Chapter-1. Introduction to Database Fundamentals

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, Microsoft ACCESS, or EXCEL. This is a collection of related data with an implicit meaning and hence is a 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. The word *database* is in such common use that we must begin by defining a database.

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 miniworld 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.

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.

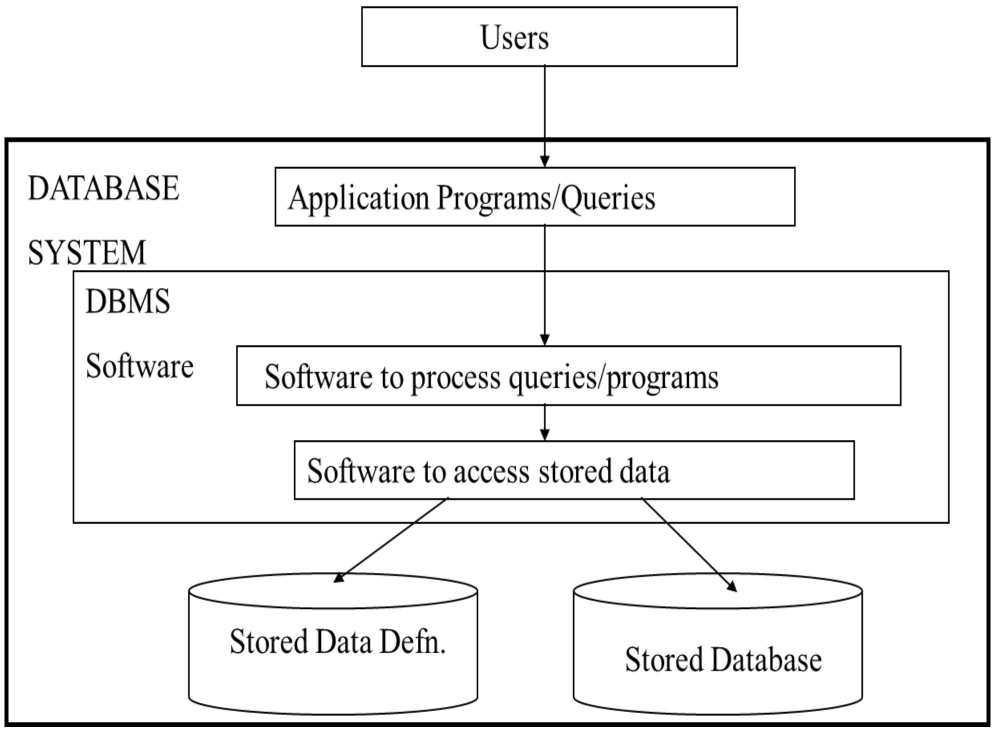
A database management system (DBMS) is a collection of programs that enables users to create and maintain a database. Hence, the DBMS is considered as a *general-purpose software system* that facilitates the processes of *defining, constructing,* and *manipulating* databases for various applications.

Defining a database involves specifying the data types, structures, and constraints for the data to be stored in the database.

Constructing the database is the process of storing the data itself on some storage medium that is controlled by the DBMS.

Manipulating a database includes such functions as querying the database to retrieve specific data, updating the database to reflect changes in the mini world, and generating reports from the data.

It is not necessary to use general-purpose DBMS software to implement a computerized database. We could write our own set of programs to create and maintain the database, in effect creating our own *special-purpose* DBMS software. We will call the database and DBMS software together a database system.


Users
DATABASE SYSTEM
Application Programs/Queries
DBMS Software
Software to process queries/programs
Software to access stored data
Stored Data Defn.
Stored Database

A simplified database system environment :

What is a Database ?

To find out what database is, we have to start from data, which is the basic building block of any DBMS. In a database, data is organized strictly in row and column format. The rows are called Tuple or Record. The data items within one row may belong to different data types. On the other hand, the columns are often called Domain or Attribute. All the data items within a single attribute are of the same data type.

Data: Facts, figures, statistics etc. having no particular meaning (e.g. 1, ABC, 19 etc).

Record: Collection of related data items, e.g. in the above example the three data items had no meaning. But if we organize them in the following way, then they collectively represent meaningful information.

Roll	Name	Age
1	ABC	19

Table or Relation: Collection of related records.

Roll	Name	Age
1	ABC	19
2	DEF	22
3	XYZ	28

The columns of this relation are called Fields, Attributes or Domains. The rows are called Tuples or Records.

Database: Collection of related relations. Consider the following collection of tables:

T1

Roll	Name	Age
1	ABC	19
2	DEF	22
3	XYZ	28

T2

Roll	Address
1	KOL
2	DEL
3	MUM

T3

Roll	Year
1	I
2	II
3	I

T4

Year	Hostel
I	H1
II	H2

We now have a collection of 4 tables. They can be called a “related collection”, because we can clearly find out that there are some common attributes existing in a selected pair of tables. Because of these common attributes we may combine the data of two or more tables together to find out the complete details of a student. Questions like “Which hostel does the youngest student live in?” can be answered now, although *Age* and *Hostel* attributes are in different tables.

What is Management System?

A management system is a set of rules and procedures which help us to create organize and manipulate the database. It also helps us to add, modify delete data items in the database. The management system can be either manual or computerized. It is important because without the existence of some kind of rules and regulations it is not possible to maintain the database. We have to select the particular attributes which should be included in a particular table; the common attributes to create relationship between two tables; if a new record has to be inserted or deleted then which tables should have to be handled etc. These issues must be resolved by having some kind of rules to follow in order to maintain the integrity of the database.

1.1 Data Processing Vs. Data Management Systems

Although Data Processing and Data Management Systems both refer to functions that take raw data and transform it into usable information, the usage of the terms is very different.

Data Processing is the term generally used to describe what was done by large mainframe computers from the late 1940's until the early 1980's (and which continues to be done in most large organizations to a greater or lesser extent even today): large volumes of raw transaction data fed into programs that update a master file, with fixed-format reports written to paper.

The term Data Management Systems refers to an expansion of this concept, where the raw data, previously copied manually from paper to punched cards, and later into data-entry terminals, is now fed into the system from a variety of sources, including ATMs, EFT, and direct customer entry through the Internet. The master file concept has been largely displaced by database management systems, and static reporting replaced or augmented by ad-hoc reporting and direct inquiry, including downloading of data by customers. The ubiquity of the Internet and the Personal Computer have been the driving force in the transformation of Data Processing to the more global concept of Data Management Systems.

1.2 File Oriented Approach

The earliest business computer systems were used to process business records and produce information. They were generally faster and more accurate than equivalent manual systems. These systems stored groups of records in separate files, and so they were called file processing systems. In a typical file processing systems, each department has its own files, designed specifically for those applications. The department itself working with the data processing staff, sets policies or standards for the format and maintenance of its files.

Programs are dependent on the files and vice-versa; that is, when the physical format of the file is changed, the program has also to be changed. Although the traditional file oriented approach to information processing is still widely used, it does have some very important disadvantages.

This typical file-processing system is supported by a conventional operating system. The system stores permanent records in various files, and it needs different application programs to extract records from, and add records to, the appropriate files. Before database management systems (DBMSs) came along, organizations usually stored information in such systems.

Keeping organizational information in a file-processing system has a number of major disadvantages:

1.2.1 Disadvantages of File – Processing Systems :

i) Data redundancy and inconsistency.

Since different programmers create the files and application programs over a long period, the various files are likely to have different formats and the programs may be written in several programming languages. Moreover, the same information may be duplicated in several places (files). For example, the address and telephone number of a particular customer may appear in a file that consists of savings-account records and in a file that consists of checking-account records. This redundancy leads to higher storage and access cost. In addition, it may lead to data inconsistency; that is, the various copies of the same data may no longer agree. For example, a changed customer address may be reflected in savings-account records but not elsewhere in the system.

ii) Difficulty in accessing data.

Suppose that one of the bank officers needs to find out the names of all customers who live within a particular postal-code area. The officer asks the data-processing department to generate such a list. Because the designers of the original system did not anticipate this request, there is no application program on hand to meet it. There is, however, an application program to generate the list of *all* customers. The bank officer has now two choices: either obtain the list of all customers and extract the needed information manually or ask a system programmer to write the necessary application program. Both alternatives are obviously unsatisfactory. Suppose that such a program is written, and that, several days later, the same officer needs to trim that list to include only those customers who have an account balance of $10,000 or more. As expected, a program to generate such a list does not exist. Again, the officer has the preceding two options, neither of which is satisfactory.

The point here is that conventional file-processing environments do not allow needed data to be retrieved in a convenient and efficient manner. More responsive data-retrieval systems are required for general use.

iii) Data isolation.

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

iv) Integrity problems.

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

v) Atomicity problems.

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

vi) Concurrent access anomalies.

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

vii) Security problems.

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

These difficulties, among others, prompted the development of database systems. In what follows, we shall see the concepts and algorithms that enable database systems to solve the problems with file-processing systems.

1.3 Characteristics of the Database Approach

i) Self-Describing Nature of a Database System

ii) Insulation between Programs and Data, and Data Abstraction

iii) Support of Multiple Views of the Data

iv) Sharing of Data and Multiuser Transaction Processing

i) Self-Describing Nature of a Database System

A fundamental characteristic of the database approach is that the database system contains not only the database itself but also a complete definition or description of the database structure and constraints. This definition is stored in the system catalog, which contains information such as the structure of each file, the type and storage format of each data item, and various constraints on the data. The information stored in the catalog is called meta-data, and it describes the structure of the primary database.

The catalog is used by the DBMS software and also by database users who need information about the database structure. A general purpose DBMS software package is not written for a specific database application, and hence it must refer to the catalog to know the structure of the files in a specific database, such as the type and format of data it will access. The DBMS software must work equally well with *any number of database applications*—for example, a university database, a banking database, or a company database—as long as the database definition is stored in the catalog.

ii) Insulation between Programs and Data, and Data Abstraction

The structure of data files is stored in the DBMS catalog separately from the access programs. We call this property program-data independence.

For example, a file access program may be written in such a way that it can access only STUDENT records of the structure. If we want to add another piece of data to each STUDENT record, say the Birthdate, such a program will no longer work and must be changed. By contrast, in a DBMS environment, we just need to change the description of STUDENT records in the catalog to reflect the inclusion of the new data item Birthdate; no programs are changed. The next time a DBMS program refers to the catalog, the new structure of STUDENT records will be accessed and used.

In object-oriented and object-relational databases, users can define operations on data as part of the database definitions. An operation (also called a *function*) is specified in two parts. The *interface* of an operation includes the operation name and the data types of its arguments (or parameters). The *implementation* of the operation is specified separately and can be changed without affecting the interface. User application programs can operate on the data by invoking these operations through their names and arguments, regardless of how the operations are implemented. This may be termed program-operation independence.

The characteristic that allows program-data independence and program-operation independence is called data abstraction. A DBMS provides users with a conceptual representation of data that does not include many of the details of how the data is stored or how the operations are implemented. Informally, a data model is a type of data abstraction that is used to provide this conceptual representation. The data model uses logical concepts, such as objects, their properties, and their interrelationships, that may be easier for most users to understand than computer storage concepts. Hence, the data model *hides* storage and implementation details that are not of interest to most database users.

iii) Support of Multiple Views of the Data.

A database typically has many users, each of whom may require a different perspective or view of the database. A view may be a subset of the database or it may contain virtual data that is derived from the database files but is not explicitly stored. Some users may not need to be aware of whether the data they refer to is stored or derived. A multiuser DBMS whose users have a variety of applications must provide facilities for defining multiple views.

For example, one user of the database may be interested only in the transcript of each student. A second user, who is interested only in checking that students have taken all the prerequisites of each course they register for, may require the view.

iv) Sharing of Data and Multiuser Transaction Processing.

A multiuser DBMS, as its name implies, must allow multiple users to access the database at the same time. This is essential if data for multiple applications is to be integrated and maintained in a single database. The DBMS must include concurrency control software to ensure that several users trying to update the same data do so in a controlled manner so that the result of the updates is correct.

For example, when several reservation clerks try to assign a seat on an airline flight, the DBMS should ensure that each seat can be accessed by only one clerk at a time for assignment to a passenger. These types of applications are generally called on-line transaction processing (OLTP) applications. A fundamental role of multiuser DBMS software is to ensure that concurrent transactions operate correctly.

1.4 Actors on the Scene (The People who works with the database):

Many persons are involved in the design, use, and maintenance of a large database with a few hundred users. Here, we identify the people whose jobs involve the day-to-day use of a large database; we call them the "actors on the scene." We consider people who may be called "workers behind the scene"—those who work to maintain the database system environment, but who are not actively interested in the database itself.

1.4.1 Database Administrators

In any organization where many persons 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, for coordinating and monitoring its use, and for acquiring software and hardware resources as needed. The DBA is accountable for problems such as breach of security or poor system response time. In large organizations, the DBA is assisted by a staff that helps carry out these functions.

1.4.2 Database Designers

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 come up with 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 a view of the database that meets the data and processing requirements of this group. These views are 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.

1.4.3 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:

i) *Casual end users* occasionally access the database, but they may need different information each time. They use a sophisticated database query language to specify their requests and are typically middle- or high-level managers or other occasional browsers.

ii) *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. The tasks that such users perform are varied:

- Bank tellers check account balances and post withdrawals and deposits.
- Reservation clerks for airlines, hotels, and car rental companies check availability for a given request and make reservations.
- Clerks at receiving stations for courier mail enter package identifications via bar codes and descriptive information through buttons to update a central database of received and in-transit packages.

iii) *Sophisticated end users* include engineers, scientists, business analysts, and others who thoroughly familiarize themselves with the facilities of the DBMS so as to implement their applications to meet their complex requirements.

iv) *Stand-alone users* maintain personal databases by using ready-made program packages that provide easy-to-use menu- or graphics-based interfaces. An example is the user of a tax package that stores a variety of personal financial data for tax purposes.

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 have to understand only the types of 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.

Stand-alone users typically become very proficient in using a specific software package.

1.4.4 System Analysts and Application Programmers (Software Engineers)

System analysts determine the requirements of end users, especially naive and parametric end users, and develop specifications for canned transactions that meet these requirements.

Application programmers implement these specifications as programs; then they test, debug, document, and maintain these canned transactions. Such analysts and programmers (nowadays called software engineers) should be familiar with the full range of capabilities provided by the DBMS to accomplish their tasks.

1.5 Workers behind the Scene

In addition to those who design, use, and administer a database, others are associated with the design, development, and operation of the DBMS *software and system environment.* These persons are typically not interested in the database itself. We call them the "workers behind the scene," and they include the following categories.

- *DBMS system designers and implementers* are persons who design and implement the DBMS modules and interfaces as a software package. A DBMS is a complex software system that consists of many components or modules, including modules for implementing the catalog, query language, interface processors, data access, concurrency control, recovery, and security. The DBMS must interface with other system software, such as the operating system and compilers for various programming languages.

- *Tool developers* include persons who design and implement tools—the software packages that facilitate database system design and use, and help improve performance. Tools are optional packages that are often purchased separately. They include packages for database design, performance monitoring, natural language or graphical interfaces, prototyping, simulation, and test data generation. In many cases, independent software vendors develop and market these tools.

- *Operators and maintenance personnel* are the system administration personnel who are responsible for the actual running and maintenance of the hardware and software environment for the database system.

Although the above categories of workers behind the scene are instrumental in making the database system available to end users, they typically do not use the database for their own purposes.

1.6 Advantages of Using a DBMS:

1.6.1 Controlling Redundancy

Redundancy means storing same data more than one. This redundancy in storing the same data multiple times leads to several problems.

First, there is the need to perform a single logical update—such as entering data on a new student—multiple times: once for each file where student data is recorded. This leads to *duplication of effort.*

Second, *storage space is wasted* when the same data is stored repeatedly, and this problem may be serious for large databases.

Third, files that represent the same data may become *inconsistent.* This may happen because an update is applied to some of the files but not to others. Even if an update—such as adding a new student—is applied to all the appropriate files, the data concerning the student may still be *inconsistent* since the updates are applied independently by each user group. For example, one user group may enter a student's birthdate erroneously as JAN-19-1974, whereas the other user groups may enter the correct value of JAN-29-1974.

A DBMS should provide the capability to automatically enforce the rule that no inconsistencies are introduced when data is updated.

1.6.2 Restricting Unauthorized Access

When multiple users share a database, it is likely that some users will not be authorized to access all information in the database. A DBMS should provide a security and authorization subsystem, which the DBA uses to create accounts and to specify account restrictions. The DBMS should then enforce these restrictions automatically. Notice that we can apply similar controls to the DBMS software. For example, only the DBA's staff may be allowed to use certain privileged software, such as the software for creating new accounts. Similarly, parametric users may be allowed to access the database only through the canned transactions developed for their use.

1.6.3 Providing Persistent Storage for Program Objects and Data Structures

Databases can be used to provide persistent storage for program objects and data structures. This is one of the main reasons for the emergence of the object-oriented database systems. Programming languages typically have complex data structures, such as record types in PASCAL or class definitions in C++.

The persistent storage of program objects and data structures is an important function of database systems. Traditional database systems often suffered from the so-called impedance mismatch problem, since the data structures provided by the DBMS were incompatible with the programming language's data structures. Object-oriented database systems typically offer data structure compatibility with one or more object-oriented programming languages.

1.6.4 Permitting Inferencing and Actions Using Rules

Some database systems provide capabilities for defining *deduction rules* for *inferencing* new information from the stored database facts. Such systems are called deductive database systems. For example, there may be complex rules in the miniworld application for determining when a student is on probation. These can be specified *declaratively* as rules, which when compiled and maintained by the DBMS can determine all students on probation. In a traditional DBMS, an explicit *procedural program code* would have to be written to support such applications. But if the miniworld rules change, it is generally more convenient to change the declared deduction rules than to recode procedural programs. More powerful functionality is provided by active database systems, which provide active rules that can automatically initiate actions.

1.6.5 Providing Multiple User Interfaces

Because many types of users with varying levels of technical knowledge use a database, a DBMS should provide a variety of user interfaces.

These include

- query languages for casual users;
- programming language interfaces for application programmers;
- forms and command codes for parametric users; and
- menu-driven interfaces and natural language interfaces for stand-alone users.

Both forms-style interfaces and menu-driven interfaces are commonly known as graphical user interfaces (GUIs). Many specialized languages and environments exist for specifying GUIs. Capabilities for providing World Wide Web access to a database—or web-enabling a database—are also becoming increasingly common.

1.6.6 Representing Complex Relationships Among Data

A database may include numerous varieties of data that are interrelated in many ways. A DBMS must have the capability to represent a variety of complex relationships among the data as well as to retrieve and update related data easily and efficiently.

1.6.7 Enforcing Integrity Constraints

Most database applications have certain integrity constraints that must hold for the data. A DBMS should provide capabilities for defining and enforcing these constraints.

- The simplest type of integrity constraint involves specifying a data type for each data item.
- A more complex type of constraint that occurs frequently involves specifying that a record in one file must be related to records in other files.
- Another type of constraint specifies uniqueness on data item values, such as "every course record must have a unique value for CourseNumber".

These constraints are derived from the meaning or semantics of the data and of the miniworld it represents. It is the database designers' responsibility to identify integrity constraints during database design. Some constraints can be specified to the DBMS and automatically enforced. Other constraints may have to be checked by update programs or at the time of data entry.

1.6.8 Providing Backup and Recovery

A DBMS must provide facilities for recovering from hardware or software failures. The backup and recovery subsystem of the DBMS is responsible for recovery. For example, if the computer system fails in the middle of a complex update program, the recovery subsystem is responsible for making sure that the database is restored to the state it was in before the program started executing. Alternatively, the recovery subsystem could ensure that the program is resumed from the point at which it was interrupted so that its full effect is recorded in the database.

1.7 Additional Implications of using the Database Approach

In addition to the above issues, there are other implications of using the database approach that can benefit most organizations.

1.7.1 Potential for Enforcing Standards

The database approach permits the DBA to define and enforce standards among database users in a large organization. This facilitates communication and cooperation among various departments, projects, and users within the organization. Standards can be defined for names and formats of data elements, display formats, report structures, terminology, and so on. The DBA can enforce standards in a centralized database environment more easily than in an environment where each user group has control of its own files and software.

1.7.2 Reduced Application Development Time

A prime selling feature of the database approach is that developing a new application—such as the retrieval of certain data from the database for printing a new report—takes very little time. Designing and implementing a new database from scratch may take more time than writing a single specialized file application. However, once a database is up and running, substantially less time is generally required to create new applications using DBMS facilities. Development time using a DBMS is estimated to be one-sixth to one-fourth of that for a traditional file system.

1.7.3 Flexibility

It may be necessary to change the structure of a database as requirements change. For example, a new user group may emerge that needs information not currently in the database. In response, it may be necessary to add a file to the database or to extend the data elements in an existing file. Modern DBMSs allow certain types of changes to the structure of the database without affecting the stored data and the existing application programs.

1.7.4 Availability of Up-to-Date Information

A DBMS makes the database available to all users. As soon as one user’s update is applied to the database, all other users can immediately see this update. This availability of up-to-date information is essential for many transaction-processing applications, such as reservation systems or banking databases, and it is made possible by the concurrency control and recovery subsystems of a DBMS.

1.7.5 Economies of Scale

The DBMS approach permits consolidation of data and applications, thus reducing the amount of wasteful overlap between activities of data-processing personnel in different projects or departments. This enables the whole organization to invest in more powerful processors, storage devices, or communication gear, rather than having each department purchase its own (weaker) equipment. This reduces overall costs of operation and management.

1.8 When Not to Use a DBMS

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

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

Additional problems may arise if the database designers and DBA do not properly design the database or if the database systems applications are not implemented properly. Hence, it may be more desirable to use regular files under the following circumstances:

- The database and applications are simple, well defined, and not expected to change.
- There are stringent real-time requirements for some programs that may not be met because of DBMS overhead.
- Multiple-user access to data is not required.

Review Questions

a) Define the following terms: *data, database, DBMS, database system, database catalog, program-data independence, user view, DBA, end user, canned transaction, deductive database system, persistent object, meta-data, transaction processing application.*

b) What three main types of actions involve databases? Briefly discuss each.

c) Discuss the main characteristics of the database approach and how it differs from traditional file systems.

d) What are the responsibilities of the DBA and the database designers?

e) What are the different types of database end users? Discuss the main activities of each.

f) Discuss the capabilities that should be provided by a DBMS.

Chapter 2: Database System Concepts and Architecture

The architecture of DBMS packages has evolved from the early monolithic systems, where the whole DBMS software package is one tightly integrated system, to the modern DBMS packages that are modular in design, with a client-server system architecture. This evolution mirrors the trends in computing, where the large centralized mainframe computers are being replaced by hundreds of distributed workstations and personal computers connected via communications networks. In a basic client-server architecture, the system functionality is distributed between two types of modules. A client module is typically designed so that it will run on a user workstation or personal computer. Typically, application programs and user interfaces that access the database run in the client module. Hence, the client module handles user interaction and provides the user-friendly interfaces such as forms or menu-based GUIs (*g*raphical *u*ser *i*nterfaces). The other kind of module, called a server module, typically handles data storage, access, search, and other functions.

2.1 Data Models, Schemas, and Instances

One fundamental characteristic of the database approach is that it provides some level of *data abstraction* by hiding details of data storage that are irrelevant to database users.

A data model —It is a collection of concepts that can be used to describe the conceptual/logical structure of a database. It provides the necessary means to achieve this abstraction.

- By *structure* is meant the data types, relationships, and constraints that should hold for the data.
- Most data models also include a set of basic operations for specifying retrievals/updates.
- Object-oriented data models include the idea of objects having behavior (i.e., applicable methods) being stored in the database (as opposed to purely "passive" data).

According to C.J. Date (one of the leading database experts), a data model is an abstract, self-contained, logical definition of the objects, operators, and so forth, that together constitute the *abstract machine* with which users interact. The objects allow us to model the *structure* of data; the operators allow us to model its *behavior*.

In the *relational* data model, data is viewed as being organized in two-dimensional tables comprised of tuples of attribute values. This model has operations such as Project, Select, and Join.

A data model is not to be confused with its implementation, which is a physical realization on a real machine of the components of the *abstract machine* that together constitute that model.

Logical vs. physical!!

There are other well-known data models that have been the basis for database systems. The best-known models pre-dating the relational model are the hierarchical (in which the entity types form a tree) and the network (in which the entity types and relationships between them form a graph).

2.1.1 Categories of Data Models

Many data models have been proposed, and we can categorize them according to the types of concepts they use to describe the database structure.

High-level or conceptual data models provide concepts that are close to the way many users perceive data.

Low-level or physical data models provide concepts that describe the details of how data is stored in the computer. Concepts provided by low-level data models are generally meant for computer specialists, not for typical end users.

Between these two extremes is a class of representational (or implementation) data models, which provide concepts that may be understood by end users but that are not too far removed from the way data is organized within the computer.

Representational data models hide some details of data storage but can be implemented on a computer system in a direct way.

Conceptual data models use concepts such as entities, attributes, and relationships.

Entity : An entity represents a real-world object or concept, such as an employee or a project, that is described in the database.

Attribute : An attribute represents some property of interest that further describes an entity, such as the employee's name or salary.

Relationship : A relationship among two or more entities represents an interaction among the entities; for example, a works-on relationship between an employee and a project.

Representational or implementation data models are the models used most frequently in traditional commercial DBMSs, and they include the widely-used relational data model, as well as the so-called legacy data models—the network and hierarchical models—that have been widely used in the past.

We can regard object data models as a new family of higher-level implementation data models that are closer to conceptual data models. Object data models are also frequently utilized as high-level conceptual models, particularly in the software engineering domain.

Physical data models describe how data is stored in the computer by representing information such as record formats, record orderings, and access paths. An access path is a structure that makes the search for particular database records efficient.

What is a database model ?

A database model shows the logical structure of a database, including the relationships and constraints that determine how data can be stored and accessed. Individual database models are designed based on the rules and concepts of whichever broader data model the designers adopt. Most data models can be represented by an accompanying database diagram.

Types of Database Models :

There are many kinds of data models. Some of the most common ones include:

a) Hierarchical database model
b) Network model
c) Relational model
d) Object-oriented database model
e) Entity-relationship model

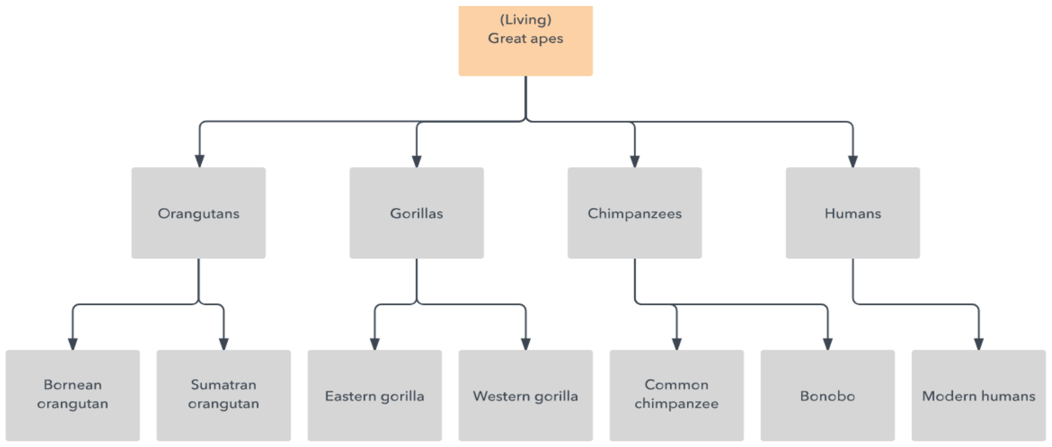
You may choose to describe a database with any one of these depending on several factors. The biggest factor is whether the database management system you are using supports a particular model. Most database management systems are built with a particular data model in mind and require their users to adopt that model, although some do support multiple models.

In addition, different models apply to different stages of the database design process. High-level conceptual data models are best for mapping out relationships between data in ways that people perceive that data. Record-based logical models, on the other hand, more closely reflect ways that the data is stored on the server.

Selecting a data model is also a matter of aligning your priorities for the database with the strengths of a particular model, whether those priorities include speed, cost reduction, usability, or something else.

a) Hierarchical model

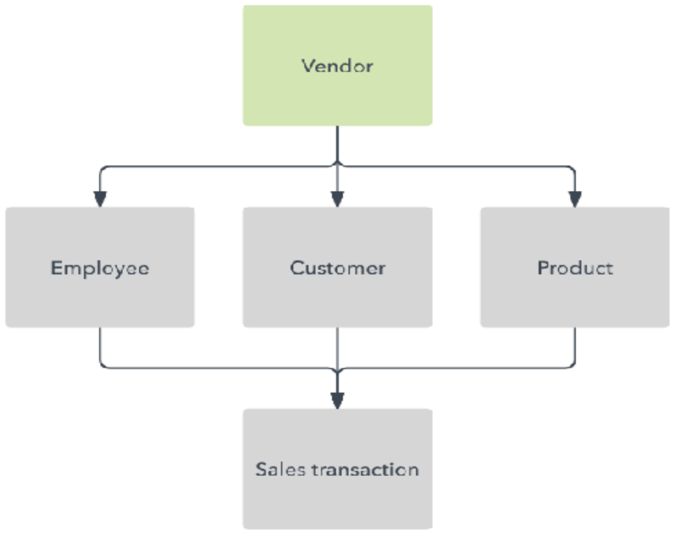
The hierarchical model organizes data into a tree-like structure, where each record has a single parent or root. Sibling records are sorted in a particular order. That order is used as the physical order for storing the database. This model is good for describing many real-world relationships.



This model was primarily used by IBM's Information Management Systems in the 60s and 70s, but they are rarely seen today due to certain operational inefficiencies.

b) Network Model :

The network model builds on the hierarchical model by allowing many-to-many relationships between linked records, implying multiple parent records. Based on mathematical set theory, the model is constructed with sets of related records. Each set consists of one owner or parent record and one or more member or child records. A record can be a member or child in multiple sets, allowing this model to convey complex relationships.



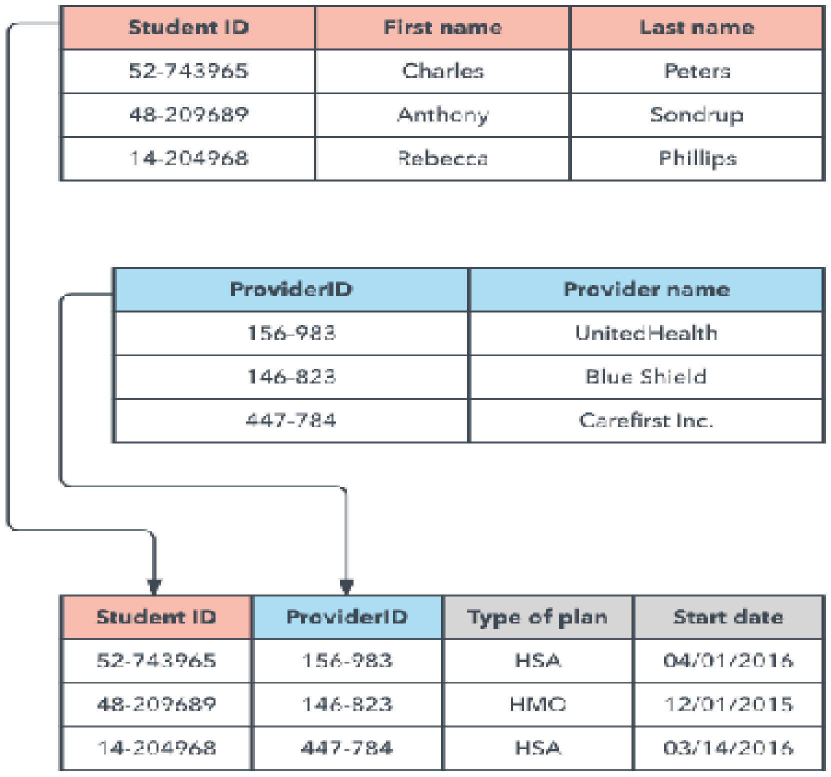
It was most popular in the 70s after it was formally defined by the Conference on Data Systems Languages (CODASYL).

c) Relational Model

The most common model, the relational model sorts data into tables, also known as relations, each of which consists of columns and rows. Each column lists an attribute of the entity in question, such as price, zip code, or birth date. Together, the attributes in a relation are called a domain. A particular attribute or combination of attributes is chosen as a primary key that can be referred to in other tables, when it’s called a foreign key.

Each row, also called a tuple, includes data about a specific instance of the entity in question, such as a particular employee.

The model also accounts for the types of relationships between those tables, including one-to-one, one-to-many, and many-to-many relationships. Here’s an example:



Student ID	First name	Last name
52-743965	Charles	Peters
48-209689	Anthony	Sondrup
14-204968	Rebecca	Phillips

ProviderID	Provider name
156-983	UnitedHealth
146-823	Blue Shield
447-784	Carefirst Inc.

Student ID	ProviderID	Type of plan	Start date
52-743965	156-983	HSA	04/01/2016
48-209689	146-823	HMO	12/01/2015
14-204968	447-784	HSA	03/14/2016

Within the database, tables can be normalized, or brought to comply with normalization rules that make the database flexible, adaptable, and scalable. When normalized, each piece of data is atomic, or broken into the smallest useful pieces.

Relational databases are typically written in Structured Query Language (SQL). The model was introduced by E.F. Codd in 1970.

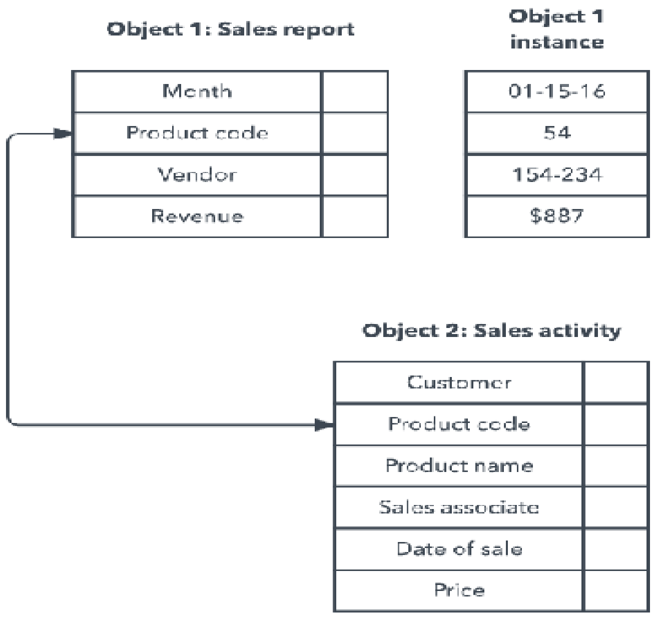
a) Object-oriented database model

This model defines a database as a collection of objects, or reusable software elements, with associated features and methods. There are several kinds of object-oriented databases:

A multimedia database incorporates media, such as images, that could not be stored in a relational database.

A hypertext database allows any object to link to any other object. It's useful for organizing lots of disparate data, but it's not ideal for numerical analysis.

The object-oriented database model is the best known post-relational database model, since it incorporates tables, but isn't limited to tables. Such models are also known as hybrid database models.



b) Entity Relationship Model :

This model captures the relationships between real-world entities much like the network model, but it isn't as directly tied to the physical structure of the database. Instead, it's often used for designing a database conceptually.

Here, the people, places, and things about which data points are stored are referred to as entities, each of which has certain attributes that together make up their domain. The cardinality, or relationships between entities, are mapped as well.

2.1.2 Schemas, Instances, and Database State

Schema : In any data model it is important to distinguish between the *description* of the database and the *database itself.* The description of a database is called the database schema, which is specified during database design and is not expected to change frequently.

