



Module on Fundamentals of Database Systems

Table of Contents

CHAPTER ONE	1
1. Fundamental concept of Database system	1
1.1. What is database?.....	2
1.1.1. Evolution of a Database System	4
1.1.2. Landmarks in Database System History	5
1.1.3. Database System Requirements	5
1.1.4. Database System Architecture	6
1.2. Database Versus File system.....	9
1.3. What is database management system(DBMS)	9
1.4. The evolution of database management systems	10
1.5. Typical roles and career path for database professionals.....	13
1.6. Database Languages.....	17
CHAPTER TWO	21
2. Relational Data Model	21
2.1. Introduction to information models and data models	41
2.2. Types of Data models	42
2.2.1. Hierarchical model.....	42
2.2.2. Network model.....	43
2.2.3. Relational data model.....	43
2.2.4. Entity Relation model	53
2.2.5. Object-Data model	55
2.3. Difference between Hierarchical and Relational Data Model:	55
2.4. Difference between Network and Relational Data Model :	56
2.5. Relational database management system (RDBMS)	58
CHAPTER THREE	64
3. Conceptual Database Design and E-R Modeling.....	64
3.1. Introduction.....	64
3.2. Conceptual Database Design	65
3.2.1. Steps to Build Conceptual Data Model.....	65
3.2.2. Symbols Used in ER Diagram	66

3.3.	Design ER Diagram	67
3.4.	Entity-Relationship Diagram Building Blocks	68
3.5.	Mapping ER Diagram to Relational Tables	72
3.6.	Problem with ER Models.....	74
3.7.	Enhanced Entity Relationship (EER) Models.....	79
3.7.1.	Features of EER Model.....	80
CHAPTER FOUR.....		84
4.	Logical Database Design	84
4.1.	Introduction.....	84
4.2.	Logical Database Design for Relational Model.....	84
4.3.	Normalization	85
4.4.	Pitfalls of Normalization.....	92
4.5.	Denormalization.....	93
CHAPTER FIVE		94
5.	Physical Database Design	94
5.1.	What is physical database design process?	94
5.2.	Moving from Logical to Physical Design	95
5.3.	DBMS - Storage System.....	96
5.4.	DBMS - File Structure	97
5.5.	File Organization	97
5.6.	DBMS - Indexing.....	99
5.7.	DBMS - Hashing.....	102
CHAPTER SIX.....		106
6.	Query Languages	106
6.1.	Introduction.....	106
6.2.	Query Languages	107
6.3.	Relational Algebra	108
6.4.	Relational calculus	118
6.5.	Introduction to SQL	121
Reference		160

CHAPTER ONE

1. Fundamental concept of Database system

At the end of this chapter, you will find yourself being able to:

- ☒ Define what a file system is
- ☒ Define what a database system is
- ☒ Discuss the demerits of file based system
- ☒ Describe the advantages of database system
- ☒ Understand the characteristics of Database System

Know what a **DBMS** is

Data is one of the most critical assets of any business. It is used and collected practically everywhere, from businesses trying to determine consumer patterns based on credit card usage, to space agencies trying to collect data from other planets. Data, as important as it is, needs robust, secure, and highly available software that can store and process it quickly. The answer to these requirements is a solid and a reliable database.

Database software usage is pervasive, yet it is taken for granted by the billions of daily users worldwide. Its presence is everywhere-from retrieving money through an automatic teller machine to budging access at a secure office location.

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 order a magazine subscription from a publisher, chances are that our activities will involve someone 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.

The above interactions are examples of what we may call traditional database applications, where 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 (GIS) can store and analyze maps, weather data, and satellite images. Data warehouses and on-line 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 through the Internet.

1.1. What is database?

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, engineering, medicine, law, education, and library science, to name a few. The word database is in such common use that we must begin by defining a database. Our initial definition is quite general.

Since its advent, databases have been among the most researched knowledge domains in computer science. A **database** is a repository of data, designed to support efficient data storage, retrieval and maintenance. Multiple types of databases exist to suit various industry requirements. A database may be specialized to store binary files, documents, images, videos, relational data, multidimensional data, transactional data, analytic data, or geographic data to name a few.

Data can be stored in various forms, namely tabular, hierarchical and graphical forms. If data is stored in a tabular form then it is called a relational database. When data is organized in a tree structure form, it is called a hierarchical database. Data stored as graphs representing relationships between objects is referred to as a network database.

In its very simplest form, a Database can be viewed as a “repository for data” or “a collection of data.” The repository is tasked with storing, maintaining and presenting large amounts of data in

a consistent and efficient fashion to the applications, and the users of such applications.

A database is a collection of related data . By data, we mean known facts that can be recorded and that have implicit 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 diskette, using a personal computer and software such as DBASE IV or V, Microsoft ACCESS, or EXCEL. This is a collection of related data with an implicit meaning and hence is a database.

The preceding definition of database is quite general; for example, we may consider the collection of words that make up this page of text to be related data and hence to constitute a database. However, the common use of the term database is usually more restricted. A database has the following implicit properties:

- A database represents some aspect of the real world, sometimes called the miniworld or the Universe of Discourse (UoD). Changes to the mini world are reflected in the database.
- A database is a logically coherent collection of data with some inherent meaning. A random assortment of data cannot correctly be referred to as a database.
- A database is designed, built, and populated with data for a specific purpose. It has an intended group of users and some preconceived applications in which these users are interested.

In other words, a database has some source from which data are derived, some degree of interaction with events in the real world, and an audience that is actively interested in the contents of the database.

A database can be of any size and of varying complexity. For example, the list of names and addresses referred to earlier may consist of only a few hundred records, each with a simple structure. On the other hand, the card catalog of a large library may contain half a million cards stored under different categories—by primary author's last name, by subject, by book title—with each category organized in alphabetic order. A database of even greater size and complexity is maintained by the Internal Revenue Service to keep track of the tax forms filed by U.S.

taxpayers. If we assume that there are 100 million tax-payers and if each taxpayer files an average of five forms with approximately 200 characters of information per form, we would get a database of $100 \times (106) \times 200 \times 5$ characters (bytes) of information.

If the IRS keeps the past three returns for each taxpayer in addition to the current return, we would get a database of $4 \times (1011)$ bytes (400 gigabytes). This huge amount of information must be organized and managed so that users can search for, retrieve, and update the data as needed.

A database may be generated and maintained manually or it may be computerized. The library card catalog is an example of a database that may be created and maintained manually. A computerized database may be created and maintained either by a group of application programs written specifically for that task or by a database management system.

1.1.1. Evolution of a Database System

During the past three decades, the database technology for information systems has undergone four generations of evolution, and we are nearly on the fifth generation database.

- The **first generation** was file system, such as ISAM and VSAM.
- The **second generation** was hierarchical database systems, such as IMS and System 2000.
- The **third generation** was the network model Conference on Data Systems Languages (CODASYL) database systems, such as IDS, TOTAL, ADABAS, IDMS, etc. The second and third generation systems realized the sharing of an integrated database among many users within an application environment.
- The **fourth-generation** database technology, namely relational database technology arises to solve the lack of data independence and the tedious navigational access to the database in the second and third generations. Relational database technology is characterized by the notion of a declarative query.
- **Fifth-generation** database technology will be characterized by a richer data model and a richer set of database facilities necessary to meet the requirements of applications beyond the business data-processing applications for which the first four generations of database technology have been developed.

1.1.2. Landmarks in Database System History

-1950s and early 1960s: Magnetic disc into the usage of data storage. Data reading from tapes and punched cards for processing were sequential.

- **Late 1962s and 1970s:** Hard disks come into play in late 1960s and direct data access was made possible. A paper by Codd [1970] on relation model, querying and relational database brighten the database system industry.

-1980s: During the 1970s research and development activities in databases were focused on realizing the relational database technology. These efforts culminated in the introduction of commercially available systems in late 70s and early 80s, such as Oracle, SQL/DB and DB2 and INGRES that became competitive to the hierarchical and network database systems. A number of researches had also been published on distributed and parallel database system.

-Late 1990s: The WWW and multimedia advancement forces the database system for reliable and extensive operations. Moreover, the object-oriented programming languages put a strain to a unified programming and database language. The reason is that an object-oriented programming language is built on the object-oriented concepts, and object-oriented concepts consist of a number of data modeling concepts, such as aggregation, generalization, and membership relationships. An object-oriented database system which supports such a unified object-oriented programming and database language will be better platform for developing object-oriented database applications than an extended relational database system which supports an extended relational database language.

1.1.3. Database System Requirements

Databases evolved to take responsibility for the data away from the application, and most importantly to enable data to be shared. Hence a database system must provide:

- **Consistency:** It must ensure that the data itself is not only consistently stored but can be retrieved and shared efficiently.

- **Concurrency:** It must enable multiple users and systems to all retrieve the data at the same time and to do so logically and consistently.

-**Performance:** It must support reasonable response times.

- **Standard adherence:** It should support a standard language for common understanding.

Standard Query Language (SQL) has to be supported. The two categories of the SQL are:

□ *Data Definition Language (DDL):* allow users to create new databases and specify their schema.

□ *Data Manipulation Language (DML):* enables users to query and manipulate data

- **Security:** It should provide away to set access permissions (much like files at the operating system level) and specific database mechanisms such as triggers.

- **Reliability:** It must keep the stored data intact. Additionally, it must cope well when things go awry and it must, if set up properly, be able to recover to a known consistent point.

1.1.4. Database System Architecture

Centralized Database System Architecture

Centralized database systems are those that run on a single computer system and that do not interact with the other computer system except for displaying information on display terminals.

Such database systems span from single-user database system that run on a single personal computer to a high-performance database systems that run on a main frame.

Client/Server Architecture for a Database System

In the Client/Server architecture the client processes run separately from the server processes, usually on a different computer. The architecture enables to specialized servers and workstations (clients). The general structure of client/server architecture is shown below.

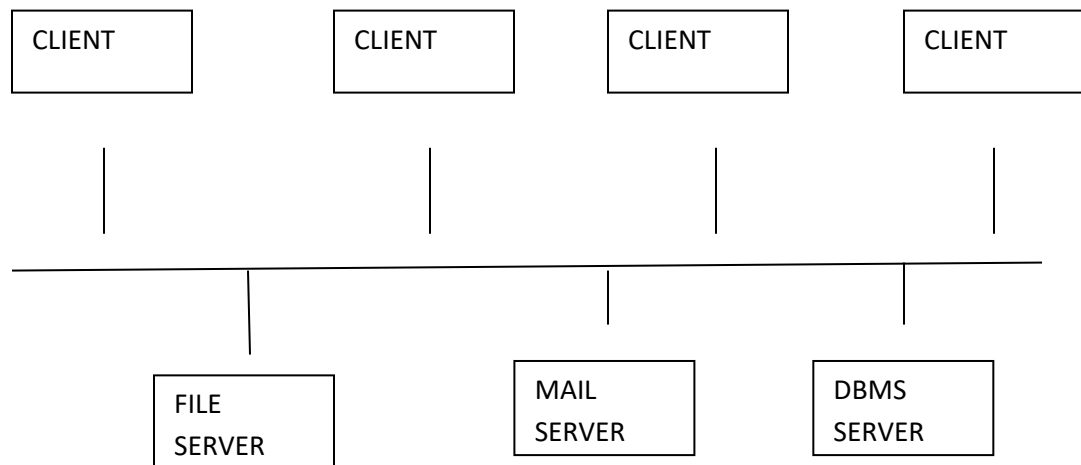


Figure 1: Structure of client/server architecture

Two-Tier Client/Server Architecture is the simplest client/server application. In this architecture the client processes provide an interface for the user, and gather and present data usually either on a screen on the user's computer or in a printed report. The server processes provide an interface with the data storage. The logic that validates data, monitors security and permissions, and performs other business rules can be fully contained on either the client or the server, or partly on the client and partly on the server. The exact division of the logic varies from system to system.

The logic for the application can also be designed to form a separate middle tier. Applications that are designed with separate middle tier have three logical tiers but still run into two physical tiers. The middle tier may be contained in either the client or the server. Client/server applications that are designed to run the user and business tiers of the application on the client side, and the data tier on the server side are known as **fat client** applications. On the other hand, applications that are designed to run the user tier on the client side and the business and data tiers on the server side are known as **thin client** applications. Though fat and thin client/server architectures have three tiers, such applications are intended to run on two computers as two physical tiers. If the three tiers are separated so that the application can be run on three separate computers, the implementation is known as a three-tier application.

Three-Tier Client/Server Architecture is an application that has three modularly separated tiers that can be run on three machines. The standard model for a three-tier application has User tier (GUI or Web Interface), Business tier (Application Server or Web Server) and Data tier (Data Server).

User tier presents the user interface for the application, displays data and collects user input. It also sends and requests for data to the next tier. It is often known as the **presentation tier**. The business tier incorporates the business rules for the application. It receives requests for data from the user tier, evaluates them against the business rules and passes them on to the data tier. It then receives data from the data tier and passes back to the user tier. It is also known as the **business logic tier**. And finally at the base, the data tier comprises the data storage and a layer that passes data from the data storage to the business tier and vice versa. It is also known as the **data tier**.

Components and Functionalities of a Database System

A database system can be partitioned into two modules as **storage manager** and **query processor**.

1. Storage manager: is a program module that provides interface between the low level data stored in the database and the application programs or queries submitted to the system. The storage manager translates the various DML statements into low level file system commands (the conventional operating system commands); this it is responsible for storing, retrieving and updating data. The main components of the storage manager are:

- **Authorization and integrity manager:** checks for credentials of the users and tests for the integrity constraints.
- **Transaction Manager:** enables to preserve consistency despite system failure and avoid conflict at the time of concurrent transaction.
- **File manager:** manages disk storage allocation and data structure for stored data.
- **Buffer manger:** is responsible for fetching data from disk storage to the main memory.

2. Query Processor: is a module that handles queries as well as requests for modification of the data and metadata. Some of the components are:

- **DDL interpreter (compiler):** processes DDL statements for schema definition (meta data) and records the definitions in the data dictionary.

- **DML compiler:** analyze, translates and optimizes DML statements in a high-level query language into an evaluation plan consisting of low-level instructions codes to the query evaluation (execution) engine.
- **Query evaluation engine:** execute low-level instructions generated by the DML compiler.

1.2. Database Versus File system

The traditional file processing system is file-directory structure supported by a conventional operating system. A file system organization of data lacks a number of major features of a database system, such as:

- **Data redundancy and inconsistency:** It is more likely that files and applications in a file system to be of different format and standards. Moreover, same information may exist in duplicate.
- **Difficulty in accessing data:** It does not support convenient and efficient responsive data-retrieval system for new request in an existing data.
- **Data isolation:** Related data may be scattered across files.
- **Integrity problems:** Maintaining constraints across files and applications would be difficult.
- **Atomicity problems:** In case of all-or-none set of operations it is crucial that, if a failure occurs the data need to be restored to its consistent state. That is the set of operations must be performed as a single unified operation.
- **Concurrent access anomalies:** Supervision of application is difficult to provide because data may be accessed by any of the programs that are not coordinated.
- **Security problems:** Adding application programs to the system in ad hoc fashion makes the system more vulnerable to security treats and attacks.

1.3. What is database management system(DBMS)

While a database is a repository of data, a database management system, or simply DBMS, is a set of software tools that control access, organize, store, manage, retrieve and maintain data in a database. In practical use, the terms database, database server, database system, data server, and database management systems are often used interchangeably.

Why do we need database software or a DBMS? Can we not just store data in simple text files

for example? The answer lies in the way users access the data and the handle of corresponding challenges. First, we need the ability to have multiple users insert, update and delete data to the same data file without "stepping on each other's toes". This means that different users will not cause the data to become inconsistent, and no data should be inadvertently lost through these operations. We also need to have a standard interface for data access, tools for data backup, data restore and recovery, and a way to handle other challenges such as the capability to work with huge volumes of data and users. Database software has been designed to handle all of these challenges.

The most mature database systems in production are relational database management systems (RDBMS's). RDBMS's serve as the backbone of applications in many industries including banking, transportation, health, and so on. The advent of Web-based interfaces has only increased the volume and breadth of use of RDBMS, which serve as the data repositories behind essentially most online commerce.

In general Database Management System (DBMS) is then a tool for creating and managing this large amounts of data efficiently and allowing it to persist for a long periods of time. Hence DBMS is a general-purpose software that facilitates the processes of defining, constructing, manipulating, and sharing database.

Defining: involves specifying data types, structure and constraints.

Constructing: is the process of storing the data into a storage media.

Manipulating: is retrieving and updating data from and into the storage.

Sharing: allows multiple users to access data.

1.4. The evolution of database management systems

In the 1960s, network and hierarchical systems such as CODASYL and IMSTM were the state-of-the-art technology for automated banking, accounting, and order processing systems enabled by the introduction of commercial mainframe computers. While these systems provided a good basis for the early systems, their basic architecture mixed the physical manipulation of data with its logical manipulation.

A revolutionary paper by E.F. Codd, an IBM San Jose Research Laboratory employee in 1970,

changed all that. The paper titled “A relational model of data for large shared data banks introduced the notion of data independence, which separated the physical representation of data from the logical representation presented to applications. Data could be moved from one part of the disk to another or stored in a different format without causing applications to be rewritten. Application developers were freed from the tedious physical details of data manipulation, and could focus instead on the logical manipulation of data in the context of their specific application.

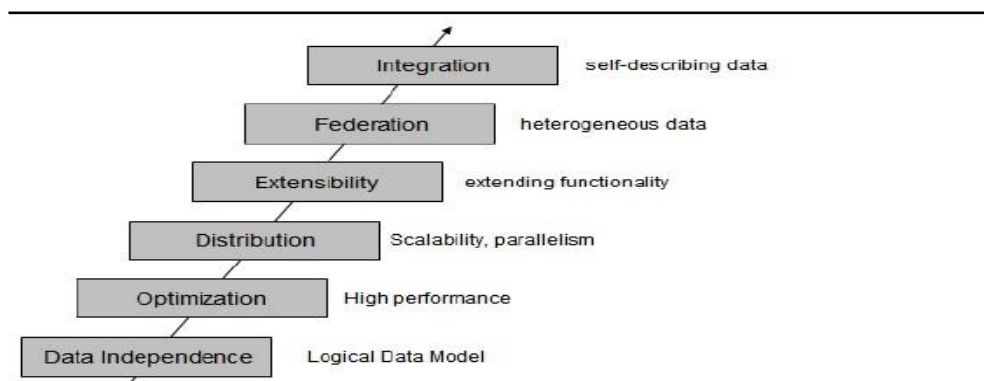


Figure 1.1 illustrates the evolution of database management systems

The above figure describes the evolution of database management systems with the relational model that provide for data independence. IBM's System R was the first system to implement Codd's ideas. System R was the basis for SQL/DS, which later became DB2. It also has the merit to introduce SQL, a relational database language used as a standard.

Today, relational database management systems are the most used DBMS's and are developed by several software companies. IBM is one of the leaders in the market with DB2 database server.

Other relational DBMS's include Oracle, Microsoft SQL Server, INGRES, PostgreSQL, MySQL, and dBASE.

As relational databases became increasingly popular, the need to deliver high performance queries has arisen. DB2's optimizer is one of the most sophisticated components of the product. From a user's perspective, you treat DB2's optimizer as a black box, and pass any SQL query to it. The DB2's optimizer will then calculate the fastest way to retrieve your data by taking into account many factors such as the speed of your CPU and disks, the amount of data available, the location of the data, the type of data, the existence of indexes, and so on. DB2's optimizer is cost-based.

As increased amounts of data were collected and stored in databases, DBMS's scaled. In DB2 for Linux, UNIX and Windows, for example, a feature called Database Partitioning Feature (DPF) allows a database to be spread across many machines using a shared-nothing architecture. Each machine added brings its own CPUs and disks; therefore, it is easier to scale almost linearly. A query in this environment is parallelized so that each machine retrieves portions of the overall result.

Next in the evolution of DBMS's is the concept of extensibility. The Structured Query Language (SQL) invented by IBM in the early 1970's has been constantly improved through the years. Even though it is a very powerful language, users are also empowered to develop their own code that can extend SQL. For example, in DB2 you can create user-defined functions, and stored procedures, which allow you to extend the SQL language with your own logic.

Then DBMS's started tackling the problem of handling different types of data and from different sources. At one point, the DB2 data server was renamed to include the term "Universal" as in "DB2 universal database" (DB2 UDB). Though this term was later dropped for simplicity reasons, it did highlight the ability that DB2 data servers can store all kinds of information including video, audio, binary data, and so on. Moreover, through the concept of federation a query could be used in DB2 to access data from other IBM products, and even non-IBM products.

Lastly, in the figure the next evolutionary step highlights integration. Today many businesses need to exchange information, and the eXtensible Markup Language (XML) is the underlying technology that is used for this purpose. XML is an extensible, self- describing language. Its usage has been growing exponentially because of Web 2.0, and service-oriented architecture (SOA). IBM recognized early the importance of XML;

therefore, it developed a technology called pureXML that is available with DB2 database servers. Through this technology, XML documents can now be stored in a DB2 database in hierarchical format (which is the format of XML). In addition, the DB2 engine was extended to natively handle XQuery, which is the language used to navigate XML documents. With pureXML, DB2 offers the best performance to handle XML, and at the same time provides the security, robustness and scalability it has delivered for relational data through the years.

The current "hot" topic at the time of writing is Cloud Computing. DB2 is well positioned to work on the Cloud. In fact, there are already DB2 images available on the Amazon EC2 cloud, and on the IBM Smart Business Development and Test on the IBM Cloud (also known as IBM Development and Test Cloud). DB2's Database Partitioning Feature previously described fits perfectly in the cloud where you can request standard nodes or servers on demand, and add them to your cluster. Data rebalancing is automatically performed by DB2 on the go. This can be very useful during the time when more power needs to be given to the database server to handle end-of-the-month or end-of-the-year transactions.

1.5. Typical roles and career path for database professionals

As people are one of the components in DBMS environment, there are group of roles played by different stakeholders of the designing and operation of a database system.

I. Database Designer

Database designers are responsible for identifying the data to be stored in the database and for choosing appropriate structures to represent and store this data. These tasks are mostly undertaken before the database is actually implemented and populated with data. It is the

responsibility of database designers to communicate with all prospective database users in order to understand their requirements and to create a design that meets these requirements. In many cases, the designers are on the staff of the DBA and may be assigned other staff responsibilities after the database design is completed.

Database designers typically interact with each potential group of users and develop views of the database that meet the data and processing requirements of these groups. Each view is then analyzed and integrated with the views of other user groups. The final database design must be capable of supporting the requirements of all user groups.

In large database design projects, we can distinguish between two types of designer: **logical** database designers and **physical** database designers. The logical database designer is concerned with identifying the data (that is, the entities and attributes), the relationships between the data, and the constraints on the data that is to be stored in the database.

The logical database designer must have a thorough and complete understanding of the organization's data and any constraints on this data (the constraints are sometimes called **business rules**). These constraints describe the main characteristics of the data as viewed by the organization. Examples of constraints for DreamHome are:

- a member of staff cannot manage more than 100 properties for rent or sale at the same time;
- a member of staff cannot handle the sale or rent of his or her own property; • a solicitor cannot act for both the buyer and seller of a property.

To be effective, the logical database designer must involve all prospective database users in the development of the data model, and this involvement should begin as early in the process as possible. In this book, we split the work of the logical database designer into two stages:

- Conceptual database design, which is independent of implementation details, such as the target DBMS, application programs, programming languages, or any other physical considerations;
- Logical database design, which targets a specific data model, such as relational, network, hierarchical, or object-oriented.

The physical database designer decides how the logical database design is to be physically realized. This involves:

- Mapping the logical database design into a set of tables and integrity constraints;
- Selecting specific storage structures and access methods for the data to achieve good performance;
- Designing any security measures required on the data.

Many parts of physical database design are highly dependent on the target DBMS, and there may be more than one way of implementing a mechanism. Consequently, the physical database designer must be fully aware of the functionality of the target DBMS and must understand the advantages and disadvantages of each alternative implementation. The physical database designer must be capable of selecting a suitable storage strategy that takes account of usage. Whereas conceptual and logical database designs are concerned with the what, physical database design is concerned with the how. It requires different skills, which are often found in different people.

II. Database Administrator

- Responsible to oversee, control and manage the database resources (the database itself, the DBMS and other related software)
- Authorizing access to the database
- Coordinating and monitoring the use of the database
- Responsible for determining and acquiring hardware and software resources
- Accountable for problems like poor security, poor performance of the system
- Involves in all steps of database development
- We can have further classifications of this role in big organizations having huge amount of data and user requirement.
- **Data Administrator (DA):** is responsible on management of data resources. This involves in database planning, development, maintenance of standards policies and procedures at the conceptual and logical design phases.
- **Database Administrator (DBA):** This is more technically oriented role. DBA is responsible for the physical realization of the database. It is involved in physical design, implementation, security and integrity control of the database.

In any organization where many people use the same resources, there is a need for a chief administrator to oversee and manage these resources. In a database environment, the primary resource is the database itself, and the secondary resource is the DBMS and related software. Administering these resources is the responsibility of the database administrator (DBA). The DBA is responsible for authorizing access to the database, coordinating and monitoring its use, and acquiring software and hardware resources as needed. The DBA is accountable for problems such as security breaches and poor system response time. In large organizations, the DBA is assisted by a staff that carries out these functions.

III. Application Developers

Once the database has been implemented, the application programs that provide the required functionality for the end-users must be implemented. This is the responsibility of the application developers. Typically, the application developers work from a specification produced by systems analysts. Each program contains statements that request the DBMS to perform some operation on the database, which includes retrieving data, inserting, updating, and deleting data. The programs may be written in a third-generation or fourth-generation programming language, as discussed previously.

IV. End-Users

End users are the people whose jobs require access to the database for querying, updating, and generating reports; the database primarily exists for their use. There are several categories of end users:

- **Casual end users** occasionally access the database, but they may need different information each time. They use a sophisticated database query interface to specify their requests and are typically middle- or high-level managers or other occasional browsers.
- **Naive or parametric end users** make up a sizable portion of database end users. Their main job function revolves around constantly querying and updating the database, using standard types of queries and updates—called canned transactions—that have been carefully programmed and tested. Many of these tasks are now available as mobile apps for use with mobile devices. The tasks that such users perform are varied. A few examples are:
 - Bank customers and tellers check account balances and post withdrawals and deposits.

- Reservation agents or customers 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.
- Social media users post and read items on social media Web sites.
- **Sophisticated end users** include engineers, scientists, business analysts, and others who thoroughly familiarize themselves with the facilities of the DBMS in order to implement their own applications to meet their complex requirements.
- **Standalone users** maintain personal databases by using ready-made program packages that provide easy-to-use menu-based or graphics-based interfaces. An example is the user of a financial software package that stores a variety of personal financial data.

A typical DBMS provides multiple facilities to access a database. Naive end users need to learn very little about the facilities provided by the DBMS; they simply have to understand the user interfaces of the mobile apps or standard transactions designed and implemented for their use. Casual users learn only a few facilities that they may use repeatedly. Sophisticated users try to learn most of the DBMS facilities in order to achieve their complex requirements.

1.6. Database Languages

A database language consists of four parts: a **Data Definition Language (DDL)**, **Data Manipulation Language (DML)**, **Transaction control language (TCL)** and **Data control language (DCL)**. The DDL is used to specify the database schema and the DML is used to both read and update the database. These languages are called data sublanguages because they do not include constructs for all computing needs such as conditional or iterative statements, which are provided by the high-level programming languages. Many DBMSs have a facility for embedding the sublanguage in a high-level programming language such as COBOL, FORTRAN, Pascal, Ada, and _C_, C++, Java, or Visual Basic. In this case, the high-level language is sometimes referred to as the host language. To compile the embedded file, the commands in the data sublanguage are first removed from the hostlanguage program and replaced by function calls. The pre-processed file is then compiled, placed in an object module, linked with a DBMS-specific library containing the replaced functions, and executed when required. Most data

sublanguages also provide non-embedded, or interactive, commands that can be input directly from a terminal.

A. The Data Definition Language (DDL)

It is a language that allows the DBA or user to describe and name the entities, attributes, and relationships required for the application, together with any associated integrity and security constraints.

The database schema is specified by a set of definitions expressed by means of a special language called a Data Definition Language. The DDL is used to define a schema or to modify an existing one. It cannot be used to manipulate data.

The result of the compilation of the DDL statements is a set of tables stored in special files collectively called the system catalog. The system catalog integrates the metadata that is data that describes objects in the database and makes it easier for those objects to be accessed or manipulated. The metadata contains definitions of records, data items, and other objects that are of interest to users or are required by the DBMS. The DBMS normally consults the system catalog before the actual data is accessed in the database. The terms data dictionary and data directory are also used to describe the system catalog, although the term ‘data dictionary’ usually refers to a more general software system than a catalog for a DBMS. Which defines the database structure or schema. Specifies additional properties or constraints of the data. The database system is checks these constraints every time the database is updated.

Example: CREATE: create object in database

ALTER: alter the structure of database

DROP: deletes object from database

RENAME: rename the object

- **The Data Manipulation Language (DML)**

It is a language that provides a set of operations to support the basic data manipulation operations on the data held in the databases. Data manipulation operations usually include the following:

- insertion of new data into the database;
- modification of data stored in the database;
- retrieval of data contained in the database; ü Deletion of data from the database.

Therefore, one of the main functions of the DBMS is to support a data manipulation language in which the user can construct statements that will cause such data manipulation to occur. Data manipulation applies to the external, conceptual, and internal levels. However, at the internal level we must define rather complex low-level procedures that allow efficient data access. In contrast, at higher levels, emphasis is placed on ease of use and effort is directed at providing efficient user interaction with the system.

- **Transaction Control language(TCL)**

It used to manage transaction in database and the change made by data manipulation language statements.

Transaction: the logical unit of work which consists of some operations to control some tasks.

Example: COMMIT: used to permanently save any transaction into the database.

ROLLBACK: restore the database to last committee state.

- **Data Control Language (DCL)**

It used to control access to data stored in database (Authorization)

Example: GRANT: allows specified users to perform specified tasks

REVOKE: cancel pervious granted or denied permission

The part of a DML that involves data retrieval is called a query language. A query language can be defined as a high-level special-purpose language used to satisfy diverse requests for the

retrieval of data held in the database. The term `_query` is therefore reserved to denote a retrieval statement expressed in a query language or specifies data to retrieve rather than how to retrieve it.

CHAPTER TWO

2. Relational Data Model

Relational Data Model is an implementation (representational) model proposed by E.F. Codd in 1970. The model is an approach in a database design towards the Relational Database Management System (RDBMS).

3.1 Structure of Relational Database

The main construct for representing data in the relational database is a two-dimensional table called a relation.

Example

- “EMPLOYEES” relation

EMPID	NAME	BDATE	SUBSITY	KEBELE	PHONE
E001	Alemu Girma	1/10/70	Bole	06	011-663-0712
E004	Kelem Belete	12/04/68	Gulele	03	011-227-2525

Fig 1. Typical Employee relation instance

The columns in the table are representing the **attributes** of the relationship, and the rows (other than the heading row) represent **tuples (records)** of the relation.

A relation in a relational model consists of:

- The **Relation schema**: - that describes the column heads for the table and
- The **Relation instance**: - that is the table with the set of tuples.

The set of relation schema forms schema for the relational database called **database schema (relational database schema)**.

In relational model the relation schema are described first. And the schema specifies

- *The relation's name*

- *Name for each attribute (field or column)*

- *Domain* of each attribute: - A domain is rereferred to in a relation schema by the domain name and has a set of associated values.

Example

- Employees (EmpId:string, Name:string, BDate:date, SubCity:string, KebeI:integer, Phone:string)
- Projects (PrjId:integer, Name:string, SDate:date, DDate:date, CDate:date)
- Teams (Name:string, Descr:string)

Properties of Relations

Rows (tuples) in a single relation are unique (that is; no two tuples are identical).

- ☐ Relations are set of tuples not lists (that is; order of tuples in a relation is immaterial).
- ☐ Attributes are atomic.
- ☐ The values that appear in a column must be drawn from the domain associated with that column.
- ☐ The **degree**, also called **arity**, of a relation is the number of attributes in the relation.
- ☐ The relation names in a relational database are distinct.

Key Constraints

A **key constraint** is a statement that a certain minimal subset of the attributes of a relation is a unique identifier for a tuple in the relation.

A set of attributes that uniquely identifies a tuple according to a key constraint is called a **candidate key** for the relation; often abbreviated just as **key**.

Key attributes in relational model are indicated by *underlying* the attributes in the relational.

