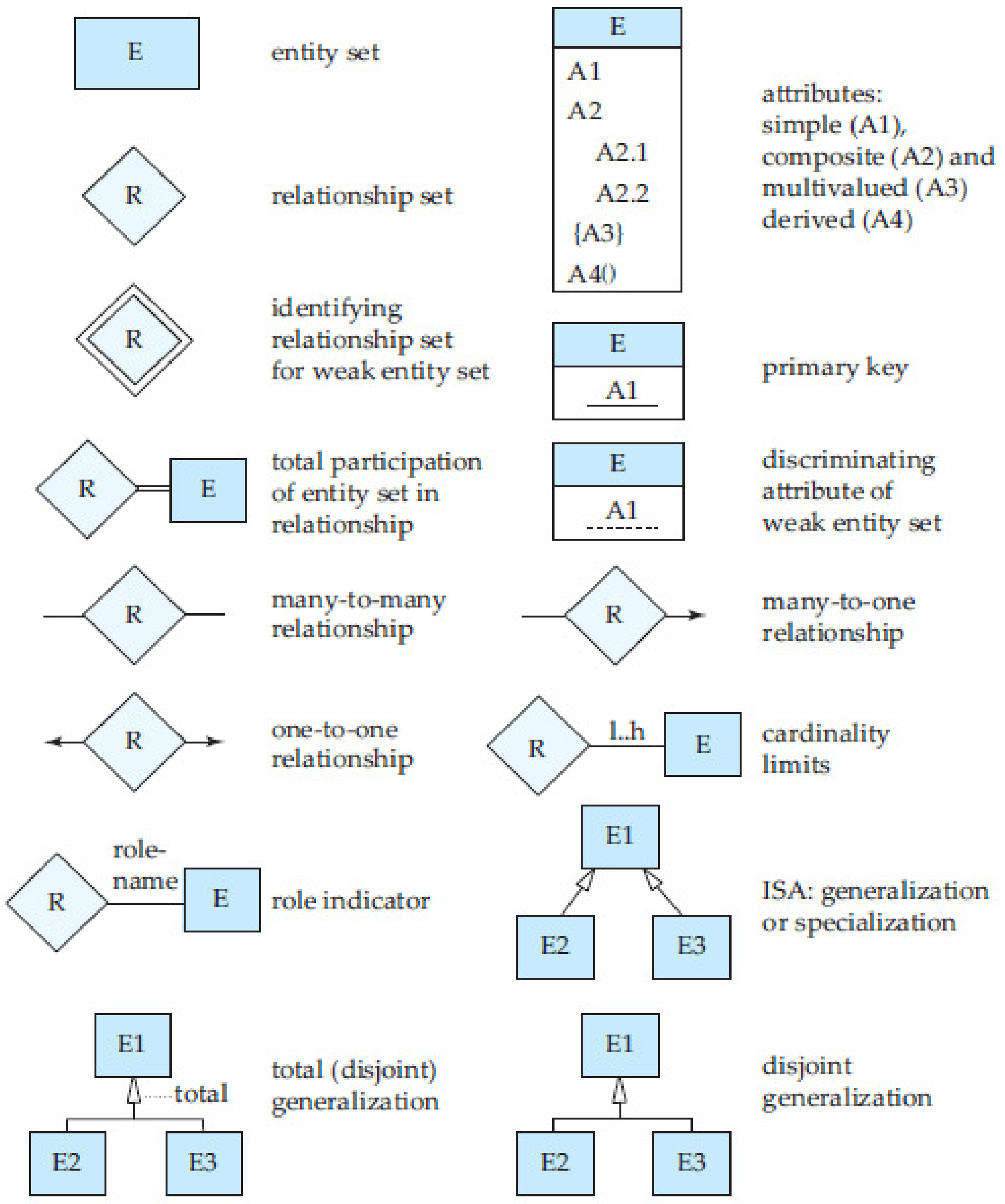
E-R diagram with redundant relationships.



Ξ-R diagram with aggregation.

Alternative Notations for Modeling Data

A diagrammatic representation of the data model of an application is a very important part of designing a database schema. Creation of a database schema requires not only data modeling experts, but also domain experts who know the requirements of the application but may not be familiar with data modeling. An intuitive diagrammatic representation is particularly important since it eases communication of information between these groups of experts. A number of alternative notations for modeling data have been proposed, of which E-R diagrams and UML class diagrams are the most widely used. There is no universal standard for E-R diagram notation, and different books and E-R diagram software use different notations. We have chosen a particular notation


E
entity set
R
relationship set
R
identifying relationship set for weak entity set
R
E
total participation of entity set in relationship
R
many-to-many relationship
R
one-to-one relationship
R
role-name
E
role indicator
E1
total
E2
E3
total (disjoint) generalization
E
A1
A2
A2.1
A2.2
{A3}
A4()
attributes: simple (A1), composite (A2) and multivalued (A3) derived (A4)
E
A1
primary key
E
A1
discriminating attribute of weak entity set
R
many-to-one relationship
R
1..h
E
cardinality limits
E1
E2
E3
ISA: generalization or specialization
E1
E2
E3
disjoint generalization

Symbols used in the E-R notation.

Other Aspects of Database Design

Our extensive discussion of schema design in this chapter may create the false impression that schema design is the only component of a database design. There are indeed several other considerations that we address more fully in subsequent chapters, and survey briefly here.

Data Constraints and Relational Database Design

We have seen a variety of data constraints that can be expressed using SQL, including primary-key constraints, foreign-key constraints, **check** constraints, assertions, and triggers. Constraints serve several purposes. The most obvious one is the automation of consistency preservation. By expressing constraints in the SQL data-definition language, the designer is able to ensure that the database system itself enforces the constraints. This is more reliable than relying on each application program individually to enforce constraints. It also provides a central location for the update of constraints and the addition of new ones.

A further advantage of stating constraints explicitly is that certain constraints are particularly useful in designing relational database schemas.

If we know, for example, that a social-security number uniquely identifies a person, then we can use a person's social-security number to link data related to that person even if these data appear in multiple relations. Contrast that with, for example, eye color, which is not a unique identifier. Eye color could not be used to link data pertaining to a specific person across relations because that person's data could not be distinguished from data pertaining to other people with the same eye color.

Data constraints are useful as well in determining the physical structure of data. It may be useful to store data that are closely related to each other in physical proximity on disk so as to gain efficiencies in disk access. Certain index structures work better when the index is on a primary key.

Constraint enforcement comes at a potentially high price in performance each time the database is updated. For each update, the system must check all of the constraints and either rejects updates that fail the constraints or execute appropriate triggers. The significance of the performance penalty depends not only on the frequency of update but also on how the database is designed

Usage Requirements: Queries, Performance

Database system performance is a critical aspect of most enterprise information systems. Performance pertains not only to the efficient use of the computing and storage hardware being used, but also to the efficiency of people who interact with the system and of processes that depend upon database data.

There are two main metrics for performance:

Throughput— the number of queries or updates (often referred to as *trans-actions*) that can be processed on average per unit of time.

Response time— the amount of time a *single* transaction takes from start to finish in either the average case or the worst case.

Systems that process large numbers of transactions in a batch style focus on having high throughput. Systems that interact with people or time-critical systems often focus on response time. These two metrics are not equivalent. High throughput arises from obtaining high utilization of system components. Doing so may result in certain transactions being delayed until such time that they can be run more efficiently. Those delayed transactions suffer poor response time.

Most commercial database systems historically have focused on throughput; however, a variety of applications including Web-based applications and telecommunication information systems require good response time on average and a reasonable bound on worst-case response time.

An understanding of types of queries that are expected to be the most frequent helps in the design process. Queries that involve joins require more resources to evaluate than those that do not. In cases where a join is required, the database administrator may choose to create an index that facilitates evaluation of that join. For queries — whether a join is involved or not — indices can be created to speed evaluation of selection predicates (SQL **where** clause) that are likely to appear. Another aspect of queries that affects the choice of indices is the relative mix of update and read operations. While an index may speed queries, it also slows updates, which are forced to do extra work to maintain the accuracy of the index.

Authorization Requirements

Authorization constraints affect design of the database as well because SQL allows access to be granted to users on the basis of components of the logical design of the database. A relation schema may need to be decomposed into two or more schemas to facilitate the granting of access rights in SQL. For example, an employee record may include data relating to payroll, job functions, and medical benefits. Because different administrative units of the enterprise may manage each of these types of data, some users will need access to payroll data while being denied access to the job data, medical data, etc. If these data are all in one relation, the desired division of access, though still feasible through the use of views, is more cumbersome.

Data Flow, Workflow

Database applications are often part of a larger enterprise application that interacts not only with the database system but also with various specialized applications.

For example, in a manufacturing company, a computer-aided design (CAD) system may assist in the design of new products. The CAD system may extract data from the database via an SQL statement, process the data internally, perhaps interacting with a product designer, and then update the database. During this process, control of the data may pass among several product designers as well as other people. As another example, consider a travel-expense report. It is created by an employee returning from a business trip (possibly by means of a special software package) and is subsequently routed to the employee's manager, perhaps other higher-level managers, and eventually to the accounting department for payment (at which point it interacts with the enterprise's accounting information systems).

The term *workflow* refers to the combination of data and tasks involved in processes like those of the preceding examples. Workflows interact with the database system as they move among users and users perform their tasks on the workflow. In addition to the data on which workflows operate, the database may store data about the workflow itself, including the tasks making up a workflow and how they are to be routed among users. Workflows thus specify a series of queries and updates to the database that may be taken into account as part of the database-design process. Put in other terms, modeling the enterprise requires us not only to understand the semantics of the data but also the business processes that use those data.

UNIT-III

Introduction to Schema Refinement:

We now present an overview of the problems that schema refinement is intended to address and a refinement approach based on decompositions. Redundant storage of information is the root cause of these problems. Although decomposition can eliminate redundancy, it can lead to problems of its own and should be used with caution.

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.

In general, the goal of relational database design is to generate a set of relation schemas that allows us to store information without unnecessary redundancy, yet also allows us to retrieve information easily. This is accomplished by designing schemas that are in an appropriate ***normal form*.**

To determine whether a relation schema is in one of the desirable normal forms, we need information about the real-world enterprise that we are modeling with the database. Some of this information exists in a well-designed E-R diagram, but additional information about the enterprise may be needed as well.

Features of Good Relational Designs

It is possible to generate a set of relation schemas directly from the E-R design. Obviously, the goodness (or badness) of the resulting set of schemas depends on how good the E-R design was in the first place However, we can go a long way toward a good design using concepts we have already studied.

Feature of Good Relational Database Design Normalization

i) In the Relational Database Design, the process of organizing data to minimizing redundancy is known as Normalization

ii) The main aim of the Normalization is to decompose complex relation into smaller, well-structured relation
iii) Normalization is the process that involves dividing a large table into smaller table(which contain less redundant data) and stating the relationship among the tables.
iv) Data normalization or Database Normalization is also canonical synthesis is mean for preventing the inconsistent in a set of data by using unique values to reference common information
v) The main objective of the normalization is to isolate the data so that user can apply the operation such as addition, deletion and modification of a field in one table and then its propagated to the rest of the database through the well defined relationships
vi) The same set of data is repeated in multiple tables of database so there are chances that data in the database may lead to be inconsistent, so while updating , deleting or inserting the data into the inconsistent database which leads to problem of data integrity
vii) If we can apply the normalization on the table we can reduce the problem of data inconsistency for some extent

Definition of Normalization:

In the Relational Database Design, the process of organizing data to minimizing redundancy is known as Normalization

Main aim of the Normalization

1. **Ensure data integrity**

i) The correct data should be stored in the database

ii) This can be achieved by applying integrity rules in the database iii) Integrity rules prevent duplicate values in the database

2. **Prevent Data Redundancy in database**

i) Non-Normalized data is more vulnerable to data anomalies. The same set of information is present in the multiple rows, now if we applying the updating rule on the table then it lead to logical inconsistence this is known as update anomaly

ADVANTAGES OF NORMALIZATION

The following are the advantages of the normalization.

- More efficient data structure.
- Avoid redundant fields or columns.
- More flexible data structure i.e. we should be able to add new rows and data values easily Better understanding of data.
- Ensures that distinct tables exist when necessary.
- Easier to maintain data structure i.e. it is easy to perform operations and complex queries can be easily handled.
- Minimizes data duplication.
- Close modeling of real world entities, processes and their relationships.

DISADVANTAGES OF NORMALIZATION

The following are disadvantages of normalization.

- Requires much more CPU, memory, and I/O to process thus normalized data gives reduced database performance.
- Requires more joins to get the desired result. A poorly-written query can bring the database down.
- Maintenance overhead. The higher the level of normalization, the greater the number of tables in the database.
- You cannot start building the database before you know what the user needs.
- On Normalizing the relations to higher normal forms i.e. 4NF, 5NF the performance degrades.
- It is very time consuming and difficult process in normalizing relations of higher degree.
- Careless decomposition may leads to bad design of database which may leads to serious problems.

How many normal forms are there?

They are

First Normal Form
Second Normal Form
Third Normal Form
Boyce-Codd Normal Form
Fourth Normal Form
Fifth Normal Form Sixth

Atomic Domains and First Normal Form

The E-R model allows entity sets and relationship sets to have attributes that have some degree of substructure. Specifically, it allows multivalued attributes such as *phone number* and composite attributes (such as an attribute *address* with component attributes *street*, *city*, *state*, and *zip*). When we create tables from E-R designs that contain these types of attributes, we eliminate this substructure. For composite attributes, we let each component be an attribute in its own right. For multivalued attributes, we create one tuple for each item in a multivalued set.

In the relational model, we formalize this idea that attributes do not have any substructure. A domain is **atomic** if elements of the domain are considered to be indivisible units. We say that a relation schema *R* is in **first normal form** (1NF) if the domains of all attributes of *R* are atomic.

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

Composite attributes, such as an attribute *address* with component attributes *street*, *city*, *state*, and *zip* also have nonatomic domains.

Integers are assumed to be atomic,
so the set of integers is an atomic domain;

however, the set of all sets of integers is a nonatomic domain.

The distinction is that we do not normally consider integers to have subparts, but we consider sets of integers to have subparts — namely, the integers making up the set. But the important issue is not what the domain itself is, but rather how we use domain elements in our database. The domain of all integers would be nonatomic if we considered each integer to be an ordered list of digits.

As a practical illustration of the above point, consider an organization that assigns employees identification numbers of the following form: The first two letters specify the department and the remaining four digits are a unique number within the department for the employee. Examples of such numbers would be "CS001" and "EE1127". Such identification numbers can be divided into smaller units, and are therefore nonatomic. If a relation schema had an attribute whose domain consists of identification numbers encoded as above, the schema would not be in first normal form.

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

From the above discussion, it may appear that our use of course identifiers such as "CS-101", where "CS" indicates the Computer Science department, means that the domain of course identifiers is not atomic. Such a domain is not atomic as far as humans using the system are concerned. However, the database application still treats the domain as atomic, as long as it does not attempt to split the identifier and interpret parts of the identifier as a department abbreviation. The *course* schema stores the department name as a separate attribute, and the database application can use this attribute value to find the department of a course, instead of interpreting particular characters of the course identifier. Thus, our university schema can be considered to be in first normal form.

The use of set-valued attributes can lead to designs with redundant storage of data, which in turn can result in inconsistencies. For instance, instead of having the relationship between instructors and sections being represented as a separate relation *teaches*, a database designer may be tempted to store a set of course section identifiers with each instructor and a set of instructor identifiers with each section. (The primary keys of *section* and *instructor* are used as identifiers.) Whenever data pertaining to which instructor teaches which section is changed, the update has to be performed at two places: in the set of instructors for the section, and the set of sections for the instructor. Failure to perform both updates can leave the database in an inconsistent state. Keeping only one of these sets, that either the set of instructors of a section, or the set of sections of an instructor, would avoid repeated information; however keeping only one of these would complicate some queries, and it is unclear which of the two to retain.

Some types of nonatomic values can be useful, although they should be used with care.

For example, composite-valued attributes are often useful, and set-valued attributes are also useful in many cases, which is why both are supported in the E-R model. In many domains where entities have a complex structure, forcing a first normal form representation represents an unnecessary burden on the application programmer, who has to write code to convert data into atomic form. There is also the runtime overhead of converting data back and forth from the atomic form. Support for nonatomic values can thus be very useful in such domains. In fact, modern database systems do support many types of nonatomic values.

Basics of Functional Dependency

Functional dependency (FD) is a property of the information represented by the relation.Functional dependency allows the database designer to express facts about the enterprise that the designer is modeling with the enterprise databases. It allows the designer to express constraints, which cannot be expressed with super keys. Functional dependency is a term derived from mathematical theory, which states that for every element in the attribute (which appears on some row), there is a unique corresponding element (on the same row).

Let us assume that rows (tuples) of a relational table T is represented by the notation r1 ,r2 ,…….., and individual attributes (columns) of the table is represented by letters A, B,…. The letters X, Y , ….., represent the subsets of attributes.

Thus, as per mathematical theory, for a given table T containing at least two attributes A and B, we can say that A -> B. The arrow notation '->' is read as "functionally determines".

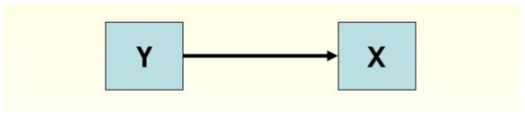
Thus, we can say that, A functionally determines B or B is functionally dependent on A. In other words, we can say that, given two rows R1,and R2, in table T, if R1(A)=R2(A) then R1(B)=R2(B).

The attributes in subset A are sometimes known as the determinant of **FD: A -> B**.

The left hand side of the functional dependency is sometimes called determinant whereas that of the right hand side is called the dependent. The determinant and dependent are both sets of attributes. A functional dependency is a many-to-one relationship between two sets of attributes X and Y of a given table T. Here X and Y are subsets of the set of attributes of table T. Thus, the functional dependency X -> Y is said to hold in relation R if and only if, whenever two tuples (rows or records) of T have the same value of X, they also have the same value for Y.

Functional Dependency Diagram and Examples

In a functional dependency diagram (FDD), functional dependency is represented by rectangles representing attributes and a heavy arrow showing dependency. Fig. shows A functional dependency diagram for the simplest functional dependency, that is, FD: Y -> X. In functional dependency diagram, each FD is displayed as a **horizontal** line.



The left-hand side attributes of the FD, i.e. determinants, are connected by **Vertical** lines to line representing the FD.
The right-hand side attributes are connected by arrows pointing towards the attibutes.

Relation schema R along with FD: Y -> X

Basic Concepts

Functional dependencies.

Functional dependencies are a constraint on the set of legal relations in a database.

They allow us to express facts about the real world we are modeling.

The notion generalizes the idea of a superkey.

Let $\alpha \subseteq R$ and $\beta \subseteq R$.

Then the functional dependency $\alpha \rightarrow \beta$ holds on *R* if in any legal relation *r*(*R*), for all pairs of tuples t_1 and t_2 in *r* such that $t_1[\alpha] = t_2[\alpha]$, it is also the case that $t_1[\beta] = t_2[\beta]$.

Using this notation, we say *K* is a superkey of *R* if $K \rightarrow R$.

In other words, *K* is a superkey of *R* if, whenever $t_1[K] = t_2[K]$, then $t_1[R] = t_2[R]$ (and thus $t_1 = t_2$).

Functional dependencies allow us to express constraints that cannot be expressed with superkeys.

Consider the scheme

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

if a loan may be made jointly to several people (e.g. husband and wife) then we would not expect *loan#* to be a superkey. That is, there is no such dependency

loan# $\rightarrow$ *cname*

We do however expect the functional dependency

loan# $\rightarrow$ *amount*

loan# $\rightarrow$ *bname*

to hold, as a loan number can only be associated with one amount and one branch.

A set *F* of functional dependencies can be used in two ways:

To specify constraints on the set of legal relations. (Does *F* hold on *R*?)

To test relations to see if they are legal under a given set of functional dependencies. (Does *r* satisfy *F*?)

Figure: shows a relation *r* that we can examine.

A	B	C	D
a_1	b_1	c_1	d_1
a_1	b_2	c_1	d_2
a_2	b_2	c_2	d_2
a_2	b_3	c_2	d_3
a_3	b_3	c_2	d_4

Figure: Sample relation *r*.

We can see that $A \rightarrow C$ is satisfied (in this particular relation), but $C \rightarrow A$ is not. $AB \rightarrow C$ is also satisfied.

Functional dependencies are called **trivial** if they are satisfied by all relations.

In general, a functional dependency $\alpha \rightarrow \beta$ is trivial if $\beta \subseteq \alpha$.

In the *customer* relation of figure 5.4, we see that $street \rightarrow ccity$ is satisfied by this relation. However, as in the real world two cities can have streets with the same names (e.g. Main, Broadway, etc.), we would not include this functional dependency in our list meant to hold on *Customer-scheme*.

The list of functional dependencies for the example database is:

On *Branch-scheme:*

bname $\rightarrow$ *bcity*

bname $\rightarrow$ *assets*

On *Customer-scheme:*

cname $\rightarrow$ *ccity*

cname $\rightarrow$ *street*

On *Loan-scheme:*

loan# $\rightarrow$ *amount*

loan# $\rightarrow$ *bname*

On *Account-scheme:*

account# $\rightarrow$ *balance*

account# $\rightarrow$ *bname*

There are no functional dependencies for *Borrower-schema*, nor for *Depositor-schema*.

Boyce–Codd Normal Form

One of the more desirable normal forms that we can obtain is **Boyce – Codd normal form** (**BCNF**). It eliminates all redundancy that can be discovered based on functional dependencies, though, as we shall see in Section 8.6, there may be other types of redundancy remaining. A relation schema *R* is in BCNF with respect to a set *F* of functional dependencies if, for all functional dependencies in F^+ of the form a → b, where a ⊆ *R* and b ⊆ *R*, at least one of the following holds:

a → b is a trivial functional dependency (that is, b ⊆ a).

a is a superkey for schema *R*.

A database design is in BCNF if each member of the set of relation schemas that constitutes the design is in BCNF.

We have already seen in Section 8.1 an example of a relational schema that is not in BCNF:

inst dept (*ID*, *name*, *salary*, *dept name*, *building*, *budget*)

The functional dependency *dept name* → *budget* holds on *inst dept*, but *dept name* is not a superkey (because, a department may have a number of different instruc-tors). In Section 8.1.2, we saw that the decomposition of *inst dept* into *instructor* and *department* is a better design. The *instructor* schema is in BCNF. All of the nontrivial functional dependencies that hold, such as:

ID → *name*, *dept name*, *salary*

include *ID* on the left side of the arrow, and *ID* is a superkey (actually, in this case, the primary key) for *instructor*. (In other words, there is no nontrivial functional dependency with any combination of *name*, *dept name*, and *salary*, without *ID*, on the side.) Thus, *instructor* is in BCNF.

Similarly, the *department* schema is in BCNF because all of the nontrivial func-tional dependencies that hold, such as:

dept name → *building*, *budget*

include *dept name* on the left side of the arrow, and *dept name* is a superkey (and the primary key) for *department*. Thus, *department* is in BCNF.

We now state a general rule for decomposing that are not in BCNF. Let *R* be a schema that is not in BCNF. Then there is at least one nontrivial functional dependency a → b such that a is not a superkey for *R*. We replace *R* in our design with two schemas:

(a ∪ b)

(*R* – (b – a))

In the case of *inst dept* above, a = *dept name*, b = *{building*, *budget}*, and *inst dept* is replaced by

(a ∪ b) = (*dept name*, *building*,*budget*)

–

(*R* – (b – a)) = (*ID*, *name*, *dept name*, *salary*)

In this example, it turns out that b – a = b. We need to state the rule as we did so as to deal correctly with functional dependencies that have attributes that appear on both sides of the arrow. The technical reasons for this are covered later in Section 8.5.1.

When we decompose a schema that is not in BCNF, it may be that one or more of the resulting schemas are not in BCNF. In such cases, further decomposition is required, the eventual result of which is a set of BCNF schemas.

Third Normal Form

BCNF requires that all nontrivial dependencies be of the form a → b, where a is a superkey. Third normal form (3NF) relaxes this constraint slightly by allowing certain nontrivial functional dependencies whose left side is not a superkey. Before we define 3NF, we recall that a candidate key is a minimal superkey — that is, a superkey no proper subset of which is also a superkey.

A relation schema *R* is in **third normal form** with respect to a set *F* of functional dependencies if, for all functional dependencies in F^+ of the form a → b, where

⊆ *R* and b ⊆ *R*, at least one of the following holds:

a → b is a trivial functional dependency.

a is a superkey for *R*.

Each attribute *A* in b – a is contained in a candidate key for *R*.

Note that the third condition above does not say that a single candidate key must contain all the attributes in b – a; each attribute *A* in b – a may be contained in a *different* candidate key.

The first two alternatives are the same as the two alternatives in the definition of BCNF. The third alternative of the 3NF definition seems rather unintuitive, and it is not obvious why it is useful. It represents, in some sense, a minimal relaxation of the BCNF conditions that helps ensure that every schema has a dependency-preserving decomposition into 3NF. Its purpose will become more clear later, when we study decomposition into 3NF.

Observe that any schema that satisfies BCNF also satisfies 3NF, since each of its functional dependencies would satisfy one of the first two alternatives. BCNF is therefore a more restrictive normal form than is 3NF.

The definition of 3NF allows certain functional dependencies that are not allowed in BCNF. A dependency a → b that satisfies only the third alternative of the 3NF definition is not allowed in BCNF, but is allowed in 3NF.[6]

Now, let us again consider the *dept advisor* relationship set, which has the following functional dependencies:

i ID → dept name
_ _

s ID, *dept name → i ID*
_ _ _

we argued that the functional dependency "*i ID → dept name*" caused the *dept advisor* schema not to be in BCNF. Note that here a = *i ID*, b = *dept name*, and b − a = *dept name*. Since the functional dependency *s ID*, *dept name →ID* holds on *dept advisor*, the attribute *dept name* is contained in a candidate key and, therefore, *dept advisor* is in 3NF.

We have seen the trade-off that must be made between BCNF and 3NF when there is no dependency-preserving BCNF design.

FUNCTIONAL DEPENDENCY THEORY

Closure of a Set of Functional Dependencies

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

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

$$A \rightarrow B$$

$$A \rightarrow C$$

$$CG \rightarrow H$$

$$CG \rightarrow I$$

$$B \rightarrow H$$

Then the functional dependency A → H is logically implied. To see why, → let t_1 and t_2 be tuples such that

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

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

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

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

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

Thus, whenever two tuples have the same value on *A*, they must also have the same value on *H*, and we can say that $A \rightarrow H$.

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

We denote the closure of *F* by F^+ .

To compute F^+ , we can use some rules of inference called **Armstrong's Axioms**:

Reflexivity rule: if α is a set of attributes and $\beta \subseteq \alpha$, then $\alpha \rightarrow \beta$ holds.

Augmentation rule: if $\alpha \rightarrow \beta$ holds, and γ is a set of attributes, then $\gamma\alpha \rightarrow \gamma\beta$ holds.

Transitivity rule: if $\alpha \rightarrow \beta$ holds, and $\beta \rightarrow \gamma$ holds, then $\alpha \rightarrow \gamma$ holds.

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

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

Union rule: if $\alpha \rightarrow \beta$ and $\alpha \rightarrow \gamma$, then $\alpha \rightarrow \beta\gamma$ holds.

Decomposition rule: if $\alpha \rightarrow \beta\gamma$ holds, then $\alpha \rightarrow \beta$ and $\alpha \rightarrow \gamma$ both hold.

Pseudotransitivity rule: if $\alpha \rightarrow \beta$ holds, and $\gamma\beta \rightarrow \delta$ holds, then $\alpha\gamma \rightarrow \delta$ holds.

Let us apply our rules to the example of schema $R = (A, B, C, G, H, I)$ and the set *F* of functional dependencies $\{ A \rightarrow B, A \rightarrow C, CG \rightarrow H, CG \rightarrow I , B \rightarrow H\}$. We list several members of F^+ here:

$A \rightarrow H$. Since $A \rightarrow B$ and $B \rightarrow H$ hold, we apply the transitivity rule. Observe that it was much easier to use Armstrong's axioms to show that $A \rightarrow H$ holds than it was to argue directly from the definitions, as we did earlier in this section.

$CG \rightarrow HI$. Since $CG \rightarrow H$ and $CG \rightarrow I$, the union rule implies that $CG \rightarrow HI$.

$AG \rightarrow I$. Since $A \rightarrow C$ and $CG \rightarrow I$, the pseudotransitivity rule implies that $AG \rightarrow I$ holds.

Another way of finding that $AG \rightarrow I$ holds is as follows: We use the augmentation rule on $A \rightarrow C$ to infer $AG \rightarrow C\,G$. Applying the transitivity rule to this dependency and $C\,G \rightarrow I$, we infer $AG \rightarrow I$.

A procedure that demonstrates formally how to use Arm-strong's axioms to compute F^+. In this procedure, when a functional dependency is added to F^+, it may be already present, and in that case there is no change to F^+.

```
F+ = F
repeat
    for each functional dependency f in F+
        apply reflexivity and augmentation rules on f
        add the resulting functional dependencies to F+

    for each pair of functional dependencies f1 and f2 in F+
        if f1 and f2 can be combined using transitivity
            add the resulting functional dependency to F+
until F+ does not change any further
```

The left-hand and right-hand sides of a functional dependency are both sub-sets of *R*. Since a set of size *n* has 2^n subsets, there are a total of $2^n \times 2^n = 2^{2n}$ possible functional dependencies, where *n* is the number of attributes in *R*. Each iteration of the repeat loop of the procedure, except the last iteration, adds at least one functional dependency to F^+. Thus, the procedure is guaranteed to terminate.

Closure of Attribute Sets

We say that an attribute *B* is **functionally determined** by a if a → *B*. To test whether a set a is a superkey, we must devise an algorithm for computing the set of attributes functionally determined by a. One way of doing this is to compute F^+, take all functional dependencies with a as the left-hand side, and take the union of the right-hand sides of all such dependencies. However, doing so can be expensive, since F^+ can be large.

An efficient algorithm for computing the set of attributes functionally deter-mined by a is useful not only for testing whether a is a superkey, but also for several other tasks

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

To illustrate how the algorithm works, we shall use it to compute $(AG)^+$ We start with *result* = *AG*. The first time that we execute the **repeat** loop to test each functional dependency, we find that:

$A \rightarrow B$ causes us to include *B* in *result*. To see this fact, we observe that $A \rightarrow B$ is in *F*, $A \subseteq$ *result*

(which is *AG*), so *result* := *result* ∪ *B*.

$A \rightarrow C$ causes *result* to become *ABCG*.

$CG \rightarrow H$ causes *result* to become *ABCGH*.

$CG \rightarrow I$ causes *result* to become *ABCGHI*.

The second time that we execute the **repeat** loop, no new attributes are added to *result*, and the algorithm terminates.

The following algorithm computes α^+ :

```
result := a;

repeat
        for each functional dependency b → g in F do
            begin
                if b ⊆ result then result := result ∪ g ;
            end
 until (result does not change)
```

If we use this algorithm on our example to calculate (AG^+) then we find:

We start with *result* = AG.

$A \rightarrow B$ causes us to include B in *result*.

$A \rightarrow C$ causes *result* to become ABCG.

$CG \rightarrow H$ causes *result* to become ABCGH.

$CG \rightarrow I$ causes *result* to become ABCGHI.

The next time we execute the while loop, no new attributes are added, and the algorithm terminates.