Most data models have certain conventions for displaying the schemas as diagrams. A displayed schema is called a schema diagram. A schema diagram displays only *some aspects* of a schema, such as the names of record types and data items, and some types of constraints. Other aspects are not specified in the schema diagram. Many types of constraints are not represented in schema diagrams.

Instance : The data in the database at a particular moment in time is called a database state or snapshot. It is also called the *current* set of occurrences or instances in the database. The actual data in a database may change quite frequently. In a given database state, each schema construct has its own *current set* of instances. Many database states can be constructed to correspond to a particular database schema. Every time we insert or delete a record, or change the value of a data item in a record, we change one state of the database into another state.

Database State : It is the current state of the database. The distinction between database schema and database state is very important. When we define a new database, we specify its database schema only to the DBMS. At this point, the corresponding database state is the *empty state* with no data. We get the *initial state* of the database when the database is first populated or loaded with the initial data. From then on, every time an update operation is applied to the database, we get another database state. At any point in time, the database has a *current state.*

The DBMS is partly responsible for ensuring that *every* state of the database is a valid state—that is, a state that satisfies the structure and constraints specified in the schema. Hence, specifying a correct schema to the DBMS is extremely important, and the schema must be designed with the utmost care.

Meta-data : It is the description of the schema constructs and constraints. The DBMS stores schema constructs and its constraints. It is also called as the system catalog. The DBMS software can refer to the schema whenever it needs. The schema is sometimes called the intension, and a database state an extension of the schema.

Schema Evolution : The schema is not supposed to change frequently, it is not uncommon that changes need to be applied to the schema once in a while as the application requirements change. This is known as schema evolution. Most modern DBMSs include some operations for schema evolution that can be applied while the database is operational.

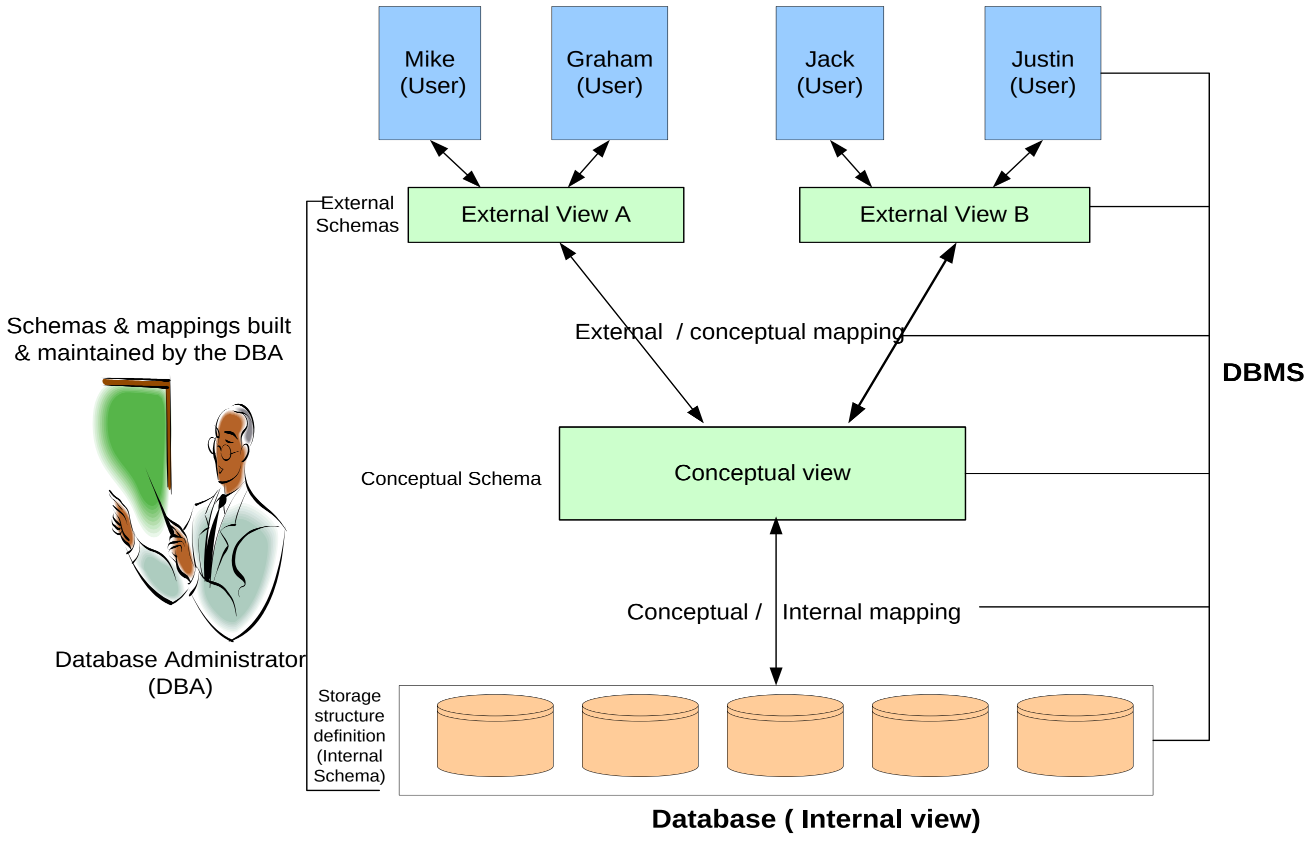
2.2 DBMS Architecture and Data Independence :

2.2.1 The Three-Schema Architecture

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

1. The internal level has an internal schema, which describes the physical storage structure of the database. The internal schema uses a physical data model and describes the complete details of data storage and access paths for the database.
2. The conceptual level has a conceptual schema, which describes the structure of the whole database for a community of users. The conceptual schema hides the details of physical storage structures and concentrates on describing entities, data types, relationships, user operations, and constraints. A high-level data model or an implementation data model can be used at this level.
3. The external or view level includes a number of external schemas or user views. Each external schema describes the part of the database that a particular user group is interested in and hides the rest of the database from that user group. A high-level data model or an implementation data model can be used at this level.

The three-schema architecture is a convenient tool for the user to visualize the schema levels in a database system. Most DBMSs do not separate the three levels completely, but support the three-schema architecture to some extent. Some DBMSs may include physical-level details in the conceptual schema. In most DBMSs that support user views, external schemas are specified in the same data model that describes the conceptual-level information. Some DBMSs allow different data models to be used at the conceptual and external levels.


Mike (User)
Graham (User)
Jack (User)
Justin (User)
External Schemas
External View A
External View B
Schemas & mappings built & maintained by the DBA
External / conceptual mapping
DBMS
Conceptual Schema
Conceptual view
Conceptual / Internal mapping
Database Administrator (DBA)
Storage structure definition (Internal Schema)
Database (Internal view)

In the above diagram,

- It shows the architecture of DBMS.
- Mapping is the process of transforming request response between various database levels of architecture.
- Mapping is not good for small database, because it takes more time.
- In External / Conceptual mapping, DBMS transforms a request on an external schema against the conceptual schema.
- In Conceptual / Internal mapping, it is necessary to transform the request from the conceptual to internal levels.

1. Physical Level

- Physical level describes the physical storage structure of data in database.
- It is also known as Internal Level.
- This level is very close to physical storage of data.

- At lowest level, it is stored in the form of bits with the physical addresses on the secondary storage device.
- At highest level, it can be viewed in the form of files.
- The internal schema defines the various stored data types. It uses a physical data model.

2. Conceptual Level

- Conceptual level describes the structure of the whole database for a group of users.
- It is also called as the data model.
- Conceptual schema is a representation of the entire content of the database.
- These schema contains all the information to build relevant external records.
- It hides the internal details of physical storage.

3. External Level

- External level is related to the data which is viewed by individual end users.
- This level includes a no. of user views or external schemas.
- This level is closest to the user.
- External view describes the segment of the database that is required for a particular user group and hides the rest of the database from that user group.

2.2.2 Data Independence :

It can be defined as the capacity to change the schema at one level of a database system without having to change the schema at the next higher level. We can define two types of data independence:

1. *Logical data independence :* It is the capacity to change the conceptual schema without having to change external schemas or application programs. We may change the conceptual schema to expand the database (by adding a record type or data item), or to reduce the database (by removing a record type or data item). In the latter case, external schemas that refer only to the remaining data should not be affected. Only the view definition and the mappings need be changed in a DBMS that supports logical data independence. Application programs that reference the external schema constructs must work as before, after the conceptual schema undergoes a logical reorganization. Changes to constraints can be applied also to the conceptual schema without affecting the external schemas or application programs.
2. *Physical data independence :* It is the capacity to change the internal schema without having to change the conceptual (or external) schemas. Changes to the internal schema may be needed because some physical files had to be reorganized—for example, by creating additional access structures—to improve the performance of retrieval or update. If the same data as before remains in the database, we should not have to change the conceptual schema.

2.3 Database Languages and Interfaces

A DBMS supports a variety of users and must provide appropriate languages and interfaces for each category of users.

2.3.1 DBMS Languages :

DDL (Data Definition Language): used (by the DBA and/or database designers) to specify the conceptual schema. The DBMS will have a DDL compiler whose function is to process DDL statements in order to identify descriptions of the schema constructs and to store the schema description in the DBMS catalog.

In DBMSs where a clear separation is maintained between the conceptual and internal levels. The mappings between the two schemas may be specified in either one of these languages.

SDL (Storage Definition Language): used for specifying the internal schema.

VDL (View Definition Language): used for specifying the external schemas (i.e., user views).

DML (Data Manipulation Language): used for performing operations such as retrieval, insertion, deletion and modification of data of the database.

SQL : In current DBMSs, the preceding types of languages are usually *not considered distinct languages;* rather, a comprehensive integrated language is used that includes constructs for conceptual schema definition, view definition, and data manipulation. Storage definition is typically kept separate, since it is used for defining physical storage structures to fine-tune the performance of the database system, and it is usually utilized by the DBA staff. A typical example of a comprehensive database language is the SQL relational database language, which represents a combination of DDL, VDL, and DML, as well as statements for constraint specification and schema evolution. The SDL was a component in earlier versions of SQL but has been removed from the language to keep it at the conceptual and external levels only.

2.3.2 DBMS Interfaces

User-friendly interfaces provided by a DBMS may include the following.

Menu-Based Interfaces for Browsing

These interfaces present the user with lists of options, called **menus,** that lead the user through the formulation of a request. Menus do away with the need to memorize the specific commands and syntax of a query language; rather, the query is composed step by step by picking options from a menu that is displayed by the system. Pull-down menus are becoming a very popular technique in window-based user interfaces. They are often used in **browsing interfaces,** which allow a user to look through the contents of a database in an exploratory and unstructured manner.

Forms-Based Interfaces

A forms-based interface displays a form to each user. Users can fill out all of the form entries to insert new data, or they fill out only certain entries, in which case the DBMS will retrieve matching data for the remaining entries. Forms are usually designed and programmed for naive users as interfaces to canned transactions. Many DBMSs have forms specification languages, special languages that help programmers specify such forms. Some systems have utilities that define a form by letting the end user interactively construct a sample form on the screen.

Graphical User Interfaces

A graphical interface (GUI) typically displays a schema to the user in diagrammatic form. The user can then specify a query by manipulating the diagram. In many cases, GUIs utilize both menus and forms. Most GUIs use a pointing device, such as a mouse, to pick certain parts of the displayed schema diagram.

Natural Language Interfaces

These interfaces accept requests written in English or some other language and attempt to "understand" them. A natural language interface usually has its own "schema," which is similar to the database conceptual schema. The natural language interface refers to the words in its schema, as well as to a set of standard words, to interpret the request. If the interpretation is successful, the interface generates a high-level query corresponding to the natural language request and submits it to the DBMS for processing; otherwise, a dialogue is started with the user to clarify the request.

Interfaces for Parametric Users

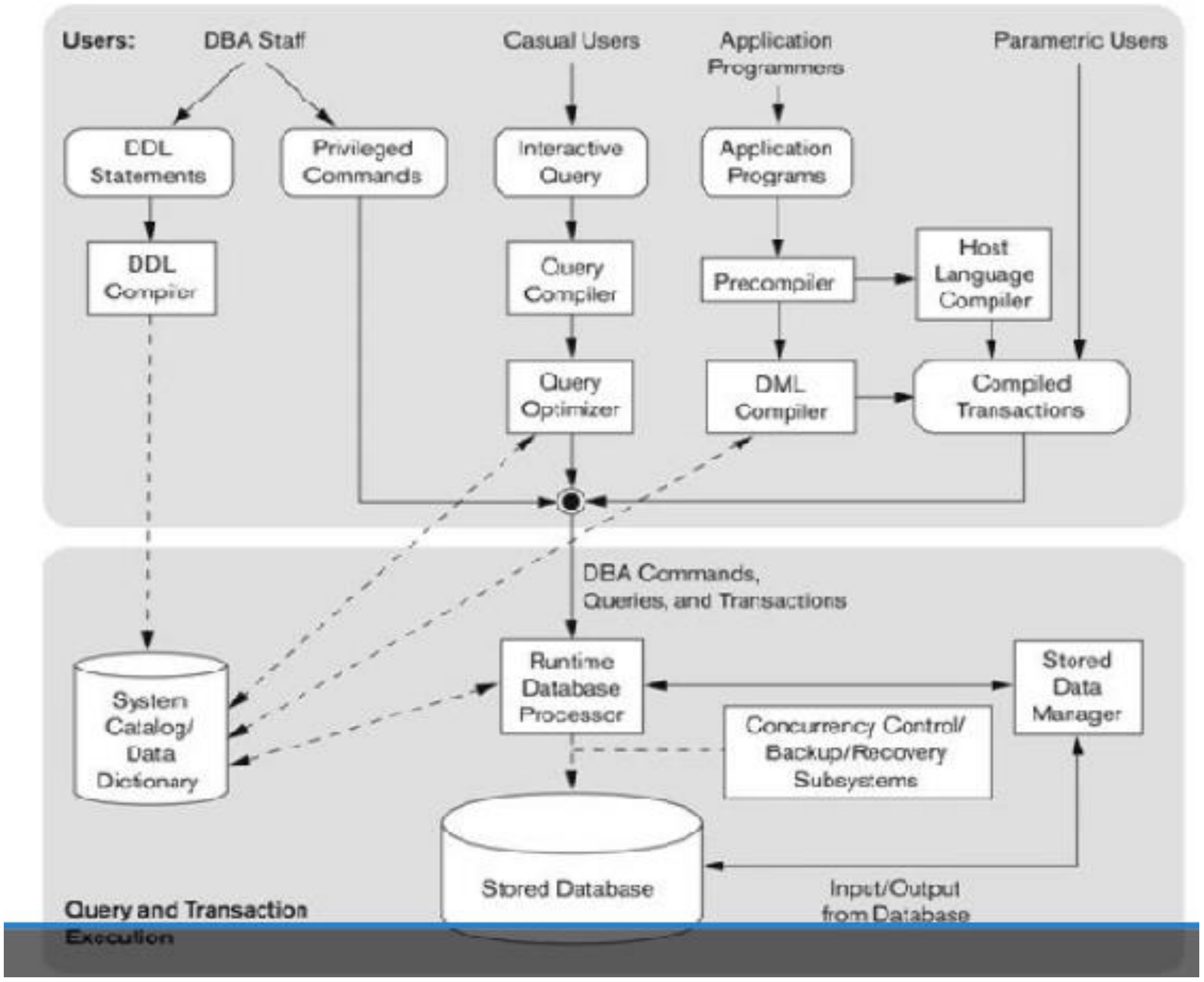
Parametric users, such as bank tellers, often have a small set of operations that they must perform repeatedly. Systems analysts and programmers design and implement a special interface for a known class of naive users. Usually, a small set of abbreviated commands is included, with the goal of minimizing the number of keystrokes required for each request. For example, function keys in a terminal can be programmed to initiate the various commands. This allows the parametric user to proceed with a minimal number of keystrokes.

Interfaces for the DBA

Most database systems contain privileged commands that can be used only by the DBA's staff. These include commands for creating accounts, setting system parameters, granting account authorization, changing a schema, and reorganizing the storage structures of a database.

2.4 The Database System Environment

A DBMS is a complex software system. Here, it has been discussed the types of software components that constitute a DBMS and the types of computer system software with which the DBMS interacts. Below Figure illustrates, the simplified form along with the typical DBMS components. The database and the DBMS catalog are usually stored on disk. Access to the disk is controlled primarily by the operating system (OS), which schedules disk input/output. A higher-level stored data manager module of the DBMS controls access to DBMS information that is stored on disk, whether it is part of the database or the catalog. The dotted lines and circles marked in below Figure illustrate accesses that are under the control of this stored data manager. The stored data manager may use basic OS services for carrying out low-level data transfer between the disk and computer main storage, but it controls other aspects of data transfer, such as handling buffers in main memory. Once the data is in main memory buffers, it can be processed by other DBMS modules, as well as by application programs.



The DDL compiler processes schema definitions, specified in the DDL, and stores descriptions of the schemas (meta-data) in the DBMS catalog. The catalog includes information such as the names of files, data items, storage details of each file, mapping information among schemas, and constraints, in addition to many other types of information that are needed by the DBMS modules. DBMS software modules then look up the catalog information as needed.

The run-time database processor handles database accesses at run time; it receives retrieval or update operations and carries them out on the database. Access to disk goes through the stored data manager. The query compiler handles high-level queries that are entered interactively. It parses, analyzes, and compiles or interprets a query by creating database access code, and then generates calls to the run-time processor for executing the code.

The pre-compiler extracts DML commands from an application program written in a host programming language. These commands are sent to the DML compiler for compilation into object code for database access. The rest of the program is sent to the host language compiler. The object codes for the DML commands and the rest of the program are linked, forming a canned transaction whose executable code includes calls to the runtime database processor.

The DBMS interacts with the operating system when disk accesses—to the database or to the catalog—are needed. If the computer system is shared by many users, the OS will schedule DBMS disk access requests and DBMS processing along with other processes. The DBMS also interfaces with compilers for general-purpose host programming languages. User-friendly interfaces to the DBMS can be provided to help any of the user types shown in above Figure to specify their requests.

2.4.2 Database System Utilities

In addition to possessing the software modules just described, most DBMSs have database utilities that help the DBA in managing the database system. Common utilities have the following types of functions:

1. *Loading:* A loading utility is used to load existing data files—such as text files or sequential files—into the database. Usually, the current (source) format of the data file and the desired (target) database file structure are specified to the utility, which then automatically reformats the data and stores it in the database. With the proliferation of DBMSs, transferring data from one DBMS to another is becoming common in many organizations. Some vendors are offering products that generate the appropriate loading programs, given the existing source and target database storage descriptions (internal schemas). Such tools are also called conversion tools.

2. *Backup:* A backup utility creates a backup copy of the database, usually by dumping the entire database onto tape. The backup copy can be used to restore the database in case of catastrophic failure. Incremental backups are also often used, where only changes since the previous backup are recorded. Incremental backup is more complex but it saves space.

3. *File reorganization:* This utility can be used to reorganize a database file into a different file organization to improve performance.

4. *Performance monitoring:* Such a utility monitors database usage and provides statistics to the DBA. The DBA uses the statistics in making decisions such as whether or not to reorganize files to improve performance.

Other utilities may be available for sorting files, handling data compression, monitoring access by users, and performing other functions.

2.4.3 Tools, Application Environments, and Communications Facilities

Other tools are often available to database designers, users, and DBAs. CASE tools are used in the design phase of database systems. Another tool that can be quite useful in large organizations is an expanded data dictionary (or data repository) system. In addition to storing catalog information about schemas and constraints, the data dictionary stores other information, such as design decisions, usage standards, application program descriptions, and user information. Such a system is also called an information repository. This information can be accessed *directly* by users or the DBA when needed. A data dictionary utility is similar to the DBMS catalog, but it includes a wider variety of information and is accessed mainly by users rather than by the DBMS software.

Application development environments, such as the PowerBuilder system, are becoming quite popular. These systems provide an environment for developing database applications and include facilities that help in many facets of database systems, including database design, GUI development, querying and updating, and application program development.

The DBMS also needs to interface with communications software, whose function is to allow users at locations remote from the database system site to access the database through computer terminals, workstations, or their local personal computers. These are connected to the database site through data communications hardware such as phone lines, long-haul networks, local-area networks, or satellite communication devices. Many commercial database systems have communication packages that work with the DBMS. The integrated DBMS and data communications system is called a DB/DC system. In addition, some distributed DBMSs are physically distributed over multiple machines. In this case, communications networks are needed to connect the machines. These are often local area networks (LANs) but they can also be other types of networks.

2.5 Architectures of DBMS

The design of a DBMS depends on its architecture. It can be centralized or decentralized or hierarchical. The architecture of a DBMS can be seen as either single tier or multi-tier. An n-tier architecture divides the whole system into related but independent **n** modules, which can be independently modified, altered, changed, or replaced.

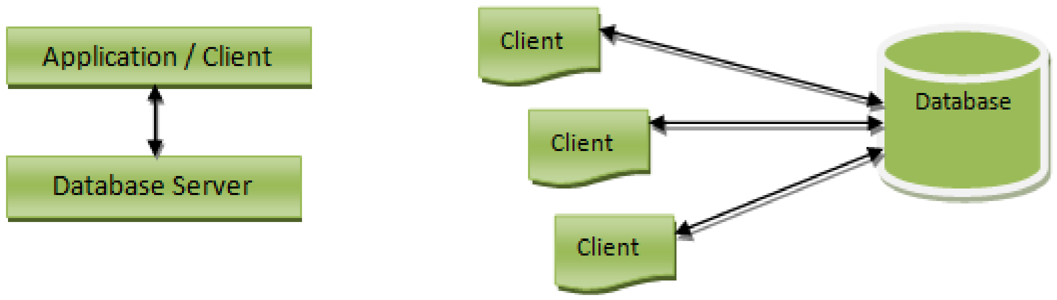
a) 1-tier architecture:

Here, the DBMS is the only entity where the user directly sits on the DBMS and uses it. Both client programs and server programs runs on the same machine. i.e. only one computer can be used as both client machine and server machine. Any changes done here will directly be done on the DBMS itself. It does not provide handy tools for end-users. Database designers and programmers normally prefer to use single-tier architecture.

b) 2-tier architecture :

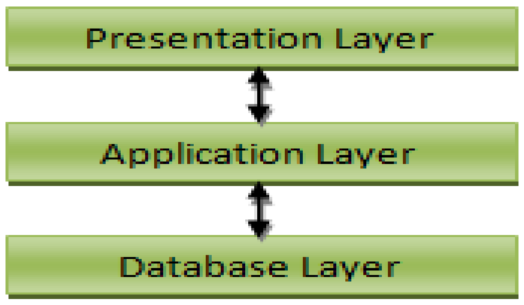
Here, there will be two computers in which one machine runs client application and another machine runs server part of the application. In 2-tier architecture, application program directly interacts with the database. There will not be any user interface or the user involved with database interaction. Imagine a front end application of School, where we need to display the reports of all the students who are opted for different subjects. In this case, the application will directly interact with the database and retrieve all required data. Here no inputs from the user are required. This involves 2-tier architecture of the database.

Programmers use 2-tier architecture where they access the DBMS by means of an application. Here the application tier is entirely independent of the database in terms of operation, design, and programming.



c) 3-tier architecture :

3-tier architecture is the most widely used database architecture. It can be viewed as below.



- *Presentation layer / User layer* is the layer where user uses the database. He does not have any knowledge about underlying database. He simply interacts with the database as though he has all data in front of him. You can imagine this layer as a registration form where you will be inputting your details. Did you ever guessed, after pressing 'submit' button where the data goes? No right? You just know that your details are saved. This is the presentation layer where all the details from the user are taken, sent to the next layer for processing.

- *Application layer* is the underlying program which is responsible for saving the details that you have entered, and retrieving your details to show up in the page. This layer has all the business logics like validation, calculations and manipulations of data, and then sends the requests to database to get the actual data. If this layer sees that the request is invalid, it sends back the message to presentation layer. It will not hit the database layer at all.

- *Data layer or Database layer* is the layer where actual database resides. In this layer, all the tables, their mappings and the actual data present. When you save you details from the front end, it will be inserted into the respective tables in the database layer, by using the programs in the application layer. When you want to view your details in the web browser, a request is sent to database layer by application layer. The database layer fires queries and gets the data. These data are then transferred to the browser (presentation layer) by the programs in the application layer.

d) N-tier / Multi-tier architecture :

Multitier architecture (often referred to as *n*-tier architecture) or multilayered architecture is a client–server architecture in which presentation, application processing, and data management functions are physically separated. *N*-tier application architecture provides a model by which developers can create flexible and reusable applications. By segregating an application into tiers, developers acquire the option of modifying or adding a specific layer, instead of reworking the entire application. It will distribute the work load to several machines so that processing time will be reduced.

2.6 Classification of Database Management Systems :

Several criteria are normally used to classify DBMSs.

i) Based on the data model :

The two types of data models used in many current commercial DBMSs are the relational data model and the object data model. Many legacy applications still run on database systems based on the hierarchical and network data models.

The relational DBMSs are evolving continuously, and, in particular, have been incorporating many of the concepts that were developed in object databases. This has led to a new class of DBMSs that are being called object-relational DBMSs. We can hence categorize DBMSs based on the data model: relational, object, object-relational, hierarchical, network, and other.

ii) Based on number of users :

Single-user systems support only one user at a time and are mostly used with personal computers. Multiuser systems, which include the majority of DBMSs, support multiple users concurrently.

iii) Based on the number of sites over which database is distributed :

A DBMS is centralized if the data is stored at a single computer site.

A centralized DBMS can support multiple users, but the DBMS and the database themselves reside totally at a single computer site.

A distributed DBMS (DDBMS) can have the actual database and DBMS software distributed over many sites, connected by a computer network.

Homogeneous DDBMSs use the same DBMS software at multiple sites. A recent trend is to develop software to access several autonomous preexisting databases stored under heterogeneous DBMSs. This leads to a federated DBMS (or multidatabase system), where the participating DBMSs are loosely coupled and have a degree of local autonomy. Many DDBMSs use a client-server architecture.

iv) Based on the cost of database :

A fourth criterion is the **cost** of the DBMS. The majority of DBMS packages cost between $10,000 and $100,000. Single-user low-end systems that work with microcomputers cost between $100 and $3000. At the other end, a few elaborate packages cost more than $100,000.

v) Based on the type of access path options for storing files :

One well-known family of DBMSs is based on inverted file structures. Finally, a DBMS can be general-purpose or special-purpose. When performance is a primary consideration, a special-purpose DBMS can be designed and built for a specific application; such a system cannot be used for other applications without major changes. Many airline reservations and telephone directory systems developed in the past are special-purpose DBMSs. These fall into the category of on-line transaction processing (OLTP) systems, which must support a large number of concurrent transactions without imposing excessive delays.

Review Questions :

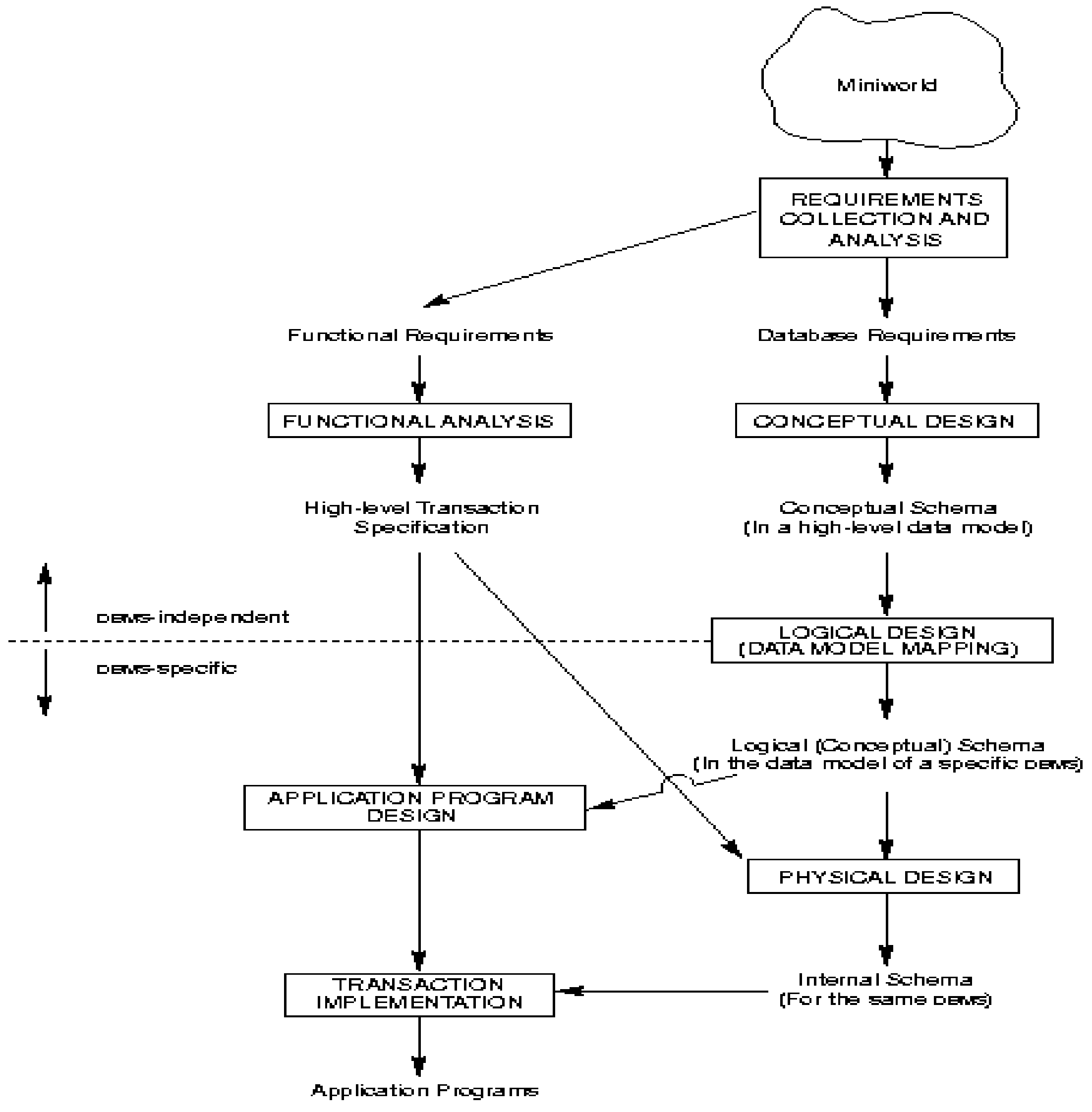
a) Define the following terms: *data model, database schema, database state, internal schema, conceptual schema, external schema, data independence, DDL, DML, SDL, VDL, query language, host language, data sublanguage, database utility, catalog, client-server architecture*
b) Discuss the main categories of data models.
c) What is the difference between a database schema and a database state?
d) Describe the three-schema architecture. Why do we need mappings between schema levels? How do different schema definition languages support this architecture?
e) What is the difference between logical data independence and physical data independence?
f) What is the difference between procedural and nonprocedural DMLs?
g) Discuss the different types of user-friendly interfaces and the types of users who typically use each.
h) With what other computer system software does a DBMS interact?
i) Discuss some types of database utilities and tools and their functions.
j) How do you classify the DBMSs ?

Chapter-3. Data Modeling Using the Entity – Relationship (ER) Model.

The term database application refers to a particular database and the associated programs that implement the database queries and updates. Conceptual modeling is a very important phase in designing a successful database application. Entity-Relationship (ER) model is a popular high-level conceptual data model. This model and its variations are frequently used for the conceptual design of the database applications and many database design tools are having its concepts.

Using High-Level Conceptual Data Models for database design : The following figure shows a simplified description of the database design process.

Figure 3.1


Miniworld
REQUIREMENTS COLLECTION AND ANALYSIS
Functional Requirements
Database Requirements
FUNCTIONAL ANALYSIS
CONCEPTUAL DESIGN
High-level Transaction Specification
Conceptual Schema (In a high-level data model)
DBMS-independent
DBMS-specific
LOGICAL DESIGN (DATA MODEL MAPPING)
Logical (Conceptual) Schema (In the data model of a specific DBMS)
APPLICATION PROGRAM DESIGN
PHYSICAL DESIGN
TRANSACTION IMPLEMENTATION
Internal Schema (For the same DBMS)
Application Programs

© The Benjamin/Cummings Publishing Company, Inc. 1994, Elmasri/Navathe, Fundamentals of Database Systems, Second Edition

The first step shown is requirements collection and analysis. During this step, the database designers will interview prospective database users to understand and document their data requirements. These requirements should be specified in as detailed and complete form as possible. Simultaneously, while specifying data requirements, it is useful to specify the known functional requirements of the application. These consists of the user defined operations that includes retrievals and updates.

Once all the requirements have been collected and analyzed, the next step is to create a conceptual schema for the database, using a high-level conceptual data model. This step is called conceptual design. It is a concise description of data requirements of the users and includes detailed description of the entity types, relationships, and constraints. These are expressed using the concepts provided by the high level data model. This model is also considered as a reference to ensure that all users data requirements are met and that the requirements do not conflict. This approach enables database designers to concentrate on specifying properties of the data without being concerned with storage details.

The next step is the actual implementation of the database, using a commercial DBMS. Most of the current commercial DBMSs use an implementation data model such as relational or object-relational database model. This step is called logical design or data model mapping.

The last step is the physical design, in which the internal storage structures, indexes, access paths, and file organizations for the database files are specified. In parallel with these activities, application programs are designed and implemented as database transactions corresponding to the high level transaction specifications.

Entity types, Entity sets, Attributes and Keys :

ER-Model : The ER model is a high-level conceptual data model. This was introduced by Peter Chen in 1976, and is now the most widely used conceptual data model. Much work has been done on the ER model, and various extensions and enhancements have been proposed. ER model describes data as entities, relationships and attributes.

Entity : It is a thing or real time object in the world with an independent existence. An entity may be an object with a physical existence or it may be an object with a conceptual existence. Each entity has attributes (properties). Entity is represented by a rectangle

Example : An employee is the entity and the attributes are empployeeID, Employeename, DOB, Address, designation, salary, ….etc

Entity Set : Group of similar entities are called as an entity set. i.e. entity with same attributes.

Example : Group of employees, group of companies….etc

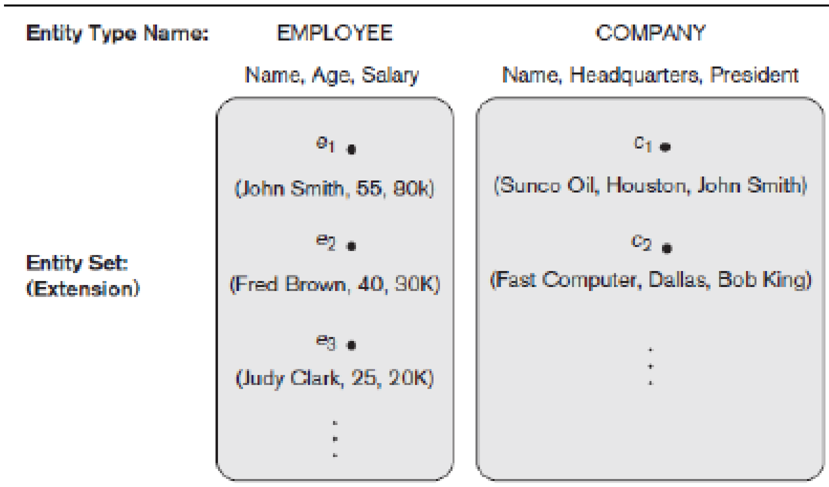
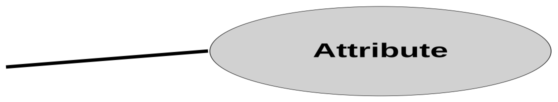
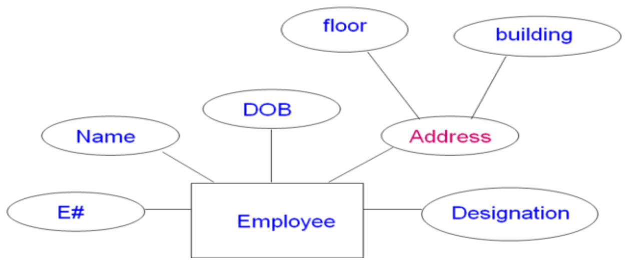


Figure 7.6
Two entity types, EMPLOYEE and COMPANY, and some member entities of each.