Example

- Employees (EmpId, Name, BDate, SubCity, KebeI, Phone)
- Projects (PrjId, Name, SDate, DDate, CDate)
- Teams (Name, Descr)

REMARK: Note that a key for a relation may not be directly inferred from the high-level conceptual models in some cases.

Foreign Key Constraints

The most common integrity constraint involving two relations is a foreign key constraint. It keeps data consistency when a data modification is done on a relation.

The foreign key in the referencing relation requires a match to a primary key in the referenced relation. That is, there must be a compatible data type attribute in the referenced relation so as the referencing relation may make the referencing.

Example

- Employees (EmpId, Name, BDate, SubCity, Kebele, Phone)
- WorkSchedule (SDate, EDate, HoursPerDay, Employee,)

In the above example for the “WorkSchedule” to refer to the “Employees” relation instance, it has an attribute ‘Employee’ of the same type as the ‘EmpId’ in the “Employees” relation which is a primary key. The foreign key constraint is implemented through the ‘Employee’ attribute in the referencing relation “WorkSchedule”.

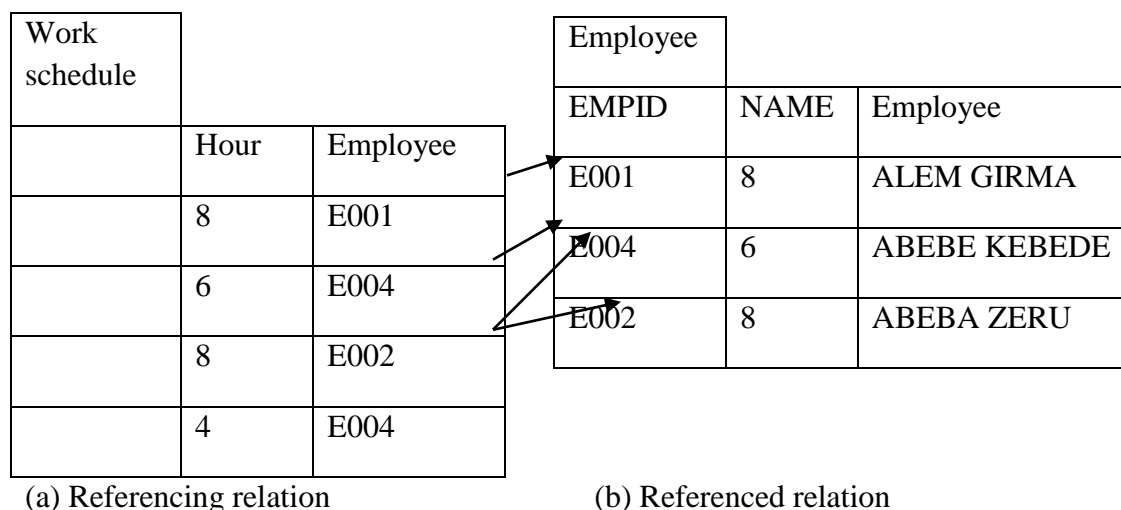


Fig 2. Foreign Constraint in Relational Model

NOTE: - A single tuple can be referenced by zero or more tuples in the referencing relation, but a single tuple with a single foreign key attribute can only reference one tuple.

- A foreign key could refer to the same relation.

- A relational database consists of related relations through a foreign key.

The second phase in database design is implementation design that transforms the conceptual data model into an internal model - schema such as a relational data model for an implementation into relational database management system (RDBMS).

E/R diagram's entity sets and relationship are ways of describing a relational schema and the sets of entities and relationship sets form the relational instance of the E/R schema which is not part of the database design.

Entity Sets to Relations

Strong entity sets in E/R model are mapped to relations in relational model with the same name and attributes. The primary keys assigned for the entity sets are also represented as keys in the relations.

Handling Weak Entity Sets

Suppose W is a weak entity set with attribute set $\{a_1, a_2, a_3, \dots a_n\}$ and identifying strong entity set E . And let the primary key of E is the set $\{b_1, b_2, \dots b_m\}$, then the attributes of the relation for the weak entity set must include attributes for its complete key (including those belonging to the identifying strong entity set) and its own, non-key attributes. That is, the set of attributes of the mapping relation is $\{a_1, a_2, a_3, \dots a_n\} \cup \{b_1, b_2, \dots b_m\}$.

The primary key for the weak entity set relation thus include:

- ☐ The *discriminator* of the weak entity set, and
- ☐ The *primary key* of the identifying strong entity set.

Handling Composite and Multivalued Attributes

- **Composite attributes** from E/R model to a relational model can be represented by creating separate attributes for each of the components of the attributes (Note that the composite attribute is not mapped directly into a separate attribute).
- **Multivalued attributes** are handled by creating relations with the name of the attribute having attributes that corresponds to the components of the multivalued attribute and the primary key of the entity set or relationship set of which the attribute belongs. The primary key for the newly created relation consists of:
 - The primary key of the entity set or relationship set, and
 - The attribute or set of attributes from the multivalued attribute.

REMARK

Note that; if the multivalued attribute has a fixed size of multiplicity (small size), it can be represented by separate attributes for each multiplicity. For example consider phone attribute above.

Relationship Sets to Relations

Suppose entity set E with a primary key $\{a11, a12, a13, \dots a1n\}$ is related to an entity set F with a primary key $\{a21, a22, a23, \dots a2m\}$ through a relationship R. Let the relationship R has a descriptive attribute set $\{b1, b2, b3, \dots bp\}$, then the relationship is represented by a relation whose attributes are:

- The keys of the connected entity sets: $\{a11, a12, a13, \dots a1n\} \cup \{a21, a22, a23, \dots a2m\}$, and
- Attributes of the relationship itself: $\{b1, b2, b3, \dots bp\}$.

The union of the primary keys of the related entity sets forms super key for the relationship relation. If the relationship is many-to-many the super key also becomes a primary key for the relation, otherwise the primary key from the many said becomes the primary key for the relation.

Suppose entity set E and F are related through a many-to-one relationship R from E to F, then it is possible to join the relations for E and R that come out of this E/R model into a single relation S with a schema consisting of:

- ☐ All attributes of the entity set E,
- ☐ The keys attributes of the entity set F, and
- ☐ All Attributes of the relationship R.

If the participation of E into R is total it is also possible to include all attributes of F in the relation S and have one single relation S in place of the three relations E, F and R.

The primary key for S would be the primary key of E.

Representation of Generalization and Specialization

Hierarchical structure (Specialization and Generalization or Inheritance) in relation model can be represented in three different ways:

1. **E/R Style:** One relation for each lower-level entity set and the higher-level entity set.

Every relation of the lower-level entity set will include:

- ☐ Key attribute(s) of the higher-level entity set which forms the primary key of the entity set, and
- ☐ Attributes of that lower-level entity set.

For total and disjoint generalization the higher-level entity set may not be mapped into a relation instead all its attributes are passed to all immediate lower-level entity sets relations.

2. **Use of Nulls:** One relation having a large set of attributes of all the lower-level entity sets and higher-level entity set; entities have NULL in attributes that don't belong to them. Involves large number of NULL values for disjoint generalization.

3. **Object-Oriented Approach:** One relation per subset of subclasses, with all relevant attributes including:

- Attributes of the higher-level entity set, and
- Attributes of that lower-level entity set.

The primary key of the higher-level entity set becomes the primary key of each relation.

Dependencies

In a database design the two most common pitfalls that result in bad designing are:

- Repetition of information, and
- Inability to present certain information (Loss of information).

□ **Functional Dependencies**

Functional dependency is a kind of constraint that helps to remove redundancy in relational database design.

Definition: Functional dependency denoted by $X \twoheadrightarrow A$ is an assertion about a relation R that whenever two tuples of R agree on all the attributes of X, then they must also agree on the attribute A. We say that “ $X \twoheadrightarrow A$ holds in R” or “X functional determines A”

Note that in the notation $X \twoheadrightarrow A$; X represent sets of attributes and A represent single attribute. That is $A_1 A_2 A_3 \dots A_n \twoheadrightarrow B$

The functional dependency is a generalization of the notion of super key.

Example:

- Consider the Teams relation: Teams(PrjId, Name, Descr), then

PrjId, Name \square Descr

- For the Employees relation:

Employees(EmpId, NationalId, Name, BDate, Age, Gender, City, HAddr, Phone)

EmpId \square Name; EmpId \square Age; Name BDate \square Gender

A functional dependency $A_1 A_2 A_3 \dots A_n \square B$ is said to be trivial dependency if B is an element or of $\{A_1, A_2, A_3 \dots A_n\}$.

Rules of Functional Dependency

Combining Rule:

The functional dependencies:

$A_1 A_2 A_3 \dots A_n \square B_1$

$A_1 A_2 A_3 \dots A_n \square B_2$

:

$A_1 A_2 A_3 \dots A_n \square B_m$

can be written as:

$A_1 A_2 A_3 \dots A_n \square B_1 B_2 \dots B_m$

Splitting Rule

The functional dependency $A_1 A_2 A_3 \dots A_n \square B_1 B_2 \dots B_m$ can be written as $A_1 A_2 A_3 \dots A_n \square B_i$ for $i=1, 2, 3, \dots m$

Closure of Attributes

Suppose $\{A_1, A_2, \dots A_n\}$ is a set of attributes and S is a set of functional dependencies in relation R. The closure of the set $\{A_1, A_2, \dots A_n\}$ under the functional dependency set S is the set of attributes B that are functionally determined from the set S. That is; $A_1 A_2 \dots A_n \square B$ follows from the set S. The closure set of attributes $A_1, A_2, \dots A_n$ is denoted by $\{A_1, A_2, \dots A_n\}_+$

The closure set of attributes can be determined by repeatedly applying the following three rules known as **Armstrong's Axioms**:

Reflexivity Rule

If α is set of attributes and $\beta \subseteq \alpha$ then, $\alpha \twoheadrightarrow \beta$ holds.

Augmentation Rule

If $\alpha \twoheadrightarrow \beta$ holds and γ is set of attributes, then $\gamma\alpha \twoheadrightarrow \gamma\beta$ holds.

Transitivity Rule

If $\alpha \twoheadrightarrow \beta$ holds and $\beta \twoheadrightarrow \gamma$ holds, then $\alpha \twoheadrightarrow \gamma$ holds.

Algorithm for computing the closure of X , X^+ is given below.

1. Let X be a set of attributes that eventually will become the closure. First, we initialize X to be X .
2. Now, we repeatedly search for some functional dependency $B_1 B_2 \dots B_m \twoheadrightarrow C$ Such that all of B_1, B_2, \dots, B_m are in the set of attributes X but C is not. We then add C to the set X .
3. Repeat step 2 as many times as necessary until no more attributes can be added to X .
4. The set X , after no more attributes can be added to it, is the closure set X^+ .

Example: Consider a relation with attributes A, B, C, D, E , and F . Suppose that this relation has the functional dependencies $AB \twoheadrightarrow C, BC \twoheadrightarrow AD, D \twoheadrightarrow E$, and $CF \twoheadrightarrow B$. What is the closure of $\{A, B\}$, that is $\{A, B\}^+$?

Solution:

$X = \{A, B\}$

From the function dependency $AB \twoheadrightarrow C$, we add C to X that is $X = \{A, B, C\}$

Similarly; $BC \twoheadrightarrow AD \twoheadrightarrow X = \{A, B, C, D\}$

$D \twoheadrightarrow E \twoheadrightarrow X = \{A, B, C, D, E\}$

No more changes in X are possible. Thus $\{A, B\}^+ = \{A, B, C, D, E\}$

From the closure set it is to follow that $AB \twoheadrightarrow D$

Exercise: Test whether $D \twoheadrightarrow A$ flows from the functional dependency set?

To test for $D \twoheadrightarrow A$, first determine the closure set of $\{D\}$

$X = \{D\}$

From the function dependency $D \twoheadrightarrow E$, we add E to X that is $X = \{D, E\}$

No more changes in X are possible. Thus $\{D\}^+ = \{D, E\}$

From the closure set $D \twoheadrightarrow A$ does not hold.

\twoheadrightarrow Multivalued Dependencies

Multivalued dependency for a relation R , is defined as a constraint when the values of one set of attributes is fixed, then the values in certain other attributes are independent of values of all the other attributes in R .

That is; for a multivalued dependency $X \twoheadrightarrow Y$ in R where X and Y are subsets of the set of attributes in R , if t and u are tuples in the relational instance r for the schema R , then there exist a third tuple v that agrees:

1. with both t and u on X 's,
2. with t on Y 's, and
3. with u on all attributes of R that are not among X 's or Y 's ($R - (X \cup Y)$).

Rules of Multivalued Dependency

Multivalued dependency is a generalization for the functional dependency. That is;

If $\alpha \twoheadrightarrow \beta$ holds, then $\alpha \twoheadrightarrow \beta$ also holds.

All the rules except the splitting rule for the functional dependency are also applicable for a multivalued dependency.

Complementation Rule

One additional rule in a multivalued dependency that does not have a counterpart in functional dependency is the complementation rule.

The rule states that if $X \twoheadrightarrow Y$ holds then $X \twoheadrightarrow (R - (X \cup Y))$, where R is a set of attributes for the relational schema R .

3.4 Normalization and Normal Forms

In relational databases, **normalization** is a process that helps to

- eliminates redundancy,
- organizes data efficiently,
- reduces the potential for anomalies during data operations, and
- improves data consistency.

The formal classifications used for quantifying "how normalized" a relational database are called **normal forms** (abbreviated as **NF**).

□ Normalization and Denormalization

Following standard database normalization recommendations when designing databases can greatly maximize a database's performance by helping to:

Reduce the total amount of redundant data in the database. The less data, the less work on the RDBMS has to perform, hence, speeding its performance.

- *Reduce the use of NULLS in the database.* The use of NULLs in a database can greatly

reduce database performance, especially in WHERE clauses.

□ *Reduce the number of columns in tables.* The less number of columns in tables, the more rows can fit on a single data page, which helps to boost read performance of the RDBMS.

Reduce the amount of SQL code. The less code there is, the less that has to run, speeding your application's performance.

□ *Maximize the use of clustered indexes.* The more data is separated into multiple tables because of normalization, the more clustered indexes become available to help speed up data access.

□ *Reduce the total number of indexes.* The less columns tables have, the less need there is for multiple indexes to retrieve it. And the fewer indexes, the less negative is the performance effect of data insertion, modification and deletion.

Redundancy in a database design results in data anomalies classified as:

- Insertion Anomalies
- Deletion Anomalies
- Modification Anomalies

Table 4.1 shows an example of data redundancy where college information and student information are stored together. Under this design we store the same college information a number of times, once for each student record from a particular college. For example, the information that IIT college has a level of 1 is repeated twice, one for student George Smith, the other one for student Will Brown.

STUDENT_ID	STUDENT	RANK	COLLEGE	COLLEGE_LEVEL
------------	---------	------	---------	---------------

0001	Ria Sinha	6	MICRO LINK	2
0002	Vivek Kaul	10	ROYAL	3
0003	George Smith	9	CPU	1
0004	Will Brown	1	CPU	1

Table 4.1 – A Data Redundancy Example (Student schema)

Table 4.1 will be used for all the examples in this chapter. The STUDENT_ID column, representing university roll numbers, is the unique primary key in this Student schema.

4.1.1 Insertion Anomalies

An insertion anomaly happens when the insertion of a data record is not possible unless we also add some additional unrelated data to the record. For example, inserting information about a student requires us to insert information about the college as well (COLLEGE_LEVEL column in Table 4.2).

STUDENT_ID	STUDENT	RANK	COLLEGE	COLLEGE_LEVEL
0005	Susan Fuller	11	Micro Link	2

Table 4.2 – Insertion Anomalies – Example

Moreover, there will be a repetition of the information in different locations in the database since the COLLEGE_LEVEL has to be input for each student.

4.1.2 Deletion Anomalies

A deletion anomaly happens when deletion of a data record results in losing some unrelated information that was stored as part of the record that was deleted from a table.

For example, by deleting all such rows that refer to students from a given college, we lose the COLLEGE and COLLEGE_LEVEL mapping information for that college. This is illustrated in Table 4.3, where we lose the information about the college 'IIT' if we delete the information about the two students George Smith and Will Brown.

STUDENT_ID	STUDENT	RANK	COLLEGE	COLLEGE_LEVEL
0003	George Smith	9	CPU	1
0004	Will Brown	1	CPU	1

Table 4.3 – Deletion Anomalies - Example

4.1.3 Update Anomalies

An update anomaly occurs when updating data for an entity in one place may lead to inconsistency, with the existing redundant data in another place in the table. For example, if we update the COLLEGE_LEVEL to '1' for STUDENT_ID 0003, then college 'Fergusson' has conflicting information with two college levels, 1 and 2.

STUDENT_ID	STUDENT	RANK	COLLEGE	COLLEGE_LEVEL
0001	Ria Sinha	6	MICRO LINK	2
0003	George Smith	9	MICRO LINK	1

- Modification Anomalies: During data update the consistency may also be violated as in the case of insertion.

Although normalization is a way to remove redundancy anomalies and preserve consistency, integrity and maintainability, it may also lead:

- ☐ Increase in storage space
- ☐ Complex queries (queries with many multiple joins of tables)

In such situations it may be desired to denormalize some of the tables in order to reduce storage space and the number of required joins.

Denormalization is the process of selectively taking normalized tables and re-combining the data in them. Sometimes the addition of a single column of redundant data to a table from another table can reduce a 4-way join into a 2-way join, significantly boosting performance by reducing the time it takes to perform the join.

Databases intended for Online Transaction Processing (OLTP) are normalized. By contrast, databases intended for On Line Analytical Processing (OLAP) operations are primarily "read only" databases and tend to extract historical data that has accumulated in the project for quite a

long time. For such databases, redundant or "denormalized" data may facilitate Business Intelligence applications.

While denormalization can boost storage and query performance, it can also have negative effects. For example, by adding redundant data to tables, you risk the following problems:

- More data means the RDBMS has to read more data pages than otherwise needed, hurting performance.
- Redundant data can lead to data anomalies and bad data.
- In many cases, extra code will have to be written to keep redundant data in separate tables in synch, which adds to database overhead.

3.4 Normal Forms

Normalization procedure provides:

- A framework for analyzing relation schemas based on functional and multivalued dependencies.
- A series of normal form test that can be carried out on individual relation schemas so that the relational database can be normalized to any degree.

Normalization through decomposition need to preserve the existence of two additional properties of a relational schema:

- **Lossless or Nonadditive Join:** Nonadditive join property guarantees that the spurious tuple generation does not occur after decomposition
- **Dependency Preservation:** Dependency preservation ensures that each functional dependency is presented in one of the individual relation resulting after decomposition.

First Normal Form (1NF)

A relation (table) R is in 1NF if and only if all underlying domains of attributes contain only atomic (simple, indivisible) values, i.e. the value of any attribute in a tuple (row) must be a single value from the domain of their attribute.

1NF allows removal of multivalued attributes, composite attributes and their combination in the relational schema.

Normalization (Decomposition)

Form new relation for each non-atomic attribute or nested relation.

Second Normal Form (2NF)

A relation schema R is in 2NF if it is in 1NF and every non-prime attribute A in R is fully functionally dependent on the primary key. (i.e. not partially dependent on candidate key).

Functional dependency $X \twoheadrightarrow Y$ is said to be fully functionally dependent if removal of any attribute from X result in for the dependency not hold.

NOTE: Mostly relational schemas that are mapped carefully from E/R model are in 2NF.

Normalization (Decomposition)

Decompose and set up a new relation for each partial key with its dependent attribute(s). Make sure to keep relation with the original primary key and any attributes that are fully functionally dependent on it.

Example: Consider a relation schemas for Employees and Teams in a single relation as follows

Emp_Teams(EmpId, Name, BDate, Gender, TeamId, Project, TeamName)

EmpId \twoheadrightarrow Name, BDate, Gender

TeamId \twoheadrightarrow Project, TeamName

Then upon decomposition we will have

Employees(EmpId, Name, BDate, Gender)

Teams(TeamId, Project, TeamName)

Emp_Teams(EmpId, TeamId)

Third Normal Form (3NF)

3NF for a relation schema R requires that the R be in 2NF, and that there would be no nonprime attribute of R that has transitive dependencies on the primary key. In summary, all non-key attributes are mutually independent. Thus, any relation in which all the attributes are prime attributes (part of some key) is guaranteed to be in at least 3NF.

That is; if $X \twoheadrightarrow Y$ is non-trivial functional dependency in R, then

- X is superkey for schema R, or
- Attribute Y is a member of a candidate key (prime attribute).

Normalization (Decomposition)

Decompose and set up a relation that includes the non-key attribute(s) that functionally determine(s) other non-key attributes.

Boyce-Codd Normal Form (BCNF)

BCNF requires that there will be no non-trivial functional dependencies of attributes on something other than a superset of a candidate key (called a superkey). At this stage, all attributes are dependent on a key, a whole key and nothing but a key (excluding trivial dependencies).

A table is said to be in the BCNF if and only if it is in the 3NF and every non-trivial, leftirreducible

functional dependency has a candidate key as its determinant. In more informal terms, a table is in BCNF if it is in 3NF and the only determinants are the candidate keys.

That is; if $X \twoheadrightarrow Y$ is non-trivial functional dependency in R, then

X is superkey for schema R.

Note that major goals of database design with functional dependencies are:

- BCNF,
- Lossless join, and
- Dependency preservation;

However; in certain situations it is needed to compromise BCNF need with 3NF to preserve dependency.

Example:

A relation that is in 3NF form but not in BCNF:

$R(A, B, C, D)$ and $F = \{AB \twoheadrightarrow CD, BC \twoheadrightarrow AD, A \twoheadrightarrow C\}$

AB and BC are candidate keys, thus

$A \twoheadrightarrow C$ will not violate 3NF where as it violates BCNF since A is not superkey.

A relation that is in 3NF form and in BCNF:

$R(A,B)$ is guaranteed to be in BCNF since its only possible functional dependencies are $A \twoheadrightarrow B$, $B \twoheadrightarrow A$ and/or the trivial $AB \twoheadrightarrow AB$.

Example: Consider the Project relation from the E/R model

- Projects(ProjId, Name, SDate, DDate, CustId, Name, Address)

$\text{ProjId} \twoheadrightarrow \text{Name, SDate, DDate, CustId, Name, Address}$

$\text{CustId} \twoheadrightarrow \text{Name, Address}$

Then upon decomposition we will have

Projects(ProjId, Name, SDate, DDate, CustId)

Customers(CustId, Name, Address)

Fourth Normal Form (4NF)

4NF requires that there be no non-trivial multivalued dependencies of attribute sets on something other than a superset of a candidate key.

A table is said to be in 4NF if and only if it is in the BCNF and multivalued dependencies are functional dependencies. The 4NF removes unwanted data structures (redundancy): multivalued dependencies.

That is; if $X \twoheadrightarrow Y$ is non-trivial multivalued dependency in R, then

\square X is superkey for schema R.

A **join dependency (JD)**, denoted by $JD(R_1, R_2, \dots R_n)$, specified on relational schema R, specifies a constraint on the state r of R. The constraint states that every legal state r of R have a nonadditive join decomposition into $R_1, R_2, \dots R_n$.

Join dependency is a general form of multivalued dependency where $n = 2$. (i.e. $JD(R_1, R_2)$

implies $(\) R_1 \cap R_2 \twoheadrightarrow R_1 - R_2$ and using complement property $(\) R_1 \cap R_2 \twoheadrightarrow R_2 - R_1$).

5NF also known as Project-Join Normal Form (PJNF) requires that there are no non-trivial join dependencies that do not follow from the key constraints. A table is said to be in the 5NF if and only if it is in 4NF and every join dependency in it is implied by the candidate keys.

That is; if $JD(R_1, R_2, \dots R_n)$ is non-trivial join dependency in R, then

\square Every R_i is superkey of R.

A join dependency (JD), denoted by $JD(R_1, R_2, \dots R_n)$, specified on relational schema R, specifies

a constraint on the state r of R . The constraint states that every legal state r of R have a nonadditive join decomposition into $R_1, R_2, \dots R_n$.

Although, there are also other higher level normalizations such as DKNF and 6NF, most relational database designs are sufficiently normalized at BCNF level or even at 3NF.

2.1. Introduction to information models and data models

An information model is an abstract, formal representation of entities that includes their properties, relationships and the operations that can be performed on them. The entities being modeled may be from the real world, such as devices on a network, or they may themselves be abstract, such as the entities used in a billing system.

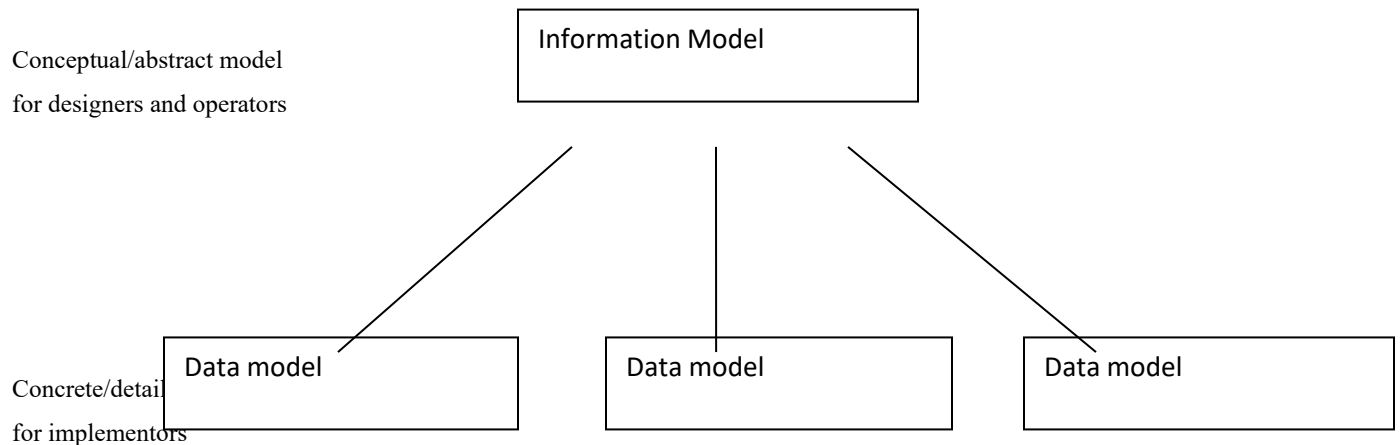
The primary motivation behind the concept is to formalize the description of a problem domain without constraining how that description will be mapped to an actual implementation in software. There may be many mappings of the Information Model. Such mappings are called data models, irrespective of whether they are object models (for example, using unified modeling language - UML), entity relationship models, or XML schemas.

Modeling is important as it considers the flexibility required for possible future changes without significantly affecting usage. Modeling allows for compatibility with its predecessor models and has provisions for future extensions.

Information Models and Data Models are different because they serve different purposes. The main purpose of an Information Model is to model managed objects at a conceptual level, independent of any specific implementations or protocols used to transport the data. The degree of detail of the abstractions defined in the Information Model depends on the modeling needs of its designers. In order to make the overall design as clear as possible, an Information Model should hide all protocol and implementation details. Another important characteristic of an Information Model is that it defines relationships between managed objects.

Data Models, on the other hand, are defined at a more concrete level and include many details.

They are intended for software developers and include protocol-specific constructs. A data model is the blueprint of any database system. Figure 1.1 illustrates the relationship between an Information Model and a Data Model.



2.2. Types of Data models

Data model proposals can be split into the following category in historical epochs:

- Hierarchical data model (IMS): late 1960's and 1970's
- Network data model(CODASYL): 1970's
- Relational data model: 1970's and early 1980's
- Object-oriented data model: late 1980's and early 1990's
- Object-relational data model: late 1980's and early 1990's

The next sections discuss some of these models in more detail

2.2.1. Hierarchical model

The hierarchical data model organizes data in a tree structure. There is a hierarchy of parent and child data segments. This structure implies that a record can have repeating information, generally in the child data segments. Data in a series of records will have a set of field values attached to it. It collects all the instances of a specific record together as a record type. These record types are the equivalent of tables in the relational model, and with the individual records being the equivalent of rows. To create links between these record types, the hierarchical model

uses Parent Child Relationships. In a hierarchical database the parent-child relationship is one to many. This restricts a child segment to having only one parent segment. Hierarchical DBMSs were popular from the late 1960s, with the introduction of IBM's Information Management System (IMS) DBMS, through the 1970s.

2.2.2. Network model

Some data may naturally be modeled with more than one parent per child. So, the network model permitted the modeling of many-to-many relationships in data. In 1971, the Conference on Data Systems Languages (CODASYL) formally defined the network model. The basic data modeling construct in the network model is the set construct. A set consists of an owner record type, a set name, and a member record type. A member record type can have that role in more than one set, hence the multi-parent concept is supported. An owner record type can also be a member or owner in another set. The data model is a simple network, and link and intersection record types may exist, as well as sets between them.

2.2.3. Relational data model

In the relational model, relations are used to hold information about the objects to be represented in the database. A relation is represented as a two-dimensional table in which the rows of the table correspond to individual records and the table columns correspond to attributes. Attributes can appear in any order and the relation will still be the same relation, and therefore convey the same meaning. For example, the information on branch offices is represented by the *Branch* relation, with columns for attributes *branchNo* (the branch number), *street*, *city*, and *postcode*. Similarly, the information on staff is represented by the *Staff* relation, with columns for attributes *staffNo* (the staff number), *fName*, *lName*, *position*, *sex*, *DOB* (date of birth), *salary*, and *branchNo* (the number of the branch the staff member works at).

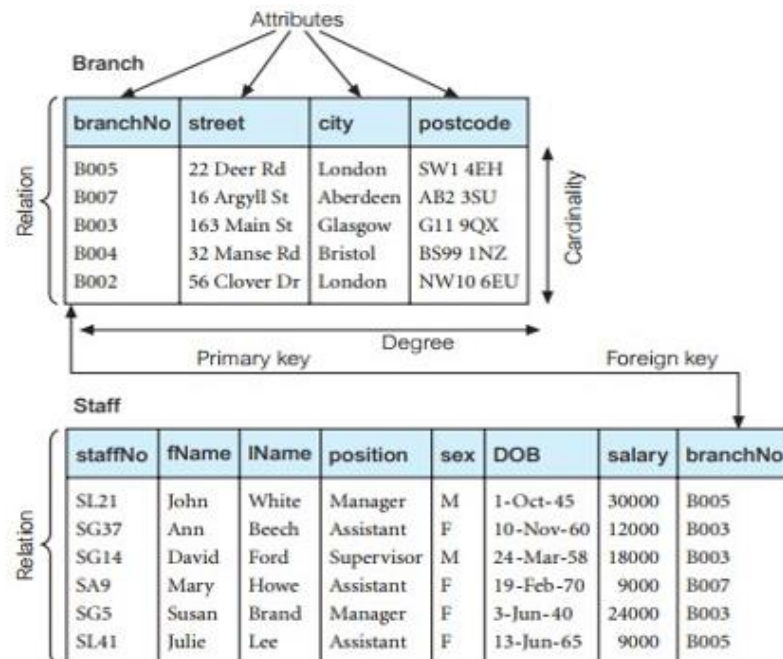
Relation: a table with rows and columns

Attribute: a named column of a relation

Domain: a set of allowable values for one or more attributes

Domains are an extremely powerful feature of the relational model. Every attribute in a relation is defined on a domain. Domains may be distinct for each attribute, or two or more attributes may be defined on the same domain. The domain concept is important because it allows the user to define in a central place the

meaning and source of values that attributes can hold. As a result, more information is available to the system when it undertakes the execution of a relational operation, and operations that are semantically incorrect can be avoided. For example, it is not sensible to compare a street name with a telephone number, even though the domain definitions for both these attributes are character strings. On the other hand, the monthly rental on a property and the number of months a property has been leased have different domains (the first a monetary value, the second an integer value), but it is still a legal operation to multiply two values from these domains.



Attribute	Domain Name	Meaning	Domain Definition
branchNo	BranchNumbers	The set of all possible branch numbers	character: size 4, range B001–B999
street	StreetNames	The set of all street names in Britain	character: size 25
city	CityNames	The set of all city names in Britain	character: size 15
postcode	Postcodes	The set of all postcodes in Britain	character: size 8
sex	Sex	The sex of a person	character: size 1, value M or F
DOB	DatesOfBirth	Possible values of staff birth dates	date, range from 1-Jan-20, format dd-mmm-yy
salary	Salaries	Possible values of staff salaries	monetary: 7 digits, range 6000.00–40000.00

Tuple: a row of a relation

The elements of a relation are the rows or tuples in the table. In the *Branch* relation, each row contains four values, one for each attribute. Tuples can appear in any order and the relation will still be the same relation, and therefore convey the same meaning. The structure of a relation, together with a specification of the domains and any other restrictions on possible values, is

sometimes called its intension, which is usually fixed unless the meaning of a relation is changed to include additional attributes. The tuples are called the extension (or state) of a relation, which changes over time.