This algorithm has worst case behavior quadratic in the size of *F*. There is a linear algorithm that is more complicated.

Canonical Cover

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

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

F_c must also have the following properties:

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

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

logically implies F_c.

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

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

logically implies F_c.

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

To compute a canonical cover F_c for F,

```
$F_c = F$
repeat
    Use the union rule to replace any dependencies in $F_c$ of the form
        $a_1 \rightarrow b_1$ and $a_1 \rightarrow b_2$ with $a_1 \rightarrow b_1 b_2$.
    Find a functional dependency $a \rightarrow b$ in $F_c$ with an extraneous attribute either in a or in
        b.
        /* Note: the test for extraneous attributes is done using $F_c$ , not $F$ */ If an extraneous
    attribute is found, delete it from $a \rightarrow b$ in $F_c$ .
until ($F_c$ does not change)
```

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

$$A \rightarrow BC$$

$$B \rightarrow C$$

$$A \rightarrow B$$

$$AB \rightarrow C$$

we will compute F_c.

We have two dependencies with the same left hand side:

$$A \rightarrow BC$$

$$A \rightarrow B$$

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

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

Then our set is

$$A \rightarrow BC$$

$$B \rightarrow C$$

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

So we end up with, our canonical cover is

$$A \rightarrow B$$

$$B \rightarrow C$$

Decomposition

A functional **decomposition** is the process of breaking down the functions of an organization into progressively greater (finer and finer) levels of detail.

In decomposition, one function is described in greater detail by a set of other supporting functions. The decomposition of a relation scheme R consists of replacing the relation schema by two or more relation schemas that each contain a subset of the attributes of R and together include all attributes in R. Decomposition helps in eliminating some of the problems of bad design such as redundancy, inconsistencies and anomalies.

There are two types of decomposition :

1. Lossy Decomposition
2. Lossless Join Decomposition

Decomposition

1. The previous example might seem to suggest that we should decompose schema as much as possible.

   Careless decomposition, however, may lead to another form of bad design.

2. Consider a design where *Lending-schema* is decomposed into two schemas

   Branch-customer-schema = (bname, bcity, assets, cname)

   Customer-loan-schema = (cname, loan#, amount)

3. We construct our new relations from *lending* by:

   branch-customer = $\Pi_{bname, bcity, assets, cname}(lending)$

   customer-loan = $\Pi_{cname, loan\#, amount}(lending)$

bname	bcity	assets	cname
SFU	Burnaby	2M	Tom
SFU	Burnaby	2M	Mary
Downtown	Vancouver	8M	Tom

cname	loan#	amount
Tom	L-10	10K
Mary	L-20	15K
Tom	L-50	50K

Figure: The decomposed *lending* relation.

4. It appears that we can reconstruct the *lending* relation by performing a natural join on the two new schemas.
5. The following Figure shows what we get by computing *branch-customer* **1** *customer-loan*.

bname	bcity	assets	cname	loan#	amount
SFU	Burnaby	2M	Tom	L-10	10K
SFU	Burnaby	2M	Tom	L-50	50K
SFU	Burnaby	2M	Mary	L-20	15K
Downtown	Vancouver	8M	Tom	L-10	10K
Downtown	Vancouver	8M	Tom	L-50	50K

Figure : Join of the decomposed relations.

6. We notice that there are tuples in *branch-customer* **1** *customer-loan* that are not in *lending*.
7. How did this happen?
   - The intersection of the two schemas is *cname*, so the natural join is made on the basis of equality in the cname.
   - If two lendings are for the same customer, there will be four tuples in the natural join.
   - Two of these tuples will be spurious - they will not appear in the original *lending* relation, and should not appear in the database.
   - Although we have **more** tuples in the join, we have **less** information.
   - Because of this, we call this a **lossy** or **lossy-join decomposition**.
   - A decomposition that is not lossy-join is called a **lossless-join decomposition**.
   - The only way we could make a connection between *branch-customer* and *customer-loan* was through *cname*.
8. When we decomposed *Lending-schema* into *Branch-schema* and *Loan-info-schema*, we will not have a similar problem. Why not?

   Branch-schema = (bname, bcity, assets)

   Branch-loan-schema = (bname, cname, loan#, amount)

   - The only way we could represent a relationship between tuples in the two relations is through *bname*.
   - This will not cause problems.
   - For a given branch name, there is exactly one assets value and branch city.
9. For a given branch name, there is exactly one assets value and exactly one bcity; whereas a similar statement associated with a loan depends on the customer, not on the amount of the loan (which is not unique).
10. We'll make a more formal definition of lossless-join:

   Let *R* be a relation schema.

   A set of relation schemas $\{R_1, R_2, \ldots, R_n\}$ is a **decomposition** of *R* if

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

   That is, every attribute in *R* appears in at least one R_i for $1 \leq i \leq n$.

   Let *r* be a relation on *R*, and let

$$r_i = \Pi_{R_i}(r) \text{ for } 1 \leq i \leq n$$

That is, $\{r_1, r_2, \ldots r_n\}$ is the database that results from decomposing *R* into $\{R_1, R_2, \ldots R_n\}$.

It is always the case that:

$$r \subseteq r_1 \; 1 \; r_2 \; 1 \ldots 1 \; r_n$$

To see why this is, consider a tuple $t \in r$.

When we compute the relations $\{r_1, r_2, \ldots r_n\}$, the tuple *t* gives rise to one tuple t_i in each r_i.

These *n* tuples combine together to regenerate *t* when we compute the natural join of the r_i.

Thus every tuple in *r* appears in $1_{i=1}^{n} r_i$.

However, in general,

$$r \neq r_1 \; 1 \; r_2 \; 1 \ldots 1 \; r_n$$

We saw an example of this inequality in our decomposition of *lending* into *branch-customer* and *customer-loan*.

In order to have a lossless-join decomposition, we need to impose some constraints on the set of possible relations.

Let *C* represent a set of constraints on the database.

A decomposition $\{R_1, R_2, \ldots, R_n\}$ of a relation schema *R* is a **lossless-join decomposition** for *R* if, for all relations *r* on schema *R* that are legal under *C*:

$$r = \Pi_{R_1}(r) \; 1 \; \Pi_{R_2}(r) \; 1 \ldots 1 \; \Pi_{R_n}(r)$$

11. In other words, a lossless-join decomposition is one in which, for any legal relation *r*, if we decompose *r* and then ``recompose'' *r*, we get what we started with - no more and no less.

Desirable Properties of Decomposition

1. We'll take another look at the schema

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

   which we saw was a bad design.

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

bname $\rightarrow$ *assets bcity*

loan# $\rightarrow$ *amount bname*

3. If we decompose it into

   Branch-schema = (bname, assets, bcity)

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

   Borrow-schema = (cname, loan#)

   we claim this decomposition has several desirable properties.

Lossless-Join Decomposition

1. We claim the above decomposition is lossless. How can we decide whether a decomposition is lossless?
   - Let *R* be a relation schema.
   - Let *F* be a set of functional dependencies on *R*.
   - Let R_1 and R_2 form a decomposition of *R*.
   - The decomposition is a lossless-join decomposition of *R* if at least one of the following functional dependencies are in F^+ :
     1. $R_1 \cap R_2 \rightarrow R_1$
     2. $R_1 \cap R_2 \rightarrow R_2$

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

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

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

   First we decompose *Lending-schema* into

   Branch-schema = (bname, bcity, assets)

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

   Since *bname* $\rightarrow$ *assets bcity*, the augmentation rule for functional dependencies implies that

   bname $\rightarrow$ *bname assets bcity*

   Since *Branch-schema* $\cap$ *Borrow-schema* = *bname*, our decomposition is lossless join.

   Next we decompose *Borrow-schema* into

Loan-schema = (bname, loan#, amount)

Borrow-schema = (cname, loan#)

As *loan#* is the common attribute, and

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

This is also a lossless-join decomposition.

Dependency Preservation

1. Another desirable property in database design is **dependency preservation**.
   - We would like to check easily that updates to the database do not result in illegal relations being created.
   - It would be nice if our design allowed us to check updates without having to compute natural joins.
   - To know whether joins must be computed, we need to determine what functional dependencies may be tested by checking each relation individually.
   - Let *F* be a set of functional dependencies on schema *R*.
   - Let $\{R_1, R_2, \ldots, R_n\}$ be a decomposition of *R*.
   - The **restriction** of *F* to R_i is the set of all functional dependencies in F^+ that include only attributes of R_i.
   - Functional dependencies in a restriction can be tested in one relation, as they involve attributes in one relation schema.
   - The set of restrictions $F_1, F_2, \ldots, F_n$ is the set of dependencies that can be checked efficiently.
   - We need to know whether testing only the restrictions is sufficient.
   - Let $F' = F_1, F_2, \ldots, F_n$.
   - *F*' is a set of functional dependencies on schema *R*, but in general, $F' \neq F$.
   - However, it may be that $F'^+ = F^+$.
   - If this is so, then every functional dependency in *F* is implied by *F*', and if *F*' is satisfied, then *F* must also be satisfied.
   - A decomposition having the property that $F'^+ = F^+$ is a **dependency-preserving** decomposition.
2. The algorithm for testing dependency preservation follows this method:

```
compute $F^+$;
for each schema $R_i$ in $D$ do
begin
$F_i$ : = the restriction of $F^+$ to $R_i$;
end
$F$ := $\varnothing$
for each restriction $F_i$ do
begin
$F = F \cup F_i$
```

```
end
compute $F^{+}$;

if ($F^{+} = F^{+}$) then return (true)

else return (false);
```

3. We can now show that our decomposition of *Lending-schema* is dependency preserving.
   - The functional dependency
   - *bname* $\rightarrow$ *assets bcity*

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

   - The functional dependency
   - *loan#* $\rightarrow$ *amount bname*

     can be tested in *Loan-schema*.

4. As the above example shows, it is often easier not to apply the algorithm shown to test dependency preservation, as computing F^+ takes exponential time.
5. **An Easier Way To Test For Dependency Preservation**

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

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

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

   - Find *F* - *F*', the functional dependencies not checkable in one relation.
   - See whether this set is obtainable from *F*' by using Armstrong's Axioms.
   - This should take a great deal less work, as we have (usually) just a few functional dependencies to work on.

   Use this simpler method on exams and assignments (unless you have exponential time available to you).

Repetition of Information

1. Our decomposition does not suffer from the repetition of information problem.
   - Branch and loan data are separated into distinct relations.
   - Thus we do not have to repeat branch data for each loan.
   - If a single loan is made to several customers, we do not have to repeat the loan amount for each customer.
   - This lack of redundancy is obviously desirable.
   - We will see how this may be achieved through the use of **normal forms**.

Boyce-Codd Normal Form

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

$\alpha \rightarrow \beta$ is a trivial functional dependency (i.e. $\beta \subseteq \alpha$).

α is a superkey for schema *R*.

A database design is in BCNF if each member of the set of relation schemas is in BCNF.

Let's assess our example banking design:

```
Customer-schema = (cname, street, ccity)

          cname  →  street ccity

Branch-schema = (bname, assets, bcity)

          bname  →  assets bcity

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

          loan#  →  amount bname
```

Customer-schema and *Branch-schema* are in BCNF.

Let's look at *Loan-info-schema*:

We have the non-trivial functional dependency *loan#* $\rightarrow$ *amount*, and

loan# is not a superkey.

Thus *Loan-info-schema* is not in BCNF.

We also have the repetition of information problem.

For each customer associated with a loan, we must repeat the branch name and amount of the loan.

We can eliminate this redundancy by decomposing into schemas that are all in BCNF.

If we decompose into

```
Loan-schema = (bname, loan#, amount)

          Borrow-schema = (cname, loan#)
```

we have a lossless-join decomposition. (Remember why?)

To see whether these schemas are in BCNF, we need to know what functional dependencies apply to them.

- For *Loan-schema*, we have *loan#* $\rightarrow$ *amount bname* applying.
- Only trivial functional dependencies apply to *Borrow-schema*.
- Thus both schemas are in BCNF.

We also no longer have the repetition of information problem. Branch name and loan amount information are not repeated for each customer in this design.

Now we can give a general method to generate a collection of BCNF schemas.

```
result := {R};
done := false;
compute $F^+$;
while (not done) do
  if (there is a schema $R_i$ in result that is not in BCNF)
    then begin
        let a → b be a nontrivial functional dependency that holds on $R_i$ such that a
        → $R_i$ is not in $F^+$, and a ∩ b = ∅ ;
        result := (result – $R_i$ ) ∪ ($R_i$ – b) ∪ ( a, b);
      end
else done := true;
```

This algorithm generates a lossless-join BCNF decomposition. Why?

We replace a schema R_i with $(R_i - \beta)$ and (α, β).

The dependency $\alpha \rightarrow \beta$ holds on R_i.

$(R_i - \beta) \cap (\alpha, \beta) = \alpha$.

So we have $R_1 \cap R_2 \rightarrow R_2$, and thus a lossless join.

Let's apply this algorithm to our earlier example of poor database design:

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

The set of functional dependencies we require to hold on this schema are

```
bname  →  assets bcity

loan#  →  amount bname
```

A candidate key for this schema is {*loan#, cname*}.

We will now proceed to decompose:

The functional dependency

```
bname  →  assets bcity
```

holds on *Lending-schema*, but *bname* is not a superkey.

We replace *Lending-schema* with

```
Branch-schema = (bname, assets, bcity)

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

Branch-schema is now in BCNF.

The functional dependency

```
loan#  →  amount bname
```

holds on *Loan-info-schema*, but *loan#* is not a superkey.

We replace *Loan-info-schema* with

```
Loan-schema = (bname, loan#, amount)

Borrow-schema = (cname, loan#)
```

These are both now in BCNF.

We saw earlier that this decomposition is both lossless-join and dependency-preserving.

2. Not every decomposition is dependency-preserving.

Consider the relation schema

```
Banker-schema = (bname, cname, banker-name)
```

The set *F* of functional dependencies is

```
banker-name  →  bname
```

cname bname $\rightarrow$ *banker-name*

The schema is not in BCNF as *banker-name* is not a superkey.

If we apply our algorithm, we may obtain the decomposition

Banker-branch-schema = (bname, banker-name)

Cust-banker-schema = (cname, banker-name)

The decomposed schemas preserve only the first (and trivial) functional dependencies.

The closure of this dependency does not include the second one.

Thus a violation of *cname bname* $\rightarrow$ *banker-name* cannot be detected unless a join is computed.

This shows us that not every BCNF decomposition is dependency-preserving.

It is not always possible to satisfy all three design goals:

BCNF.

Lossless join.

Dependency preservation.

We can see that any BCNF decomposition of *Banker-schema* must fail to preserve

cname bname $\rightarrow$ *banker-name*

Some Things To Note About BCNF

There is sometimes more than one BCNF decomposition of a given schema.

The algorithm given produces only one of these possible decompositions.

Some of the BCNF decompositions may also yield dependency preservation, while others may not.

Changing the order in which the functional dependencies are considered by the algorithm may change the decomposition.

For example, try running the BCNF algorithm on

$$R = (A, B, C, D)$$

$$A \rightarrow B, C$$

$$B \rightarrow D$$

$$D \rightarrow B$$

Then change the order of the last two functional dependencies and run the algorithm again. Check the two decompositions for dependency preservation.

Third Normal Form

When we cannot meet all three design criteria, we abandon BCNF and accept a weaker form called **third normal form (3NF)**.

It is always possible to find a dependency-preserving lossless-join decomposition that is in 3NF.

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

$\alpha \rightarrow \beta$ is a trivial functional dependency.

α is a superkey for schema *R*.

Each attribute *A* in $\beta - \alpha$ is contained in a candidate key for *R*.

A database design is in 3NF if each member of the set of relation schemas is in 3NF.

We now allow functional dependencies satisfying only the third condition. These dependencies are called **transitive dependencies**, and are not allowed in BCNF.

As all relation schemas in BCNF satisfy the first two conditions only, a schema in BCNF is also in 3NF.

BCNF is a more restrictive constraint than 3NF.

Our *Banker-schema* decomposition did not have a dependency-preserving lossless-join decomposition into BCNF. The schema was already in 3NF though (check it out).

We now present an algorithm for finding a dependency-preserving lossless-join decomposition into 3NF.

Note that we require the set *F* of functional dependencies to be in **canonical form**.

```
let $F_c$ be a canonical cover for $F$;

i:= 0;
for each functional dependency a → b in $F_c$
```

```
        i:= $i$ + 1; $R_i$ := a b;
    if none of the schemas $R_j$ , $j$ = 1, 2, . . . , $i$ contains a candidate key for $R$
      then
        i:= $i$ + 1;
        $R_i$ := any candidate key for $R$;
    /* Optionally, remove redundant relations */
    repeat
        if any schema $R_j$ is contained in another schema $R_k$
          then
            /* Delete $R_j$ */
            $R_j$ := $R_i$ ;

            i:= $i$ + 1;
    until no more $R_j$ s can be deleted
return ($R_1$, $R_2$, . . . , $R_i$)
```

Each relation schema is in . Why? (A proof is given is [Ullman 1988].)

The design is as a schema is built for each given dependency.

is guaranteed by the requirement that a candidate key for *R* be in at least one of the schemas.

To review our *Banker-schema* consider an extension to our example:

Banker-info-schema = (bname, cname, banker-name, office#)

The set *F* of functional dependencies is

$$\textit{banker-name} \rightarrow \textit{bname office\#}$$

$$\textit{cname bname} \rightarrow \textit{banker-name}$$

The **for** loop in the algorithm gives us the following decomposition:

Banker-office-schema = (banker-name, bname, office#)

Banker-schema = (cname, bname, banker-name)

Since *Banker-schema* contains a candidate key for *Banker-info-schema*, the process is finished.

Comparison of BCNF and 3NF

1. We have seen BCNF and 3NF.

   - It is always possible to obtain a 3NF design without sacrificing lossless-join or dependency-preservation.
   - If we do not eliminate all transitive dependencies, we may need to use null values to represent some of the meaningful relationships.
   - Repetition of information occurs.
2. These problems can be illustrated with *Banker-schema*.
   - As *banker-name* $\rightarrow$ *bname* , we may want to express relationships between a banker and his or her branch.

cname	banker-name	bname
Bill	John	SFU
Tom	John	SFU
Mary	John	SFU
null	Tim	Austin

Figure : An instance of *Banker-schema*.

   - The above Figure shows how we must either have a corresponding value for customer name, or include a null.
   - Repetition of information also occurs.
   - Every occurrence of the banker's name must be accompanied by the branch name.
3. If we must choose between BCNF and dependency preservation, it is generally better to opt for 3NF.
   - If we cannot check for dependency preservation efficiently, we either pay a high price in system performance or risk the integrity of the data.
   - The limited amount of redundancy in 3NF is then a lesser evil.
4. To summarize, our goal for a relational database design is
   - BCNF.
   - Lossless-join.
   - Dependency-preservation.
5. If we cannot achieve this, we accept
   - 3NF
   - Lossless-join.
   - Dependency-preservation.
6. **A final point:** there is a price to pay for decomposition. When we decompose a relation, we have to use natural joins or Cartesian products to put the pieces back together. This takes computation

Normalization Using Multivalued Dependencies (not to be covered)

Suppose that in our banking example, we had an alternative design including the schema:

BC-schema = (loan#, cname, street, ccity)

We can see this is not BCNF, as the functional dependency

cname $\rightarrow$ *street ccity*

holds on this schema, and *cname* is not a superkey.

If we have customers who have several addresses, though, then we no longer wish to enforce this functional dependency, and the schema is in BCNF.

However, we now have the repetition of information problem. For each address, we must repeat the loan numbers for a customer, and vice versa.

Multivalued Dependencies

Functional dependencies rule out certain tuples from appearing in a relation.

If $A \rightarrow B$, then we cannot have two tuples with the same *A* value but different *B* values.

Multivalued dependencies do not rule out the existence of certain tuples.

Instead, they **require** that other tuples of a certain form be present in the relation.

Let *R* be a relation schema, and let $\alpha \subseteq R$ and $\beta \subseteq R$.

The **multivalued dependency**

$$\alpha \twoheadrightarrow \beta$$

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

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

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

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

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

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

Figure 1 (textbook 6.10) shows a tabular representation of this. It looks horrendously complicated, but is really rather simple. A simple example is a table with the schema (*name*, *address*, *car*), as shown in Figure 2.

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

Figure 1: Tabular representation of $\alpha \twoheadrightarrow \beta$.

name	address	car
Tom	North Rd.	Toyota
Tom	Oak St.	Honda
Tom	North Rd.	Honda
Tom	Oak St.	Toyota

Figure 2: (*name*, *address*, *car*) where $name \twoheadrightarrow address$ and $name \twoheadrightarrow car$.

- Intuitively, $\alpha \twoheadrightarrow \beta$ says that the relationship between α and β is independent of the relationship between α and $R-\beta$.
- If the multivalued dependency $\alpha \twoheadrightarrow \beta$ is satisfied by all relations on schema *R*, then we say it is a **trivial** multivalued dependency on schema *R*.
- Thus $\alpha \twoheadrightarrow \beta$ is trivial if $\beta \subseteq \alpha$ or $\beta \cup \alpha = R$.

Look at the example relation *bc* relation in Figure 3.

loan#	cname	street	ccity
23	Smith	North	Rye
23	Smith	Main	Manchester
93	Curry	Lake	Horseneck

Figure 3: Relation bc, an example of redundancy in a BCNF relation.

- We must repeat the loan number once for each address a customer has.
- We must repeat the address once for each loan the customer has.
- This repetition is pointless, as the relationship between a customer and a loan is independent of the relationship between a customer and his or her address.
- If a customer, say ``Smith", has loan number 23, we want all of Smith's addresses to be associated with that loan.
- Thus the relation of Figure 4 is illegal.
- If we look at our definition of multivalued dependency, we see that we want the multivalued dependency
- $cname \twoheadrightarrow street\ ccity$
-

to hold on *BC-schema*.

loan#	cname	street	ccity
23	Smith	North	Rye
27	Smith	Main	Manchester

Figure 4: An illegal be relation.

Note that if a relation *r* fails to satisfy a given multivalued dependency, we can construct a relation *r*' that does satisfy the multivalued dependency by adding tuples to *r*.

Theory of Multivalued Dependencies

We will need to compute all the multivalued dependencies that are logically implied by a given set of multivalued dependencies.

- Let *D* denote a set of functional and multivalued dependencies.
- The closure D^+ of *D* is the set of all functional and multivalued dependencies logically implied by *D*.
- We can compute D^+ from *D* using the formal definitions, but it is easier to use a set of inference rules.

The following set of inference rules is **sound** and **complete**. The first three rules are Armstrong's axioms from Chapter 5.

Reflexivity rule: if α is a set of attributes and $\beta \subseteq \alpha$, then $\alpha \rightarrow \beta$ holds.

Augmentation rule: if $\alpha \rightarrow \beta$ holds, and γ is a set of attributes, then $\gamma\alpha \rightarrow \gamma\beta$ holds.

Transitivity rule: if $\alpha \rightarrow \beta$ holds, and $\beta \rightarrow \gamma$ holds, then $\alpha \rightarrow \gamma$ holds.

Complementation rule: if $\alpha \twoheadrightarrow \beta$ holds, then $\alpha \twoheadrightarrow R - \beta - \alpha$ holds.

Multivalued augmentation rule: if $\alpha \twoheadrightarrow \beta$ holds, and $\gamma \subseteq R$ and $\delta \subseteq \gamma$, then $\gamma\alpha \twoheadrightarrow \delta\beta$ holds.

Multivalued transitivity rule: if $\alpha \twoheadrightarrow \beta$ holds, and $\beta \twoheadrightarrow \gamma$ holds, then $\alpha \twoheadrightarrow \gamma - \beta$ holds.

Replication rule: if $\alpha \rightarrow \beta$ holds, then $\alpha \twoheadrightarrow \beta$.

Coalescence rule: if $\alpha \twoheadrightarrow \beta$ holds, and $\gamma \subseteq \beta$, and there is a δ such that $\delta \subseteq R$ and $\delta \cap \beta = \emptyset$ and $\delta \rightarrow \gamma$, then $\alpha \rightarrow \gamma$ holds.

An example of *multivalued transitivity rule* is as follows. $loan\# \twoheadrightarrow cname$ and $cname \twoheadrightarrow \{cname, caddress\}$. Thus we have $loan\# \twoheadrightarrow caddress$, where $caddress = \{cname, caddress\} - cname$.

An example of *coalescence rule* is as follows. If we have $student\_name \twoheadrightarrow \{bank, account\}$, and $student\_id \rightarrow bank$, then we have $student\_name \rightarrow bank$.

Let's do an example:

Let *R*=(*A*,*B*,*C*,*G*,*H*,*I*) be a relation schema.

Suppose $A \twoheadrightarrow BC$ holds.

The definition of multivalued dependencies implies that if $t_1[A] = t_2[A]$, then there exists tuples t_3 and t_4 such that:

$$t_1[A] = t_2[A] = t_3[A] = t_4[A]$$

$$t_3[BC] = t_1[BC]$$

$$t_3[GHI] = t_2[GHI]$$

$$t_4[GHI] = t_1[GHI]$$

$$t_4[BC] = t_2[BC]$$

The complementation rule states that if $A \twoheadrightarrow BC$ then $A \twoheadrightarrow GHI$.

Tuples t_3 and t_4 satisfy $A \twoheadrightarrow GHI$ if we simply change the subscripts.

We can simplify calculating D^+, the closure of *D* by using the following rules, derivable from the previous ones:

- **Multivalued union rule:** if $\alpha \twoheadrightarrow \beta$ holds and $\alpha \twoheadrightarrow \gamma$ holds, then $\alpha \twoheadrightarrow \beta\gamma$ holds.
- **Intersection rule:** if $\alpha \twoheadrightarrow \beta$ holds and $\alpha \twoheadrightarrow \gamma$ holds, then $\alpha \twoheadrightarrow \beta \cap \gamma$ holds.
- **Difference rule:** if $\alpha \twoheadrightarrow \beta$ holds and $\alpha \twoheadrightarrow \gamma$ holds, then $\alpha \twoheadrightarrow \beta - \gamma$ holds and $\alpha \twoheadrightarrow \gamma - \beta$ holds.

An example will help:

Let *R*=(*A*,*B*,*C*,*G*,*H*,*I*) with the set of dependencies:

$$A \twoheadrightarrow B$$

$$B \twoheadrightarrow HI$$

$$CG \rightarrow H$$

We list some members of D^+:

$A \twoheadrightarrow CGHI$: since $A \twoheadrightarrow B$, complementation rule implies that $A \twoheadrightarrow R - B - A$, and *R - B - A = CGHI.*

$A \twoheadrightarrow HI$: Since $A \twoheadrightarrow B$ and $B \twoheadrightarrow HI$, multivalued transitivity rule implies that $A \twoheadrightarrow HI - B$.

$B \rightarrow H$: coalescence rule can be applied. $B \twoheadrightarrow HI$ holds, $H \subseteq HI$ and $CG \rightarrow H$ and $CG \cap HI = \emptyset$, so we can satisfy the coalescence rule with α being *B*, β being *HI*, δ being *CG*, and γ being *H*. We conclude that $B \rightarrow H$.

$A \twoheadrightarrow CG$: now we know that $A \twoheadrightarrow CGHI$ and $A \twoheadrightarrow HI$. By the difference rule, $A \twoheadrightarrow CGHI - HI = CG$.

Fourth Normal Form (4NF)

We saw that *BC-schema* was in BCNF, but still was not an ideal design as it suffered from repetition of information. We had the multivalued dependency *cname* $\twoheadrightarrow$ *street ccity*, but no non-trivial functional dependencies.

We can use the given multivalued dependencies to improve the database design by decomposing it into **fourth normal form**.

A relation schema *R* is in 4NF with respect to a set *D* of functional and multivalued dependencies if for all multivalued dependencies in D^+ of the form $\alpha \twoheadrightarrow \beta$, where $\alpha \subseteq R$ and $\beta \subseteq R$, at least one of the following hold:

- $\alpha \twoheadrightarrow \beta$ is a trivial multivalued dependency.
- α is a superkey for schema *R*.

A database design is in 4NF if each member of the set of relation schemas is in 4NF.

The definition of 4NF differs from the BCNF definition only in the use of multivalued dependencies.

- Every 4NF schema is also in BCNF.
- To see why, note that if a schema is not in BCNF, there is a non-trivial functional dependency $\alpha \rightarrow \beta$ holding on *R*, where α is not a superkey.
- Since $\alpha \rightarrow \beta$ implies $\alpha \twoheadrightarrow \beta$, by the replication rule, *R* cannot be in 4NF.

We have an algorithm similar to the BCNF algorithm for decomposing a schema into 4NF:

```
result := {R};
done := false;
```

```
compute $D^+$ ; Given schema $R_i$ , let $D_i$ denote the restriction of $D^+$ to $R_i$ while (not done) do
  if (there is a schema $R_i$ in result that is not in 4NF w.r.t. $D_i$ )
    then begin
        let a →→ b be a nontrivial multivalued dependency that holds

        on $R_i$ such that a → $R_i$ is not in $D_i$ , and a ∩ b = ∅; result := (result – $R_i$ ) ∪ ($R_i$ – b) ∪ (a, b);
      end
else done := true;
```

If we apply this algorithm to *BC-schema*:

cname $\twoheadrightarrow$ *loan#* is a nontrivial multivalued dependency and *cname* is not a superkey for the schema.

We then replace *BC-schema* by two schemas:

Cust-loan-schema=(cname, loan#)

Customer-schema=(cname, street, ccity)

These two schemas are in 4NF.

We can show that our algorithm generates only lossless-join decompositions.

Let *R* be a relation schema and *D* a set of functional and multivalued dependencies on *R*.

Let R_1 and R_2 form a decomposition of *R*.

This decomposition is lossless-join if and only if at least one of the following multivalued dependencies is in D^+ :

$$R_1 \cap R_2 \twoheadrightarrow R_1$$

$$R_1 \cap R_2 \twoheadrightarrow R_2$$

We saw similar criteria for functional dependencies.

This says that for **every** lossless-join decomposition of *R* into two schemas R_1 and R_2 , one of the two above dependencies must hold.

You can see, by inspecting the algorithm, that this must be the case for every decomposition.

Dependency preservation is not as simple to determine as with functional dependencies.

Let *R* be a relation schema.

Let $R_1, R_2, \ldots R_n$ be a decomposition of *R*.

Let *D* be the set of functional and multivalued dependencies holding on *R*.

The **restriction** of *D* to R_i is the set D_i consisting of:

All functional dependencies in D^+ that include only attributes of R_i.

All multivalued dependencies of the form $\alpha \twoheadrightarrow \beta \cap R_i$ where $\alpha \subseteq R_i$ and $\alpha \twoheadrightarrow \beta$ is in D^+.

A decomposition of schema *R* is dependency preserving with respect to a set *D* of functional and multivalued dependencies if for every set of relations $r_1(R_1), r_2(R_2), \ldots r_n(R_n)$ such that for all *i*, r_i satisfies D_i, there exists a relation *r*(*R*) that satisfies *D* and for which $r_i = \Pi_{R_i}(r)$ for all *i*.

What does this formal statement say? It says that a decomposition is dependency preserving if for every set of relations on the decomposition schema satisfying only the restrictions on *D* there exists a relation *r* on the entire schema *R* that the decomposed schemas can be derived from, and that *r* also satisfies the functional and multivalued dependencies.

We'll do an example using our decomposition algorithm and check the result for dependency preservation.

Let *R*=(*A*,*B*,*C*,*G*,*H*,*I*).

Let D be

$$A \twoheadrightarrow B$$

$$B \twoheadrightarrow HI$$

$$CG \rightarrow H$$

R is not in 4NF, as we have $A \twoheadrightarrow B$ and *A* is not a superkey.

The algorithm causes us to decompose using this dependency into

$$R_1 = (A, B)$$

$$R_2 = (A, C, G, H, I)$$

R_1 is now in 4NF, but R_2 is not.

Applying the multivalued dependency $CG \twoheadrightarrow H$ (how did we get this?), our algorithm then decomposes R_2 into

$$R_3 = (C, G, H)$$

$$R_4 = (A, C, G, I)$$

R_3 is now in 4NF, but R_4 is not.

Why? As $A \twoheadrightarrow HI$ is in D^+ (why?) then the restriction of this dependency to R_4 gives us $A \twoheadrightarrow I$.

Applying this dependency in our algorithm finally decomposes R_4 into

$$R_5 = (A, I)$$

$$R_6 = (A, C, G)$$

The algorithm terminates, and our decomposition is R_1, R_3, R_5 and R_6 .

Let's analyze the result.

r_1 :

A	B
a_1	b_1
a_2	b_1

r_2 :

C	G	H
c_1	g_1	h_1
c_2	g_2	h_2

r_3 :

A	I
a_1	i_1
a_1	i_2

r_4 :

A	C	G
a_1	c_1	g_1
a_2	c_2	g_2

Figure 1: Projection of relation *r* onto a 4NF decomposition of *R*.

This decomposition is not dependency preserving as it fails to preserve $B \twoheadrightarrow HI$.

Figure 1 - shows four relations that may result from projecting a relation onto the four schemas of our decomposition.

The restriction of *D* to (*A*,*B*) is $A \twoheadrightarrow B$ and some trivial dependencies.

We can see that r_1 satisfies $A \twoheadrightarrow B$ as there are no pairs with the same *A* value.

Also, r_2 satisfies all functional and multivalued dependencies since no two tuples have the same value on any attribute.

We can say the same for r_3 and r_4 .

So our decomposed version satisfies all the dependencies in the restriction of *D*.

However, there is no relation *r* on (*A*,*B*,*C*,*G*,*H*,*I*) that satisfies *D* and decomposes into r_1, r_2, r_3 and r_4.

Figure 2 (textbook 6.15) shows $r = r_1 1 r_2 1 r_3 1 r_4$.

Relation *r* does not satisfy $B \twoheadrightarrow HI$.

Any relation *s* containing *r* and satisfying $B \twoheadrightarrow HI$ must include the tuple $(a_2, b_1, c_2, g_2, h_1, i_1)$.

However, $\Pi_{CGH}(s)$ includes a tuple (c_2, g_2, h_1) that is not in r_2.

Thus our decomposition fails to detect a violation of $B \twoheadrightarrow HI$.

A	B	C	G	H	I
a_1	b_1	c_1	g_1	h_1	i_1
a_2	b_1	c_2	g_2	h_2	i_2

Figure 2: A relation *r*(*R*) that does not satisfy $B \twoheadrightarrow HI$.

We have seen that if we are given a set of functional and multivalued dependencies, it is best to find a database design that meets the three criteria:

- 4NF.
- Dependency Preservation.
- Lossless-join.

If we only have functional dependencies, the first criteria is just BCNF.

We cannot always meet all three criteria. When this occurs, we compromise on 4NF, and accept BCNF, or even 3NF if necessary, to ensure dependency preservation.

More Normal Forms

The fourth normal form is by no means the "ultimate" normal form. As we saw earlier, multivalued dependencies help us understand and eliminate some forms of repetition of information that cannot be understood in terms of functional dependencies. There are types of constraints called **join dependencies** that generalize multivalued dependencies, and lead to another normal form called **project-join normal form (PJNF)** (PJNF is called **fifth normal form** in some books). There is a class of even more general constraints that leads to a normal form called **domain-key normal form (DKNF).**

A practical problem with the use of these generalized constraints is that they are not only hard to reason with, but there is also no set of sound and complete inference rules for reasoning about the constraints. Hence PJNF and DKNF are used quite rarely.

Database-Design Process

So far we have looked at detailed issues about normal forms and normalization. In this section, we study how normalization fits into the overall database-design process.

we assumed that a relation schema *r*(*R*) is given, and proceeded to normalize it. There are several ways in which we could have come up with the schema *r*(*R*):

r(*R*) could have been generated in converting an E-R diagram to a set of relation schemas.

r(*R*) could have been a single relation schema containing *all* attributes that are of interest. The normalization process then breaks up *r*(*R*) into smaller schemas.

r(*R*) could have been the result of an ad-hoc design of relations that we then test to verify that it satisfies a desired normal form.

We also examine some practical issues in database design, including denormalization for performance and examples of bad design that are not detected by normalization.

E-R Model and Normalization

When we define an E-R diagram carefully, identifying all entities correctly, the relation schemas generated from the E-R diagram should not need much further normalization. However, there can be functional dependencies between attributes of an entity. For instance, suppose an *instructor* entity set had attributes *dept name* and *dept address*, and there is a functional dependency *dept name* → *dept address*. We would then need to normalize the relation generated from *instructor*.

Most examples of such dependencies arise out of poor E-R diagram design.

In the above example, if we had designed the E-R diagram correctly, we would have created a *department* entity set with attribute *dept address* and a relationship set between *instructor* and *department*. Similarly, a relationship set involving more than two entity sets may result in a schema that may not be in a desirable normal form. Since most relationship sets are binary, such cases are relatively rare.

(In fact, some E-R-diagram variants actually make it difficult or impossible to specify nonbinary relationship sets.)

Functional dependencies can help us detect poor E-R design. If the generated relation schemas are not in desired normal form, the problem can be fixed in the E-R diagram. That is, normalization can be done formally as part of data modeling. Alternatively, normalization can be left to the designer's intuition during E-R modeling, and can be done formally on the relation schemas generated from the E-R model.

A careful reader will have noted that in order for us to illustrate a need for multivalued dependencies and fourth normal form, we had to begin with schemas that were not derived from our E-R design. Indeed, the process of creating an E-R design tends to generate 4NF designs. If a multivalued dependency holds and is not implied by the corresponding functional dependency,

it usually arises from one of the following sources:

A many-to-many relationship set.

A multivalued attribute of an entity set.

For a many-to-many relationship set each related entity set has its own schema and there is an additional schema for the relationship set. For a multivalued attribute, a separate schema is created consisting of that attribute and the primary key of the entity set (as in the case of the *phone number* attribute of the entity set *instructor*).

The universal-relation approach to relational database design starts with an assumption that there is one single relation schema containing all attributes of interest. This single schema defines how users and applications interact with the database.

Naming of Attributes and Relationships

desirable feature of a database design is the **unique-role assumption**, which means that each attribute name has a unique meaning in the database. This prevents us from using the same attribute to mean different things in different schemas.

For example, we might otherwise consider using the attribute *number* for phone number in the *instructor* schema and for room number in the *classroom* schema. The join of a relation on schema *instructor* with one on *classroom* is meaningless. While users and application developers can work carefully to ensure use of the right *number* in each circumstance, having a different attribute name for phone number and for room number serves to reduce user errors.

While it is a good idea to keep names for incompatible attributes distinct, if attributes of different relations have the same meaning, it may be a good idea to use the same attribute name. For this reason we used the same attribute name "*name*" for both the *instructor* and the *student* entity sets. If this was not the case (that is, we used different naming conventions for the instructor and student names), then if we wished to generalize these entity sets by creating a *person* entity set, we would have to rename the attribute. Thus, even if we did not currently have a generalization of *student* and *instructor*, if we foresee such a possibility it is best to use the same name in both entity sets (and relations).

Denormalization for Performance

Occasionally database designers choose a schema that has redundant information; that is, it is not normalized. They use the redundancy to improve performance for specific applications. The penalty paid for not using a normalized schema is the extra work (in terms of coding time and execution time) to keep redundant data consistent.

For instance, suppose all course prerequisites have to be displayed along with a course information, every time a course is accessed. In our normalized schema, this requires a join of *course* with *prereq*.

One alternative to computing the join on the fly is to store a relation containing all the attributes of *course* and *prereq*. This makes displaying the "full" course information faster. However, the information for a course is repeated for every course prerequisite, and all copies must be updated by the application, whenever a course prerequisite is added or dropped. The process of taking a normalized schema and making it nonnormalized is called **denormalization**, and designers use it to tune performance of systems to support time-critical operations.

Other Design Issues

There are some aspects of database design that are not addressed by normalization, and can thus lead to bad database design. Data pertaining to time or to ranges of time have several such issues. We give examples here; obviously, such designs should be avoided.

Consider a university database, where we want to store the total number of instructors in each department in different years. A relation *total inst*(*dept name*, *year*, *size*) could be used to store the desired information. The only functional dependency on this relation is *dept name*, *year*→ *size*, and the relation is in BCNF.An alternative design is to use multiple relations, each storing the size information for a different year. Let us say the years of interest are 2007, 2008, and 2009; we would then have relations of the form *total inst 2007*, *total inst 2008*, *total inst 2009*, all of which are on the schema (*dept name*, *size*). The only functional dependency here on each relation would be *dept name* → *size*, so these relations are also in BCNF. However, this alternative design is clearly a bad idea — we would have to create a new relation every year, and we would also have to write new queries every year, to take each new relation into account. Queries would also be more complicated since they may have to refer to many relations.

UNIT - IV

Basic Concepts

An index for a file in a database system works in much the same way as the index in this textbook. If we want to learn about a particular topic (specified by a word or a phrase) in this textbook, we can search for the topic in the index at the back of the book, find the pages where it occurs, and then read the pages to find the information for which we are looking. The words in the index are in sorted order, making it easy to find the word we want. Moreover, the index is much smaller than the book, further reducing the effort needed.

Database-system indices play the same role as book indices in libraries. For example, to retrieve a *student* record given an *ID*, the database system would look up an index to find on which disk block the corresponding record resides, and then fetch the disk block, to get the appropriate *student* record.

Keeping a sorted list of students' *ID* would not work well on very large databases with thousands of students, since the index would itself be very big; further, even though keeping the index sorted reduces the search time, finding a student can still be rather time-consuming. Instead, more sophisticated indexing techniques may be used.

There are two basic kinds of indices:

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

We shall consider several techniques for both ordered indexing and hashing. No one technique is the best. Rather, each technique is best suited to particular database applications. Each technique must be evaluated on the basis of these factors:

- **Access types**: The types of access that are supported efficiently. Access types can include finding records with a specified attribute value and finding records whose attribute values fall in a specified range.

- **Access time**: The time it takes to find a particular data item, or set of items, using the technique in question.

- **Insertion time**: The time it takes to insert a new data item. This value includes the time it takes to find the correct place to insert the new data item, as well as the time it takes to update the index structure.

- **Deletion time**: The time it takes to delete a data item. This value includes the time it takes to find the item to be deleted, as well as the time it takes to update the index structure.

- **Space overhead**: The additional space occupied by an index structure. Pro-vided that the amount of additional space is moderate, it is usually worth-while to sacrifice the space to achieve improved performance.

We often want to have more than one index for a file. For example, we may wish to search for a book by author, by subject, or by title.

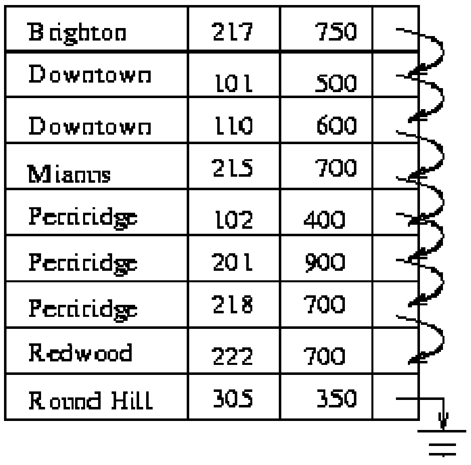
An attribute or set of attributes used to look up records in a file is called a **search key**. Note that this definition of *key* differs from that used in *primary key*, *candidate key*, and *superkey*. This duplicate meaning for *key* is (unfortunately) well established in practice. Using our notion of a search key, we see that if there are several indices on a file, there are several search keys.

Ordered Indices

1. In order to allow fast **random** access, an index structure may be used.
2. A file may have several indices on different search keys.
3. If the file containing the records is sequentially ordered, the index whose search key specifies the sequential order of the file is the **primary index**, or **clustering index**. Note: The search key of a primary index is usually the primary key, but it is not necessarily so.
4. Indices whose search key specifies an order different from the sequential order of the file are called the **secondary indices**, or **nonclustering indices**.

Primary Index

1. *Index-sequential files*: Files are ordered sequentially on some search key, and a primary index is associated with it.



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

Figure 11.1: Sequential file for *deposit* records.

Dense and Sparse Indices

An **index entry**, or **index record**, consists of a search-key value and pointers to one or more records with that value as their search-key value. The pointer to a record consists of the identifier of a disk block and an offset within the disk block to identify the record within the block.

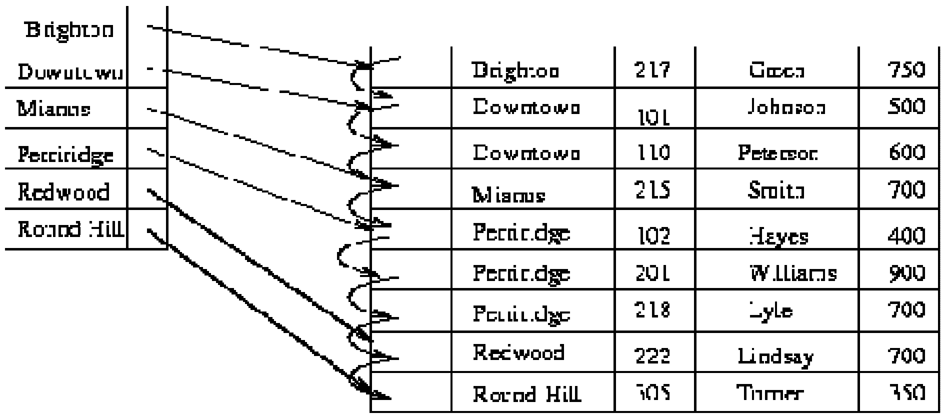
There are Two types of ordered indices:

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

In a dense nonclustering index, the index must store a list of pointers to all records with the same search-key value.

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

Figures 11.2 and 11.3 show dense and sparse indices for the deposit file.

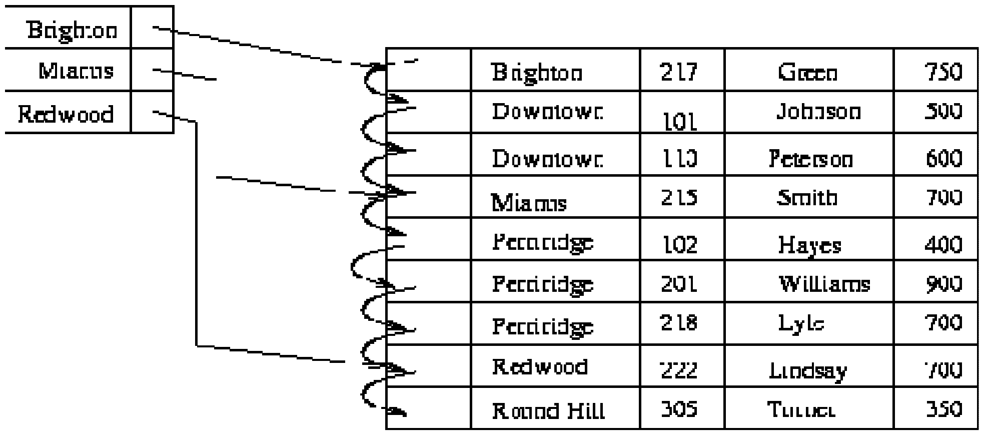


Brighton
Downtown
Mianus
Perryridge
Redwood
Round Hill

Brighton	217	Green	750
Downtown	101	Johnson	500
Downtown	110	Peterson	600
Mianus	215	Smith	700
Perridge	102	Hayes	400
Perridge	201	Williams	900
Perridge	218	Lyle	700
Redwood	222	Lindsay	700
Round Hill	305	Turner	350

Figure 11.2: Dense index.

Notice how we would find records for Perryridge branch using both methods. (Do it!)



Brighton
Mianus
Redwood

Brighton	217	Green	750
Downtown	101	Johnson	500
Downtown	110	Peterson	600
Mianus	215	Smith	700
Perryridge	102	Hayes	400
Perryridge	201	Williams	900
Perryridge	218	Lyle	700
Redwood	222	Lindsay	700
Round Hill	305	Turner	350

Figure 11.3: Sparse index.

1. Dense indices are faster in general, but sparse indices require less space and impose less maintenance for insertions and deletions. (Why?)
2. A good compromise: to have a sparse index with one entry per block.

Why is this good?

- Biggest cost is in bringing a block into main memory.
- We are guaranteed to have the correct block with this method, unless record is on an overflow block (actually could be **several** blocks).
- Index size still small.

Multi-Level Indices

Even with a sparse index, index size may still grow too large. For 100,000 records, 10 per block, at one index record per block, that's 10,000 index records! Even if we can fit 100 index records per block, this is 100 blocks.

If index is too large to be kept in main memory, a search results in several disk reads.

- If there are no overflow blocks in the index, we can use binary search.
- This will read as many as $1 + \log_2(b)$ blocks (as many as 7 for our 100 blocks).
- If index has overflow blocks, then sequential search typically used, reading **all** *b* index blocks.

Solution: Construct a sparse index on the index (Figure 11.4).

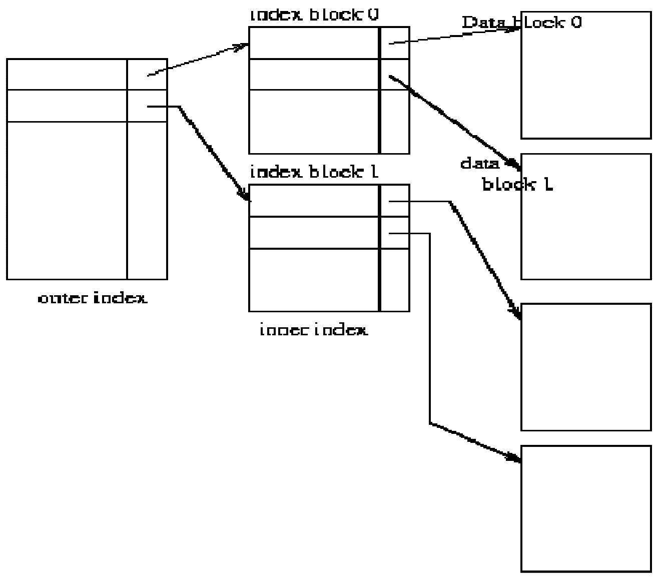


Figure 11.4: Two-level sparse index.

Use binary search on outer index. Scan **index block** found until correct index record found. Use index record as before - scan block pointed to for desired record.

For very large files, additional levels of indexing may be required.

Indices must be updated at all levels when insertions or deletions require it.

Frequently, each level of index corresponds to a unit of physical storage (e.g. indices at the level of track, cylinder and disk).

Index Update

Regardless of what form of index is used, every index must be updated whenever a record is either inserted into or deleted from the file.

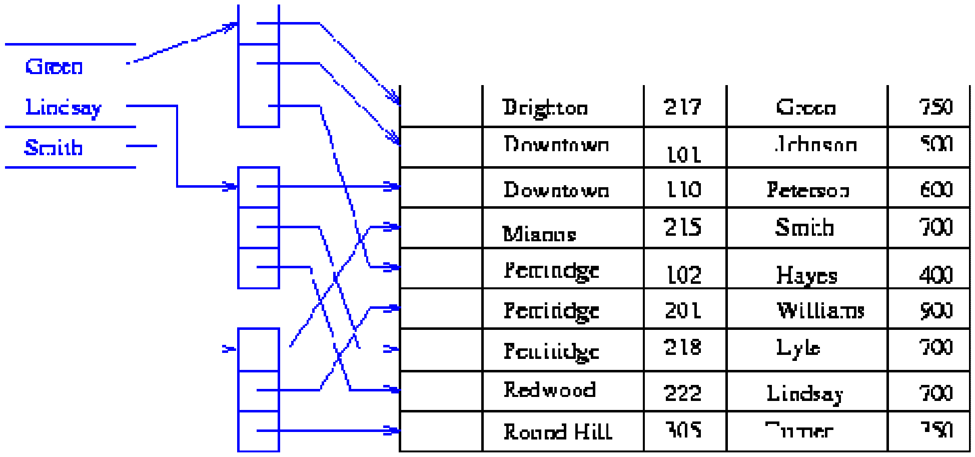
1. **Deletion:**

- Find (look up) the record
- If the last record with a particular search key value, **delete** that search key value from index.
- For dense indices, this is like deleting a record in a file.
- For sparse indices, delete a key value by replacing key value's entry in index by next search key value. If that value already has an index entry, delete the entry.

2. **Insertion:**
   - Find place to insert.
   - Dense index: insert search key value if not present.
   - Sparse index: no change unless new block is created. (In this case, the first search key value appearing in the new block is inserted into the index).

Secondary Indices

1. If the search key of a secondary index is not a candidate key, it is not enough to point to just the first record with each search-key value because the remaining records with the same search-key value could be anywhere in the file. Therefore, a secondary index must contain pointers to all the records.



Brighton	217	Green	750
Downtown	101	Johnson	500
Downtown	110	Peterson	600
Mianus	215	Smith	700
Perryridge	102	Hayes	400
Perryridge	201	Williams	900
Perryridge	218	Lyle	700
Redwood	222	Lindsay	700
Round Hill	305	Turner	350

Figure 11.5: Sparse secondary index on *cname*.

2. We can use an extra-level of indirection to implement secondary indices on search keys that are not candidate keys. A pointer does not point directly to the file but to a bucket that contains pointers to the file.
   - See Figure 11.5 on secondary key *cname*.
   - To perform a lookup on Peterson, we must read all three records pointed to by entries in bucket 2.
   - Only one entry points to a Peterson record, but three records need to be read.
   - As file is not ordered physically by *cname*, this may take 3 block accesses.
3. Secondary indices must be **dense**, with an index entry for every search-key value, and a pointer to every record in the file.
4. Secondary indices improve the performance of queries on non-primary keys.
5. They also impose serious overhead on database modification: whenever a file is updated, every index must be updated.
6. Designer must decide whether to use secondary indices or not.

B + -Tree Index Files

1. Primary disadvantage of index-sequential file organization is that performance degrades as the file grows. This can be remedied by costly re-organizations.
2. B + -tree file structure maintains its efficiency despite frequent insertions and deletions. It imposes some acceptable update and space overheads.
3. A B + -tree index is a *balanced tree* in which every path from the root to a leaf is of the same length.
4. Each nonleaf node in the tree must have between $|n/2|$ and *n* children, where *n* is fixed for a particular tree.

Structure of a B + -Tree

1. A B + -tree index is a *multilevel* index but is structured differently from that of multi-level index sequential files.
2. A typical node (Figure 11.6) contains up to *n*-1 search key values $K_1, K_2, \ldots, K_{n-1}$, and *n* pointers $P_1, P_2, \ldots, P_n$. Search key values in a node are kept in sorted order.

P_1	K_1	P_2	$\cdots\cdots$	P_{n-1}	K_{n-1}	P_n

Figure 11.6: Typical node of a B+-tree.

3. For leaf nodes, P_i ($i = 1, \ldots, n-1$) points to either a file record with search key value K_i, or a bucket of pointers to records with that search key value. Bucket structure is used if search key is not a primary key, and file is not sorted in search key order.

   Pointer P_n (*n*th pointer in the leaf node) is used to chain leaf nodes together in linear order (search key order). This allows efficient **sequential** processing of the file.

   The range of values in each **leaf** do not overlap.

4. Non-leaf nodes form a multilevel index on leaf nodes.

   A non-leaf node may hold up to *n* pointers and must hold $n/2|$ pointers. The number of pointers in a node is called the *fan-out* of the node.

   Consider a node containing *m* pointers. Pointer P_i ($i = 2, \ldots, m$) points to a subtree containing search key values $\geq K_{i-1}$ and $< K_i$. Pointer P_m points to a subtree containing search key values $\geq K_{m-1}$. Pointer P_1 points to a subtree containing search key values $< K_1$.

5. Figures 11.7 (textbook Fig. 11.8) and textbook Fig. 11.9 show B + -trees for the *deposit* file with *n*=3 and *n*=5.

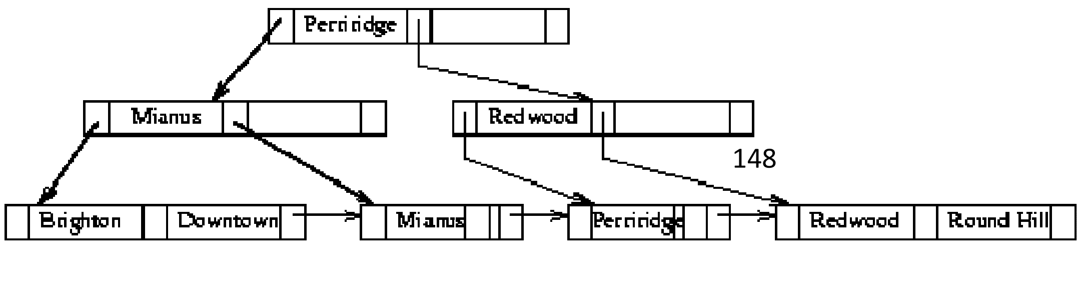


Figure 11.7: B+-tree for *deposit* file with $n = 3$.

Queries on B + -Trees

1. Suppose we want to find all records with a search key value of *k*.
   - Examine the root node and find the smallest search key value $K_i > k$.
   - Follow pointer P_i to another node.
   - If $k < K_1$ follow pointer P_1.
   - Otherwise, find the appropriate pointer to follow.
   - Continue down through non-leaf nodes, looking for smallest search key value > *k* and following the corresponding pointer.
   - Eventually we arrive at a leaf node, where pointer will point to the desired record or bucket.
2. In processing a query, we traverse a path from the root to a leaf node. If there are *K* search key values in the file, this path is no longer than $\log_{\lceil n/2 \rceil}(K)$.

   This means that the path is not long, even in large files. For a 4*k* byte disk block with a search-key size of 12 bytes and a disk pointer of 8 bytes, *n* is around 200. If *n* =100, a look-up of 1 million search-key values may take $\log_{50}(1,000,000) = 4$ nodes to be accessed. Since root is in usually in the buffer, so typically it takes only 3 or fewer disk reads.

Updates on B + -Trees

1. **Insertions and Deletions:**

   Insertion and **deletion** are more complicated, as they may require splitting or combining nodes to keep the tree balanced. If splitting or combining are not required, insertion works as follows:

   - Find leaf node where search key value should appear.
   - If value is present, add new record to the bucket.
   - If value is not present, insert value in leaf node (so that search keys are still in order).
   - Create a new bucket and insert the new record.

   If splitting or combining are not required, deletion works as follows:

- Deletion: Find record to be deleted, and remove it from the bucket.
- If bucket is now empty, remove search key value from leaf node.

2. **Insertions Causing Splitting:**

   When insertion causes a leaf node to be too large, we **split** that node. In Figure 11.8, assume we wish to insert a record with a *bname* value of ``Clearview".

   - There is no room for it in the leaf node where it should appear.
   - We now have *n* values (the *n*-1 search key values plus the new one we wish to insert).
   - We put the first $\lceil n/2 \rceil$ values in the existing node, and the remainder into a new node.
   - Figure 11.10 shows the result.
   - The new node must be inserted into the B + -tree.
   - We also need to update search key values for the parent (or higher) nodes of the split leaf node. (Except if the new node is the leftmost one)
   - Order must be preserved among the search key values in each node.
   - If the parent was already full, it will have to be split.
   - When a non-leaf node is split, the children are divided among the two new nodes.
   - In the worst case, splits may be required all the way up to the root. (If the root is split, the tree becomes one level deeper.)
   - **Note:** when we start a B + -tree, we begin with a single node that is both the root and a single leaf. When it gets full and another insertion occurs, we split it into two leaf nodes, requiring a new root.

3. **Deletions Causing Combining:**

   Deleting records may cause tree nodes to contain too few pointers. Then we must combine nodes.

   - If we wish to delete ``Downtown'' from the B + -tree of Figure 11.11, this occurs.
   - In this case, the leaf node is empty and must be deleted.
   - If we wish to delete ``Perryridge'' from the B + -tree of Figure 11.11, the parent is left with only one pointer, and must be coalesced with a sibling node.
   - Sometimes higher-level nodes must also be coalesced.
   - If the root becomes empty as a result, the tree is one level less deep (Figure 11.13).
   - Sometimes the pointers must be redistributed to keep the tree balanced.
   - Deleting ``Perryridge'' from Figure 11.11 produces Figure 11.14.

4. To summarize:
   - Insertion and deletion are complicated, but require relatively few operations.
   - Number of operations required for insertion and deletion is proportional to logarithm of number of search keys.

- B + -trees are fast as index structures for database.

B + Tree Extensions:

B + -Tree File Organization

1. The B + -tree structure is used not only as an index but also as an organizer for records into a file.
2. In a B + -tree file organization, the leaf nodes of the tree store records instead of storing pointers to records, as shown in Fig. 11.17.
3. Since records are usually larger than pointers, the maximum number of records that can be stored in a leaf node is less than the maximum number of pointers in a nonleaf node.
4. However, the leaf node are still required to be at least half full.
5. Insertion and deletion from a B + -tree file organization are handled in the same way as that in a B + -tree index.
6. When a B + -tree is used for file organization, space utilization is particularly important. We can improve the space utilization by involving more sibling nodes in redistribution during splits and merges.
7. In general, if *m* nodes are involved in redistribution, each node can be guaranteed to contain at least $\lceil (m-1)n/m \rceil$ entries. However, the cost of update becomes higher as more siblings are involved in redistribution.

B-Tree Index Files

1. B-tree indices are similar to B + -tree indices.
   - Difference is that B-tree eliminates the redundant storage of search key values.
   - In B + -tree of Figure 11.11, some search key values appear twice.
   - A corresponding B-tree of Figure 11.18 allows search key values to appear only once.
   - Thus we can store the index in less space.

P_1	K_1	P_2	$\cdots\cdots P_{n-1}$	K_{n-1}	P_n

(a)

P_1	B_1	K_1	P_2	B_2	K_2	$\cdots\cdots P_{n-1}$	B_{n-1}	K_{n-1}	P_n

(b)

Figure 11.8: Leaf and nonleaf node of a B-tree.

2. **Advantages:**
   - Lack of redundant storage (but only marginally different).
   - Some searches are faster (key may be in non-leaf node).
3. **Disadvantages:**
   - Leaf and non-leaf nodes are of different size (complicates storage)

- Deletion may occur in a non-leaf node (more complicated)

Generally, the structural simplicity of B + -tree is preferred.

Indexing Strings

Creating B^+-tree indices on string-valued attributes raises two problems. The first problem is that strings can be of variable length. The second problem is that strings can be long, leading to a low fanout and a correspondingly increased tree height.

With variable-length search keys, different nodes can have different fanouts even if they are full. A node must then be split if it is full, that is, there is no space to add a new entry, regardless of how many search entries it has. Similarly, nodes can be merged or entries redistributed depending on what fraction of the space in the nodes is used, instead of being based on the maximum number of entries that the node can hold.

The fanout of nodes can be increased by using a technique called **prefix compression**. With prefix compression, we do not store the entire search key value at nonleaf nodes. We only store a prefix of each search key value that is sufficient to distinguish between the key values in the subtrees that it separates. For example, if we had an index on names, the key value at a nonleaf node could be a prefix of a name; it may suffice to store "Silb" at a nonleaf node, instead of the full "Silberschatz" if the closest values in the two subtrees that it separates are, say, "Silas" and "Silver" respectively.

Multiple-Key Access

For some queries, it is advantageous to use multiple indices if they exist.

If there are two indices on *deposit*, one on *bname* and one on *cname*, then suppose we have a query like