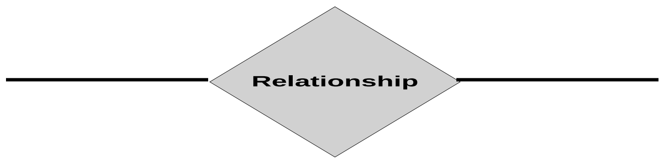
Attributes : It is a property of an entity. It is represented by oval shape with a line.



Example : Attributes of an entity employee are emppployeeID, Employeename, DOB, Address, designation, salary, ….etc

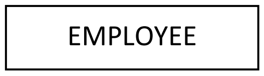


Relationship : It is the type of association between two entities, more entities or with in the entity itself. It is represented by rhombus with associated lines.

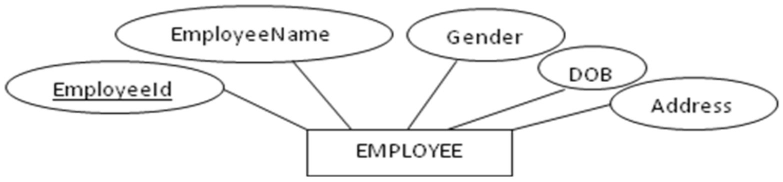


Types of Entities : There are two types of entities. i.e. Strong entity and Weak entity. This depends on the type of the key attributes involved in the entities.

Strong Entity : Strong entities are the ones which are having their own key attributes. By default all the entities are strong. Only at the special occasions, some entities will become weak. Pictorially it is represented by rectangle.

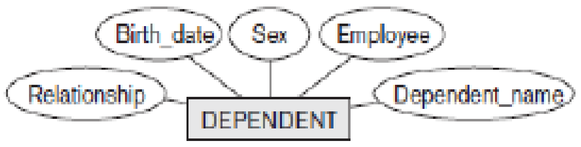


Example : Employee is a strong entity. Because, Employee is a independent entity, and every employee is identified by a key attribute EmployeeId which does not depends on any other entity.



Weak Entity : A weak entity does not having its own key attribute. Even if it is having a key attribute, it depends on the key attribute of other entity. It does not having its individual identity. It is always identified under the shadows of other entity. It is represented by double lined rectangle.

Example : Dependents of an employee.

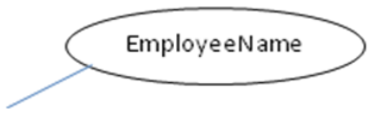


In an organization, any dependent of an employee is always identified by the help of EmployeeId, because, without that nobody knows that he belongs to which employee.

Types of attributes : There are several types of attributes that depends upon the type of the values and number of values they hold. They are

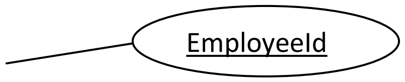
i) Simple or atomic attribute
ii) Key attribute
iii) Composite attribute
iv) Multivalued attribute
v) Derived attribute

i) Simple/atomic attribute : This attribute is having a single value. i.e. its value cannot be divisible further. It is represented by a oval shape with a line. By default every attribute is of simple type.



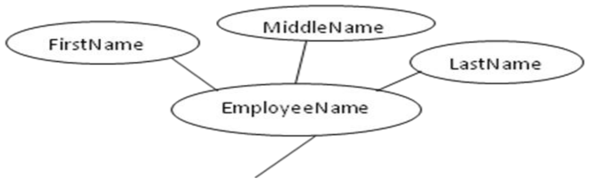
ii) Key attribute : This attribute is having a key value. i.e. by using its value we can able to identify the values of remaining attributes of a given entity. It is always acts as primary identifier. It is represented by a oval shape with an underline.

Example : EmployeeId attribute in an Employee entity is a key attribute.



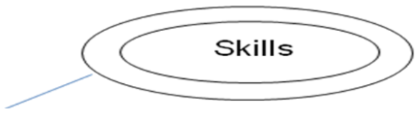
iii) Composite attribute : This attribute is having few more sub attributes in the form of extended components. i.e. Its value can be extended further in the form of separate components. It is represented by a oval shape with an extended components

Example : EmployeeName attribute in an Employee entity can be extended as firstname, middlename, and lastname.



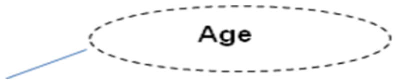
iv) Multivalued attribute : This attribute is having more than one value. i.e. Its attribute is only one, buts its values are more than one. It is represented by a double lined oval shape.

Example : Skills attribute in an employee entity is having more than one value some times. Because an employee is having expertise in various skill sets.



v) Derived attribute : This attribute is having a value which is derived from the value of other attribute. It is represented by a dotted lined oval shape.

Example : Age attribute in an employee entity is deriving its value from the dateofbirth attribute’s value.

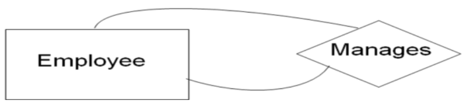


Types of Relationships : There are three relationships. This depends upon the number of entities involved in the relationship. It is also called as degree of relationship.

i) Unary relationship
ii) Binary relationship
iii) Ternary relationship

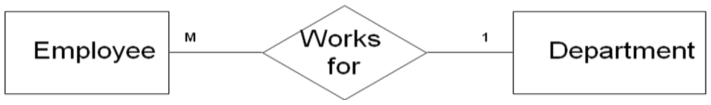
i) Unary relationship : In this relationship, the number of entities involved is only one. The association exists with in the entity itself. It is also called as recursive relationship.

Example : An employee (supervisor) is managing other employees. Here, basically supervisor is an employee. He is managing other employees. Manage is the relationship.



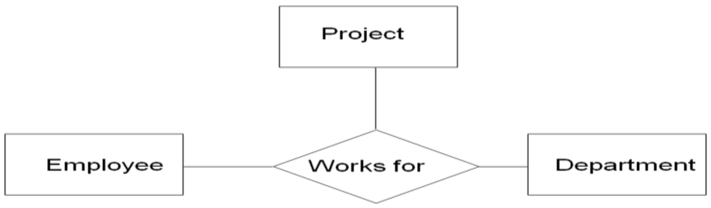
ii) Binary relationship : In this relationship, the number of entities involved is two. The association exists with two entities.

Example : An employee works for the department. Here employee and department are the two entities. Works is the relationship exists between two entities.

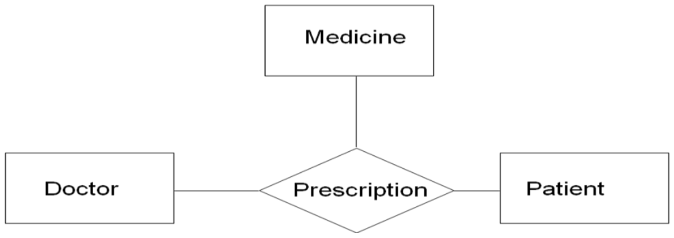


iii) Ternary relationship : In this relationship, the number of entities involved is more than two. The association exists with more than two entities.

Example 1 : An employee works on a particular project in a particular department. Here employee, project and department are the three entities. Works is the relationship exists between three entities.



Example 2 : A doctor is giving prescription to patients on medicines. Here, doctor, patient and medicine are the three entities. Prescribe is the relationship exists between three entities.

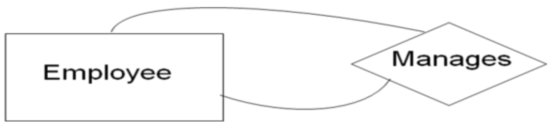


Cardinality ratio : This ratio will show the association among members of the involving entities in a relationship. It also shows number of instances that a relationship exists among the members of the entities. There are three types.

i) One to one (1:1)
ii) One to many or Many to one (1:M / M:1)
iii) Many to many (M:N)

c) Cardinality ratio in unary relationship : In unary relationship, the number of participating entities are only one. Here, we need to verify whether this is possible to apply it or not.

For example an employee is managing other employees.



i) One to one : According to this ratio, every employee is managed by another employee. Rarely, we can see this situation in real time environment.

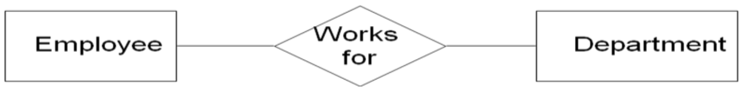
ii) One to many : According to this ratio, a group of employees are managed by one employee (supervisor). Commonly, we can see this situation in real time environment. i.e. one manager with many employees.

Many to one : According to this ratio, many managers are managing a single employee. Rarely, we can see this situation in real time environment. i.e. Many managers with one employee.

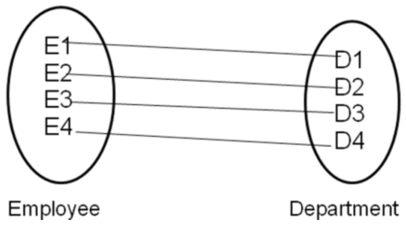
iii) Many to Many : According to this ratio, a manager is managing group of employees and an employee is managed by many managers. Rarely, we can see this situation in real time environment. i.e. one manager with many employees and an employee with many managers.

d) Cardinality ratio in Binary relationship : In binary relationship, the number of participating entities are two. Here, we need to verify whether these ratios can be apply it or not.

For example an employee is working in a department

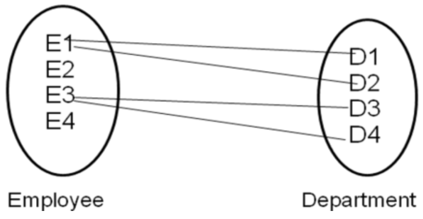


i) One to one : According to this ratio, every employee is working for a separate department. i.e. for each of the employee, there should be a individual department. Rarely, we can see this situation in real time environment.



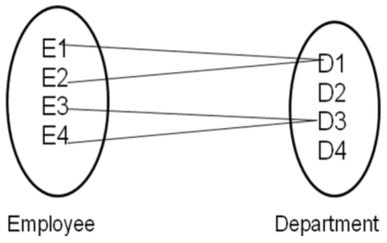
From the above diagram, every employee from the set employee is working individually for a separate department in the set department.

ii) One to many : According to this ratio, an employee is working for more than one department. It is shown in the following diagram. Here, the employee E1 is working for the departments D1 and D2. Similarly, employee E3 is working for the departments D3 and D4. Rarely, we can see this situation in real time environment. i.e. an employee is working for many departments.

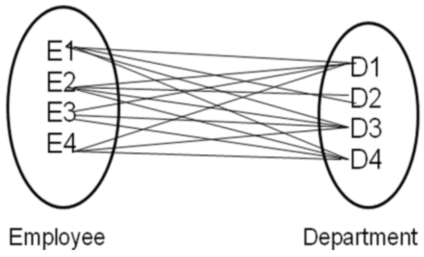


Many to one : According to this ratio, many employees are working for a department. It is shown in the following diagram. Here, the employee E1 and E2 are working for the departments D1. Similarly, employee E3 and E4 are working for the department D3. Commonly, we can see this situation in real time environment. i.e. many employees are working for single department.

Rarely, we can see this situation in real time environment. i.e. Many managers with one employee.

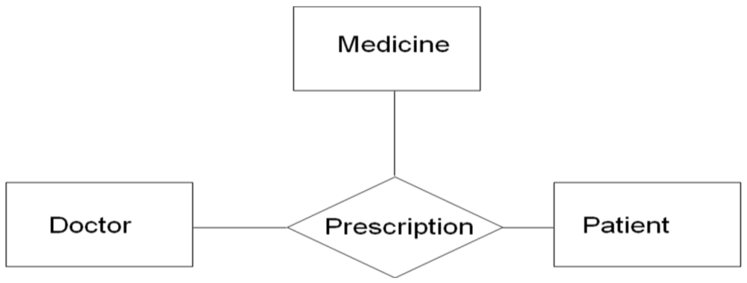


iii) Many to Many : According to this ratio, an employee is working for different departments and every department is having many employees. Rarely, we can see this situation in real time environment. i.e. one employee works for many departments and every department works with many employees.

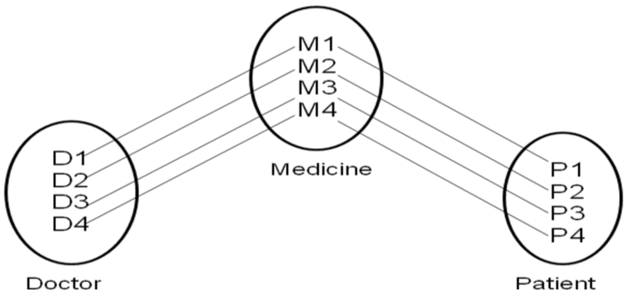


e) Cardinality ratio in Ternary relationship : In ternary relationship, the number of participating entities are more than two. Here, we need to verify whether these ratios can be applicable or not.

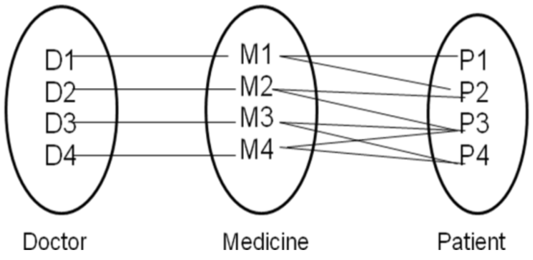
Example : A doctor is giving prescription to patients on medicines. Here, doctor, patient and medicine are the three entities. Prescribe is the relationship exists between three entities.



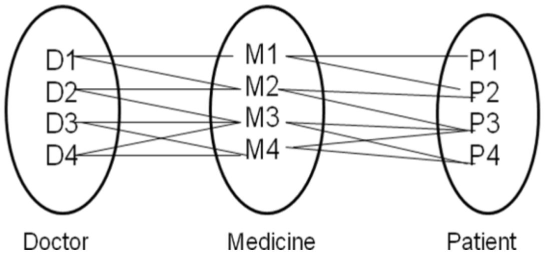
i) One to one : According to this ratio, every doctor is giving prescription on single medicine to a single patient. i.e. for each of the doctor, there will be a single patient and prescription will be given on only one medicine. Rarely, we can see this situation in real time environment.



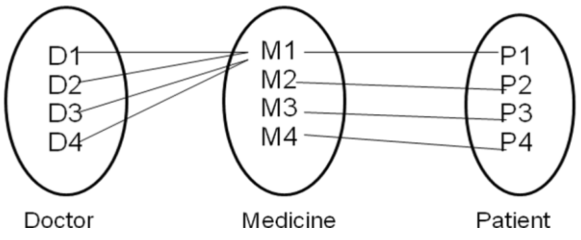
ii) One to many : According to this ratio, every doctor is giving prescription on single medicine to many patients. i.e. for each of the doctor, there will be a single medicine and prescription will be given to many patients. Rarely, we can see this situation in real time environment. This is show in the following diagram.



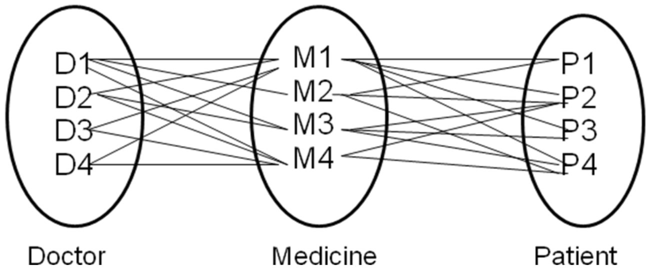
In another case, every doctor is giving prescription on many medicine to many patients. i.e. for each of the doctor, there will be many medicines and prescription will be given to many patients. Commonly, we can see this situation in real time environment. This is show in the following diagram.



Many to one : In this case, many doctors are prescribing a single medicine to a single patient. It is shown in the following diagram. Here, the doctors D1, D2, D3 and D4 are prescribing a medicine M1 to a single patient P1. Rarely, we can see this situation in real time environment.



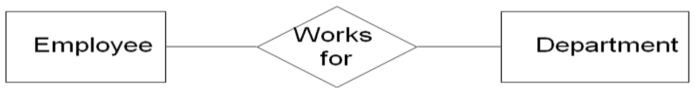
iii) Many to Many : In this case, many doctors are prescribing many medicines to many patients. It is shown in the following diagram. Here, the doctors D1 is prescribing many medicines M1, M2, M3.. to many patients P1, P2 and P3. Similar is the case for other doctors, medicines and patients. Commonly, we can see this situation in real time environment.

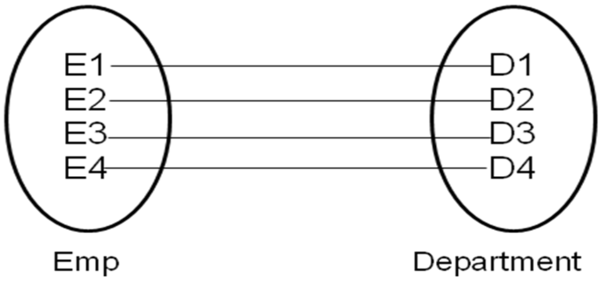


Participation Constraints and Existence Dependencies : The participation constraint is the degree of involvement of the relationship among the members of one entity with members of another participating entity. It specifies whether the existence of an entity depends on its being related o another entity via the relationship type. This constraint specifies the minimum number of relationship instances that each entity can participate in and is sometimes called as the minimum cardinality constraint.

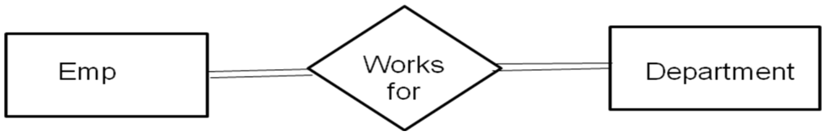
There are two types of participation constraints. i) Total participation and ii) Partial participation.

i) Total participation / Existence dependency : To explain this concept, consider the following two sets (entities), Employee and Department and relationships exists among these two is works for.

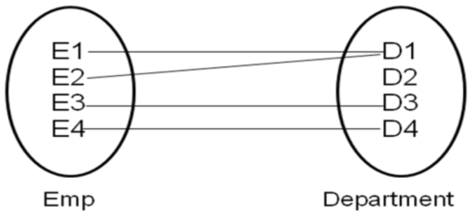




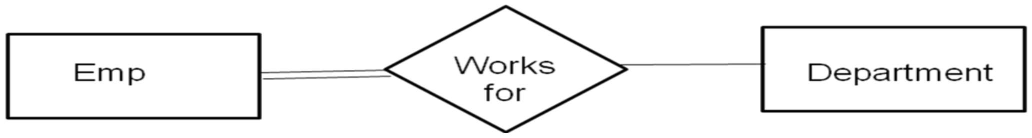
In the above diagram, every employee in the set Emp is working for department. The relationship “works for” is true for all the members of both the entities. i.e. every employee is working in a department and in each of the department, there is an employee is working. This is called total or full participation. It is also called as existence dependency. It is represented by thick line or double lines which as shown below.



ii) Partial participation : Consider the following diagram with two entities Emp and Department.

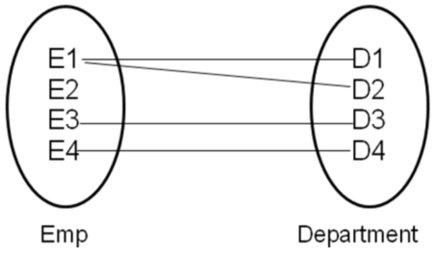


Here, the relationship "works for" is not true with every members of the department entity and it is true for all the members of emp entity. This is called as partial participation. It is represented by a thin line or single line which is shown in the following figure.

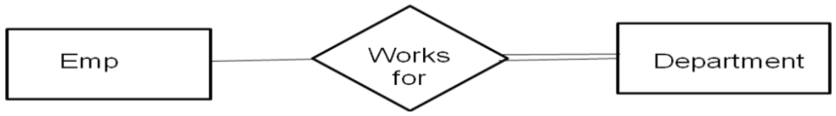


Here, the entity Emp is having total participation with relationship "works for", since all its members are working in different departments. But, the department entity is partial, because, in one of its departments 'D2', no employee is working which results in false with relationship "works for". Therefore, emp is total and department is partial.

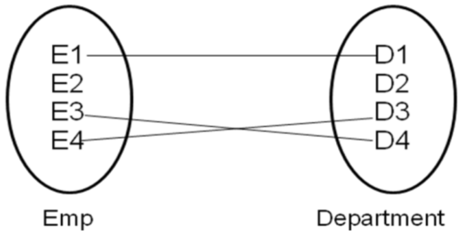
Consider the following case:



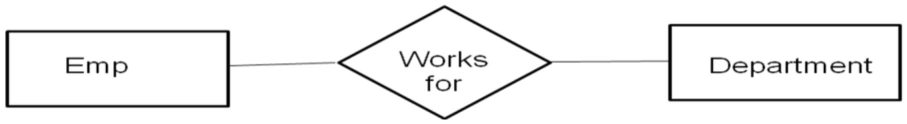
Here, the entity department is having total participation with relationship "works for", since all its departments are working by some employees. But, the emp entity is partial, because, one of its employees 'E2' is not working for any department which results in false with relationship "works for". Therefore, department is total and emp is partial. Pictorially it is represented as follows.



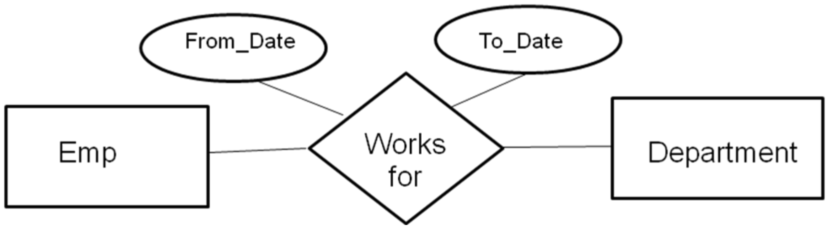
Consider the following case:



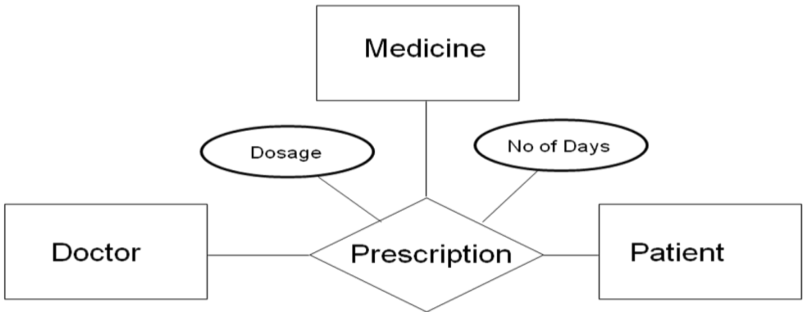
Here, the entity emp is having partial participation with relationship "works for", since one of its employees 'E2' is not working for any of the departments. Similarly, the department entity is also partial, because, in one of its departments 'D2', no employee is working which results in false with relationship "works for". Therefore, emp is partial and department is also partial. Pictorially it is represented as follows.



Relationship attributes : These are the attributes defined on the relationship. Consider the following ER diagram.



Here, the attributes “From_date” and “To_date” are of relationship type attributes, because they are defined on the relationship “works for”. These attributes best defines the relationship rather than any individual entity. i.e. an employee is working from so and so starting date to so and so ending date for a particular department. Following is the one more example.

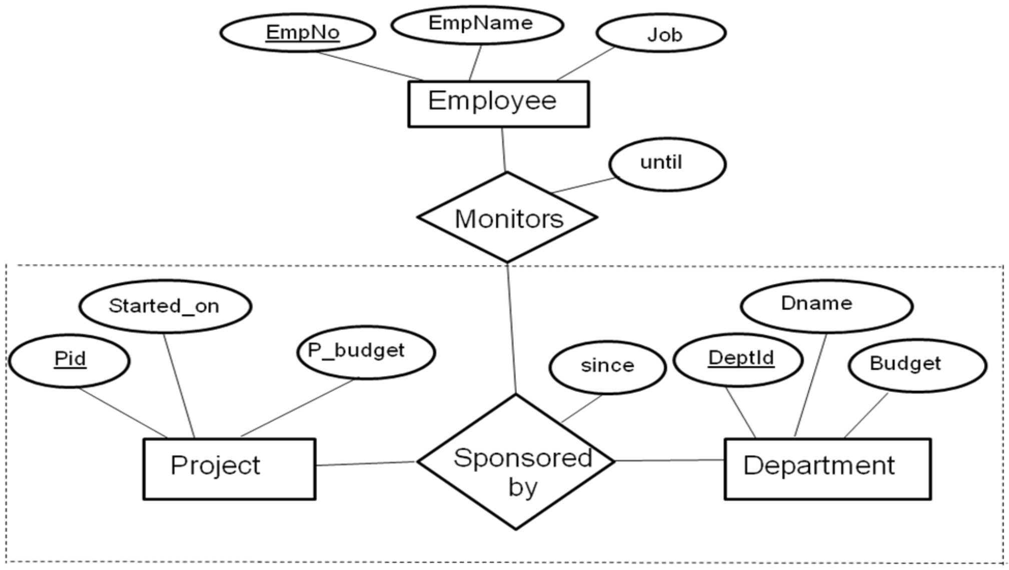


Here, the attributes “Dosage” and “No of Days” are of relationship type attributes, because they are defined on the relationship “Prescription”. i.e. a doctor is prescribing some dosage of medicine to patient for some number of days.

Aggregation : It is the relationship exists with the another relationship. A relationship is an association between entity sets. Sometimes, we have to model a relationship between collection of entities and relationships. Aggregation is meant to represent a relationship between a whole object and its component parts. It is used when we have to model a relationship involving entity sets and relationship sets. It allows us to treat a relationship set as an entity set for the purpose of participation in other relationships.

Example: A project is sponsored by a department. Here, project is an entity set and each project is sponsored by one or more departments. Aggregation allows relationship set participates in another relationship set. A department that sponsors project might assign employees to monitor the relationship.

Intuitively, monitors should be a relationship set that associates a sponsored by relationship with an employee entity rather than project. Here, we have defined relationships to associate two or more entities.



When we use aggregation ?

We use it, when we need to express a relationship among relationships.

Extended Entity Relationship features : These are the new features added to the previous ER model to accommodate the present trend requirements of industries such as manufacturing, telecom and geographic information systems. These types of databases have more complex requirements than the more traditional applications. This led to the development of additional semantic data modeling concepts that were incorporated into conceptual data models. Various semantic data models have been proposed in the literature. Many of these concepts were also developed independently in related areas of computer science such as knowledge representation area of artificial intelligence and the object modeling areas in software engineering.

Superclasses, Subclasses and Inheritance : These are the concepts of extended ER model's specialization and generalization.

Superclass: It is the entity type with common features which can be extended as sub classes.

Subclass: It is the subgroup in a entity type with some specialized features along with common features of its superclass. In many cases an entity type has numerous sub groupings of its entities that are meaningful and need to be represented explicitly because of their significance to the database application.

Example : In an entity type of EMPLOYEE, we have many group of employees in the form of their designations such as secretary, engineer, manager, technician, operator, accountant and so on. The set of entities in each of the latter groupings is a subset of the entities that belong to the EMPLOYEE entity set. We call each of these subgroupings as subclass of the EMPLOYEE entity type and the EMPLOYEE entity type is called the superclass for each of these subclasses.

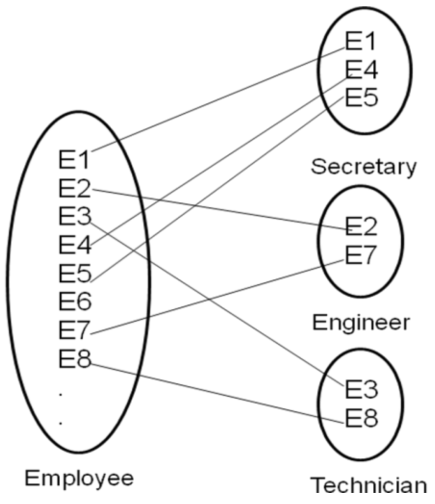
We call the relationship between a superclass and any one of its subclasses a superclass/subclass or simply class/subclass relationship.

Inheritance : The entity in the subclass represents the same features from the superclass, as it possess values for its specific attributes as well as values of its attributes as a member of the superclass. We say that an entity that is a member of a subclass inherits all the attributes of the entity as a member of the superclass. The entity also inherits all the relationships in which the superclass participates.

Specialization and Generalization :

Specialization : It is the process of defining a set of subclasses of an entity type. The set of subclasses forms a specialization and it is defined on the basis of some distinguishing characteristics of the entities in the superclass.

Example-1 : The set of subclasses { secretary, engineer, technician} is a specialization of the superclass EMPLOYEE that distinguishes among employee entities based on the job type of each employee entity. Following diagram shows specialization among employee entity.



Here, a few entity instances that belong to subclasses of the { secretary, engineer, technician} specialization. Again notice that an entity that belongs to a subclass represents the same real world entity connected to it in the EMPLOYEE superclass.

The specialization allows us to the do the following :

i) Define a set of subclasses of an entity type.

ii)Establish additional specific attributes with each subclass.

iii) Establish additional specific relationship types between each subclass and other entity types or other subclasses.

The set of subclasses { secretary, engineer, technician} is a specialization of the superclass EMPLOYEE that distinguishes among employee entities based on the job type of each employee entity. Following diagram shows specialization among employee entity.

Generalization : It is the reverse process of specialization in which we suppress the differences among several entity types, identify their common features and generalize them into a single superclass of which the original entity types are special subclasses. It can be viewed functionally the inverse of the specialization process.

Example : Consider the entity types CAR and TRUCK which are shown in the following diagram-A. They have several common attributes, they can be generalized into the entity type VEHICLE. Both CAR and TRUCK are now subclasses of the generalized superclass VEHICLE.

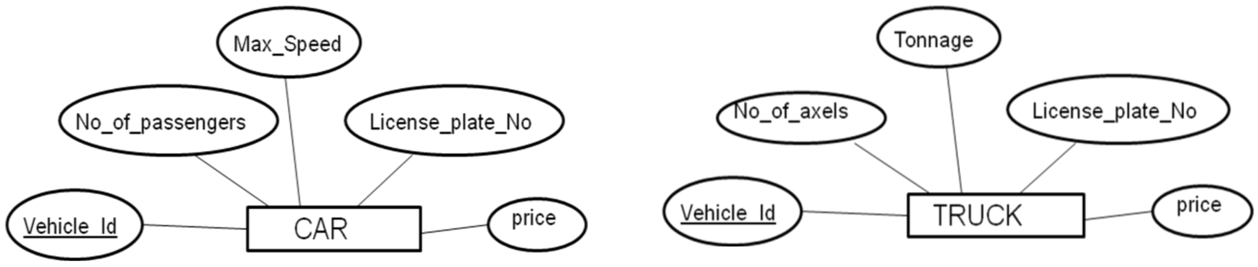


Diagram-A

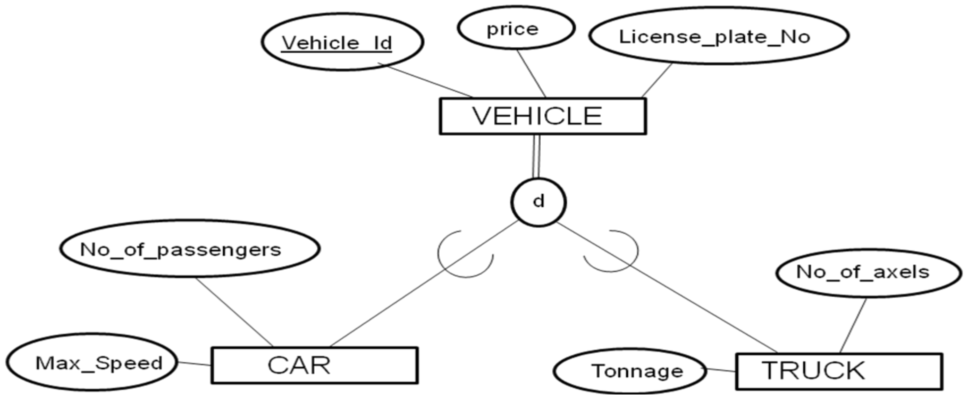
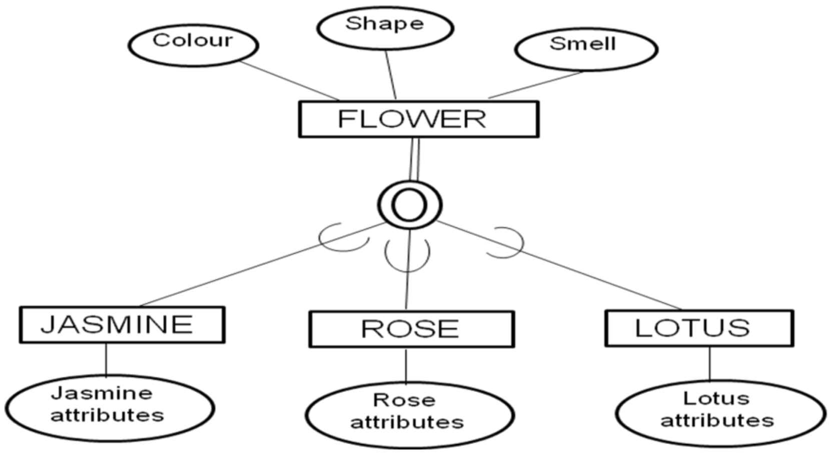


Diagram-B

Here, we are using the term 'generalization' to refer the process of defining a generalized entity types. The set {CAR, TRUCK} is the specialization of VEHICLE rather than viewing VEHICLE as a generalization of CAR and TRUCK. An arrow pointing to the generalized superclass represents a generalization, whereas arrows pointing to the specialized subclass represent a specialization.

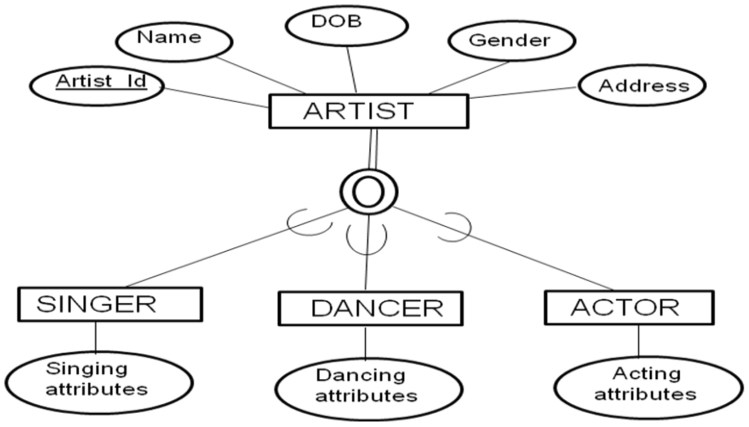
Example-2 : Sometimes different entity types are actually specializations of a more general entity type:

Basically, rose, jasmine, lotus are all flowers. Here, some attributes are common to all, others are specific to one entity type which are represented by generalization hierarchy. Subtypes may be disjoint or overlapping type. The attributes that are common belong to the supertype and those that are specific belong to the particular subtype.