Degree: the degree of a relation is the number of attributes it contains Unary relation, Binary relation, Ternary relation, N-ary relation.

The Branch relation in the above Figure has four attributes or degree four. This means that each row of the table is a four-tuple, containing four values. A relation with only one attribute would have degree one and be called a *unary* relation or one-tuple. A relation with two attributes is called *binary*, one with three attributes is called *ternary*, and after that the term *nary* is usually used. The degree of a relation is a property of the *intension* of the relation

Cardinality: of a relation is the number of tuples the relation has By contrast, the number of tuples is called the cardinality of the relation and this changes as tuples are added or deleted. The cardinality is a property of the *extension* of the relation and is determined from the particular instance of the relation at any given moment. Finally, we have the definition of a relational database.

Relational Database: a collection of normalized relations with distinct relation names. A relational database consists of relations that are appropriately structured.

Relation Schema: a named relation defined by a set of attribute-domain name pair Let A_1, A_2, \dots, A_n be attributes with domain D_1, D_2, \dots, D_n .

Then the sets $\{A_1:D_1, A_2:D_2 \dots A_n:D_n\}$ is a Relation Schema. A relation R , defined by a relation schema S , is a set of mappings from attribute names to their corresponding domains.

Thus a relation is a set of n - tuples of the form $(A_1:d_1, A_2:d_2, \dots, A_n:d_n)$ where $d_1 \in D_1, d_2 \in D_2, \dots, d_n \in D_n$,

Example Student (studentId char(10), studentName char(50), DOB date) is a *relation schema* for the student entity in SQL

Relational Database schema: a set of relation schema each with distinct names. Suppose

R_1, R_2, \dots, R_n is the set of relation schema in a relational database then the relational database schema (R) can be stated as: $R = \{ R_1, R_2, \dots, R_n \}$.

Properties of Relational Tables:

- The Sequence of Columns is Insignificant
- The Sequence of Rows is Insignificant
- Each Column Has a Unique Name.
- A relation has a name that is distinct from all other relation names in the relational schema.
- Each tuple in a relation must be unique
- All tables are *LOGICAL ENTITIES*
- Each cell of a relation contains exactly one atomic (single) value.
- Each column (field or attribute) has a distinct name.
- The values of an attribute are all from the same domain.
- A table is either a BASE TABLES (Named Relations) or VIEWS (Unnamed Relations)
- Only Base Tables are physically stored
- VIEWS are derived from BASE TABLES with SQL statements like: [SELECT .. FROM .. WHERE .. ORDER BY]
- Relational database is the collection of tables. Each entity in one table has Attributes are fields (columns) in table
- Order of rows theoretically (but practically has impact on performance) and columns is immaterial
- Entries with repeating groups are said to be un-normalized. All values in a column represent the same attribute and have the same data format.

Building blocks of the relational data model

The building blocks of the relational data model are:

- **Entities:** real world physical or logical object
- **Attributes:** properties used to describe each Entity or real world object.
- **Relationship:** the association between Entities
- **Constraints:** rules that should be obeyed while manipulating the data.

The ENTITIES

The Entities are persons, places; things etc. which the organization has to deal with Relations can also describe relationships

The name given to an entity should always be a singular noun descriptive of each item to be stored in it.

Example: student NOTstudents.

Every relation has a schema, which describes the columns, or fields the relation itself corresponds to our familiar notion of a table:

A relation is a collection of *tuples*, each of which contains values for a fixed number of *attributes*

- Existence Dependency: the dependence of an entity on the existence of one or more entities.
- Weak entity : an entity that cannot exist without the entity with which it has a relationship – it is indicated by a **double rectangle**

The ATTRIBUTES

The attributes are the items of information which characterize and describe these entities. Attributes are pieces of information ABOUT entities. The analysis must of course identify those which are actually relevant to the proposed application. Attributes will give rise to recorded items of data in the database

At this level we need to know such things as:

- Attribute name (be explanatory words or phrases)
- The domain from which attribute values are taken (A DOMAIN is a set of values from which attribute values may be taken)
- **For example**, the domain of Name is string and en.) Each attribute has values taken from a domain. that for salary is real. However these are not shown on **E-R models**
- Whether the attribute is part of the **entity identifier** (attributes which just describe an entity and those which help to identify it uniquely)

- Whether it is **permanent or time-varying** (which attributes may change their values over time)
- Whether it is **required or optional** for the entity (whose values will sometimes be unknown or irrelevant)

Types of Attributes

1. Simple (atomic) Vs Composite attributes

- **Simple** : contains a single value (not divided into sub parts)

Example: Age, gender

- **Composite:** Divided into sub parts (composed of other attributes) **Example:** Name, address
- Single-valued Vs multi-valued attributes
- **Single-valued** : have only single value(the value may change but has only one value at one time)

Example: Name, Sex, Id. No. color_of_eyes

- **Multi-Valued:** have more than one value

Example: Address, dependent-name

Person may have several college degrees

- Stored vs. Derived Attribute
- **Stored** : not possible to derive or compute

Example: Name, Address

- **Derived:** The value may be derived (computed) from the values of other attributes.

Example: Age (current year – year of birth)

Length of employment (current date- start date) Profit (earning cost) G.P.A (grade point/credit hours)

- Null Values
- NULL applies to attributes which are not applicable or which do not have values.
- You may enter the value NA (meaning not applicable)
- Value of a key attribute cannot be null.
- Default value- assumed value if no explicit value

Entity versus Attributes

When designing the conceptual specification of the database, one should pay attention to the distinction between an Entity and an Attribute.

- Consider designing a database of employees for an organization:
- Should *address* be an attribute of Employees or an entity (connected to Employees by a relationship)?
 - If we have several addresses per employee, *address* must be an entity

(attributes cannot be set-valued/multi valued)

- If the structure (city, Woreda, Kebele, etc) is important, e.g. want to retrieve employees in a given city, address must be modeled as an entity *(attribute values are atomic)*

The RELATIONSHIPS

The Relationships between entities which exist and must be taken into account when processing information. In any business processing one object may be associated with another object due to some event. Such kind of association is what we call a RELATIONSHIP between entity objects.

- One external event or process may affect several related entities.
- Related entities require setting of LINKS from one part of the database to another.
- A relationship should be named by a word or phrase which explains its function
- Role names are different from the names of entities forming the relationship: one entity may take on many roles, the same role may be played by different entities
- For each RELATIONSHIP, one can talk about the Number of Entities and the

Number of Tuples participating in the association. These two concepts are called **DEGREE** and **CARDINALITY** of a relationship respectively.

Degree of a Relationship

- An important point about a relationship is how many entities participate in it. The number of entities participating in a relationship is called the **DEGREE** of the relationship. Among the Degrees of relationship, the following are the basic:
- **UNARY/RECURSIVE RELATIONSHIP:** *Tuples/records of a Single entity are related withy each other.*
- **BINARY RELATIONSHIPS:** *Tuples/records of two entities are associated in a relationship*
- **TERNARY RELATIONSHIP:** *Tuples/records of three different entities are associated*
- And a generalized one:

o **N-ARY RELATIONSHIP:** *Tuples from arbitrary number of entity sets are participating in a relationship.*

Cardinality of a Relationship

Another important concept about relationship is the number of instances/tuples that can be associated with a single instance from one entity in a single relationship. The number of instances participating or associated with a single instance from an entity in a relationship is called the **CARDINALITY** of the relationship. The major cardinalities of a relationship are:

- ONE-TO-ONE: one tuple is associated with only one other tuple.
- o **Example:** Building – Location as a single building will be located in a single location and as a single location will only accommodate a single Building.
- ONE-TO-MANY, one tuple can be associated with many other tuples, but not the reverse.
- o **Example:** Department-Student as one department can have multiple students.
- MANY-TO-ONE, many tuples are associated with one tuple but not the reverse.
- o **Example:** Employee – Department: as many employees belong to a single department.
- MANY-TO-MANY: one tuple is associated with many other tuples and from the other side, with a different role name one tuple will be associated with many tuples
- o **Example:** Student – Course as a student can take many courses and a single course can be attended by many students.

However, the degree and cardinality of a *relation* are different from degree and cardinality of a *relationship*

Key constraints

If tuples are need to be unique in the database, and then we need to make each tuple distinct. To do this we need to have relational keys that uniquely identify each record.

1. **Super Key:** an attribute/set of attributes that uniquely identify a tuple within a relation.
2. **Candidate Key:** a super key such that no proper subset of that collection is a Super Key within the relation.

A candidate key has two properties:

- I. **Uniqueness**
- II. **Irreducibility:-** If a super key is having only one attribute, it is automatically a Candidate key.

If a candidate key consists of more than one attribute it is called Composite Key.

- 3. **Primary Key:** the candidate key that is selected to identify tuples uniquely within the relation. The entire set of attributes in a relation can be considered as a primary case in a worst case.
- 4. **Foreign Key:** an attribute, or set of attributes, within one relation that matches the candidate key of some relation. A foreign key is a link between different relations to create a view or an unnamed relation

Integrity, Referential Integrity and Foreign Keys Constraints

The entity integrity constraint states that no primary key value can be NULL. This is because the primary key value is used to identify individual tuples in a relation. Having NULL values for the primary key implies that we cannot identify some tuples. For example, if two or more tuples had NULL for their primary keys, we may not be able to distinguish them if we try to reference them from other relations. Key constraints and entity integrity constraints are specified on individual relations.

The referential integrity constraint is specified between two relations and is used to maintain the consistency among tuples in the two relations. Informally, the referential integrity constraint states that a tuple in one relation that refers to another relation must refer to an existing tuple in that relation. For example, the attribute Dept_Num of EMPLOYEE gives the department number for which each employee works; hence, its value in every EMPLOYEE tuple must match the Dept_Num value of some tuple in the DEPARTMENT relation.

To define referential integrity more formally, first we define the concept of a foreign key. The conditions for a foreign key, given below, specify a referential integrity constraint between the two relation schemas R1 and R2.

A set of attributes FK (foreign key) in relation schema R1 is a foreign key of R1 that references relation R2 if it satisfies the following rules:

- **Rule 1:** The attributes in FK have the same domain(s) as the primary key attributes PK of R2; the attributes FK are said to reference or refer to the relation R2.
- **Rule 2:** A value of FK in a tuple $t1$ of the current state $r1(R1)$ either occurs as a value

of PK for some tuple $t2$ in the current state $r2(R2)$ or is *NULL*. In the former case, we have $t1[FK] = t2[PK]$, and we say that the tuple $t1$ references or refers to the tuple $t2$.

- In this definition, R1 is called the referencing relation and R2 is the referenced relation. If these two conditions hold, a referential integrity constraint from R1 to R2 is said to hold. In a database of many relations, there are usually many referential integrity constraints.

To specify these constraints, first we must have a clear understanding of the meaning or roles that each attribute or set of attributes plays in the various relation schemas of the database.

Referential integrity constraints typically arise from the relationships among the entities represented by the relation schemas.

For example, consider the database. In the *EMPLOYEE* relation, the attribute *Dept_Num* refers to the department for which an employee works; hence, we designate *Dept_Num* to be a foreign key of *EMPLOYEE* referencing the *DEPARTMENT* relation. This means that a value of *Dept_Num* in any tuple $t1$ of the *EMPLOYEE* relation must match a value of *DEPARTMENT* relation, or the value of *Dept_Num* can be *NULL* if the employee does not belong to a department or will be assigned to a department later. For example, the tuple for employee ‘John Smith’ references the tuple for the ‘Research’ department, indicating that ‘John Smith’ works for this department.

Notice that a foreign key can *refer to its own relation*. **For example**, the attribute *Super_ssn* in *EMPLOYEE* refers to the supervisor of an employee; this is another employee, represented by a tuple in the *EMPLOYEE* relation. Hence, *Super_ssn* is a foreign key that references the *EMPLOYEE* relation itself. The tuple for employee ‘John Smith’ references the tuple for employee ‘Franklin Wong’, indicating that ‘Franklin Wong’ is the supervisor of ‘John Smith’.

We can diagrammatically display referential integrity constraints by drawing a directed arc from each foreign key to the relation it references. For clarity, the arrowhead may point to the primary key of the referenced relation with the referential integrity constraints displayed in this manner.

All integrity constraints should be specified on the relational database schema (i.e., defined as part of its definition) if we want to enforce these constraints on the data- base states. Hence, the DDL includes provisions for specifying the various types of constraints so that the DBMS can automatically enforce them. Most relational DBMSs support key, entity integrity, and referential integrity constraints. These constraints are specified as a part of data definition in the DDL.

The following constraints are specified as a part of data definition using DDL:

- **Domain Integrity:** No value of the attribute should be beyond the allowable limits
- **Entity Integrity:** In a base relation, no attribute of a Primary Key can assume a value of NULL
- **Referential Integrity:** If a Foreign Key exists in a relation, either the Foreign Key value must match a Candidate Key value in its home relation or the Foreign Key value must be NULL
- **Enterprise Integrity:** Additional rules specified by the users or database administrators of a database are incorporated

2.2.4. Entity Relation model

In the mid 1970's, Peter Chen proposed the entity-relationship (E-R) data model. This was to be an alternative to the relational, CODASYL, and hierarchical data models. He proposed thinking of a database as a collection of instances of entities. Entities are objects that have an existence independent of any other entities in the database. Entities have attributes, which are the data elements that characterize the entity. One or more of these attributes could be designated to be a key. Lastly, there could be relationships between entities. Relationships could be 1-to-1, 1-to-n, n-to-1 or m-to-n, depending on how the entities participated in the relationship. Relationships could also have attributes that described the relationship. Figure 1.5 provides an example of an E-R diagram.

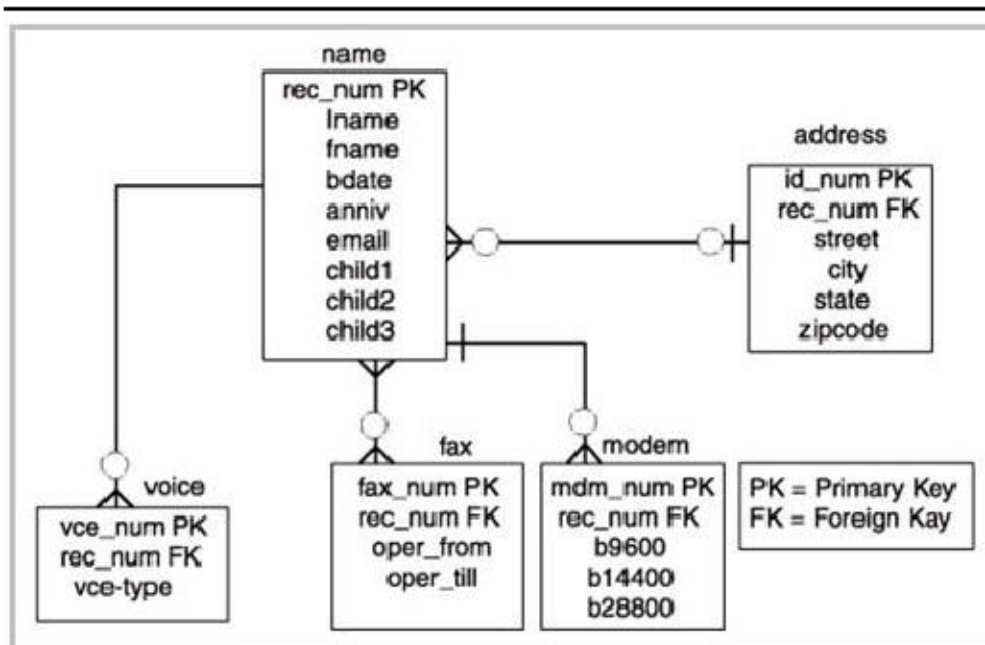


Figure 1.5 - An E-R Diagram for a telephone directory data model

In the figure, entities are represented by rectangles and they are name, address, voice, fax, and modem. Attributes are listed inside each entity. For example, the voice entity has the vce_num, rec_num, and vce-type as attributes. PK represents a primary key, and FK a foreign key.

Entity – Relationship (E/R) model is a conceptual model based on a perception of the world based on the concept of entities, attributes and relationships.

- **Entity**: represents a real-world object or concept; such as employee and account.
- **Attribute**: describes an entity in the database; such as name and birth date for employee and account number and balance for account.
- **Relationship**: is an association among the entities. For example a customer entity is related to the account entity in a banking system.

Rather than being used as a model on its own, the E-R model has found success as a tool to design relational databases. Chen's papers contained a methodology for constructing an initial E-R diagram. In addition, it was a simple process to convert an E-R diagram into a collection of tables in third normal form.

2.2.5. Object-Data model

The advancement of the Object-Oriented Programming (OOP) tends to evolve a new database management system namely the Object DBMS (ODBMS). The **object data model** is a way for the modeling of a database in ODBMS. It can be regarded as high-level implementation data model that is closer to the conceptual model. It is based on the object-oriented concept mainly for ODBMS implementation but can also be used in the data model of RDBMS implementation. This combination of object-oriented data model with the relational model leads into a data model known as **object-relational data model**. The Object-Relational (OR) model is very similar to the relational model; however, it treats every entity as an object (instance of a class), and a relationship as an inheritance. Some features and benefits of an Object-Relational model are:

- Support for complex, user-defined types
- Object inheritance
- Extensible objects

Object-Relational databases have the capability to store object relationships in relational form.

2.3. Difference between Hierarchical and Relational Data Model:

S. No.	Hierarchical Data Model	Relational Data Model
1.	In this model, to store data hierarchy method is used. It is the oldest method.	It is the most flexible and efficient database model. It is the most used database in today.
2.	It implements 1:1 and 1:n.	In addition to 1:1 and 1:n, it also implements many-to-many relationships.
3.	To organize records, it uses a tree structure.	To organize records, it uses a table.
4.	More chances of complexity.	No chance of complexity.
5.	There is a lack of declarative query facility. In current times they are being modeled using NoSQL.	It provides facility of declarative query facility using SQL.

S. No.	Hierarchical Data Model	Relational Data Model
6.	Records are linked with help of pointers.	Records are linked with help of rows and columns.
7.	Insertion anomaly exists in this model i.e. child node cannot be inserted without parent node.	There is no insertion anomaly.
8.	Deletion anomaly exists in this model i.e. it is difficult to delete parent node.	There is no deletion anomaly.
9.	It is used to access data which is complex and asymmetric.	It is used to access data which is complex and symmetric.
10.	This model lacks data independence.	This model provides data independence.
11.	This design is used in modern times for faster access of data. This is obtained by trade offs i.e. on giving up on redundancy where the levels(parents to child) are relatively less.	Due to many to many to many relationship joins take a heavy toll on search with multiple parameter query.
12.	Complexity leads to difficulty in designing database.	It is not complex as physical level details are not visible to user.
13.	It is having an inconsistency problem while updating the records due to multiple instances of a child record.	The normalization is used to remove the redundancy while updating the records.
14.	Currently this model is being used in shopping carts and search engine. There are tools that can emulate hierarchical database e.g. MongoDB, firebase	Most of the traditional software are using relational database common e.g. Oracle dB, MS sql server, IBM DB2

2.4. Difference between Network and Relational Data Model :

S. No.	Network Data Model	Relational Data Model
---------------	---------------------------	------------------------------

S. No.	Network Data Model	Relational Data Model
1.	It organizes records to one another through links or pointers.	It organizes records in form of table and relationship between tables are set using common fields.
2.	It organizes records in form of directed graphs.	It organizes records in form of tables.
3.	In this relationship between various records is represented physically via linked list.	In this relationship between various records is represented logically via tables.
4.	There is lack of declarative querying facilities.	It provides declarative query facility using SQL.
5.	Complexity increases burden on programmer for database design as well as data manipulation.	As physical level details are hidden from end users so this model is very simple to understand.
6.	Retrieval algorithms are complex but symmetric.	Retrieval algorithms are simple and symmetric.
7.	There is partial data independence in this model.	This model provides data independence.
8.	There is no inconsistency problem in updating the records because of the single instance of the child records.	The updating of records is quite easy because of the normalization which is used to remove the redundancy in the relations.
9.	Searching for a record is easy in the network model as there are multiple access paths to reach data item.	In the relational model, a unique, indexed key serves the purpose of searching a record.
10.	Here, is the physical existence of record relations in this model.	It maintains logical organization of records using rows and columns and stored in relation.

S. No.	Network Data Model	Relational Data Model
11.	VAX-DBMS, DMS-1100 of UNIVAC and SUPRADBMS's use this model.	It is mostly used in real world applications. Oracle, SQL.

2.5. Relational database management system (RDBMS)

A relational database management system (RDBMS) is a collection of programs and capabilities that enable IT teams and others to create, update, administer and otherwise interact with a relational database. RDBMSes store data in the form of tables, with most commercial relational database management systems using Structured Query Language (SQL) to access the database. However, since SQL was invented after the initial development of the relational model, it is not necessary for RDBMS use.

The RDBMS is the most popular database system among organizations across the world. It provides a dependable method of storing and retrieving large amounts of data while offering a combination of system performance and ease of implementation.

RDBMS vs. DBMS

In general, databases store sets of data that can be queried for use in other applications. A database management system supports the development, administration and use of database platforms.

An RDBMS is a type of database management system (DBMS) that stores data in a row-based table structure which connects related data elements. An RDBMS includes functions that maintain the security, accuracy, integrity and consistency of the data. This is different than the file storage used in a DBMS.

Other differences between database management systems and relational database management systems include:

- **Number of allowed users.** While a DBMS can only accept one user at a time, an RDBMS can operate with multiple users.
- **Hardware and software requirements.** A DBMS needs less software and hardware than an RDBMS.
- **Amount of data.** RDBMSes can handle any amount of data, from small to large, while a DBMS can only manage small amounts.
- **Database structure.** In a DBMS, data is kept in a hierarchical form, whereas an RDBMS utilizes a table where the headers are used as column names and the rows contain the corresponding values.
- **ACID implementation.** DBMSes do not use the atomicity, consistency, isolation and durability (ACID) model for storing data. On the other hand, RDBMSes base the structure of their data on the ACID model to ensure consistency.
- **Distributed databases.** While an RDBMS offers complete support for distributed databases, a DBMS will not provide support.
- **Types of programs managed.** While an RDBMS helps manage the relationships between its incorporated tables of data, a DBMS focuses on maintaining databases that are present within the computer network and system hard disks.
- **Support of database normalization.** An RDBMS can be normalized, but a DBMS cannot.

Features of relational database management systems

Elements of the relational database management system that overarch the basic relational database are so intrinsic to operations that it is hard to dissociate the two in practice.

The most basic RDBMS functions are related to create, read, update and delete operations -- collectively known as CRUD. They form the foundation of a well-organized system that promotes consistent treatment of data.

The RDBMS typically provides data dictionaries and metadata collections that are useful in data handling. These programmatically support well-defined data structures and relationships. Data storage management is a common capability of the RDBMS, and this has come to be defined by data objects that range from binary large object -- or blob -- strings to stored procedures. Data objects like this extend the scope of basic relational database operations and can be handled in a variety of ways in different RDBMSes.

The most common means of data access for the RDBMS is SQL. Its main language components comprise data manipulation language and data definition language statements. Extensions are available for development efforts that pair SQL use with common programming languages, such as the Common Business-Oriented Language (COBOL), Java and .NET.

RDBMSes use complex algorithms that support multiple concurrent user access to the database while maintaining data integrity. Security management, which enforces policy-based access, is yet another overlay service that the RDBMS provides for the basic database as it is used in enterprise settings.

RDBMSes support the work of database administrators (DBAs) who must manage and monitor database activity. Utilities help automate data loading and database backup. RDBMSes manage log files that track system performance based on selected operational parameters. This enables measurement of database usage, capacity and performance, particularly query performance. RDBMSes provide graphical interfaces that help DBAs visualize database activity.

While not limited solely to the RDBMS, ACID compliance is an attribute of relational technology that has proved important in enterprise computing. These

capabilities have particularly suited RDBMSes for handling business transactions. As RDBMSes have matured, they have achieved increasingly higher levels of query optimization, and they have become key parts of reporting, analytics and data warehousing applications for businesses as well. RDBMSes are intrinsic to operations of a variety of enterprise applications and are at the center of most master data management systems.

How RDBMS works

As mentioned before, an RDBMS will store data in the form of a table. Each system will have varying numbers of tables with each table possessing its own unique primary key. The primary key is then used to identify each table.

Within the table are rows and columns. The rows are known as records or horizontal entities; they contain the information for the individual entry. The columns are known as vertical entities and possess information about the specific field.

Before creating these tables, the RDBMS must check the following constraints:

- Primary keys -- this identifies each row in the table. One table can only contain one primary key. The key must be unique and without null values.
- Foreign keys -- this is used to link two tables. The foreign key is kept in one table and refers to the primary key associated with another table.
- Not null -- this ensures that every column does not have a null value, such as an empty cell.
- Check -- this confirms that each entry in a column or row satisfies a precise condition and that every column holds unique data.
- Data integrity -- the integrity of the data must be confirmed before the data is created.

Assuring the integrity of data includes several specific tests, including entity, domain, referential and user-defined integrity. Entity integrity confirms that the

rows are not duplicated in the table. Domain integrity makes sure that data is entered into the table based on specific conditions, such as file format or range of values. Referential integrity ensures that any row that is re-linked to a different table cannot be deleted. Finally, user-defined integrity confirms that the table will satisfy all user-defined conditions.

Advantages of relational database management system

The use of an RDBMS can be beneficial to most organizations; the systematic view of raw data helps companies better understand and execute the information while enhancing the decision-making process. The use of tables to store data also improves the security of information stored in the databases. Users are able to customize access and set barriers to limit the content that is made available. This feature makes the RDBMS particularly useful to companies in which the manager decides what data is provided to employees and customers.

Furthermore, RDBMSes make it easy to add new data to the system or alter existing tables while ensuring consistency with the previously available content.

Other advantages of the RDBMS include:

- Flexibility -- updating data is more efficient since the changes only need to be made in one place.
- Maintenance -- database administrators can easily maintain, control and update data in the database. Backups also become easier since automation tools included in the RDBMS automate these tasks.
- Data structure -- the table format used in RDBMSes is easy to understand and provides an organized and structural manner through which entries are matched by firing queries.

On the other hand, relational database management systems do not come without their disadvantages. For example, in order to implement an RDBMS, special software must be purchased. This introduces an additional cost for execution. Once

the software is obtained, the setup process can be tedious since it requires millions of lines of content to be transferred into the RDBMS tables. This process may require the additional help of a programmer or a team of data entry specialists. Special attention must be paid to the data during entry to ensure sensitive information is not placed into the wrong hands.

Some other drawbacks of the RDBMS include the character limit placed on certain fields in the tables and the inability to fully understand new forms of data -- such as complex numbers, designs and images.

Furthermore, while isolated databases can be created using an RDBMS, the process requires large chunks of information to be separated from each other. Connecting these large amounts of data to form the isolated database can be very complicated.

Uses of RDBMS

Relational database management systems are frequently used in disciplines such as manufacturing, human resources and banking. The system is also useful for airlines that need to store ticket service and passenger documentation information as well as universities maintaining student databases.

Some examples of specific systems that use RDBMS include IBM, Oracle, MySQL, Microsoft SQLServer and PostgreSQL.

CHAPTER THREE

3. Conceptual Database Design and E-R Modeling

3.1. Introduction

As it is one component in most information system development tasks, there are several steps in designing a database system. Here more emphasis is given to the design phases of the system development life cycle.

The major steps in database design are;

- A. **Planning:** that is identifying information gap in an organization and propose a database solution to solve the problem.
- B. **Analysis:** that concentrates more on fact finding about the problem or the opportunity. Feasibility analysis, requirement determination and structuring, and selection of best design method are also performed at this phase.
- C. **Design:** in database development more emphasis is given to this phase. The phase is further divided into three sub-phases.
 - I. **Conceptual Design:** concise description of the data, data type, relationship between data and constraints on the data.
 - ✓ There is no implementation or physical detail consideration.
 - ✓ Used to elicit and structure all information requirements

Conceptual design revolves around discovering and analyzing organizational and user data requirements.

The important activities are to identify:-

- I. Entities
- II. Attributes
- III. Relationships
- IV. Constraints
- V. And based on these identified components then develop the ER model using ER diagrams

Designing conceptual model for the database is not a one linear process but an iterative activity where the design is refined again and again.

To identify the entities, attributes, relationships, and constraints on the data, there are different set of methods used during the analysis phase. These include information gathered by:

- Interviewing end users individually and in a group

- Questionnaire survey
- Direct observation
- Examining different documents Analysis of requirements gathered:
- Nouns prospective entities
- Adjectives prospective attributes
- Verbs/verb phrases prospective relationships

The basic E-R model is graphically depicted and presented for review. The process is repeated until the end users and designers agree that the E-R diagram is a fair representation of the organizations activities and functions. Checking for Redundant Relationships in the ER Diagram Relationships between entities indicate access from one entity to another – it is therefore possible to access one entity occurrence from another entity occurrence even if there are other entities and relationships that separate them – this is often referred to as *Navigation*’ of the ER diagram. The last phase in ER modeling is validating an ER Model against requirement of the user.

II. Logical Design: a higher level conceptual abstraction with selected specific data model to implement the data structure.

✓ It is particular DBMS independent and with no other physical considerations.

III. Physical Design: physical implementation of the logical design of the database with respect to internal storage and file structure of the database for the selected DBMS.

✓ To develop all technology and organizational specification.

D. Implementation: the testing and deployment of the designed database for use.

E. Operation and Support: administering and maintaining the operation of the database system and providing support to users. Tuning the database operations for best performance.