```
select balance from deposit
where bname = ``Perryridge'' and balance = 1000
```

There are 3 possible strategies to process this query:

- Use the index on *bname* to find all records pertaining to Perryridge branch. Examine them to see if *balance* = 1000
- Use the index on *balance* to find all records pertaining to Williams. Examine them to see if *bname* = ``Perryridge".
- Use index on *bname* to find pointers to records pertaining to Perryridge branch. Use index on *balance* to find pointers to records pertaining to 1000. Take the **intersection** of these two sets of pointers.

The third strategy takes advantage of the existence of multiple indices. This may still not work well if

- There are a large number of Perryridge records **AND**
- There are a large number of 1000 records **AND**
- Only a small number of records pertain to both Perryridge and 1000.

To speed up multiple search key queries special structures can be maintained.

Static Hashing

1. Index schemes force us to traverse an index structure. Hashing avoids this.

Hash File Organization

1. **Hashing** involves computing the address of a data item by computing a function on the search key value.
2. A **hash function h** is a function from the set of all search key values *K* to the set of all bucket addresses *B*.
    - We choose a number of buckets to correspond to the number of search key values we will have stored in the database.
    - To perform a lookup on a search key value K_i, we compute $h(K_i)$, and search the bucket with that address.
    - If two search keys *i* and *j* map to the same address, because $h(K_i) = h(K_j)$, then the bucket at the address obtained will contain records with both search key values.
    - In this case we will have to check the search key value of every record in the bucket to get the ones we want.
    - Insertion and deletion are simple.

Hash Functions

1. A good hash function gives an average-case lookup that is a small constant, independent of the number of search keys.
2. We hope records are distributed uniformly among the buckets.
3. The worst hash function maps all keys to the same bucket.
4. The best hash function maps all keys to distinct addresses.
5. Ideally, distribution of keys to addresses is uniform and random.
6. Suppose we have 26 buckets, and map names beginning with *i*th letter of the alphabet to the *i*th bucket.
    - Problem: this does not give uniform distribution.
    - Many more names will be mapped to ``A'' than to ``X''.

- Typical hash functions perform some operation on the internal binary machine representations of characters in a key.
- For example, compute the sum, modulo # of buckets, of the binary representations of characters of the search key.
- See Figure 11.18, using this method for 10 buckets (assuming the *i*th character in the alphabet is represented by integer *i*).

Handling of bucket overflows

1. **Open** hashing occurs where records are stored in different buckets. Compute the hash function and search the corresponding bucket to find a record.
2. **Closed** hashing occurs where all records are stored in **one** bucket. Hash function computes addresses within that bucket. (Deletions are difficult.) Not used much in database applications.
3. **Drawback to our approach:** Hash function must be chosen at implementation time.
    - Number of buckets is fixed, but the database may grow.
    - If number is too large, we waste space.
    - If number is too small, we get too many ``collisions'', resulting in records of many search key values being in the same bucket.
    - Choosing the number to be twice the number of search key values in the file gives a good space/performance tradeoff.

Hash Indices

1. A hash index organizes the search keys with their associated pointers into a hash file structure.
2. We apply a hash function on a search key to identify a bucket, and store the key and its associated pointers in the bucket (or in overflow buckets).
3. Strictly speaking, hash indices are only secondary index structures, since if a file itself is organized using hashing, there is no need for a separate hash index structure on it.

Dynamic Hashing

1. As the database grows over time, we have three options:
    - Choose hash function based on current file size. Get performance degradation as file grows.
    - Choose hash function based on anticipated file size. Space is wasted initially.
    - Periodically re-organize hash structure as file grows. Requires selecting new hash function, recomputing all addresses and generating new bucket assignments. Costly, and shuts down database.
2. Some hashing techniques allow the hash function to be modified dynamically to accommodate the growth or shrinking of the database. These are called **dynamic hash functions**.

- **Extendable hashing** is one form of dynamic hashing.
- Extendable hashing splits and coalesces buckets as database size changes.
- This imposes some performance overhead, but space efficiency is maintained.
- As reorganization is on one bucket at a time, overhead is acceptably low.

3. How does it work?

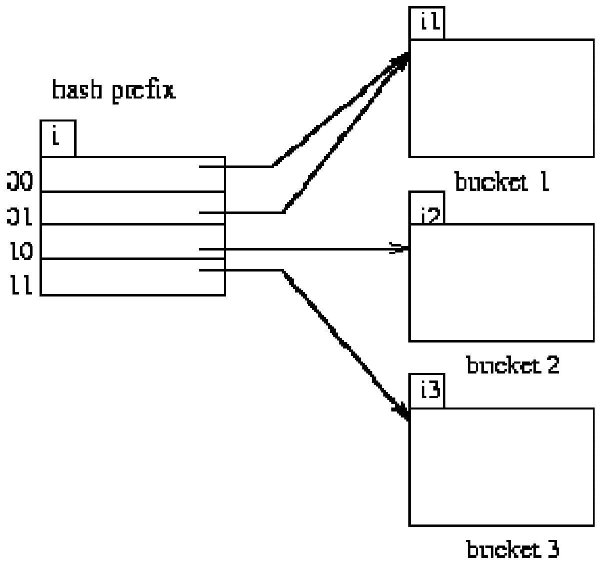


Figure 11.9: General extendable hash structure.

- We choose a hash function that is uniform and random that generates values over a relatively large range.
- Range is *b*-bit binary integers (typically b=32).
- 2^{32} is over 4 billion, so we don't generate that many buckets!
- Instead we create buckets on demand, and do not use all *b* bits of the hash initially.
- At any point we use *i* bits where $0 \leq i \leq b$.
- The *i* bits are used as an offset into a table of bucket addresses.
- Value of *i* grows and shrinks with the database.
- Figure 11.19 shows an extendable hash structure.
- Note that the *i* appearing over the bucket address table tells how many bits are required to determine the correct bucket.
- It may be the case that several entries point to the same bucket.
- All such entries will have a common hash prefix, but the length of this prefix may be less than *i*.
- So we give each bucket an integer giving the length of the common hash prefix.
- This is shown in Figure 11.9 (textbook 11.19) as i_j.
- Number of bucket entries pointing to bucket *j* is then $2^{(i-i_j)}$.

4. To find the bucket containing search key value K_l:
   - Compute $h(K_l)$.
   - Take the first *i* high order bits of $h(K_l)$.
   - Look at the corresponding table entry for this *i*-bit string.

- Follow the bucket pointer in the table entry.

5. We now look at insertions in an extendable hashing scheme.
   - Follow the same procedure for lookup, ending up in some bucket *j*.
   - If there is room in the bucket, insert information and insert record in the file.
   - If the bucket is full, we must split the bucket, and redistribute the records.
   - If bucket is split we may need to increase the number of bits we use in the hash.
6. Two cases exist:

   1. If $i = i_j$, then only one entry in the bucket address table points to bucket *j*.

   - Then we need to increase the size of the bucket address table so that we can include pointers to the two buckets that result from splitting bucket *j*.
   - We increment *i* by one, thus considering more of the hash, and doubling the size of the bucket address table.
   - Each entry is replaced by two entries, each containing original value.
   - Now two entries in bucket address table point to bucket j.
   - We allocate a new bucket *z*, and set the second pointer to point to *z*.
   - Set i_j and i_z to *i*.
   - Rehash all records in bucket *j* which are put in either *j* or *z*.
   - Now insert new record.
   - It is remotely possible, but unlikely, that the new hash will still put all of the records in one bucket.
   - If so, split again and increment *i* again.

   2. If $i > i_j$, then more than one entry in the bucket address table points to bucket *j*.

   - Then we can split bucket *j* without increasing the size of the bucket address table (why?).
   - Note that all entries that point to bucket *j* correspond to hash prefixes that have the same value on the leftmost i_j bits.
   - We allocate a new bucket *z*, and set i_j and i_z to the original i_j value plus 1.
   - Now adjust entries in the bucket address table that previously pointed to bucket *j*.
   - Leave the first half pointing to bucket *j*, and make the rest point to bucket *z*.
   - Rehash each record in bucket *j* as before.
   - Reattempt new insert.
7. Note that in both cases we only need to rehash records in bucket *j*.
8. Deletion of records is similar. Buckets may have to be coalesced, and bucket address table may have to be halved.
9. Insertion is illustrated for the example *deposit* file of Figure 11.20.
   - 32-bit hash values on *bname* are shown in Figure 11.21.
   - An initial empty hash structure is shown in Figure 11.22.
   - We insert records one by one.
   - We (unrealistically) assume that a bucket can only hold 2 records, in order to illustrate both situations described.

- As we insert the Perryridge and Round Hill records, this first bucket becomes full.
- When we insert the next record (Downtown), we must split the bucket.
- Since $i = i_0$, we need to increase the number of bits we use from the hash.
- We now use 1 bit, allowing us $2^1 = 2$ buckets.
- This makes us double the size of the bucket address table to two entries.
- We split the bucket, placing the records whose search key hash begins with 1 in the new bucket, and those with a 0 in the old bucket (Figure 11.23).
- Next we attempt to insert the Redwood record, and find it hashes to 1.
- That bucket is full, and $i = i_1$.
- So we must split that bucket, increasing the number of bits we must use to 2.
- This necessitates doubling the bucket address table again to four entries (Figure 11.24).
- We rehash the entries in the old bucket.
- We continue on for the deposit records of Figure 11.20, obtaining the extendable hash structure of Figure 11.25.

Advantages:

- Extendable hashing provides performance that does not degrade as the file grows.
- Minimal space overhead - no buckets need be reserved for future use. Bucket address table only contains one pointer for each hash value of current prefix length.

Disadvantages:

- Extra level of indirection in the bucket address table
- Added complexity

Summary: A highly attractive technique, provided we accept added complexity.

Static Hashing versus Dynamic Hashing

We now examine the advantages and disadvantages of extendable hashing, com-pared with static hashing. The main advantage of extendable hashing is that performance does not degrade as the file grows. Furthermore, there is minimal space overhead. Although the bucket address table incurs additional overhead, it contains one pointer for each hash value for the current prefix length. This table is thus small. The main space saving of extendable hashing over other forms of hashing is that no buckets need to be reserved for future growth; rather, buckets can be allocated dynamically.

A disadvantage of extendable hashing is that lookup involves an additional level of indirection, since the system must access the bucket address table before accessing the bucket itself. This extra reference has only a minor effect on per-formance.

Thus, extendable hashing appears to be a highly attractive technique, pro-vided that we are willing to accept the added complexity involved in its im-plementation. The bibliographical notes reference more detailed descriptions of extendable hashing implementation.

The bibliographical notes also provide references to another form of dynamic hashing called **linear hashing**, which avoids the extra level of indirection associ-ated with extendable hashing, at the possible cost of more overflow buckets.

Comparison of Ordered Indexing and Hashing

We have seen several ordered-indexing schemes and several hashing schemes. We can organize files of records as ordered files by using index-sequential organi-zation or B^+-tree organizations. Alternatively, we can organize the files by using hashing. Finally, we can organize them as heap files, where the records are not ordered in any particular way.

Each scheme has advantages in certain situations. A database-system imple-mentor could provide many schemes, leaving the final decision of which schemes to use to the database designer. However, such an approach requires the imple-mentor to write more code, adding both to the cost of the system and to the space that the system occupies. Most database systems support B^+-trees and may additionally support some form of hash file organization or hash indices.

To make a choice of file organization and indexing techniques for a relation, the implementor or the database designer must consider the following issues:

Is the cost of periodic reorganization of the index or hash organization acceptable?
What is the relative frequency of insertion and deletion?

Is it desirable to optimize average access time at the expense of increasing the worst-case access time?

What types of queries are users likely to pose?

We have already examined the first three of these issues, first in our review of the relative merits of specific indexing techniques, and again in our discussion of hashing techniques. The fourth issue, the expected type of query, is critical to the choice of ordered indexing or hashing.

If most queries are of the form:

select $A_1, A_2, \ldots, A_n$ **from** r **where** $A_i = c$;

then, to process this query, the system will perform a lookup on an ordered index or a hash structure for attribute A_i , for value c. For queries of this form, a hashing scheme is preferable. An ordered-index lookup requires time proportional to the log of the number of values in r for A_i . In a hash structure, however, the average lookup time is a constant independent of the size of the database. The only advantage to an index over a hash structure for this form of query is that the worst-case lookup time is proportional to the log of the number of values in r for A_i . By contrast, for hashing, the worst-case lookup time is proportional to the number of values in r for A_i . However, the worst-case lookup time is unlikely to occur with hashing, and hashing is preferable in this case.

Ordered-index techniques are preferable to hashing in cases where the query specifies a range of values. Such a query takes the following form:

select $A_1, A_2, ..., A_n$ **from** r **where** $A_i \le c_2$ **and** $A_i \ge c_1$;

In other words, the preceding query finds all the records with A_i values between c_1 and c_2.

Let us consider how we process this query using an ordered index. First, we perform a lookup on value c_1. Once we have found the bucket for value c_1, we follow the pointer chain in the index to read the next bucket in order, and we continue in this manner until we reach c_2.

If, instead of an ordered index, we have a hash structure, we can perform a lookup on c_1 and can locate the corresponding bucket—but it is not easy, in general, to determine the next bucket that must be examined. The difficulty arises because a good hash function assigns values randomly to buckets. Thus, there is no simple notion of "next bucket in sorted order." The reason we cannot chain buckets together in sorted order on A_i is that each bucket is assigned many search-key values. Since values are scattered randomly by the hash function, the values in the specified range are likely to be scattered across many or all of the buckets. Therefore, we have to read all the buckets to find the required search keys.

Usually the designer will choose ordered indexing unless it is known in advance that range queries will be infrequent, in which case hashing would be chosen. Hash organizations are particularly useful for temporary files created during query processing, if lookups based on a key value are required, but no range queries will be performed.

Bitmap Indices

Bitmap indices are a specialized type of index designed for easy querying on multiple keys, although each bitmap index is built on a single key.

For bitmap indices to be used, records in a relation must be numbered sequentially, starting from, say, 0. Given a number *n*, it must be easy to retrieve the record numbered *n*. This is particularly easy to achieve if records are fixed in size, and allocated on consecutive blocks of a file. The record number can then be translated easily into a block number and a number that identifies the record within the block.

Consider a relation *r* , with an attribute *A* that can take on only one of a small number (for example, 2 to 20) values. For instance, a relation *instructor info* may have an attribute *gender*, which can take only values m (male) or f (female). Another example would be an attribute *income level*, where income has been broken up into 5 levels: *L*1: \$0−9999, *L*2: \$10,000−19,999, *L*3: 20,000−39,999, *L*4: 40,000−74,999, and *L*5: 75,000−∞. Here, the raw data can take on many values, but a data analyst has split the values into a small number of ranges to simplify analysis of the data.

Bitmap Index Structure

A **bitmap** is simply an array of bits. In its simplest form, a **bitmap index** on the attribute *A* of relation *r* consists of one bitmap for each value that *A* can take. Each bitmap has as many bits as the number of records in the relation. The *i*th bit of the bitmap for value v_j is set to 1 if the record numbered *i* has the value v_j for attribute *A*. All other bits of the bitmap are set to 0.

In our example, there is one bitmap for the value m and one for f. The *i*th bit of the bitmap for m is set to 1 if the *gender* value of the record numbered *i* is m. All other bits of the bitmap for m are set to 0. Similarly, the bitmap for f has the value 1 for bits corresponding to records with the value f for the *gender* attribute; all other bits have the value 0. Figure 11.35 shows an example of bitmap indices on a relation *instructor info*.

We now consider when bitmaps are useful. The simplest way of retrieving all records with value m (or value f) would be to simply read all records of the relation and select those records with value m (or f, respectively). The bitmap index doesn't really help to speed up such a selection. While it would allow us to read only those records for a specific gender, it is likely that every disk block for the file would have to be read anyway.

In fact, bitmap indices are useful for selections mainly when there are selec-tions on multiple keys. Suppose we create a bitmap index on attribute *income level*, which we described earlier, in addition to the bitmap index on *gender*.

Consider now a query that selects women with income in the range \$10,000 *to* \$19, 999. This query can be expressed as

```
select *
from r
where gender = 'f' and income level = 'L2';
```

To evaluate this selection, we fetch the bitmaps for *gender* value f and the bitmap for *income level* value *L*2, and perform an **intersection** (logical-and) of the two bitmaps. In other words, we compute a new bitmap where bit *i* has value 1 if the *i*th bit of the two bitmaps are both 1, and has a value 0 otherwise. In the example in Figure 11.35, the intersection of the bitmap for *gender* = f (01101) and the bitmap for *i ncome level* = *L*2 (01000) gives the bitmap 01000.

Bitmaps and B$^+$-Trees

Bitmaps can be combined with regular B$^+$-tree indices for relations where a few attribute values are extremely common, and other values also occur, but much less frequently. In a B$^+$-tree index leaf, for each value we would normally maintain a list of all records with that value for the indexed attribute. Each element of the list would be a record identifier, consisting of at least 32 bits, and usually more. For a value that occurs in many records, we store a bitmap instead of a list of records.

Suppose a particular value v_i occurs in $\frac{1}{16}$ of the records of a relation. Let N be the number of records in the relation, and assume that a record has a 64-bit number identifying it. The bitmap needs only 1 bit per record, or N bits in total. In contrast, the list representation requires 64 bits per record where the value occurs, or $64 * N/16 = 4N$ bits. Thus, a bitmap is preferable for representing the list of records for value v_i. In our example (with a 64-bit record identifier), if fewer than 1 in 64 records have a particular value, the list representation is preferable for identifying records with that value, since it uses fewer bits than the bitmap representation. If more than 1 in 64 records have that value, the bitmap representation is preferable.

Thus, bitmaps can be used as a compressed storage mechanism at the leaf nodes of B$^+$-trees for those values that occur very frequently.

Index Definition in SQL

The SQL standard does not provide any way for the database user or administra-tor to control what indices are created and maintained in the database system. Indices are not required for correctness, since they are redundant data structures. However, indices are important for efficient processing of transactions, includ-ing both update transactions and queries. Indices are also important for efficient enforcement of integrity constraints.

In principle, a database system can decide automatically what indices to create. However, because of the space cost of indices, as well as the effect of indices on update processing, it is not easy to automatically make the right choices about what indices to maintain. Therefore, most SQL implementations provide the programmer control over creation and removal of indices via data-definition-language commands.

We illustrate the syntax of these commands next. Although the syntax that we show is widely used and supported by many database systems, it is not part of the SQL standard. The SQL standard does not support control of the physical database schema; it restricts itself to the logical database schema.

We create an index with the **create index** command, which takes the form:

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

The *attribute-list* is the list of attributes of the relations that form the search key for the index.

To define an index named *dept index* on the *instructor* relation with *dept name* as the search key, we write:

create index *dept index* **on** *instructor* (*dept name*);

If we wish to declare that the search key is a candidate key, we add the attribute **unique** to the index definition. Thus, the command:

create unique index *dept index* **on** *instructor* (*dept name*);

declares *dept name* to be a candidate key for *instructor* (which is probably not what we actually would want for our university database). If, at the time we enter the **create unique index** command, *dept name* is not a candidate key, the system will display an error message, and the attempt to create the index will fail. If the index-creation attempt succeeds, any subsequent attempt to insert a tuple that violates the key declaration will fail. Note that the **unique** feature is redundant if the database system supports the **unique** declaration of the SQL standard.

Many database systems also provide a way to specify the type of index to be used (such as B^+-tree or hashing). Some database systems also permit one of the indices on a relation to be declared to be clustered; the system then stores the relation sorted by the search-key of the clustered index.

The index name we specified for an index is required to drop an index. The **drop index** command takes the form:

drop index <index-name>;

Query Processing

Query processing refers to the range of activities involved in extracting data from a database. The activities include translation of queries in high-level database languages into expressions that can be used at the physical level of the file system, a variety of query-optimizing transformations, and actual evaluation of queries.

Overview

The steps involved in processing a query appear in Figure 12.1. The basic steps are:

Parsing and translation.
Optimization.
Evaluation.

Before query processing can begin, the system must translate the query into a usable form. A language such as SQL is suitable for human use, but is ill suited to be the system's internal representation of a query. A more useful internal representation is one based on the extended relational algebra.

Thus, the first action the system must take in query processing is to translate a given query into its internal form. This translation process is similar to the work performed by the parser of a compiler. In generating the internal form of the query, the parser checks the syntax of the user's query, verifies that the relation names appearing in the query are names of the relations in the database, and so on. The system constructs a parse-tree representation of the query, which it then translates into a relational-algebra expression. If the query was expressed in terms of a view, the translation phase also replaces all uses of the view by the relational-algebra expression that defines the view

Measures of Query Cost

There are multiple possible evaluation plans for a query, and it is important to be able to compare the alternatives in terms of their (estimated) cost, and choose the best plan. To do so, we must estimate the cost of individual operations, and combine them to get the cost of a query evaluation plan. Thus, as we study evaluation algorithms for each operation later in this chapter, we also outline how to estimate the cost of the operation.

The cost of query evaluation can be measured in terms of a number of dif-ferent resources, including disk accesses, CPU time to execute a query, and, in a distributed or parallel database system, the cost of communication (which we discuss later, in Chapters 18 and 19).

In large database systems, the cost to access data from disk is usually the most important cost, since disk accesses are slow compared to in-memory operations. Moreover, CPU speeds have been improving much faster than have disk speeds. Thus, it is likely that the time spent in disk activity will continue to dominate the total time to execute a query. The CPU time taken for a task is harder to estimate since it depends on low-level details of the execution code. Although real-life query optimizers do take CPU costs into account, for simplicity in this book we ignore CPU costs and use only disk-access costs to measure the cost of a query-evaluation plan.

We use the *number of block transfers* from disk and the *number of disk seeks* to estimate the cost of a query-evaluation plan. If the disk subsystem takes an average of t_T seconds to transfer a block of data, and has an average block-access time (disk seek time plus rotational latency) of t_S seconds, then an operation that transfers b blocks and performs S seeks would take $b * t_T + S * t_S$ seconds. The values of t_T and t_S must be calibrated for the disk system used, but typical values for high-end disks today would be $t_S = 4$

milliseconds and $t_T = 0.1$ milliseconds, assuming a 4-kilobyte block size and a transfer rate of 40 megabytes per second.[2]

We can refine our cost estimates further by distinguishing block reads from block writes, since block writes are typically about twice as expensive as reads (this is because disk systems read sectors back after they are written to verify that the write was successful). For simplicity, we ignore this detail, and leave it to you to work out more precise cost estimates for various operations.

The cost estimates we give do not include the cost of writing the final result of an operation back to disk. These are taken into account separately where required. The costs of all the algorithms that we consider depend on the size of the buffer in main memory. In the best case, all data can be read into the buffers, and the disk does not need to be accessed again. In the worst case, we assume that the buffer can hold only a few blocks of data — approximately one block per relation. When presenting cost estimates, we generally assume the worst case.

In addition, although we assume that data must be read from disk initially, it is possible that a block that is accessed is already present in the in-memory buffer. Again, for simplicity, we ignore this effect; as a result, the actual disk-access cost during the execution of a plan may be less than the estimated cost.

The **response time** for a query-evaluation plan (that is, the wall-clock time required to execute the plan), assuming no other activity is going on in the computer, would account for all these costs, and could be used as a measure of the cost of the plan. Unfortunately, the response time of a plan is very hard to estimate without actually executing the plan, for the following reasons:

The response time depends on the contents of the buffer when the query begins execution; this information is not available when the query is opti-mized, and is hard to account for even if it were available.

In a system with multiple disks, the response time depends on how accesses are distributed among disks, which is hard to estimate without detailed knowledge of data layout on disk.

Interestingly, a plan may get a better response time at the cost of extra resource consumption. For example, if a system has multiple disks, a plan *A* that requires extra disk reads, but performs the reads in parallel across multiple disks may finish faster than another plan *B* that has fewer disk reads, but from only one disk. However, if many instances of a query using plan *A* run concurrently, the overall response time may actually be more than if the same instances are executed using plan *B*, since plan *A* generates more load on the disks.

As a result, instead of trying to minimize the response time, optimizers gen-erally try to minimize the total **resource consumption** of a query plan. Our model of estimating the total disk access time (including seek and data transfer) is an example of such a resource consumption–based model of query cost.

Selection Operation

In query processing, the **file scan** is the lowest-level operator to access data. File scans are search algorithms that locate and retrieve records that fulfill a selection condition. In relational systems, a file scan allows an entire relation to be read in those cases where the relation is stored in a single, dedicated file.

Selections Using File Scans and Indices

Consider a selection operation on a relation whose tuples are stored together in one file. The most straightforward way of performing a selection is as follows:

A1 (**linear search**). In a linear search, the system scans each file block and tests all records to see whether they satisfy the selection condition. An initial seek is required to access the first block of the file. In case blocks of the file are not stored contiguously, extra seeks may be required, but we ignore this effect for simplicity.

Although it may be slower than other algorithms for implementing selec-tion, the linear-search algorithm can be applied to any file, regardless of the ordering of the file, or the availability of indices, or the nature of the selection operation. The other algorithms that we shall study are not applicable in all cases, but when applicable they are generally faster than linear search.

Cost estimates for linear scan, as well as for other selection algorithms, are shown in Figure 12.3. In the figure, we use h_i to represent the height of the B$^+$-tree. Real-life optimizers usually assume that the root of the tree is present in the in-memory buffer since it is frequently accessed. Some optimizers even assume that all but the leaf level of the tree is present in memory, since they are accessed relatively frequently, and usually less than 1 percent of the nodes of a B$^+$-tree are nonleaf nodes. The cost formulae can be modified appropriately.

Index structures are referred to as **access paths**, since they provide a path through which data can be located and accessed. In Chapter 11, we pointed out that it is efficient to read the records of a file in an order corresponding closely to physical order. Recall that a *primary index* (also referred to as a *clustering index*) is an index that allows the records of a file to be read in an order that corresponds to the physical order in the file. An index that is not a primary index is called a *secondary index*.

Search algorithms that use an index are referred to as **index scans**. We use the selection predicate to guide us in the choice of the index to use in processing the query. Search algorithms that use an index are:

A2 (**primary index, equality on key**). For an equality comparison on a key attribute with a primary index, we can use the index to retrieve a single record that satisfies the corresponding equality condition. Cost estimates are shown in Figure 12.3.

A3 (**primary index, equality on nonkey**). We can retrieve multiple records by using a primary index when the selection condition specifies an equality comparison on a nonkey attribute, *A*. The only difference from the previous case is that multiple records may need to be fetched. However, the records must be stored consecutively in the file since the file is sorted on the search key. Cost estimates are shown in Figure 12.3.

A4 (**secondary index, equality**). Selections specifying an equality condition can use a secondary index. This strategy can retrieve a single record if the equality condition is on a key; multiple records may be retrieved if the index-ing field is not a key.

In the first case, only one record is retrieved. The time cost in this case is the same as that for a primary index (case A2).

In the second case, each record may be resident on a different block, which may result in one I/O operation per retrieved record, with each I/O operation requiring a seek and a block transfer. The worst-case time cost in this case is

	Algorithm	Cost	Reason
A1	Linear Search	$t_S + b_r * t_T$	One initial seek plus b_r block transfers, where b_r denotes the number of blocks in the file.
A1	Linear Search, Equality on Key	Average case $t_S + (b_r/2) * t_T$	Since at most one record satisfies con-dition, scan can be terminated as soon as the required record is found. In the worst case, b_r blocks transfers are still required.
A2	Primary B$^+$-tree Index, Equality on Key	$(h_i + 1) * (t_T + t_S)$	(Where h_i denotes the height of the in-dex.) Index lookup traverses the height of the tree plus one I/O to fetch the record; each of these I/O operations re-quires a seek and a block transfer.
A3	Primary B$^+$-tree Index, Equality on Nonkey	$h_i * (t_T + t_S) + b * t_T$	One seek for each level of the tree, one seek for the first block. Here b is the number of blocks containing records with the specified search key, all of which are read. These blocks are leaf blocks assumed to be stored sequen-tially (since it is a primary index) and

			don't require additional seeks.
A4	Secondary B^+-tree Index, Equality on Key	$(h_i + 1) * (t_T + t_S)$	This case is similar to primary index.
A4	Secondary B^+-tree Index, Equality on Nonkey	$(h_i + n) * (t_T + t_S)$	(Where n is the number of records fetched.) Here, cost of index traversal is the same as for A3, but each record may be on a different block, requiring a seek per record. Cost is potentially very high if n is large.
A5	Primary B^+-tree Index, Comparison	$h_i * (t_T + t_S) + b * t_T$	Identical to the case of A3, equality on nonkey.
A6	Secondary B^+-tree Index, Comparison	$(h_i + n) * (t_T + t_S)$	Identical to the case of A4, equality on nonkey.

Figure 12.3 Cost estimates for selection algorithms.

$(h_i + n) * (t_S + t_T)$, where n is the number of records fetched, if each record is in a different disk block, and the block fetches are randomly ordered. The worst-case cost could become even worse than that of linear search if a large number of records are retrieved.

If the in-memory buffer is large, the block containing the record may already be in the buffer. It is possible to construct an estimate of the *average* or *expected* cost of the selection by taking into account the probability of the block containing the record already being in the buffer. For large buffers, that estimate will be much less than the worst-case estimate.

In certain algorithms, including A2, the use of a B^+-tree file organization can save one access since records are stored at the leaf-level of the tree.

As described in Section 11.4.2, when records are stored in a B^+-tree file organi-zation or other file organizations that may require relocation of records, secondary indices usually do not store pointers to the records.[3] Instead, secondary indices store the values of the attributes used as the search key in a B^+-tree file organiza-tion. Accessing a record through such a secondary index is then more expensive: First the secondary index is searched to find the primary index search-key val-ues, then the primary index is looked up to find the records. The cost formulae described for secondary indices have to be modified appropriately if such indices are used.

Sorting

Sorting of data plays an important role in database systems for two reasons. First, SQL queries can specify that the output be sorted. Second, and equally important for query processing, several of the relational operations, such as joins, can be implemented efficiently if the input relations are first sorted.

We can sort a relation by building an index on the sort key, and then using that index to read the relation in sorted order. However, such a process orders the relation only *logically*, through an index, rather than *physically*. Hence, the reading of tuples in the sorted order may lead to a disk access (disk seek plus

block transfer) for each record, which can be very expensive, since the number of records can be much larger than the number of blocks. For this reason, it may be desirable to order the records physically.

The problem of sorting has been studied extensively, both for relations that fit entirely in main memory and for relations that are bigger than memory. In the first case, standard sorting techniques such as quick-sort can be used. Here, we discuss how to handle the second case.

External Sort-Merge Algorithm

Sorting of relations that do not fit in memory is called **external sorting**. The most commonly used technique for external sorting is the **external sort – merge** algorithm. We describe the external sort – merge algorithm next. Let *M* denote the number of blocks in the main-memory buffer available for sorting, that is, the number of disk blocks whose contents can be buffered in available main memory.

In the first stage, a number of sorted **runs** are created; each run is sorted, but contains only some of the records of the relation.