This depicts a disjoint subset. For example, a particular flower that is jasmine can only belong to the entity set jasmine and it cannot belongs to set rose.

Overlapping Subtypes :



The above diagram depicts a overlapping subtype relationship.

Example : An artist is basically a human being and he is always identified by the common attributes like artist_id, name, dob, gender and an address. An artist is considered here as a entity of supertype, but it can be extended as subtypes {singer, dancer, actor}. Singer is a subtype under artist and he is having singing attributes along with the common attributes of artist. Similarly, the case of dancer and actor.

Case Study : Here, it is given with a problem statement with their requirements. We need to read the case study carefully and then do analysis with following steps.

i) Identify all possible existing entities

ii) Identify all given attributes for all the identified entities.

iii) Identify the key attributes for every entity

iv) Identify strong and weak entities

v) Identify all possible relationships among entities based on the given case study

vi) Identity the type of the cardinality ratios on all the identified relationships

vii) Identify the type of participation on entities with relationships

viii) After all these steps, draw an ER diagram which represents all the requirements of the case study.

i) Banking scenario :

- A Bank is identified by name, Id number, location of the main office and an address
- A bank has many branches
- Each branch is identified by branchId, branch name, and an address
- A branch has many customers who hold the account
- A customer is identified by customerId, customer_name and an address
- Account is held by Customer and a customer may have single account and many accounts (SB account, Joint Account, Business Account)
- Each Account is identified by account_no, account_type, balance_amount
- A bank offers Loans to customers through branches
- Loans are identified by loanId, loanType (House, Car, Business, Personal) , loan_amount

Draw an ER diagram to represent this application

Solution :

Step-1 : Identify all Entities :

Bank, Branch, Customer, Account, Loan

Step-2 : Identify all given attributes for all the identified entities.

Bank { Bank_Id, Bank_name, Bank_Location}

Branch { Branch_Id, Branch_Name, Branch_Address}

Customer { Customer_Id, Customer_Name, Customer_Address}

Account { Account_No, Account_Type, Balance_Amount}

Loan { Loan_Id, Loan_Type, Loan_Amount}

Step-3: Identify the key attributes for every entity

Bank { **<u>Bank_Id</u>**}, Branch { **<u>Branch_Id</u>**}, Customer { **<u>Customer_Id</u>**},

Account { **<u>Account_No</u>**}, Loan { **<u>Loan_Id</u>**}

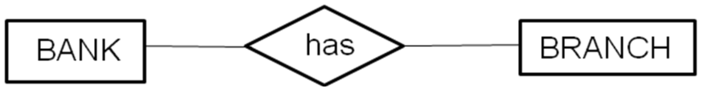
Step-4: Identify strong and weak entities

Strong entities : Bank, Customer, Account, Loan

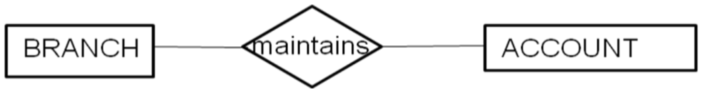
Weak entities : Branch

Step-5 : Identify Relationships among all entities based on the given case study :

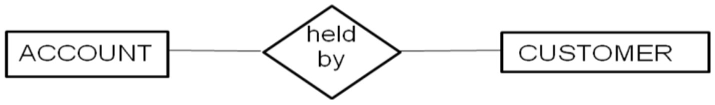
Bank has branches



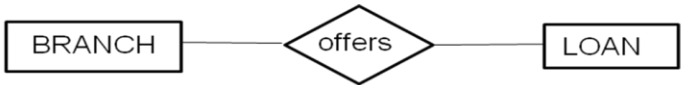
Branch maintains account



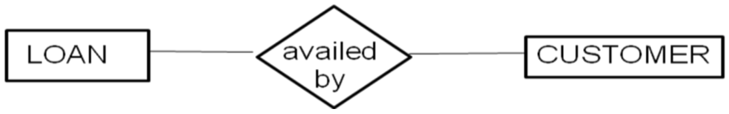
Account is held by Customer



Branch Offers Loans

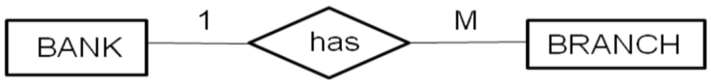


Customers can avail loans

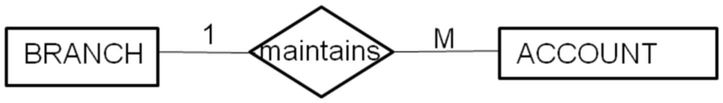


Step-6 : Identify cardinality ratios among the entities with their relationships

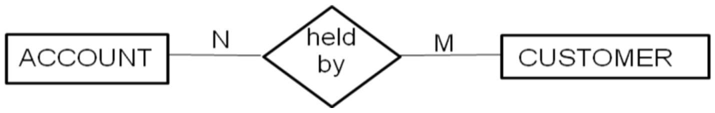
Bank has many branches → 1 : M



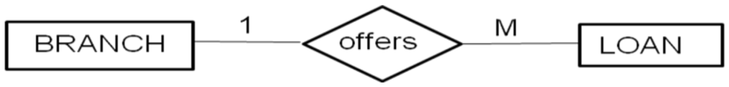
Branch maintains Accounts → 1 : M



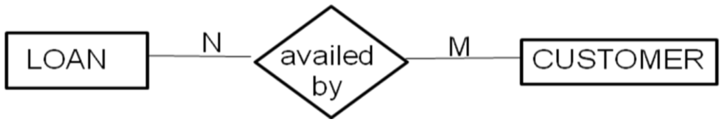
Account is held by Customer → N : M



Branch Offers Loans → 1 : N

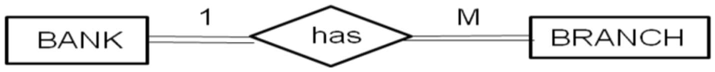


Customers can avail loans → N : M

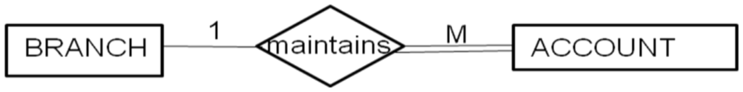


Step - 7 : Identify type of participation among entities with their relationships :

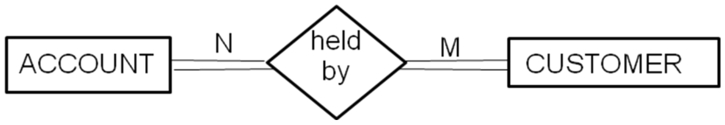
Bank has many branches → Bank : Total Branches : Total



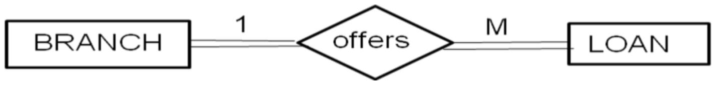
Branch maintains Accounts → Branch : Partial Account : Total



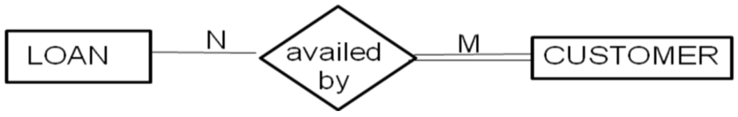
Account is held by Customer → Account : Total Customer : Total



Branch Offers Loans → Branch : Total Loans : Total

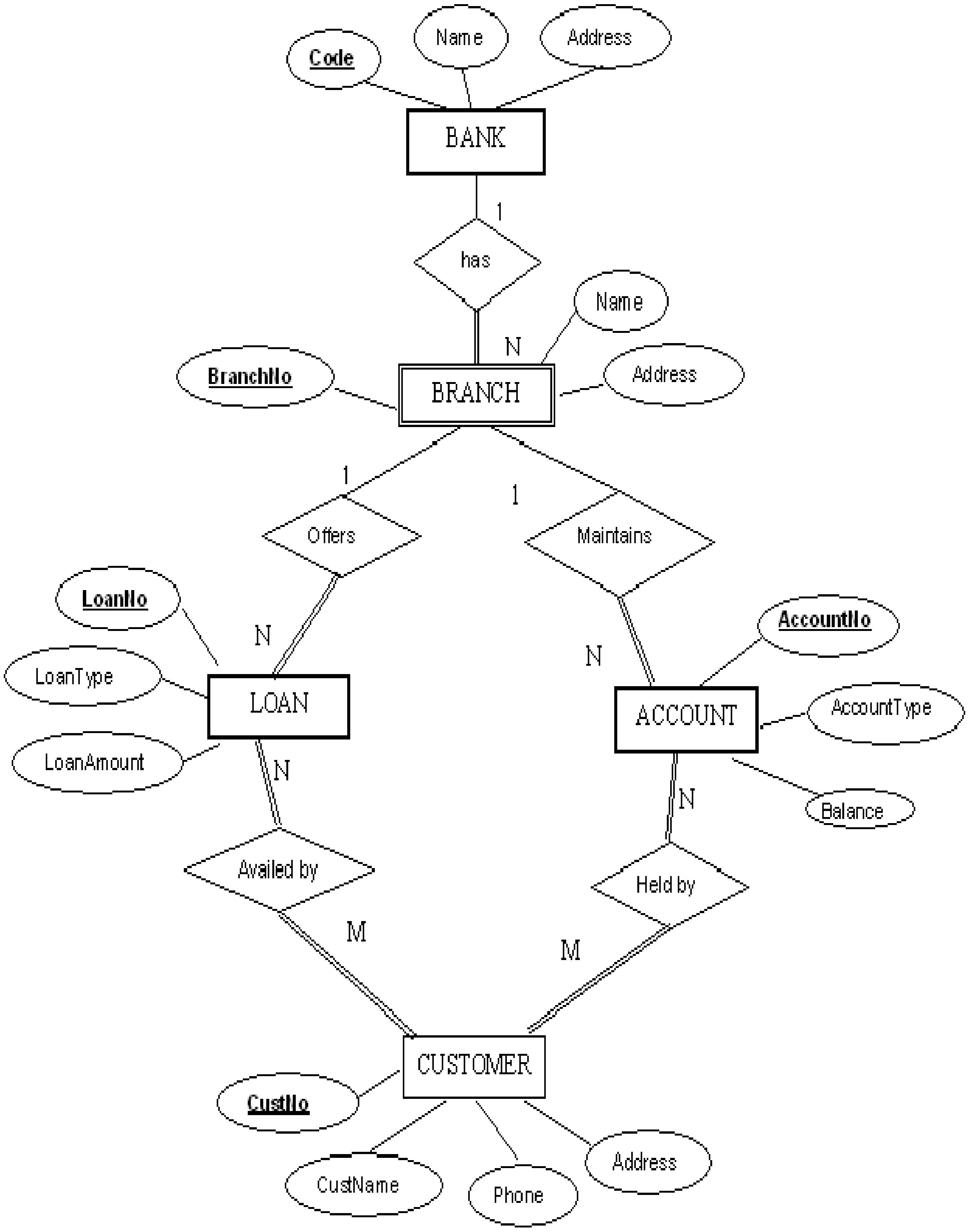


Customers can avail loans → Customers : Partial Loans : Total



Step-8 : Write an ER diagram :

Note : Here, we have given identification of relationships, cardinality ratios and participation types in separate steps (i.e. step-5,step-6,step-7), But we can also write these steps in one single step.


Code
Name
Address
BANK
1
has
N
Name
BranchNo
BRANCH
Address
1
1
Offers
Maintains
LoanNo
N
N
AccountNo
LoanType
LOAN
ACCOUNT
AccountType
LoanAmount
N
N
Balance
Availed by
Held by
M
M
CUSTOMER
CustNo
CustName
Phone
Address

ii) A Company Scenario :

- Company
  - Organized into departments, Each department has a name, number and manager who manages the department. The company keeps track of the date that employee managing the department. A department may have a several locations.
- Department
  - A department controls a number of projects each of which has a unique name, number and a single Location.
- Employee
  - Name, Age, Gender, BirthDate, SSN, Address, Salary. An employee is assigned to one department, may work on several projects which are not controlled by the department. Track of the number of hours per week is also controlled.
- Dependant
  - Keep track of the dependents of each employee for insurance policies : We keep each dependant first name, gender, date_of_birth and relationship to the employee.

Draw an ER diagram to represent this application

Solution :

Step-1 : Identify all Entities :

Employee, Department, Project and Dependent

Step-2 : Identify all given attributes for all the identified entities.

Employee { SSN, Name (Fname, Lname), Bdate, Sex, Salary, Address }

Department { DeptNo, Dept_Name, Location}

Project { Project_No, Project_Name, Project_location}

Dependent { Name, Bdate, Sex, Relationship }

Step-3: Identify the key attributes for every entity

Employee { <u>SSN}, </u> Department { <u>DeptNo</u>}, Project { <u>Project_No</u>}, Dependent { <u>Name</u>}

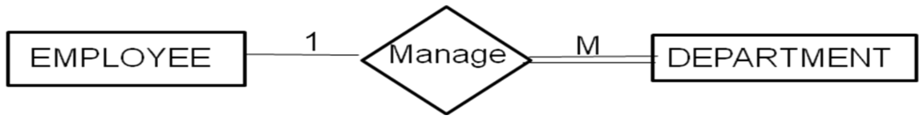
Step-4: Identify strong and weak entities

Strong entities : Employee, Department, Project

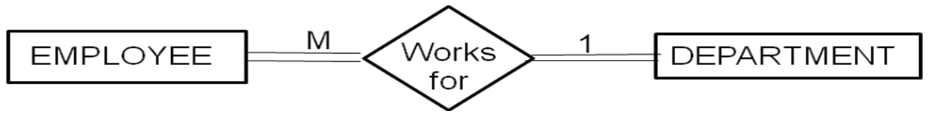
Weak entities : Dependent

Step-5 : Identify Relationships, Cardinality ratios and Participation types among all entities :

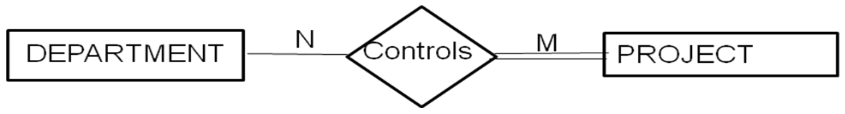
Employee manages the department



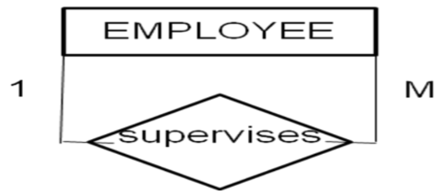
Employee works for department



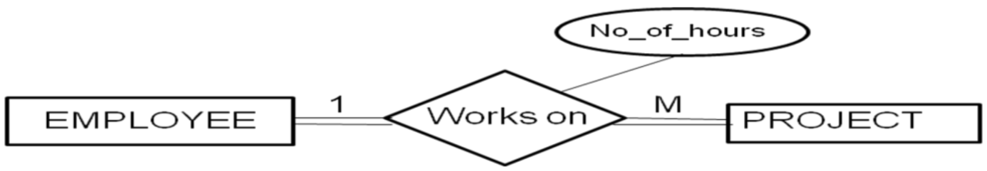
Department controls project



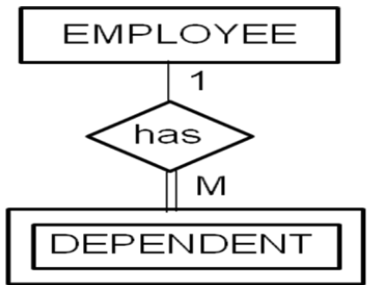
Employee supervises his subordinates



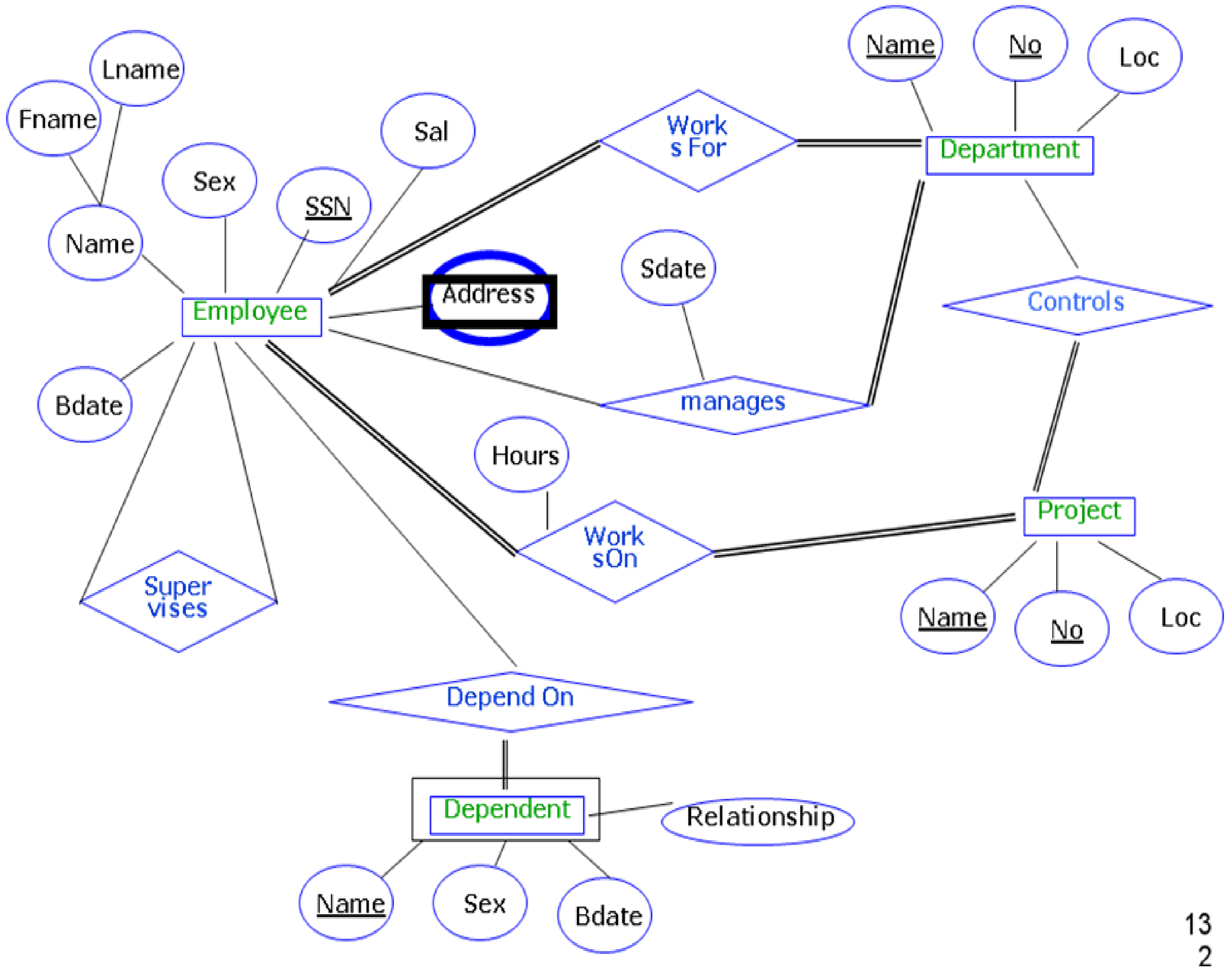
Employee works on project



Employee has dependents



Step-6 : Write an ER diagram :



Converting entity types to table :

i) Each entity type becomes a table.

ii) Each single-valued attribute becomes a column.

iii) Derived attributes are ignored.

iv) Composite attributes are represented by its equivalent parts.(i.e. each extended attribute will become a individual column)

v) Multi-valued attributes are represented by a separate table.

vi) The key attribute of the entity type will becomes the primary key of the table.

Entity – Relationship examples :

i) Employee has many skills

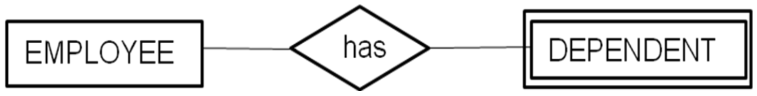
Employee { Employee#, EmpName, DoorNo, Street, City, Pincode, Date_of_join)

AND

Emp_skillset { Employee#, Skillset)

In the above table Employee, Employee# is the primary key and same key acts as a foreign key to the child table Emp_skillset.

ii) Employee has dependents



Employee { Employee#, EmpName, Date_of_join, Designation)

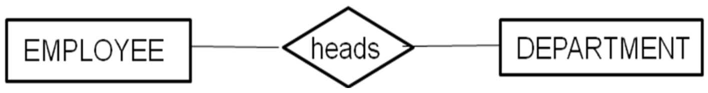
AND

Dependent { Employee#, DependentId, DependentName, Gender, DOB, Age)

Converting Relationships :

Binary relationship 1:1 :

i) Employee is heading the department



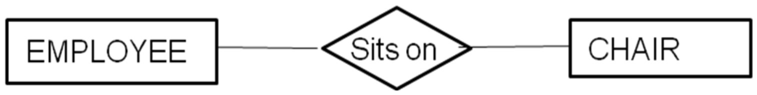
Employee { Employee#, EmpName, Date_of_join, Designation)

AND

Department { Department#, DepartmentName, Location, Head)

In the above table Employee, Employee# is the primary key and same key acts as a foreign key to the child table department in the name head. In department table Department# acts as a primary key.

ii) Employee sits on chair



Employee { Employee#, EmpName, Date_of_join, Designation)

Chair { Chair#, Model, Location, Used_by)

In the above table Employee, Employee# is the primary key and same key acts as a foreign key to the child table Chair in the name Used_by. In Chair table Chair# acts as a primary key.

It can also be designed in the following way.

Employee { Employee#, EmpName, Date_of_join, Sits_on)

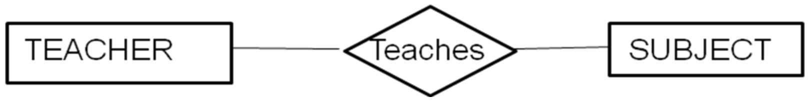
AND

Chair { Chair#, Model, Location)

In the above table Chair, Chair# is the primary key and same key acts as a foreign key to the child table Employee in the name Sits_on. In Employee table Employee# acts as a primary key.

Binary relationship 1: N

Teacher teaches a subject:



Teacher { Teacher#, TeacherName, ContactNo, EmailId, Branch)

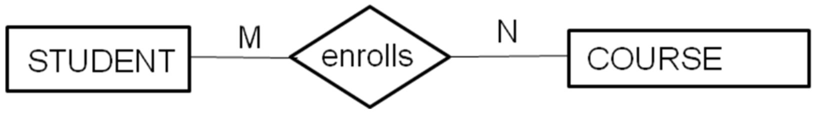
AND

Subject { Subject#, SubjectName, Duration, Taught_by)

In the above table Teacher, Teacher# is the primary key and same key acts as a foreign key to the child table Subject in the name Taught_by. In Subject table Subject# acts as a primary key.

Binary relationship M : N

Student enrolls course:



Student { Student#, StudentName, DOB..)   Course { Course#, CourseName, Duration)

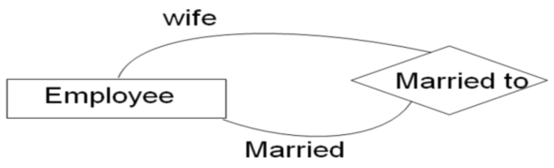
AND

Enrolls { Student#, Course#,)

In the above table Student, Student# is the primary key and same key acts as a foreign key to the child table Enrolls. In Course table, Course# is the primary key and same key acts as a foreign key to the child table Enrolls.

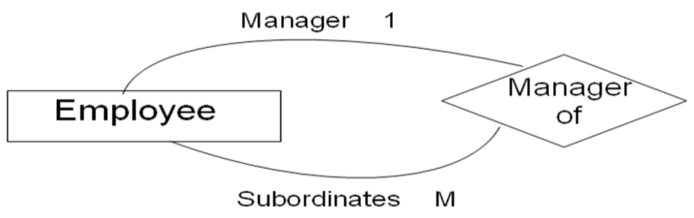
Unary relationship : 1 : 1

Employee married with another employee (Assuming both husband and wife are working in the same company)



Employee { Employee#, EmployeeName, DOJ, Designation, Salary, Spouse)

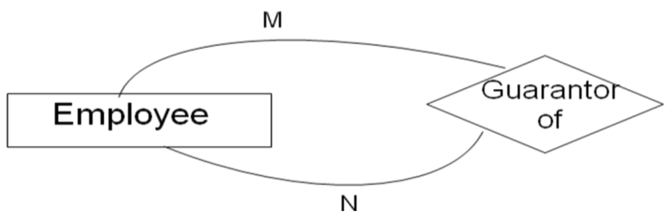
Unary 1 : M :-



Employee { Employee#, EmployeeName, DOJ, Designation, Salary, Manager}

In the above table Employee, Employee# is the primary key and it acts as a foreign key in the same table in the name of Manager. This foreign key is also called as self referential key. It is an example for unary relationship. This shows that, one primary key acts as a foreign key in same table itself. Because, both manager and employee are employees working in the same organization.

Unary M : N :-



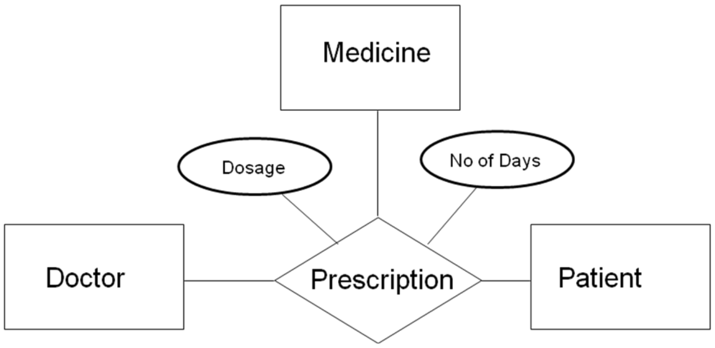
Employee { Employee#, EmployeeName, DOJ, Designation, Salary}

AND

Guaranty { Guarantor, Beneficiary)

In the above table Employee, Employee# is the primary key and same key acts as a foreign key to the child table Guaranty in the name of Guarantor and Beneficiary. This shows that, one primary key acts as many foreign keys in one table itself. Because, both guarantor and beneficiary are employees working in the same organization.

Ternary Relationship :-



Doctor { Doctor#, DoctorName, Specialization, PlaceofWork, ContactNo}

Medicine { Medicine#, MedicineName, MedicineType, Manufactureddate, ExpiryDate}

Patient { Patient#, PatientName, SufferingDesease, Age, Gender, Address, ContactNo}

Prescription { Doctor#, Medicine#, Patient#, From_Date, To_date, Dosage, No_Times_a_day }

In the above table Doctor, Doctor# is the primary key and same key acts as a foreign key to the child table Prescription. In the table Medicine, Medicine# is the primary key and same key acts as a foreign key to the child table Prescription. In the table Patient, Patient# is the primary key and same key acts as a foreign key to the child table Prescription. In the child table prescription, there are three foreign keys, doctor#, medicine#, patient# and other attributes. This means that a doctor is prescribing a dosage of medicine to the patient about number of times a day from starting date to ending date.

The Relational Data Model and Relational Database Constraints :

Relational Model concepts : The relational model represents the database as a collection of relations. Informally, each relation resembles a table of values or file of records. Each row in the table represents collection of related data values and a fact that typically corresponds to a real-world entity or relationship. The table name and column names are used to help to interpret the meaning of the values in each row.

In the formal relational model terminology, a row is called an attribute, and the table is called a relation. The data type describing the types of values that can appear in each column is represented by a domain of possible values.

Domains, Attributes, Tuples and Relations :

Relation : It is a table. The way storing the data in the form rows and columns.

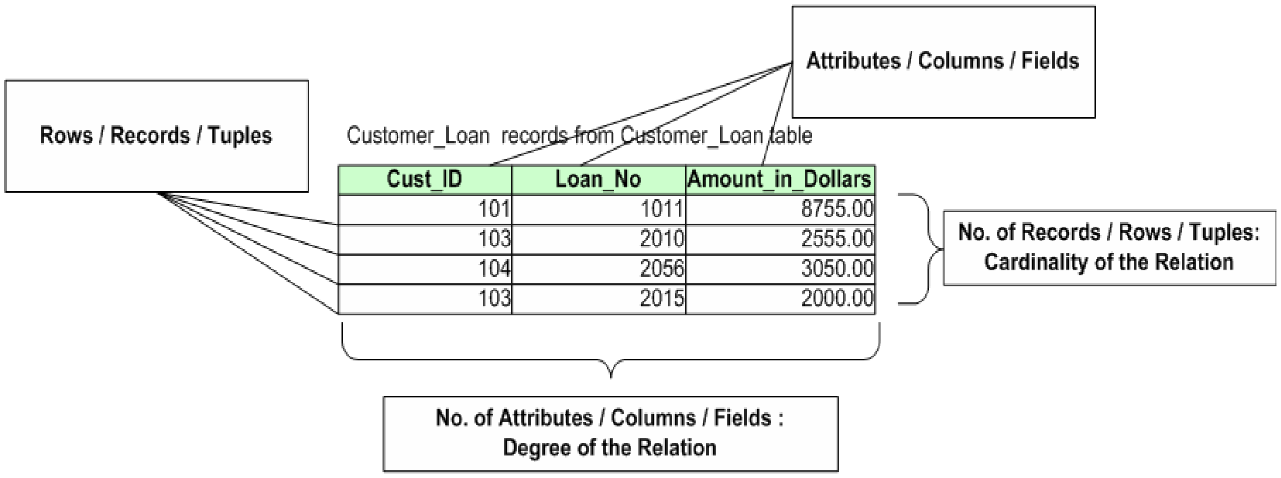
Tuple : Each row in a table is called tuple. It is also called record.

Field / Column header : Attribute of a table is called a column/ field.

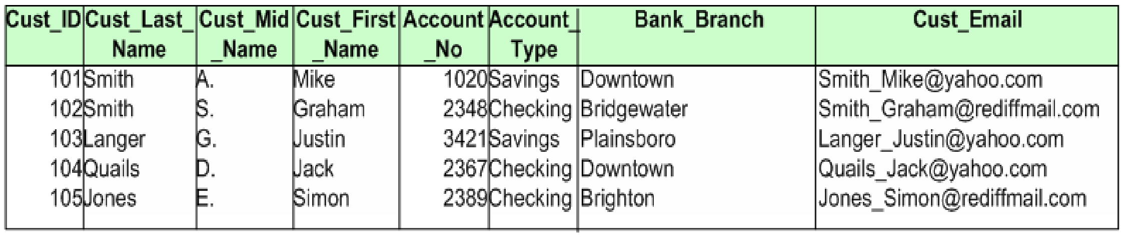
Domain : The data type describing the types of values that can appear in each column is called as domain. In simple sense, range / set of values for a given column.

Degree of a relation : Number of columns present in a relation is called a degree of relation. i.e. number of columns in a table.

Cardinality of relation : Number of tuples in a relation is called a cardinality of a relation. i.e. number of records / rows in a table.


Attributes / Columns / Fields
Rows / Records / Tuples
Customer_Loan records from Customer_Loan table
No. of Records / Rows / Tuples: Cardinality of the Relation
No. of Attributes / Columns / Fields : Degree of the Relation

Cust_ID	Loan_No	Amount_in_Dollars
101	1011	8755.00
103	2010	2555.00
104	2056	3050.00
103	2015	2000.00



Cust_ID	Cust_Last_Name	Cust_Mid_Name	Cust_First_Name	Account_No	Account_Type	Bank_Branch	Cust_Email
101	Smith	A.	Mike	1020	Savings	Downtown	Smith_Mike@yahoo.com
102	Smith	S.	Graham	2348	Checking	Bridgewater	Smith_Graham@rediffmail.com
103	Langer	G.	Justin	3421	Savings	Plainsboro	Langer_Justin@yahoo.com
104	Quails	D.	Jack	2367	Checking	Downtown	Quails_Jack@yahoo.com
105	Jones	E.	Simon	2389	Checking	Brighton	Jones_Simon@rediffmail.com

records from Customer_Details table

- A relation schema 'R', denoted by $R(A_1, A_2, - - - , A_n)$, is made up of a relation name 'R' and a list of attributes $A_1, A_2, - - - - , A_n$.
- Each attribute Ai is the name of a role played by some domain 'D' in the relation schema 'R'. 'D' is called the domain of A_i and is denoted by $dom(A_i)$.
- A relation schema is used to describe a relation ; 'R' is called the name of this relation.
- The degree (or arity) of a relation is the number of attributes 'n' of its relation schema.
- A relation is a set of n-tuples $r = \{t_1, t_2, - - - -, t_n\}$. Each n-tuple 't' is an ordered list of n values $t = <v_1, v_2, - - - - - V_n>$ where each value Vi, $1 <= i <= n$ is an element of $dom(A_i)$ or is a special NULL value.

Example : A relation schema for a relation of degree Seven, which describes university students, STUDENT { Name, Ssn, Home_Phone, Address, Office_phone, Age, Gpa}

For this relation, STUDENT is the name of the relation, with seven attributes (i.e. Name, Ssn, Home_Phone, Address, Office_phone, Age, Gpa)

Characteristics of a Relations :

Ordering of tuples in a relation : A relation is defined as a set of tuples. Mathematically, elements of a set have no order among them, hence, tuples in a relation do not have any particular order.

Ordering of values within a tuple and an alternative definition of a relation (i.e. ordering of attribute values) : The order of attributes and their values is not important as long as the correspondence between attributes and values is maintained. When a relation is implemented as a file, the attributes are physically ordered as fields with in a record. We generally use the first definition of a relation, where the attributes and the values within tuples are ordered, because, it simplifies much of the notation.

Values and NULLs in the tuples : Each value in a tuple is an atomic value; that is it is not divisible into components within the framework of the basic relational model. Hence, composite and multivalued attributes are not allowed. Multivalued attributes must be represented by separate relations and composite attributes are represented only by their component attributes in the basic relational model.

There are several meanings for NULL values, such as unknown value, empty value, undefined value and value exists but is not available…etc. NULL values arise due to several reasons at the time of data entry into the relations.

Interpretation of a relation : The relation schema can be interpreted as a declaration or a type of assertion. Each tuple in a relation can be interpreted as a fact or a particular instance of the assertion. Notice that some relations may represent facts about entities, whereas other relations may represent facts about the relationships. Hence, the relational model represents facts about both entities and relationships uniformly as relations.

In simple sense :

i) There are no duplicate rows / tuples

ii) Tuples are unordered, top to bottom

iii) Attributes are unordered, left to right

iv) All attribute values are atomic in nature

v) Relational databases do not allow repeating groups

Relational Model Constraints :

Constraints are nothing but restrictions applied on relations in a database. Generally, there are many constraints on the actual values in a database state. These constraints are derived from the rules in the miniworld that the database represents. Constraints on database can generally be divided into three main categories :

i) Constraints that are inherent in the data model called as inherent model-based constraints or implicit constraints
ii) Constraints that can be directly expressed in schemas of the data model called as schema-based constraints or explicit constraints
iii) Constraints that cannot be directly expressed in schemas of the data model, and hence must be expressed and enforced by the application programs called as application-based constraints or semantic constraints or business rules.

In general, constraints are classified as

- Entity integrity constraints or Key constraints
- Domain constraints
- Referential integrity constraints

i) Entity – Integrity constraints or Key constraints : A relation is defined as a set of tuples. By the definition, all the elements of a set are distinct, hence, all the tuples in a relation must also be distinct. This means that no two tuples can have the same combination of values for all their attributes. This is possible with the inclusion of key constraints in a relation.

Examples : Primary key, Candidate key and Super key.

Primary Key **:** An attribute or set of attributes whose values uniquely identify each entity in an entity set is called a key for that entity set. We also call it as a primary key. It can be applied on one column or multiple columns in a relation. For a given table / relation, we can include maximum of only one primary key. By default, primary key does not accept NULL value.