3.2. Conceptual Database Design

3.2.1. Steps to Build Conceptual Data Model

4 steps in designing a conceptual data model using the E-R diagram

- Identify entity sets
- Define the value sets, attributes and primary key for each entity set

- Identify relationship sets and semantic information (cardinality, subtype/super type) for each relationship set
- Integrate multiple views of entities, attributes, and relationships

3.2.2. Symbols Used in ER Diagram

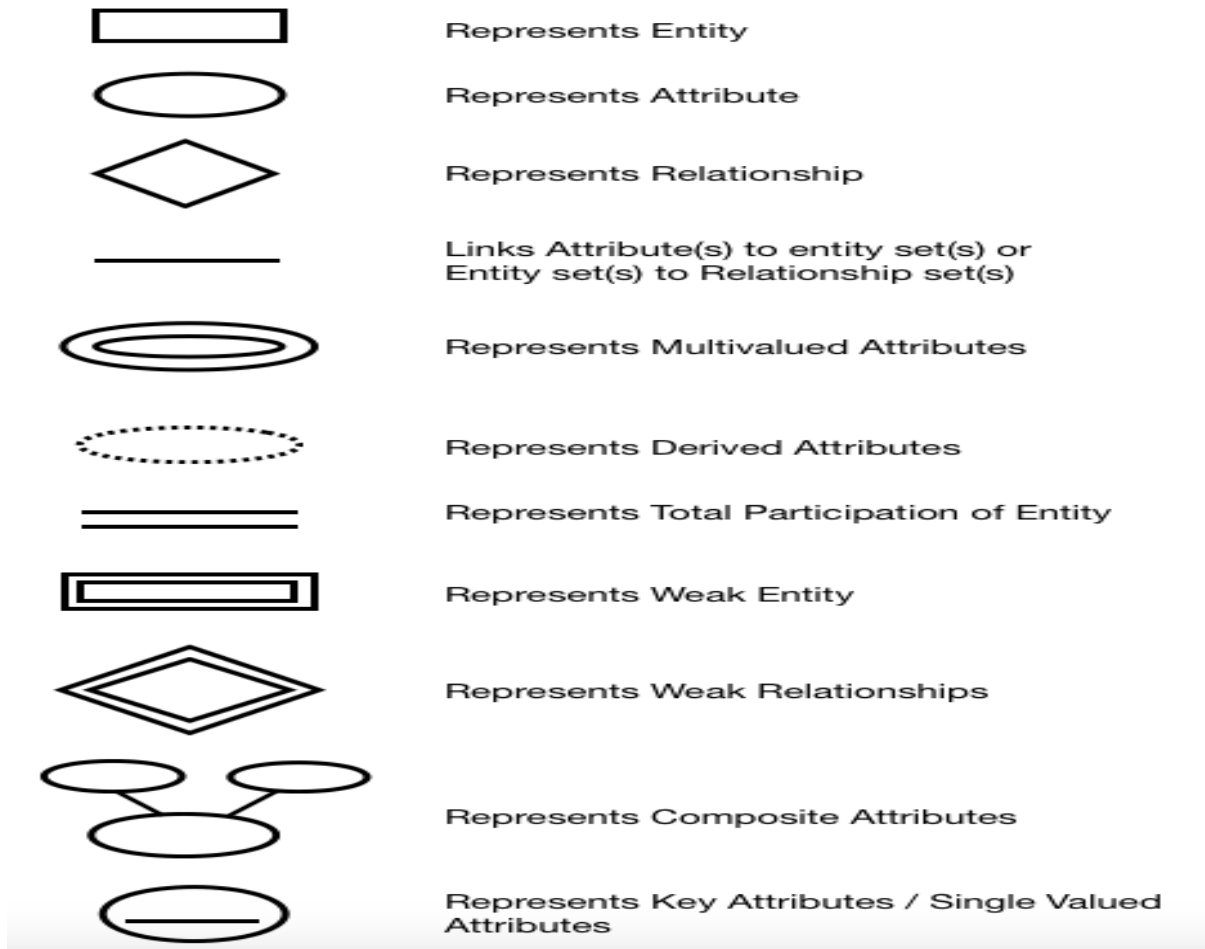


Figure 2:- Symbols Used in ER Diagram:-

Concept	Symbol
Entity	Rectangle
Weak entity	Double rectangle
Associative entity	Diamond within a rectangle
Relationship	Diamond
Identifying relationship	Double diamond
Cardinality	Crow's foot

Mandatory cardinality	Solid " " (s) superimposed on the relationship line
Optional cardinality	A "0" superimpose on the relationship line
Subtype/Super type	Circle
Direction of subtype/super type	Open-end of the "U" points towards a super type
Total specialization	Double line extending from a super type
Disjoint rule	A "d" in the circle joining the super type & its subtypes
Overlap rule	An "o" in the circle joining the super type & its subtypes
Attribute	Ellipse
Multi-valued attribute	Double ellipse
Identifier	Underlined
Derived attribute	Broken ellipse



Fig 4. Relationship Attributes

3.3. Design ER Diagram

Basic constructs of the E-R model

Concept	Definition	Examples
Entity	A person, place, object, event or concept	Employee, department, building, sale, account
Relationship	An entity that serves to interconnect two or more entity types	Assignment (Employee-Department)
Attribute	A property or characteristic of an entity/relationship type	Employee_name, department_location, sale_date
Constraints	Guiding policies or rules that defines or restricts the structure and processing of a database	All business majors must have a GPA of 2.9 or above

The Entity Relationship (E/R) data model is a diagrammatical data model. The elements of the E/R model are represented by the following symbols:-

Example

- “EMPLOYEES” are *Assigned* to “TEAMS”
- “CUSTOMERS” *Owns* “PROJECTS”
- “TEAMS” *works on* “PROJECTS”

Composite attributes are represented by linked ellipses as depicted in the above figure with the attributes “Address” and “H Addr”.

Relationship Attributes: Attribute(s) may be used in some relationships to describe the relationship further. Consider the relationship “*WorksOn*” between the “TEAMS” and “PROJECTS” entity sets. The relationship can be further described if an attribute “Task” is added to it as follows.

Multi-way Relationship: Consider the three way relationship between the “PROJECTS”, “TEAMS”, and “SOFTWARE” entity sets.

Multi-way Relationship: Consider the three way relationship between the “PROJECTS”, “TEAMS”, and “SOFTWARE” entity sets.

Cardinality Limits of a Relationship: The cardinal limit of a relationship is labeled as:

- **0..*** or **0..∞** indicating zero or more participation of the entity in the relationship.
- **1..*** or **1..∞** indicating one or more participation of the entity in the relationship.
- **0..1** indicating zero or one participation of the entity in the relationship.
- **1..1** indicating exactly one participation of the entity in the relationship.

3.4. Entity-Relationship Diagram Building Blocks

The three basic notions of the E/R model are:

□ **Entity:** represents existing real-world objects or concepts, such as *places, objects, events, persons, orders, customers*, and so on.

□ **Relationship:** represents associations between objects, such as the fact that a customer *may place an order*.

Attribute: describes the entity, such as the invoice *date* or the customer *first name*.

Case Study (The case in here is only for teaching purpose and it is no way related to any company)

Consider a database system to be developed for “XYZ Software Share Company”. The following are brief and short listed structures of the Company.

- The company runs various projects with a total of 68 full-time employees and over 120 part-time employees.
- A project has a unique id and a name that may be designed for a new software development or for a release of a new version of software that had been developed by the company.
- The projects are having start date, due date, complete date and status that describe their progress. Every project is lead by a senior manager organized into teams of five to eight programmers coordinated by a team leader.
- The owners of the projects are the customers of the company. A single customer can own one or more projects. The customers have unique id, name and address.
- The company is organized into departments that are identified by a unique name and lead by department heads. A department head can only lead a single department in his/her employment by the company.
- An employee can only belong to one department. Every employee is identified by an Id, a name, an address, and a position. In addition full-time employees have monthly salary and allowance rate; and part-time employees have contract period and hourly rate. Working schedule of both full-time employees and part-time employees is maintained on weekly bases.

Entity Sets

Entities are the principal data objects about which information is to be collected in E/R model.

Entities are usually recognizable concepts, either concrete or abstract, such as person, places, things, or events which have relevance to the database. An **entity set** is then a set consisting of the same type of entities that share same properties.

Consider the case study; some specific examples of entities are then:

EMPLOYEES, PROJECTS, CUSTOMERS ...

The candidate entities from the requirement statements are the *nouns* and the *adjective noun phrases*. The “EMPLOYEES” entity set represents all the set of employees and the “Projects” entity set represents all the set of projects.

Entities are classified as independent (Strong) or dependent (Weak).

- A **strong** (independent) entity is one that does not rely on other entities for identification.
- A **weak** (dependent) entity is one that relies on other entities for identification.

An individual occurrence of an entity set is also known as an **instance (object)**.

Attributes

Attributes are descriptive properties that are associated with an entity. A set of attributes describe an entity.

A particular instance of an attribute is called a **value**.

For example, “Employee Id” and “Name” are the attributes of the “EMPLOYEES” entity set; and

“Kevin Jones” is one value of the attribute “Name”.

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

Attributes can be classified as identifiers or descriptors.

Identifiers: more commonly called *keys*, uniquely identify an instance of an entity. Example: “Employee Id” uniquely identifies an employee entity from the entity set. **Criteria for selecting identifiers**

- ✓ Permanent
- ✓ Non-null
- ✓ Non-derived
- ✓ Simple

- **Descriptor:** describes a non-unique characteristic of an entity instance.

Example: “Name” is a descriptor for the “EMPLOYEES” entity set.

Other way of categorizing Attributes is as **Simple** and **Composite** attributes.

- **Simple Attributes:** are attributes also known as **Atomic Attributes** that cannot be divided into subparts mainly of primitive types.

Example: “Age” and “Gender” of the “EMPLOYEES” entity set.

Composite Attributes: are attributes that are composed of smaller subparts that can be subdivided into the subparts (Attributes).

Example: “Address” of the “EMPLOYEES” entity set that can be divided into “City”, “Home Address”, “Phone”, and “P.O. Box”

Another classification of attributes is based on the values that they can hold as: **Single-valued** and **Multi-valued** attributes.

✓ **Single-valued Attributes:** are attributes having only one possible value at any time.

Example: “Name” and “Gender” of the “EMPLOYEES” entity set.

✓ **Multi-valued Attributes:** are attributes that are having possibly more than one value.

Example: “Address” of the “EMPLOYEES” entity set.

Attributes can also be categorized **Stored** and **Derived** attributes.

□ **Derived Attributes:** are attributes that can be calculated from the related stored attributes, entities or general states.

Stored Attributes: on the other hand are attributes that can not be calculated in any way from the stored attributes.

Example: “Birth Date” of the “EMPLOYEES” entity set is a stored attribute, where as “Age” is a derived attribute that can be calculated from the “Birth Date” and “Current Date”.

Relationship Sets

A **Relationship** represents an association between two or more entities. An example of a relationship would be:

- “EMPLOYEES” are *Assigned* to “TEAMS”
- “CUSTOMERS” *Owns* “PROJECTS”
- “TEAMS” *works on* “PROJECTS”

A **Relationship Set** is then a set consisting same types of relationships. The entities involved in the relationship are known as **participating entities** and the function the entity plays in a relationship is called the entity’s **role**.

Example: In the *Assigned* relationship “EMPLOYEES” and “TEAMS” entity sets are the participating entity sets; and the “EMPLOYEES” entity has a role as a “*Programmer*” or

“Team Leader” in the relationship.

Relationships are classified in terms of *degree*, *connectivity*, *cardinality*, and *existence*.

Degree: The degree of a relationship is the number of entities associated with the relationship.

The n-ary (multi-way) relationship is the general form for degree n. Special cases are the binary, and ternary, where the degree is 2, and 3, respectively

Connectivity: The connectivity of a relationship describes the mapping of associated entity instances in the relationship. The values of connectivity are “one” or “many”.

Cardinality: The cardinality of a relationship is the actual number of related occurrences for each of the two entities. The basic types of connectivity for relations are: *one-to-one*, *one-to-many*, and *many-to-many*.

Existence: denotes whether the existence of an entity instance is dependent upon the existence of another, related, entity instance. The existence of an entity in a relationship is defined as either *mandatory* or *optional*.

3.5. Mapping ER Diagram to Relational Tables

ER Model, when conceptualized into diagrams, gives a good overview of entity-relationship, which is easier to understand. ER diagrams can be mapped to relational schema, that is, it is possible to create relational schema using ER diagram. We cannot import all the ER constraints into relational model, but an approximate schema can be generated.

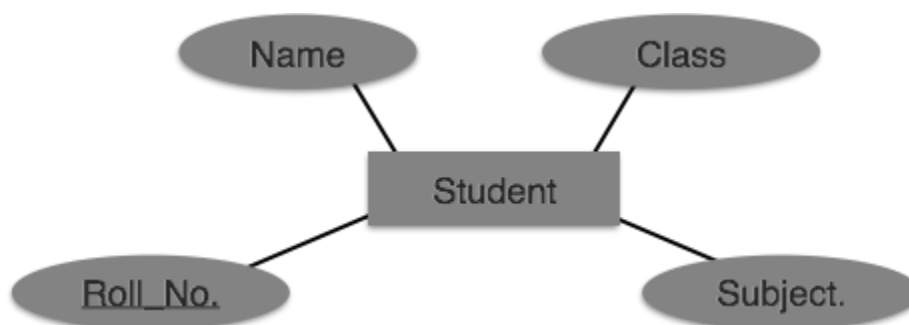
There are several processes and algorithms available to convert ER Diagrams into Relational Schema. Some of them are automated and some of them are manual. We may focus here on the mapping diagram contents to relational basics.

ER diagrams mainly comprise of –

- Entity and its attributes
- Relationship, which is association among entities.

Mapping Entity

An entity is a real-world object with some attributes.

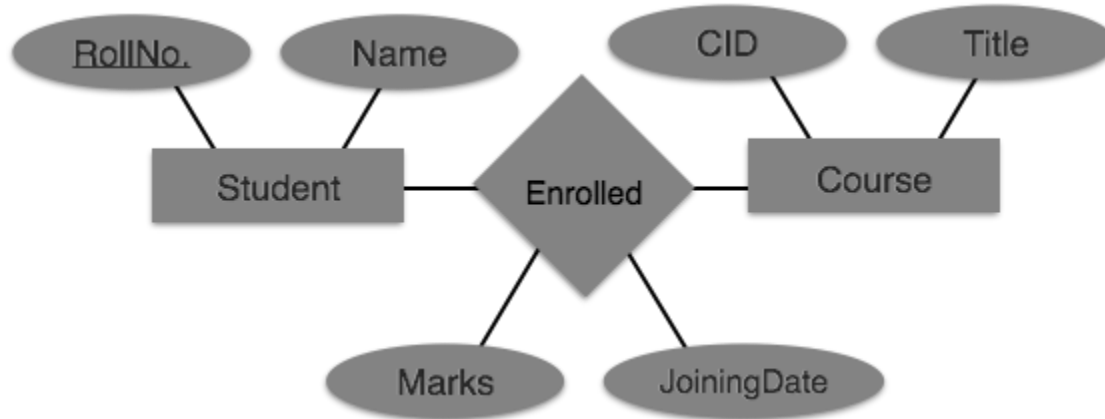


Mapping Process (Algorithm)

- Create table for each entity.
- Entity's attributes should become fields of tables with their respective data types.
- Declare primary key.

Mapping Relationship

A relationship is an association among entities.

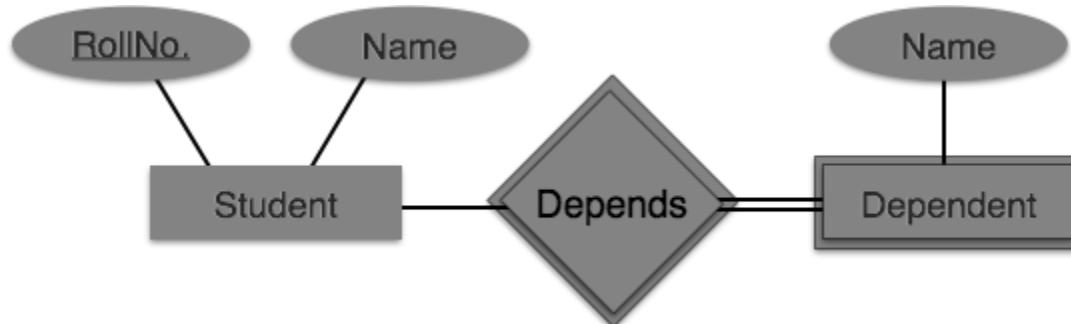


Mapping Process

- Create table for a relationship.
- Add the primary keys of all participating Entities as fields of table with their respective data types.
- If relationship has any attribute, add each attribute as field of table.
- Declare a primary key composing all the primary keys of participating entities.
- Declare all foreign key constraints.

Mapping Weak Entity Sets

A weak entity set is one which does not have any primary key associated with it.

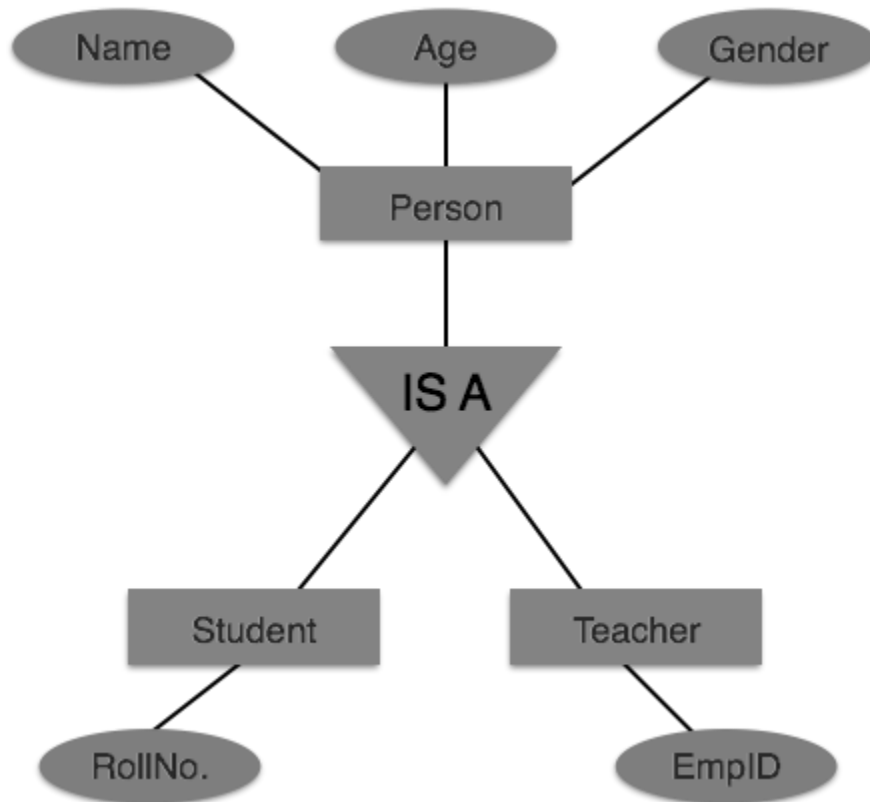


Mapping Process

- Create table for weak entity set.
- Add all its attributes to table as field.
- Add the primary key of identifying entity set.
- Declare all foreign key constraints.

Mapping Hierarchical Entities

ER specialization or generalization comes in the form of hierarchical entity sets.



Mapping Process

- Create tables for all higher-level entities.
- Create tables for lower-level entities.
- Add primary keys of higher-level entities in the table of lower-level entities.
- In lower-level tables, add all other attributes of lower-level entities.
- Declare primary key of higher-level table and the primary key for lower-level table.
- Declare foreign key constraints.

3.6. Problem with ER Models

The E-R model can result problems due to limitations in the way the entities are related in the relational databases. These problems are called connection traps. These problems often occur due to a misinterpretation of the meaning of certain relationships.

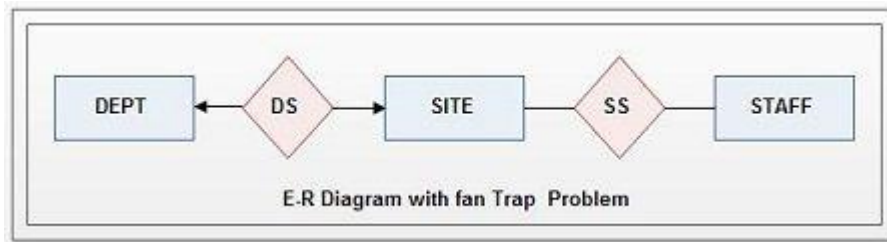
Two main types of connection traps are called fan traps and chasm traps.

Fan Trap. It occurs when a model represents a relationship between entity types, but pathway between certain entity occurrences is ambiguous.

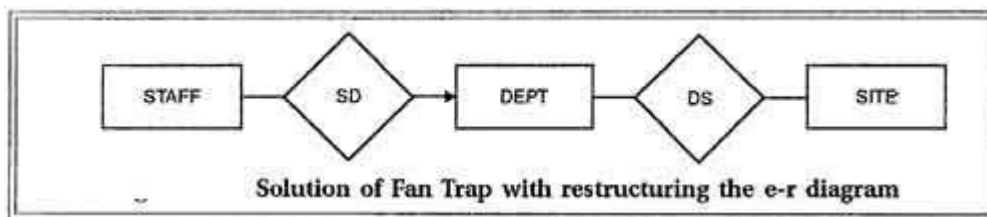
A fan trap occurs when one to many relationships fan out from a single entity.

Chasm Trap. It occurs when a model suggests the existence of a relationship between entity types, but pathway does not exist between certain entity occurrences.

For example: Consider a database of Department, Site and Staff, where one site can contain number of department, but a department is situated only at a single site. There are multiple staff members working at a single site and a staff member can work from a single site. The above case is represented in e-r diagram shown.

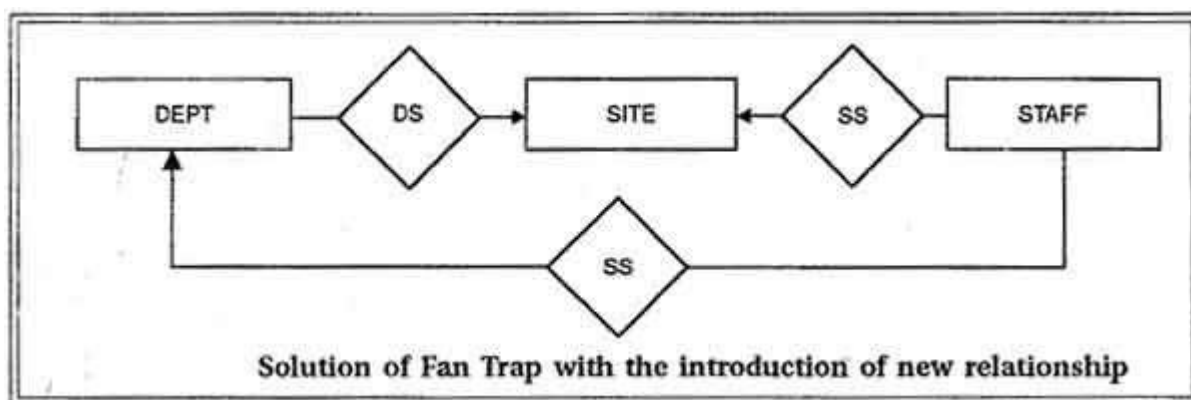


The problem of above e-r diagram is that, which staff works in a particular department remain answered. The solution is to restructure the original E-R model to represent the correct association as shown.

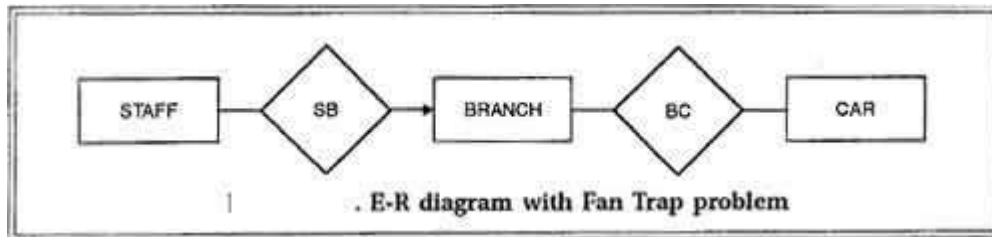


In other words the two entities should have a direct relationship between them to provide the necessary information.

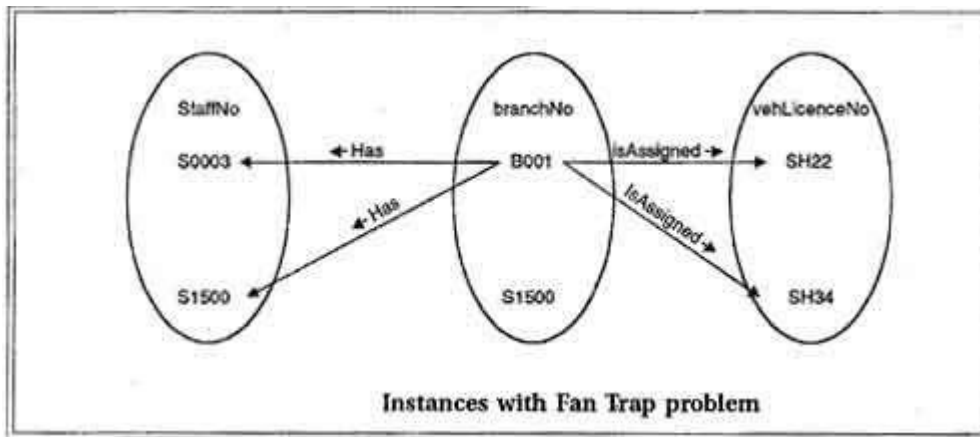
There is one another way to solve the problem of e-r diagram of figure, by introducing direct relationship between DEPT and STAFF as shown in figure.



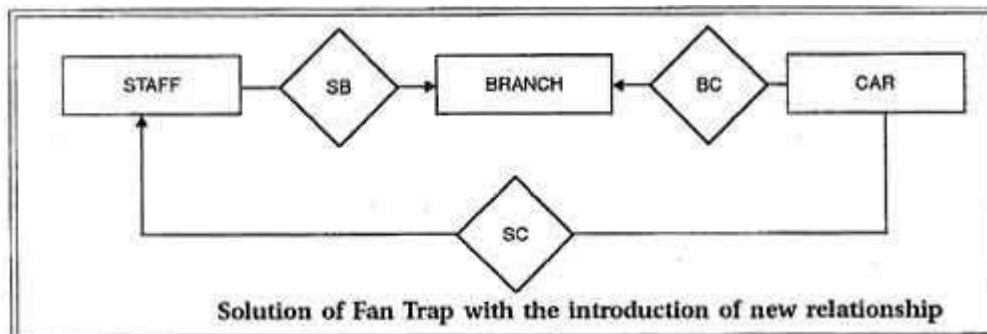
Another example: Let us consider another case, where one branch contains multiple staff members and cars, which are represented.



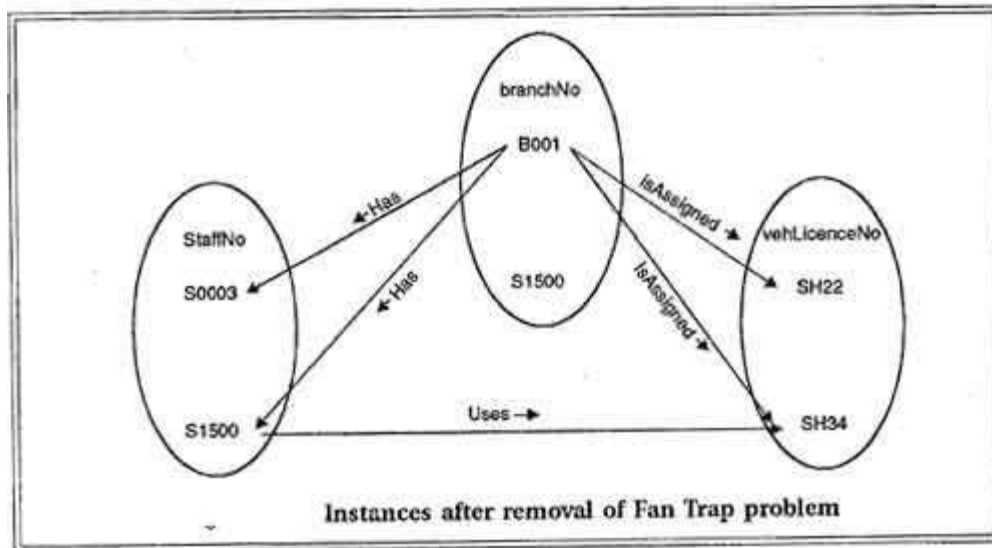
The problem of above E-R diagram is that, it is unable to tell which member of staff uses a particular, which is represented. It is not possible to tell which member of staff uses' car SH34.



The solution is to show the relationship between STAFF and CAR as shown.



With this relationship the fan trap is resolved and now it is possible to tell car SH34 is used by S1500 as shown in figure. It means it is now possible to tell which car is used by which staff.

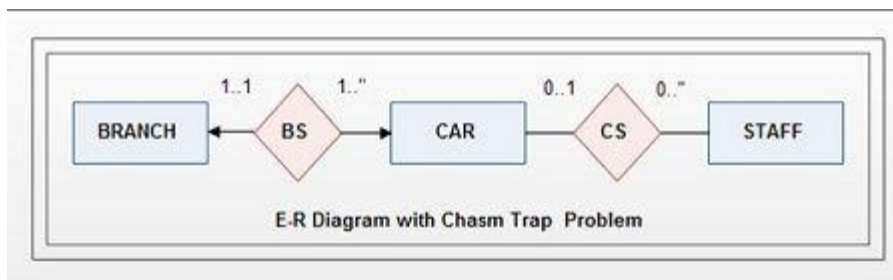


Chasm Trap

As discussed earlier, a chasm trap occurs when a model suggests the existence of a relationship between entity types, but the pathway does not exist between certain entity occurrences.

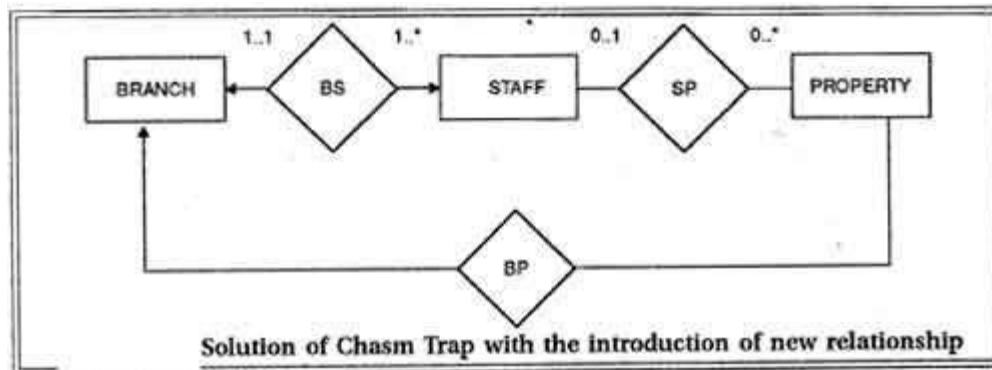
It occurs where there is a relationship with partial participation, which forms part of the pathway between entities that are related.

For example: Let us consider a database where, a single branch is allocated many staff who handles the management of properties for rent. Not all staff members handle the property and not all property is managed by a member of staff. The above case is represented in the e-r diagram.

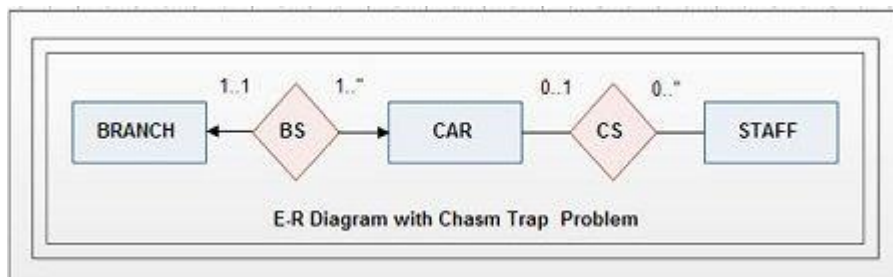


Now, the above e-r diagram is not able to represent what properties are available at a branch. The partial participation of Staff and Property in the SP relation means that some properties cannot be associated with a branch office through a member of staff.

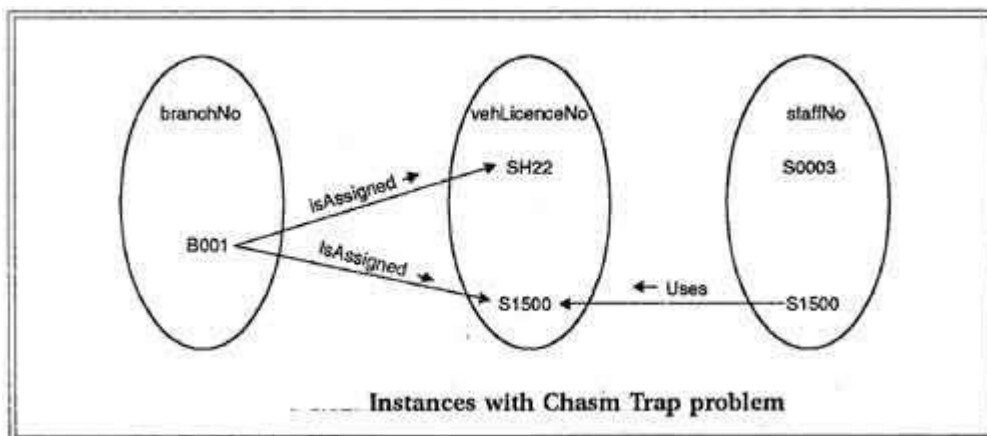
We need to add the missing relationship which is called BP between the Branch and the Property entities as shown.



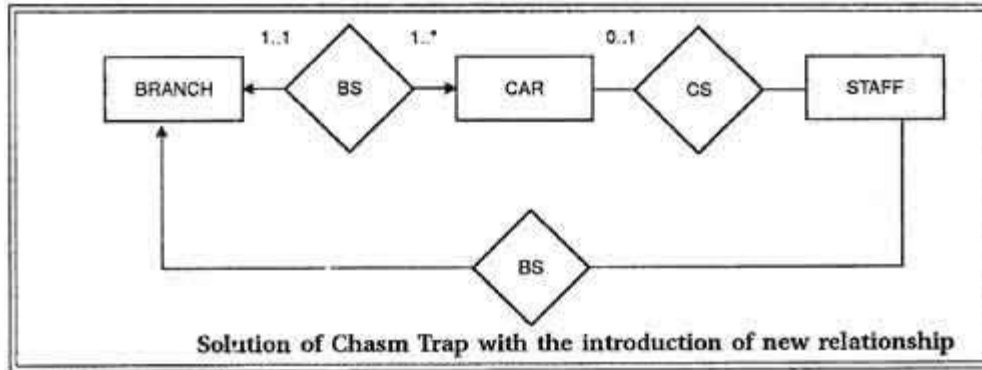
Another example: Consider another case, where a branch has multiple cars but a car can be associated with a single branch. The car is handled by a single staff and a staff can use only a single car. Some of staff members have no car available for their use. The above case is represented in E-R diagram with appropriate connectivity and cardinality.