```
$i = 0$;
repeat
    read $M$ blocks of the relation, or the rest of the relation, whichever is smaller;
    sort the in-memory part of the relation;
    write the sorted data to run file $R_i$ ;
    $i = i + 1$;
until the end of the relation
```

In the second stage, the runs are *merged*. Suppose, for now, that the total number of runs, *N*, is less than *M*, so that we can allocate one block to each run and have space left to hold one block of output. The merge stage operates as follows:

```
read one block of each of the $N$ files $R_i$ into a buffer block in memory; repeat

    choose the first tuple (in sort order) among all buffer blocks; write the tuple to the
    output, and delete it from the buffer block; if the buffer block of any run $R_i$ is empty and
    not end-of-file($R_i$ )
```

then read the next block of R_i into the buffer block; **until** all input buffer blocks are empty

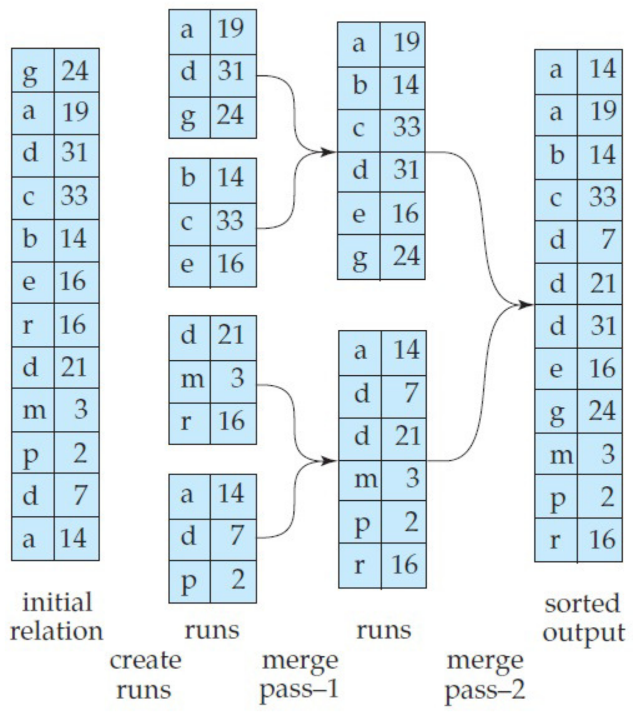

g 24, a 19, d 31, c 33, b 14, e 16, r 16, d 21, m 3, p 2, d 7, a 14
initial relation
create runs
a 19, d 31, g 24
b 14, c 33, e 16
d 21, m 3, r 16
a 14, d 7, p 2
runs
merge pass–1
a 19, b 14, c 33, d 31, e 16, g 24
a 14, d 7, d 21, m 3, p 2, r 16
runs
merge pass–2
a 14, a 19, b 14, c 33, d 7, d 21, d 31, e 16, g 24, m 3, p 2, r 16
sorted output

Figure 12.4 External sorting using sort–merge.

The output of the merge stage is the sorted relation. The output file is buffered to reduce the number of disk write operations. The preceding merge operation is a generalization of the two-way merge used by the standard in-memory sort – merge algorithm; it merges N runs, so it is called an N**-way merge**.

In general, if the relation is much larger than memory, there may be M or more runs generated in the first stage, and it is not possible to allocate a block for each run during the merge stage. In this case, the merge operation proceeds in multiple passes. Since there is enough memory for $M - 1$ input buffer blocks, each merge can take $M - 1$ runs as input.

The initial *pass* functions in this way: It merges the first $M - 1$ runs (as desc-ribed in item 2 above) to get a single run for the next pass. Then, it merges the next $M - 1$ runs similarly, and so on, until it has processed all the initial runs. At this point, the number of runs has been reduced by a factor of $M - 1$. If this reduced number of runs is still greater than or equal to M, another pass is made, with the runs created by the first pass as input. Each pass reduces the number of runs by a factor of $M - 1$. The passes repeat as many times as required, until the number of runs is less than M; a final pass then generates the sorted output.

Figure 12.4 illustrates the steps of the external sort – merge for an example relation. For illustration purposes, we assume that only one tuple fits in a block ($f_r = 1$), and we assume that memory holds at most three blocks. During the merge stage, two blocks are used for input and one for output.

Join Operation

In this section, we study several algorithms for computing the join of relations, and we analyze their respective costs.

We use the term **equi-join** to refer to a join of the form $r\ {}_{r.\,A=s.\,B}\ s$, where A and B are attributes or sets of attributes of relations r and s, respectively.

We use as a running example the expression:

student *takes*

using the same relation schemas that we used in Chapter 2. We assume the following information about the two relations:

Number of records of *student*: $n_{student} = 5,000$.

Number of blocks of *student*: $b_{student} = 100$.

Number of records of *takes*: $n_{takes} = 10,000$.

Number of blocks of *takes*: $b_{takes} = 400$.

Nested-Loop Join

Figure 12.5 shows a simple algorithm to compute the theta join, r u s, of two relations r and s. This algorithm is called the **nested-loop join** algorithm, since it basically consists of a pair of nested **for** loops. Relation r is called the **outer relation** and relation s the **inner relation** of the join, since the loop for r encloses the loop for s. The algorithm uses the notation $t_r \cdot t_s$, where t_r and t_s are tuples; $t_r \cdot t_s$ denotes the tuple constructed by concatenating the attribute values of tuples

t_r and t_s .

Like the linear file-scan algorithm for selection, the nested-loop join algorithm requires no indices, and it can be used regardless of what the join condition is. Extending the algorithm to compute the natural join is straightforward, since the natural join can be expressed as a theta join followed by elimination of repeated attributes by a projection. The only change required is an extra step of deleting repeated attributes from the tuple $t_r \cdot t_s$, before adding it to the result.

The nested-loop join algorithm is expensive, since it examines every pair of tuples in the two relations. Consider the cost of the nested-loop join algorithm. The number of pairs of tuples to be considered is $n_r * n_s$, where n_r denotes the number of tuples in r , and n_s denotes the number of tuples in s. For each record in r , we have to perform a complete scan on s. In the worst case, the buffer can hold only one block of each relation, and a total of $n_r * b_s + b_r$ block transfers would be required, where b_r and b_s denote the number of blocks containing tuples of

and s, respectively. We need only one seek for each scan on the inner relation
since it is read sequentially, and a total of b_r seeks to read r , leading to a total of $n_r + b_r$ seeks. In the best case, there is enough space for both relations to fit simultaneously in memory, so each block would have to be read only once; hence, only $b_r + b_s$ block transfers would be required, along with 2 seeks.

```
for each tuple $t_r$ in $r$ do begin
   for each tuple $t_s$ in $s$ do begin
      test pair ($t_r$, $t_s$) to see if they satisfy the join condition u if they do, add
      $t_r \cdot t_s$ to the result;
   end
end
```

Figure 12.5 Nested-loop join.

```
for each block $B_r$ of $r$ do begin
   for each block $B_s$ of $s$ do begin
       for each tuple $t_r$ in $B_r$ do begin
          for each tuple $t_s$ in $B_s$ do begin
              test pair ($t_r$ , $t_s$ ) to see if they satisfy the join condition if they do, add $t_r \cdot t_s$ to the result;
          end
       end
   end
end
```

Figure 12.6 Block nested-loop join.

If one of the relations fits entirely in main memory, it is beneficial to use that relation as the inner relation, since the inner relation would then be read only once. Therefore, if *s* is small enough to fit in main memory, our strategy requires only a total $b_r + b_s$ block transfers and 2 seeks — the same cost as that for the case where both relations fit in memory.

Now consider the natural join of *student* and *takes*. Assume for now that we have no indices whatsoever on either relation, and that we are not willing to create any index. We can use the nested loops to compute the join; assume that *student* is the outer relation and *takes* is the inner relation in the join. We will have to examine 5000 * 10, 000 = 50 * 10^6 pairs of tuples. In the worst case, the number of block transfers is 5000 * 400 + 100 = 2,000,100, plus 5000 + 100 = 5100 seeks. In the best-case scenario, however, we can read both relations only once, and perform the computation. This computation requires at most 100 + 400 = 500 block transfers, plus 2 seeks — a significant improvement over the worst-case scenario. If we had used *takes* as the relation for the outer loop and *student* for the inner loop, the worst-case cost of our final strategy would have been 10,000 * 100 + 400 = 1,000,400 block transfers, plus 10,400 disk seeks. The number of block transfers is significantly less, and although the number of seeks is higher, the overall cost is reduced, assuming t_S = 4 milliseconds and t_T = 0.1 milliseconds.

12.5.4 Merge Join

The **merge-join** algorithm (also called the **sort-merge-join** algorithm) can be used to compute natural joins and equi-joins. Let *r* (*R*) and *s*(*S*) be the relations whose natural join is to be computed, and let $R \cap S$ denote their common attributes. Suppose that both relations are sorted on the attributes $R \cap S$. Then, their join can be computed by a process much like the merge stage in the merge – sort algorithm.

Merge-Join Algorithm

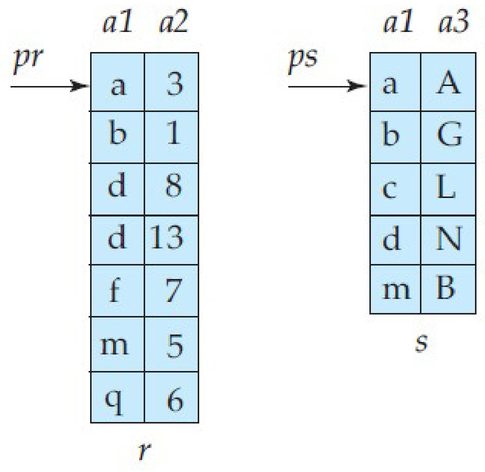
Figure 12.7 shows the merge-join algorithm. In the algorithm, *JoinAttrs* refers to the attributes in $R \cap S$, and $t_r\ t_s$, where t_r and t_s are tuples that have the same

```
$pr$ := address of first tuple of $r$;
```

```
ps := address of first tuple of s;
while (ps  =null and pr  =null) do
    begin
        $t_s$ := tuple to which ps points;
        $S_s$ := {$t_s$ };
        set ps to point to next tuple of s;
        done := false;
        while (not done and ps  =null) do
            begin
                $t_s$  := tuple to which ps points;
                if ($t_s$ [JoinAttrs] =  $t_s$ [JoinAttrs])
                  then begin
                          $S_s$ := $S_s \cup$ {$t_s$ };
                          set ps to point to next tuple of s;
                       end
                          else done := true;
            end
        $t_r$ := tuple to which pr points;
        while ( pr =null and $t_r$ [JoinAttrs] < $t_s$ [JoinAttrs]) do begin

                set pr to point to next tuple of r;
                $t_r$ := tuple to which pr points;
            end

        while ( pr =null and $t_r$ [JoinAttrs] = $t_s$ [JoinAttrs]) do begin
                for each $t_s$ in $S_s$ do
                  begin
                     add $t_s$     $t_r$ to result;
                  end

                set pr to point to next tuple of r;
                $t_r$ := tuple to which pr points;
            end
    end.
```

Figure 12.7 Merge join.

values for *JoinAttrs*, denotes the concatenation of the attributes of the tuples, fol-lowed by projecting out repeated attributes. The merge-join algorithm associates one pointer with each relation. These pointers point initially to the first tuple of the respective relations. As the algorithm proceeds, the pointers move through the relation. A group of tuples of one relation with the same value on the join attributes is read into S_s . The algorithm in Figure 12.7 *requires* that every set of tuples S_s fit in main memory; we discuss extensions of the algorithm to avoid this requirement shortly. Then, the corresponding tuples (if any) of the other relation are read in, and are processed as they are read.



a1	a2
a	3
b	1
d	8
d	13
f	7
m	5
q	6

a1	a3
a	A
b	G
c	L
d	N
m	B

Figure 12.8 Sorted relations for merge join.

Sorted relations for merge join.

Figure 12.8 shows two relations that are sorted on their join attribute *a* 1. It is instructive to go through the steps of the merge-join algorithm on the relations shown in the figure.

The merge-join algorithm of Figure 12.7 requires that each set S_s of all tuples with the same value for the join attributes must fit in main memory. This require-ment can usually be met, even if the relation *s* is large. If there are some join attribute values for which S_s is larger than available memory, a block nested-loop join can be performed for such sets S_s , matching them with corresponding blocks of tuples in *r* with the same values for the join attributes.

If either of the input relations *r* and *s* is not sorted on the join attributes, they can be sorted first, and then the merge-join algorithm can be used. The merge-join algorithm can also be easily extended from natural joins to the more general case of equi-joins.

Other Operations

Other relational operations and extended relational operations — such as dupli-cate elimination, projection, set operations, outer join, and aggregation — can be implemented as outlined in Sections 12.6.1 through 12.6.5.

Duplicate Elimination

We can implement duplicate elimination easily by sorting. Identical tuples will appear adjacent to each other as a result of sorting, and all but one copy can be removed. With external sort – merge, duplicates found while a run is being created can be removed before the run is written to disk, thereby reducing the number of block transfers. The remaining duplicates can be eliminated during merging, and the final sorted run has no duplicates. The worst-case cost estimate for duplicate elimination is the same as the worst-case cost estimate for sorting of the relation.

We can also implement duplicate elimination by hashing, as in the hash-join algorithm. First, the relation is partitioned on the basis of a hash function on the whole tuple. Then, each partition is read in, and an in-memory hash index is constructed. While constructing the hash index, a tuple is inserted only if it is not already present. Otherwise, the tuple is discarded. After all tuples in the partition have been processed, the tuples in the hash index are written to the result. The cost estimate is the same as that for the cost of processing (partitioning and reading each partition) of the build relation in a hash join.

Because of the relatively high cost of duplicate elimination, SQL requires an explicit request by the user to remove duplicates; otherwise, the duplicates are retained.

Projection

We can implement projection easily by performing projection on each tuple, which gives a relation that could have duplicate records, and then removing duplicate records. Duplicates can be eliminated by the methods described in Section 12.6.1. If the attributes in the projection list include a key of the relation, no duplicates will exist; hence, duplicate elimination is not required. Generalized projection can be implemented in the same way as projection.

Set Operations

We can implement the *union*, *intersection*, and *set-difference* operations by first sorting both relations, and then scanning once through each of the sorted relations to produce the result. In $r \cup s$, when a concurrent scan of both relations reveals the same tuple in both files, only one of the tuples is retained. The result of $r \cap s$ will contain only those tuples that appear in both relations. We implement *set difference*, $r - s$, similarly, by retaining tuples in *r* only if they are absent in *s*.

For all these operations, only one scan of the two sorted input relations is required, so the cost is $b_r + b_s$ block transfers if the relations are sorted in the same order. Assuming a worst case of one block buffer for each relation, a total of $b_r + b_s$ disk seeks would be required in addition to $b_r + b_s$ block transfers. The number of seeks can be reduced by allocating extra buffer blocks.

If the relations are not sorted initially, the cost of sorting has to be included. Any sort order can be used in evaluation of set operations, provided that both inputs have that same sort order

Hashing provides another way to implement these set operations. The first step in each case is to partition the two relations by the same hash function, and thereby create the partitions $r_0, r_1, \ldots, r_{nh}$ and $s_0, s_1, \ldots, s_{nh}$. Depending on the operation, the system then takes these steps on each partition $i = 0, 1, \ldots, n_h$:

• $r \cup s$

Build an in-memory hash index on r_i.

Add the tuples in s_i to the hash index only if they are not already present.

Add the tuples in the hash index to the result.

$r \cap s$

Build an in-memory hash index on r_i.

For each tuple in s_i, probe the hash index and output the tuple to the result only if it is already present in the hash index.

$r - s$

Build an in-memory hash index on r_i.

For each tuple in s_i, probe the hash index, and, if the tuple is present in the hash index, delete it from the hash index.

Add the tuples remaining in the hash index to the result.

Outer Join

Recall the *outer-join operations* described in Section 4.1.2. For example, the natural left outer join *takes* ⟕ *student* contains the join of *takes* and *student*, and, in *addition, for each takes tuple t that has no matching tuple in student (that is, where ID is not in student), the following tuple t_1 is added to the result. For all attributes in the schema of takes, tuple t_1 has the same values as tuple t. The remaining attributes (from the schema of student) of tuple t_1 contain the value null.*

We can implement the outer-join operations by using one of two strategies:

Compute the corresponding join, and then add further tuples to the join result to get the outer-join result. Consider the left outer-join operation and two relations: *r* (*R*) and *s*(*S*). To evaluate *r* ⊐u *s*, we first compute *r* u *s*, and save that result as temporary relation q_1. Next, we compute *r* − $_R(q_1)$ to obtain those tuples in *r* that do not participate in the theta join. We can use any of the algorithms for computing the joins, projection, and set difference described earlier to compute the outer joins. We pad each of these tuples with null values for attributes from *s*, and add it to q_1 to get the result of the outer join.

The right outer-join operation *r* ⊐u *s* is equivalent to *s* ⊐u *r* , and can therefore be implemented in a symmetric fashion to the left outer join. We can implement the full outer-join operation *r* ⊐⊐u *s* by computing the join *r s*, and then adding the extra tuples of both the left and right outer-join operations, as before.

Modify the join algorithms. It is easy to extend the nested-loop join algo-rithms to compute the left outer join: Tuples in the outer relation that do not match any tuple in the inner relation are written to the output after being padded with null values. However, it is hard to extend the nested-loop join to compute the full outer join.

Natural outer joins and outer joins with an equi-join condition can be computed by extensions of the merge-join and hash-join algorithms. Merge join can be extended to compute the full outer join as follows: When the merge of the two relations is being done, tuples in either relation that do not match any tuple in the other relation can be padded with nulls and written to the output. Similarly, we can extend merge join to compute the left and right outer joins by writing out nonmatching tuples (padded with nulls) from only one of the relations. Since the relations are sorted, it is easy to detect whether or not a tuple matches any tuples from the other relation. For example, when a merge join of *takes* and *student* is done, the tuples are read in sorted order of *ID*, and it is easy to check, for each tuple, whether there is a matching tuple in the other.

The cost estimates for implementing outer joins using the merge-join algorithm are the same as are those for the corresponding join. The only difference lies in size of the result, and therefore in the block transfers for writing it out, which we did not count in our earlier cost estimates.

The extension of the hash-join algorithm to compute outer joins is left for you to do as an exercise (Exercise 12.15).

Aggregation

Recall the aggregation function (operator), discussed in Section 3.7. For example, the function

select *dept name*, **avg** (*salary*)
from *instructor*
group by *dept name*;

computes the average salary in each university department.

The aggregation operation can be implemented in the same way as duplicate elimination. We use either sorting or hashing, just as we did for duplicate elimina-tion, but based on the grouping attributes (*branch name* in the preceding example). However, instead of eliminating tuples with the same value for the grouping at-tribute, we gather them into groups, and apply the aggregation operations on each group to get the result.

The cost estimate for implementing the aggregation operation is the same as the cost of duplicate elimination, for aggregate functions such as **min**, **max**, **sum**, **count**, and **avg**.

Instead of gathering all the tuples in a group and then applying the aggre-gation operations, we can implement the aggregation operations **sum**, **min**, **max**, **count**, and **avg** on the fly as the groups are being constructed. For the case of **sum**, **min**, and **max**, when two tuples in the same group are found, the system replaces them by a single tuple containing the **sum**, **min**, or **max**, respectively, of the columns being aggregated. For the **count** operation, it maintains a running count for each group for which a tuple has been found. Finally, we implement the **avg** operation by computing the sum and the count values on the fly, and finally dividing the sum by the count to get the average.

If all tuples of the result fit in memory, both the sort-based and the hash-based implementations do not need to write any tuples to disk. As the tuples are read in, they can be inserted in a sorted tree structure or in a hash index. When we use on-the-fly aggregation techniques, only one tuple needs to be stored for each of the groups. Hence, the sorted tree structure or hash index fits in memory, and the aggregation can be processed with just b_r block transfers (and 1 seek) instead of the $3b_r$ transfers (and a worst case of up to $2b_r$ seeks) that would be required otherwise.

Evaluation of Expressions

So far, we have studied how individual relational operations are carried out. Now we consider how to evaluate an expression containing multiple operations. The obvious way to evaluate an expression is simply to evaluate one operation at a time, in an appropriate order. The result of each evaluation is **materialized** in a temporary relation for subsequent use. A disadvantage to this approach is the need to construct the temporary relations, which (unless they are small) must be written to disk. An alternative approach is to evaluate several operations simultaneously in a **pipeline**, with the results of one operation passed on to the next, without the need to store a temporary relation.

In Sections 12.7.1 and 12.7.2, we consider both the *materialization* approach and the *pipelining* approach. We shall see that the costs of these approaches can differ substantially, but also that there are cases where only the materialization approach is feasible.

Materialization

It is easiest to understand intuitively how to evaluate an expression by looking at a pictorial representation of the expression in an **operator tree**. Consider the expression:

$$_{name}\ (\mathrm{s}_{building\ =\ \text{“Watson”}}(department) instructor\ )$$

in Figure 12.11.

If we apply the materialization approach, we start from the lowest-level op-erations in the expression (at the bottom of the tree). In our example, there is only one such operation: the selection operation on *department*. The inputs to the lowest-level operations are relations in the database. We execute these operations by the algorithms that we studied earlier, and we store the results in temporary relations. We can use these temporary relations to execute the operations at the next level up in the tree, where the inputs now are either temporary relations or relations stored in the database. In

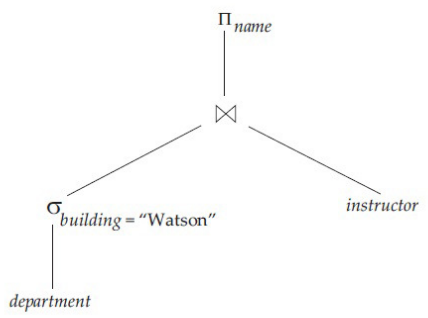


Figure 12.11 Pictorial representation of an expression.

our example, the inputs to the join are the *in-structor* relation and the temporary relation created by the selection on *department*. The join can now be evaluated, creating another temporary relation.

By repeating the process, we will eventually evaluate the operation at the root of the tree, giving the final result of the expression. In our example, we get the final result by executing the projection operation at the root of the tree, using as input the temporary relation created by the join.

Evaluation as just described is called **materialized evaluation**, since the re-sults of each intermediate operation are created (materialized) and then are used for evaluation of the next-level operations.

The cost of a materialized evaluation is not simply the sum of the costs of the operations involved. When we computed the cost estimates of algorithms, we ignored the cost of writing the result of the operation to disk. To compute the cost of evaluating an expression as done here, we have to add the costs of all the operations, as well as the cost of writing the intermediate results to disk. We assume that the records of the result accumulate in a buffer, and, when the buffer is full, they are written to disk. The number of blocks written out, b_r , can be estimated as n_r / f_r , where n_r is the estimated number of tuples in the result relation r , and f_r is the *blocking factor* of the result relation, that is, the number of records of r that will fit in a block. In addition to the transfer time, some disk seeks may be required, since the disk head may have moved between successive writes. The number of seeks can be estimated as b_r /b_b where b_b is the size of the output buffer (measured in blocks).

Double buffering (using two buffers, with one continuing execution of the algorithm while the other is being written out) allows the algorithm to execute more quickly by performing CPU activity in parallel with I/O activity. The number of seeks can be reduced by allocating extra blocks to the output buffer, and writing out multiple blocks at once.

Pipelining

We can improve query-evaluation efficiency by reducing the number of tem-porary files that are produced. We achieve this reduction by combining several relational operations into a *pipeline* of operations, in which the results of one operation are passed along to the next operation in the pipeline. Evaluation as just described is called **pipelined evaluation**.

For example, consider the expression ($_{a\ 1,a\ 2}(r\ s)$). If materialization were applied, evaluation would involve creating a temporary relation to hold the result of the join, and then reading back in the result to perform the projection. These operations can be combined: When the join operation generates a tuple of its result, it passes that tuple immediately to the project operation for processing. By combining the join and the projection, we avoid creating the intermediate result, and instead create the final result directly.

Creating a pipeline of operations can provide two benefits:

It eliminates the cost of reading and writing temporary relations, reducing the cost of query evaluation.

It can start generating query results quickly, if the root operator of a query-evaluation plan is combined in a pipeline with its inputs. This can be quite useful if the results are displayed to a user as they are generated, since otherwise there may be a long delay before the user sees any query results.

Implementation of Pipelining

We can implement a pipeline by constructing a single, complex operation that combines the operations that constitute the pipeline. Although this approach may be feasible for some frequently occurring situations, it is desirable in general to reuse the code for individual operations in the construction of a pipeline.

In the example of Figure 12.11, all three operations can be placed in a pipeline, which passes the results of the selection to the join as they are generated. In turn, it passes the results of the join to the projection as they are generated. The memory requirements are low, since results of an operation are not stored for long. However, as a result of pipelining, the inputs to the operations are not available all at once for processing.

Pipelines can be executed in either of two ways:

In a **demand-driven pipeline**, the system makes repeated requests for tuples from the operation at the top of the pipeline. Each time that an operation receives a request for tuples, it computes the next tuple (or tuples) to be returned, and then returns that tuple. If the inputs of the operation are not pipelined, the next tuple(s) to be returned can be computed from the input relations, while the system keeps track of what has been returned so far. If it has some pipelined inputs, the operation also makes requests for tuples from its pipelined inputs. Using the tuples received from its pipelined inputs, the operation computes tuples for its output, and passes them up to its parent.

In a **producer-driven pipeline**, operations do not wait for requests to pro-duce tuples, but instead generate the tuples **eagerly**. Each operation in a producer-driven pipeline is modeled as a separate process or thread within the system that takes a stream of tuples from its pipelined inputs and gen-erates a stream of tuples for its output.

We describe below how demand-driven and producer-driven pipelines can be implemented.

Each operation in a demand-driven pipeline can be implemented as an **iter-ator** that provides the following functions: *open*(), *next*(), and *close*(). After a call to *open*(), each call to *next*() returns the next output tuple of the operation. The implementation of the operation in turn calls *open*() and *next*() on its inputs, to get its input tuples when required. The function *close*() tells an iterator that no more tuples are required. The iterator maintains the **state** of its execution in between calls, so that successive *next*() requests receive successive result tuples.

For example, for an iterator implementing the select operation using linear search, the *open*() operation starts a file scan, and the iterator's state records the point to which the file has been scanned. When the *next*() function is called, the file scan continues from after the previous point; when the next tuple satisfying the selection is found by scanning the file, the tuple is returned after storing the point where it was found in the iterator state. A merge-join iterator's *open*() operation would open its inputs, and if they are not already sorted, it would also sort the inputs. On calls to *next*(), it would return the next pair of matching tuples. The state information would consist of up to where each input had been scanned. Details of the implementation of iterators are left for you to complete in Practice Exercise 12.7.

Producer-driven pipelines, on the other hand, are implemented in a different manner. For each pair of adjacent operations in a producer-driven pipeline, the system creates a buffer to hold tuples being passed from one operation to the next. The processes or threads corresponding to different operations execute concurrently. Each operation at the bottom of a pipeline continually generates output tuples, and puts them in its output buffer, until the buffer is full. An operation at any other level of a pipeline generates output tuples when it gets input tuples from lower down in the pipeline, until its output buffer is full. Once the operation uses a tuple from a pipelined input, it removes the tuple from its input buffer. In either case, once the output buffer is full, the operation waits until its parent operation removes tuples from the buffer, so that the buffer has space for more tuples. At this point, the operation generates more tuples, until the buffer is full again. The operation repeats this process until all the output tuples have been generated.

It is necessary for the system to switch between operations only when an output buffer is full, or an input buffer is empty and more input tuples are needed to generate any more output tuples. In a parallel-processing system, operations in a pipeline may be run concurrently on distinct processors (see Chapter 18).

Using producer-driven pipelining can be thought of as **pushing** data up an operation tree from below, whereas using demand-driven pipelining can be thought of as **pulling** data up an operation tree from the top. Whereas tuples are generated *eagerly* in producer-driven pipelining, they are generated **lazily**, on demand, in demand-driven pipelining. Demand-driven pipelining is used

Evaluation Algorithms for Pipelining

Some operations, such as sorting, are inherently **blocking operations**, that is, they may not be able to output any results until all tuples from their inputs have been examined.[5]

Other operations, such as join, are not inherently blocking, but specific eval-uation algorithms may be blocking. For example, the hash-join algorithm is a blocking operation, since it requires both its inputs to be fully retrieved and parti-tioned, before it outputs any tuples. On the other hand, the indexed nested loops join algorithm can output result tuples as it gets tuples for the outer relation. It is therefore said to be **pipelined** on its outer (left-hand side) relation, although it is blocking on its indexed (right-hand side) input, since the index must be fully constructed before the indexed nested-loop join algorithm can execute.

Hybrid hash join can be viewed as partially pipelined on the probe relation, since it can output tuples from the first partition as tuples are received for the probe relation. However, tuples that are not in the first partition will be output only after the entire pipelined input relation is received. Hybrid hash join thus provides pipelined evaluation on its probe input if the build input fits entirely in memory, or nearly pipelined evaluation if most of the build input fits in memory.

If both inputs are sorted on the join attribute, and the join condition is an equi-join, merge join can be used, with both its inputs pipelined.

However, in the more common case that the two inputs that we desire to pipeline into the join are not already sorted, another alternative is the **double-pipelined join** technique, shown in Figure 12.12. The algorithm assumes that the input tuples for both input relations, *r* and *s*, are pipelined. Tuples made available for both relations are queued for processing in a single queue. Special queue entries, called End_r and End_s , which serve as end-of-file markers, are inserted in the queue after all tuples from *r* and *s* (respectively) have been generated. For efficient evaluation, appropriate indices should be built on the relations *r* and *s*. As tuples are added to *r* and *s*, the indices must be kept up to date. When hash indices are used on *r* and *s*, the resultant algorithm is called the **double-pipelined hash-join** technique.

The double-pipelined join algorithm in Figure 12.12 assumes that both inputs fit in memory. In case the two inputs are larger than memory, it is still possible to use the double-pipelined join technique as usual until available memory is full. When available memory becomes full, *r* and *s* tuples that have arrived up to that point can be treated as being in partition r_0 and s_0, respectively. Tuples for r and s that arrive subsequently are assigned to partitions r_1 and s_1, respectively, which