NOTE : Inclusion of a primary key to every relation / table in a database is not mandatory. It depends on our requirement.

Super Key : If we add additional attributes to a key, the resulting combination would still uniquely identify an instance of the entity set. Such augmented keys are called as super keys.

Candidate Keys : There may be two or more attributes or combinations of attributes that uniquely identify an instance of an entity set. These attributes or combinations of attributes are called candidate keys.

– Primary Key
– Alternate Key

Example : In an employee relation, ContactNo and EmailID are the candidate keys, because, their values can also be considered to identify every record / tuple uniquely.

Secondary Keys : A secondary key is an attribute or combination of attributes that may not be a candidate key but that classifies the entity set on a particular characteristic.
A case in point is the entity set EMPLOYEE having the attribute department, which identifies by its value all instances EMPLOYEE who belong to a given department.

Simple Key :
Any key consisting of a single attribute is called a **simple key**

Composite Key :
Any key that consisting of a combination of attributes is called a **composite key**.

ii) Domain Constraints : Domain constraints specify that within each tuple, the value of each attribute Ai must be atomic value from the domain dom(A). These constraints mainly concentrates the type of the data values they accepts to the corresponding attributes / columns. We have three constraints with this type. NULL, NOT NULL and CHECK

NULL constraint : It is the constraint which facilitate the given column to accept the NULL value. No explicit create statement is required to include this constraint on any column. Because, by default every column is of NULL type except key attribute/column.

NOT NULL constraint : It is the constraint which restricts the given column to accept the NULL value. We have explicit create statement to include this constraint on any column.

CHECK constraint : It is the constraint which restrict the given column to accept only predefined value. It can be used generally for validation purpose. Explicit create statement is required to include this constraint on any column.

iii) Referential Integrity Constraints : These constraints establish the relationship between master table and child table. Foreign key is the only constraint available under this type.

FOREIGN KEY : The primary key of a master table acts as a foreign key to the child table. In this way, a relationship is formed.

Features of Foreign key :

- For a given table, any number of foreign keys can be included that depends upon the number of existing columns and our requirements.
- The primary key of one table can acts as a foreign key to any number of columns of the other tables.
- The primary key of one table can acts as a foreign key to more than one columns of the same table.
- The primary key of a table can also acts as a foreign key to another column of the same table. This is called as self referential key.
- During the creation of these tables, first, we need to create master table with a primary key and then to create child table with a foreign key.
- During the insertion of the data values, first we need to insert the data into master table and then to child table, because, child table's foreign key column will accept the value which is there in the master table's primary key column. If we enter other

value, it is giving an error message ‘Integrity constraint violated, parent key not found’.

- During the deletion of the data values, first we need to delete the data from child table and then from master table, If we try to delete master table values, then it is giving an error message ‘Integrity constraint violated, child record found’.
- A foreign key column can accept NULL value and redundant values.

The Relational Algebra and Relational Calculus :

The basic set of operations for the relational model is the relational algebra. These operations enable a user to specify basic retrievals requests. The result of a retrieval is a new relation, which may have been formed from one or more relations. The algebra operations thus produce new relations, which can be further manipulated using operations of the same algebra. A sequence of relational algebra operations forms a relational algebra expression, whose result will also be a relation that represents the result of a database query.

The relational algebra is very important for several reasons.

First, it provides a formal foundation for relational model operations.

Second, it is used as a basis for implementing and optimizing queries in relational database management systems.

Third, some of its concepts are incorporated into the SQL, structured query language for RDBMSs.

Although, no commercial RDBMS in use today provides an interface for relational algebra queries, the core operations and functions of any relational system are based on relational algebra operations. Whereas the algebra defines a set of operations for the relational model, the relational calculus provides a higher –level declarative notation for specifying relational queries. A relational calculus expression creates a new relation, which is specified in terms of variables that range over rows of the stored database relations or over columns of the stored relations.

In a query result, a calculus expression specifies only what information the result should contain. This is the main distinguishing feature between relational algebra and relational calculus. The relational calculus is important, because, it has a firm basis in mathematical logic and because the standard query language for RDBMSs has some of its foundation in the tuple relational calculus.

The relational algebra is often considered to be an integral part of the relational data model. Its operations can be divided into two groups. One group includes set operations from mathematical set theory, these are applicable, because each relation is defined to be a set of tuples in the formal relation model. Set operations include UNION, INTERSECTION, SET DIFFERENCE, and CARTESIAN PRODUCT. The other group consists of operations developed specially for relational databases, these include SELECT, PROJECT and JOIN.

Relational Operations

The PROJECT operation : It selects certain columns from the table and discards the remaining columns. If we want to retrieve only certain attributes of a relation or all the attributes, we use the PROJECT operation. The general form of the PROJECT operation is

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

Here, π is the symbol used to represent Projection, <attribute list> is the list of columns (attributes) of a relation R and (R) is the relation name (table name)

How to write SQL queries in relational algebraic expressions ?

Example : To display employee details from the table emp, the SQL query can be written as

```
SQL> select empno, empname, job from emp;
```

The same SQL query can be represented in relational algebraic expression as follows :

$$R \rightarrow \pi_{<\text{empno, empname, job}>} (\text{Emp})$$

The SELECT operation : It is used to select a subset of the records from a relation that satisfies a select condition. It acts like a filter by selecting only required records by putting the condition in a query. The general form of the SELECT operation is

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

The Boolean condition specified in <select condition> is made up of a number of clauses of the form

<attribute name> <comparison operator> <constant value>

Here, attribute name is column name, comparison operator is equal operator '=' and constant value is a user defined value.

To display employee details from the table emp with the designation CLERK, the SQL query can be written as

```
SQL> select empno, empname, job from emp where job='CLERK' ;
```

The above query is a combination of projection and selection. It can be represented in relational algebraic expression as follows :

$$R1 \rightarrow \pi_{<\text{empno, empname, job}>} (\text{Emp})$$

$$R2 \rightarrow \sigma_{<\text{job='CLERK'}>} (\text{Emp})$$

$$R \rightarrow R1(R2)$$

The same query can also be represented as follows :

$R \rightarrow \pi_{< empno, empname, job>} (\sigma_{< job='CLERK'>} (Emp))$

The JOIN operation : It is used to combine related records from one, two or more relations in to a single record. A general form of JOIN operation on two relations is

$R \infty_{<join\ condition>} S$

Here R is the first relation, S is the second relation,
<join condition> contains R.Column=S.Column, R and S are the two relations, Column is the common column available from both the relations.

If number conditions increases, then conditions will be included by using Logical operators (i.e. AND, OR, NOT)

Example1 : To display employee details from the table emp and dept, the SQL query can be written as

```
SQL> select empno, empname, job, dname
      from emp, dept
       Where emp.deptno=dept.deptno
```

The above query is a combination of projection, join. It can be represented in relational algebraic expression as follows :

$R1 \rightarrow \pi_{< empno, empname, job,dname>} (Emp, Dept)$
$R2 \rightarrow Emp \infty_{< deptno = deptno >} Dept$
$R \rightarrow R1(R2)$

The same query can also be represented as follows :

$R \rightarrow \pi_{< empno, empname, job,dname>} (Emp \infty_{< deptno = deptno >} Dept)$

Example2: To display employee details from the table emp and dept with the designation CLERK, the SQL query can be written as

```
SQL> select empno, empname, job, dname
      from emp, dept
       Where emp.deptno=dept.deptno
        and  job='CLERK' ;
```

The above query is a combination of projection, selection and join. It can be represented in relational algebraic expression as follows :

$R1 \rightarrow \pi_{< empno, empname, job, dname>} (Emp, Dept)$

$R2 \rightarrow Emp \infty_{< deptno = deptno >} Dept$

$R3 \rightarrow \sigma_{< job='CLERK'>} (Emp, Dept)$

$R \rightarrow R1(R2(R3))$

The same query can also be represented as follows :

$R \rightarrow \pi_{< empno, empname, job, dname>} (\sigma_{< job='CLERK'>} (Emp \infty_{< deptno = deptno >} Dept))$

The DIVISION operation : It is used for a special kind of query that sometimes occurs in database applications. It is denoted by ÷ , In general, the DIVISION operation is applied to two relations

$R(Z) \div S(X)$, where $X \subseteq Z$.

Let Y = Z – X (and hence Z = X U Y); that is, let Y be the set of attributes of R that are not attributes of S. The result of DIVISION is a relation T(Y) that includes a tuple 't' if tuples t_R appear in R with $t_R[Y] = t$, and with $t_R[X] = ts$ for every tuple ts in S. This means that, for a tuple t to appear in the result T of the DIVISION, the values in t must appear in R in combination with every tuple in S.

The Cartesian Product : If two tables in a join query have no join condition, then query results in a type of a result set in the form of **Cartesian product**. Query combines each row of one table with each row of the other [5]. It is in the form of M X N. i.e. M is the number of records in first table and N is the number of records in the second table [4]. A Cartesian product always generates many rows and is rarely useful. For example, the Cartesian product of two tables, each with 50 rows, has 2,500 rows. Therefore always be careful in using join conditions. If a query joins three or more tables and you do not specify a join condition for a specific pair, then the optimizer may choose a join order that avoids producing an intermediate Cartesian product.

Relational Algebra Operations from Set Theory :

The next group of relational algebra operations are the standard mathematical operations on sets. Several set theoretic operations are used to merge the elements of two sets in various ways, that include UNION, INTERSECTION, and SET DIFFERENCE (MINUS). These are binary operations; that is, each is applied to two sets (of tuples) When these operations are adapted to relational databases, the two relations on which any of these three operations are applied must have the same type of tuples. This condition has been called union compatibility.

Two relations $R(A_1, A_2, - - - A_n)$ and $S(B_1, B_2, - - - - B_n)$ are said to be union compatible if they have the same degree 'n' and if $dom(A_i) = dom(B_i)$ for $1 <= i <= n$. This means that the two relations have the same number of attributes and each corresponding pair of attributes has the same domain.

UNION : The result of this operation, denoted by R U S, is a relation that includes all tuples that are either in 'R' or in 'S' or in both R and S. Duplicate tuples are eliminated.

INTERSECTION : The result of this operation, denoted by R Π S, is a relation that includes all tuples that includes all tuples that are in both R and S.

SET DIFFERENCE (or MINUS) : The result of this operation, denoted by R - S, is a relation that includes all tuples that are in 'R' but not in 'S'.

Notice that both UNION and INTERSECTION are commutative operations, that is

R U S = S U R and R Π S = S Π R

Both UNION and INTERSECTION can be treated as n-ary operations applicable to any number of relations because both are associative operations; that is

R U (S U T) = (R U S) U T and (R Π S) Π T = R Π (S Π T)

The MINUS operation is not commutative; that is, in general,

$R – S \neq S - R$

Aggregate Functions and Grouping : Another type of request that cannot be expressed in the basic relational algebra is to specify mathematical aggregate functions on collections of values from the database. These functions are used in simple statistical queries that summarize information from database tuples. Common functions applied to collections of numeric values include SUM, AVERAGE, MAXIMUM, and MINIMUM. The COUNT function is used for counting tuples or values.

Another common type of request involves grouping the tuples in a relation by the value of some of their attributes and then applying an aggregate function independently to each group. We can define an AGGREGATE FUNCTION operation using the symbol ' ζ ' to specify these types of requests.

<Grouping attributes> ζ <Function List> (R)

Where <Grouping attributes> is a list of attributes of the relation specified in R, and
<Function List> is a list of (< function > < attribute>) pairs.

In each such pair, <function> is one of the allowed functions such as SUM, AVG, MAX, MIN, COUNT and <attribute> is an attribute of the relation specified by R.

The resulting relation has the grouping attributes plus one attribute for each element in the function list.

To display the Number of employees according to their designation, we write SQL query in a following way :

```
SQL> select Job, count(job) as noofemps, sum(salary) as Total_Salary
        From emp
        Group by job;
```

The same SQL query can be represented in relational algebraic expression as follows :

R → < job > ζ < Count (job), Sum(salary) > (Emp)

Unit – III. SQL – The Relational Database Standards.

The objectives of this unit is to :

- Understand what is SQL ? its features, its working environment,
- Understand How to work with SQL language, its interfaces (SQLPLUS) and commands ?
- Understand How many ways we can access the data from the tables ? Select statement with various syntaxes.
- Understand How to use Arithmetic operators, NULL values, Single row analytical functions, and aggregate functions in SQL queries ?
- Understand different Joining types, the usage of Group by clause with Having clause, sub queries their types
- Understand How to create tables, alter tables, remove tables, ?
- Understand How to make manipulation of the data (INSERT, UPDATE, DELETE)
- Understand DCL (Grant, Revoke) and TCL (Commit, Rollback) commands.

Introduction to SQL

The Structured Query Language (SQL) is the language of databases. All modern relational databases, including MS Access, FileMaker Pro, Microsoft SQL Server and Oracle use SQL as their basic building block. In fact, it's often the only way that you can truly interact with the database itself. All of the fancy graphical user interfaces that provide data entry and manipulation functionality are nothing more than SQL translators. They take the actions you perform graphically and convert them to SQL commands understood by the database.

SQL is Similar to English. SQL is a simple language. It has a limited number of commands and those commands are very readable and are almost structured like English sentences. SQL was originally developed at IBM in the SEQUEL-XRM and System-R projects (1974-1977). Almost immediately, other vendors introduced DBMS products based on SQL, and it is now a de facto standard. SQL continues to evolve in response to changing needs in the database area.

Here, in this notes, to work with various concepts of SQL, we took the working language as Oracle8i. All the given examples are based on Oracle8i.

Some facts about Oracle8i :

- First ever commercially developed RDBMS (1979)
- First database to have procedural language support
- First ever Client/Server database (1986)
- First 64 bit RDBMS
- First web database
- First to complete 3terrabyte data volume (world record)
- Supports XML to large extent

Getting started with SQL*PLUS :

To start SQL*Plus on a typical installation on windows machine go to programs->Oracle Home->Application development->SQL*Plus or type SQLPLUSW at the command prompt

SQL*Plus is a oracle proprietary interface to interact with Oracle database. SQL*Plus is an interactive and batch query tool that is installed with every Oracle Database installation. It has a command-line user interface, a Windows Graphical User Interface (GUI) and the *i*SQL*Plus web-based user interface.

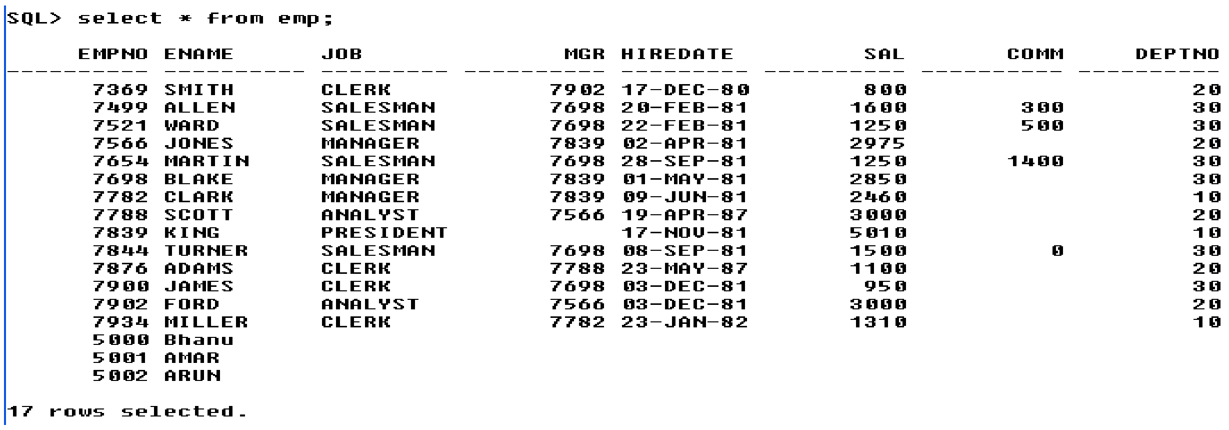
There is also the SQL*Plus Instant Client which is a stand-alone command-line interface available on platforms that support the OCI Instant Client. SQL*Plus Instant Client connects to any available Oracle database, but does not require its own Oracle database installation. SQL*Plus has its own commands and environment, and it provides access to the Oracle Database. It enables you to enter and execute SQL, PL/SQL, SQL*Plus and operating system commands to perform the following:

- Format, perform calculations on, store, and print from query results
- Examine table and object definitions
- Develop and run batch scripts
- Perform database administration

How to check existing objects in the database ?
SQL > Select * from tab ;
This will list out all the tables present in scott schema (any user who has logged in).
What is TAB?
A table containing details about all the tables in a user's area
Anything everything in the database can be stored only by means of a row in a table

What is a Table ?

Table is an object which stores the values in the form of rows and columns in the database. A database has got group of tables and a table may have group of columns. No table is available with fixed number of columns and no database is available with fixed number of tables. Number of tables in a database is always depends on the user requirements. Similarly number of columns of a table is always depends on the user requirements. Here is a table shown in the following figure which has got the basic information of the employees who are working in different departments. In this table, empno is a primary key, deptno and mgr are the foreign keys.



```
SQL> select * from emp;

     EMPNO ENAME      JOB              MGR HIREDATE         SAL       COMM     DEPTNO
---------- ---------- --------- ---------- --------- ---------- ---------- ----------
      7369 SMITH      CLERK           7902 17-DEC-80        800                    20
      7499 ALLEN      SALESMAN        7698 20-FEB-81       1600        300         30
      7521 WARD       SALESMAN        7698 22-FEB-81       1250        500         30
      7566 JONES      MANAGER         7839 02-APR-81       2975                    20
      7654 MARTIN     SALESMAN        7698 28-SEP-81       1250       1400         30
      7698 BLAKE      MANAGER         7839 01-MAY-81       2850                    30
      7782 CLARK      MANAGER         7839 09-JUN-81       2460                    10
      7788 SCOTT      ANALYST         7566 19-APR-87       3000                    20
      7839 KING       PRESIDENT            17-NOV-81       5010                    10
      7844 TURNER     SALESMAN        7698 08-SEP-81       1500          0         30
      7876 ADAMS      CLERK           7788 23-MAY-87       1100                    20
      7900 JAMES      CLERK           7698 03-DEC-81        950                    30
      7902 FORD       ANALYST         7566 03-DEC-81       3000                    20
      7934 MILLER     CLERK           7782 23-JAN-82       1310                    10
      5000 Bhanu
      5001 AMAR
      5002 ARUN

17 rows selected.
```

Customizing SQL*PLUS :

- SQLPLUS is a interface provided from oracle.
- It is a command oriented interface which works by receiving commands and sql statements.
- SQL*Plus window can be customized to show the output and set certain behaviors as needed by you.
- Type HELP SET at SQL prompt and try to know many settings that could be customized for your need.
- Use *ed* for editing last executed SQL statement

Getting Help for SQL PLUS commands :

To get help in case of sqlplus related commands, we need to type the key word help followed by sql plus command. That will give brief description about the command.

Example : SQL>Help SQL or SQL>Help SET or SQL>Help ed or SQL>Help Desc

Setting number of lines per screen size :
SQL > Set linesize 120;

Setting Page size :
SQL > Set pagesize 60;

How to select the values from the table :

Basic SELECT statement :

The basic syntax of select statement is

```
SQL > SELECT < column list > from < table name> where [< condition >];
```

Here, query should start with key word SELECT, followed by list of required columns which separated by commas. From clause consists of table name. Where clause is optional one which is required to include the condition.

SELECT statement is the only way of retrieving (querying) stored data from tables.

Example : SQL > Select * from emp;

Here, ***** represents all the columns of the table. Emp is the table name. Columns appear in the order in which it would appear when you describe the table. If we give a query like "Select * from emp;", it selects all column values from every records of emp table.

How to select only required columns from the table ?
Consider the second query :

SQL > Select empno, ename, job, sal from emp;

Here, Only required columns can be displayed. There is no restriction to display the columns in only specific order. There is a liberty to select the required columns in any of your order.

How to write SQL statements ?

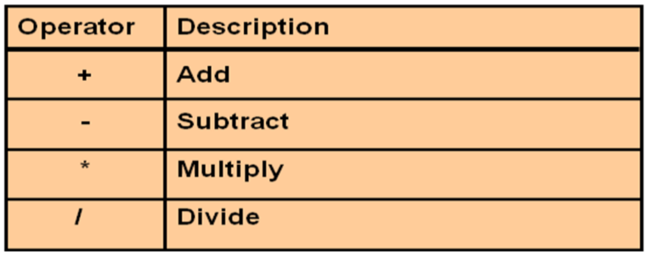
- SQL statements are not case sensitive.
- SQL statements can be on one or more lines.
- Keywords cannot be abbreviated or split across lines.
- Clauses are usually placed on separate lines.
- Indents are used to enhance readability.

SQLPLUS Columns heading defaults :

- Character and Date column headings are left- justified
- Number column headings are right-justified
- Default heading display: Uppercase

How to use arithmetic expressions in SQL ?

Create expressions with number and date data by using arithmetic operators.



Operator	Description
+	Add
-	Subtract
*	Multiply
/	Divide

If we want to add Rs. 100 to every employee of emp table, here is the query:

SQL > select empno, ename, sal, sal+100 from emp;

If we want to subtract Rs. 100 from every employee of emp table, here is the query:

SQL > select empno, ename, sal, sal - 100 from emp;

If we want to increase the salary of every employee by 25 % of emp table, here is the query:

SQL > select empno, ename, sal, sal*25/100 from emp;

If we want to reduce the salary of every employee by 50 % of emp table, here is the query:

SQL > select empno, ename, sal, sal/2 from emp;

Some times, we need to use more than one operator in given expression, then we need to follow the priorities of the operators.

- Multiplication and division take priority over addition and subtraction.
- Operators of the same priority are evaluated from left to right.
- Parentheses are used to force prioritized evaluation and to clarify statements.

Consider the following query:

SQL > select empno, ename, sal + sal * 0.25 from emp;

Here, since we have two operators, * is higher priority over + operator, sal value is multiplied with 0.25 and the result is added with existing salary.

Using the parenthesis : Consider the following query :

SQL> SQL > select empno, ename, 12 * (sal + 100) from emp;

Here, even though, * is having higher priority over +, that will be evaluated first, because it is given with in the parenthesis.

Find the Output :

SQL > Select 5 + 7 / 2 * 5 from dual;

SQL > Select 5/(2 + 3)*2 – 25/5 from dual;

Defining a null value :

NULL value : A null is a value that is unavailable, unassigned, unknown, or inapplicable. In other sense, it is a empty value. A null is not the same as zero or a blank space.

NULL values in arithmetic expressions :

Arithmetic expressions containing a null value evaluate to null. If we perform any arithmetic operations with null value, the result is also a null value.

Defining a column alias :

- Renames a column heading
- Is useful with calculations
- Immediately follows the column name - there can also be the optional AS keyword between the column name and alias
- Requires double quotation marks if contains spaces or special characters

SQL> Select empno, sal, sal*12 as annual_salary, comm commission from emp;

- Annual_salary is the column alias given to the calculated column sal*12 in the output result set.
- When writing sub-queries column alias usage becomes mandatory for calculated columns
- ‘as’ key word is optional
- Once aliased, only the aliased name can be used to refer to that column from that result set

Note :

Giving column alias will not change the column name in the table, but only in the result set generated by the query. Column aliasing for expressions in the column list is essential in sub-queries.

Using Concatenation Operator (||) :

- || is the string concatenation operator used to concatenate values of two columns or more number of columns.
- Any data type value would be typecast implicitly to a equivalent string representation of the value and then concatenated with the result string

Example :

i) SQL > Select empno || ename from emp;

ii) SQL > Select empno || ename||job from emp;

Literal character strings :

- A literal is a character, a number, or a date included in the SELECT list.
- Date and character literal values must be enclosed within single quotation marks.
- Each character string is output once for each row returned.

Example :

i) SQL> Select empno, ename||' is working as '||job from emp;

ii) SQL> Select empno,ename||' is working as '||job||' since '||hiredate||' and gets salary of $'||sal as Details from emp;

Duplicate Rows :

If row is repeated by more than once is called as duplicate row. In other sense, we call it as a redundant data.

How to avoid duplicate row in a output of the data ? or How to select non duplicate rows ?

SQL > Select distinct Job from emp;

or

SQL > Select unique job from emp;

Distinct key word specifies that only unique values from the column(s) selected could be displayed in the output. Output would be sorted in the ascending order of selected column(s).

Limiting the rows selected :

- Restrict the rows returned by using the WHERE clause.
- The WHERE clause follows the FROM clause.

  Syntax : SELECT *|{[DISTINCT] *column|expression* [*alias*],...}
  FROM *table*
  [WHERE *condition(s)*];

  Using the WHERE Clause, condition can be framed by using following syntax :

  Where <column-name> <any of the relational operator> <' constant value or data value'>;

Relational operators :

> → greater than, < → Less than, = → Equal to, <> → not equal to ,
>= → greater than or equal to , <= → Less than or equal to

Example : i) Sal > 1200 ii) deptno <> 10 iii) Sal <= 2000 iv) hiredate > '10-jul-1980'

Other operators :

IN List operator, Example : deptno IN (10,30)
BETWEEN lower range..AND.. Higher range, Example : Sal between 1000 and 2000
LIKE operator : Pattern matching operator.

Example :

```
SQL> SELECT empno, ename, job, deptno
         FROM   emp
        WHERE  deptno = 10 ;
```

Character strings and dates :

- Character strings and date values are enclosed in single quotation marks.
- Character values are case sensitive, and date values are format sensitive.
- The default date format is DD-MON-RR.

Example :

```
i)  SQL>SELECT ename, job, deptno FROM   emp WHERE  ename = 'KING';
II) SQL> Select * from emp where sal = 3000;
III) SQL> Select * from emp where sal > 3000;
IV) SQL> Select * from emp where sal >= 3000;
V) SQL> Select * from emp where sal < 3000;
VI) SQL> Select * from emp where sal <= 3000;
VII) SQL> Select * from emp where sal <> 3000;
```

Date formats : In SQL, the date will take this format "dd-Mon-yy".

Here, 'dd' represents day in two digits number,
'Mon' represents month name in three lettered format,
'yy' represents year in two digits format.

Example :

```
SQL > select * from emp where hiredate = '12-Jan-81';
```

Other comparison operators :

IN operator : It is a list operator which replaces OR operator.

Example : i) Find the employees who are working with designations 'CLERK','SALESMAN' and 'ANALYST'

```
SQL> select * from emp where job='CLERK'  OR  job='SALESMAN'  OR  job='ANALYST';
```

This query can be written by using IN operator in the place of OR operator.

```
SQL> select * from emp where job IN ('CLERK','SALESMAN','ANALYST');
```

ii) Find the employees who are working in the departments 10, 20 and 30.

SQL> select * from emp where deptno IN (10,20,30);

BETWEEN Operator : It will be used to compare the values in the range.

Syntax : Where <column-name> BETWEEN <lower value> AND <higher value> ;

It will replace the relational operators >= and <=.

Example :

SQL > Select * from emp Where sal >=1000 and sal <=2500;

This query can also be written by using BETWEEN operator in the following way:

SQL > Select * from emp Where sal between 1000 and 2500;

Note : Be careful in using BETWEEN operator with respect to the parameters : The first parameter is lower value and second parameter is higher value.

LIKE operator : It is a string or pattern matching operator. It is used to perform wildcard searches of valid search string values. Search conditions can contain either literal characters or numbers:

Here it is used with two symbols : % and _

- % denotes zero or many characters.
- _ denotes one character.

Example :

i) Find the employee whose name starts with 'S' as the first character from emp table.

SQL > select ename from emp where ename like 'S%';

ii) Find the employee whose name starts with 'S' as the first character and 'R' as the last character from emp table.

SQL > select ename from emp where ename like 'S%R';

iii) Find the employee whose name in which 'O' as the third character from emp table.

SQL > select ename from emp where ename like '__O%';

iv) Find the employee whose name in which number of characters are 5 from emp table.

SQL > select ename from emp where ename like '_____';

Using IS NULL condition: To find the column values with null type, we will be using IS NULL operator.

Example :

i) Find the employees who are getting commission as the NULL value from emp table.

SQL > select * from emp where comm = NULL;

The above query does not give proper result. The correct query is follows :

SQL > select * from emp where comm is NULL;

ii) Similarly, find the employees who are getting commission as the not NULL value from emp table.

SQL > select * from emp where comm is not NULL;

Logical Operators : These operators can be used to give more than one condition in the where clause which combines logical results (true / false) of two relational expressions (or negates logical result of a relational expression). The final result will work similar to truth table of AND, OR and NOT operators.

Examples :

i) Find the employees whose salary is greater than 2000 and their designation is CLERK type.

```
SQL > Select * from emp  Where job='CLERK' AND sal > 2000;
```

ii) Find the employees either whose salary is greater than 2000 or their designation is CLERK type.

```
SQL > Select * from emp  Where job='CLERK' OR sal > 2000;
```

iii) Find the employees other than CLERK and SALESMAN type.

```
SQL > Select * from emp  Where job NOT IN ( 'CLERK','SALESMAN');
```

Order of evaluation operators in arithmetic expressions :

1. Arithmetic and string manipulation operators
2. All relational operators (including IN, BETWEEN ..AND , LIKE)
3. NOT operator will be applied next
4. Logical operator AND,
5. Logical operator OR

Sorting the Result :

The records of the table can be ordered according to our requirement. We need not follow its existing order. This is possible by using order by clause.

Syntax : ORDER BY < column name > [DESC];

By default, it will display is ascending order. To display in descending order, we need to use the key word desc. The ORDER BY clause comes last in the SELECT statement.

Example :

i) To display employee details in an ascending order by taking empno as the reference, here is the SQL query:

```
SQL> SELECT empno, ename, deptno, hiredate from EMP ORDER BY empno
```

The above query, orders the result records in the ascending order of empno

ii) To display employee details in an descending order by taking hiredate as the reference, here is the SQL query:

```
SQL> SELECT empno, ename, sal, hiredate From EMP ORDER BY hiredate DESC
```

The above query, orders records in the descending order of hiredate (latest date first and earliest date last)

prestigious

iii) To display employee details in an ascending order by taking ename as the reference, here is the SQL query:

```
SQL > Select  empno, ename, job, sal from emp order by ename;
```

iv) To display employee details in an ascending order by taking salary as the reference, here is the SQL query:

```
SQL > Select  empno, ename, job, sal from emp order by sal;
```

How to apply order by clause on more than one column ?

Example :

```
SQL > Select  deptno, sal from emp order by deptno, sal ;
```

Here, it is applied with deptno and sal coulmns. The first preference will be given to deptno followed by sal column.

How to apply order by clause in ascending order on one column and descending order on the other column in a single query?

```
SQL > Select  deptno, sal from emp order by deptno, sal desc  ;
```

How to apply order by clause in descending order on both columns in a single query?

```
SQL > Select  deptno, sal from emp order by deptno desc, sal desc  ;
```

Sorting on columns with NULL values :

Null values are neglected by Sorting clause and they will be displayed immediately after not null values

Example :

```
SQL > Select  * from emp order by comm ;
```

Null values will be displayed as last records when it is in ascending order, displayed in beginning when it is displaying in descending order.

SQL functions :

There are many inbuilt functions available in SQL which return a value when you call them in an SQL query. Mainly, they can be classified as

a) Single row analytical functions
b) Conversion functions
c) Aggregate functions

a) Single row analytical functions : These functions will return a value for every row of a table. In this type, we have several types.

   i) Single row number functions
   ii) Single row character functions
   iii) Single row date functions
   iv) Single row miscellaneous functions

Note : While using these functions, to get single result, we need to use the reference table as DUAL table. This table has got single column with single value. This will be used commonly in order to get single row value.

i) Single row number functions :

These functions will be used on the columns of the number data type.

Example : abs(n), ceil(n), exp(n), mod(m,n), power(m,n), sqrt(n), round(m,n)…

SQL> Select abs(-25) from dual ;

abs function returns a positive integer, here result is 25.

SQL> select sqrt(25) from dual;

sqrt functions returns a square root value for a number 25, here result is 5.

SQL> select mod(27,5) from dual;

mod(m,n) function returns a remainder after diving 27 by 5, here result is 2

SQL> select ceil(27.01) from dual;

Ceil function returns a next nearest integer number, here result is 28.

ii) Single row character functions:

These functions will be used on the columns of the character data type.

Example : concat, initcap, upper, lower, ltrim, rtrim, trim, …

SQL> select concat (empno,ename) from emp;

It combines the column values of empno and ename to single value.

SQL> select initcap(ename) from emp;

It returns a ename in which first letter is in upper case and remaining letters in lower case.

SQl>select lower(ename) from emp;

It returns a ename in lower case letters.

SQl>select upper(ename) from emp;

It returns a ename in upper case letters.

iii) Single row date functions:

These functions will be used on the columns of the date data type.

Example : Add_months(date, n_months), months_between(date1, date2), sysdate, ….

SQL> select add_months ('12-Nov-14',4) from dual;

It returns a date by adding 4 months to the specified date, here the result is '12-feb-15'.

SQL> select months_between('1-jan-14','30-jun-14') from dual;

It returns a value by finding the number of months between '1-jan-14' and '30-jun-14' date, here the result is 6.

SQL> select sysdate from dual;

It returns a system date, here the result is '17-Nov-14'.

iv) Single row miscellaneous functions: These functions will be used on the columns of the character or number data type for special occasions.

Example : decode(n), greatest(n) least(n), nvl(n)……

b) Conversion functions : These functions will be used to convert the existing data from one format to another format. Some of the conversion functions are to_char, to_number, to_date….
Example :

SQL > select to_char(sal,'L99G999D99') from emp;

It returns a value in $ 1,250.00 format. It converts the existing value 1250 to $1,250.00. Here L represents currency sign, G represents group separator, D represents decimal point.

SQL> select to_date('November, 12, 2014','Month, dd, YYYY') from dual;

It returns a date value to SQL acceptable format. i.e. 12-Nov-14

c) Aggregate functions : These functions will be used on the column of number data type to get aggregate value. These will return a single value by considering many values in a column.

Example : count(n), max(n), min(n), sum(n), avg(sal)…..

SQL> select count(*) from emp;

It returns a value which represents number of rows in emp table.

SQL> select max(sal) from emp;

It returns a value which represents highest salary in emp table.

SQL> select min(sal) from emp;

It returns a value which represents lowest salary in emp table.

SQL> select sum(sal) from emp;

It returns a value which represents total salary in emp table.

SQL> select avg(sal) from emp;

It returns a value which represents average salary in emp table.

Note : We cannot use both single row function and aggregate function in a single query. Because, single row function is returning a value for every row whereas aggregate function returning a single value which leads to ambiguity for the SQL.

SQL joins :

Whenever the required data is not available from a single table, it requires to get from more than one table, we need to use sql joins.

What is Join ?

- Retrieving data from more than one tables related rows between them on the basis of relation between a set of column(s) between them
- Mandatory condition to join two tables
- At least one set of column(s) should be taking values from same domain in each table

In join processes, the accessing of the data depends on the joining conditions with different operators. Here, join condition is a must. For this purpose, generally we are using relational operators along with logical operators.

Joining conditions :

Many of the joining queries contain WHERE clause conditions that compare two columns, each from a separate table. Such a condition is called a join condition. To execute a join, database software combines pairs of record sets, each of the record set containing one record from each table, for which the join condition evaluates to TRUE. The columns in the join conditions need not be appear in the select list. It is an optional one.

To execute a join of three or more tables, SQL engine first joins two of the tables based on the join conditions comparing their columns and then joins the result to another table based on join conditions containing columns of the joined tables and the new table. SQL engine continues this process until all tables are joined into the result. The optimizer specifies the order in which SQL engine joins tables based on the join conditions, indexes on the tables, and, any available statistics for the tables .