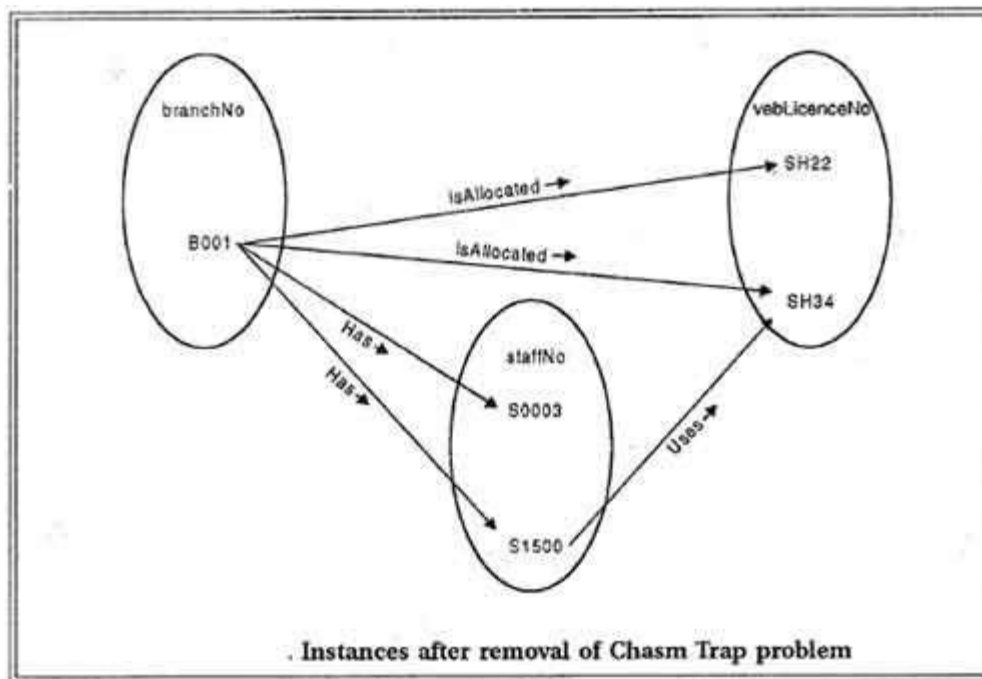
The problem of the above E-R diagram is that, it is not possible to tell in which branch staff member S0003 works at as shown.



It means the above e-r diagram is not able to represent the relationship between the BRANCH and STAFF due to the partial participation of CAR and STAFF entities. We need to add the missing relationship which is called BS between the Branch and STAFF entities as shown.



With this relationship the Chasm trap resolved and now it is possible to represent to which branch each member of staff works at, as for our example of staff S003 as shown.



3.7. Enhanced Entity Relationship (EER) Models

EER is a high-level data model that incorporates the extensions to the original ER model. **It is a diagrammatic technique for displaying the following concepts**


- Sub Class and Super Class
- Specialization and Generalization
- Union or Category
- Aggregation

These concepts are used when they come in EER schema and the resulting schema diagrams are called as EER Diagrams.

3.7.1. Features of EER Model

- EER creates a design more accurate to database schemas.
- It reflects the data properties and constraints more precisely.
- It includes all modeling concepts of the ER model.
- Diagrammatic technique helps for displaying the EER schema.
- It includes the concept of specialization and generalization.
- It is used to represent a collection of objects that is union of objects of different of different entity types.

A. Sub Class and Super Class

- Sub class and Super class relationship leads the concept of Inheritance.
- The relationship between sub class and super class is denoted with  symbol.

1. Super Class

- Super class is an entity type that has a relationship with one or more subtypes.
- An entity cannot exist in database merely by being member of any super class.

For example: Shape super class is having sub groups as Square, Circle, Triangle.

2. Sub Class

- Sub class is a group of entities with unique attributes.
- Sub class inherits properties and attributes from its super class.

For example: Square, Circle, Triangle are the sub class of Shape super class.

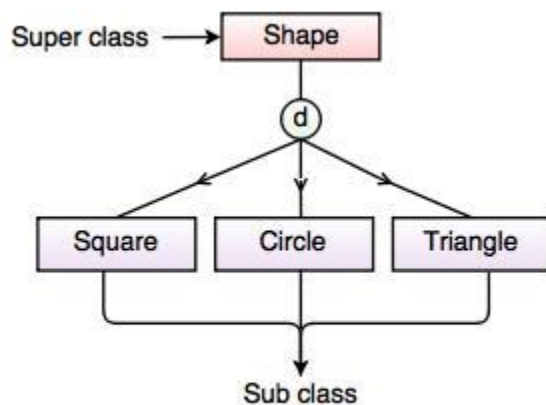


Fig. Super class/Sub class Relationship

B. Specialization and Generalization

1. Generalization

- Generalization is the process of generalizing the entities which contain the properties of all the generalized entities.
- It is a bottom approach, in which two lower level entities combine to form a higher level entity.
- Generalization is the reverse process of Specialization.
- It defines a general entity type from a set of specialized entity type.
- It minimizes the difference between the entities by identifying the common features.

For example:

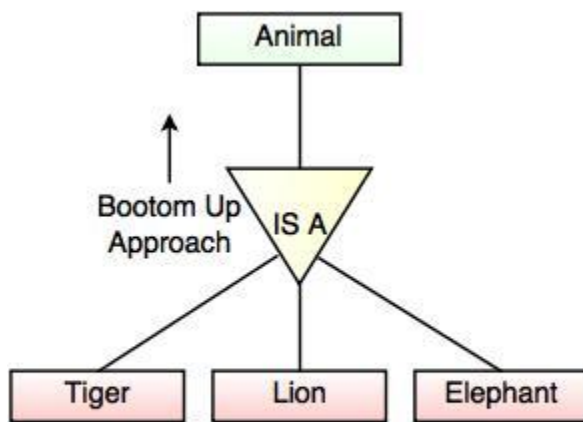


Fig. Generalization

In the above example, Tiger, Lion, Elephant can all be generalized as Animals.

2. Specialization

- Specialization is a process that defines a group entities which is divided into sub groups based on their characteristic.
- It is a top down approach, in which one higher entity can be broken down into two lower level entity.
- It maximizes the difference between the members of an entity by identifying the unique characteristic or attributes of each member.
- It defines one or more sub class for the super class and also forms the superclass/subclass relationship.

For example

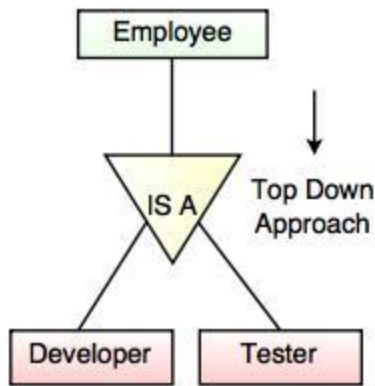


Fig. Specialization

In the above example, Employee can be specialized as Developer or Tester, based on what role they play in an Organization.

C. Category or Union

- Category represents a single super class or sub class relationship with more than one super class.
- It can be a total or partial participation.

For example Car booking, Car owner can be a person, a bank (holds a possession on a Car) or a company. Category (sub class) → Owner is a subset of the union of the three super classes → Company, Bank, and Person. A Category member must exist in at least one of its super classes.

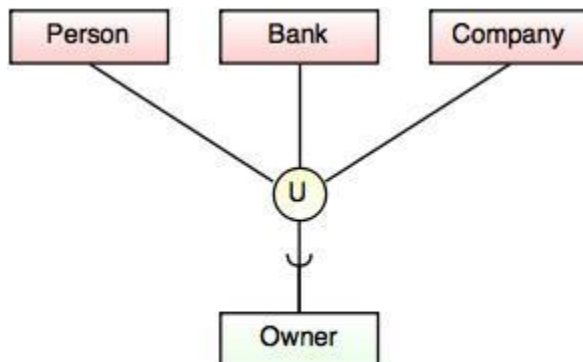


Fig. Categories (Union Type)

D. Aggregation

- Aggregation is a process that represent a relationship between a whole object and its component parts.
- It abstracts a relationship between objects and viewing the relationship as an object.
- It is a process when two entity is treated as a single entity.

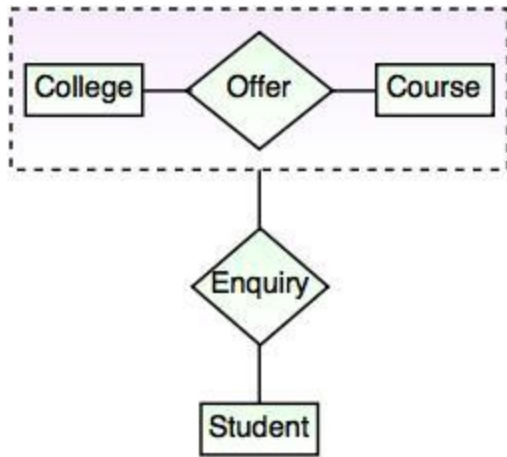


Fig. Aggregation

In the above example, the relation between College and Course is acting as an Entity in Relation with Student.

CHAPTER FOUR

4. Logical Database Design

4.1. Introduction

Logical data modeling is one of three types or stages of database design or modeling, along with conceptual and physical database design or modeling. Sometimes referred to as information modeling, logical data modeling is the second of these stages. It helps organizations develop a visual understanding of the information they must process to successfully complete specific tasks or business processes.

Logical database design is the process of deciding how to arrange the attributes of the entities in a given business environment into database structures, such as the tables of a relational database. The goal of logical database design is to create well-structured tables that properly reflect the company's business environment. The tables will be able to store data about the company's entities in a non-redundant manner and foreign keys will be placed in the tables so that all the relationships among the entities will be supported. As a graphical representation of the information requirements for a given business area, a logical data model is constructed by taking the data descriptions depicted in a conceptual data model and introducing associated elements, definitions and greater context for the data's structure.

4.2. Logical Database Design for Relational Model

Converting ER Diagram to Relational Tables

Three basic rules to convert ER into tables or relations:

Rule 1: Entity Names will automatically be table names

Rule 2: Mapping of attributes: attributes will be columns of the respective tables.

- Atomic or single-valued or derived or stored attributes will be columns
 - Composite attributes: the parent attribute will be ignored and the decomposed attributes (child attributes) will be columns of the table.

- Multi-valued attributes: will be mapped to a new table where the primary key of the main table will be posted for cross referencing.

Rule 3: Relationships: relationship will be mapped by using a foreign key attribute. Foreign key is a primary or candidate key of one relation used to create association between tables.

For a relationship with One-to-One Cardinality: post the primary or candidate key of one of the table into the other as a foreign key. In cases where one entity is having partial participation on the relationship, it is recommended to post the candidate key of the partial participants to the total participant so as to save some memory location due to null values on the foreign key attribute. E.g.: for a relationship between Employee and Department where employee manages a department, the cardinality is one-to-one as one employee will manage only one department and one department will have one manager. here the PK of the Employee can be posted to the Department or the PK of the Department can be posted to the Employee. But the Employee is having partial participation on the relationship “Manages” as not all employees are managers of departments. thus, even though both way is possible, it is recommended to post the primary key of the employee to the Department table as a foreign key.

For a relationship with One-to-Many Cardinality: Post the primary key or candidate key from the —one|| side as a foreign key attribute to the —many|| side. E.g.: For a relationship called —*Belongs To*|| between Employee (Many) and Department (One) the primary or candidate key of the one side which is Department should be posted to the many side which is Employee table.

For a relationship with Many-to-Many Cardinality: for relationships having many to many cardinality, one has to create a new table (which is the associative entity) and post primary key or candidate key from the participant entities as foreign key attributes in the new table along with some additional attributes (if applicable). The same approach should be used for relationships with degree greater than binary.

For a relationship having Associative Entity property: in cases where the relationship has its own attributes (associative entity), one has to create a new table for the associative entity and post primary key or candidate key from the participating entities as foreign key attributes in the new table.

4.3. Normalization

Database normalization is a series of steps followed to obtain a database design that allows for consistent storage and efficient access of data in a relational database. These steps reduce data redundancy and the risk of data becoming inconsistent. **NORMALIZATION** is the process of identifying the logical associations between data items and designing a database that will

represent such associations but without suffering the update anomalies which are; Insertion Anomalies, Deletion Anomalies and Modification Anomalies.

NORMALIZATION is:-

- Is a formal process for deciding which attributes should be grouped together in a relation.
- Is the process of successively reducing relations with anomalies to produce smaller, well-structured relations. Following are some of the main goals of normalization:
 1. Minimize data redundancy, thereby avoiding anomalies and conserving storage space.
 2. Simplify the enforcement of referential integrity constraints.
 3. Make it easier to maintain data (insert, update and delete).
 4. Provide a better design that is an improved representation of the real world and a stronger basis for future growth.

Steps in Normalization

A **normal form** is a state of a relation that results from applying simple rules regarding functional dependencies for relationships between attributes to that relation.

1. **First normal form**- Any multivalued attributes (also called repeating groups) have been removed, so there is a single value (possibly null) at the intersection of each row and column of the table.
2. **Second normal form** – Any partial functional dependencies have been removed (i. e., nonkeys are identified by the whole primary key).
3. **Third normal form**- Any transitive dependencies have been removed (i. e., nonkeys are identified by only the primary key).
4. **Boyce- Codd normal form** – Any remaining anomalies that result from functional dependencies have been removed (because there was more than one primary key for the same nonkeys).
5. **Fourth normal form** – Any multivalued dependencies have been removed.
6. **Fifth normal form** – Any remaining anomalies have been removed.

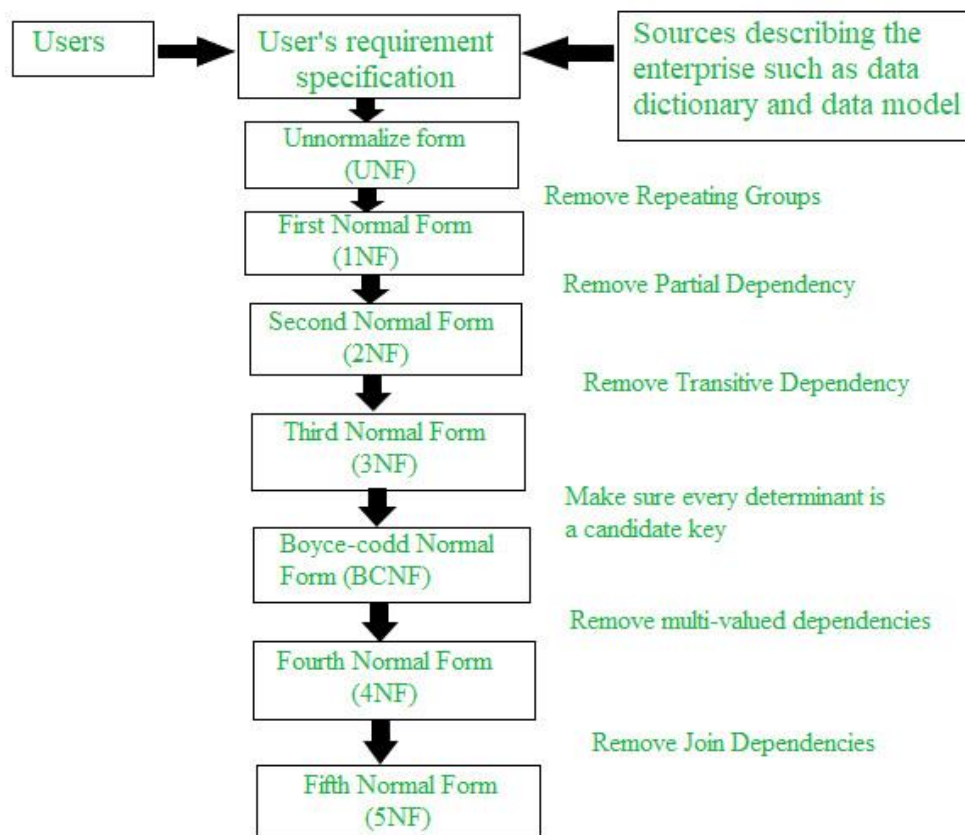


Fig 1:- Normalization in figure format

Summary of Normalization in a Nutshell :

Normal Form	Test	Remedy (Normalization)
1NF	Relation should have no non-atomic attributes or nested relations.	Form name relation for each non-atomic attribute or nested relation.
2NF	For relations where primary key contains multiple attributes, no non-key attributes should be functionally dependent on a part of the primary key.	Decompose and set up a new relation for each partial key with its dependent attributes. Make sure to keep a relation with the original primary key and any attributes that are fully functionally dependent on it.

3NF	Relation should not have a non-key attribute functionally determined by another non-key attribute (or by a sets of non-key attributes) i.e., there should be no transitive dependency of a non-key attribute of the primary key.	Decompose and set up a relation that includes the non-key attribute(s) that functionally determine(s) other non-key attribute(s).
BCNF	Relation should not have any attribute in Functional Dependency which is non-prime, the attribute that doesn't occur in any candidate key.	Make sure that the left side of every functional dependency is a candidate key.
4NF	The relation should not have a multi-value dependency means it occur when two attributes of a table are independent of each other but both depend on a third attribute.	Decompose the table into two subtables.
5NF	The relation should not have join dependency means if a table can be recreated by joining multiple tables and each of the tables has a subset of the attributes of the table, then the table is in Join Dependency.	Decompose all the tables into as many as possible numbers in order to avoid dependency.

Normalization may reduce system performance since data will be cross referenced from many tables. Thus denormalization is sometimes used to improve performance, at the cost of reduced consistency guarantees.

All the normalization rules will eventually remove the update anomalies that may exist during data manipulation after the implementation. The update anomalies are;

The type of problems that could occur in insufficiently normalized table is called update anomalies which includes;

- **Insertion anomalies**

An “insertion anomaly” is a failure to place information about a new database entry into all the places in the database where information about that new entry needs to be stored. *Additionally, we may have difficulty to insert some data.* In a properly normalized database, information about a new entry needs to be inserted into only one place in the database; in an inadequately

normalized database, information about a new entry may need to be inserted into more than one place and, human fallibility being what it is, some of the needed additional insertions may be missed.

- **Deletion anomalies**

- “deletion anomaly” is a failure to remove information about an existing database entry when it is time to remove that entry. *Additionally, deletion of one data may result in lose of other information.* In a properly normalized database, information about an old, to-be-gotten-rid-of entry needs to be deleted from only one place in the database; in an inadequately normalized database, information about that old entry may need to be deleted from more than one place, and, human fallibility being what it is, some of the needed additional deletions may be missed.

- **Modification anomalies**

- modification of a database involves changing some value of the attribute of a table. In a properly normalized database table, what ever information is modified by the user, the change will be effected and used accordingly.

In order to avoid the update anomalies we in a given table, the solution is to decompose it to smaller tables based on the rule of normalization. However, the decomposition has *two important properties*

- The **Lossless-join** property insures that any instance of the original relation can be identified from the instances of the smaller relations.
- The **Dependency preservation** property implies that constraint on the original dependency can be maintained by enforcing some constraints on the smaller relations. i.e. we don’t have to perform Join operation to check whether a constraint on the original relation is violated or not.

The purpose of normalization is to reduce the chances for anomalies to occur in a database

Example of problems related with Anomalies

<i>EmpID</i>	<i>FName</i>	<i>LName</i>	<i>SkillID</i>	<i>Skill</i>	<i>SkillType</i>	<i>School</i>	<i>SchoolAdd</i>	<i>Skill Level</i>
12	Abebe	Mekuria	2	SQL	Database	AAU	Sidist_Kilo	5
16	Lemma	Alemu	5	C++	Programming	Unity	Gerji	6
28	Chane	Kebede	2	SQL	Database	AAU	Sidist_Kilo	10
25	Abera	Taye	6	VB6	Programming	Helico	Piazza	8
65	Almaz	Belay	2	SQL	Database	Helico	Piazza	9
24	Dereje	Tamiru	8	Oracle	Database	Unity	Gerji	5
51	Selam	Belay	4	Prolog	Programming	Jimma	Jimma City	8
94	Alem	Kebede	3	Cisco	Networking	AAU	Sidist_Kilo	7
18	Girma	Dereje	1	IP	Programming	Jimma	Jimma City	4
13	Yared	Gizaw	7	Java	Programming	AAU	Sidist_Kilo	6

Deletion Anomalies:

If employee with **ID 16** is deleted then ever information about skill **C++** and the type of skill is deleted from the database. Then we will not have any information about **C++** and its skill type.

Insertion Anomalies:

What if we have a new employee with a skill called **Pascal**? We can not decide weather **Pascal** is allowed as a value for skill and we have no clue about the *type of skill* that **Pascal** should be categorized as.

Modification Anomalies:

What if the address for **Helico** is changed from **Piazza** to **Mexico**? We need to look for every occurrence of **Helico** and change the value of **School_Add** from **Piazza** to **Mexico**, which is prone to error.

Database-management system can work only with the information that we put explicitly into its tables for a given database and into its rules for working with those tables, where such rules are appropriate and possible.

Functional Dependency (FD)

Before moving to the definition and application of normalization, it is important to have an understanding of “functional dependency.”

Data Dependency

The logical associations between data items that point the database designer in the direction of a good database design are referred to as determinant or dependent relationships.

Two data items A and B are said to be in a determinant or dependent relationship if certain values of data item B always appears with certain values of data item A. if the data item A is the determinant data item and B the dependent data item then the direction of the association is from A to B and not vice versa.

The essence of this idea is that if the existence of something, call it A, implies that B must exist and have a certain value, then we say that “**B is functionally dependent on A.**” We also often express this idea by saying that “A functionally determines B,” or that “B is a function of A,” or that “A functionally governs B.” Often, the notions of functionality and functional dependency are expressed briefly by the statement, “If A, then B.” It is important to note that the value of B must be *unique* for a given value of A, i.e., any given value of A must imply just one and only one value of B, in order for the relationship to qualify for the name “function.” (However, this does not necessarily prevent different values of A from implying the same value of B.)

However, for the purpose of normalization, we are interested in finding 1..1 (one to one) dependencies, lasting for all times (*intension* rather than *extension* of the database), and the determinant having the minimal number of attributes.

X à Y holds if whenever two tuples have the same value for X, they must have the same value for Y

The notation is: $A \rightarrow B$ which is read as; B is functionally dependent on A

In general, a ***functional dependency*** is a relationship among attributes. In relational databases, we can have a determinant that governs one or several other attributes.

FDs are derived from the real-world constraints on the attributes and they are properties on the database intension **not** extension.

Example

Dinner Course

Meat

Fish

Cheese

Type of Wine

Red

White

Rose

Since the type of ***Wine*** served depends on the type of ***Dinner***, we say ***Wine*** is functionally dependent on ***Dinner***.

Dinner à Wine

Dinner Course

Meat

Fish

Cheese

Type of Wine

Red

White

Rose

Type of Fork

Meat fork

Fish fork

Cheese fork

Since both ***Wine*** type and ***Fork*** type are determined by the ***Dinner*** type, we say ***Wine*** is functionally dependent on ***Dinner*** and ***Fork*** is functionally dependent on ***Dinner***.

Dinner à Wine

Dinner à Fork

Partial Dependency

If an attribute which is not a member of the primary key is dependent on some part of the primary key (if we have composite primary key) then that attribute is partially functionally dependent on the primary key.

Let {A,B} is the Primary Key and C is no key attribute.

Then if {A,B} → C and B → C

Then C is partially functionally dependent on {A,B}

Full Functional Dependency

If an attribute which is not a member of the primary key is not dependent on some part of the primary key but the whole key (if we have composite primary key) then that attribute is fully functionally dependent on the primary key.

Let {A,B} be the Primary Key and C is a non- key attribute

Then if {A,B} \rightarrow C and B \rightarrow C and A \rightarrow C does not hold

Then C Fully functionally dependent on {A,B}

Transitive Dependency

In mathematics and logic, a transitive relationship is a relationship of the following form: “If A implies B, and if also B implies C, then A implies C.”

Example:

If Mr X is a Human, and if every Human is an Animal, then Mr X must be an Animal.

Generalized way of describing transitive dependency is that:

If A functionally governs B, AND

If B functionally governs C

THEN A functionally governs C

Provided that neither C nor B determines A i.e. (B \nrightarrow A and C \nrightarrow A) In the normal notation:

$\{(A \rightarrow B) \text{ AND } (B \rightarrow C)\} \implies A \rightarrow C$ provided that **$B \nrightarrow A$ and $C \nrightarrow A$**

4.4. Pitfalls of Normalization

There are a few drawbacks in normalization :

1. Creating a longer task, because there are more tables to join, the need to join those tables increases and the task become more tedious (longer and slower). The database become harder to realize as well.
2. Tables will contain codes rather than real data as the repeated data will be stored as lines of codes rather than the true data. Therefore, there is always a need to go to the lookup table, thus the system will be slower to perform.
3. Making query more difficult, because it consists of an SQL that is constructed dynamically and is usually constructed by desktop friendly query tools, hence it is hard to model the database without knowing what the customers desires.

4. Requires Detailed Analysis and Design, Normalizing a database is a complex and difficult task, because analyst have to know the purpose of the database, such as whether it should be optimized for reading data, writing data or both, also affects how it is normalized. A poorly normalized database may perform badly and store data inefficiently.

4.5. Denormalization

Denormalization is a strategy used on a previously-normalized database to increase performance. The idea behind it is to add redundant data where we think it will help us the most. It is a process of trying to improve the read performance of a database, at the expense of losing some write performance, by adding redundant copies of data or by grouping data.

Denormalization refers to a refinement to relational schema such that the degree of normalization for a modified relation is less than the degree of at least one of the original relations.

It refers to situations where two relations are combined into one new relation, which is still normalized but contains more nulls than original relations.

Things that needs to be considered when doing denormalization :

1. Makes implementation more complex, because there will be more redundant data and anomalies that will effect the implementation process.
2. Often sacrifices flexibility.
3. May speed up retrievals but it slows down updates, data redundancy will slow the data updates because there will be more tables to be updated.

When to use Denormalization :

1. **Maintaining history**
2. **Improving query performance**, Some of the queries may use multiple tables to access data that we frequently need
3. **Speeding up reporting**, We need certain statistics very frequently. Creating them from live data is quite time-consuming and can affect overall system performance.
4. **Computing commonly-needed values up front**, We want to have some values ready-computed so we don't have to generate them in real time.

CHAPTER FIVE

5. Physical Database Design

5.1. What is physical database design process?

Physical database design is **the process of transforming logical data models into physical data models**. An experienced database designer will make a physical database design in parallel with conceptual data modeling if they know the type of database technology that will be used. Conceptual data modeling is about understanding the organization and gathering the right requirements. Physical database design, on the other hand, is about creating stable database structures correctly expressing the requirements in a technical language.

The physical design of your database optimizes performance while ensuring data integrity by avoiding unnecessary data redundancies. During physical design, you transform the entities into tables, the instances into rows, and the attributes into columns.

After completing the logical design of your database, you now move to the physical design. You and your colleagues need to make many decisions that affect the physical design, some of which are listed below.

- How to translate entities into physical tables
- What attributes to use for columns of the physical tables
- Which columns of the tables to define as keys
- What indexes to define on the tables
- What views to define on the tables
- How to denormalize the tables
- How to resolve many-to-many relationships
- What designs can take advantage of hash access

5.2. Moving from Logical to Physical Design

How can we translate logical database design for target RDBMS?

Logical design is what you draw with a pen and paper or design with Oracle Warehouse Builder or Oracle Designer before building your data warehouse. Physical design is the creation of the database with SQL statements.

During the physical design process, you convert the data gathered during the logical design phase into a description of the physical database structure. Physical design decisions are mainly driven by query performance and database maintenance aspects. For example, choosing a partitioning strategy that meets common query requirements enables Oracle Database to take advantage of partition pruning, a way of narrowing a search before performing it.

Comparison between Logical and Physical Design:

During the logical design phase, you defined a model for your data warehouse consisting of entities, attributes, and relationships. The entities are linked together using relationships. Attributes are used to describe the entities. The unique identifier (UID) distinguishes between one instance of an entity and another.

Logical Design	Physical Design
Entity	Table
Relationship	Foreign Key
Attribute	Column
Unique Identifier	Primary Key

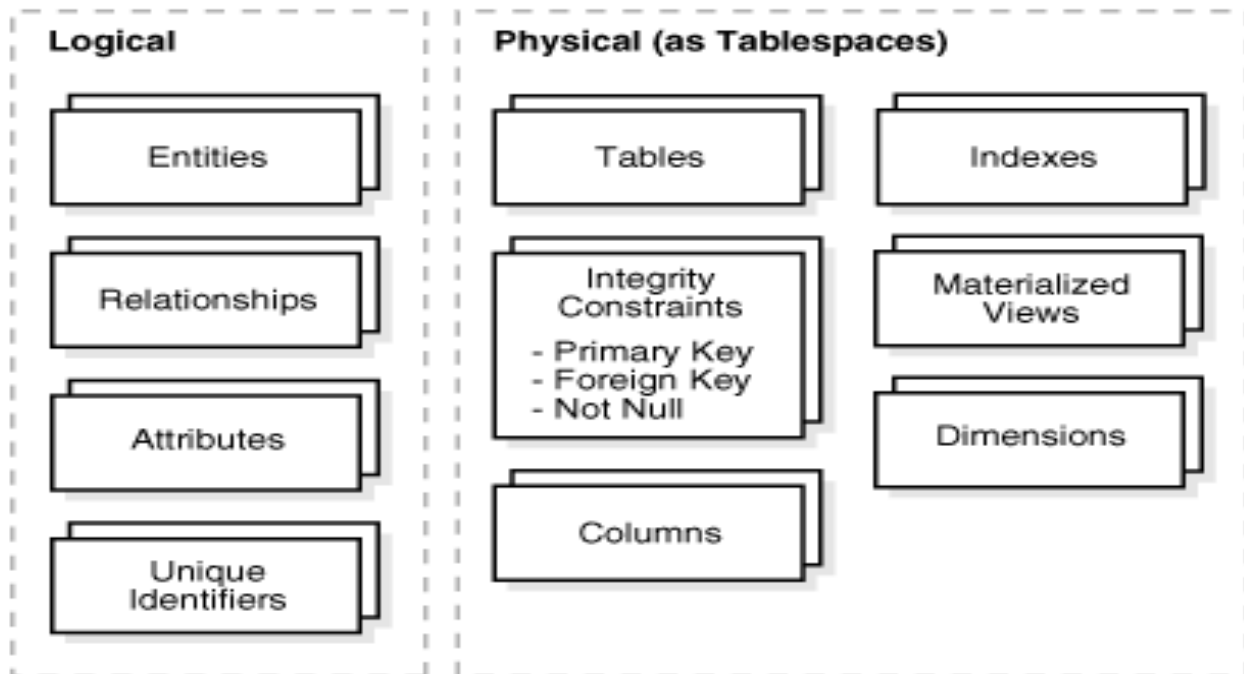
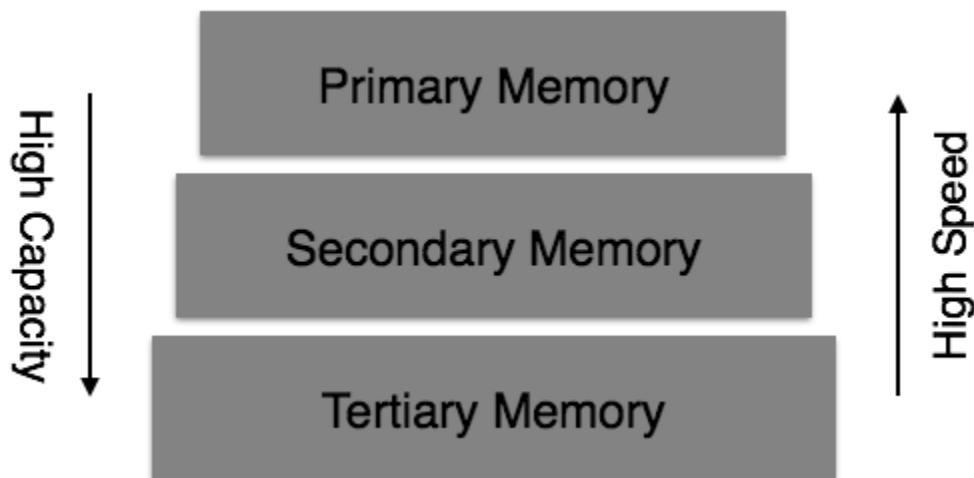


Figure 1 Logical Design Compared with Physical Design

5.3. DBMS - Storage System

Databases are stored in file formats, which contain records. At physical level, the actual data is stored in electromagnetic format on some device. These storage devices can be broadly categorized into three types:-



- **Primary Storage** – The memory storage that is directly accessible to the CPU comes under this category. CPU's internal memory (registers), fast memory (cache), and main memory (RAM) are directly accessible to the CPU, as they are all placed on the motherboard or CPU chipset. This storage is typically very small, ultra-fast, and volatile.