```
    $done_r$ := false;
    $done_s$ := false;

while not $done_r$ or not $done_s$ do begin

    if queue is empty, then wait until queue is not empty; $t$ := top entry in queue;
    if $t = End_r$ then $done_r$ := true
else if $t = End_s$ then $done_s$ := true else if $t$ is from input $r$
    then
    begin
    $r := r \cup \{t\}$;
    $result := result \cup (\{t\}$                    $s)$;
    end
    else /* $t$ is from input $s$ */
    begin
    $s := s \cup \{t\}$;
    $result := result \cup (r$                    $\{t\})$;
    end
    end
```

Figure 12.12 Double-pipelined join algorithm.

are written to disk, and are not added to the in-memory index. However, tuples assigned to r_1 and s_1 are used to probe s_0 and r_0, respectively, before they are written to disk. Thus, the join of r_1 with s_0, and s_0 with r_1, is also carried out in a pipelined fashion. After r and s have been fully processed, the join of r_1 tuples with s_1 tuples must be carried out, to complete the join; any of the join techniques we have seen earlier can be used to join r_1 with s_1.

Query Optimization

Query optimization is the process of selecting the most efficient query-evaluation plan from among the many strategies usually possible for processing a given query, especially if the query is complex. We do not expect users to write their queries so that they can be processed efficiently. Rather, we expect the system to construct a query-evaluation plan that minimizes the cost of query evaluation. This is where query optimization comes into play.

One aspect of optimization occurs at the relational-algebra level, where the system attempts to find an expression that is equivalent to the given expression, but more efficient to execute. Another aspect is selecting a detailed strategy for processing the query, such as choosing the algorithm to use for executing an operation, choosing the specific indices to use, and so on.

The difference in cost (in terms of evaluation time) between a good strategy and a bad strategy is often substantial, and may be several orders of magnitude. Hence, it is worthwhile for the system to spend a substantial amount of time on the selection of a good strategy for processing a query, even if the query is executed only once.

Transformation of Relational Expressions

A query can be expressed in several different ways, with different costs of eval-uation. In this section, rather than take the relational expression as given, we consider alternative, equivalent expressions.

Two relational-algebra expressions are said to be **equivalent** if, on every legal database instance, the two expressions generate the same set of tuples. (Recall that a legal database instance is one that satisfies all the integrity constraints specified in the database schema.) Note that the order of the tuples is irrelevant; the two expressions may generate the tuples in different orders, but would be considered equivalent as long as the set of tuples is the same.

In SQL, the inputs and outputs are multisets of tuples, and the multiset version of the relational algebra (described in the box in page 238) is used for evaluating SQL queries. Two expressions in the *multiset* version of the relational algebra are said to be equivalent if on every legal database the two expressions generate the same multiset of tuples. The discussion in this chapter is based on the relational

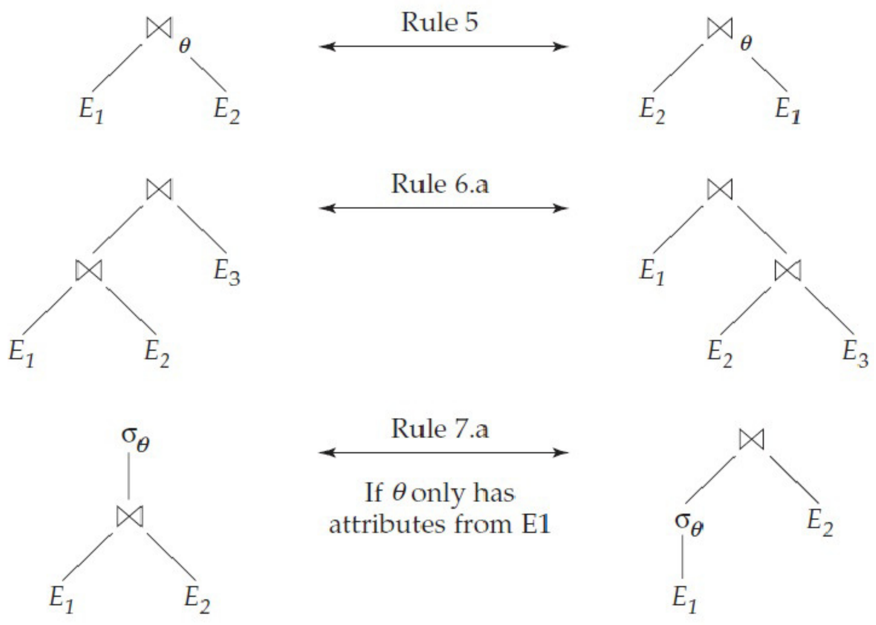


Figure 13.3 Pictorial representation of equivalences.

Pictorial representation of equivalences.

algebra. We leave extensions to the multiset version of the relational algebra to you as exercises.

Equivalence Rules

An **equivalence rule** says that expressions of two forms are equivalent. We can replace an expression of the first form by an expression of the second form, or vice versa — that is, we can replace an expression of the second form by an expression of the first form — since the two expressions generate the same result on any valid database. The optimizer uses equivalence rules to transform expressions into other logically equivalent expressions.

We now list a number of general equivalence rules on relational-algebra expressions. Some of the equivalences listed appear in Figure 13.3. We use u, u_1, u_2, and so on to denote predicates, L_1, L_2, L_3, and so on to denote lists of attributes, and E, E_1, E_2, and so on to denote relational-algebra expressions. A relation name r is simply a special case of a relational-algebra expression, and can be used wherever E appears.

Conjunctive selection operations can be deconstructed into a sequence of individual selections. This transformation is referred to as a cascade of s.

$$su_1 \wedge u_2 {}^{(E)} = su_1 {}^{(}su_2 {}^{(E))}$$

Selection operations are **commutative**.

$$s_{u1} (s_{u2} (E)) = s_{u2} (s_{u1} (E))$$

Only the final operations in a sequence of projection operations are needed; the others can be omitted. This transformation can also be referred to as a cascade of .

$$_{L1} (\; _{L2} (. . . (\; _{Ln} (E)) . . .)) = \; _{L1} (E)$$

Selections can be combined with Cartesian products and theta joins.

$$s_u (E_1 \times E_2) = E_1 \; _u \, E_2$$

This expression is just the definition of the theta join.

$$su_1 {}^{(E}1 \quad u_2 \; {}^{E}2^{)} = {}^{E}1 \quad u_1 \wedge u_2 \; {}^{E}2$$

Theta-join operations are commutative.

$$E_1 \quad _u \quad E_2 = E_2 \quad _u \quad E_1$$

Actually, the order of attributes differs between the left-hand side and right-hand side, so the equivalence does not hold if the order of attributes is taken into account. A projection operation can be added to one of the sides of the equivalence to appropriately reorder attributes, but for simplicity we omit the projection and ignore the attribute order in most of our examples.

Recall that the natural-join operator is simply a special case of the theta-join operator; hence, natural joins are also commutative.

a. Natural-join operations are **associative**.

$$(E_1 \quad E_2) \quad E_3 = E_1 \quad (E_2 \quad E_3)$$

b. Theta joins are associative in the following manner:

$${}^{(E}1u_1 \; {}^{E}2^{)} \; u_2 \wedge u_3 \; {}^{E}3 = {}^{E}1 \quad u_1 \wedge u_3 \; {}^{(E}2 \quad u_2 \; {}^{E}3^{)}$$

where u_2 involves attributes from only E_2 and E_3. Any of these condi-tions may be empty; hence, it follows that the Cartesian product (×) operation is also associative. The commutativity and associativity of join operations are important for join reordering in query optimization.

The selection operation distributes over the theta-join operation under the following two conditions:

It distributes when all the attributes in selection condition u_0 involve only the attributes of one of the expressions (say, E_1) being joined.

$$s_{u0} (E_1 \ {}_{u} E_2) = (s_{u0} (E_1)) \ {}_{u} E_2$$

It distributes when selection condition u_1 involves only the attributes of E_1 and u_2 involves only the attributes of E_2.

$$su_1 \wedge u_2 {}^{(E}1u{}^{E}2^{)} = {}^{(}su_1{}^{(E}1^{))} \ u \ {}^{(}su_2{}^{(E}2^{))}$$

The projection operation distributes over the theta-join operation under the following conditions.

Let L_1 and L_2 be attributes of E_1 and E_2, respectively. Suppose that the join condition u involves only attributes in $L_1 \cup L_2$. Then,

$$L_1 \cup L_2 {}^{(E}1 \ \ u{}^{E}2^{)} = {}^{(} \ L_1{}^{(E}1^{))} \ u{}^{(} \ L_2{}^{(E}2^{))}$$

Consider a join $E_1 \ {}_{u} E_2$. Let L_1 and L_2 be sets of attributes from E_1 and E_2, respectively. Let L_3 be attributes of E_1 that are involved in join condition u, but are not in $L_1 \cup L_2$, and let L_4 be attributes of E_2 that are involved in join condition u, but are not in $L_1 \cup L_2$. Then,

$$L_1 \cup L_2 {}^{(E}1 \ \ u{}^{E}2^{)} = L_1 \cup L_2 {}^{((} \ L_1 \cup L_3{}^{(E}1^{))} \ u{}^{(} \ L_2 \cup L_4{}^{(E}2^{)))}$$

The set operations union and intersection are commutative.

$$E_1 \cup E_2$$

$$E_1 \cap E_2$$

Set difference is not commutative.

$E_2 \cup E_1$

$E_2 \cap E_1$

Set union and intersection are associative.

$$(E_1 \cup E_2) \cup E_3 = E_1 \cup (E_2 \cup E_3)$$

$$(E_1 \cap E_2) \cap E_3 = E_1 \cap (E_2 \cap E_3)$$

The selection operation distributes over the union, intersection, and set-difference operations.

$$s_P (E_1 - E_2) = s_P (E_1) - s_P (E_2)$$

Similarly, the preceding equivalence, with – replaced with either ∪ or ∩, also holds. Further:

$$s_P (E_1 - E_2) = s_P (E_1) - E_2$$

The preceding equivalence, with – replaced by ∩, also holds, but does not hold if – is replaced by ∪.

The projection operation distributes over the union operation. $_L$ (E_1 ∪ E_2) = ($_L$ (E_1)) ∪ ($_L$ (E_2))

This is only a partial list of equivalences. More equivalences involving ex-tended relational operators, such as the outer join and aggregation, are discussed in the exercises.

Estimating Statistics of Expression Results

The cost of an operation depends on the size and other statistics of its inputs. Given an expression such as *a* (*b c*) to estimate the cost of joining *a* with (*b c*), we need to have estimates of statistics such as the size of *b c*.

In this section, we first list some statistics about database relations that are stored in database-system catalogs, and then show how to use the statistics to estimate statistics on the results of various relational operations.

One thing that will become clear later in this section is that the estimates are not very accurate, since they are based on assumptions that may not hold exactly. A query-evaluation plan that has the lowest estimated execution cost may therefore not actually have the lowest actual execution cost. However, real-world experience has shown that even if estimates are not precise, the plans with the lowest estimated costs usually have actual execution costs that are either the lowest actual execution costs, or are close to the lowest actual execution costs.

Catalog Information

The database-system catalog stores the following statistical information about database relations:

n_r , the number of tuples in the relation r.

b_r , the number of blocks containing tuples of relation r .

l_r , the size of a tuple of relation r in bytes.

f_r , the blocking factor of relation r — that is, the number of tuples of relation r that fit into one block.

$V(A, r)$, the number of distinct values that appear in the relation r for attribute A. This value is the same as the size of $_A(r)$. If $A$is a key for relation r , $V(A, r)$ is n_r .

The last statistic, $V(A, r)$, can also be maintained for sets of attributes, if desired, instead of just for individual attributes. Thus, given a set of attributes, A, $V(A, r)$ is the size of $_A(r)$.

If we assume that the tuples of relation r are stored together physically in a file, the following equation holds:

$$b_r = \frac{nr}{f_r}$$

Statistics about indices, such as the heights of B$^+$-tree indices and number of leaf pages in the indices, are also maintained in the catalog.

If we wish to maintain accurate statistics, then, every time a relation is modi-fied, we must also update the statistics. This update incurs a substantial amount of overhead. Therefore, most systems do not update the statistics on every mod-ification. Instead, they update the statistics during periods of light system load. As a result, the statistics used for choosing a query-processing strategy may not be completely accurate. However, if not too many updates occur in the intervals between the updates of the statistics, the statistics will be sufficiently accurate to provide a good estimation of the relative costs of the different plans.

The statistical information noted here is simplified. Real-world optimizers often maintain further statistical information to improve the accuracy of their cost estimates of evaluation plans. For instance, most databases store the distribution of values for each attribute as a **histogram**: in a histogram the values for the attribute are divided into a number of ranges, and with each range the histogram associates the number of tuples whose attribute value lies in that range. Figure 13.6 shows an example of a histogram for an integer-valued attribute that takes values in the range 1 to 25.

Histograms used in database systems usually record the number of distinct values in each range, in addition to the number of tuples with attribute values in that range.

As an example of a histogram, the range of values for an attribute *age* of a re-lation *person* could be divided into 0 – 9, 10 – 19, . . . , 90 – 99 (assuming a maximum age of 99). With each range we store a count of the number of *person* tuples whose *age* values lie in that range, and the number of distinct age values that lie in that

range. Without such histogram information, an optimizer would have to assume that the distribution of values is uniform; that is, each range has the same count.

A histogram takes up only a little space, so histograms on several different at-tributes can be stored in the system catalog. There are several types of histograms used in database systems. For example, an **equi-width histogram** divides the range of values into equal-sized ranges, whereas an **equi-depth** histogram ad-justs the boundaries of the ranges such that each range has the same number of values.

Selection Size Estimation

The size estimate of the result of a selection operation depends on the selection predicate. We first consider a single equality predicate, then a single comparison predicate, and finally combinations of predicates.

$s_{A=a}(r)$: If we assume uniform distribution of values (that is, each value ap-pears with equal probability), the selection result can be estimated to have $n_r / V(A, r)$ tuples, assuming that the value *a* appears in attribute *A* of some record of *r*. The assumption that the value *a* in the selection appears in some record is generally true, and cost estimates often make it implicitly. However, it is often not realistic to assume that each value appears with equal prob-ability. The *course id* attribute in the *takes* relation is an example where the assumption is not valid. It is reasonable to expect that a popular undergradu-ate course will have many more students than a smaller specialized graduate course. Therefore, certain *course id* values appear with greater probability than do others. Despite the fact that the uniform-distribution assumption is often not correct, it is a reasonable approximation of reality in many cases, and it helps us to keep our presentation relatively simple.

If a histogram is available on attribute A, we can locate the range that contains the value a , and modify the above-mentioned estimate $n_r / V(A, r)$

by using the frequency count for that range instead of n_r , and the number of distinct values that occurs in that range instead of $V(A, r)$.

s$_{A\leq v}$ (r): Consider a selection of the form s$_{A\leq v}$ (r). If the actual value used in the comparison (v) is available at the time of cost estimation, a more
accurate estimate can be made. The lowest and highest values (min(A, r) and max(A, r)) for the attribute can be stored in the catalog. Assuming that values are uniformly distributed, we can estimate the number of records that will satisfy the condition $A \leq v$ as 0 if $v <$ min(A, r), as n_r if $v \geq$ max(A, r), and:

$$nr \cdot \frac{v - \min(A, r)}{\max(A, r) - \min(A, r)}$$

otherwise.

If a histogram is available on attribute A, we can get a more accurate estimate; we leave the details as an exercise for you. In some cases, such as when the query is part of a stored procedure, the value v may not be available when the query is optimized. In such cases, we assume that approximately one-half the records will satisfy the comparison condition. That is, we assume the result has $n_r /2$ tuples; the estimate may be very inaccurate, but is the best we can do without any further information.

Complex selections:

Conjunction: A *conjunctive selection* is a selection of the form:

$$\text{s}_{\text{u}_1 \wedge \text{u}_2 \wedge \cdots \wedge \text{u}_n}(r)$$

We can estimate the result size of such a selection: For each u$_i$, we estimate the size of the selection s$_{\text{u}_i}$ (r), denoted by s_i , as described previously. Thus, the probability that a tuple in the relation satisfies selection condition u$_i$ is

s_i / n_r .

The preceding probability is called the **selectivity** of the selection s$_{\text{u}_i}$ (r). Assuming that the conditions are *independent* of each other, the probability that a tuple satisfies all the conditions is simply the product of all these probabilities. Thus, we estimate the number of tuples in the full selection as:

$$n_r * \frac{s_1 * s_2 * \cdot \cdot \cdot * s_n}{n_r^n}$$

Disjunction: A *disjunctive selection* is a selection of the form:

$$\text{s}_{\text{u}_1 \vee \text{u}_2 \vee \cdots \vee \text{u}_n}(r)$$

A disjunctive condition is satisfied by the union of all records satisfying the individual, simple conditions u_i .

As before, let s_i / n_r denote the probability that a tuple satisfies condition u_i . The probability that the tuple will satisfy the disjunction is then 1 minus the probability that it will satisfy *none* of the conditions:

$$1 - (1 - \frac{s_1}{n_r}) * (1 - \frac{s_2}{n_r}) * \cdot \cdot \cdot * (1 - \frac{s_n}{n_r})$$

Multiplying this value by n_r gives us the estimated number of tuples that satisfy the selection.

Negation: In the absence of nulls, the result of a selection $s_{\neg u}(r)$ is simply the tuples of r that are not in $s_u(r)$. We already know how to estimate the number of tuples in $s_u(r)$. The number of tuples in $s_{\neg u}(r)$ is therefore estimated to be $n(r)$ minus the estimated number of tuples in $s_u(r)$.

We can account for nulls by estimating the number of tuples for which the condition u would evaluate to *unknown*, and subtracting that number from the above estimate, ignoring nulls. Estimating that number would require extra statistics to be maintained in the catalog.

Choice of Evaluation Plans

Generation of expressions is only part of the query-optimization process, since each operation in the expression can be implemented with different algorithms. An evaluation plan defines exactly what algorithm should be used for each op-eration, and how the execution of the operations should be coordinated.

Given an evaluation plan, we can estimate its cost using statistics estimated by the techniques in Section 13.3 coupled with cost estimates for various algorithms and evaluation methods described in Chapter 12.

A **cost-based optimizer** explores the space of all query-evaluation plans that are equivalent to the given query, and chooses the one with the least estimated cost. We have seen how equivalence rules can be used to generate equivalent plans. However, cost-based optimization with arbitrary equivalence rules is fairly complicated. We first cover a simpler version of cost-based optimization, which involves only join-order and join algorithm selection, in Section 13.4.1. Later in Section 13.4.2 we briefly sketch how a general-purpose optimizer based on equivalence rules can be built, without going into details.

Exploring the space of all possible plans may be too expensive for complex queries. Most optimizers include heuristics to reduce the cost of query optimiza-tion, at the potential risk of not finding the optimal plan. We study some such heuristics in Section 13.4.3.

Cost-Based Join Order Selection

The most common type of query in SQL consists of a join of a few relations, with join predicates and selections specified in the **where** clause. In this section we consider the problem of choosing the optimal join order for such a query.

For a complex join query, the number of different query plans that are equiv-alent to the query can be large. As an illustration, consider the expression:

$$r_1\, r_2 \quad \cdots \quad r_n$$

where the joins are expressed without any ordering. With $n = 3$, there are 12 different join orderings:

$r_1 \bowtie (r_2 \bowtie r_3)$	$r_1 \bowtie (r_3 \bowtie r_2)$	$(r_2 \bowtie r_3) \bowtie r_1$	$(r_3 \bowtie r_2) \bowtie r_1$
$r_2 \bowtie (r_1 \bowtie r_3)$	$r_2 \bowtie (r_3 \bowtie r_1)$	$(r_1 \bowtie r_3) \bowtie r_2$	$(r_3 \bowtie r_1) \bowtie r_2$
$r_3 \bowtie (r_1 \bowtie r_2)$	$r_3 \bowtie (r_2 \bowtie r_1)$	$(r_1 \bowtie r_2) \bowtie r_3$	$(r_2 \bowtie r_1) \bowtie r_3$

In general, with n relations, there are $(2(n - 1))!/(n - 1)!$ different join orders. (We leave the computation of this expression for you to do in Exercise 13.10.) For joins involving small numbers of relations, this number is acceptable; for example, with $n = 5$, the number is 1680. However, as n increases, this number rises quickly. With $n = 7$, the number is 665,280; with $n = 10$, the number is greater than 17.6 billion!

Luckily, it is not necessary to generate all the expressions equivalent to a given expression. For example, suppose we want to find the best join order of the form:

$$(r_1 r_2 \quad r_3) \qquad r_4 \qquad r_5$$

which represents all join orders where r_1, r_2, and r_3 are joined first (in some order), and the result is joined (in some order) with r_4 and r_5. There are 12 different join orders for computing $r_1\ r_2\ r_3$, and 12 orders for computing the join of this result with r_4 and r_5. Thus, there appear to be 144 join orders to examine. However, once we have found the best join order for the subset of relations $\{r_1, r_2, r_3\}$, we can use that order for further joins with r_4 and r_5, and can ignore all costlier join orders of $r_1\ r_2\ r_3$. Thus, instead of 144 choices to examine, we need to examine only 12 + 12 choices.

Using this idea, we can develop a *dynamic-programming* algorithm for finding optimal join orders. Dynamic-programming algorithms store results of computa-tions and reuse them, a procedure that can reduce execution time greatly.

A recursive procedure implementing the dynamic-programming algorithm appears in Figure 13.7. The procedure applies selections on individual relations at the earliest possible point, that is, when the relations are accessed. It is easiest to understand the procedure assuming that all joins are natural joins, although the procedure works unchanged with any join condition. With arbitrary join con-ditions, the join of two subexpressions is understood to include all join conditions that relate attributes from the two subexpressions **procedure** FindBestPlan(*S*)

```
if (bestpla n[S].cost = ∞) /* bestpla n[S] already computed */ return
    bestpla n[S]
if (S contains only 1 relation)
    set bestpla n[S]. pla n and bestpla n[S].cost based on best way of accessing S else for
each non-empty subset S1 of S such that S1 =S
    P1 = FindBestPlan(S1)
```

```
        P2 = FindBestPlan($S - S1$)

        A = best algorithm for joining results of $P1$ and $P2$ cost =
        $P1.cost + P2.cost$ + cost of $A$

        if cost < $bestpla\ n[S].cost$

            $bestpla\ n[S].cost$ = cost

            $bestpla\ n[S].\ pla\ n$ = "execute $P1.\ pla\ n$; execute $P2.\ pla\ n$; join results
                                      of $P1$ and $P2$ using $A$"

    return $bestpla\ n[S]$
```

Figure 13.7 Dynamic-programming algorithm for join order optimization.

The procedure stores the evaluation plans it computes in an associative array *bestpla n*, which is indexed by sets of relations. Each element of the associative array contains two components: the cost of the best plan of S, and the plan itself. The value of $bestpla\ n[S].cost$ is assumed to be initialized to ∞ if $bestpla\ n[S]$ has not yet been computed.

The procedure first checks if the best plan for computing the join of the given set of relations S has been computed already (and stored in the associative array *bestpla n*); if so, it returns the already computed plan.

If S contains only one relation, the best way of accessing S (taking selections on S, if any, into account) is recorded in *bestpla n*. This may involve using an index to identify tuples, and then fetching the tuples (often referred to as an *index scan*), or scanning the entire relation (often referred to as a *relation scan*).[1] If there is any selection condition on S, other than those ensured by an index scan, a selection operation is added to the plan, to ensure all selections on S are satisfied.

Otherwise, if S contains more than one relation, the procedure tries every way of dividing S into two disjoint subsets. For each division, the procedure recursively finds the best plans for each of the two subsets, and then computes the cost of the overall plan by using that division.[2] The procedure picks the cheapest plan from among all the alternatives for dividing S into two sets. The cheapest plan and its cost are stored in the array *bestpla n*, and returned by the procedure. The time complexity of the procedure can be shown to be $O(3^n)$ (see Practice Exercise 13.11).

Actually, the order in which tuples are generated by the join of a set of relations is also important for finding the best overall join order, since it can affect the cost of further joins (for instance, if merge join is used). A particular sort order of the tuples is said to be an **interesting sort order** if it could be useful for a later operation. For instance, generating the result of $r_1\ r_2\ r_3$ sorted on the attributes common with r_4 or r_5 may be useful, but generating it sorted on the attributes common to only r_1 and r_2 is not useful. Using merge join for computing $r_1\ r_2\ r_3$ may be costlier than using some other join technique, but it may provide an output sorted in an interesting sort order.

Hence, it is not sufficient to find the best join order for each subset of the set of n given relations. Instead, we have to find the best join order for each subset, for each interesting sort order of the join result for that subset. The number of subsets of n relations is 2^n. The number of interesting sort orders is generally not large. Thus, about 2^n join expressions need to be stored. The dynamic-programming algorithm for finding the best join order can be easily extended to handle sort orders. The cost of the extended algorithm depends on the number of interesting orders for each subset of relations; since this number has been found to be small in practice, the cost remains at $O(3^n)$. With $n = 10$, this number is around 59,000, which is much better than the 17.6 billion different join orders. More important, the storage required is much less than before, since we need to store only one join order for each interesting sort order of each of 1024 subsets of $r_1, \ldots, r_{10}$. Although both numbers still increase rapidly with n, commonly occurring joins usually have less than 10 relations, and can be handled easily.

13.4.2 Cost-Based Optimization with Equivalence Rules

The join order optimization technique we just saw handles the most common class of queries, which perform an inner join of a set of relations. However, clearly many queries use other features, such as aggregation, outer join, and nested queries, which are not addressed by join order selection.

Many optimizers follow an approach based on using heuristic transforma-tions to handle constructs other than joins, and applying the cost-based join order selection algorithm to subexpressions involving only joins and selections. Details of such heuristics are for the most part specific to individual optimizers, and we do not cover them. However, heuristic transformations to handle nested queries are widely used, and are considered in more detail in Section 13.4.4.

In this section, however, we outline how to create a general-purpose cost-based optimizer based on equivalence rules, which can handle a wide variety of query constructs.

The benefit of using equivalence rules is that it is easy to extend the optimizer with new rules to handle different query constructs. For example, nested queries can be represented using extended relational-algebra constructs, and transforma-tions of nested queries can be expressed as equivalence rules. We have already seen equivalence rules with aggregation operations, and equivalence rules can also be created for outer joins.

In Section 13.2.4, we saw how an optimizer could systematically generate all expressions equivalent to the given query. The procedure for generating equiv-alent expressions can be modified to generate all possible evaluation plans as follows: A new class of equivalence rules, called **physical equivalence rules**, is added that allows a logical operation, such as a join, to be transformed to a phys-ical operation, such as a hash join, or a nested-loops join. By adding such rules to the original set of equivalence rules, the procedure can generate all possible evaluation plans. The cost estimation techniques we have seen earlier can then be used to choose the optimal (that is, the least-cost) plan.

However, the procedure shown in Section 13.2.4 is very expensive, even if we do not consider generation of evaluation plans. To make the approach work efficiently requires the following:

A space-efficient representation of expressions that avoids making multiple copies of the same subexpressions when equivalence rules are applied.

Efficient techniques for detecting duplicate derivations of the same expression.
A form of dynamic programming based on **memoization**, which stores the optimal query evaluation plan for a subexpression when it is optimized for the first time; subsequent requests to optimize the same subexpression are handled by returning the already memoized plan.

Techniques that avoid generating all possible equivalent plans, by keeping track of the cheapest plan generated for any subexpression up to any point of time, and pruning away any plan that is more expensive than the cheapest plan found so far for that subexpression.

The details are more complex than we wish to deal with here. This approach was pioneered by the Volcano research project, and the query optimizer of SQL Server is based on this approach. See the bibliographical notes for references containing further information.

Heuristics in Optimization

A drawback of cost-based optimization is the cost of optimization itself. Although the cost of query optimization can be reduced by clever algorithms, the number of different evaluation plans for a query can be very large, and finding the optimal plan from this set requires a lot of computational effort. Hence, optimizers use **heuristics** to reduce the cost of optimization.

An example of a heuristic rule is the following rule for transforming relational-algebra queries:

Perform selection operations as early as possible. A heuristic optimizer would use this rule without finding out whether the cost is reduced by this transformation. In the first transformation example in Section 13.2, the selection operation was pushed into a join.

We say that the preceding rule is a heuristic because it usually, but not always, helps to reduce the cost. For an example of where it can result in an increase in cost, consider an expression $\mathrm{s}_{\mathrm{u}}(r\ s)$, where the condition u refers to only attributes in *s*. The selection can certainly be performed before the join. However, if *r* is extremely small compared to *s*, and if there is an index on the join attributes of *s*, but no index on the attributes used by u, then it is probably a bad idea to perform the selection early. Performing the selection early — that is, directly on *s* — would require doing a scan of all tuples in *s*. It is probably cheaper, in this case, to compute the join by using the index, and then to reject tuples that fail the selection.

The projection operation, like the selection operation, reduces the size of relations. Thus, whenever we need to generate a temporary relation, it is advan-tageous to apply immediately any projections that are possible. This advantage suggests a companion to the “perform selections early” heuristic:

UNIT-V

Distributed Databases

In a distributed database, there are a number of databases that may be geographically distributed all over the world. A distributed DBMS manages the distributed database in a manner so that it appears as one single database to users. In the later part of the chapter, we go on to study the factors that lead to distributed databases, its advantages and disadvantages.

A **distributed database** is a collection of multiple interconnected databases, which are spread physically across various locations that communicate via a computer network.

Features

- Databases in the collection are logically interrelated with each other. Often they represent a single logical database.
- Data is physically stored across multiple sites. Data in each site can be managed by a DBMS independent of the other sites.
- The processors in the sites are connected via a network. They do not have any multiprocessor configuration.
- A distributed database is not a loosely connected file system.
- A distributed database incorporates transaction processing, but it is not synonymous with a transaction processing system.

Distributed Database Management System

A distributed database management system (DDBMS) is a centralized software system that manages a distributed database in a manner as if it were all stored in a single location.

Features

- It is used to create, retrieve, update and delete distributed databases.
- It synchronizes the database periodically and provides access mechanisms by the virtue of which the distribution becomes transparent to the users.
- It ensures that the data modified at any site is universally updated.
- It is used in application areas where large volumes of data are processed and accessed by numerous users simultaneously.
- It is designed for heterogeneous database platforms.
- It maintains confidentiality and data integrity of the databases.

Factors Encouraging DDBMS

The following factors encourage moving over to DDBMS –

- **Distributed Nature of Organizational Units –** Most organizations in the current times are subdivided into multiple units that are physically distributed over the globe. Each unit requires its own set of local data. Thus, the overall database of the organization becomes distributed.
- **Need for Sharing of Data –** The multiple organizational units often need to communicate with each other and share their data and resources. This demands common databases or replicated databases that should be used in a synchronized manner.
- **Support for Both OLTP and OLAP –** Online Transaction Processing (OLTP) and Online Analytical Processing (OLAP) work upon diversified systems which may have common data. Distributed database systems aid both these processing by providing synchronized data.
- **Database Recovery –** One of the common techniques used in DDBMS is replication of data across different sites. Replication of data automatically helps in data recovery if database in any site is damaged. Users can access data from other sites while the damaged site is being reconstructed. Thus, database failure may become almost inconspicuous to users.
- **Support for Multiple Application Software –** Most organizations use a variety of application software each with its specific database support. DDBMS provides a uniform functionality for using the same data among different platforms.

Advantages of Distributed Databases

Following are the advantages of distributed databases over centralized databases.

Modular Development – If the system needs to be expanded to new locations or new units, in centralized database systems, the action requires substantial efforts and disruption in the existing functioning. However, in distributed databases, the work simply requires adding new computers and local data to the new site and finally connecting them to the distributed system, with no interruption in current functions.

More Reliable – In case of database failures, the total system of centralized databases comes to a halt. However, in distributed systems, when a component fails, the functioning of the system continues may be at a reduced performance. Hence DDBMS is more reliable.

Better Response – If data is distributed in an efficient manner, then user requests can be met from local data itself, thus providing faster response. On the other hand, in centralized systems, all queries have to pass through the central computer for processing, which increases the response time.

Lower Communication Cost – In distributed database systems, if data is located locally where it is mostly used, then the communication costs for data manipulation can be minimized. This is not feasible in centralized systems.

Adversities of Distributed Databases

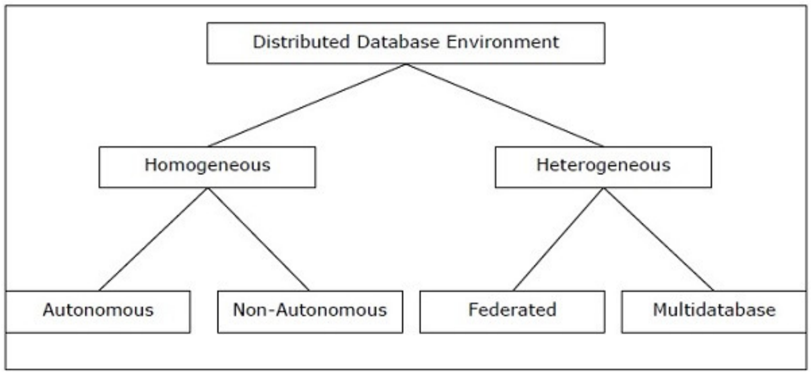
Following are some of the adversities associated with distributed databases.

- **Need for complex and expensive software –** DDBMS demands complex and often expensive software to provide data transparency and co-ordination across the several sites.
- **Processing overhead –** Even simple operations may require a large number of communications and additional calculations to provide uniformity in data across the sites.
- **Data integrity –** The need for updating data in multiple sites pose problems of data integrity.
- **Overheads for improper data distribution –** Responsiveness of queries is largely dependent upon proper data distribution. Improper data distribution often leads to very slow response to user requests.

In this part of the tutorial, we will study the different aspects that aid in designing distributed database environments. This chapter starts with the types of distributed databases. Distributed databases can be classified into homogeneous and heterogeneous databases having further divisions. The next section of this chapter discusses the distributed architectures namely client – server, peer – to – peer and multi – DBMS. Finally, the different design alternatives like replication and fragmentation are introduced.

<u>Types of Distributed Databases</u>

Distributed databases can be broadly classified into homogeneous and heterogeneous distributed database environments, each with further sub-divisions, as shown in the following illustration.



Homogeneous Distributed Databases

In a homogeneous distributed database, all the sites use identical DBMS and operating systems. Its properties are –

- The sites use very similar software.
- The sites use identical DBMS or DBMS from the same vendor.
- Each site is aware of all other sites and cooperates with other sites to process user requests.
- The database is accessed through a single interface as if it is a single database.

Types of Homogeneous Distributed Database

There are two types of homogeneous distributed database –

- **Autonomous –** Each database is independent that functions on its own. They are integrated by a controlling application and use message passing to share data updates.

- **Non-autonomous –** Data is distributed across the homogeneous nodes and a central or master DBMS co-ordinates data updates across the sites.

Heterogeneous Distributed Databases

In a heterogeneous distributed database, different sites have different operating systems, DBMS products and data models. Its properties are –

- Different sites use dissimilar schemas and software.
- The system may be composed of a variety of DBMSs like relational, network, hierarchical or object oriented.
- Query processing is complex due to dissimilar schemas.
- Transaction processing is complex due to dissimilar software.
- A site may not be aware of other sites and so there is limited co-operation in processing user requests.

Types of Heterogeneous Distributed Databases

- **Federated –** The heterogeneous database systems are independent in nature and integrated together so that they function as a single database system.
- **Un-federated –** The database systems employ a central coordinating module through which the databases are accessed.

Distributed Data Storage

1.Data Replication

Data replication is the process of storing separate copies of the database at two or more sites. It is a popular fault tolerance technique of distributed databases.

Advantages of Data Replication

- **Reliability –** In case of failure of any site, the database system continues to work since a copy is available at another site(s).
- **Reduction in Network Load –** Since local copies of data are available, query processing can be done with reduced network usage, particularly during prime hours. Data updating can be done at non-prime hours.
- **Quicker Response –** Availability of local copies of data ensures quick query processing and consequently quick response time.
- **Simpler Transactions –** Transactions require less number of joins of tables located at different sites and minimal coordination across the network. Thus, they become simpler in nature.

Disadvantages of Data Replication

- **Increased Storage Requirements –** Maintaining multiple copies of data is associated with increased storage costs. The storage space required is in multiples of the storage required for a centralized system.
- **Increased Cost and Complexity of Data Updating –** Each time a data item is updated, the update needs to be reflected in all the copies of the data at the different sites. This requires complex synchronization techniques and protocols.
- **Undesirable Application – Database coupling –** If complex update mechanisms are not used, removing data inconsistency requires complex co-ordination at application level. This results in undesirable application – database coupling.

Some commonly used replication techniques are –

- Snapshot replication
- Near-real-time replication
- Pull replication

2. Fragmentation

Fragmentation is the task of dividing a table into a set of smaller tables. The subsets of the table are called **fragments**. Fragmentation can be of three types: horizontal, vertical, and hybrid (combination of horizontal and vertical). Horizontal fragmentation can further be classified into two techniques: primary horizontal fragmentation and derived horizontal fragmentation.

Fragmentation should be done in a way so that the original table can be reconstructed from the fragments. This is needed so that the original table can be reconstructed from the fragments whenever required. This requirement is called "reconstructiveness."

Advantages of Fragmentation

- Since data is stored close to the site of usage, efficiency of the database system is increased.
- Local query optimization techniques are sufficient for most queries since data is locally available.
- Since irrelevant data is not available at the sites, security and privacy of the database system can be maintained.

Disadvantages of Fragmentation

- When data from different fragments are required, the access speeds may be very high.
- In case of recursive fragmentations, the job of reconstruction will need expensive techniques.
- Lack of back-up copies of data in different sites may render the database ineffective in case of failure of a site.

Vertical Fragmentation

In vertical fragmentation, the fields or columns of a table are grouped into fragments. In order to maintain reconstructiveness, each fragment should contain the primary key field(s) of the table. Vertical fragmentation can be used to enforce privacy of data.

For example, let us consider that a University database keeps records of all registered students in a Student table having the following schema.

STUDENT

Regd_No	Name	Course	Address	Semester	Fees	Marks

Now, the fees details are maintained in the accounts section. In this case, the designer will fragment the database as follows –