In addition to join conditions, the WHERE clause of a join query can also contain other conditions that refer to columns of only one table. These conditions can further filter the records returned by the join query.

Types of Joins :

Mainly there are three types of joins based on the way they retrieve the records. They are Inner join, Outer join and Joining more than two tables.

a) Inner Join : An inner join (sometimes called a simple join) is a join of two or more tables that returns only those records that satisfy the join condition. i.e. it is looking for only the matching records . There are three types in this inner join. They are

i) Inner Equi join,
ii) Inner Non-Equi join
iii) Inner Self join

i) Inner Equi-Join : An equijoin is a join with a join condition containing an equality operator. An equijoin combines records that have equivalent values for the specified columns. Depending on the implicit algorithmic plan, the optimizer chooses the execution plan for this equijoin.

To give an example to this join, here it is used with EMP and DEPT tables. This join needs one common column. i.e. DEPTNO which acts as Primary key in DEPT table and Foreign key in EMP table. The joining condition requires the usage of EQUAL operator. The following SQL query retrieves the employee number, employee name, designation and department name from the tables EMP and DEPT

```
Example : SQL>select empno,ename,job,dname
                 From emp, dept
                 Where emp.deptno=dept.deptno;
```

ii) Inner Non Equijoin : A Non equijoin is a join with a join condition without containing an equality operator. A non-equijoin combines records that have equivalent values for the specified columns. Depending on the implicit algorithmic plan the optimizer chooses the execution plan for this join.

To give an example to this join, here it is used with EMP and SALGRADE tables. This join does not require any common column or master child relationship among the tables. The joining condition prohibits the usage of EQUAL operator. The following SQL query retrieves the employee number, employee name, salary and grade from the tables EMP and SALGRADE.

```
Example : SQL>select empno,ename,sal,grade
                From emp, salgrade
                Where sal>=losal  and sal <= hisal;
```

iii) Inner Self Join : A self-join is a join of a table to itself. Since a join requires minimum of two tables, here, we refer the same table twice in the FROM clause and is followed by table aliases that qualify column names in the join condition. To perform a self-join, SQL combines and retrieves records of the table that satisfy the join condition.

To give an example to this join, here it is used with EMP table only. In this table, the column EMPNO is primary key and MGR is foreign key which references primary key of the same table. This join is the concept works under the UNARY relationship. The following SQL query retrieves the employee name in one column and their respective managers in the other column. Here, the same EMP table will be referenced twice with alias name.

```
Example : SQL>select E.ename as Employee, M.ename as Manager
                From emp E, emp M
                Where M.empno=E.mgr;
```

b) Outer Join : An outer join extends the result of a simple join. An outer join returns all records that satisfy the join condition and also returns some or all of those records from one table for which no record from the other table satisfy the join condition. In other words, first of all it is looking for matching records and remaining records (usually with null values) from one table or both the tables. There are three types in this outer join. They are

i) Left outer join
ii) Right outer join
iii) Full outer join

i) Left Outer Join **:** As the name itself indicates that, this query mainly looking for matching records and remaining records from left table. How do we know that which is left table ? and which is right table ?. It will be decided by looking the following join condition.

```
Table-A.Column_Id = Table-B.Column_Id(+)
```

Here, the table which is located at the left hand side of equality operator is considered as Left table and the table specified at the right hand side of equality operator is considered as Right table. In the above case, Table-A is left table and Table-B is right table. There is a provision to use any table in any of the side.

To give an example to this join, here it is used with EMP table as left table and DEPT table as right table. The following SQL query retrieves the records containing the employee number, employee name, designation, and department name in which he is working.

```
Example : SQL>select empno, ename, job, dname
                    from emp, dept
                    Where  emp.deptno=dept.deptno (+);
```

The same query can also be written as follows.

```
SQL > select empno,ename,job,dname
        From emp left outer join dept
        On emp.deptno=dept.deptno;
```

ii) Right Outer Join : This join query mainly looking for matching records and remaining records from right table.

To give an example to this join, here it is used with EMP table as left table and DEPT table as right table. The following SQL query retrieves the records containing the employee number, employee name, designation, and department name in which he is working.

```
Example : SQL>select empno,ename,job,dname
                   from emp, dept
                    Where  emp.deptno(+)=dept.deptno;
```

The same query can also be written as follows.

```
SQL > select empno,ename,job,dname
          From emp right outer join dept
          On emp.deptno=dept.deptno;
```

iii) FULL Outer Join : This join query mainly looking for matching records and remaining records from both the tables.

The following SQL query retrieves the employee number, employee name, designation, and department name in which he is working.

```
Example : SQL>select empno,ename,job,dname
                    from emp Full outer join dept
                    on  emp.deptno=dept.deptno;
```

c) Joining more than Two tables : It is a generalized joining process in which number of tables is exceeding by two. Here, the minimum number of joining conditions required depends on the number of tables being joined. To join N number of tables, at minimum, we need (N – 1) joining conditions.

Example :

```
SQL> select studentid, studentname, BranchName, ClassName, TeacherName
         From Student, Branch, Class, Teacher
         Where student.branchid=Branch.BranchId
         And student.ClassId=Class.ClassId
         And Student.TeacherId=Teacher.TeacherId;
```

Cartesian product :

When we join the tables, if we don’t put the join condition, that results in Cartesian product type. i.e. M X N type. The number of records in first table will be multiplied by the number of records in the second table.

Example : SQL > select empno, ename, job, dname from emp, dept

It returns number of rows as 56 by assuming number of rows in emp table as 14 and number of rows in dept table as 4. (i.e. 14 X 4 = 56)

Group Functions :

To group the similar data items in a row, we use group functions. In SQL, we have group by clause.

Group by and Having Clauses :

Group by clause acts only on the selected rows after applying where clause if present in the select statement. Grouping column contains a Null value, it will form a separate group and hence there will be one row with Null group column(s) value in the result set. Group functions will ignore Null values in the aggregate columns

When to use Group by clause in Select Statement?

By default entire table records together formed as one group by group functions.

When we need aggregations like below, we have to use Group by Clause

– Department wise total salary of employees
– Month wise count of employees joined
– Stock category wise inventory value
– Account type and branch wise average balance maintained

Usage of Group by Clause :

i) To list out designation wise number of employees working from emp table.

```
SQL >  select job, count(job) as noofemps from emp group by job;
```

ii) To list out designation wise number of employees working, along with their total salary from emp table.

```
SQL >  select job, count(job) as noofemps, sum(sal) as total_salary  from emp group by job;
```

iii) To list out department wise number of employees working from emp table.

```
SQL >  select deptno, count(deptno) as noofemps from emp group by deptno;
```

iv) To list out department wise number of employees working along with total salary, average salary, maximum salary and minimum salary from emp table.

```
SQL >  select deptno, count(deptno) as noofemps, sum(sal) as total_salary,
          max(sal) as highest_salary, min(sal) as Lowest_salary, avg(sal) as Average_salary
         from emp
         group by job;
```

Having Clause :
It can be used to put condition on grouped data item. This will work only with group by clause.

Example : i) List out the departments in which minimum number of employees are 4.

```
SQL > select deptno, count(deptno) as noofemps
        From emp
       Group by deptno
       Having count(deptno) >= 4;
```

ii) List out the employees designation wise in which their average salary is greater than 2500

```
SQL > select job, count(job) as noofemps, avg(sal) as average_salary
        From emp
        Group by job
       Having avg(sal) >= 2500;
```

iii) List out the number of employees according to department name wise in which there should not be any analysts.

```
SQL >   select dname, count(dname) as noofemps
        From emp, dept
        Where emp.deptno=dept.deptno
        And job <> 'ANALYST'
        Group by dname;
```

Rules to follow for using Group by clause :

i) The column on which we apply group by clause should be there in select clause.

Example :
```
SQL > select job, count(job) as noofemps
        from emp
        group by deptno;    ( This leads to error)
```

ii) Having clause should be used along with group by clause.

iii) Having clause must be specified only after where clause, if there exists a condition.

iv) Group by and having clause will support aggregate functions.

v) Where clause does not support aggregate function.

Example :
```
SQL> select * from emp
      Where sal=max(sal);  (This leads to error)
```

Differences between Where and Having clause :

- Where clause is executed on actual rows of the table being queried
- Where clause can not be used for the following
    - To filter output records of grouped results satisfying some grouped functions' result(s)
    - To filter out records satisfying some other grouped function (other than group function result sought for) result(s)

Example : i) To list out department wise No. of employees and average salary, of those employees who do not get a commission and only for those departments in which employees get a at least salary of 1000 or above.

```
SQL > Select  deptno, count(*) as no_of_emps, avg(sal) as avg_sal
        from     emp
        where   comm is null
        group  by deptno
        having  min(sal) >= 1000;
```

Set theory Operators in SQL :

You can combine multiple queries using the set operators.

UNION, UNION ALL, INTERSECT, and MINUS.

All set operators have equal precedence.

If a SQL statement contains multiple set operators, then Oracle database evaluates them from the left to right unless parentheses explicitly specify another order.

Rules to use Set Operators

- If both queries select values of datatype CHAR, then the returned values have datatype CHAR.
- If either or both of the queries select values of datatype VARCHAR2, then the returned values have datatype VARCHAR2.
- If component queries select numeric data, then the datatype of the return values is determined by numeric precedence:
- The order of columns and their data type should match in both queries.
- Both queries should be specified with equal number of columns.

Example : Assume there two tables with the name empinfy and empwipro in which they keep the details of their employees. The number of columns and their data types are same.

UNION : This will list out the records from both the tables by avoiding the redundancy. It combines the results of two queries with the UNION operator, which eliminates duplicate selected rows.

```
Example :   SQL> select empno,ename,job from empinfy
                        Union
                  Select empno,ename,job from empwipro;
```

UNION ALL: This will list out the records from both the tables by neglecting the redundancy (i.e. all the records). The UNION operator returns only distinct rows that appear in either result, while the UNION ALL operator returns all rows. The UNION ALL operator does not eliminate duplicate selected rows:

```
Example :   SQL> select empno,ename,job from empinfy
                        Union all
                  Select empno,ename,job from empwipro;
```

INTERSECT : This will list out the common records from both the tables. It combines the results with the INTERSECT operator, which returns only those rows returned by both queries:

```
Example :   SQL> select empno,ename,job from empinfy
                        intersect
                  Select empno,ename,job from empwipro;
```

MINUS : This will list out the records which are there in first table, but not in second table. It combines results with the MINUS operator, which returns only rows returned by the first query but not by the second

Example :
```
SQL> select empno,ename,job from empinfy
                minus
          Select empno,ename,job from empwipro;
```

Sub Queries :

What is sub query ?

A query written in some part of another query where it is syntactically accepted. Simply, a query within another Query. The syntax for using sub query is as follows :

```
SELECT select_list   → (Main query)
FROM  table
WHERE  column_name operator (SELECT select_list FROM table); → (Sub query)
```
Here, The subquery (inner query) executes once before the main query.
The result of the subquery is used by the main query (outer query).

Example : i) To list out employees who work for department 10 and who's salary is above the average salary of the department employees.

```
SQL > SELECT  * FROM     emp
      WHERE   deptno = 10
      AND SAL > (SELECT AVG(SAL) FROM EMP WHERE DEPTNO = 10)
```

Here is the sub query which gets the answer average salary of all the employees who are working in the department no 10

ii) List out number of employees working in the department 'ACCOUNTING'

```
SQL > Select count(deptno) as No_of_emps from emp
        Where deptno in (
                select deptno from dept  where dname='ACCOUNTING');
```

iii) List out the employee details who is drawing the highest salary in an organization

```
SQL > Select * from  emp
        Where sal = (  select max(sal) from emp);
```

Types of sub queries :

Based on number of values/rows the sub query is going to return, it is classified as

- Single row subquery
- Multiple row subquery
- Multiple column subquery

i) Single row subquery : This subquery returns a single value to the main query.

Example : To list out the employee details who is drawing the lowest salary.

```
SQL> select * from emp where sal = (select min(sal) from emp);
```

ii) Multiple row subquery : This subquery returns multiple values to the main query.

Example : To list out the employee details who are working for the departments 'Accounting' or 'Sales'.

```
SQL> select * from emp
        where deptno in (
                                  Select deptno
                                  From dept where dname='ACCOUNTING' or dname='SALES');
```

iii) Multiple columns subquery : This subquery returns values to the main query for more than one column. (i.e. In the where clause, it is mentioned with multiple columns)

Example : To list out the employee details who are working with the same designation and the department of Mr. ALLEN.

```
SQL> select * from emp
        where (deptno, job) in (
                                  Select deptno, job From emp
                                  Where ename='ALLEN');
```

Note : The order of columns mentioned in the where clause should be same as order of the columns mentioned in the select clause of subquery.

Exists operator :

- The EXISTS operator tests for existence of rows in the results set of the sub query.
- If a subquery row value is found:
  - The search does not continue in the inner query
  - The condition is flagged TRUE
- If a subquery row value is not found:
  - The condition is flagged FALSE
  - The search continues in the inner query

Example : i) Find employees who have at least one person reporting to them

```
SQL > SELECT empno, ename, job, deptno
        FROM   emp outer
        WHERE  EXISTS ( SELECT 'X'
                        FROM   emp
                        WHERE  mgr = outer.empno);
```

ii) Find all departments that do not have any employees.

```
SQL> SELECT deptno, dname
       FROM dept d
      WHERE NOT EXISTS (
                          SELECT 'X'
                            FROM   emp
                            WHERE  deptno = d.deptno);
```

Correlated Sub queries :

Correlated sub queries are used for row-by-row processing. Each subquery is executed once for every row of the outer query.

Syntax :

```
SELECT   select_list
FROM    table1   table1_alias1
WHERE   expr  operator
                        (
                          SELECT  column_list
                          FROM   table2   table2_alias2
                          WHERE   table1_alias1.column   operator   table2_alias2.column );
```

Here, subquery is evaluated once for each row of the main query.

Example : i) Find all employees who earn more than the average salary in their department.

```
SQL> SELECT ename, sal, deptno
        FROM   emp outer
        WHERE  sal > (
                        SELECT AVG(sal)
                        FROM   emp
                        WHERE  deptno = outer.deptno) ;
```

ii) Display details of those employees who have switched their jobs at least twice.

```
SQL>  SELECT e.employee_id, last_name,e.job_id
      FROM   employees e
      WHERE  2 <= (
                  SELECT COUNT(*)
                   FROM   job_history
                   WHERE  employee_id = e.employee_id);
```

iii) The following query returns data about employees whose salaries exceed their department average. The following statement assigns an alias to employees, the table containing the salary information, and then uses the alias in a correlated subquery:

```
Example :    SQL>  SELECT deptno, ename, sal
                   FROM emp x
                   WHERE sal > (
                               SELECT AVG(sal)
                               FROM emp
                              WHERE x.deptno = deptno )
                   ORDER BY deptno ;
```

For each row of the emp table, the parent query uses the correlated subquery to compute the average salary for members of the same department.

The correlated subquery performs the following steps for each row of the emp table:

1. The department_id of the row is determined.
2. The department_id is then used to evaluate the parent query.
3. If the salary in that row is greater than the average salary of the departments of that row, then the row is returned.

The subquery is evaluated once for each row of the employees table.

Nested Sub queries :

A subquery can be nested inside other sub queries. SQL has an ability to nest queries within one another. A subquery is a SELECT statement that is nested within another SELECT statement and which return intermediate results. SQL executes innermost subquery first, then next level.

Example : i) To find the department name in which the president is working in a emp table ?

```
SQL> select dname
        from dept
        Where deptno in  (
                                Select deptno
                                from emp
                                Where empno in (
                                                        Select empno
                                                        from emp
                                                        Where job='PRESIDENT'));
```

iI) To find the employee details who are working at the location 'DALLAS' ?

```
SQL> select * from emp
        Where empno in  (
                                Select empno
                                from emp
                                Where deptno in (
                                                        Select deptno
                                                        from dept
                                                        Where loc='DALLAS'));
```

Restrictions on Sub queries :

- A multiple row operator should be used when sub query (may) return more than one row
- Avoid order by clause in the sub-query
- Multiple column sub query result will be compared with values at corresponding position of columns appearing on the left hand side of the operator

Guidelines for writing sub query :

- Ensure the sub query (as a separate query) result before embedding it in a main query
- Sub query returning no values (NULL result set), will NOT result in any error
- Ensure the main query behavior, when some values returned by sub query could be Null.
- Appropriate operator to use based on expected query result
- Order of execution within a query having sub-query

DDL (Data Definition Language) Commands :

This language statements lets us create, alter, and drop the objects in a database. These are self-committed statements.

Objectives :

- Creating new table
- Data types
- Constraints
- Altering existing table
- Restrictions on Altering a table
- Managing voluminous tables

CREATE Command :

Create table command creates a new table in the present schema. Table name should not duplicate the existing object name in the same namespace.

Object naming guidelines :

1. Object name should start with an alphabet
2. Can be followed by one or more character/number and special characters in #, _ (underscore) , $
3. Name should be unique in the namespace allocated for the object
4. Relevantly name the table to reflect its role in the given business context

Oracle SQL data types

Column should be of any oracle SQL data type or user defined data type

Char : Character can be used for fixed width character data storage

Varchar2 : Varchar2 can be used for variables length character data which might vary in its actual values. This optimizes the space usage

Date : Date can be used to specify the date data type

Number : Number type may or may not contain decimal part in it. If present, can be mentioned like number(8,2) which specifies that total width of the column is 8 digits out of which 2 digits are used for decimal storage. No need of a place for decimal point in the storage.

Syntax to create a simple table :

```
Create  Table  < table name >  (
        Column1         datatype (size),
        Column2         datatype (size),
        ….
        ….
        ColumnN         datatype (size)    );
```

Example table : Student

```
Create  table  student  (
        StudentId               number(2),
        StudentName             varchar2(15),
        Sex                     char(1),
        DOB                     date,
        Marks                   Number(3) );
```

Constraints :

What is a constraint?

Restriction to be obeyed by the data when rows are being manipulated in a table

Why is it needed?

To ensure validity and/or meaningfulness of the data

Types of Constraints : Three Types of Constraints :

1. Entity Integrity Constraints
2. Domain Constraints
3. Referential Integrity Constraints

1. Entity Integrity Constraints :

Primary key, Unique key, Candidate key, Super key

Primary Key constraint :

- A table usually has a column or combination of columns whose values uniquely identify each row in the table.
- This column or column(s) is called the primary key.
- Primary key ensures that no duplicate or null values are entered in the column (or columns) defined as primary key columns.
- This enforces integrity of the table.
- A primary key can be created by defining a primary key constraint while creating or altering a table.
- A table can have only one primary key constraint.
- As primary key constraints ensure uniqueness, they are often defined by using an identity column.

Unique Key constraint

– Non duplicate values in the column(s) governed

How to add a primary key to the table :

Syntax-1 : To create the table with a column level primary key :

```
Create  Table  < table name >  (
Column1         datatype (size)   primary key,
Column2         datatype (size),
….
```

Example :

```
SQL >  Create  table  student  (
            StudentId               number(2)  Primary key,
            StudentName             varchar2(15),
            Sex                     char(1),
            Dob                     date,
            Marks                   Number(3) );
```

Syntax-2 : To create the table with a column level primary key by giving user defined name :

```
Create  Table  < table name >  (
Column1         datatype (size)   Constraint <name of the constraint> primary key,
Column2         datatype (size),
….
);
```

Example :
```
SQL > Create  table  student  (
          StudentId                number(2)  Constraint  pk_student_studentid  Primary key,
          StudentName              varchar2(15),
          ……
          );
```
Syntax-3 : To create the table with a primary key at a table level   (Standard format : This the one can be used at the industry level):

```
          Create  Table  < table name >  (
          Column1          datatype (size),
          ….
          ColumnN          datatype (size),
          Constraint <name of the constraint> primary key (column name)    );
```

Example :
```
          SQL > Create  table  student  (
                    StudentId                number(2)
                    StudentName              varchar2(15),
                    Sex                      char(1),
                    DOB                      date,
                    Marks                    Number(3),
                    Constraint  pk_student_studentid  Primary key (StudentId)  );
```

How to add a Unique key to the table :

To create the table with a column level Unique key :
```
                    Create  Table  < table name >  (
                    Column2          datatype (size) Unique key,
                    ….
                    );
```
Example :
```
          SQL > Create  table  student  (
                    StudentId                number(2)  Primary key,
                    StudentName              varchar2(15) Unique ,
                    Sex                      char(1),
                    DOB                      date,
                    Marks                    Number(3)  );
```

How to add a Candidate key to the table : ?

Basically, candidate key is a primary key, but it is always applied on more than one column of a table. In the following example, we created candidate key by applying it on three columns. i.e. studentid, studentname, sex.

Example :
```
          SQL > Create  table  student  (
                    StudentId                number(2)
                    StudentName              varchar2(15),
                    Sex                      char(1),
                    DOB                      date,
                    Marks                    Number(3),
                      Constraint  pk_student  Primary key (StudentId,Studentname,Sex) );
```

2. Domain Constraints :

– to preserve domain validity of data in a table column
– To preserve meaningfulness of data in a table column

Check constraint, Not null, Null

How to add a Check key to the table : ?

Syntax-1 : To create the table with a column level Check Constraint :

```
Create  Table  < table name >  (
---
Column2          datatype (size) Check ( <condition>),
….
ColumnN          datatype (size)  );
```

Example :

If we want to restrict the user to enter only either of the two characters 'M' or 'F' or 'm' or 'f' for the column Sex , then we can use check constraint. Here is the example

```
SQL > Create  table  student  (
        StudentId               number(2) ,
        StudentName             varchar2(15) Not null,
        Sex                     char(1)  Check (sex='M' or sex='F' or sex='m' or sex='f'),
        DOB                     date,
        Marks                   Number(3)   );
```

OR We can also include check constraint by using IN operator in the following way.

```
        Sex                     char(1)  Check (sex IN ('M', 'F', 'm', 'f')),
```

How to add a NOT NULL constraint to the table : ?

Syntax-1 : To create the table with a Not null constraint at column level :

```
Create  Table  < table name >  (
---
Column2          datatype (size) Not null;
….
);
```

Example :

```
SQL > Create  table  student  (
        StudentId               number(2) ,
        StudentName             varchar2(15) Not null,
        ------
        );
```

How to add a NULL constraint to the table : ?

By default, other than primary key, other columns are of type NULL type. i.e. they can accept null values. No explicit command is required include NULL constraint to any of the columns.

3. Referential Integrity Constraints : (Foreign key constraint)

- A foreign key is a column or combination of columns used to establish and enforce a relationship between the data in two tables.
- This relationship is created by adding a column(s) in one of the tables to refer to the other table's column(s) protected by PRIMARY KEY or UNIQUE constraint.
- This column becomes a foreign key in the first table.
- A foreign key can be created by defining a FOREIGN KEY constraint when creating or altering a table.
- It is used to protect referential integrity across tables.

[Note : For a table to enforce a foreign key constraint on a set of column(s), the referenced table should have the primary key defined for it already.]

How to add a Foreign key to the table : ?

Syntax-1 : To create the table with a column level Foreign key Constraint :

```
Create  Table  < table name >  (
Column1         datatype (size) ,
Column2         datatype (size) references table2(primary key column),
….    );
```

Here, table2(primary key) represents the table to which this column is to be referred and its data type, size should match.

Example : In the following example, table Class is considered as master table and table Student is considered as child table. The column ClassId is primary key in Class table, and same column acts as foreign key in student table.

Master table : Class

```
SQL > Create table class (
        ClassId  Number(2)  Primary key,
        ClassName       varchar2(15)  );
```

Child table : Student

```
SQL > Create  table  student  (
        StudentId                   number(2) ,
        StudentName                 varchar2(15) ,
        Sex                         char(1),
        DOB                         date,
        Marks                       Number(3),
        Classid          references  class (classid)  );
```

Syntax-2 : To create the table with a Foreign key (Standard format) by giving user defined name :

```
Create  Table  < table name >  (
Column1         datatype (size),
….
ColumnN         datatype (size),
Constraint <name of the constraint> Foreign key (column name) references table(primary key
column)  );
```

Example :
```
SQL > Create  table  student  (
          StudentId                 number(2)
          StudentName               varchar2(15),
          Sex                       char(1),
          DOB                       date,
          Marks                     Number(3),
          ClassId                   number(2),
                   Constraint  Fk_student_ClassId  Foreign key (ClassId) references Class (classid)   );
```

How to create table from another table (Source or Base table) : ?

It copies the original structure along with records of the base table to the new table

Syntax :
```
Create table <tab-name> as select <column list> or  * from < base table name>;
```
Example :
```
SQL > Create table NewStudent as select * from student;
```

How to create table from another table without any records : ?

Syntax :
```
Create table <tab-name> as select <column list> or * from < base table name> where <Un matched condition>;
```

Example :
```
SQL > Create table NewStudent as select * from student  Where  1=2;
```

ALTER Command :

This can be used to change the structure of the table :

```
Syntax :      Alter table  <table name>  Add / modify / drop / rename / disable / enable
                   <Column name [datatype(size)]> ;
```

a. Alter command with ADD option :

This can be used add new columns or constraints to the database object.

Syntax : Alter table <table-name> add <new-column-name datatype (size);

How to add one new column to the existing table : ?
```
SQL > Alter table student add address varchar2(20);
```

How to add more than one columns to the existing table : ?
```
SQL > Alter table student add (address1 varchar2(20), address2 varchar2(20), address3 varchar2(20));
```

How to add Constraints to the existing table :?

Primary key :
```
SQL >  Alter table student add constraint pk_student_studentid Primary key (StudentId);
```
Here, pk_student_studentid is the user defined name given to the primary key.

OR

SQL > Alter table student add Primary key (StudentId);

Foreign key :

SQL > Alter table student add constraint Fk_student_Classid Foreign key (ClassId)
References class (classId);

b. Alter command with Modify option : This can be used to change the datatype and size of the columns.

Syntax : Alter table <table-name> modify <existing column-name> <new datatype><size> ;

Example : SQL> Alter table student Modify sex varchar2(10);

Conditions to decrease the size / change datatype of the columns :

Note : Delete existing records of the table, then modify the column size / datatype. This is not required when we increase the size of the columns

c. Alter command with Drop option : This can be used to remove columns . constraints from the table.

i) To drop one column at a time
Syntax : Alter table <table-name> drop column <existing column-name> ;
Example : SQL > Alter table student drop column address;

ii) To drop more than one columns at a time
Syntax : Alter table <table-name> drop (col1, col2,col3…) ;
Example : SQL > Alter table student drop (address1, address2, address3);

iii) To drop constraints from a table:
Syntax : Alter table <table-name> drop constraint < constraint name>;
Example : i) SQL > Alter table student drop primary key;
ii) SQL > Alter table student drop constraint pk_student_studentId;

d. Alter command with Rename option : This can be used to rename the existing column name to the new name.

Syntax : Alter table <table-name> rename <old column name> To <New column-name> ;
Example : SQL > Alter table student Rename sex to Gender;

e. Alter command with Disable option : This can be used to disable existing constraints from the table :

Syntax : Alter table <table-name> disable constraint <constraint-name>;
Example : SQL > Alter table student disable constraint pk_student_studentid;
SQL > Alter table student disable primary key;

f. Alter command with Enable option : This can be used to enable already disabled constraints to the table.

Syntax : Alter table <table-name> enable constraint <constraint-name>;
Example : SQL > Alter table student enable constraint pk_student_studentid;

SQL > Alter table student enable primary key;

Note : While enabling primary key, the key will search for the unique criteria among all the values of the column…if it violates, it will give you an error. To overcome this, first delete all the values from the table, then enable the primary key.

DROP command :

This can be used to remove the database objects permanently from the database. When once the object is dropped from the database, we cannot recover it back to the database.

Dropping a table :

- Drops the table from the schema
    - Once dropped can not regain the table again
- All the data stored along with data structure will be lost
- All related constraints defined on the table are dropped
- All the indexes defined on the table will also be dropped

Syntax : drop object type <object-name>;

Example : SQL > drop table student;

TRUNCATE Command :

- Truncate is a DDL command and hence no rollback is possible on it
- Removes all the rows in a table and releases the storage space immediately for other purposes (against delete all rows and commit)
- You should be owner of the table or should have delete table privilege to execute
- truncate table command

Note : Truncate will NOT remove the structure, constraints, triggers defined on a table. Only rows will be zapped off permanently.

Syntax : Truncate table <table-name>;

Example : SQL > Truncate table student;

Points to remember for DDL commands :

- DDL commands are auto committed ones, hence no rollback is possible
- Present data in the table would get validated against any newly defined/enabled constraint
- Truncate releases the storage space

Differences between DROP and TRUNCATE commands:

Drop :

- Dropping a table will invalidate the dependent objects, but truncate does not
- Dropping a table drops all the related constraints, triggers and indexes of the table and hence everything has to be recreated when needed.

Truncate :

- Truncate does keep all the dependent objects intact and valid but releasing the storage space.
- Therefore avoids unwanted recompilation /validation of dependent objects

Points to remember :

- DDL commands are auto committed ones, hence no rollback is possible
- Present data in the table would get validated against any newly defined/enabled constraint
- Truncate releases the storage space

DML (Data Manipulation Language) Commands :

These commands will do manipulation with the database. i.e. Insertion, updation and deletion Commands are INSERT, UPDATE, DELETE, SELECT. These commands are explicit commit statements. i.e. it is necessary to give commit statement explicitly after their execution. These statements will be supported by ROLLBACK statement.

INSERT command : This can be used to inserting the values in to the table.

Syntax-1 : SQL > Insert into <table-name > [< column list>] values (<value list>);

Here, order of the columns in the column list should match order of the values of the value list.

Example : SQL > Insert into student (StudentId, Studentname, Sex, dob, Marks)
Values (1,'Amar','M','12-jan-1998',780);

Syntax-2 : SQL > Insert into <table-name > values (<value list>);

Here, order of the values of the value list should match the existing order of the columns of the table and we need to necessarily enter the values for all the columns.

Example : SQL > Insert into student Values (1,'Amar','M','12-jan-1998',780);

Syntax-3 : This is an interactive type which accept the values from the user
SQL > Insert into <table-name > < column list> values (<value list preceded by '&' symbol>);

Example : SQL > Insert into student values (&StudentId, '&StudentName', '&Sex', '&dob',
&Marks);

INSERT with Select statement :

This can be used to inserting the values in to the table by copying from some other table.

Syntax-1 : SQL > Insert into <table-name > [< column list>] Select statement;

Example : SQL > Insert into Newstudent select * from student;
OR
SQL > Insert into Newstudent (studentid, studentname, marks) select studentid, studentname, marks from student;

INSERT with select statement with where condition :

Example : SQL > Insert into Newstudent (studentid, studentname, marks)
Select studentid, studentname, marks from student
where marks > 500;

UPDATE Command : This can be used to change the existing values of the table :

- Updates set of column(s) in (by default) all the rows of the table
- When a 'where' clause is specified only rows that satisfy the where clause are updated
- It is part of the ongoing transaction starts a new transaction if given as first statement

Syntax : Update <table-name> SET <column-name1=New value, [column-name2=new value,...>
[Where <condition>];

Note : If we specify the condition for the where clause, then updation happens to only the records which will meet the condition criteria, otherwise updation happens to all the records of the table.

To Change the marks of a student to 900 and new name as Amarkulkarni for the studentid = 1;
Example : SQL > Update student set studentname='Amar Kulkarni', Marks=900
Where studentid=1

To change the marks of every student to 900:

SQL > Update student set Marks=900;

UPDATE with sub queries : This can be used with sub queries both in set clause and where clause.

Syntax : update <table name> set <column name1 = (sub query1),
<column name2 = (sub query2)...
Where < condition>

Example : i) To change the salary of the employee 'SMITH' as same as the another employee 'MILLER'

SQL> update emp set sal=(select sal from emp where ename='MILLER') where ename='SMITH';

Example : ii) To change the salary of the employee 'SMITH' as highest in the company and designation as same as employee 'KING'

```
SQL> update emp set sal=(select max(sal) from emp),
                        Job=(select job from emp where ename='KING')
         Where ename='SMITH';
```

DELETE Command : This can be used to remove the existing values of the table :

- Deletes row(s) in a table
- Without Where clause all rows of the table will be deleted
- Part of a transaction

- It will be supported from rollback statement.

Syntax-1 : Delete from <table-name> [Where <condition>];
OR
Syntax-2 : Delete <table-name> [Where <condition>];

Example : i) To remove the student details of studentid=1 :

SQL > Delete from student where studentid=1; OR SQL > Delete student where studentid=1;

ii) To remove all the employees with designation 'CLERK' from emp table :

SQL > Delete from emp where job='CLERK ;

iii) To remove all the employees from emp table :

SQL > Delete from emp; OR SQL > delete emp; (Removes all the rows)

Transaction Management : (TCL : Transaction Control Language commands)

What is a Transaction in oracle?

– Any set of DML commands ending with a commit
– A Single DDL/DCL command execution

When does transaction start?

– When a first DML operation is executed, the transaction starts.

Transaction, by definition of RDBMS has a unique starting point and an ending point. Transaction is specific to a particular session.

Performing a Transaction :

- Start a session
- Start doing DML operation(s) on one or many tables
- When all necessary changes are made use COMMIT to end the transaction
- When some changes have to be undone, use ROLLBACK

Commit command :

- Commit makes the changes made by the transaction to database, permanent.
- Once Committed, transaction can not be undone. (database can not be taken back to previous state)
- All the locks acquired on the rows in different table(s) will be released
- All the save points created in the transaction will be removed

Rollback Command :

- ROLLBACK commands oracle to take the database to the previous committed state before start of the current transaction
- i.e. Undo the transaction changes made.
- All the acquired locks on rows of different tables will be released on rollback

Data Control commands :

DCL commands are used to enforce database security in a multiple user database environment. Two types of DCL commands are GRANT and REVOKE. Only Database Administrator's or owner's of the database object can provide/remove privileges on a database object.

Grant command : It used to provide access or privileges on the database objects to the users.

The Syntax for the GRANT command is:

```
GRANT privilege_name
ON object_name
TO {user_name |PUBLIC |role_name}
[WITH GRANT OPTION];
```

- privilege_name is the access right or privilege granted to the user. Some of the access rights are ALL, EXECUTE, and SELECT.
- object_name is the name of an database object like TABLE, VIEW, STORED PROC and SEQUENCE.
- user_name is the name of the user to whom an access right is being granted.
- PUBLIC is used to grant access rights to all users.
- ROLES are a set of privileges grouped together.
- WITH GRANT OPTION - allows a user to grant access rights to other users.

For Example: SQL> GRANT SELECT ON employee TO user1;
This command grants a SELECT permission on employee table to user1.

You should use the WITH GRANT option carefully because for example if you GRANT SELECT privilege on employee table to user1 using the WITH GRANT option, then user1 can GRANT SELECT privilege on employee table to another user, such as user2 etc. Later, if you REVOKE the SELECT privilege on employee from user1, still user2 will have SELECT privilege on employee table.

Revoke command : The REVOKE command removes user access rights or privileges to the database objects.

The Syntax for the REVOKE command is:

```
REVOKE  privilege_name
ON  object_name
FROM  {user_name |PUBLIC |role_name}
```

For Example: SQL> REVOKE SELECT ON employee FROM user1;

This command will REVOKE a SELECT privilege on employee table from user1.

When you REVOKE SELECT privilege on a table from a user, the user will not be able to SELECT data from that table anymore. However, if the user has received SELECT privileges on that table from more than one users, he/she can SELECT from that table until everyone who granted the permission revokes it. You cannot REVOKE privileges if they were not initially granted by you.