Primary storage requires continuous power supply in order to maintain its state. In case of a power failure, all its data is lost.

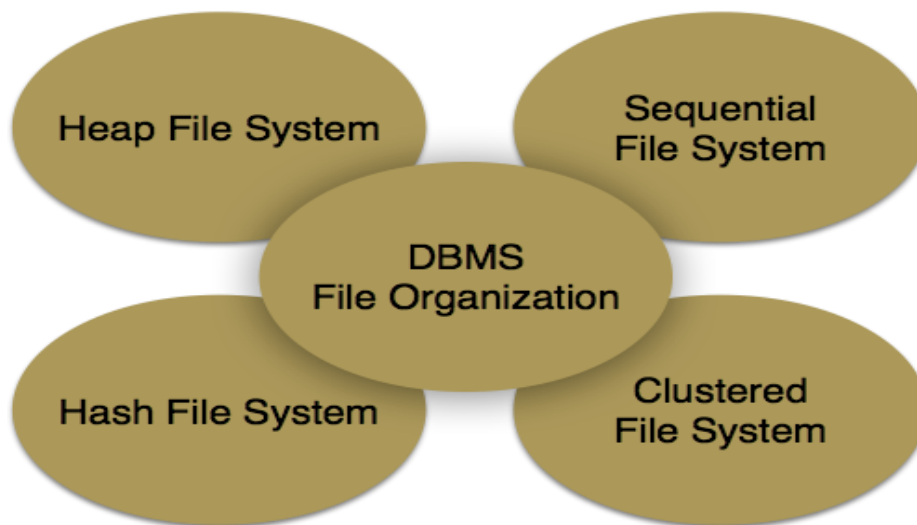
- **Secondary Storage** – Secondary storage devices are used to store data for future use or as backup. Secondary storage includes memory devices that are not a part of the CPU chipset or motherboard, for example, magnetic disks, optical disks (DVD, CD, etc.), hard disks, flash drives, and magnetic tapes.
- **Tertiary Storage** – Tertiary storage is used to store huge volumes of data. Since such storage devices are external to the computer system, they are the slowest in speed. These storage devices are mostly used to take the back up of an entire system. Optical disks and magnetic tapes are widely used as tertiary storage.

5.4. DBMS - File Structure

Relative data and information is stored collectively in file formats. A file is a sequence of records stored in binary format. A disk drive is formatted into several blocks that can store records. File records are mapped onto those disk blocks.

5.5. File Organization

File Organization defines how file records are mapped onto disk blocks. We have four types of File Organization to organize file records –



5.5.1. Heap File Organization

When a file is created using Heap File Organization, the Operating System allocates memory area to that file without any further accounting details. File records can be placed anywhere in that memory area. It is the responsibility of the software to manage the records. Heap File does not support any ordering, sequencing, or indexing on its own.

5.5.2. Sequential File Organization

Every file record contains a data field (attribute) to uniquely identify that record. In sequential file organization, records are placed in the file in some sequential order based on the unique key

field or search key. Practically, it is not possible to store all the records sequentially in physical form.

Hash File Organization

Hash File Organization uses Hash function computation on some fields of the records. The output of the hash function determines the location of disk block where the records are to be placed.

Clustered File Organization

Clustered file organization is not considered good for large databases. In this mechanism, related records from one or more relations are kept in the same disk block, that is, the ordering of records is not based on primary key or search key.

File Operations

Operations on database files can be broadly classified into two categories –

- **Update Operations**
- **Retrieval Operations**

Update operations change the data values by insertion, deletion, or update. Retrieval operations, on the other hand, do not alter the data but retrieve them after optional conditional filtering. In both types of operations, selection plays a significant role. Other than creation and deletion of a file, there could be several operations, which can be done on files.

- **Open** – A file can be opened in one of the two modes, **read mode** or **write mode**. In read mode, the operating system does not allow anyone to alter data. In other words, data is read only. Files opened in read mode can be shared among several entities. Write mode allows data modification. Files opened in write mode can be read but cannot be shared.
- **Locate** – Every file has a file pointer, which tells the current position where the data is to be read or written. This pointer can be adjusted accordingly. Using find (seek) operation, it can be moved forward or backward.
- **Read** – By default, when files are opened in read mode, the file pointer points to the beginning of the file. There are options where the user can tell the operating system where to locate the file pointer at the time of opening a file. The very next data to the file pointer is read.
- **Write** – User can select to open a file in write mode, which enables them to edit its contents. It can be deletion, insertion, or modification. The file pointer can be located at the time of opening or can be dynamically changed if the operating system allows to do so.
- **Close** – This is the most important operation from the operating system's point of view. When a request to close a file is generated, the operating system
 - removes all the locks (if in shared mode),
 - saves the data (if altered) to the secondary storage media, and
 - releases all the buffers and file handlers associated with the file.

The organization of data inside a file plays a major role here. The process to locate the file pointer to a desired record inside a file varies based on whether the records are arranged sequentially or clustered.

5.6. DBMS - Indexing

We know that data is stored in the form of records. Every record has a key field, which helps it to be recognized uniquely.

Indexing is a data structure technique to efficiently retrieve records from the database files based on some attributes on which the indexing has been done. Indexing in database systems is similar to what we see in books.

Indexing is defined based on its indexing attributes. Indexing can be of the following types –

- **Primary Index** – Primary index is defined on an ordered data file. The data file is ordered on a **key field**. The key field is generally the primary key of the relation.
- **Secondary Index** – Secondary index may be generated from a field which is a candidate key and has a unique value in every record, or a non-key with duplicate values.
- **Clustering Index** – Clustering index is defined on an ordered data file. The data file is ordered on a non-key field.

Ordered Indexing is of two types –

- I. Dense Index
- II. Sparse Index

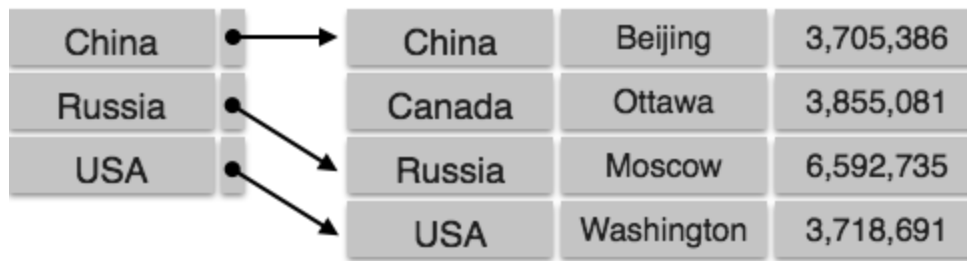
I. Dense Index

In dense index, there is an index record for every search key value in the database. This makes searching faster but requires more space to store index records itself. Index records contain search key value and a pointer to the actual record on the disk.



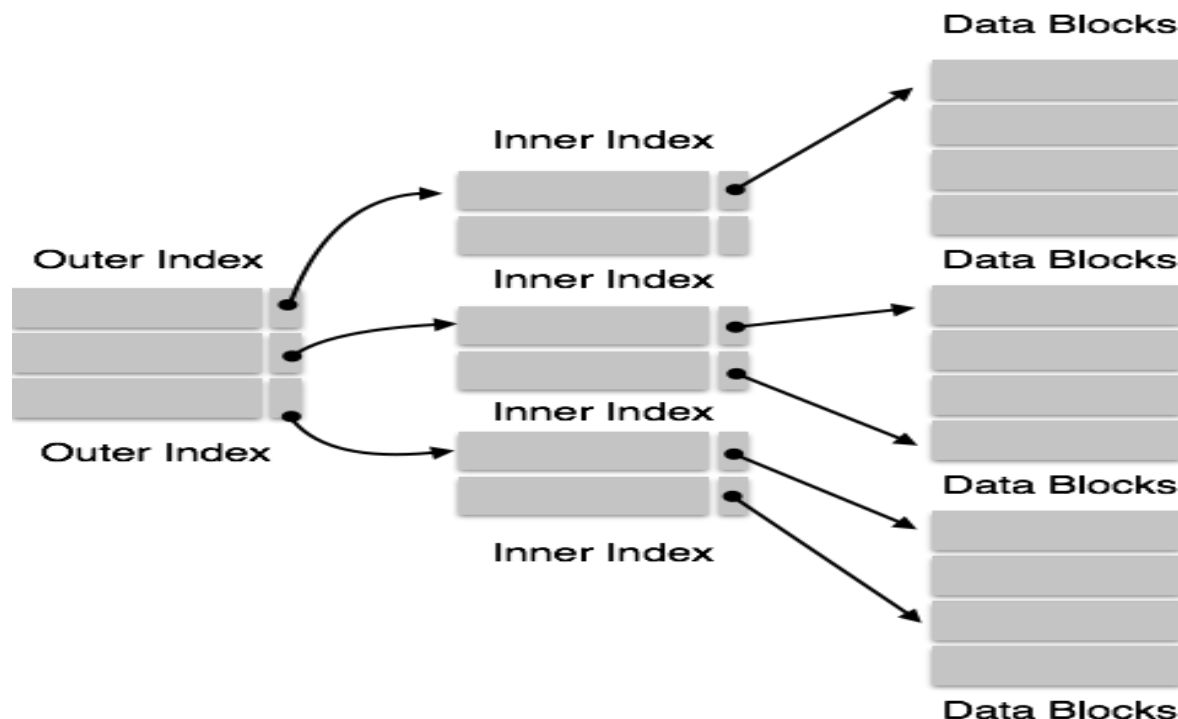
II. Sparse Index

In sparse index, index records are not created for every search key. An index record here contains a search key and an actual pointer to the data on the disk. To search a record, we first proceed by index record and reach at the actual location of the data. If the data we are looking for is not where we directly reach by following the index, then the system starts sequential search until the desired data is found.



Multilevel Index

Index records comprise search-key values and data pointers. Multilevel index is stored on the disk along with the actual database files. As the size of the database grows, so does the size of the indices. There is an immense need to keep the index records in the main memory so as to speed up the search operations. If single-level index is used, then a large size index cannot be kept in memory which leads to multiple disk accesses.



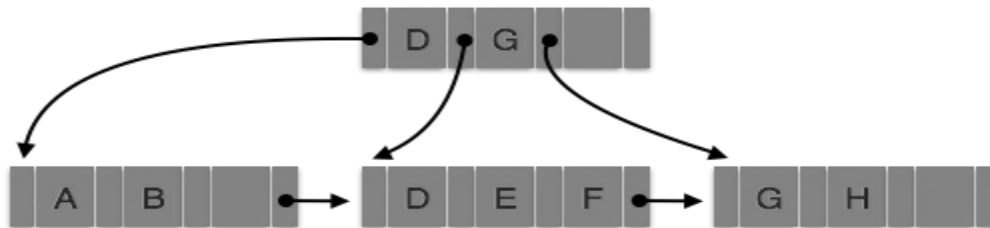
Multi-level Index helps in breaking down the index into several smaller indices in order to make the outermost level so small that it can be saved in a single disk block, which can easily be accommodated anywhere in the main memory.

B⁺ Tree

A B⁺ tree is a balanced binary search tree that follows a multi-level index format. The leaf nodes of a B⁺ tree denote actual data pointers. B⁺ tree ensures that all leaf nodes remain at the same height, thus balanced. Additionally, the leaf nodes are linked using a link list; therefore, a B⁺ tree can support random access as well as sequential access.

Structure of B⁺ Tree

Every leaf node is at equal distance from the root node. A B⁺ tree is of the order **n** where **n** is fixed for every B⁺ tree.



Internal nodes –

- Internal (non-leaf) nodes contain at least $\lceil n/2 \rceil$ pointers, except the root node.
- At most, an internal node can contain **n** pointers.

Leaf nodes –

- Leaf nodes contain at least $\lceil n/2 \rceil$ record pointers and $\lceil n/2 \rceil$ key values.
- At most, a leaf node can contain **n** record pointers and **n** key values.
- Every leaf node contains one block pointer **P** to point to next leaf node and forms a linked list.

B⁺ Tree Insertion

- B⁺ trees are filled from bottom and each entry is done at the leaf node.
- If a leaf node overflows –
 - Split node into two parts.
 - Partition at $i = \lfloor (m+1)/2 \rfloor$.
 - First **i** entries are stored in one node.
 - Rest of the entries (**i**+1 onwards) are moved to a new node.
 - **ith** key is duplicated at the parent of the leaf.
- If a non-leaf node overflows –
 - Split node into two parts.
 - Partition the node at $i = \lfloor (m+1)/2 \rfloor$.
 - Entries up to **i** are kept in one node.
 - Rest of the entries are moved to a new node.

B⁺ Tree Deletion

- B⁺ tree entries are deleted at the leaf nodes.
- The target entry is searched and deleted.
 - If it is an internal node, delete and replace with the entry from the left position.
- After deletion, underflow is tested,
 - If underflow occurs, distribute the entries from the nodes left to it.
- If distribution is not possible from left, then
 - Distribute from the nodes right to it.
- If distribution is not possible from left or from right, then
 - Merge the node with left and right to it.

5.7. DBMS - Hashing

For a huge database structure, it can be almost next to impossible to search all the index values through all its level and then reach the destination data block to retrieve the desired data. Hashing is an effective technique to calculate the direct location of a data record on the disk without using index structure.

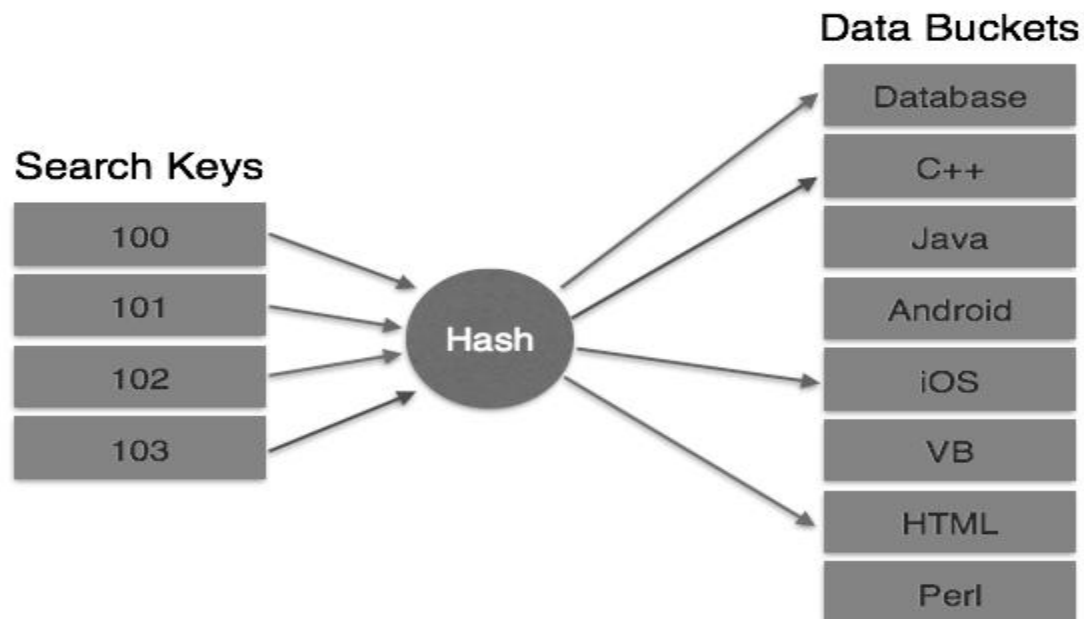
Hashing uses hash functions with search keys as parameters to generate the address of a data record.

Hash Organization

- **Bucket** – A hash file stores data in bucket format. Bucket is considered a unit of storage. A bucket typically stores one complete disk block, which in turn can store one or more records.
- **Hash Function** – A hash function, **h**, is a mapping function that maps all the set of search-keys **K** to the address where actual records are placed. It is a function from search keys to bucket addresses.

Static Hashing

In static hashing, when a search-key value is provided, the hash function always computes the same address. For example, if mod-4 hash function is used, then it shall generate only 5 values. The output address shall always be same for that function. The number of buckets provided remains unchanged at all times.



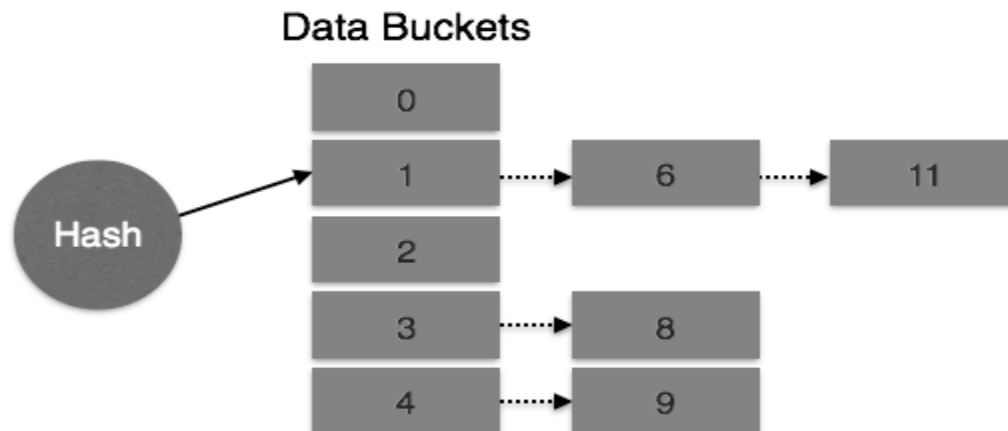
Operation

- **Insertion** – When a record is required to be entered using static hash, the hash function h computes the bucket address for search key K , where the record will be stored. Bucket address = $h(K)$
- **Search** – When a record needs to be retrieved, the same hash function can be used to retrieve the address of the bucket where the data is stored.
- **Delete** – This is simply a search followed by a deletion operation.

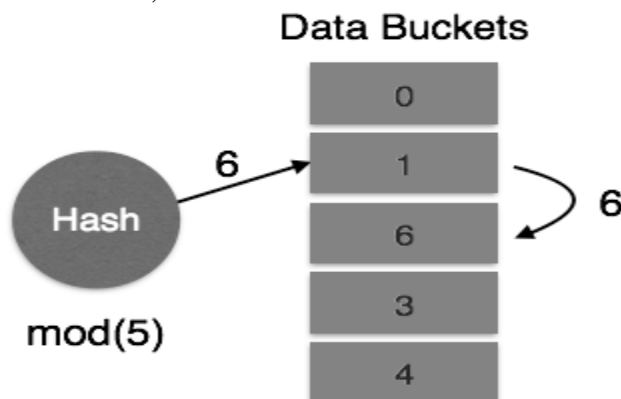
Bucket Overflow

The condition of bucket-overflow is known as **collision**. This is a fatal state for any static hash function. In this case, overflow chaining can be used.

- **Overflow Chaining** – When buckets are full, a new bucket is allocated for the same hash result and is linked after the previous one. This mechanism is called **Closed Hashing**.



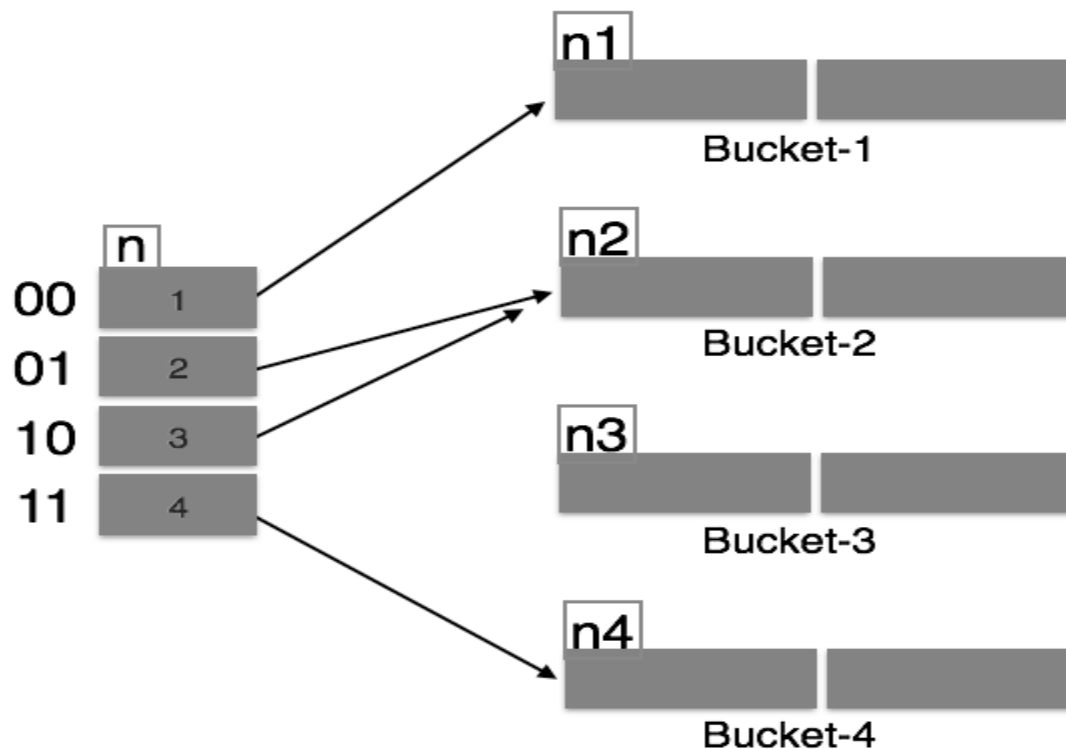
- **Linear Probing** – When a hash function generates an address at which data is already stored, the next free bucket is allocated to it. This mechanism is called **Open Hashing**.



Dynamic Hashing

The problem with static hashing is that it does not expand or shrink dynamically as the size of the database grows or shrinks. Dynamic hashing provides a mechanism in which data buckets are added and removed dynamically and on-demand. Dynamic hashing is also known as **extended hashing**.

Hash function, in dynamic hashing, is made to produce a large number of values and only a few are used initially.



Organization

The prefix of an entire hash value is taken as a hash index. Only a portion of the hash value is used for computing bucket addresses. Every hash index has a depth value to signify how many bits are used for computing a hash function. These bits can address 2^n buckets. When all these bits are consumed – that is, when all the buckets are full – then the depth value is increased linearly and twice the buckets are allocated.

Operation

- **Querying** – Look at the depth value of the hash index and use those bits to compute the bucket address.
- **Update** – Perform a query as above and update the data.
- **Deletion** – Perform a query to locate the desired data and delete the same.
- **Insertion** – Compute the address of the bucket
 - If the bucket is already full.
 - Add more buckets.
 - Add additional bits to the hash value.
 - Re-compute the hash function.
 - Else
 - Add data to the bucket,

- If all the buckets are full, perform the remedies of static hashing.

Hashing is not favorable when the data is organized in some ordering and the queries require a range of data. When data is discrete and random, hash performs the best.

Hashing algorithms have high complexity than indexing. All hash operations are done in constant time.

CHAPTER SIX

6. Query Languages

6.1. Introduction

We cover in this chapter the different kinds of queries normally posed to text retrieval systems. This is in part dependent on the retrieval model the system adopts, i.e., a full-text system will not answer the same kinds of queries as those answered by a system based on keyword ranking (as Web search engines) or on a hypertext model. In this chapter we show which queries can be formulated. The type of query the user might formulate is largely dependent on the underlying information retrieval model. An important issue is that most query languages try to use the content (i.e., the semantics) and the structure of the text (i.e., the text syntax) to find relevant documents. In that sense, the system may fail to find the relevant answers (see Chapter 3). For this reason, a number of techniques meant to enhance the usefulness of the queries exist. Examples include the expansion of a word to the set of its synonyms or the use of a thesaurus and stemming to 99 100 QUERY LANGUAGES put together all the derivatives of the same word.

Moreover, some words which are very frequent and do not carry meaning (such as ‘the’), called stopwords, and may be removed. Here we assume that all the query preprocessing has already been done. Although these operations are usually done for information retrieval, many of them can also be useful in a data retrieval context. When we want to emphasize the difference between words that can be retrieved by a query and those which cannot, we call the former ‘keywords.’ Orthogonal to the kind of queries that can be asked is the subject of the retrieval unit the information system adopts. The retrieval unit is the basic element which can be retrieved as an answer to a query (normally a set of such basic elements is retrieved, sometimes ranked by relevance or other criterion). The retrieval unit can be a file, a document, a Web page, a paragraph,

or some other structural unit which contains an answer to the search query. From this point on, we will simply call those retrieval units ‘documents,’ although as explained this can have different meanings.

This chapter is organized as follows. We first show the queries that can be formulated with keyword-based query languages. They are aimed at information retrieval, including simple words and phrases as well as Boolean operators which manipulate sets of documents. In the second section we cover pattern matching, which includes more complex queries and is generally aimed at complementing keyword searching with more powerful data retrieval capabilities. Third, we cover querying on the structure of the text, which is more dependent on the particular text model. Finally, we finish with some standard protocols used on the Internet and by CD-ROM publishers.

6.2. Query Languages

What Does Query Language Mean?

Query language (QL) refers to any computer programming language that requests and retrieves data from database and information systems by sending queries. It works on user entered structured and formal programming command based queries to find and extract data from host databases. Query language may also be termed database query language.

A query languages, also known as data query language or database query language (DQL), is a computer language used to make queries in databases and information systems. A well known example is the Structured Query Language (SQL).

Query language is primarily created for creating, accessing and modifying data in and out from a database management system (DBMS). Typically, QL requires users to input a structured command that is similar and close to the English language querying construct.

For example, the SQL query: `SELECT * FROM`

The customer will retrieve all data from the customer records/table.

The simple programming context makes it one of the easiest programming languages to learn. There are several different variants of QL and it has wide implementation in various database-centered services such as extracting data from deductive and OLAP databases, providing API based access to remote applications and services and more.

Broadly, query languages can be classified according to whether they are database query languages or information retrieval query languages. The difference is that a database query language attempts to give factual answers to factual questions, while an information retrieval query language attempts to find documents containing information that is relevant to an area of inquiry.

6.3 Relational Algebra

The relational algebra is a theoretical language with operations that work on one or more relations to define another relation without changing the original relation(s). Thus both the operands and the results are relations, and so the output from one operation can become the input to another operation. This ability allows expressions to be nested in the relational algebra, just as we can nest arithmetic operations. This property is called **closure**: relations are closed under the algebra, just as numbers are closed under arithmetic operations.

The relational algebra is a relation-at-a-time (or set) language in which all tuples, possibly from several relations, are manipulated in one statement without looping. There are many variations of the operations that are included in relational algebra. The five fundamental operations in relational algebra—Selection, Projection, Cartesian product, Union, and Set difference—perform most of the data retrieval operations that we are interested in. In addition, there are also the Join, Intersection, and Division operations, which can be expressed in terms of the five basic operations.

The Selection and Projection operations are unary operations, as they operate on one relation. The other operations work on pairs of relations and are therefore called binary operations. In the following definitions, let R and S be two relations defined over the attributes $A = (a_1, a_2, \dots, a_N)$ and $B = (b_1, b_2, \dots, b_M)$, respectively. Use the following figures for the examples in this chapter.

(a)

branchNo	street	city	postcode
B005	22 Deer Rd	London	SW1 4EH
B007	16 Argyll St	Aberdeen	AB2 3SU
B003	163 Main St	Glasgow	G11 9QX
B004	32 Manse Rd	Bristol	BS99 1NZ
B002	56 Clover Dr	London	NW10 6EU

(b)

clientNo	propertyNo	viewDate	comment
CR56	PA14	24-May-13	too small
CR76	PG4	20-Apr-13	too remote
CR56	PG4	26-May-13	
CR62	PA14	14-May-13	no dining room
CR56	PG36	28-Apr-13	

(c)

staffNo	fName	lName	position	sex	DOB	salary	branchNo
SL21	John	White	Manager	M	1-Oct-45	30000	B005
SG37	Ann	Beech	Assistant	F	10-Nov-60	12000	B003
SG14	David	Ford	Supervisor	M	24-Mar-58	18000	B003
SA9	Mary	Howe	Assistant	F	19-Feb-70	9000	B007
SG5	Susan	Brand	Manager	F	3-Jun-40	24000	B003
SL41	Julie	Lee	Assistant	F	13-Jun-65	9000	B005

(d)

propertyNo	street	city	postcode	type	rooms	rent	ownerNo	staffNo	branchNo
PA14	16 Holhead	Aberdeen	AB7 5SU	House	6	650	CO46	SA9	B007
PL94	6 Argyll St	London	NW2	Flat	4	400	CO87	SL41	B005
PG4	6 Lawrence St	Glasgow	G11 9QX	Flat	3	350	CO40		B003
PG36	2 Manor Rd	Glasgow	G32 4QX	Flat	3	375	CO93	SG37	B003
PG21	18 Dale Rd	Glasgow	G12	House	5	600	CO87	SG37	B003
PG16	5 Novar Dr	Glasgow	G12 9AX	Flat	4	450	CO93	SG14	B003

(e)

clientNo	fName	lName	telNo	prefType	maxRent	eMail
CR76	John	Kay	0207-774-5632	Flat	425	john.kay@gmail.com
CR56	Aline	Stewart	0141-848-1825	Flat	350	astewart@hotmail.com
CR74	Mike	Ritchie	01475-392178	House	750	mrRitchie01@yahoo.co.uk
CR62	Mary	Tregear	01224-196720	Flat	600	maryt@hotmail.co.uk

Figure 6.1. (a) Branch, (b) Viewing, (c) Staff, (d) PropertyForRent and (e) Client relations

6.1.1. Unary Operations

Selection (σ predicate (R))

The Selection operation works on a single relation R and defines a relation that contains only those tuples of R that satisfy the specified condition (predicate). Example 6.1: List all staff with a salary greater than 10000 birr.

$\sigma_{\text{salary} > 10000}(\text{Staff})$

Here, the input relation is Staff and the predicate is salary > 10000. The Selection operation defines a relation containing only those Staff tuples with a salary greater than 10000 birr. The

result of this operation is shown in Figure 6.2. More complex predicates can be generated using the logical operators (AND), (OR), and \sim (NOT).

staffNo	fName	IName	position	sex	DOB	salary	branchNo
SL21	John	White	Manager	M	1-Oct-45	30000	B005
SG37	Ann	Beech	Assistant	F	10-Nov-60	12000	B003
SG14	David	Ford	Supervisor	M	24-Mar-58	18000	B003
SG5	Susan	Brand	Manager	F	3-Jun-40	24000	B003

Figure 6.2. *Selecting salary > 10000 from the Staff relation*

Projection ($\pi_{a_1 \dots a_n}(R)$)

The Projection operation works on a single relation R and defines a relation that contains a vertical subset of R, extracting the values of specified attributes and eliminating duplicates.

Example 6.2: Produce a list of salaries for all staff, showing only the staffNo, fName, IName, and salary details.

$\pi_{\text{staffNo, fName, IName, salary}}(\text{Staff})$ In this example, the Projection operation defines a relation that contains only the designated Staff attributes staffNo, fName, IName, and salary, in the specified order. The result of this operation is shown in Figure 6.3

staffNo	fName	lName	salary
SL21	John	White	30000
SG37	Ann	Beech	12000
SG14	David	Ford	18000
SA9	Mary	Howe	9000
SG5	Susan	Brand	24000
SL41	Julie	Lee	9000

Figure 6.3. Projecting the Staff relation over the staffNo, fName, lName, and salary attributes.

6.1.2. Set Operations

The Selection and Projection operations extract information from only one relation. There are obviously cases where we would like to combine information from several relations. In the remainder of this section, we examine the binary operations of the relational algebra, starting with the set operations of Union, Set difference, Intersection, and Cartesian product.

Union (R U S)

The union of two relations R and S defines a relation that contains all the tuples of R, or S, or both R and S, duplicate tuples being eliminated. R and S must be union-compatible. If R and S have I and J tuples, respectively, their union is obtained by concatenating them into one relation with a maximum of (I + J) tuples. Union is possible only if the schemas of the two relations match, that is, if they have the same number of attributes with each pair of corresponding attributes having the same domain. In other words, the relations must be union-compatible. Note that attributes names are not used in defining union-compatibility. In some cases, the Projection operation may be used to make two relations union-compatible. Example 6.3: List all cities where there is either a branch office or a property for rent.