```
CREATE TABLE STD_FEES AS
  SELECT Regd_No, Fees
  FROM STUDENT;
```

Horizontal Fragmentation

Horizontal fragmentation groups the tuples of a table in accordance to values of one or more fields. Horizontal fragmentation should also confirm to the rule of reconstructiveness. Each horizontal fragment must have all columns of the original base table.

For example, in the student schema, if the details of all students of Computer Science Course needs to be maintained at the School of Computer Science, then the designer will horizontally fragment the database as follows –

```
CREATE COMP_STD AS
  SELECT * FROM STUDENT
  WHERE COURSE = "Computer Science";
```

Hybrid Fragmentation

In hybrid fragmentation, a combination of horizontal and vertical fragmentation techniques are used. This is the most flexible fragmentation technique since it generates fragments with minimal extraneous information. However, reconstruction of the original table is often an expensive task.

Hybrid fragmentation can be done in two alternative ways –

- At first, generate a set of horizontal fragments; then generate vertical fragments from one or more of the horizontal fragments.
- At first, generate a set of vertical fragments; then generate horizontal fragments from one or more of the vertical fragments.

3. Transparency

Distribution transparency is the property of distributed databases by the virtue of which the internal details of the distribution are hidden from the users. The DDBMS designer may choose to fragment tables, replicate the fragments and store them at different sites. However, since users are oblivious of these details, they find the distributed database easy to use like any centralized database.

The three dimensions of distribution transparency are –

- Location transparency
- Fragmentation transparency
- Replication transparency

Location Transparency

Location transparency ensures that the user can query on any table(s) or fragment(s) of a table as if they were stored locally in the user's site. The fact that the table or its fragments are stored at remote site in the distributed database system, should be completely oblivious to the end user. The address of the remote site(s) and the access mechanisms are completely hidden.

In order to incorporate location transparency, DDBMS should have access to updated and accurate data dictionary and DDBMS directory which contains the details of locations of data.

Fragmentation Transparency

Fragmentation transparency enables users to query upon any table as if it were unfragmented. Thus, it hides the fact that the table the user is querying on is actually a fragment or union of some fragments. It also conceals the fact that the fragments are located at diverse sites.

This is somewhat similar to users of SQL views, where the user may not know that they are using a view of a table instead of the table itself.

Replication Transparency

Replication transparency ensures that replication of databases are hidden from the users. It enables users to query upon a table as if only a single copy of the table exists.

Replication transparency is associated with concurrency transparency and failure transparency. Whenever a user updates a data item, the update is reflected in all the copies of the table. However, this operation should not be known to the user. This is concurrency transparency. Also, in case of failure of a site, the user can still proceed with his queries using replicated copies without any knowledge of failure. This is failure transparency.

Combination of Transparencies

In any distributed database system, the designer should ensure that all the stated transparencies are maintained to a considerable extent. The designer may choose to fragment tables, replicate them and store them at different sites; all oblivious to the end user. However, complete distribution transparency is a tough task and requires considerable design efforts.

<u>Distributed Transactions</u>

Access to the various data items in a distributed system is usually accomplished through transactions, which must preserve the ACID properties.

There are two types of transaction that we need to consider.

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

The failure of one of these sites, or the failure of a communication link connecting these sites, may result in erroneous computations.

System Structure

Each site has its own *local* transaction manager, whose function is to ensure the ACID properties of those transactions that execute at that site. The various trans-action managers cooperate to execute global transactions. To understand how such a manager can be implemented, consider an abstract model of a transaction system, in which each site contains two subsystems:

The **transaction manager** manages the execution of those transactions (or subtransactions) that access data stored in a local site. Note that each such transaction may be either a local transaction (that is, a transaction that exe-cutes at only that site) or part of a global transaction (that is, a transaction that executes at several sites).

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

The overall system architecture appears in Figure 1.

The structure of a transaction manager is similar in many respects to the structure of a centralized system. Each transaction manager is responsible for:

Maintaining a log for recovery purposes.

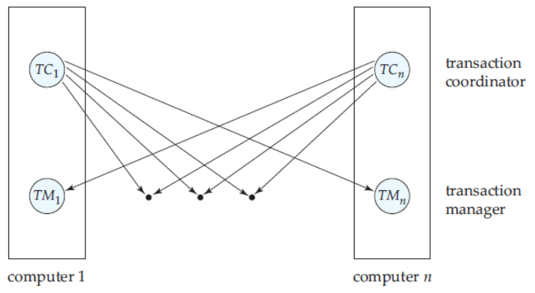


Figure 1 System architecture.

Participating in an appropriate concurrency-control scheme to coordinate the concurrent execution of the transactions executing at that site.

As we shall see, we need to modify both the recovery and concurrency schemes to accommodate the distribution of transactions.

The transaction coordinator subsystem is not needed in the centralized en-vironment, since a transaction accesses data at only a single site. A transaction coordinator, as its name implies, is responsible for coordinating the execution of all the transactions initiated at that site. For each such transaction, the coordinator is responsible for:

Starting the execution of the transaction.

Breaking the transaction into a number of subtransactions and distributing these subtransactions to the appropriate sites for execution.

Coordinating the termination of the transaction, which may result in the transaction being committed at all sites or aborted at all sites.

System Failure Modes

Distributed system may suffer from the same types of failure that a centralized system does (for example, software errors, hardware errors, or disk crashes). There are, however, additional types of failure with which we need to deal in a distributed environment. The basic failure types are:

Failure of a site.

Loss of messages

Failure of a communication link.

Network partition.

The loss or corruption of messages is always a possibility in a distributed system. The system uses transmission-control protocols, such as TCP/IP, to handle such errors. Information about such protocols may be found in standard textbooks on networking (see the bibliographical notes).

However, if two sites *A* and *B* are not directly connected, messages from one to the other must be *routed* through a sequence of communication links. If a communication link fails, messages that would have been transmitted across the link must be rerouted. In some cases, it is possible to find another route through the network, so that the messages are able to reach their destination. In other cases, a failure may result in there being no connection between some pairs of sites. A system is **partitioned** if it has been split into two (or more) subsystems, called **partitions**, that lack any connection between them. Note that, under this definition, a partition may consist of a single node.

Distributed DBMS - Controlling Concurrency

Concurrency controlling techniques ensure that multiple transactions are executed simultaneously while maintaining the ACID properties of the transactions and serializability in the schedules.

Locking Based Concurrency Control Protocols

Locking-based concurrency control protocols use the concept of locking data items. A **lock** is a variable associated with a data item that determines whether read/write operations can be performed on that data item. Generally, a lock compatibility matrix is used which states whether a data item can be locked by two transactions at the same time.

Locking-based concurrency control systems can use either one-phase or two-phase locking protocols.

One-phase Locking Protocol

In this method, each transaction locks an item before use and releases the lock as soon as it has finished using it. This locking method provides for maximum concurrency but does not always enforce serializability.

Two-phase Locking Protocol

In this method, all locking operations precede the first lock-release or unlock operation. The transaction comprise of two phases. In the first phase, a transaction only acquires all the locks it needs and do not release any lock. This is called the expanding or the **growing phase**. In the second phase, the transaction releases the locks and cannot request any new locks. This is called the **shrinking phase**.

Every transaction that follows two-phase locking protocol is guaranteed to be serializable. However, this approach provides low parallelism between two conflicting transactions.

Timestamp Concurrency Control Algorithms

Timestamp-based concurrency control algorithms use a transaction's timestamp to coordinate concurrent access to a data item to ensure serializability. A timestamp is a unique identifier given by DBMS to a transaction that represents the transaction's start time.

These algorithms ensure that transactions commit in the order dictated by their timestamps. An older transaction should commit before a younger transaction, since the older transaction enters the system before the younger one.

Timestamp-based concurrency control techniques generate serializable schedules such that the equivalent serial schedule is arranged in order of the age of the participating transactions.

Some of timestamp based concurrency control algorithms are –

- Basic timestamp ordering algorithm.
- Conservative timestamp ordering algorithm.
- Multiversion algorithm based upon timestamp ordering.

Timestamp based ordering follow three rules to enforce serializability –

- **Access Rule –** When two transactions try to access the same data item simultaneously, for conflicting operations, priority is given to the older transaction. This causes the younger transaction to wait for the older transaction to commit first.
- **Late Transaction Rule –** If a younger transaction has written a data item, then an older transaction is not allowed to read or write that data item. This rule prevents the older transaction from committing after the younger transaction has already committed.
- **Younger Transaction Rule –** A younger transaction can read or write a data item that has already been written by an older transaction.

Optimistic Concurrency Control Algorithm

In systems with low conflict rates, the task of validating every transaction for serializability may lower performance. In these cases, the test for serializability is postponed to just before commit. Since the conflict rate is low, the probability of aborting transactions which are not serializable is also low. This approach is called optimistic concurrency control technique.

In this approach, a transaction's life cycle is divided into the following three phases –

- **Execution Phase –** A transaction fetches data items to memory and performs operations upon them.

- **Validation Phase –** A transaction performs checks to ensure that committing its changes to the database passes serializability test.
- **Commit Phase –** A transaction writes back modified data item in memory to the disk.

This algorithm uses three rules to enforce serializability in validation phase –

Rule 1 – Given two transactions T_i and T_j, if T_i is reading the data item which T_j is writing, then T_i's execution phase cannot overlap with T_j's commit phase. T_j can commit only after T_i has finished execution.

Rule 2 – Given two transactions T_i and T_j, if T_i is writing the data item that T_j is reading, then T_i's commit phase cannot overlap with T_j's execution phase. T_j can start executing only after T_i has already committed.

Rule 3 – Given two transactions T_i and T_j, if T_i is writing the data item which T_j is also writing, then T_i's commit phase cannot overlap with T_j's commit phase. T_j can start to commit only after T_i has already committed.

Concurrency Control in Distributed Systems

In this section, we will see how the above techniques are implemented in a distributed database system.

Distributed Two-phase Locking Algorithm

The basic principle of distributed two-phase locking is same as the basic two-phase locking protocol. However, in a distributed system there are sites designated as lock managers. A lock manager controls lock acquisition requests from transaction monitors. In order to enforce co-ordination between the lock managers in various sites, at least one site is given the authority to see all transactions and detect lock conflicts.

Depending upon the number of sites who can detect lock conflicts, distributed two-phase locking approaches can be of three types –

- **Centralized two-phase locking –** In this approach, one site is designated as the central lock manager. All the sites in the environment know the location of the central lock manager and obtain lock from it during transactions.
- **Primary copy two-phase locking –** In this approach, a number of sites are designated as lock control centers. Each of these sites has the responsibility of managing a defined set of locks. All the sites know which lock control center is responsible for managing lock of which data table/fragment item.
- **Distributed two-phase locking –** In this approach, there are a number of lock managers, where each lock manager controls locks of data items stored at its local site. The location of the lock manager is based upon data distribution and replication.

Distributed Timestamp Concurrency Control

In a centralized system, timestamp of any transaction is determined by the physical clock reading. But, in a distributed system, any site's local physical/logical clock readings cannot be used as global timestamps, since they are not globally unique. So, a timestamp comprises of a combination of site ID and that site's clock reading.

For implementing timestamp ordering algorithms, each site has a scheduler that maintains a separate queue for each transaction manager. During transaction, a transaction manager sends a lock request to the site's scheduler. The scheduler puts the request to the corresponding queue in increasing timestamp order. Requests are processed from the front of the queues in the order of their timestamps, i.e. the oldest first.

Conflict Graphs

Another method is to create conflict graphs. For this transaction classes are defined. A transaction class contains two set of data items called read set and write set. A transaction belongs to a particular class if the transaction's read set is a subset of the class' read set and the transaction's write set is a subset of the class' write set. In the read phase, each transaction issues its read requests for the data items in its read set. In the write phase, each transaction issues its write requests.

A conflict graph is created for the classes to which active transactions belong. This contains a set of vertical, horizontal, and diagonal edges. A vertical edge connects two nodes within a class and denotes conflicts within the class. A horizontal edge connects two nodes across two classes and denotes a write-write conflict among different classes. A diagonal edge connects two nodes across two classes and denotes a write-read or a read-write conflict among two classes.

The conflict graphs are analyzed to ascertain whether two transactions within the same class or across two different classes can be run in parallel.

Distributed Optimistic Concurrency Control Algorithm

Distributed optimistic concurrency control algorithm extends optimistic concurrency control algorithm. For this extension, two rules are applied –

Rule 1 – According to this rule, a transaction must be validated locally at all sites when it executes. If a transaction is found to be invalid at any site, it is aborted. Local validation guarantees that the transaction maintains serializability at the sites where it has been executed. After a transaction passes local validation test, it is globally validated.

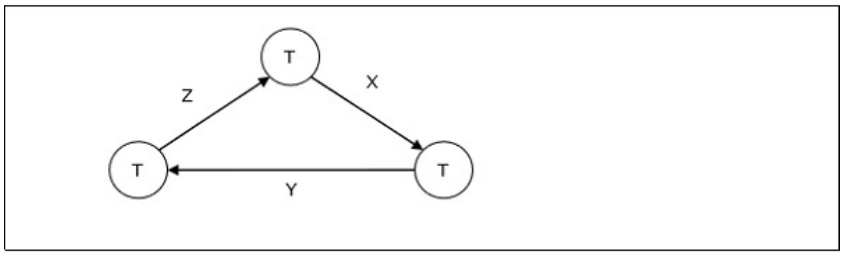
Rule 2 – According to this rule, after a transaction passes local validation test, it should be globally validated. Global validation ensures that if two conflicting transactions run together at more than one site, they should commit in the same relative order at all the sites they run together. This may require a transaction to wait for the other conflicting transaction, after validation before commit. This requirement makes the algorithm less optimistic since a transaction may not be able to commit as soon as it is validated at a site.

Distributed DBMS - Deadlock Handling

What are Deadlocks?

Deadlock is a state of a database system having two or more transactions, when each transaction is waiting for a data item that is being locked by some other transaction. A deadlock can be indicated by a cycle in the wait-for-graph. This is a directed graph in which the vertices denote transactions and the edges denote waits for data items.

For example, in the following wait-for-graph, transaction T1 is waiting for data item X which is locked by T3. T3 is waiting for Y which is locked by T2 and T2 is waiting for Z which is locked by T1. Hence, a waiting cycle is formed, and none of the transactions can proceed executing.



Deadlock Handling in Centralized Systems

There are three classical approaches for deadlock handling, namely –

- Deadlock prevention.
- Deadlock avoidance.
- Deadlock detection and removal.

All of the three approaches can be incorporated in both a centralized and a distributed database system.

Deadlock Prevention

The deadlock prevention approach does not allow any transaction to acquire locks that will lead to deadlocks. The convention is that when more than one transactions request for locking the same data item, only one of them is granted the lock.

One of the most popular deadlock prevention methods is pre-acquisition of all the locks. In this method, a transaction acquires all the locks before starting to execute and retains the locks for the entire duration of transaction. If another transaction needs any of the already acquired locks, it has to wait until all the locks it needs are available. Using this approach, the system is prevented from being deadlocked since none of the waiting transactions are holding any lock.

Deadlock Avoidance

The deadlock avoidance approach handles deadlocks before they occur. It analyzes the transactions and the locks to determine whether or not waiting leads to a deadlock.

The method can be briefly stated as follows. Transactions start executing and request data items that they need to lock. The lock manager checks whether the lock is available. If it is available, the lock manager allocates the data item and the transaction acquires the lock. However, if the item is locked by some other transaction in incompatible mode, the lock manager runs an algorithm to test whether keeping the transaction in waiting state will cause a deadlock or not. Accordingly, the algorithm decides whether the transaction can wait or one of the transactions should be aborted.

There are two algorithms for this purpose, namely **wait-die** and **wound-wait**. Let us assume that there are two transactions, T1 and T2, where T1 tries to lock a data item which is already locked by T2. The algorithms are as follows –

- **Wait-Die –** If T1 is older than T2, T1 is allowed to wait. Otherwise, if T1 is younger than T2, T1 is aborted and later restarted.
- **Wound-Wait –** If T1 is older than T2, T2 is aborted and later restarted. Otherwise, if T1 is younger than T2, T1 is allowed to wait.

Deadlock Detection and Removal

The deadlock detection and removal approach runs a deadlock detection algorithm periodically and removes deadlock in case there is one. It does not check for deadlock when a transaction places a request for a lock. When a transaction requests a lock, the lock manager checks whether it is available. If it is available, the transaction is allowed to lock the data item; otherwise the transaction is allowed to wait.

Since there are no precautions while granting lock requests, some of the transactions may be deadlocked. To detect deadlocks, the lock manager periodically checks if the wait-forgraph has cycles. If the system is deadlocked, the lock manager chooses a victim transaction from each cycle. The victim is aborted and rolled back; and then restarted later. Some of the methods used for victim selection are –

- Choose the youngest transaction.
- Choose the transaction with fewest data items.
- Choose the transaction that has performed least number of updates.
- Choose the transaction having least restart overhead.
- Choose the transaction which is common to two or more cycles.

This approach is primarily suited for systems having transactions low and where fast response to lock requests is needed.

Deadlock Handling in Distributed Systems

Transaction processing in a distributed database system is also distributed, i.e. the same transaction may be processing at more than one site. The two main deadlock handling concerns in a distributed database system that are not present in a centralized system are **transaction location** and **transaction control**. Once these concerns are addressed, deadlocks are handled through any of deadlock prevention, deadlock avoidance or deadlock detection and removal.

Transaction Location

Transactions in a distributed database system are processed in multiple sites and use data items in multiple sites. The amount of data processing is not uniformly distributed among these sites. The time period of processing also varies. Thus the same transaction may be active at some sites and inactive at others. When two conflicting transactions are located in a site, it may happen that one of them is in inactive state. This condition does not arise in a centralized system. This concern is called transaction location issue.

This concern may be addressed by Daisy Chain model. In this model, a transaction carries certain details when it moves from one site to another. Some of the details are the list of tables required, the list of sites required, the list of visited tables and sites, the list of tables and sites that are yet to be visited and the list of acquired locks with types. After a transaction terminates by either commit or abort, the information should be sent to all the concerned sites.

Transaction Control

Transaction control is concerned with designating and controlling the sites required for processing a transaction in a distributed database system. There are many options regarding the choice of where to process the transaction and how to designate the center of control, like –

- One server may be selected as the center of control.
- The center of control may travel from one server to another.
- The responsibility of controlling may be shared by a number of servers.

Distributed Deadlock Prevention

Just like in centralized deadlock prevention, in distributed deadlock prevention approach, a transaction should acquire all the locks before starting to execute. This prevents deadlocks.

The site where the transaction enters is designated as the controlling site. The controlling site sends messages to the sites where the data items are located to lock the items. Then it waits for confirmation. When all the sites have confirmed that they have locked the data items, transaction starts. If any site or communication link fails, the transaction has to wait until they have been repaired.

Though the implementation is simple, this approach has some drawbacks –

- Pre-acquisition of locks requires a long time for communication delays. This increases the time required for transaction.
- In case of site or link failure, a transaction has to wait for a long time so that the sites recover. Meanwhile, in the running sites, the items are locked. This may prevent other transactions from executing.
- If the controlling site fails, it cannot communicate with the other sites. These sites continue to keep the locked data items in their locked state, thus resulting in blocking.

Distributed Deadlock Avoidance

As in centralized system, distributed deadlock avoidance handles deadlock prior to occurrence. Additionally, in distributed systems, transaction location and transaction control issues needs to be addressed. Due to the distributed nature of the transaction, the following conflicts may occur –

- Conflict between two transactions in the same site.
- Conflict between two transactions in different sites.

In case of conflict, one of the transactions may be aborted or allowed to wait as per distributed wait-die or distributed wound-wait algorithms.

Let us assume that there are two transactions, T1 and T2. T1 arrives at Site P and tries to lock a data item which is already locked by T2 at that site. Hence, there is a conflict at Site P. The algorithms are as follows –

- **Distributed Wound-Die**
  - If T1 is older than T2, T1 is allowed to wait. T1 can resume execution after Site P receives a message that T2 has either committed or aborted successfully at all sites.
  - If T1 is younger than T2, T1 is aborted. The concurrency control at Site P sends a message to all sites where T1 has visited to abort T1. The controlling site notifies the user when T1 has been successfully aborted in all the sites.
- **Distributed Wait-Wait**
  - If T1 is older than T2, T2 needs to be aborted. If T2 is active at Site P, Site P aborts and rolls back T2 and then broadcasts this message to other relevant sites. If T2 has left Site P but is active at Site Q, Site P broadcasts that T2 has been aborted; Site L then aborts and rolls back T2 and sends this message to all sites.
  - If T1 is younger than T1, T1 is allowed to wait. T1 can resume execution after Site P receives a message that T2 has completed processing.

Distributed Deadlock Detection

Just like centralized deadlock detection approach, deadlocks are allowed to occur and are removed if detected. The system does not perform any checks when a transaction places a lock request. For implementation, global wait-for-graphs are created. Existence of a cycle in the global wait-for-graph indicates deadlocks. However, it is difficult to spot deadlocks since transaction waits for resources across the network.

Alternatively, deadlock detection algorithms can use timers. Each transaction is associated with a timer which is set to a time period in which a transaction is expected to finish. If a transaction does not finish within this time period, the timer goes off, indicating a possible deadlock.

Another tool used for deadlock handling is a deadlock detector. In a centralized system, there is one deadlock detector. In a distributed system, there can be more than one deadlock detectors. A deadlock detector can find deadlocks for the sites under its control. There are three alternatives for deadlock detection in a distributed system, namely.

- **Centralized Deadlock Detector –** One site is designated as the central deadlock detector.
- **Hierarchical Deadlock Detector –** A number of deadlock detectors are arranged in hierarchy.

- **Distributed Deadlock Detector –** All the sites participate in detecting deadlocks and removing them.

Distributed DBMS - Commit Protocols

In local database system, for committing a transaction, the transaction manager has to only convey the decision to commit to the recovery manager. However, in a distributed system, the transaction manager should convey the decision to commit to all the servers in the various sites where the transaction is being executed and uniformly enforce the decision. When processing is complete at each site, it reaches the partially committed transaction state and waits for all other transactions to reach their partially committed states. When it receives the message that all the sites are ready to commit, it starts to commit. In a distributed system, either all sites commit or none of them does.

The different distributed commit protocols are –

- One-phase commit
- Two-phase commit
- Three-phase commit

Distributed One-phase Commit

Distributed one-phase commit is the simplest commit protocol. Let us consider that there is a controlling site and a number of slave sites where the transaction is being executed. The steps in distributed commit are –

- After each slave has locally completed its transaction, it sends a “DONE” message to the controlling site.
- The slaves wait for “Commit” or “Abort” message from the controlling site. This waiting time is called **window of vulnerability**.
- When the controlling site receives “DONE” message from each slave, it makes a decision to commit or abort. This is called the commit point. Then, it sends this message to all the slaves.
- On receiving this message, a slave either commits or aborts and then sends an acknowledgement message to the controlling site.

Distributed Two-phase Commit

Distributed two-phase commit reduces the vulnerability of one-phase commit protocols. The steps performed in the two phases are as follows –

Phase 1: Prepare Phase

- After each slave has locally completed its transaction, it sends a "DONE" message to the controlling site. When the controlling site has received "DONE" message from all slaves, it sends a "Prepare" message to the slaves.
- The slaves vote on whether they still want to commit or not. If a slave wants to commit, it sends a "Ready" message.
- A slave that does not want to commit sends a "Not Ready" message. This may happen when the slave has conflicting concurrent transactions or there is a timeout.

Phase 2: Commit/Abort Phase

- After the controlling site has received "Ready" message from all the slaves –
  - The controlling site sends a "Global Commit" message to the slaves.
  - The slaves apply the transaction and send a "Commit ACK" message to the controlling site.
  - When the controlling site receives "Commit ACK" message from all the slaves, it considers the transaction as committed.
- After the controlling site has received the first "Not Ready" message from any slave –
  - The controlling site sends a "Global Abort" message to the slaves.
  - The slaves abort the transaction and send a "Abort ACK" message to the controlling site.
  - When the controlling site receives "Abort ACK" message from all the slaves, it considers the transaction as aborted.

Distributed Three-phase Commit

The steps in distributed three-phase commit are as follows –

Phase 1: Prepare Phase

The steps are same as in distributed two-phase commit.

Phase 2: Prepare to Commit Phase

- The controlling site issues an "Enter Prepared State" broadcast message.
- The slave sites vote "OK" in response.

Phase 3: Commit / Abort Phase

The steps are same as two-phase commit except that "Commit ACK"/"Abort ACK" message is not required.

Availability

One of the goals in using distributed databases is **high availability**; that is, the database must function almost all the time. In particular, since failures are more likely in large distributed systems, a distributed database must continue func-tioning even when there are various types of failures. The ability to continue functioning even during failures is referred to as **robustness**.

For a distributed system to be robust, it must *detect* failures, *reconfigure* the system so that computation may continue, and *recover* when a processor or a link is repaired.

The different types of failures are handled in different ways. For example, message loss is handled by retransmission. Repeated retransmission of a message across a link, without receipt of an acknowledgment, is usually a symptom of a link failure. The network usually attempts to find an alternative route for the message. Failure to find such a route is usually a symptom of network partition.

It is generally not possible, however, to differentiate clearly between site failure and network partition. The system can usually detect that a failure has occurred, but it may not be able to identify the type of failure. For example, suppose that site S_1 is not able to communicate with S_2. It could be that S_2 has failed. However, another possibility is that the link between S_1 and S_2 has failed, resulting in network partition. The problem is partly addressed by using multiple links between sites, so that even if one link fails the sites will remain connected. However, multiple link failure can still occur, so there are situations where we cannot be sure whether a site failure or network partition has occurred.

Suppose that site S_1 has discovered that a failure has occurred. It must then initiate a procedure that will allow the system to reconfigure, and to continue with the normal mode of operation.

If transactions were active at a failed/inaccessible site at the time of the failure, these transactions should be aborted. It is desirable to abort such transactions promptly, since they may hold locks on data at sites that are still active; waiting for the failed/inaccessible site to become accessible again may impede other transactions at sites that are operational. However, in some cases, when data objects are replicated it may be possible to proceed with reads and updates even though some replicas are inaccessible. In this case, when a failed site recovers, if it had replicas of any data object, it must obtain the current values of these data objects, and must ensure that it receives all future updates. We address this issue in Section 19.6.1.

If replicated data are stored at a failed/inaccessible site, the catalog should be updated so that queries do not reference the copy at the failed site. When a site rejoins, care must be taken to ensure that data at the site are consistent.

If a failed site is a central server for some subsystem, an *election* must be held to determine the new server (see Section 19.6.5). Examples of central servers include a name server, a concurrency coordinator, or a global deadlock detector.

Since it is, in general, not possible to distinguish between network link failures and site failures, any reconfiguration scheme must be designed to work correctly in case of a partitioning of the network. In particular, these situations must be avoided to ensure consistency:

Two or more central servers are elected in distinct partitions.

More than one partition updates a replicated data item.

Distributed Query Processing:

When a query is placed, it is at first scanned, parsed and validated. An internal representation of the query is then created such as a query tree or a query graph. Then alternative execution strategies are devised for retrieving results from the database tables. The process of choosing the most appropriate execution strategy for query processing is called query optimization.

Query Optimization Issues in DDBMS

In DDBMS, query optimization is a crucial task. The complexity is high since number of alternative strategies may increase exponentially due to the following factors –

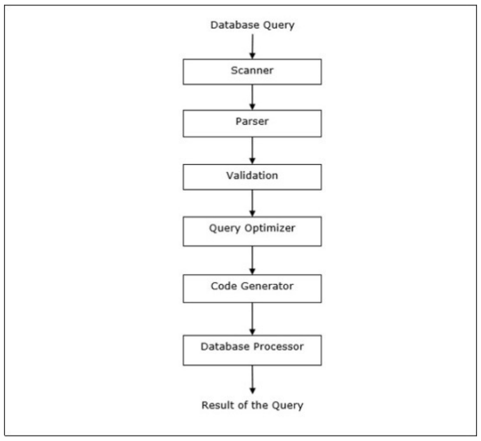
- The presence of a number of fragments.
- Distribution of the fragments or tables across various sites.
- The speed of communication links.
- Disparity in local processing capabilities.

Hence, in a distributed system, the target is often to find a good execution strategy for query processing rather than the best one. The time to execute a query is the sum of the following –

- Time to communicate queries to databases.
- Time to execute local query fragments.
- Time to assemble data from different sites.
- Time to display results to the application.

Query Processing

Query processing is a set of all activities starting from query placement to displaying the results of the query. The steps are as shown in the following diagram –



Relational Algebra

Relational algebra defines the basic set of operations of relational database model. A sequence of relational algebra operations forms a relational algebra expression. The result of this expression represents the result of a database query.

The basic operations are –

- Projection
- Selection
- Union
- Intersection
- Minus
- Join

Projection

Projection operation displays a subset of fields of a table. This gives a vertical partition of the table.

Syntax in Relational Algebra

$$\pi_{<{AttributeList}>}{(<{Table Name}>)}$$

For example, let us consider the following Student database –

STUDENT				
Roll_No	**Name**	**Course**	**Semester**	**Gender**
2	Amit Prasad	BCA	1	Male
4	Varsha Tiwari	BCA	1	Female
5	Asif Ali	MCA	2	Male
6	Joe Wallace	MCA	1	Male
8	Shivani Iyengar	BCA	1	Female

If we want to display the names and courses of all students, we will use the following relational algebra expression –

$$\pi_{Name,Course}{(STUDENT)}$$

Selection

Selection operation displays a subset of tuples of a table that satisfies certain conditions. This gives a horizontal partition of the table.

Syntax in Relational Algebra

$$\sigma_{<{Conditions}>}{(<{Table Name}>)}$$

For example, in the Student table, if we want to display the details of all students who have opted for MCA course, we will use the following relational algebra expression –

$$\sigma_{Course} = {\small "BCA"}^{(STUDENT)}$$

Combination of Projection and Selection Operations

For most queries, we need a combination of projection and selection operations. There are two ways to write these expressions –

- Using sequence of projection and selection operations.
- Using rename operation to generate intermediate results.

For example, to display names of all female students of the BCA course –

- Relational algebra expression using sequence of projection and selection operations

$$\pi_{Name}(\sigma_{Gender = \small "Female" AND \: Course = \small "BCA"}{(STUDENT)})$$

- Relational algebra expression using rename operation to generate intermediate results

$$FemaleBCAStudent \leftarrow \sigma_{Gender = \small "Female" AND \: Course = \small "BCA"}{(STUDENT)}$$

$$Result \leftarrow \pi_{Name}{(FemaleBCAStudent)}$$

Union

If P is a result of an operation and Q is a result of another operation, the union of P and Q ($p \cup Q$) is the set of all tuples that is either in P or in Q or in both without duplicates.

For example, to display all students who are either in Semester 1 or are in BCA course –

$$Sem1Student \leftarrow \sigma_{Semester = 1}{(STUDENT)}$$

$$BCAStudent \leftarrow \sigma_{Course = \small "BCA"}{(STUDENT)}$$

$$Result \leftarrow Sem1Student \cup BCAStudent$$

Intersection

If P is a result of an operation and Q is a result of another operation, the intersection of P and Q ($p \cap Q$) is the set of all tuples that are in P and Q both.

For example, given the following two schemas –

EMPLOYEE

EmpID	Name	City	Department	Salary

PROJECT

PId	City	Department	Status

To display the names of all cities where a project is located and also an employee resides –

$$CityEmp \leftarrow \pi_{City}{(EMPLOYEE)}$$

$$CityProject \leftarrow \pi_{City}{(PROJECT)}$$

$$Result \leftarrow CityEmp \cap CityProject$$

Minus

If P is a result of an operation and Q is a result of another operation, P - Q is the set of all tuples that are in P and not in Q.

For example, to list all the departments which do not have an ongoing project (projects with status = ongoing) –

$$AllDept \leftarrow \pi_{Department}{(EMPLOYEE)}$$

$$ProjectDept \leftarrow \pi_{Department} (\sigma_{Status = \small "ongoing"}{(PROJECT)})$$

$$Result \leftarrow AllDept - ProjectDept$$

Join

Join operation combines related tuples of two different tables (results of queries) into a single table.

For example, consider two schemas, Customer and Branch in a Bank database as follows –

CUSTOMER

CustID	AccNo	TypeOfAc	BranchID	DateOfOpening

BRANCH

BranchID	BranchName	IFSCcode	Address

To list the employee details along with branch details –

$$Result \leftarrow CUSTOMER \bowtie_{Customer.BranchID=Branch.BranchID}{BRANCH}$$

Translating SQL Queries into Relational Algebra

SQL queries are translated into equivalent relational algebra expressions before optimization. A query is at first decomposed into smaller query blocks. These blocks are translated to equivalent relational algebra expressions. Optimization includes optimization of each block and then optimization of the query as a whole.

Examples

Let us consider the following schemas –

EMPLOYEE

EmpID	Name	City	Department	Salary

PROJECT

PId	City	Department	Status

WORKS

EmpID	PID	Hours

Example 1

To display the details of all employees who earn a salary LESS than the average salary, we write the SQL query –