Privileges and Roles:

Privileges: Privileges defines the access rights provided to a user on a database object. There are two types of privileges.

System privileges - This allows the user to CREATE, ALTER, or DROP database objects.
Object privileges - This allows the user to EXECUTE, SELECT, INSERT, UPDATE, or DELETE data from database objects to which the privileges apply.

Roles: Roles are a collection of privileges or access rights. When there are many users in a database it becomes difficult to grant or revoke privileges to users. Therefore, if you define roles, you can grant or revoke privileges to users, thereby automatically granting or revoking privileges. You can either create Roles or use the system roles pre-defined by oracle.

Other database objects :

Index :

- Index is a faster routing/searching mechanism to reach the actual location of (row of) data
- Indexes, in general, does not change the physical order of rows in a table rather facilitates faster reach to the location using application/domain specific data value based searching
- Brings the data organization/searching closer to the domain

Using an Index :

- There is, in general, no control in the user's hand to dictate oracle to make use of index for a query execution
- Wherever in a query the where clause is based on indexed columns, Oracle query optimizer better decides when/where to make use of an index
- User's scope is only to create the index and leave it to oracle to maintain and use it

Index guidelines :

- Unnecessary creation of index will penalize the performance of DML operations for no cause
- Too many indexes (if it can be avoided) on a table undergoing high rate of transaction will pull down the performance
- Good practice is to avoid updating primary key column(s) of a table rather delete and insert a new row

Views :

A VIEW is a virtual table, through which a selective portion of the data from one or more tables can be seen. Views do not contain data of their own. They are used to restrict access to the database or to hide data complexity. A view is stored as a SELECT statement in the database. DML operations on a view like INSERT, UPDATE, DELETE affects the data in the original table upon which the view is based.

The Syntax to create a view is

```
CREATE VIEW view_name
AS
SELECT column_list
FROM table_name [WHERE condition];
```

Here, view_name is the name of the VIEW. The SELECT statement is used to define the columns and rows that you want to display in the view.

Example : i) Create view to display employee details those who are working in deptno 30 from the table emp of scott schema.

```
SQL> Create view Emp_view1
        as select * from emp
        Where deptno=30;
```

Synonyms :

Synonyms are just place holders representing actual schema objects, but facilitate data independence fairly. Irrespective of where the data is located , which could be within the schema/database/some other database server thru db link etc., application can refer to the target object thru synonym and application need not be touched to reflect the actual name/location of the db object which may subject to change in its storage parameters

How to create synonym ?

Syntax : Create synonym < synonym name > for < existing database object-name> ;

Example : SQL> Create synonym accounts for scott.salgrade;

Sequence :

- Any table would ideally have a set of primary key columns defined which would have unique values across all the rows of the table
- Without sequence, in a typical application, same code has to be repeated to generate the next primary key value from the system
- Though possible, it does not guarantee uniqueness of document numbers generated in a shared environment
- Sequence is an oracle shared object, which generates next unique number from it, guaranteeing the uniqueness, every single time it is demanded for
- Numbers generated by sequence can not be rolled back
- Once generated for any reason, same number will not be generated for further requests from same or other sessions

Syntax : create SEQUENCE <SEQUENCE-NAME> START WITH < initial value > INCREMENT BY < incremental value> ;

Example : SQL > Create sequence new_seq start with 1 increment by 1;

To get current value of sequence

```
SQL > Select seq-name.CurrVal from dual;
```

To get Next value of sequence

```
SQL > Select seq-name.NextVal from dual;
```

Note : When we sequence for the first time, to get current value, first we should call by seq-name.nextval then followed by seq-name.currval

Examination – January 2010 : Consider the following tables with their attributes :

Staff (StaffId, StaffName, JoinDate, Designation, Salary, BranchId)
Branch (BranchId, BranchName, Manager)

Write the SQL Queries for the following requirements

i)	Display the staff details who are working to a particular branch.	1
ii)	Display the staff details who are drawing the salary in the range of 20000 to 30000.	1
iii)	Display the staff name whose name starts with 'R' as first character and 'A' as he 3rd character	1
iv)	Display the staff details who are working with a particular designation	1
v)	Display the Staff name, designation, salary and Branch name for all the staff members.	2
vi)	Display the Staff name, designation, salary and Branch name who are working under a particular manager.	2
vii)	Display the Staff details who is drawing lowest salary in a particular branch.	2

Here are the SQL queries for the above requirements :

i) Display the staff details who are working to a particular branch.

```
SQL >  select * from staff  where BranchId= < value of Branchid> ;
```

ii) Display the staff details who are drawing the salary in the range of 20000 to 30000.

```
SQL >  select * from staff  where  salary  between  20000  AND  30000 ;
Or
SQL> Select * from staff where salary >= 20000 and salary <= 30000;
```

iii) Display the staff name whose name starts with 'R' as first character and 'A' as the 3rd character

```
SQL >  select  StaffName  from  staff   where  staffname like ' R_A%' ;
```

iv) Display the staff details who are working with a particular designation

```
SQL >  select * from staff where  designation = < value of  designation >;
```

v) Display the Staff name, designation, salary and Branch name for all the staff members.

```
SQL >  select  Staffname, Designation, Salary, BranchName
       from staff, Branch
       where  Staff.BranchId = Branch.BranchId ;
```

vi) Display the Staff name, designation, salary and Branch name who are working under a particular manager.

```
SQL >  Select  StaffName, Designation, Salary, BranchName
       from staff, Branch
      where  Staff.BranchId = Branch.BranchId
      AND Manager =  <'Name of Manager ' >
```

vii) Display the Staff details who is drawing lowest salary in a particular branch.

```
SQL > Select * from Staff
        Where BranchId= <Id of a particular Branch>
         AND Salary = (Select Min(salary)
                         From Staff where BranchId= < Id of a particular Branch>)
```

Examination – December 2011 : Consider the following tables with their attributes :

Student (StudentNo, StudentName, class, dob, marks, BranchNo)

Branch (BranchNo, BranchName)

Write the Queries in relational algebraic notations for the following requirements

i)	Display the student details whose marks greater than 400.	1
ii)	Display the Student details who is studying in a particular class.	1
iii)	Display the studentno, studentName, class, branchname for all the students.	2

Write the SQL Queries for the following requirements.

iv)	Display the student name in which it starts with 'A' as the first character and 'V' as the 3rd character	1
v)	Display the student name who has scored highest marks.	1
vi)	Display class and number of students belong to each class.	2
vii)	Display the studentno, studentname, class, branchname for a particular branch.	2

Here are the Relational Algebraic notations for SQL statements for the above requirements:

First try to write the queries in SQL syntax :

i) Display the student details whose marks greater than 400.

```
SQL> Select * from Student where Marks > 400 ;
```

The above SQL query is a combination of Projection and Selection. It can be written as

R1 = $\pi_{< StudentNo, StudentName, Class, Dob, Marks, BranchNo >}$(Student)

R2=$\sigma_{< Marks > 400 >}$(Student)

The resultant R = R1(R2)

OR We can also write R1 and R2 by combining in the following way

R = $\pi_{< StudentNo, StudentName, Class, Dob, Marks, BranchNo >}(\sigma_{< Marks > 400 >}$(Student))

ii) Display the Student details who is studying in a particular class.

```
SQL > Select * from student where BranchId=< value of a particular Branch>;
```

The above SQL query is a combination of Projection and Selection. It can be written as

R1 = $\pi_{< StudentNo, StudentName, Class, Dob, Marks, BranchNo >}$(Student)

R2= σ < BranchId = Value of a particular Branch > (Student)

The resultant R = R1(R2)

OR We can also write R1 and R2 by combining in the following way

R = π < StudentNo, StudentName, Class, Dob, Marks… > (σ < BranchId = Value of a particular Branch> (Student))

iii) Display the StudentNo, StudentName, class, branchname for all the students.

The above SQL query is a combination of Projection and Join. It can be written as

R1 = π < StudentNo, StudentName, Class, BranchName > (Student, Branch)

R2 = Student ∞ < BranchNo=BranchNo > Branch

The resultant R = R1(R2)

OR We can also write R1 and R2 by combining in the following way

R= π < StudentNo, StudentName, Class, BranchName > (Student ∞ < BranchNo=BranchNo > Branch)

Here are the SQL queries for the above requirements :

iv) Display the student name in which it starts with ‘A’ as the first character and ‘V’ as the 3rd character

```
SQL> select studentName from student  Where studentName like ‘A_V%’;
```

v) Display the student name who has scored highest marks.

```
SQL> select studentName from student
        Where marks = (select max(marks) from student );
```

vi) Display class and number of students belong to each class.

```
SQL> Select class, count(class) as NoofStudents
        From  student
        Group by class;
```

vii) Display the stdno, stdname, class, branchname for a particular branch.

```
SQL > Select StudentNo, StudentName, Class, Branchname
          From Student, Branch
          Where Student.BranchId=Branch.BranchId
          AND Branchname=< name of a particular branch> ;
```

Unit – IV. Database Design Theory and Methodology.

Each relation schema consists of a number of attributes and the relational database schema consists of a number of relation schemas. The attributes are grouped to form a relation schema by using the common sense of database designer or by mapping a database schema design from a conceptual data model such as ER model. These models make the designer to identify entity types and relationship types and their respective attributes. However, we still need some formal measure of why one grouping of attributes into a relation schema may be better than another. We have not developed any measure of appropriateness of goodness to measure the quality of the design.

There are two levels at which we can measure the goodness of a relation schemas.

i) Logical level : Here users interpret the relation schemas and the meaning of their attributes. Having good relation schemas at this level enables users to understand clearly the meaning of the data in the relations and hence to formulate their queries correctly.

ii) Implementation level : Here it explains regarding how the tuples/records in a base relation are stored and updated. This level applies only to schemas of base relations which will be physically stored as files.

Database design may be performed using two approaches : bottom-up and top-down.

a) Bottom-up approach (Design by synthesis) : It considers the basic relationships among individual attributes as the starting point and uses those to construct relation schemas. This approach is not very popular in practice, because it suffers from the problem of having to collect a large number of binary relationships among attributes as the starting point.

b) Top-Down approach (Design by analysis) : It starts with a number of groupings of attributes into relations that exist together naturally. The relations are then analyzed individually and collectively, leading to further decomposition until all desirable properties are met.

Problems caused by Redundancy :

Redundancy means storing the same data more than once in a database. i.e. repetitive storage of the same data. This lead to lot of problems. This is due to the deficiency in the table design or usage of the un-normalized tables. Technically we call them as anomalies.

Consider the following table with the name Emp_Dept: This is an un-normalized table in which it contains combined data of two entities, that is employee and department. If we try to make data manipulation to this table, we will be facing the problems during insertion, update and deletion. Such problems we called as insertion anomalies, update anomalies and deletion anomalies.

Table Name : Emp_Dept

EmpNo	EmpName	Designation	Salary	DeptNo	DeptName	Location
E1	Arjun	Operator	10,000	D1	Production	Bangalore
E2	Avinash	Supervisor	12,000	D1	Production	Bangalore
E3	Arvind	Operator	10,000	D1	Production	Bangalore
E4	Bhaskar	Designer	10,000	D2	Design	Chennai
E5	Bharath	Supervisor	12,500	D2	Design	Chennai
E6	Dhanush	Manager	16,000	D2	Design	Chennai
E7	Kavya	Manager	16,000	D1	Production	Bangalore
E8	Kiran	Operator	10,000	D1	Production	Bangalore
E9	Varun	Designer	12,500	D2	Design	Chennai
E10	Shekar	Designer	12,500	D2	Design	Chennai

Insertion anomalies :

When we try to insert the new employee data in to this table, the data of the department will be repeated. Because employee is working in any one of the departments. It may not be possible to store certain information unless some other, unrelated, information is stored as well.

Update anomalies :

If one copy of such repeated data is updated, an inconsistency is created unless all copies are similarly updated. Whenever Employee Arjun changes from one department to another department, then it needs updation in the table. When we go for updating the department change, then previous history of working data in that department will be lost. In another case, assume the department name ‘Design’ changes to ‘R&D’, then, when we go for changing the name of the department, it leads to inconsistency of the data, because in some records it will be displayed as R&D and in some records it is still displayed as ‘Design’ it self which in turn results an ambiguity with respect to the department name with DeptNo ‘D2’.

Delete anomalies :

It may not be possible to delete certain information without losing some other, unrelated, information as well. Assume that the department ‘Design’ has to be removed from the table, When we do so, then there may be chances of losing the data of the employees (Bhaskar, Bharath, Dhanush) who are working in that department but that is not of our requirement. We want to retain their data in a table.

Informal Design Guidelines for Relational Schemas :

i) Semantics of the attributes
ii) Reducing the redundant information in tuples
iii) Reducing the NULL values in tuples
iv) Disallowing the possibility of generating spurious tuples.

i) Semantics of the attributes : When ever we are going to form a relational schema, there should be some meaning among the attributes. This meaning is called semantics. This semantics relates one attribute to another with some relation. The semantics of a relation refers to the interpretation of attribute values in a tuple.

Guidelines :

- Design a relation schema so that it is easy to explain its meaning and proper interpretation associated with them.
- Do not combine attributes from multiple entity types and relationship types into a single relation.
- If a relation schema corresponds to one entity type or one relationship type, it is straight forward to interpret and to explain its meaning. Otherwise, if the relation corresponds to a mixture of multiple entities and relationships, semantic ambiguities will result and the relation cannot be easily explained.

ii) Reducing the redundant Information in Tuples and Update Anomalies : One goal of schema design is to minimize the storage space used by the base relations. Grouping attributes into relation schemas has a significant effect on storage space. The serious problems we are facing with this redundant information are Insertion anomalies, Update anomalies and Deletion anomalies.

Guidelines :

- Design the base relation schema so that no insertion, deletion or modification anomalies are present in the relations.
- If any anomalies are present , note them clearly and make sure that the programs that update the database will operate correctly.
- Make sure that data of the relation should be consistent over the period of the time. We can also see the need for a more formal approach to evaluating whether a design meets these guidelines.

In general, it is advisable to use anomaly – free base relations and to specify views that include the joins for placing together the attributes frequently referenced in important queries. This reduces the number of JOIN terms specified in the query and making it simpler to write the query correctly, and in many cases it improves the performance.

iii) NULL values in Tuples : In a given relations, many times during manipulation, most of the attributes are left with blank spaces (no values entered). These attributes will end up with a value called NULL value (It is an empty value). This can waste space at the storage level and may also lead to problems with understanding the meaning of the attributes. Another problem is during the usage of all aggregate functions. Moreover, NULLs can have multiple interpretations which are as follows :

- The attribute does not apply to this tuple
- The attribute value for this tuple is unknown
- The value is unknown but absent: i.e. it has not been recovered yet.

Guidelines :

- As far as possible, avoid placing attributes in a base relation whose values may frequently be NULL.
- If NULLs are unavoidable, make sure that they may applied with proper exceptional cases and do not apply to a majority of tuples in the relation.

iv) Generation of Spurious Tuples : A spurious tuple is, basically, a record in a database that gets created when two tables are joined badly. In database, spurious tuples are created when two tables are joined on attributes that are neither primary keys nor foreign keys.

Guidelines :

- Design relation schemas so that they can be joined with equality conditions on attributes that are (primary key, foreign key) pairs in way that guarantees that no spurious tuples are generated.
- Avoid relations that contain matching attributes that are not (primary key, foreign key) combinations because joining on such attributes may produce spurious tuples.
- The "LossLess Join" property is used to guarantee meaningful results for join operations.

Functional Dependencies : It is a constraint between two sets of attributes from the database. A relation R in a database schema has n attributes A_1, A_2, …. A_n can be described as Universal relation schema R = { A_1,A_2, …., A_n} .

Definition : A functional dependency, denoted by X → Y, between two sets of attributes X and Y that are subsets of R specifies a constraint on the possible tuples that can form a relation state r of R. The constraint is that, for any two tuples t_1 and t_2 in r that $t_1[X] = t_2[X]$,

they must also have $t_1[Y] = t_2[Y]$.

This means that the values of the Y component of a tuple in r depend on, or are determined by the values of the X component. Alternatively, the values of the X component of a tuple uniquely determine the values of the Y component. We also say that there is a functional dependency from X to Y, or that Y is functionally dependent on X.

The abbreviation for functional dependency is FD or f.d. The set of attributes X is called the left-hand side of the FD and Y is called the right-hand side of the FD.

a) Functional dependencies (FDs) are used to specify formal measures of the "goodness" of relational designs

b) FDs and keys are used to define normal forms for relations

c) FDs are constraints that are derived from the meaning and interrelationships of the data attributes

d) X → Y : A set of attributes X functionally determines a set of attributes Y if the value of X determines a unique value for Y

e) X → Y holds if whenever two tuples have the same value for X, they must have the same value for Y

f) For any two tuples t1 and t2 in any relation instance r(R): If t1[X] = t2[X], then t1[Y] = t2[Y]

g) X → Y in R specifies a constraint on all relation instances r(R)

h) Written as X → Y: can be displayed graphically on a relation schema as in Figures. (denoted by the arrow:).

i) FDs are derived from the real-world constraints on the attributes

j) Social security number determines employee name SSN → ENAME

k) Project number determines project name and location PNUMBER → {PNAME, PLOCATION}

Types of Functional Dependencies :

i) Full functional dependencies
ii) Partial functional dependencies
iii) Transitive dependencies

i) Full Functional dependencies : An attribute B of a relation R is fully functionally dependent on attribute A of R if it is functionally dependent on A and not functionally dependent on any proper subset of A.

Example : Consider a relation with the name 'Report' which contains the following attributes

Report {<u>StudentNo,CourseNo,</u> StudentName, CourseName, Marks, Grade}

Here, <u>StudentNo,CourseNo</u> are the Prime attributes, the attribute Marks is fully functionally dependent on both of the prime attributes StudentNo and CourseNo. It means that, a student has scored some marks in a particular course. It is denoted as

StudentNo,CourseNo → Marks

ii) Partial Functional dependencies :

An attribute B of a relation R is partially dependent on attribute A of R if it is functionally dependent on any proper subset of A.

Report {StudentNo,CourseNo, StudentName, CourseName, Marks, Grade}

Here, the attribute StudentName is fully functionally dependent on the attribute StudentNo but not on CourseNo, and CourseName is fully functionally dependent on CourseNo but not on StudentNo. Therefore, StudentName is partially dependent on StudentNo,CourseNo and similarly, CourseName is partially dependent on StudentNo,CourseNo. It is denoted as

StudentNo → StudentName

CourseNo → CourseName

and not as StudentNo,CourseNo → StudentName OR StudentNo,CourseNo → CourseName

iii) Transitive dependencies :

An attribute B of a relation R is transitively dependent on attribute A of R if it is functionally dependent on an attribute C which in turn is functionally dependent on A or any proper subset of A.

Report {StudentNo,CourseNo, StudentName, CourseName, Marks, Grade}

Here, the attribute Grade is said to be transitively dependent on the key attribute StudentNo,CourseNo. Since an attribute Grade is functionally dependent on attribute Marks which in turn fully dependent on StudentNo,CourseNo. It is denoted as

StudentNo,CourseNo → Marks

Marks → Grade

This results in StudentNo,CourseNo → Marks → Grade

It means that Grade is determined by marks which in turn dependent on StudentNo,CourseNo, So, Grade is transitively dependent on StudentNo,CourseNo

Inference rules for FDs: (Armstrong's Inference Rules or Axioms)

We denote the set of functional dependencies by F specified on relation schema R. Typically, the schema designer specifies the functional dependencies that are semantically obvious. However, other FDs hold in all legal relation instances among set of attributes that can be derived from and satisfy the dependencies in F. Those other dependencies can be inferred or deduced from the FDs in F. Formally it is useful to define a concept called CLOSURE that includes all possible dependencies that can be inferred from the given set F.

Definition : Formally, the set of all dependencies that include F as well as all dependencies that can be inferred from F is called the CLOSURE of F, it is denoted by F^+ . To determine a systematic way to infer dependencies , we must discover a set of inference rules that can be used to infer new dependencies from a given set of dependencies. Inference rules also known as Armstrong's Axioms are published by Armstrong. These properties are as given below:

1. Reflexivity property: X → Y is true if Y is subset of X.

Proof : Suppose that X is a superset of Y and
that two tuples t_1 and t_2 exist in some relation instance r of R
such that $t_1[X] = t_2[X]$.
Then $t_1[Y] = t_2[Y]$, Because X Y ; hence X → Y must hold in r.

2. Augmentation property: (By Contradiction) : If X → Y is true, then XZ → YZ is also true.

Proof : Assume that X → Y holds in a relation instance r of R
but that XZ → YZ does not hold. Then there must exist two tuples t1 and t2 in r
such that (1) $t_1[X] = t_2[X]$,
(2) $t_1[Y] = t_2[Y]$,
(3) $t_1[XZ] = t_2[XZ]$, and (4) $t_1[YZ] \neq t_2[YZ]$. This is not possible
from (1) and (3) we deduce (5) $t_1[Z] = t_2[Z]$, and
from (2) and (5) we deduce (6) $t_1[YZ] = t_2[YZ]$, contradicting (4)

3. Transitivity property: If X → Y and Y → Z then X → Z is implied.

Proof : Assume that (1) X → Y and (2) Y → Z both hold in a relation r.
Then for any two tuples t1 and t2 in r such that $t_1[X] = t_2[X]$,
we must have (3) $t_1[Y] = t_2[Y]$, from assumption (1);
hence, we must also have (4) $t_1[Z] = t_2[Z]$, from (3) and assumption (2);
hence , X → Z must hold in r.

4. Union property: If X → Y and X → Z are true, then X → YZ is also true. This property

indicates that if right hand side of FD contains many attributes then FD exists for each of them.

Proof : X → YZ (given)

YZ → Y (using Reflexivity property and knowing that YZ is a subset of Y)

X → Y (using Transitivity property on Reflexive property and Augmentation property)

5. Decomposition property: If X → Y is implied and Z is subset of Y, then X → Z is implied. This property is the reverse of union property.

Proof : X → Y (given)

X → Z (given)

X → XY (using Augmentation property on Reflexivity property by augmenting with X)

XY → YZ (using Augmentation property with Y)

X → YZ (using Augmentation property and Transitive property)

6. Pseudo transitivity property: If X → Y and WY → Z are given, then XW → Z is true.

Proof : X → Y (given)

WY → Z (given)

WX →WY (using Augmentation property on Reflexivity property by augmenting with W)

WX → Z (using Transitivity property on Augmentation property)

Normalization: It is the process which proceeds in a top-down approach by evaluating each relation against the criteria for normal forms and decomposing relations as necessary and thus be considered as relational design by analysis.

In simple sense, It is the set of rules used to design the database relations. It is the process of decomposing unsatisfactory "bad" relations by breaking up their attributes into smaller relations. It was first proposed by Mr. Codd in 1972. He has taken a relation schema through a series of tests to certify whether it satisfies a certain normal form. Initially, Codd proposed three normal forms, which he called first, second and third normal form. A stronger definition of 3NF – called Boycee-Codd normal form (BCNF) was proposed later by Boycee and Codd. All these normal forms are based on a single analytical tool; i.e. the functional dependencies among the attributes of a relation. Later, a 4NF proposed based on the concept of multivalued dependencies and 5NF proposed on the basis of multi joined dependencies.

Normalization of data : It can be considered as a process of analyzing the given relation schemas based on their FDs and primary keys to achieve the desirable properties of

- minimizing redundancy
- minimizing the anomalies of insertion, update and deletion

Thus, the normalization procedure provides database designers with the following.

⊕ A formal framework for analyzing relation schemas based on their keys and on the functional dependencies among their attributes

- ⊕ A series of normal form tests that can be carried out on individual relation schemas so that the relational database can be normalized to any desired degree.

The normal form of a relation refers to the highest normal form condition that it meets and hence indicates the degree to which it has been normalized. The normal form would include two properties :

i) The Lossless join or nonadditive join property : which guarantees that the spurious tuple generation problem does not occur with respect to the relation schemas created after decomposition.

ii) The dependency preservation property : Which ensures that each functional dependency is represented in some individual relation resulting after decomposition.

Denormalization : It is the process of storing the join of higher normal form relations as a base relation which is in a lower normal form is know as denormalization.

Definition of Keys :

Super Key : A super key of a relation schema R = { A1, A2, ….. , An} is a set of attributes S R with the property that no two tuples t1 and t2 in any legal relation state r of R will have $t_1[S] = t_2[S]$. A key is a super key with the additional property that removal of any attribute from K will cause K not to be a super key any more. Basically Super key is a primary key which has to serve the purpose of a primary key. This depends on the situation.

Example : {Ssn}, {Ssn,Ename}, {Ssn,Enam,Bdate} and any set of attributes that includes Ssn are all super keys.

Candidate Keys and Secondary keys : If a relation schema has more than one key attribute, then each is called a candidate key. Basically, candidate key is a primary key. Sometimes, in a given relation, there exists more than key attributes, out of that, one of the candidate keys is arbitrarily designated to be the primary key and other keys are called as secondary keys.
Example : Employee { <u>SSN</u>, Empname, Designation, Dob, Salary, ContactNo, EmailId}

In the above relation Employee, the attribute SSN is a key attribute. In addition to this, some times we can also consider ContactNo or EmailId as alternative key attributes, because their values are unique. In this case, we can consider ContactNo or EmailId as candidate keys or Secondary keys.

1NF (First Normal Form) : It was defined to disallow multivalued attributes, composite attributes, and their combinations. It states that the domain of an attribute must include only atomic (simple, indivisible) values and that the value of any attribute in a tuple must be a single value from the domain of that attribute. In other words, 1NF disallows relations within relations or relations as attribute values within tuples. The only attribute values permitted by 1NF are single atomic (indivisible) values.

Definition : A relation schema is in 1NF if all of its attributes are

- Single-valued,
- restricted to assuming atomic values,
- functionally dependent on the primary key

1NF implies:

- Composite attributes are represented only by their component attributes
- Attributes cannot have multiple values
- Attributes cannot have complete tuples as values

Example : Consider a relation for online retail application, which can stores the details of customer, item and purchase. The customer details consist of CustomerId, CustomerName and AccountNo. The item details consist of ItemId, ItemName, unit price and item class. Similarly, the purchase details consist of number of items purchased and total amount paid by the customer.

CustomerDetails	ItemDetails	PurchaseDetails
1001 John 1500012351	STN001 Pen 10 A	5 50
1002 Tom 1200354611	BAK003 Bread 10 A	1 10
1003 Maria 2134724532	GRO001 Potato 20 B	1 20

This violates the rules of 1NF, because it has got only three attributes, each attribute is storing composite values (i.e. 1001 John 1500012351). In order to bring this relation in to 1NF, this relation has to be decomposed into a new relation which is as shown below.

CustomerId	Customer Name	AccountNo	ItemId	ItemName	UnitPrice	Class	Quantity Purchased	NetAmt
1001	John	1500012351	STN001	Pen	10	A	5	50
1002	Tom	1200354611	BAK003	Bread	10	A	1	10
1003	Maria	2134724532	GRO001	Potato	20	B	1	20

Here, customer details were stored in separate attributes CustomerId, CustomerName and AccountNo, Item details were stored in separate attributes ItemId, ItemName, UnitPrice and ItemClass, similarly, total number of items purchased and net amount paid is stored in separate attributes QtyPurchased and NetAmt.

Note : In relational database design, it is not practically possible to have a table which is not in 1NF.

2NF (Second Normal Form) : 2NF is based on the concept of full functional dependency among prime and non-prime attributes. That is every non prime attributes of a relation is fully functionally dependent on prime attribute. There should not be any partial functional dependencies among prime and non-prime attributes. If any violations take place in this regard, then we need to decompose the relation so that every decomposed relation satisfies the rules of 2NF.

Definition : A relation schema R is in 2NF if every nonprime attribute A in R is fully functionally dependent on the primary key of R.

Consider the following relation 'Report': in which it consists of the details regarding marks, grade of a student scored in a particular course.

Report { <u>StudentNo,CourseNo,</u> StudentName, CourseName, DOB, Duration, Marks, Grade}

Here prime attribute is the combined key <u>StudentNo,CourseNo</u> and StudentName, CourseName, Marks, Grade are non – prime attributes.

According to the 2NF, every non prime attribute is fully functionally dependent on prime attribute. We shall check this functional dependency between non prime attributes with the prime attribute. The non-prime attributes StudentName and DOB are not fully functionally dependent on <u>StudentNo,CourseNo,</u> but partially dependent on StudentNo, similarly CourseName and Duration, are fully dependent on CourseNo not fully on <u>StudentNo,CourseNo.</u> The attribute Marks is the only one non – prime attribute which is fully functionally dependent on <u>StudentNo,CourseNo.</u> The non-prime attribute Grade is fully functionally dependent on Marks but not either on StudentNo nor CourseNo. So, this relation 'Report' needs to be decomposed to satisfy the rules of 2NF which are as follows.

Student { <u>StudentNo,</u> StudentName, DOB}

Here, in the relation 'Student', the non-prime attributes StudentName and DOB are fully functionally dependent on prime-attribute StudentNo.

Course {<u>CourseNo,</u> CourseName, Duration}

Here, in the relation 'Course', the non-prime attributes CourseName and Duration are fully functionally dependent on prime-attribute CourseNo.

Report {<u>StudentNo,CourseNo,</u> Marks, Grade}

Here, in the relation 'Report', the non-prime attribute Marks is fully functionally dependent on prime-attribute <u>StudentNo,CourseNo</u> and one more non-prime attribute Grade is fully functionally dependent on Marks but not either on StudentNo nor on CourseNo .

3NF (Third Normal Form) :

3NF is based on the concept of transitive dependency. i.e. in a relation, there should not be any transitive dependency among prime attribute and non-prime attributes. If it is there, we need to decompose the relation to satisfy the rules of 3NF.

Definition : A relation schema R is in 3NF if it satisfies 2NF and no non-prime attribute is transitively dependent on the primary key of R.

3NF means that each non-prime attribute value in any tuple is truly dependent on the primary key and not even partially on other attributes. 3NF prohibits transitive dependencies.

Example : Consider the same relation 'Report':

Report { StudentNo,CourseNo, StudentName, CourseName, DOB, Duration, Marks, Grade}

Because of so many violations w r t 2NF, the above relation is decomposed in to 3 relations which are as follows.

Student { StudentNo, StudentName, DOB}

Course {CourseNo, CourseName, Duration}

Report {StudentNo,CourseNo, Marks, Grade}

Here, the relations Student and Course are already satisfy the rules of 2NF & 3NF. If we consider the relation 'Report', the non-prime attribute Marks is fully functionally dependent on prime-attribute StudentNo,CourseNo and one more non-prime attribute Grade is fully functionally dependent on Marks but not either on StudentNo nor on CourseNo which in turn results in transitive dependency which is against to the rule of 3NF. Therefore, the relation 'Report' is not satisfying the norms of 3NF. In order to bring the relation 'Report' to 3NF, we need to decompose the relation to further relation which is as follows:

Report { StudentNo,CourseNo,Marks}

StudentGrade { Marks, Grade},

It can also be restated as StudentGrade { InitialMarks, FinalMarks, Grade}

Here, in the relation 'Report', the non-prime attribute 'Marks' fully functionally dependent on the combined key StudentNo,CourseNo and in the relation 'StudentGrade', the non-prime attribute 'Grade' is fully functionally dependent on the attribute 'Marks'. Because, amount of marks defines the value of the grade. Usually the grade will be awarded by taking the range of marks. The attribute marks can be split in to InitialMarks and FinalMarks.

BCNF (Boycee-Codd Normal Form): This was proposed from two database experts Mr. Boycee and Mr. Codd in the year 1969. It was proposed as a simpler form of 3NF, but it was found to be stricter than 3NF. That is every relation in BCNF is also in 3NF, however, a relation in 3NF is not necessarily in BCNF. In simple sense, relation with one column primary key can satisfies the rules of 3NF, but a relation with combined column primary key cannot satisfy the rules of 3NF. This time, we bring this relation to BCNF form.

Definition : A relation schema R is in BCNF if, for every non-trivial functional dependency A→B in it, it is true that A is a super key of R. In other words, every determinant is a candidate key.

* BCNF is a stronger form of 3NF

* 3NF states that every non-prime attribute must be non-transitively dependent on every key

* BCNF states that every attribute (prime or non-prime) must be non transitively dependent on every key.

Example :

StudentReport { StudentNo, CourseNo, SubjectNo, Marks}

Here, in the relation 'StudentReport', the non-prime attribute 'Marks' is fully functionally dependent on the combined key StudentNo,CourseNo,SubjectNo. The meaning is a student has scored some marks in a subject of particular course.

EmpProject { EmpNo, ProjectNo, NumberOfHours}

Here, in the relation 'EmpProject', the non-prime attribute 'NumberOfHours' is fully functionally dependent on the combined key EmpNo,ProjectNo. The meaning is an employee has worked some number of hours in a particular project.

TeacherSubject { TeacherNo, ClassNo, SubjectNo, NumberOfHours, AcadYear}

Here, in the relation 'TeacherSubject', the non-prime attributes 'NumberOfHours' and 'AcadYear' are fully functionally dependent on the combined key TeacherNo,ClassNo,SubjectNo. The meaning is a teacher has taught a subject with some number of hours to some class in a particular academic year.

DriverDrive { DriverNo,VehicleNo, NumberOfKms, DateOfDrive}

Here, in the relation 'DriverDrive', the non-prime attributes 'NumberOfKms' and 'DateOfDrive' are fully functionally dependent on the combined key DriverNo,VehicleNo. The meaning is a driver has driven a car with some number of kilometers on particular date.

Exercises :

Consider the following relation EmpProjDept with following attributes :

EmpProjDept { EmpNo,ProjCode,DeptNo, EmpName, DeptName, ProjectName, Designation, ProjectDuration, DeptBudget, NumberOfHours, Remuneration}

Bring this relation to 3NF :

Solution : If we make an analysis about the above relation EmpProjDept, we found lot of violations w r t 2NF and 3NF. In order to bring the above relation to 2NF, we need to decompose the table in a following way. Since 2NF is mainly focusing on full functional dependency among prime attribute and non prime attributes, first of all we need to identify the number of entities present in the relation EmpProjDept.

We found, there are three entities, i.e. Employee, Project and Department. Therefore, we need to design a base table for all these entities.

Employee { <u>EmpNo,</u> EmpName, Designation }

Here, in a relation Employee, Empno is the prime attribute and Empname, designation are the non prime attributes. The non prime attributes Empname and Designation are fully functionally dependent on the prime attribute EmpNo.

Project { <u>ProjCode,</u> ProjectName, ProjectDuration }

Here, in a relation Project, ProjCode is the prime attribute and ProjectName, ProjectDuration are the non prime attributes. The non prime attributes ProjectName and ProjectDuration are fully functionally dependent on the prime attribute ProjCode.

Department { <u>DeptNo</u>, DeptName, DeptBudget }

Here, in a relation Department, DeptNo is the prime attribute and DeptName, DeptBudget are the non prime attributes. The non prime attributes DeptName and DeptBudget are fully functionally dependent on the prime attribute DeptNo.

EmpProjDept { <u>EmpNo,ProjCode,Deptno,</u> NumberOfHours, Remuneration}

Here, in a relation EmpProjDept, Empno,ProjCode,DeptNo is the prime attribute and NumberOfHours, Remuneration are the non prime attributes. The non prime attribute NumberOfHours is fully functionally dependent on the prime attribute EmpNo,ProjCode,DeptNo and one more non prime attribute Remuneration is fully functionally dependent on NumberOfHours. This results in transitive dependency which is against to the rule of 3NF. Therefore, we need to decompose this table again further in order to bring this relation to 3NF.