Pcity (Branch) U Pcity (PropertyForRent)

To produce union-compatible relations, we first use Projection operation to project the Branch and PropertyForRent relations over the attribute city, eliminating duplicates where necessary. We then use the Union operation to combine these new relations to produce the result shown in Figure 6.4.

city
London
Aberdeen
Glasgow
Bristol

Figure 6.4. Union based on the city attribute from the Branch and Prop

Set difference ($R - S$)

The Set difference operation defines a relation consisting of the tuples that are in relation R, but not in S. R and S must be union-compatible.

Example 6.4: List all cities where there is a branch office but no properties for rent. $\pi_{\text{city}} (\text{Branch}) - \pi_{\text{city}} (\text{PropertyForRent})$

As in the previous example, we produce union-compatible relations by projecting the Branch and PropertyForRent relations over the attribute city . We then use the Set difference operation to combine these new relations to produce the result shown in Figure 6.5.

city
Bristol

Intersection ($R \cap S$)

The Intersection operation defines a relation consisting of the set of all tuples that are in both R and S. R and S must be union-compatible.

Example 6.5: List all cities where there is both a branch office and at least one property for rent.

$$\pi_{\text{city}}(\text{Branch}) \cap \pi_{\text{city}}(\text{PropertyForRent})$$

As in the previous example, we produce union-compatible relations by projecting the Branch and PropertyForRent relations over the attribute city. We then use the Intersection operation to combine these new relations to produce the result shown in Figure 6.6.

Note that we can express the Intersection operation in terms of the Set difference operation: $R \cap S = R - (R - S)$

city
Aberdeen
London
Glasgow

Figure 6.6. Intersection based on city attribute from the Branch and PropertyForRent relations.

Cartesian product ($R \times S$)

The Cartesian product operation defines a relation that is the concatenation of every tuple of relation R with every tuple of relation S.

The Cartesian product operation multiplies two relations to define another relation consisting of all possible pairs of tuples from the two relations. Therefore, if one relation has I tuples and N attributes and the other has J tuples and M attributes, Cartesian product relation will contain ($I * J$) tuples with ($N + M$) attributes. It is possible that the two relations may have attributes with the same name. In this case, the attribute names are prefixed with the relation name to maintain the uniqueness of attribute names within a relation.

Example 6.6: List the names and comments of all clients who have viewed a property for rent.

The names of clients are held in the Client relation and the details of viewings are held in the Viewing relation. To obtain the list of clients and the comments on properties they have viewed, we need to combine these two relations:

$$(\pi \text{ clientNo, fName, IName } (\text{Client})) \times (\pi \text{ clientNo, propertyNo, comment } (\text{Viewing}))$$

The result of this operation is shown in Figure 6.7. In its present form, this relation contains more information than we require. For example, the first tuple of this relation contains different clientNo values. To obtain the required list, we need to carry out a Selection operation on this relation to extract those tuples where Client.clientNo = Viewing.clientNo.

The complete operation is thus:

$$\sigma \text{ Client.clientNo} = \text{Viewing.clientNo} ((\pi \text{ clientNo, fName, IName } (\text{Client})) \times (\pi \text{ clientNo, propertyNo, comment } (\text{Viewing})))$$

The result of this operation is shown in Figure 6.8

client.clientNo	fName	lName	Viewing.clientNo	propertyNo	comment
CR76	John	Kay	CR56	PA14	too small
CR76	John	Kay	CR76	PG4	too remote
CR76	John	Kay	CR56	PG4	
CR76	John	Kay	CR62	PA14	no dining room
CR76	John	Kay	CR56	PG36	
CR56	Aline	Stewart	CR56	PA14	too small
CR56	Aline	Stewart	CR76	PG4	too remote
CR56	Aline	Stewart	CR56	PG4	
CR56	Aline	Stewart	CR62	PA14	no dining room
CR56	Aline	Stewart	CR56	PG36	
CR74	Mike	Ritchie	CR56	PA14	too small
CR74	Mike	Ritchie	CR76	PG4	too remote
CR74	Mike	Ritchie	CR56	PG4	
CR74	Mike	Ritchie	CR62	PA14	no dining room
CR74	Mike	Ritchie	CR56	PG36	
CR62	Mary	Tregear	CR56	PA14	too small
CR62	Mary	Tregear	CR76	PG4	too remote
CR62	Mary	Tregear	CR56	PG4	
CR62	Mary	Tregear	CR62	PA14	no dining room
CR62	Mary	Tregear	CR56	PG36	

Figure 6.7. *Cartesian product of reduced Client and Viewing relations*

6.3.4. Aggregation and Grouping Operations

As well as simply retrieving certain tuples and attributes of one or more relations, we often want to perform some form of summation or aggregation of data, similar to the totals at the bottom of a report, or some form of grouping of data, similar to subtotals in a report. These operations cannot be performed using the basic relational algebra operations considered earlier. However, additional operations have been proposed, as we now discuss.

Aggregate operations ($\sigma_{AL}(R)$)

Applies the aggregate function list, AL, to the relation R to define a relation over the aggregate list.

AL contains one or more (<aggregate_function>, <attribute>) pairs.

The main aggregate functions are:

- COUNT – returns the number of values in the associated attribute.
- SUM – returns the sum of the values in the associated attribute.
- AVG – returns the average of the values in the associated attribute.
- MIN – returns the smallest value in the associated attribute.
- MAX – returns the largest value in the associated attribute.

Example 6.9: (a) How many properties cost more than £350 per month to rent? We can use the aggregate function COUNT to produce the relation R shown in Figure 6.10(a):

$\rho R (\text{myCount}) \text{COUNT propertyNo } (\sigma \text{ rent} > 350 (\text{PropertyForRent}))$

(b) Find the minimum, maximum, and average staff salary.

We can use the aggregate functions—MIN, MAX, and AVERAGE—to produce the relation R shown in Figure 5.10(b) as follows:

$\rho R (\text{myMin} , \text{myMax} , \text{myAverage}) \text{MIN salary, MAX salary, AVERAGE salary } (\text{Staff})$

myCount
5

(a)

myMin	myMax	myAverage
9000	30000	17000

(b)

Figure 6.10. *Result of the Aggregate operations: (a) finding the number of properties whose rent is greater than £350; (b) finding the minimum, maximum, and average staff salary.*

Grouping operation ($_{GA\ GL}(R)$)

Groups the tuples of relation R by the grouping attributes, GA , and then applies the aggregate function list AL to define a new relation. AL contains one or more ($\langle \text{aggregate_function} \rangle$, $\langle \text{attribute} \rangle$) pairs. The resulting relation contains the grouping attributes, GA , along with the results of each of the aggregate functions.

The general form of the grouping operation is as follows:

$$a_1, a_2, \dots, a_n \langle A_p a_p \rangle, \langle A_q a_q \rangle, \dots, \langle A_z a_z \rangle (R)$$

where R is any relation, a_1, a_2, \dots, a_n are attributes of R on which to group, a_p, a_q, \dots, a_z are other attributes of R , and A_p, A_q, \dots, A_z are aggregate functions.

The tuples of R are partitioned into groups such that:

- all tuples in a group have the same value for a_1, a_2, \dots, a_n ;
- tuples in different groups have different values for a_1, a_2, \dots, a_n .

We illustrate the use of the grouping operation with the following example.

Example 6.10: Find the number of staff working in each branch and the sum of their salaries.

We first need to group tuples according to the branch number, branchNo , and then use the aggregate functions COUNT and SUM to produce the required relation. The relational algebra expression is as follows:

$$\rho R (\text{branchNo}, \text{myCount}, \text{mySum}) \text{branchNo COUNT staffNo, SUM salary (Staff)}$$

branchNo	myCount	mySum
B003	3	54000
B005	2	39000
B007	1	9000

The resulting relation is shown in Figure 5.11.

Figure 6.11. *Result of the grouping operation to find the number of staff working in each branch and the sum of their salaries.*

6.4. Relational calculus

A certain order is always explicitly specified in a relational algebra expression and a strategy for evaluating the query is implied. In the relational calculus, there is no description of how to evaluate a query; a relational calculus query specifies **what** is to be retrieved rather than **how** to retrieve it.

The relational calculus is not related to differential and integral calculus in mathematics, but takes its name from a branch of symbolic logic called **predicate calculus**. When applied to databases, it is found in two forms: **tuple** relational calculus, as originally proposed by Codd, and **domain** relational calculus, as proposed by Lacroix and Pirotte.

In first-order logic or predicate calculus, a **predicate** is a truth-valued function with arguments. When we substitute values for the arguments, the function yields an expression, called a **proposition**, which can be either true or false. For example, the sentences, —John White is a member of staff, and —John White earns more than Ann Beech, are both propositions, because we can determine whether they are true or false. In the first case, we have a function, —is a member of staff, with one argument (John White); in the second case, we have a function, —earns more than, with two arguments (John White and Ann Beech).

If a predicate contains a variable, as in — x is a member of staff, there must be an associated range for x . When we substitute some values of this **range** for x , the proposition may be true; for other values, it may be false. For example, if the range is the set of all people and we replace x by John White, the proposition —John White is a member of staff is true. If we replace x by the name of a person who is not a member of staff, the proposition is false.

If P is a predicate, then we can write the set of all x such that P is true for x , as:

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

We may connect predicates by the logical connectives (AND), (OR), and \sim (NOT) to form compound predicates.

2.1. Tuple Relational Calculus

In the tuple relational calculus, we are interested in finding tuples for which a predicate is true. The calculus is based on the use of **tuple variables**. A tuple variable is a variable that —ranges over a named relation: that is, a variable whose only permitted values are tuples of the relation. (The word —range here does not correspond to the mathematical use of range, but corresponds to a mathematical domain.) For example, to specify the range of a tuple variable S as the Staff relation, we write:

Staff(S)

To express the query —Find the set of all tuples S such that $F(S)$ is true, we can write:

$$\{S \mid F(S)\}$$

F is called a **formula (well-formed formula, or wff** in mathematical logic). For example, to express the query —Find the staffNo , fName , IName , position , sex , DOB , salary , and branchNo of all staff earning more than £10,000, we can write:

$$\{S \mid \text{Staff}(S) \wedge S.\text{salary} > 10000\}$$

S.salary means the value of the salary attribute for the tuple variable S. To retrieve a particular attribute, such as salary, we would write:

$$\{ S.salary \mid Staff(S) \wedge S.salary > 10000 \}$$

The existential and universal quantifiers

There are two **quantifiers** we can use with formulae to tell how many instances the predicate applies to. The **existential quantifier** \exists (—there exists!) is used in formulae that must be true for at least one instance, such as:

$$Staff(S) \wedge \exists B (Branch(B) \wedge (B.branchNo = S.branchNo) \wedge B.city = \text{‘London’})$$

This means, —There exists a Branch tuple that has the same branchNo as the branchNo of the current Staff tuple, S, and is located in London. The universal quantifier \forall (—for all) is used in statements about every instance, such as:

$$\forall B (B.city \neq \text{‘Paris’})$$

This means, —For all Branch tuples, the address is not in Paris. We can apply a generalization of De Morgan’s laws to the existential and universal quantifiers. For example:

$$(\exists X)(F(X)) \sim (\forall X)(\sim(F(X)))$$

$$(\forall X)(F(X)) \sim (\exists X)(\sim(F(X)))$$

$$(\exists X)(F_1(X) \wedge F_2(X)) \sim (\forall X)(\sim(F_1(X)) \wedge \sim(F_2(X)))$$

$$(\forall X)(F_1(X) \wedge F_2(X)) \sim (\exists X)(\sim(F_1(X)) \wedge \sim(F_2(X)))$$

Using these equivalence rules, we can rewrite the previous formula as:

$$\sim (\exists B) (B.city = \text{‘Paris’})$$

which means, —There are no branches with an address in Paris.

Tuple variables that are qualified by \$ or " are called **bound variables**; the other tuple variables are called **free variables**. The only free variables in a relational calculus expression should be those on the left side of the bar (|). For example, in the following query:

$$\{S.fName, S.lName \mid Staff(S) (\$B) (Branch(B) (B.branchNo = S.branchNo) \cup B.city = \text{'London'})\}$$

S is the only free variable and S is then bound successively to each tuple of Staff .

6.5. Introduction to SQL

Objectives of SQL

Ideally, a database language should allow a user to:

- create the database and relation structures;
- perform basic data management tasks, such as the insertion, modification, and deletion of data from the relations;
- perform both simple and complex queries.

A database language must perform these tasks with minimal user effort, and its command structure and syntax must be relatively easy to learn. Finally, the language must be portable; that is, it must conform to some recognized standard so that we can use the same command structure and syntax when we move from one DBMS to another. SQL is intended to satisfy these requirements.

SQL is an example of a transform-oriented language, or a language designed to use relations to transform inputs into required outputs. As a language, the ISO SQL standard has two major components:

- Data Definition Language (DDL) for defining the database structure and controlling access to the data;
- Data Manipulation Language (DML) for retrieving and updating data.

Until the 1999 release of the standard, known as SQL:1999 or SQL3, SQL contained only these definitional and manipulative commands; it did not contain flow of control commands, such as

IF . . . THEN . . . ELSE, GO TO, or DO . . . WHILE. These commands had to be implemented using a programming or job-control language, or interactively by the decisions of the user.

Owing to this lack of computational completeness, SQL can be used in two ways. The first way is to use SQL interactively by entering the statements at a terminal. The second way is to embed SQL statements in a procedural language. We also SQL is a relatively easy language to learn:

- It is a nonprocedural language; you specify what information you require, rather than how to get it. In other words, SQL does not require you to specify the access methods to the data.
- Like most modern languages, SQL is essentially free-format, which means that parts of statements do not have to be typed at particular locations on the screen.
- The command structure consists of standard English words such as CREATE TABLE, INSERT, SELECT.

For example:

- CREATE TABLE Staff (staffNo VARCHAR(5), IName VARCHAR(15), salary

DECIMAL(7,2));

- INSERT INTO Staff VALUES (_SG16', _Brown', 8300);
- SELECT staffNo , IName , salary FROM Staff WHERE salary > 10000;
- SQL can be used by a range of users including database administrators (DBA), management personnel, application developers, and many other types of end-user. An international standard now exists for the SQL language making it both the formal and de facto standard language for defining and manipulating relational databases.

Importance of SQL

SQL is the first and, so far, only standard database language to gain wide acceptance. The only other standard database language, the Network Database Language (NDL), based on the CODASYL network model, has few followers. Nearly every major current vendor provides database products based on SQL or with an SQL interface, and most are represented on at least one of the standard-making bodies.

There is a huge investment in the SQL language both by vendors and by users. It has become part of application architectures such as IBM's Systems Application Architecture (SAA) and is the strategic choice of many large and influential organizations, for example, the Open Group consortium for UNIX standards. SQL has also become a Federal Information Processing Standard (FIPS) to which conformance is required for all sales of DBMSs to the U.S. government. The SQL Access Group, a consortium of vendors, defined a set of enhancements to SQL that would support interoperability across disparate systems.

SQL is used in other standards and even influences the development of other standards as a definitional tool. Examples include ISO's Information Resource Dictionary System (IRDS) standard and Remote Data Access (RDA) standard. The development of the language is supported by considerable academic interest, providing both a theoretical basis for the language and the techniques needed to implement it successfully. This is especially true in query optimization, distribution of data, and security. There are now specialized implementations of SQL that are directed at new markets, such as OnLine Analytical Processing (OLAP).

Terminology

The ISO SQL standard does not use the formal terms of relations, attributes, and tuples, instead using the terms tables, columns, and rows.

6.5.1. Writing SQL Commands

An SQL statement consists of **reserved words** and **user-defined** words. Reserved words are a fixed part of the SQL language and have a fixed meaning. They must be spelled exactly as required and cannot be split across lines. User-defined words are made up by the user (according to certain syntax rules) and represent the names of various database objects such as tables, columns, views, indexes, and so on. The words in a statement are also built according to a set of syntax rules. Although the standard does not require it, many dialects of SQL require the use of a statement terminator to mark the end of each SQL statement (usually the semicolon —; is used).

Most components of an SQL statement are **case-insensitive**, which means that letters can be typed in either upper- or lowercase. The one important exception to this rule is that literal

character data must be typed exactly as it appears in the database. For example, if we store a person's surname as —SMITH and then search for it using the string —Smith, the row will not be found.

Although SQL is free-format, an SQL statement or set of statements is more readable if indentation and lineation are used. For example:

- each clause in a statement should begin on a new line;
- the beginning of each clause should line up with the beginning of other clauses;
- if a clause has several parts, they should each appear on a separate line and be indented under the start of the clause to show the relationship.

Throughout this and the next three chapters, we use the following extended form of the Backus Naur Form (BNF) notation to define SQL statements:

- uppercase letters are used to represent reserved words and must be spelled exactly as shown;
- lowercase letters are used to represent user-defined words;
- a vertical bar (|) indicates a **choice** among alternatives; for example, a | b | c;
- curly braces indicate a **required element**; for example, {a};
- square brackets indicate an **optional element**; for example, [a];
- an ellipsis (. . .) is used to indicate **optional repetition** of an item zero or more times.

For example:

{a|b} (, c . . .) means either a or b followed by zero or more repetitions of c separated by commas.

In practice, the DDL statements are used to create the database structure (that is, the tables) and the access mechanisms (that is, what each user can legally access), and then the DML statements are used to populate and query the tables.

6.5.2. SQL Data Definition and Data Types

SQL uses the terms table, row, and column for the formal relational model terms relation, tuple, and attribute, respectively. We will use the corresponding terms interchangeably. The main SQL

command for data definition is the CREATE statement, which can be used to create schemas, tables (relations), and domains (as well as other constructs such as views, assertions, and triggers).

The Create Table Command in SQL

The CREATE TABLE command is used to specify a new relation by giving it a name and specifying its attributes and initial constraints. The attributes are specified first, and each attribute is given a name, a data type to specify its domain of values, and any attribute constraints, such as NOT NULL. The key, entity integrity, and referential integrity constraints can be specified within the CREATE TABLE statement after the attributes are declared, or they can be added later using the ALTER TABLE command.

Create Table Employee (Fname varchar (20), Lname varchar (20), ID int primary key, Bdate varchar (20), Address varchar (20), Sex char, Salary decimal, SuperID int foreign key references Employee (ID))

Create Table Department (DName Varchar (15), Dnumber Int primary key, MgrID int foreign key references employee (ID), Mgrstartdate varchar (20))

Create table Dept_Locations (Dnumber Int, Dlocation Varchar (15), **Primary Key** (Dnumber, Dlocation) , **Foreign Key**(Dnumber) **References** Department(Dnumber))

Create table project (Pname Varchar (15), Pnumber Int primary key, Plocation Varchar (15), Dnum Int foreign key references Department (Dnumber))

Create table works_On (EID int, Pno Int, Hours Decimal (3, 1), **Primary Key** (EID, Pno), **Foreign Key** (EID) **References** Employee (ID), **Foreign Key**(Pno) **References**, project(Pnumber))

Create table dependent (EID int, Dependent_Name Varchar (15), Sex Char, Bdate Date, Relationship Varchar (8), **Primary Key**(EID, Dependent_Name), **Foreign Key**(EID) **References** Employee(ID))

Inserting Values into the Table

In its simplest form, INSERT is used to add a single tuple to a relation. We must specify the relation name and a list of values for the tuple. The values should be listed in *the same order* in which the corresponding attributes were specified in the CREATE TABLE command. For example, to add a new tuple to the EMPLOYEE, Department, Project, Works_On, Dept_Locations, and Dependent are shown below

insert into employee values

('sara','girma',19,'12/2/2004','hossana','F','1000','19') **insert into** employee values

('selam','john',15,'12/2/1997','hossana','F','1000','19') **insert into** department values

('computer scien',19,19,'12/2/2004') **insert into** department values

('_Mathematics',11,19,'12/2/2004') **insert into** project values

('reaearch',1,'hossana',19) **insert into** project values ('reaearch',1,'hossana',19)

When you are inserting values into the table department the value that you insert to the MgrID should exist in the employee table ID because it references employee ID. Similar concept will be applied in the entire table that references another table.

Schema Change Statements in SQL

In this section, we give an overview of the schema evolution commands available in SQL, which can be used to alter a schema by adding or dropping tables, attributes, constraints, and other schema elements.

The DROP Command

The DROP command can be used to drop *named* schema elements, such as tables, domains, or constraints. One can also drop a schema. For example, if a whole schema is not needed any more, the DROP SCHEMA command can be used. For example, to remove the COMPANY database and all its tables, domains, and other elements, it is used as follows:

Drop Database Company

In order to drop the table employee from company database we use the following command

Drop table employee

The ALTER Command

The definition of a base table or of other named schema elements can be changed by using the ALTER command. For base tables, the possible alter table actions include adding or dropping a column (attribute), changing a column definition, and adding or dropping table constraints. For example, to add an attribute for keeping track of jobs of employees to the EMPLOYEE base relations in the COMPANY schema, we can use the command

Alter Table employee **Add** Job Varchar (12)

We must still enter a value for the new attribute JOB for each individual EMPLOYEE tuple. This can be done either by specifying a default clause or by using the UPDATE command. If no default clause is specified, the new attribute will have NULLs in all the tuples of the relation immediately after the command is executed; hence, the NOT NULL constraint is not allowed in this case.

To drop a column, we must choose either CASCADE or RESTRICT for drop behavior. If CASCADE is chosen, all constraints and views that reference the column are dropped automatically from the schema, along with the column. If RESTRICT is chosen, the command is successful only if no views or constraints (or other elements) reference the column. For example, the following command removes the attribute ADDRESS from the EMPLOYEE base table:

Alter Table Employee Drop column Address

It is also possible to alter a column definition by dropping an existing default clause or by defining a new default clause. The following examples illustrate this clause:

Alter Table Employee Alter Job Drop DEFAULT;

Alter Table Employee Alter Job Set Default “333445555”;

The DELETE Command

The DELETE command removes tuples from a relation. It includes a WHERE clause, similar to that used in an SQL query, to select the tuples to be deleted. Tuples are explicitly deleted from only one table at a time. However, the deletion may propagate to tuples in other relations if *referential triggered actions* are specified in the referential integrity constraints. Depending on the number of tuples selected by the condition in the WHERE clause, zero, one, or several tuples can be deleted by a single DELETE command. A missing WHERE clause specifies that all tuples in the relation are to be deleted; however the table remains in the database as an empty table. The DELETE commands in Q4A to Q4D

Q4A: DELETE FROM EMPLOYEE WHERE LNAME='Brown'

Q4B: DELETE FROM EMPLOYEE

WHERE ID='12'

Q4C: DELETE FROM EMPLOYEE

WHERE DNO IN (SELECT DNUMBER FROM DEPARTMENT

WHERE DNAME='Research')

Q4D: DELETE FROM EMPLOYEE

The UPDATE Command

The UPDATE command is used to modify attribute values of one or more selected tuples. As in the DELETE command, a WHERE clause in the UPDATE command selects the tuples to be modified from a single relation. However, updating a primary key value may propagate to the foreign key values of tuples in other relations if such a *referential triggered action* is specified in the referential integrity constraints. An additional SET clause in the UPDATE command specifies the attributes to be modified and their new values. For example, to change the location and controlling department number of project number 10 to 'Bellaire' and 5, respectively, we use U5:

U5: UPDATE PROJECT

SET PLOCATION = 'Bellaire', DNUM = 5

WHERE PNUMBER=10;

Several tuples can be modified with a single UPDATE command. An example is to give all employees in the 'Research' department a 10 percent raise in salary, as shown in U6. In this request, the modified SALARY value depends on the original SALARY value in each tuple, so two references to the SALARY attribute are needed. In the SET clause, the reference to the SALARY attribute on the right refers to the old SALARY value *before modification*, and the one on the left refers to the new SALARY value *after modification*:

U6: UPDATE EMPLOYEE

SET SALARY = SALARY * 1.1

WHERE DNO IN (SELECT DNUMBER

FROM DEPARTMENT

WHERE DNAME='Research');

It is also possible to specify NULL or DEFAULT as the new attribute value. Notice that each UPDATE command explicitly refers to a single relation only. To modify multiple relations, we must issue several UPDATE commands.

6.5.3. Basic Queries in SQL

SQL has one basic statement for retrieving information from a database: the SELECT statement. The SELECT statement has no relationship to the SELECT operation of relational algebra, which was discussed in Chapter 6. There are many options and flavors to the SELECT statement in SQL, so we will introduce its features gradually.

The SELECT-FROM-WHERE Structure of Basic SQL Queries

Queries in SQL can be very complex. We will start with simple queries, and then progress to more complex ones in a step-by-step manner. The basic form of the SELECT statement, sometimes called a mapping or a select-from-where block, is formed of the three clauses SELECT, FROM, and WHERE and has the following form:

SELECT<attribute list>

FROM<table list>

WHERE<condition>

Where <attribute list> is a list of attribute names whose values are to be retrieved by the query.

<Table list> is a list of the relation names required to process the query.

<Condition> is a conditional (Boolean) expression that identifies the tuples to be retrieved by the query.

In SQL, the basic logical comparison operators for comparing attribute values with one another and with literal constants are =, <, <=, >, >=, and <>. SQL has many additional comparison operators that we shall present gradually as needed.

QUERY 0: Retrieve the birth date and address of the employee whose name is sara girma'.

SELECT Bdate, Address

FROM EMPLOYEE

WHERE FNAME='sara' AND LNAME='girma'

This query involves only the EMPLOYEE relation listed in the FROM clause. The query *selects* the EMPLOYEE tuples that satisfy the condition of the WHERE clause, then *projects* the result on the BDATE and ADDRESS attributes listed in the SELECT clause.

Q0 is similar to the following relational algebra expression, except that duplicates, if any, would *not* be eliminated:

Π Bdate, Address (σ Fname=' sara' And Lname=' girma' (EMPLOYEE))

Hence, a simple SQL query with a single relation name in the FROM clause is similar to a SELECT-PROJECT pair of relational algebra operations. The SELECT clause of SQL specifies the *projection attributes*, and the WHERE clause specifies the *selection condition*. The only difference is that in the SQL query we may get duplicate tuples in the result, because the constraint that a relation is a set is not enforced.

QUERY1

Retrieve the name and address of all employees who work for the 'Research' department.

Q1:

SELECT Fname, Lname, Address

FROM EMPLOYEE, DEPARTMENT

WHERE DNAME='Research' **AND** DNUMBER=DNO

Q1 is similar to a SELECT-PROJECT-JOIN sequence of relational algebra operations. Such queries are often called select-project-join queries. In the WHERE clause of Q1, the condition DNAME = 'Research' is a selection condition and corresponds to a SELECT operation in the relational algebra. The condition DNUMBER = DNO is a join condition, which corresponds to a JOIN condition in the relational algebra.

In general, any number of select and join conditions may be specified in a single SQL query.

The next example is a select-project-join query with *two* join conditions.

QUERY2

For every project located in 'Stafford', list the project number, the controlling department number, and the department manager's last name, address, and birth date.

Q2:

```
SELECT Pnumber, Dnum, Lname, Address, Bdate

FROM Project, Department, Employee

WHERE Dnum=Dnumber and MgrID=ID and Plocation='nekemte'
```

The join condition DNUM = DNUMBER relates a project to its controlling department, whereas the join condition MgrID = ID relates the controlling department to the employee who manages that department.

Ambiguous Attribute Names, Aliasing, and Tuple Variables

In SQL the same name can be used for two (or more) attributes as long as the attributes are in *different relations*. If this is the case, and a query refers to two or more attributes with the same name, we must qualify the attribute name with the relation name to prevent ambiguity. This is done by *prefixing* the relation name to the attribute name and separating the two by a period. To illustrate this, suppose that DNO and LNAME attributes of the EMPLOYEE relation were called DNUMBER and NAME, and the DNAME attribute of DEPARTMENT was also called NAME; then, to prevent ambiguity, query Q1 would be rephrased as shown in Q1A. We must prefix the attributes NAME and DNUMBER in Q1A to specify which ones we are referring to, because the attribute names are used in both relations:

Q1A:

```
SELECT Fname, Employee.Name, Address

FROM Employee, Department

WHERE Department. Name='Research' AND Department. Dnumber
```

=Employee. Dnumber;

Ambiguity also arises in the case of queries that refer to the same relation twice, as in the following example.

QUERY 8

For each employee, retrieve the employee's first and last name and the first and last name of his or her immediate supervisor.

Q8:

SELECT E.Fname, E.Lname, S.Fname, S. Lname

FROM EMPLOYEE **AS** E, EMPLOYEE **AS** S

WHERE E.SUPERID=S.ID

In this case, we are allowed to declare alternative relation names E and S, called aliases or tuple variables, for the EMPLOYEE relation. An alias can follow the keyword AS, as shown in *Q8*, or it can directly follow the relation name—for example, by writing EMPLOYEE E, EMPLOYEE S in the FROM clause of *Q8*. It is also possible to rename the relation attributes within the query in SQL by giving them aliases. For example, if we write EMPLOYEE **AS** E (FN, MI, LN, ID, SD, ADDR, SEX, SAL, SID, DNO) in the FROM clause, FN becomes an alias for FNAME, MI for MINH, LN for LNAME, and so on.

In *Q8*, we can think of E and S as two *different copies* of the EMPLOYEE relation; the first, E, represents employees in the role of supervisees; the second, S, represents employees in the role of supervisors. We can now join the two copies. Of course, in reality there is *only* one EMPLOYEE relation, and the join condition is meant to join the relation with itself by matching the tuples that satisfy the join condition E. SUPERID = S. ID. Notice that this is an example of a one-level recursive query.

The result of query Q8 is shown in Figure 8.3d. Whenever one or more aliases are given to a relation, we can use these names to represent different references to that relation. This permits multiple references to the same relation within a query. Notice that, If we want to use this alias-naming mechanism in any SQL query to specify tuple variables for every table in the WHERE clause, whether or not the same relation needs to be referenced more than once. In fact, this practice is recommended since it results in queries that are easier to comprehend.

For example, we could specify query Q1A as in Q1B:

Q1B:

```
SELECT E.FNAME, E.NAME, E.ADDRESS  
  
FROM EMPLOYEE E, DEPARTMENT D  
  
WHERE D.NAME='Research' AND D.DNUMBER=E.DNUMBER
```

If we specify tuple variables for every table in the WHERE clause, a select-project-join query in SQL closely resembles the corresponding tuple relational calculus expression (except for duplicate elimination).

Unspecified WHERE Clause and Use of the Asterisk

We discuss two more features of SQL here. A missing WHERE clause indicates no condition on tuple selection; hence, all tuples of the relation specified in the FROM clause qualify and are selected for the query result. If more than one relation is specified in the FROM clause and there is no WHERE clause, then the CROSS PRODUCT-all possible tuple combinations of these relations is selected. For example, Query 9 selects all EMPLOYEE ID and Query 10 selects all combinations of an EMPLOYEE ID and a DEPARTMENT DNAME.

QUERIES 9 AND 10 : Select all EMPLOYEE ID (Q9), and all combinations of EMPLOYEE ID and DEPARTMENT DNAME (Q10) in the database.

Q9: **SELECT** SSN

FROM EMPLOYEE

Q10: **SELECT** SSN, DNAME

FROM EMPLOYEE, DEPARTMENT

It is extremely important to specify every selection and join condition in the **WHERE** clause; if any such condition is overlooked, incorrect and very large relations may result.

Notice that Q10 is similar to a **CROSS PRODUCT** operation followed by a **PROJECT** operation in relational algebra. If we specify all the attributes of **EMPLOYEE** and **DEPARTMENT** in Q10, we get the **CROSS PRODUCT** (except for duplicate elimination, if any).

To retrieve all the attribute values of the selected tuples, we do not have to list the attribute names explicitly in SQL; we just specify an *asterisk* (*), which stands for *all the attributes*. For example, query Q1C retrieves all the attribute values of any **EMPLOYEE** who works in **DEPARTMENT** number 5, query Q1D retrieves all the attributes of an **EMPLOYEE** and the attributes of the **DEPARTMENT** in which he or she works for every employee of the ‘Research’ department, and Q10A specifies the **CROSS PRODUCT** of the **EMPLOYEE** and **DEPARTMENT** relations.

Q1C:

Select *

From Employee

Where Dno=5 Q1D:

Select *

From Employee, Department

Where Dname=‘Research’ and Dno=Dnumber Q10A:

Select *

From Employee, Department

Tables as Sets in SQL

As we mentioned earlier, SQL usually treats a table not as a set but rather as a multiset; *duplicate tuples can appear more than once* in a table, and in the result of a query. SQL does not automatically eliminate duplicate tuples in the results of queries, for the following reasons:

- Duplicate elimination is an expensive operation. One way to implement it is to sort the tuples first and then eliminate duplicates.
- The user may want to see duplicate tuples in the result of a query.
- When an aggregate function is applied to tuples, in most cases we do not want to eliminate duplicates.