```
SELECT * FROM EMPLOYEE
WHERE SALARY < ( SELECT AVERAGE(SALARY) FROM EMPLOYEE ) ;
```

This query contains one nested sub-query. So, this can be broken down into two blocks.

The inner block is –

```
SELECT AVERAGE(SALARY)FROM EMPLOYEE ;
```

If the result of this query is AvgSal, then outer block is –

```
SELECT * FROM EMPLOYEE WHERE SALARY < AvgSal;
```

Relational algebra expression for inner block –

$$AvgSal \leftarrow \Im_{AVERAGE(Salary)}{EMPLOYEE}$$

Relational algebra expression for outer block –

$$\sigma_{Salary <{AvgSal}>}{EMPLOYEE}$$

Example 2

To display the project ID and status of all projects of employee 'Arun Kumar', we write the SQL query –

```
SELECT PID, STATUS FROM PROJECT
WHERE PID = ( SELECT FROM WORKS  WHERE EMPID = ( SELECT EMPID FROM EMPLOYEE
         WHERE NAME = 'ARUN KUMAR'));
```

This query contains two nested sub-queries. Thus, can be broken down into three blocks, as follows –

```
SELECT EMPID FROM EMPLOYEE WHERE NAME = 'ARUN KUMAR';
SELECT PID FROM WORKS WHERE EMPID = ArunEmpID;
SELECT PID, STATUS FROM PROJECT WHERE PID = ArunPID;
```

(Here ArunEmpID and ArunPID are the results of inner queries)

Relational algebra expressions for the three blocks are –

$$ArunEmpID \leftarrow \pi_{EmpID}(\sigma_{Name = \small "Arun Kumar"} {(EMPLOYEE)})$$

$$ArunPID \leftarrow \pi_{PID}(\sigma_{EmpID = \small "ArunEmpID"} {(WORKS)})$$

$$Result \leftarrow \pi_{PID, Status}(\sigma_{PID = \small "ArunPID"} {(PROJECT)})$$

Computation of Relational Algebra Operators

The computation of relational algebra operators can be done in many different ways, and each alternative is called an **access path**.

The computation alternative depends upon three main factors –

- Operator type
- Available memory
- Disk structures

The time to perform execution of a relational algebra operation is the sum of –

- Time to process the tuples.
- Time to fetch the tuples of the table from disk to memory.

Since the time to process a tuple is very much smaller than the time to fetch the tuple from the storage, particularly in a distributed system, disk access is very often considered as the metric for calculating cost of relational expression.

Computation of Selection

Computation of selection operation depends upon the complexity of the selection condition and the availability of indexes on the attributes of the table.

Following are the computation alternatives depending upon the indexes –

- **No Index –** If the table is unsorted and has no indexes, then the selection process involves scanning all the disk blocks of the table. Each block is brought into the memory and each tuple in the block is examined to see whether it satisfies the selection condition. If the condition is satisfied, it is displayed as output. This is the costliest approach since each tuple is brought into memory and each tuple is processed.
- **B+ Tree Index –** Most database systems are built upon the B+ Tree index. If the selection condition is based upon the field, which is the key of this B+ Tree index, then this index is used for retrieving results. However, processing selection statements with complex conditions may involve a larger number of disk block accesses and in some cases complete scanning of the table.
- **Hash Index –** If hash indexes are used and its key field is used in the selection condition, then retrieving tuples using the hash index becomes a simple process. A hash index uses a hash function to find the address of a bucket where the key value corresponding to the hash value is stored. In order to find a key value in the index, the hash function is executed and the bucket address is found. The key values in the bucket are searched. If a match is found, the actual tuple is fetched from the disk block into the memory.

Computation of Joins

When we want to join two tables, say P and Q, each tuple in P has to be compared with each tuple in Q to test if the join condition is satisfied. If the condition is satisfied, the corresponding tuples are concatenated, eliminating duplicate fields and appended to the result relation. Consequently, this is the most expensive operation.

The common approaches for computing joins are –

Nested-loop Approach

This is the conventional join approach. It can be illustrated through the following pseudocode (Tables P and Q, with tuples tuple_p and tuple_q and joining attribute a) –

```
For each tuple_p in P
For each tuple_q in Q
If tuple_p.a = tuple_q.a Then
```

```
  Concatenate tuple_p and tuple_q and append to Result
End If
Next tuple_q
Next tuple-p
```

Sort-merge Approach

In this approach, the two tables are individually sorted based upon the joining attribute and then the sorted tables are merged. External sorting techniques are adopted since the number of records is very high and cannot be accommodated in the memory. Once the individual tables are sorted, one page each of the sorted tables are brought to the memory, merged based upon the joining attribute and the joined tuples are written out.

Hash-join Approach

This approach comprises of two phases: partitioning phase and probing phase. In partitioning phase, the tables P and Q are broken into two sets of disjoint partitions. A common hash function is decided upon. This hash function is used to assign tuples to partitions. In the probing phase, tuples in a partition of P are compared with the tuples of corresponding partition of Q. If they match, then they are written out.

Heterogeneous Distributed Databases

Many new database applications require data from a variety of preexisting databases located in a heterogeneous collection of hardware and software en-vironments. Manipulation of information located in a heterogeneous distributed database requires an additional software layer on top of existing database sys-tems. This software layer is called a **multidatabase system**. The local database systems may employ different logical models and data-definition and data-manipulation languages, and may differ in their concurrency-control and trans-action-management mechanisms. A multidatabase system creates the illusion of logical database integration without requiring physical database integration.

Full integration of heterogeneous systems into a homogeneous distributed database is often difficult or impossible:

Technical difficulties. The investment in application programs based on ex-isting database systems may be huge, and the cost of converting these appli-cations may be prohibitive.

Organizational difficulties. Even if integration is *technically* possible, it may not be *politically* possible, because the existing database systems belong to different corporations or organizations. In such cases, it is important for a multidatabase system to allow the local database systems to retain a high degree of **autonomy** over the local database and transactions running against that data.

For these reasons, multidatabase systems offer significant advantages that outweigh their overhead. In this section, we provide an overview of the challenges faced in constructing a multidatabase environment from the standpoint of data definition and query processing.

1. **Unified View of Data**

Each local database management system may use a different data model. For instance, some may employ the relational model, whereas others may employ older data models, such as the network model (see Appendix D) or the hierarchical model .

Since the multidatabase system is supposed to provide the illusion of a single, integrated database system, a common data model must be used. A commonly used choice is the relational model, with SQL as the common query language. Indeed, there are several systems available today that allow SQL queries to a nonrelational database-management system.

2. **Query Processing**

Query processing in a heterogeneous database can be complicated. Some of the issues are:

Given a query on a global schema, the query may have to be translated into queries on local schemas at each of the sites where the query has to be executed. The query results have to be translated back into the global schema.

The task is simplified by writing **wrappers** for each data source, which provide a view of the local data in the global schema. Wrappers also translate queries on the global schema into queries on the local schema, and translate results back into the global schema. Wrappers may be provided by individual sites, or may be written separately as part of the multidatabase system.

Wrappers can even be used to provide a relational view of nonrelational data sources, such as Web pages (possibly with forms interfaces), flat files, hierarchical and network databases, and directory systems.

Some data sources may provide only limited query capabilities; for instance, they may support selections, but not joins. They may even restrict the form of selections, allowing selections only on certain fields; Web data sources with form interfaces are an example of such data sources. Queries may therefore have to be broken up, to be partly performed at the data source and partly at the site issuing the query.

In general, more than one site may need to be accessed to answer a given query. Answers retrieved from the sites may have to be processed to remove duplicates. Suppose one site contains *account* tuples satisfying the selection *ba la nce* < 100, while another contains *account* tuples satisfying *ba la nce* > 50. A query on the entire account relation would require access to both sites and removal of duplicate answers resulting from tuples with balance between 50 and 100, which are replicated at both sites.

Global query optimization in a heterogeneous database is difficult, since the query execution system may not know what the costs are of alternative query plans at different sites. The usual solution is to rely on only local-level optimization, and just use heuristics at the global level.

3. **Transaction Management in Multidatabases**

A multidatabase system supports two types of transactions:

Local transactions. These transactions are executed by each local database system outside of the multidatabase system's control.

Global transactions. These transactions are executed under the multidata-base system's control.

The multidatabase system is aware of the fact that local transactions may run at the local sites, but it is not aware of what specific transactions are being executed, or of what data they may access.

Cloud-Based Databases

Cloud computing is a relatively new concept in computing that emerged in the late 1990s and the 2000s, first under the name *software as a service*. Initial vendors of software services provided specific customizable applications that they hosted on their own machines. The concept of cloud computing developed as vendors began to offer generic computers as a service on which clients could run software applications of their choosing. A client can make arrangements with a cloud-computing vendor to obtain a certain number of machines of a certain capacity as well as a certain amount of data storage. Both the number of machines and the amount of storage can grow and shrink as needed. In addition to providing computing services, many vendors also provide other services such as data storage services, map services, and other services that can be accessed using a Web-service application programming interface.

Many enterprises are finding the model of cloud computing and services beneficial. It saves client enterprises the need to maintain a large system-support staff and allows new enterprises to begin operation without having to make a large, up-front capital investment in computing systems. Further, as the needs of the enterprise grow, more resources (computing and storage) can be added as required; the cloud-computing vendor generally has very large clusters of computers, making it easy for the vendor to allocate resources on demand.

1. Data Storage Systems on the Cloud

Applications on the Web have extremely high scalability requirements. Popular applications have hundreds of millions of users, and many applications have seen their load increase manyfold within a single year, or even within a few months. To handle the data management needs of such applications, data must be partitioned across thousands of processors.

A number of systems for data storage on the cloud have been developed and deployed over the past few years to address data management requirements of such applications; these include *Bigtable* from Google, *Simple Storage Service* (*S3*) from Amazon, which provides a Web interface to *Dynamo*, which is a key-value storage system, *Cassandra*, from FaceBook, which is similar to Bigtable, and *Sherpa*/*PNUTS* from Yahoo!, the data storage component of the *Azure* environment from Microsoft, and several other systems.

1.1.Data Representation

As an example of data management needs of Web applications, consider the pro-file of a user, which needs to be accessible to a number of different applications that are run by an organization. The profile contains a variety of attributes, and there are frequent additions to the attributes stored in the profile. Some attributes may contain complex data. A simple relational representation is often not sufficient for such complex data.

Some cloud-based data-storage systems support XML (described in Chap-ter 23) for representing such complex data. Others support the **JavaScript Object Notation** (**JSON**) representation, which has found increasing acceptance for repre-senting complex data. The XML and JSON representations provide flexibility in the set of attributes that a record contains, as well as the types of these attributes. Yet others, such as Bigtable, define their own data model for complex data including support for records with a very large number of optional columns. We revisit the Bigtable data model later in this section.

1.2.Partitioning and Retrieving Data

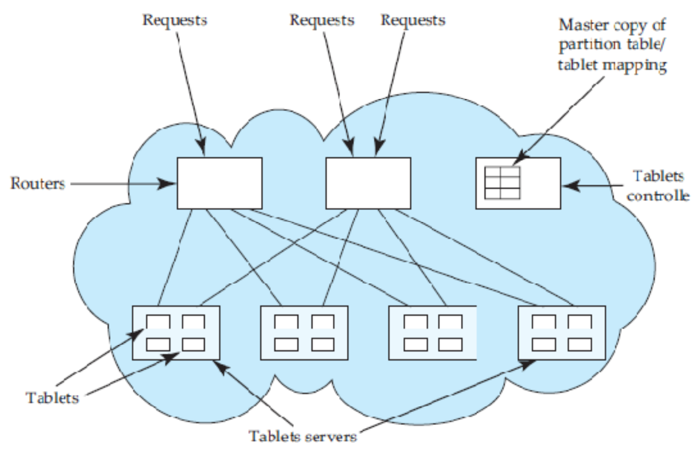
Partitioning of data is, of course, the key to handling extremely large scale in data-storage systems. Unlike regular parallel databases, it is usually not possible to decide on a partitioning function ahead of time. Further, if load increases, more servers need to be added and each server should be able to take on parts of the load incrementally.

To solve both these problems, data-storage systems typically partition data into relatively small units (small on such systems may mean of the order of hundreds of megabytes). These partitions are often called **tablets**, reflecting the fact that each tablet is a fragment of a table. The partitioning of data should be done on the search key, so that a request for a specific key value is directed to a single tablet; otherwise each request would require processing at multiple sites, increasing the load on the system greatly. Two approaches are used: either range partitioning is used directly on the key, or a hash function is applied on the key, and range partitioning is applied on the result of the hash function.

The site to which a tablet is assigned acts as the master site for that tablet. All updates are routed through this site, and updates are then propagated to replicas of the tablet. Lookups are also sent to the same site, so that reads are consistent with writes.

The partitioning of data into tablets is not fixed up front, but happens dy-namically. As data are inserted, if a tablet grows too big, it is broken into smaller parts. Further, even if a tablet is not large enough to merit being broken up, if the load (get/put operations) on that tablet are excessive, the tablet may be broken into smaller tablets, which can be distributed across two or more sites to share the load. Usually the number of tablets is much larger than the number of sites, for the same reason that virtual partitioning is used in parallel databases.

It is important to know which site in the overall system is responsible for a particular tablet. This can be done by having a tablet controller site which tracks the partitioning function, to map a get() request to one or more tablets, and a mapping function from tablets to sites, to find which site were responsible for which tablet. Each request coming into the system must be routed to the correct site; if a single tablet controller site is responsible for this task, it would soon get overloaded. Instead, the mapping



information can be replicated on a set of router sites, which route requests to the site with the appropriate tablet. Protocols to update mapping information when a tablet is split or moved are designed in such a way that no locking is used; a request may as a result end up at a wrong site. The problem is handled by detecting that the site is no longer responsible for the key specified by the request, and rerouting the request based on up-to-date mapping information.

The above Figure depicts the architecture of a cloud data-storage system, based loosely on the PNUTS architecture. Other systems provide similar functionality, although their architecture may vary. For example, Bigtable does not have sepa-rate routers; the partitioning and tablet-server mapping information is stored in the Google file system, and clients read the information from the file system, and decide where to send their requests.

2. Traditional Databases on the Cloud

We now consider the issue of implementing a traditional distributed database system, supporting ACID properties and queries, on a cloud.

The concept of computing utilities is an old one, envisioned back in the 1960s. The first manifestation of the concept was in timesharing systems in which several users shared access to a single mainframe computer. Later, in the late 1960s, the concept of **virtual machines** was developed, in which a user was given the illusion of having a private computer, while in reality a single computer simulated several virtual machines.

3. Challenges with Cloud-Based Databases

Cloud-based databases certainly have several important advantages compared to building a computing infrastructure from scratch, and are in fact essential for certain applications.

However, cloud-based database systems also have several disadvantages that we shall now explore. Unlike purely computational applications in which parallel computations run largely independently, distributed database systems require frequent communication and coordination among sites for:

access to data on another physical machine, either because the data are owned by another virtual machine or because the data are stored on a storage server separate from the computer hosting the virtual machine.

obtaining locks on remote data.

ensuring atomic transaction commit via two-phase commit.

In our earlier study of distributed databases, we assumed (implicitly) that the database administrator had control over the physical location of data. In a cloud system, the physical location of data is under the control of the vendor, not the client. As a result, the physical placement of data may be suboptimal in terms of communication cost, and this may result in a large number of remote lock requests and large transfers of data across virtual machines. Effective query optimization requires that the optimizer have accurate cost measures for opera-tions. Lacking knowledge of the physical placement of data, the optimizer has to rely on estimates that may be highly inaccurate, resulting in poor execution strategies. Because remote accesses are relatively slow compared to local access, these issues can have a significant impact on performance.

Directory Systems

Consider an organization that wishes to make data about its employees avail-able to a variety of people in the organization; examples of the kinds of data include name, designation, employee-id, address, email address, phone number, fax number, and so on. In the precomputerization days, organizations would cre-ate physical directories of employees and distribute them across the organization. Even today, telephone companies create physical directories of customers.

In general, a directory is a listing of information about some class of objects such as persons. Directories can be used to find information about a specific object, or in the reverse direction to find objects that meet a certain requirement. In the world of physical telephone directories, directories that satisfy lookups in the forward direction are called **white pages**, while directories that satisfy lookups in the reverse direction are called **yellow pages**.

In today's networked world, the need for directories is still present and, if anything, even more important. However, directories today need to be available over a computer network, rather than in a physical (paper) form.

1. Directory Access Protocols

Directory information can be made available through Web interfaces, as many organizations, and phone companies in particular, do. Such interfaces are good for humans. However, programs too need to access directory information. Direc-tories can be used for storing other types of information, much like file system directories. For instance, Web browsers can store personal bookmarks and other browser settings in a directory system. A user can thus access the same settings from multiple locations, such as at home and at work, without having to share a file system.

Several **directory access protocols** have been developed to provide a stan-dardized way of accessing data in a directory. The most widely used among them today is the **Lightweight Directory Access Protocol** (**LDAP**).

Obviously all the types of data in our examples can be stored without much trouble in a database system, and accessed through protocols such as JDBC or ODBC. The question then is, why come up with a specialized protocol for accessing directory information? There are at least two answers to the question.

First, directory access protocols are simplified protocols that cater to a limited type of access to data. They evolved in parallel with the database access protocols.

Second, and more important, directory systems provide a simple mecha-nism to name objects in a hierarchical fashion, similar to file system directory names, which can be used in a distributed directory system to specify what information is stored in each of the directory servers. For example, a partic-ular directory server may store information for Bell Laboratories employees in Murray Hill, while another may store information for Bell Laboratories employees in Bangalore, giving both sites autonomy in controlling their lo-cal data. The directory access protocol can be used to obtain data from both directories across a network. More important, the directory system can be set up to automatically forward queries made at one site to the other site, without user intervention.

For these reasons, several organizations have directory systems to make or-ganizational information available online through a directory access protocol. Information in an organizational directory can be used for a variety of purposes, such as to find addresses, phone numbers, or email addresses of people, to find which departments people are in, and to track department hierarchies. Directories are also used to authenticate users: applications can collect authentication infor-mation such as passwords from users and authenticate them using the directory.

As may be expected, several directory implementations find it beneficial to use relational databases to store data, instead of creating special-purpose storage systems.

2. LDAP: Lightweight Directory Access Protocol

In general a directory system is implemented as one or more servers, which service multiple clients. Clients use the application programmer interface defined by the directory system to communicate with the directory servers. Directory access protocols also define a data model and access control.

The **X.500 directory access protocol**, defined by the International Organiza-tion for Standardization (ISO), is a standard for accessing directory information. However, the protocol is rather complex, and is not widely used. The **Lightweight Directory Access Protocol** (**LDAP**) provides many of the X.500 features, but with less complexity, and is widely used. In the rest of this section, we shall outline the data model and access protocol details of LDAP.

2.1.LDAP Data Model

In LDAP, directories store **entries**, which are similar to objects. Each entry must have a **distinguished name (DN)**, which uniquely identifies the entry. A DN is in turn made up of a sequence of **relative distinguished names (RDNs)**. For example, an entry may have the following distinguished name:

cn=Silberschatz, ou=Computer Science, o=Yale University, c=USA

As you can see, the distinguished name in this example is a combination of a name and (organizational) address, starting with a person's name, then giving the organizational unit (ou), the organization (o), and country (c). The order of the components of a distinguished name reflects the normal postal address order, rather than the reverse order used in specifying path names for files. The set of RDNs for a DN is defined by the schema of the directory system.

Entries can also have attributes. LDAP provides binary, string, and time types, and additionally the types tel for telephone numbers, and PostalAddress for addresses (lines separated by a "$" character). Unlike those in the relational model, attributes are multivalued by default, so it is possible to store multiple telephone numbers or addresses for an entry.

LDAP allows the definition of **object classes** with attribute names and types. Inheritance can be used in defining object classes. Moreover, entries can be spec-ified to be of one or more object classes. It is not necessary that there be a single most-specific object class to which an entry belongs.

Entries are organized into a **directory information tree (DIT)**, according to their distinguished names. Entries at the leaf level of the tree usually represent specific objects. Entries that are internal nodes represent objects such as orga-nizational units, organizations, or countries. The children of a node have a DN containing all the RDNs of the parent, and one or more additional RDNs. For in-stance, an internal node may have a DN c=USA, and all entries below it have the value USA for the RDN c.

The entire distinguished name need not be stored in an entry. The system can generate the distinguished name of an entry by traversing up the DIT from the entry, collecting the RDN=value components to create the full distinguished name.

Entries may have more than one distinguished name — for example, an entry for a person in more than one organization. To deal with such cases, the leaf level of a DIT can be an **alias**, which points to an entry in another branch of the tree.

2.2.Data Manipulation

Unlike SQL, LDAP does not define either a data-definition language or a data-manipulation language. However, LDAP defines a network protocol for carrying out data definition and manipulation. Users of LDAP can either use an application programming interface or use tools provided by various vendors to perform data definition and manipulation. LDAP also defines a file format called **LDAP Data Interchange Format (LDIF)** that can be used for storing and exchanging information.

The querying mechanism in LDAP is very simple, consisting of just selections and projections, without any join. A query must specify the following:

A base — that is, a node within a DIT — by giving its distinguished name (the path from the root to the node).

A search condition, which can be a Boolean combination of conditions on individual attributes. Equality, matching by wild-card characters, and ap-proximate equality (the exact definition of approximate equality is system dependent) are supported.

A scope, which can be just the base, the base and its children, or the entire subtree beneath the base.

Attributes to return.

Limits on number of results and resource consumption.

The query can also specify whether to automatically dereference aliases; if alias dereferences are turned off, alias entries can be returned as answers.

One way of querying an LDAP data source is by using LDAP URLs. Examples of LDAP URLs are:

ldap:://codex.cs.yale.edu/o=YaleUniversity,c=USA **ldap:://codex.cs.yale.edu/o=Yale University,c=USA??sub?cn=Silberschatz**

The first URL returns all attributes of all entries at the server with organization being Yale University, and country being USA. The second URL executes a search query (selection) cn=Silberschatz on the subtree of the node with distinguished name o=Yale University, c=USA. The question marks in the URL separate different fields. The first field is the distinguished name, here o=Yale University,c=USA. The second field, the list of attributes to return, is left empty, meaning return all attributes. The third attribute, sub, indicates that the entire subtree is to be searched. The last parameter is the search condition.

A second way of querying an LDAP directory is by using an application programming interface. Figure 19.8 shows a piece of C code used to connect to an LDAP server and run a query against the server. The code first opens a connection to an LDAP server by ldap open and ldap bind. It then executes a query by ldap search s. The arguments to ldap search s are the LDAP connection handle, the DN of the base from which the search should be done, the scope of the search, the search condition, the list of attributes to be returned, and an attribute called attrsonly, which, if set to 1, would result in only the schema of the result being returned, without any actual tuples. The last argument is an output argument that returns the result of the search as an LDAPMessage structure.

2.3.Distributed Directory Trees

Information about an organization may be split into multiple DITs, each of which stores information about some entries. The **suffix** of a DIT is a sequence of RDN=value pairs that identify what information the DIT stores; the pairs are con-catenated to the rest of the distinguished name generated by traversing from the entry to the root. For instance, the suffix of a DIT may be o=Lucent, c=USA, while another may have the suffix o=Lucent, c=India. The DITs may be organizationally and geographically separated.

A node in a DIT may contain a **referral** to another node in another DIT; for instance, the organizational unit Bell Labs under o=Lucent, c=USA may have its own DIT, in which case the DIT for o=Lucent, c=USA would have a node ou=Bell Labs representing a referral to the DIT for Bell Labs.

Referrals are the key component that help organize a distributed collection of directories into an integrated system. When a server gets a query on a DIT, it may return a referral to the client, which then issues a query on the referenced DIT. Access to the referenced DIT is transparent, proceeding without the user’s knowledge. Alternatively, the server itself may issue the query to the referred DIT and return the results along with locally computed results.

The hierarchical naming mechanism used by LDAP helps break up control of information across parts of an organization. The referral facility then helps integrate all the directories in an organization into a single virtual directory.

Although it is not an LDAP requirement, organizations often choose to break up information either by geography (for instance, an organization may maintain a directory for each site where the organization has a large presence) or by orga-nizational structure (for instance, each organizational unit, such as department, maintains its own directory).

Many LDAP implementations support master–slave and multimaster replication of DITs, although replication is not part of the current LDAP version 3 standard. Work on standardizing replication in LDAP is in progress.