An SQL table with a key is restricted to being a set, since the key value must be distinct in each tuple. If we do *want* to eliminate duplicate tuples from the result of an SQL query, we use the keyword DISTINCT in the SELECT clause, meaning that only distinct tuples should remain in the result. In general, a query with SELECT DISTINCT eliminates duplicates, whereas a query with SELECT ALL does not. Specifying SELECT with neither

ALL nor DISTINCT-as in our previous examples-is equivalent to SELECT ALL. For example, Query 11 retrieves the salary of every employee; if several employees have the same salary, that salary value will appear as many times in the result of the query. If we are interested only in distinct salary values, we want each value to appear only once, regardless of how many employees earn that salary. By using the keyword DISTINCT as in *Q11A*. **QUERY 11**

Retrieve the salary of every employee (*Q11*) and all distinct salary values (*Q11A*).

Q11:

SELECT ALL SALARY

FROM EMPLOYEE

Q11A:

SELECT DISTINCT SALARY

FROM EMPLOYEE

SQL has directly incorporated some of the set operations of relational algebra. There are set union (UNION), set difference (EXCEPT), and set intersection (INTERSECT) operations. The relations resulting from these set operations are sets of tuples; that is, *duplicate tuples are eliminated from the result*. Because these set operations apply only to *union-compatible relations*, we must make sure that the two relations on which we apply the operation have the same attributes and that the attributes appear in the same order in both relations. The next example illustrates the use of UNION.

QUERY 4

Make a list of all project numbers for projects that involve an employee whose last name is 'girma', either as a worker or as a manager of the department that controls the project.

Q4:

(SELECT DISTINCT PNUMBER

FROM PROJECT, DEPARTMENT, EMPLOYEE

WHERE DNUM=DNUMBER AND MGRID=ID AND LNAME='girma')

UNION

(SELECT DISTINCT PNUMBER

FROM PROJECT, WORKS_ON, EMPLOYEE

WHERE PNUMBER=PNO AND EID=ID AND LNAME='girma');

The first SELECT query retrieves the projects that involve a 'girma' as manager of the department that controls the project, and the second retrieves the projects that involve a 'girma' as a worker on the project. Notice that if several employees have the last name 'girma', the project names involving any of them will be retrieved. Applying the UNION operation to the two SELECT queries gives the desired result. SQL also has corresponding multiset operations, which are followed by the keyword ALL (UNION ALL, EXCEPT ALL,

INTERSECT ALL). Their results are multisets (duplicates are not eliminated).

Substring Pattern Matching and Arithmetic Operators

In this section we discuss several more features of SQL. The first feature allows comparison conditions on only parts of a character string, using the LIKE comparison operator. This can be used for string pattern matching. Partial strings are specified using two reserved characters: % replaces an arbitrary number of zero or more characters, and the underscore (_) replaces a single character. For example, consider the following query.

QUERY 12 Retrieve all employees whose address is in Houston, Texas.

Q12:

SELECT FNAME, LNAME

FROM EMPLOYEE

WHERE ADDRESS **LIKE** '%Houston, TX%';

To retrieve all employees who were born during the 1950s, we can use Query 12A. Here, '5' must be the third character of the string (according to our format for date), so we use the value '_ _ 5 _ _ _', with each underscore serving as a placeholder for an arbitrary character.

QUERY 12A: Find all employees who were born during the 1950s.

Q12A:

SELECT FNAME, LNAME

FROM EMPLOYEE

WHERE BDATE **LIKE** __ _ 5 _ _ _ _ ‘

If an underscore or % is needed as a literal character in the string, the character should be preceded by an *escape character*, which is specified after the string using the keyword ESCAPE. For example, ‘AB_CD\%EF’ ESCAPE ‘\’ represents the literal string ‘AB_CD%EF’, because \ is specified as the escape character. Any character not used in the string can be chosen as the escape character. Also, we need a rule to specify apostrophes or single quotation marks (‘) if they are to be included in a string, because they are used to begin and end strings. If an apostrophe (‘) is needed, it is represented as two consecutive apostrophes (‘‘) so that it will not be interpreted as ending the string.

Another feature allows the use of arithmetic in queries. The standard arithmetic operators for addition (+), subtraction (-), multiplication (*), and division (/) can be applied to numeric values or attributes with numeric domains. For example, suppose that we want to see the effect of giving all employees who work on the ‘ProductX’ project a 10 percent raise; we can issue Query13 to see what their salaries would become. This example also shows how we can rename an attribute in the query result using AS in the SELECT clause.

QUERY 13 Show the resulting salaries if every employee working on the ‘ProductX’ project is given a 10 percent raise.

Q13:

SELECT FNAME, LNAME, 1.1*SALARY **AS** INCREASED_SAL

FROM EMPLOYEE, WORKS_ON, PROJECT

WHERE ID=EID **AND** PNO=PNUMBER **AND** PNAME=‘ProductX’

For string data types, the concatenate operator || can be used in a query to append two string values. For date, time, timestamp, and interval data types, operators include incrementing (+) or decrementing (-) a date, time, or timestamp by an interval. In addition, an interval value is the result of the difference between two date, time, or timestamp values. Another comparison operator that can be used for convenience is BETWEEN, which is illustrated in Query 14. QUERY 14 Retrieve all employees in department 5 whose salary is between \$30,000 and \$40,000.

Q14: **SELECT ***

FROM EMPLOYEE

WHERE (SALARY BETWEEN 30000 AND 40000) AND DNO =5;

The condition (SALARY BETWEEN 30000 AND 40000) in Q14 is equivalent to the condition ((SALARY >= 30000) AND (SALARY <= 40000)).

Ordering of Query Results

SQL allows the user to order the tuples in the result of a query by the values of one or more attributes, using the ORDER BY clause. This is illustrated by Query 15.

QUERY 15 Retrieve a list of employees and the projects they are working on, ordered by department and, within each department, ordered alphabetically by last name, first name.

Q15: **SELECT DNAME, LNAME, FNAME, PNAME**

FROM DEPARTMENT, EMPLOYEE, WORKS_ON, PROJECT

WHERE DNUMBER=DNO AND SSN=ESSN AND PNO=PNUMBER

ORDER BY DNAME, LNAME, FNAME

The default order is in ascending order of values. We can specify the keyword DESC if we want to see the result in a descending order of values. The keyword ASC can be used to specify

ascending order explicitly. For example, if we want descending order on DNAME and ascending order on LNAME, FNAME, the ORDER BY clause of Q15 can be written as

ORDER BY DNAME DESC, LNAME ASC, FNAME ASC

More Complex SQL Queries

In the previous section, we described some basic types of queries in SQL. Because of the generality and expressive power of the language, there are many additional features that allow users to specify more complex queries. We discuss several of these features in this section.

Nested Queries, Tuples, and Set/Multiset Comparisons

Some queries require that existing values in the database be fetched and then used in a comparison condition. Such queries can be conveniently formulated by using nested queries, which are complete select-from-where blocks within the WHERE clause of another query. That other query is called the outer query. Query 4 is formulated in Q4 without a nested query, but it can be rephrased to use nested queries as shown in Q4A. Q4A introduces the comparison operator IN, which compares a value v with a set (or multiset) of values V and evaluates to TRUE if v is one of the elements in V

```
Q4A:                SELECT DISTINCT PNUMBER

FROM PROJECT

WHERE PNUMBER IN (SELECT PNUMBER

FROM PROJECT, DEPARTMENT,

EMPLOYEE

WHERE DNUM=DNUMBER AND MGRID=ID AND

LNAME='Girma')
```


OR

PNUMBER IN (SELECT PNO

FROM WORKS_ON, EMPLOYEE

WHERE EID=ID AND LNAME='Girma')

The first nested query selects the project numbers of projects that have a 'GIRMA' involved as manager, while the second selects the project numbers of projects that have a 'GIRMA' involved as worker. **In** the outer query, we use the **OR** logical connective to retrieve a PROJECT tuple if the PNUMBER value of that tuple is in the result of either nested query. If a nested query returns a single attribute *and* a single tuple, the query result will be a single (scalar) value. **In** such cases, it is permissible to use = instead of IN for the comparison operator. **In** general, the nested query will return a **table** (relation), which is a set or multiset of tuples.

SQL allows the use of **tuples** of values in comparisons by placing them within parentheses.

To illustrate this, consider the following query:

SELECT DISTINCT EID

FROM WORKS_ON

WHERE (PNO, HOURS) IN (SELECT PNO, HOURS FROM WORKS_ON

WHERE ID='12')

This query will select the Identity numbers of all employees who work the same (project, hours) combination on some project that employee 'John Smith' (whose ID = '12') works on. **In** this example, the IN operator compares the sub tuple of values in parentheses (PNO, HOURS) for each tuple in WORKS_ON with the set of union-compatible tuples produced by the nested query.

In addition to the IN operator, a number of other comparison operators can be used to compare a single value v (typically an attribute name) to a set or multiset V (typically a nested query). The $= ANY$ (or $= SOME$) operator returns TRUE if the value v is equal to some *value* in the set V and is hence equivalent to IN. The keywords ANY and SOME have the same meaning. Other operators that can be combined with ANY (or SOME) include $>$, $>=$, $<$, $<=$, and $< >$. The keyword ALL can also be combined with each of these operators.

For example, the comparison condition ($v > ALL V$) returns TRUE if the value v is greater than *all* the values in the set (or multiset) V . An example is the following query, which returns the names of employees whose salary is greater than the salary of all the employees in department 5:

```
SELECT LNAME, FNAME
FROM EMPLOYEE
WHERE SALARY > ALL (SELECT SALARY
FROM EMPLOYEE
WHERE DNO=5)
```

In general, we can have several levels of nested queries. We can once again be faced with possible ambiguity among attribute names if attributes of the same name exist—one in a relation in the FROM clause of the *outer query*, and another in a relation in the FROM clause of the *nested query*. The rule is that a reference to an *unqualified attribute* refers to the relation declared in the innermost nested query. For example, in the SELECT clause and WHERE clause of the first nested query of Q4A, a reference to any unqualified attribute of the PROJECT relation refers to the PROJECT relation specified in the FROM clause of the nested query. To refer to an attribute of the PROJECT relation specified in the outer query, we can specify and refer to an *alias* (tuple variable) for that relation. These rules are similar to scope rules for program variables in most programming languages that allow nested procedures and functions. To illustrate the potential ambiguity of attribute names in nested queries, consider Query 16,

QUERY 16 Retrieve the name of each employee who has a dependent with the same first name and same sex as the employee.

```
Q16: SELECT E.FNAME, E.LNAME  
  
FROM EMPLOYEE AS E  
  
WHERE E.ID IN (SELECT EID  
  
FROM DEPENDENT  
  
WHERE E.FNAME=DEPENDENT_NAME  
  
AND E.SEX=SEX)
```

In the nested query of *Q16*, we must qualify E. SEX because it refers to the SEX attribute of EMPLOYEE from the outer query, and DEPENDENT also has an attribute called SEX. All unqualified references to SEX in the nested query refer to SEX of DEPENDENT. However, we do not *have* to qualify FNAME and ID because the DEPENDENT relation does not have attributes called FNAME and ID, so there is no ambiguity.

It is generally advisable to create tuple variables (aliases) for *all the tables referenced in an SQL query* to avoid potential errors and ambiguities.

Correlated Nested Queries

Whenever a condition in the WHERE clause of a nested query references some attribute of a relation declared in the outer query, the two queries are said to be correlated. We can understand a correlated query better by considering that the *nested query is evaluated once for each tuple (or combination of tuples) in the outer query*. For example, we can think of *Q16* as follows: For *each* EMPLOYEE tuple, evaluate the nested query, which retrieves the ESSN values for all DEPENDENT tuples with the same sex and name as that EMPLOYEE tuple; if the SSN value of the EMPLOYEE tuple is *in* the result of the nested query, then select that EMPLOYEE tuple.

In general, a query written with nested select-from-where blocks and using the = or IN comparison operators can *always* be expressed as a single block query. For example, *Q16* may be written as in Q16A:

```
Q16A:                SELECT E.FNAME, E.LNAME

FROM EMPLOYEE AS E, DEPENDENT AS D WHERE E.ID=D.EID AND E.SEX=D.SEX
AND

E.FNAME=D.DEPENDENT_NAME
```

The original SQL implementation on SYSTEM R also had a CONTAINS comparison operator, which was used to compare two sets or multisets. This operator was subsequently dropped from the language, possibly because of the difficulty of implementing it efficiently. Most commercial implementations of SQL do *not* have this operator. The CONTAINS operator compares two sets of values and returns TRUE if one set contains all values in the other set. Query 3 illustrates the use of the CONTAINS operator.

QUERY 3 Retrieve the name of each employee who works on *all* the projects controlled by department number 5.

```
Q3: SELECT FNAME, LNAME

FROM EMPLOYEE

WHERE ((SELECT PNO

FROM WORKS_ON

WHERE ID=EID)

CONTAINS

(SELECT PNUMBER
```

FROM PROJECT

WHERE DNUM=5))

In Q3, the second nested query (which is not correlated with the outer query) retrieves the project numbers of all projects controlled by department 5. For *each* employee tuple, the first nested query (which is correlated) retrieves the project numbers on which the employee works; if these contain all projects controlled by department 5, the employee tuple is selected and the name of that employee is retrieved. Notice that the CONTAINS comparison operator has a similar function to the DIVISION operation of the relational algebra.

Because the CONTAINS operation is not part of SQL, we have to use other techniques, such as the EXISTS function, to specify these types of queries

The EXISTS and UNIQUE Functions in SQL

The EXISTS function in SQL is used to check whether the result of a correlated nested query is empty (contains no tuples) or not. We illustrate the use of EXISTS-and NOT EXISTS-with some examples. First, we formulate Query 16 in an alternative form that a use EXISTS.

Q16B: SELECT E.FNAME, E.LNAME

FROM EMPLOYEE AS E

WHERE

EXISTS (SELECT *

FROM DEPENDENT

WHERE E.ID=EID AND E.SEX=SEX

AND E.FNAME=DEPENDENT_NAME)

EXISTS and NOTEXISTS are usually used in conjunction with a correlated nested query. In Q16B, the nested query references the ID, FNAME, and SEX attributes of the EMPLOYEE relation from the outer query. We can think of Q16B as follows: For each EMPLOYEE tuple, evaluate the nested query, which retrieves all DEPENDENT tuples with the same Identity number, sex, and name as the EMPLOYEE tuple; if at least one tuple EXISTS in the result of the nested query, then select that EMPLOYEE tuple. In general, EXISTS (Q) returns TRUE if there is *at least one tuple* in the result of the nested query Q, and it returns FALSE otherwise. On the other hand, NOTEXISTS (Q) returns TRUE if there are no *tuples* in the result of nested query Q, and it returns FALSE otherwise. Next, we illustrate the use of NOTEXISTS.

QUERY 6 Retrieve the names of employees who have no dependents.

Q6: **SELECT** FNAME, LNAME

FROM EMPLOYEE

WHERE NOT EXISTS (SELECT *

FROM DEPENDENT

WHERE ID=EID)

In *Q6*, the correlated nested query retrieves all DEPENDENT tuples related to a particular EMPLOYEE tuple. If *none exist*, the EMPLOYEE tuple is selected. We can explain Q6 as follows:

For *each* EMPLOYEE tuple, the correlated nested query selects all DEPENDENT tuples whose EID value matches the EMPLOYEE ID; if the result is empty; no dependents are related to the employee, so we select that EMPLOYEE tuple and retrieve its FNAME and

LNAME.

QUERY 7 List the names of managers who have at least one dependent.

Q7: **SELECT** FNAME, LNAME

FROM EMPLOYEE WHERE

EXISTS (SELECT *

FROM DEPENDENT WHERE ID=EID) AND

EXISTS

(SELECT *

FROM DEPARTMENT

WHERE ID=MGRID)

One way to write this query is shown in *Q7*, where we specify two nested correlated queries; the first selects all DEPENDENT tuples related to an EMPLOYEE, and the second selects all DEPARTMENT tuples managed by the EMPLOYEE. If at least one of the first and at least one of the second exists, we select the EMPLOYEE tuple. Can you rewrite this query using only a single nested query or no nested queries?

Query 3 (“Retrieve the name of each employee who works on *all* the projects controlled by department number 5,”) can be stated using EXISTS and NOTEXISTS in SQL systems. There are two options. The first is to use the well-known set theory transformation that (S1 CONTAINS S2) is logically equivalent to (S2 EXCEPT S1) is empts,” This option is shown as *Q3A*.

Q3A: SELECT FNAME, LNAME

FROM EMPLOYEE

WHERE NOT EXISTS ((SELECT PNUMBER

FROM PROJECT WHERE DNUM=5) EXCEPT

(SELECT PNO

FROM WORKS_ON

WHERE ID=EID))

In Q3A, the first sub query (which is not correlated) selects all projects controlled by department 5, and the second sub query (which is correlated) selects all projects that the particular employee being considered works on. If the set difference of the first sub query MINUS (EXCEPT) the second sub query is empty, it means that the employee works on all the projects and is hence selected. The second option is shown as Q3B. Notice that we need two-level nesting in Q3B and that this formulation is quite a bit more complex than Q3, which used the CONTAINS comparison operator, and Q3A, which uses NOT EXISTS and EXCEPT. However, CONTAINS is not part of SQL, and not all relational systems have the EXCEPT operator even though it is part of SQL-99.

Q3B: SELECT LNAME, FNAME

FROM EMPLOYEE

WHERE NOT EXISTS

(SELECT *

FROM WORKS_ON B

WHERE (B.PNO IN (SELECT PNUMBER

FROM PROJECT WHERE DNUM=5))

AND

NOT EXISTS (SELECT *

FROM WORKS_ON C

WHERE C.EID=ID

AND C.PNO=B.PNO))

In Q3B, the outer nested query selects any WORKS_ON (B) tuples whose PNO is of a project controlled by department 5, *if* there is not a WORKS_ON (C) tuple with the same PNO and the same SSN as that of the EMPLOYEE tuple under consideration in the outer query. If no such tuple exists, we select the EMPLOYEE tuple. The form of Q3B matches the following rephrasing of Query 3: Select each employee such that there does not exist a project controlled by department 5 that the employee does not work on. There is another SQL function, UNIQUE (Q), which returns TRUE if there are no duplicate tuples in the result of query Q; otherwise, it returns FALSE. This can be used to test whether the result of a nested query is a set or a multiset.

Explicit Sets and Renaming of Attributes in SQL

We have seen several queries with a nested query in the WHERE clause. It is also possible to use an explicit set of values in the WHERE clause, rather than a nested query. Such a set is enclosed in parentheses in SQL.

QUERY 17 Retrieve the IDENTITY numbers of all employees who work on project numbers

1, 2, or 3.

Q17: SELECT DISTINCT ESSN

FROM WORKS_ON

WHERE PNO IN (1, 2, 3)

In SQL, it is possible to rename any attribute that appears in the result of a query by adding the qualifier AS followed by the desired new name. Hence, the AS construct can be used to alias both attribute and relation names, and it can be used in both the SELECT and FROM clauses. For example, Q8A shows how query Q8 can be slightly changed to retrieve the last name of each employee and his or her supervisor, while renaming the resulting attribute names as EMPLOYEE_NAME and SUPERVISOR_NAME. The new names will appear as column headers in the query result.

Q8A: SELECT E.LNAME AS EMPLOYEE_NAME, S.LNAME AS

SUPERVISOR_NAME

FROM EMPLOYEE AS E, EMPLOYEE AS S

WHERE E.SUPERID=S.ID

Joined Tables in SQL

The concept of a joined table (or joined relation) was incorporated into SQL to permit users to specify a table resulting from a join operation in *the FROM clause* of a query. This construct may be easier to comprehend than mixing together all the select and join conditions in the WHERE clause. For example, consider query *Q1*, which retrieves the name and address of every employee who works for the ‘Research’ department. It may be easier first to specify the join of the EMPLOYEE and DEPARTMENT relations, and then to select the desired tuples and attributes. This can be written in SQL as in Q1A:

Q1A: SELECT FNAME, LNAME, ADDRESS

FROM (EMPLOYEE JOIN DEPARTMENT ON DNO=DNUMBER)

WHERE DNAME=‘Research’

The FROM clause in Q 1A contains a single *joined table*. The attributes of such a table are all the attributes of the first table, EMPLOYEE, followed by all the attributes of the second table, DEPARTMENT. The concept of a joined table also allows the user to specify different types of join, such as NATURAL JOIN and various types of OUTER JOIN. In a NATURAL JOIN on two relations R and S, no join condition is specified; an implicit equijoin condition for *each pair of attributes with the same name* from R and S is created. If the names of the join attributes are not the same in the base relations, it is possible to rename the attributes so that they match, and then to apply NATURAL JOIN. In this case, the AS construct can be used to rename a relation and all its attributes in the FROM clause. This is illustrated in Q1B, where the DEPARTMENT relation is renamed as DEPT and its attributes are renamed as DNAME, DNO (to match the

name of the desired join attribute DNO in EMPLOYEE), MID, and MSDATE. The implied join condition for this NATURAL JOIN is EMPLOYEE. DNO = DEPT. DNO, because this is the only pair of attributes with the same name after renaming.

Q1B: SELECT FNAME, LNAME, ADDRESS

FROM (EMPLOYEE NATURAL JOIN (DEPARTMENT AS DEPT (DNAME, DNO, MID, MSDATE)))

WHERE DNAME='Research';

The default type of join in a joined table is an inner join, where a tuple is included in the result only if a matching tuple exists in the other relation. For example, in queryQ8A, only employees that *have a supervisor* are included in the result; an EMPLOYEE tuple whose value for SUPERID is NULL is excluded. If the user requires that all employees be included, an OUTER JOIN must be used explicitly. In SQL, this is handled by explicitly specifying the OUTER JOIN in a joined table, as illustrated in Q8B:

Q8B: SELECT E.LNAME AS EMPLOYEE_NAME, S.LNAME AS

SUPERVISOR_NAME

FROM (EMPLOYEE AS E LEFT OUTER JOIN EMPLOYEE AS S ON

E.SUPERID=S.ID)

The options available for specifying joined tables in SQL include INNER JOIN (same as JOIN), LEFT OUTER JOIN, RIGHT OUTER JOIN, and FULL OUTER JOIN. In the latter three options, the keyword OUTER may be omitted. If the join attributes have the same name, one may also specify the natural join variation of outer joins by using the keyword NATURAL before the operation (for example, NATURAL LEFT OUTER JOIN). The

keyword CROSS JOIN is used to specify the Cartesian product operation

It is also possible to *nest* join specifications; that is, one of the tables in a join may itself be a joined table. This is illustrated by Q2A, which is a different way of specifying query *Q2*, using the concept of a joined table:

```
Q2A: SELECT PNUMBER, DNUM, LNAME, ADDRESS, BDATE  
  
FROM ((PROJECT JOIN DEPARTMENT ON DNUM=DNUMBER)  
  
JOIN EMPLOYEE ON MGRID=ID)  
  
WHERE PLOCATION='Stafford';
```

Aggregate Functions in SQL

In chapter six, we introduced the concept of an aggregate function as a relational operation.

Because grouping and aggregation are required in many database applications, SQL has features that incorporate these concepts. A number of built-in functions exist: COUNT, SUM, MAX, MIN, and AVG. The COUNT function returns the number of tuples or values as specified in a query. The functions SUM, MAX, MIN, and AVG are applied to a set or multiset of numeric values and return, respectively, the sum, maximum value, minimum value, and average (mean) of those values. These functions can be used in the SELECT clause or in a HAVING clause (which we introduce later). The functions MAX and MIN can also be used with attributes that have nonnumeric domains if the domain values have a *total ordering* among one another. We illustrate the use of these functions with example queries.

QUERY 19 Find the sum of the salaries of all employees, the maximum salary, the minimum salary, and the average salary.

```
Q19: SELECT SUM (SALARY), MAX (SALARY), MIN (SALARY),  
  
AVG (SALARY)  
  
FROM EMPLOYEE
```

If we want to get the preceding function values for employees of a specific department-say, the 'Research' department-we can write Query 20, where the EMPLOYEE tuples are restricted by the WHERE clause to those employees who work for the 'Research' department.

QUERY 20 Find the sum of the salaries of all employees of the 'Research' department, as well as the maximum salary, the minimum salary, and the average salary in this department.

Q20: **SELECT SUM (SALARY), MAX (SALARY), MIN (SALARY),**

AVG (SALARY)

FROM (EMPLOYEE JOIN DEPARTMENT ON DNO=DNUMBER)

WHERE DNAME='Research'

QUERIES 21 AND 22 Retrieve the total number of employees in the company (*Q21*) and the number of employees in the 'Research' department (*Q22*).

Q21: **SELECT COUNT (*)**

FROM EMPLOYEE;

Q22: **SELECT COUNT (*)**

FROM EMPLOYEE, DEPARTMENT

WHERE DNO=DNUMBER AND DNAME='Research'

Here the asterisk (*) refers to the *rows* (tuples), so COUNT (*) returns the number of rows in the result of the query. We may also use the COUNT function to count values in a column rather than tuples, as in the next example.

QUERY 23 Count the number of distinct salary values in the database.

Q23: **SELECT COUNT (DISTINCT SALARY)**

FROM EMPLOYEE

If we write COUNT (Salary) instead of COUNT (DISTINCT SALARY) in Q23, then duplicate values will not be eliminated. However, any tuples with NULL for SALARY will not be counted. In general, NULL values are discarded when aggregate functions are applied to a particular column (attribute).

The preceding examples summarize *a whole relation* (Q19, Q21, Q23) or a selected subset of tuples (Q20, Q22), and hence all produce single tuples or single values. They illustrate how functions are applied to retrieve a summary value or summary tuple from the database. These functions can also be used in selection conditions involving nested queries. We can specify a correlated nested query with an aggregate function, and then use the nested query in the WHERE clause of an outer query. For example, to retrieve the names of all employees who have two or more dependents (Query 5), we can write the following:

Q5: SELECT LNAME, FNAME

FROM EMPLOYEE

WHERE

(SELECT COUNT (*) FROM DEPENDENT

WHERE ID=EID) >= 2'

The correlated nested query counts the number of dependents that each employee has; if this is greater than or equal to two, the employee tuple is selected.

Grouping: The GROUP BY and HAVING Clauses

In many cases we want to apply the aggregate functions to *subgroups of tuples* in a *relation*, where the subgroups are based on some attribute values. For example, we may want to find the average salary of employees in *each department* or the number of employees who work on *each project*. In these cases we need to partition the relation into non overlapping subsets (or

groups) of tuples. Each group (partition) will consist of the tuples that have the same value of some attributes), called the grouping attributes). We can then apply the function to each such group independently. SQL has a GROUP BY clause for this purpose.

The GROUP BY clause specifies the grouping attributes, which should *also appear in the SELECT clause*, so that the value resulting from applying each aggregate function to a group of tuples appears along with the value of the grouping attributes).

QUERY 24 For each department, retrieve the department number, the number of employees in the department, and their average salary.

Q24: SELECT DNO, COUNT (*), AVG (SALARY)

FROM EMPLOYEE

GROUP BY DNO;

In Q24, the EMPLOYEE tuples are partitioned into groups-each group having the same value for the grouping attribute DNO. The COUNT and AVG functions are applied to each such group of tuples. Notice that the SELECT clause includes only the grouping attribute and the functions to be applied on each group of tuples. If NULLs exist in the grouping attribute, then a separate group is created for all tuples with a *NULL value in the grouping attribute*. For example, if the EMPLOYEE table had some tuples that had NULL for the grouping attribute DNO, there would be a separate group for those tuples in the result of *Q24*.

QUERY 25 For each project, retrieve the project number, the project name, and the number of employees who work on that project.

Q25: **SELECT** PNUMBER, PNAME, COUNT (*)

FROM PROJECT, WORKS_ON

WHERE PNUMBER=PNO

GROUP BY PNUMBER, PNAME

Q25 shows how we can use a join condition in conjunction with GROUP BY. In this case, the grouping and functions are applied *after* the joining of the two relations. Sometimes we want to retrieve the values of these functions only for *groups that satisfy certain conditions*. For example, suppose that we want to modify Query 25 so that only projects with more than two employees appear in the result. SQL provides a HAVING clause, which can appear in conjunction with a GROUP BY clause, for this purpose. HAVING provides a condition on the group of tuples associated with each value of the grouping attributes. Only the groups that satisfy the condition are retrieved in the result of the query. This is illustrated by Query 26.

QUERY26 For each project on *which more than two employees work*, retrieve the project number, the project name, and the number of employees who work on the project

Q26: **SELECT** PNUMBER, PNAME, **COUNT** (*)

FROM PROJECT, WORKS_ON

WHERE PNUMBER=PNO

GROUP BY PNUMBER, PNAME

HAVING **COUNT** (*) > 2

Notice that, while selection conditions in the WHERE clause limit the *tuples* to which functions are applied, the HAVING clause serves to choose *whole groups*.

QUERY27 For each project, retrieve the project number, the project name, and the number of employees from department 5 who work on the project.

Q27: **SELECT** PNUMBER, PNAME, **COUNT** (*)

FROM PROJECT, WORKS_ON, EMPLOYEE

WHERE PNUMBER=PNO **AND** ID=EID **AND** DNO=5

GROUP BY PNUMBER, PNAME

Here we restrict the tuples in the relation (and hence the tuples in each group) to those that satisfy the condition specified in the WHERE clause-namely, that they work in department number 5. Notice that we must be extra careful when two different conditions apply (one to the function in the SELECT clause and another to the function in the HAVING clause). For example, suppose that we want to count the *total* number of employees whose salaries exceed \$40,000 in each department, but only for departments where more than five employees work. Here, the condition (SALARY> 40000) applies only to the COUNT function In the SELECT clause. Suppose that we write the following incorrect query:

```
SELECT DNAME, COUNT (*)
```

```
FROM DEPARTMENT, EMPLOYEE
```

```
WHERE DNUMBER=DNO AND SALARY>40000
```

```
GROUP BY DNAME
```

```
HAVING COUNT (*) > 5
```

This is incorrect because it will select only departments that have more than five employees *who each earn more than \$40,000*. The rule is that the WHERE clause is executed first, to select individual tuples; the HAVING clause is applied later, to select individual groups of tuples. Hence, the tuples are already restricted to employees who earn more than \$40,000, *before* the function in the HAVING clause is applied. One way to write this query correctly is to use a nested query, as shown in Query 28.

QUERY28 For each department that has more than five employees, retrieve the department number and the number of its employees who are making more than \$40,000.

```
Q28: SELECT DNUMBER, COUNT (*)
```

```
FROM DEPARTMENT, EMPLOYEE
```

```
WHERE DNUMBER=DNO AND SALARY>40000 AND
```

```
DNO IN (SELECT DNO
FROM EMPLOYEE
GROUP BY DNO
HAVING COUNT (*) > 5)
GROUP BY DNUMBER
```

Reference

<https://www.techtarget.com/searchdatamanagement/definition/RDBMS-relational-database-management-system>

<https://www.uky.edu/~dsianita/622/class2.html>

<https://wachemo-elearning.net/courses>

Weddell, G. [1992] "Reasoning About Functional Dependencies Generalized for Semantic Data Models," TODS, 17:1, March 1992.

Mehrotra, S., et al. [1992] "The Concurrency Control Problem in Multidatabases: Characteristics and Solutions," in SIGMOD [1992].

McFadden, F. R., and Hoffer, J. A. [1994] Modern Database Management, 4th ed., Benjamin Cummings, 1994.

McGee, W. [1977] "The Information Management System IMS/VS, Part I: General Structure and Operation," IBM Systems Journal, 16:2, June 1977.

McLeish, M. [1989] "Further Results on the Security of Partitioned Dynamic Statistical Databases," TODS, 14:1, March 1989.

McLeod, D., and Heimbigner, D. [1985] "A Federated Architecture for Information Systems," TOOIS, 3:3, July 1985.

Mehrotra, S., et al. [1992] "The Concurrency Control Problem in Multidatabases: Characteristics and Solutions," in SIGMOD [1992].

Menasce, D., Popek, G., and Muntz, R. [1980] "A Locking Protocol for Resource Coordination in Distributed Databases," TODS, 5:2, June 1980.

Mendelzon, A., and Maier, D. [1979] "Generalized Mutual Dependencies and the Decomposition of Database Relations," in VLDB [1979].