EmpProjDept { <u>EmpNo,ProjCode,Deptno,</u> NumberOfHours}

Here, in a relation EmpProjDept, Empno,ProjCode,DeptNo is the prime attribute and NumberOfHours is the non prime attribute. The non prime attribute NumberOfHours is fully functionally dependent on the prime attribute EmpNo,ProjCode,DeptNo.

EmpBill { NumberOfHours, Remuneration}

Here, in a relation EmpBill, the non prime attribute Remuneration is fully functionally dependent on the attribute NumberOfHours.

Now, all the relations Employee, Project, Department, EmpProjDept, and EmpBill are follows the norms of 3NF. Finally the relations are as follows.

Employee { EmpNo, EmpName, Designation }

Project { ProjCode, ProjectName, ProjectDuration }

Department { DeptNo, DeptName, DeptBudget }

EmpProjDept { EmpNo,ProjCode,Deptno, NumberOfHours}

EmpBill { NumberOfHours, Remuneration}

If we convert these relations to a table format, they look like tables in the following formats :

Table : Employee

EmpNo	EmpName	Designation
E101	AbhayChandra	Operator
E102	ArunSagar	Operator
E103	Bhaskar	Supervisor
E104	Eshwar	Manager
E105	Eshan	Supervisor
E106	Kirankumar	Operator
E107	Vinay	Operator

Table : Project

ProjCode	ProjectName	ProjectDuration
PR001	LoanLendingSystem	8 months
PR002	InsuranceBanking	3 months
PR003	CreditCardSystem	2 months
PR004	InvestmentBanking	7 months
PR005	JewellaryLoanLendingSystem	6 months

Table : Department

DeptNo	DeptName	DeptBudget
D1	Dept of Design	2,50,000
D2	Dept of Coding	10,00,000
D3	Dept of Testing	5,00,000
D4	Dept of HR	2,00,000
D5	Dept of Accounts	45,00,000

Table : EmpProjDept

EmpNo	ProjCode	DeptNo	NumberOfHours
E101	PR001	D1	120
E102	PR001	D2	75
E103	PR003	D2	80
E104	PR004	D3	55
E105	PR005	D2	125
E106	PR002	D3	110
E107	PR002	D2	130

Table : EmpBill

NumberOfHours	Renumeration
1	Rs. 500 – 00
10	Rs. 5,000 - 00
50	Rs. 25,000 - 00
100	Rs. 50,000 - 00
200	Rs. 1,00,000 - 00

De-normalization : It is the process of storing the join of higher normal form relations as a base relation which is in a lower normal form is known as de-normalization. It can be described as a process for reducing the degree of normalization with the aim of improving query processing performance. However, reducing the degree of normalization of a table may lead to inconsistencies and this option has to be dealt with careful analysis and thinking. The usefulness of de-normalization is a debatable one.

Example : Here are two normalized tables Lecturer_Department and Lecturer_Course are giving the relationship among lecturer and department, Lecturer and course. i.e. lecturer is working in which department and lecturer is dealing with which course.

Before De-normalization :

Table : Lecturer_Department

Lecturer	Department
L1	D1
L2	D1
L3	D3
L4	D2
L5	D2

Table : Lecturer_Course

Lecturer	Course
L1	C1
L2	C1
L3	C2
L4	C2
L5	C3
L1	C2
L2	C2

After de-normalization,

Table : Lecturer_Course_Department

Lecturer	Course	Department
L1	C1	D1
L2	C1	D1
L3	C2	D3
L4	C2	D2
L5	C3	D2
L1	C2	D1
L2	C2	D1

Here, we have reduced the degree of normalization by combining two separate normalized tables Lecturer_Course and Lecturer_Department in to one table Lecturer_Course_Department without violating the rules of normalization by maintaining consistency of the data.

Relational Database Design Algorithms and Further Dependencies :

Multivalued Dependencies : There are consequences of first normal form which disallows an attribute in a tuple to have set of values.

If we have two or more multivalued independent attributes in the same relation schema, we get in to the problem of having to repeat every value of one of the attributes with every value of the other attribute to keep the relation state consistent and to maintain the independence among the attributes involved. This constraint is specified by a multivalued dependency.

For example :

Table : Employee_Project

EmpName	ProjectName	DependentName	TypeOfRelation
ArunSagar	InvestmentBanking	Kiran	Son
ArunSagar	InvestmentBanking	Kruthika	Daughter
ArunSagar	InvestmentBanking	Sagar	Father
ArunSagar	InvestmentBanking	SriVidya	Wife
Bhaskar	LoanLending	Varun	Son
Bhaskar	LoanLending	Varini	Daughter
Bhaskar	LoanLending	Lakshmi	Wife

Consider the relation Employee_poject, A tuple in this relation represents the fact that an employee whose name is EmpName works on the project and has a dependent. An employee may work on several projects and may have several dependents and the employee's projects and dependents are independent of one another. To keep the relation state consistent, we must have separate tuple to represent every combination of employee's dependent and an employee's project.

Employee_Project:

EmpName	ProjectName
ArunSagar	InvestmentBanking
Bhaskar	LoanLending

Employee_Dependents:

EmpName	DependentName	TypeOfRelation
ArunSagar	Kiran	Son
ArunSagar	Kruthika	Daughter
ArunSagar	Sagar	Father
ArunSagar	SriVidya	Wife
Bhaskar	Varun	Son
Bhaskar	Varini	Daughter
Bhaskar	Lakshmi	Wife

Definition : A multivalued dependency $X \rightarrow\rightarrow Y$ specified on relation schema R, where X and Y are both subsets of R, specifies the following constraint on any relation state r of R.

Whenever $X \rightarrow\rightarrow Y$ holds, we say that X multi determines Y, because of symmetry in the definition , Whenever $X \rightarrow\rightarrow Y$ holds in R, so $X \rightarrow\rightarrow Z$

Hence , $X \rightarrow\rightarrow Y \Longrightarrow X \rightarrow\rightarrow Z$

and therefore it is sometimes written as $X \rightarrow\rightarrow Y/Z$

4NF (Fourth Normal Form) : When a relation has un desirable multivalued dependencies and hence can be used to identify and decompose such relations.

Definition : A relation schema R is in 4NF with respect to a set of dependencies F (that includes functional dependencies and multivalued dependencies) if, for every non-trivial multivalued dependency $X \rightarrow\rightarrow Y$ in F, X is a super key for R.

Example : Consider the above relation Employee_project, in which it has got the columns of dependents and Projectname in which employee is working. This violates the rules of 4NF. In order to bring this relation we need to decompose in to separate relations. i.e. Employee_Project and Employee_Dependents which are satisfying the rules of 4NF.

5NF (Fifth Normal Form) : This normal form mainly focusing on join dependencies among the relations. Fifth normal form (5NF), also known as project-join normal form (PJNF) is a level of database normalization designed to reduce redundancy in relational databases recording multi-valued facts by isolating semantically related multiple relationships.

Join Dependencies : A join dependency denoted by $JD(R_1,R_2,R_3........R_n)$ specified on relation schema R, specifies a constraint on the states r of R. The constraint states that every legal state r of R should have non additive join decomposition into $R_1, R_2, , , , , R_n$.

i.e. for every such r, we have

$$* (\pi R_1(r), \pi R_2(r), \pi R_3(r), \; , \; , \; , \; , \quad \pi R_n(r)) = r$$

Definition : A relation schema R is in 5NF (project join normal form, PJNF) with respect to a set F of functional, multivalued and join dependencies if, for every non-trivial join dependency

JD(R_1, R_2, , , , , R_n) in F^+, every R_i is a super key of R.

In other words, A table is said to be in the 5NF if and only if every non-trivial join dependency in it is implied by the candidate keys.

A join dependency *{A, B, … Z} on R is implied by the candidate key(s) of R if and only if each of A, B, …, Z is a super key for R.[1]

Example : Consider the relation SUPPLY, all key relation of supplier, part_name and Project. Whenever a supplier supplies part P, and a project J uses part P, and the supplier S supplies at least one part to project J, then supplier S will also supplying part P to project J.

Example : Table : SUPPLY

Supplier	Part_Name	Project
Smith	Bolt	Proj_X
Smith	Nut	Proj_Y
Ravi	Bolt	Proj_Y
Shekhar	Nut	Proj_Z
Ravi	Nail	Proj_X
Ravi	Bolt	Proj_X
Smith	Bolt	Proj_Y

This constraint can be restated in other ways and specifies a join dependency JD(R1, R2, R3) among three projections R1(Sname, Part_name), R2(Sname, Proj_name) and R3(Part_name, Proj_name) of SUPPLY. Following relations shows how the SUPPLY relation with the join dependency is decomposed into three relations R1, R2 and R3 that are each in 5NF. Notice that applying a natural join to any two of these relations produces spurious tuples, but applying a natural join to all three together does not give proper results.

R1

Supplier	Part_name
Smith	Bolt
Smith	Nut
Ravi	Bolt
Ravi	Nut
Shekhar	Nut

R2

Supplier	Project
Smith	Proj_X
Smith	Proj_Y
Ravi	Proj_Y
Ravi	Proj_Z
Shekhar	Proj_X

R3

Part_name	Project
Bolt	Proj_X
Nut	Proj_Y
Bolt	Proj_Y
Nut	Proj_Z
Nail	Proj_X

Unit – V. Transaction Management.

5.1 Transaction Concept

A transaction is defined as an execution of a user program, seen by the DBMS as a series of read and write operations. Executing the same program several times generates several transactions.

- A transaction is a unit of program execution that accesses and possibly updates various data items.
- A transaction must see a consistent database.
- During transaction execution the database may be inconsistent.
- When the transaction is committed, the database must be consistent.

Two main issues to deal with:

i) Failures of various kinds, such as hardware failures and system crashes
ii) Concurrent execution of multiple transactions

ACID Properties : A dbms must ensure four important properties of transactions to maintain data in the face of concurrent access and system failures. To preserve integrity of data, the database system must ensure:

Atomicity : Users should be able to regard the execution of each transaction as atomic; Either all operations of the transaction are properly reflected in the database or none are. Users should not have to worry about the effect of incomplete transactions.

Consistency: It is the property that every transaction sees a consistent database instance. Database consistency follows from transaction atomicity, isolation and transaction consistency. Execution of a transaction in isolation preserves the consistency of the database. Each transaction, run by itself with no concurrent execution of other transactions, must preserve the consistency of the database. The dbms assumes that consistency holds for each transaction.

Isolation : Users should be able to understand a transaction without considering the effect of other concurrently executing transactions, even if the DBMS interleaves the actions of several transactions for performance reasons. This property is sometimes referred to as isolation. Transactions are isolated, or protected, from the effects of concurrently scheduling other transactions. Although multiple transactions may execute concurrently, each transaction must be unaware of other concurrently executing transactions. Intermediate transaction results must be hidden from other concurrently executed transactions. That is, for every pair of transactions Ti and Tj, it appears to Ti that either Tj finished execution before Ti started, or Tj started execution after Ti finished.

Durability : Once the DBMS informs the user that a transaction has been successfully completed, its effects should persist even if the system crashes before all its changes are reflected on disk. This property is called durability. After a transaction completes successfully, the changes it has made to the database should become permanent, even if there are system failures.

Transactions and Schedules :

A transaction is seen by the DBMS as a series, or list of actions. The actions that can be executed by a transaction include reads and writes of database objects. To keep the notations as simple as such here, assume that an object ‘O’ is always read into a program variable that is also named as ‘O’.

Therefore, we can denote the action of a transaction T reading an object ‘O’ as R_T(O);

Similarly, we can denote writing an object ‘O’ as W_T (O) .

In addition to reading and writing, each transaction must specify as its final action with either commit (i.e. complete successfully) or with abort (i.e. terminated and undo all the actions carried out so far).

Abort_T denotes the action of T aborting and Commit_T denotes T committing.

There are two important assumptions :

i) Transactions interact with each other only thru database read and write operations (insert, update, delete).

ii) A database is a fixed collection of independent objects. When objects are added to or deleted from a database or there are relationships between database objects that we want to exploit for performance, some additional issues arise.

Schedule : It is a list of actions (reading, writing, aborting, or committing) from a set of transactions, and the order in which two actions of a transaction T appear in a schedule must be the same as the order in which they appear in T. In other words, a schedule represents an actual execution sequence. Following figure shows an execution order for actions of two transactions T_1 and T_2.

T1	T2
R(A)	
W(A)	
	R(B)
	W(B)
R(C)	
W(C)	

In addition to these actions, a transaction may carry out other actions such as reading or writing from operating system files, evaluating arithmetic expressions, and so on. However, these actions should not affect the other transactions.

Example :

Let T1 transfer Rs. 50 from A to B, and T2 transfer 10% of the balance from A to B. The following is a serial schedule (Schedule 1 in the text), in which T1 is followed by T2.

T1	T2
read(A)	
A := A − 50	
write(A)	
read(B)	
B := B + 50	
write(B)	
	read(A)
	temp := A * 0.1;
	A := A − temp
	write(A)
	read(B)

- A schedule for a set of transactions must consist of all instructions of those transactions
- Must preserve the order in which the instructions appear in each individual transaction

Transaction State :

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

ii)Partially committed : It is the state of transaction in which it indicates the execution completion of last statement without executing commit statement.

iii)Failed : It is the state of a transaction, in which the current transaction cannot proceed further. (after the discovery that normal execution can no longer proceed).

iv)Aborted: It is the state of the transaction in which the transaction has been rolled back and the database state is restored to its previous state to the start of the transaction. Two options are possible after it has been aborted : One is to restart the transaction – only if no internal logical error and another is to kill the transaction

v)Committed : It is the state of transaction in which all the statements have been successfully executed and completed with commit statement. After successful completion, the data at that instance has become permanent.

Atomicity and Durability: Transactions can be incomplete due to the following reasons :

i) A transaction can be aborted, or terminated unsuccessfully, by the DBMS, because some anomaly arises during execution. If a transaction is aborted for some internal reason, it is automatically restarted and executed as new transaction.

ii) The system may crash (because of varied reasons like power failures, collapse of building, attacked by fire hazards) while one or more transactions are in progress.

iii) A transaction may encounter an unexpected situation (read an unexpected data value, or unable to access some disk) and decide to abort (i.e. terminated itself).

Users are always thinking that transactions are being atomic, and that are interrupted in the middle of the execution and leave the database in an inconsistent state. Therefore, a DBMS must find a way to remove the effects of partial transactions from the database. That is, it must ensure transaction atomicity; either all of a transaction’s actions are carried out or none of them are carried out. A DBMS ensures transaction atomicity by undoing the actions of incomplete transactions.

Implementation of Atomicity and Durability :

The recovery-management component of a database system implements the support for atomicity and durability.

The shadow-database scheme:

- Assume that only one transaction is active at a time.
- A pointer called db pointer always points to the current consistent copy of the database.
- All updates are made on a shadow copy of the database, and db pointer is made to point to the updated shadow copy only after the transaction reaches partial commit and all updated pages have been flushed to disk.
- In case transaction fails, old consistent copy pointed to by db pointer can be used, and the shadow copy can be deleted.

Concurrent Executions

Multiple transactions are allowed to run concurrently in the system. Advantages are :

- Increased processor and disk utilization, leading to better transaction throughput: one transaction can be using the CPU while another is reading from or writing to the disk
- Reduced average response time for transactions: short transactions need not wait behind long ones

Concurrency control schemes – mechanisms to control the interaction among the concurrent transactions in order to prevent them from destroying the consistency of the database

Serializability:

The concept of serializability of schedules is used to identify which schedules are correct when transactions executions have interleaving of their operations in the schedules.

Basic Assumption – Each transaction preserves database consistency.

Thus serial execution of a set of transactions preserves database consistency

A (possibly concurrent) schedule is serializable if it is equivalent to a serial schedule. Different forms of schedule equivalence gives rise to the notions of :

1. conflict serializability

2. view serializability

We ignore operations other than read and write instructions, and we assume that transactions may perform arbitrary computations on data in local buffers in between reads and writes. Our simplified schedules consist of only read and write instructions.

A Serializable Schedule

Example-1

T1	T2
R(A)	
W(A)	
	R(A)
	W(A)
R(B)	
W(B)	
	R(B)
	W(B)
	Commit
Commit	

Serializable schedule :

Example-2

T1	T2
	R(A)
	W(A)
R(A)	
	R(B)
	W(B)
W(A)	
R(B)	
W(B)	
	Commit
Commit	

In the above example, even though the actions of T1 and T2 are interleaved, the result of this schedule is equivalent to running T1 and then running T2. Here, T1's read and write of B is not influenced by T2's actions on A, and the net effect is the same if these actions are swapped to obtain the serial schedule T1 ; T2.

Executing transactions serially in different orders may produce different results, but all are presumed to be acceptable, the DBMS make no guarantees about which of them will be outcome of an interleaved execution.

Interleaved Transactions : A schedule involving two transactions as shown in following figure. This does not contain either an abort nor commit action for both of the transactions.

T1	T2
R(A)	
W(A)	
	R(B)
	W(B)
R(C)	
W(C)	

In real world situation, any user of database cannot wait for previous user to complete his transaction. Means all the transactions cannot happen in sequential order. If that is the case you may have to wait for days in online websites and ATMs.

So the transactions are interleaved, means second transaction is started before the first one could end. And execution can switch between the transactions back and forth. It can also switch between multiple transactions. This could actually cause inconsistency in the system. But they are handled by the database systems.

Anomalies causes due to Interleaved Transactions :

i) Dirty read : A transaction could read a database object that has been modified by another transaction which has not yet committed is called as a dirty read. This happens when a transaction updates a data item, but later on the transaction fails. It could be due to system failure or any other operational reasons or the system may have noticed later that the operation should hot have been done and cancels it.

Consider two transactions T1 and T2, each of which run alone, preserve database consistency. T1 transfers Rs 500 from A to B, and T2 increments both A and B by 10%. Assume the actions are of interleaved type, so that the account transfer program T1 deducts Rs 500 from account A, then the interest deposit program T2 reads the current values of accounts A and B and adds 10% interest to each, and then the account transfer program credits Rs 500 to account B. This is illustrated in the following figure.

T1	T2
R(A)	
W(A)	
	R(A)
	W(A)
	R(B)
	W(B)
	Commit
R(B)	
W(B)	
Commit	

The result of this schedule is different from any result that we would get by running one of the two transactions first and then the other. The problem can be traced as the value of A written by T1 is read by T2 before T1 has completed all its changes.

The general problem here is that T1 may write some value into A that makes the database inconsistent. As long as T1 overwrites this value with a correct value of ‘A’ before committing, no harm is done if T1 and T2 run in some serial order, because T2 would then not see the inconsistency.

On the other hand, interleaved execution can expose this inconsistency and lead to an inconsistent final database state.

ii) Unrepeatable Reads (RW conflicts) : In the above transaction, T2 could change the value of an object A that has been read by a transaction T1, while T1 is still in progress. If T1 tries to read the value of A again, it will get a different result, even though it has not modified A in the mean time. This is called an **unrepeatable read**. This will not happen in case of serial transactions.

iii) Overwriting Uncommitted Data (WW Conflicts) : This is another problem happens when transaction T2 could overwrite the value of an object A, which has already been modified by a transaction T1, while T1 is still in progress. Even if T2 does not read the value of A written by T1. This problem is called **overwriting uncommitted data.**
Example **:** Suppose Arun and Varun are two employees , and their salaries must be kept equal. Transaction T1 sets their salaries to Rs 5000 and transaction T2 sets their salaries

to Rs 3000. If we execute these in the serial order T1 followed by T2, both will receive the salary of Rs 3000. If we change the order of transaction, i.e. T2 followed by T1, then both will get Rs. 5000. Here, neither of the transactions reads a salary value before writing it. This type of write is called **Blind Write.**

iv) Lost update : The first transaction to commit T2, overwrote Varun's salary as set by T1. In the serial order T2 followed by T1, Varun's salary should reflect T1's update rather than T2's, but T1's update is lost. This is called **Lost update** problem.

Lock –based Concurrency control :

A dbms must be able to ensure that only serializable, recoverable schedules are allowed and that no actions of committed transactions are lost while undoing aborted transactions. A dbms typically uses a locking protocol to achieve this.

A lock is a small bookkeeping object associated with a database object.

A locking protocol is a set of rules to be followed by each transaction to ensure that, even though actions of several transactions might be interleaved, the net effect is identical to executing all transactions in some serial order. Different locking protocols use different type of locks, such as shared locks or exclusive locks.

Types of Locks :

Several types of locks are used in concurrency control. They provide more general locking capabilities and are used in practical database locking schemes.

i) Binary Locks : A binary lock can have two states or values. i.e. locked and unlocked (0 or 1). A distinct lock is associated with each database item X. If the value of the lock on X is 1, item X cannot be accessed by a database operation that requests the item. If the value of the lock on X is 0, the item can be accessed when requested. Two operations, lock_item and unlock_item are used with binary locking .

ii) Shared / Exclusive Locks : (Read/Write Locks) : In binary locking scheme, it is too restrictive for database items, because at most, one transaction can hold a lock on a given item. We should allow several transactions to access the same item X if they all access X for reading purposes only. However, if a transaction is to write an item X, it must have exclusive access to X. For this purpose, a different type of lock called multi-mode lock is used. In this scheme, we use shared / exclusive or read/ write locks. Here, there are three locking operations : read_lock(X), Write_lock(X), and Unlock(X). A lock is associated with an item X.

LOCK(X) has three possible states, read-locked, write-locked, or unlocked.

A read-locked item is also called share-locked, because other transactions are allowed to read the item, whereas a write-locked item is called exclusive-locked, because a single transaction exclusively holds the lock on the item.

One method for implementing the locking operations on a read/write lock is to keep track of the number of transactions that hold a shared lock on an item in the lock table. Each record in the lock table will have four fields : < data item name, LOCK, no of reads, Locking_transactions>. Again to save space, the system needs to maintain lock records only for locked items in the lock table. The value of LOCK is either read-locked or write-locked, suitably coded.

If LOCK(X) = write-locked, the value of locking_transaction(s) is a single transaction that holds the exclusive lock on X.

If LOCK(X) = read-locked, the value of locking_transaction(s) is a list of one or more transactions that hold the shared lock on X.

Rules of using Shared /Exclusive locking scheme :

1. A transaction T must issue the operation read_lock(X) or write_lock(X) before any read_item(X) operation is performed in T.

2. A transaction T must issue the operation write_lock(X) before any write_item(X) operation is performed in T.

3. A transaction T must issue the operation unlock(X) after all read_item(X) and write_item(X) operations are completed in T.

4. A transaction T will not issue a read_lock(X) operation if it already holds a read (shared) lock or write (exclusive) lock in item X. This rule may be relaxed.

5. A transaction T will not issue a write_lock(X) operation if it already holds a read (shared) lock or write (exclusive) lock on item X. This rule may be relaxed.

6. A transaction T will not issue an unlock(X) operation unless it already holds a read (shared) lock or write (exclusive) lock on item X.

Latches and Convoys :

Latches are simple, low-level system lock (serialization mechanisms) that coordinate multi-user access (concurrency) to shared data structures, objects, and files.

Latches protect shared memory resources from corruption when accessed by multiple processes. Specifically, latches protect data structures from the following situations:

- Concurrent modification by multiple sessions
- Being read by one session while being modified by another session

- Deallocation (aging out) of memory while being accessed

Typically, a single latch protects multiple objects in the SGA.

The implementation of latches is operating system-dependent, especially in respect to whether and how long a process waits for a latch.

Convoys : Convoys is a form on a high traffic lock, where most of the CPU cycles are spent on process switching, often the log lock in a database. The problem is that a transaction T holding a heavily used lock may be suspended by the operating system. Each update generates an entry at end of log. Few hundred instructions. Convoys are one of the drawbacks of building a DBMS on top of a general-purpose operating system with preventive scheduling.

Performance of Locking :

Lock-based schemes are designed to resolve conflicts between transactions and use two basic mechanisms : blocking and aborting.

Blocking : Blocked transactions may hold locks that force other transactions to wait.

Aborting : Aborting and restarting a transaction obviously wastes the work done so far by that transaction.

Both these mechanisms will involve in performance penalty.

Deadlock : This represents an extreme instance of blocking in which a set of transactions is blocked for ever unless one of the deadlocked transactions is aborted by the database. In real time situation, below 1% of the transactions are involved in deadlock.

Strict Two-Phase Locking Protocol (Strict 2PL) :

It is the most widely used locking protocol. In transaction processing, **two-phase commit protocol** is a type of atomic commitment protocol (ACP). It is a distributed algorithm that coordinates all the processes that participate in a distributed atomic transaction on whether to *commit* or *abort* (*roll back*) the transaction. The protocol achieves its goal even in many cases of temporary system failure and is thus widely utilized.

It has two rules.

I rule : If a transaction T wants to read an object, it first request a shared lock on the object.

II rule : All locks held by a transaction are released when the transaction is completed.

The protocol consists of two phases:

1. The *commit-request phase* (or *voting phase*), in which a *coordinator* process attempts to prepare all the transaction's participating processes to take the necessary steps for either committing or aborting the transaction and to *vote*, either "Yes": commit (if the transaction participant's local portion execution has ended properly), or "No": abort (if a problem has been detected with the local portion), and

2. The *commit phase*, in which, based on *voting* of the cohorts, the coordinator decides whether to commit (only if *all* have voted "Yes") or abort the transaction (otherwise), and notifies the result to all the cohorts. The cohorts then follow with the needed actions (commit or abort) with their local transactional resources and their respective portions in the transaction's other output (if applicable).

The protocol works in the following manner:

One node is designated the **coordinator**, which is the master site, and the rest of the nodes in the network are designated the **cohorts**. The protocol assumes that there is stable storage at each node with a write-ahead log, that no node crashes forever, that the data in the write-ahead log is never lost or corrupted in a crash, and that any two nodes can communicate with each other.

The protocol is initiated by the coordinator after the last step of the transaction has been reached. The cohorts then respond with an **agreement** message or an **abort** message depending on whether the transaction has been processed successfully at the cohort.

Basic algorithm

Commit request phase or voting phase

1. The coordinator sends a **query to commit** message to all cohorts and waits until it has received a reply from all cohorts.
2. The cohorts execute the transaction up to the point where they will be asked to commit. They each write an entry to their *undo log* and an entry to their *redo log*.
3. Each cohort replies with an **agreement** message (cohort votes **Yes** to commit), if the cohort's actions succeeded, or an **abort** message (cohort votes **No**, not to commit), if the cohort experiences a failure that will make it impossible to commit.

Commit phase or Completion phase

Success : If the coordinator received an agreement message from *all* cohorts during the commit-request phase:

1. The coordinator sends a **commit** message to all the cohorts.

2. Each cohort completes the operation, and releases all the locks and resources held during the transaction.
3. Each cohort sends an **acknowledgment** to the coordinator.
4. The coordinator completes the transaction when all acknowledgments have been received.

Failure : If *any* cohort votes No during the commit-request phase (or the coordinator's timeout expires):

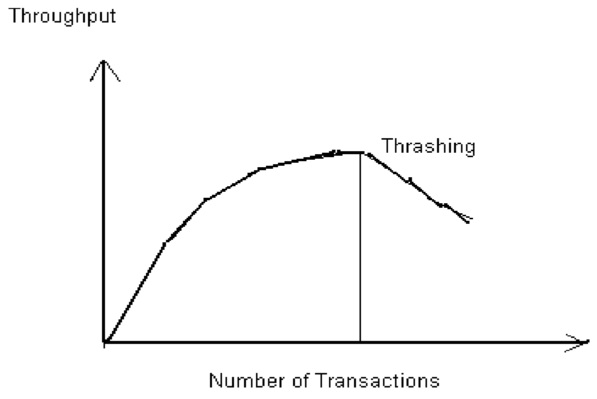
1. The coordinator sends a **rollback** message to all the cohorts.
2. Each cohort undoes the transaction using the undo log, and releases the resources and locks held during the transaction.
3. Each cohort sends an **acknowledgement** to the coordinator.
4. The coordinator undoes the transaction when all acknowledgements have been received.

Disadvantages

The greatest disadvantage of the two-phase commit protocol is that it is a blocking protocol. If the coordinator fails permanently, some cohorts will never resolve their transactions: After a cohort has sent an **agreement** message to the coordinator, it will block until a **commit** or **rollback** is received.

Thrashing :

It is a situation in the database where a new transaction will competes with the existing transactions. At this point, the system starts thrashes which results in delay and reduces the throughput which is illustrated in the following figure.



Throughput :– Number of actions held per unit time

At the beginning, number of transactions is few so that throughput is also less. In order to increase the performance of the system, it needs in increasing the throughput. This is possible by increasing the number of transactions. Performance of the system gradually increases, and reached the upper limit. If we still increase the number of transaction, the system performance will become drastically reduced and sometimes it will be stopped. This is called thrashing. This is due to increased number of transactions.

What DBA can do if the system starts thrashes ?

The DBA should reduce the number of transactions allowed to run concurrently. In real time situation, thrashing is seen to occur when 30% of active transactions are blocked and a DBA should monitor the fraction of blocked transactions to see if the system is at risk of thrashing.

Throughput can be increased in 3 ways :

- By locking the smallest sized possible objects
- By reducing the time that transaction hold locks
- By reducing the hot spots. A hot spot is a database object that is frequently accessed and modified and causes a lot of blocking delays. Hot spots can significantly affect performance.

Deadlocks :

In a multi-process system, deadlock is an unwanted situation that arises in a shared resource environment, where a process indefinitely waits for a resource that is held by another process.
For example, assume a set of transactions {T_0, T_1, T_2, ...,T_n}. T_0needs a resource X to complete its task. Resource X is held by T_1, and T_1 is waiting for a resource Y, which is held by T_2. T_2 is waiting for resource Z, which is held by T_0. Thus, all the processes wait for each other to release resources. In this situation, none of the processes can finish their task. This situation is known as a deadlock.

Deadlocks are not healthy for a system. In case a system is stuck in a deadlock, the transactions involved in the deadlock are either rolled back or restarted.

Deadlock Prevention

To prevent any deadlock situation in the system, the DBMS aggressively inspects all the operations, where transactions are about to execute. The DBMS inspects the operations and analyzes if they can create a deadlock situation. If it finds that a deadlock situation might occur, then that transaction is never allowed to be executed.
There are deadlock prevention schemes that use timestamp ordering mechanism of transactions in order to predetermine a deadlock situation.

Wait-Die Scheme

In this scheme, if a transaction requests to lock a resource (data item), which is already held with a conflicting lock by another transaction, then one of the two possibilities may occur –

- If $TS(T_i) < TS(T_j)$ – that is T_i, which is requesting a conflicting lock, is older than T_j – then T_i is allowed to wait until the data-item is available.
- If $TS(T_i) > TS(t_j)$ – that is T_i is younger than T_j – then $T_i$dies. T_i is restarted later with a random delay but with the same timestamp.

This scheme allows the older transaction to wait but kills the younger one.

Wound-Wait Scheme

In this scheme, if a transaction requests to lock a resource (data item), which is already held with conflicting lock by some another transaction, one of the two possibilities may occur –

- If $TS(T_i) < TS(T_j)$, then T_i forces T_j to be rolled back − that is T_i wounds T_j. T_j is restarted later with a random delay but with the same timestamp.

- If $TS(T_i) > TS(T_j)$, then T_i is forced to wait until the resource is available.

This scheme, allows the younger transaction to wait; but when an older transaction requests an item held by a younger one, the older transaction forces the younger one to abort and release the item.

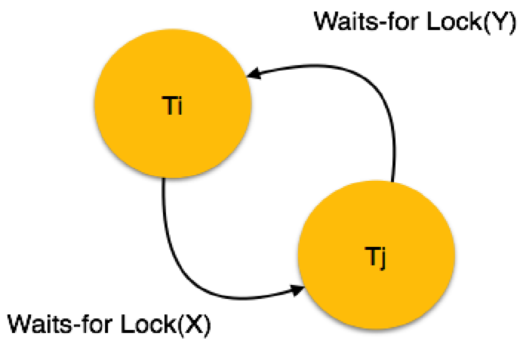
In both the cases, the transaction that enters the system at a later stage is aborted.

Deadlock Avoidance

Aborting a transaction is not always a practical approach. Instead, deadlock avoidance mechanisms can be used to detect any deadlock situation in advance. Methods like "wait-for graph" are available but they are suitable for only those systems where transactions are lightweight having fewer instances of resource. In a bulky system, deadlock prevention techniques may work well.

Wait-for Graph

This is a simple method available to track if any deadlock situation may arise. For each transaction entering into the system, a node is created. When a transaction T_i requests for a lock on an item, say X, which is held by some other transaction T_j, a directed edge is created from T_i to T_j. If T_j releases item X, the edge between them is dropped and T_i locks the data item.
The system maintains this wait-for graph for every transaction waiting for some data items held by others. The system keeps checking if there's any cycle in the graph.



Here, we can use any of the two following approaches −

- First, do not allow any request for an item, which is already locked by another transaction. This is not always feasible and may cause starvation, where a transaction indefinitely waits for a data item and can never acquire it.

- The second option is to roll back one of the transactions. It is not always feasible to roll back the younger transaction, as it may be important than the older one. With the help of some relative algorithm, a transaction is chosen, which is to be aborted. This transaction is known as the **victim** and the process is known as **victim selection**.

Transaction support in SQL :

Nested transactions : These are the transactions can have several sub transactions with themselves, each of which can be selectively rolled back. This was introduced in 1999 through the inclusion of save point feature. The introduction of save points represents the first SQL support for the concept of "Nested transactions".

Chained transactions : Here, we can commit / rollback a transaction and immediately initiate another transaction. This is done by using optional keywords AND CHAIN in the COMMIT and ROLLBACK statements.

Phantom problem : A transaction retrieves a collection of objects twice and sees different results, even though it does not modify any of these tuples itself. To prevent phantoms, the DBMS must conceptually lock all possible rows.

Transaction characteristics in SQL : In order to give programmers control over the locking overhead incurred by their transactions, SQL allows them to specify three characteristics of transaction : access mode, diagnostics size and isolation level.

i) Access mode : It is READ ONLY → Transaction will not allow any modification (no insert, delete or update). Shared locks can be used.

ii) Isolation level : It controls the extent to which a given transaction is exposed to the actions of other transactions executing concurrently. Isolation level choices are READ UNCOMMITTED, READ COMMITTED, REPEATABLE READ, SERIALIZABLE.

Level	Dirty read	Un repeatable read	Phantom
Read Uncommitted	May be	May be	May be
Read Committed	No	May be	May be
Repeatable READ	No	No	May be
Serializable	No	No	No

iii) Diagnostics size : This determines the number of error conditions that can be recorded.

Introduction to Crash Recovery :

The recovery manager of DBMS is responsible for ensuring transaction atomicity and durability. It ensures atomicity by undoing the actions of transactions that do not commit and durability by making sure that all actions of committed transactions survive system crashes (Ex. A core dump caused by a bus error) and media failures (Ex. A disk is corrupted).

When a DBMS is restarted after crashes, the recovery manager is given control and must bring the database to a consistent state. The recovery manager is also responsible for undoing the actions of an aborted transaction. To see what it takes to implement a recovery manager, it is necessary to understand what happens during normal execution.

The transaction manager of a DBMS controls the execution of transactions. Before reading and writing objects during normal execution, locks must be acquired and released later after some time, according to a chosen locking protocols.

The log : It is a history of actions executed by the DBMS. Physically, the log is a file of records stored in stable storage, which is assumed to survive crashes. This durability can be achieved by maintaining two or more copies of the log on different disks, so that the chance of all copies of the log being simultaneously lost is negligibly small.

Log tail : It is the most recent portion of the log and is kept in main memory. It is periodically forced to stable storage. This way, log records and data records are written to disk at the same granularity.

LSN : (Log Sequence Number) : It is a unique id. As with any record-id, we can fetch a log record with one disk access given the LSN. It should be assigned in increasing order and this property is required for the ARIES recovery algorithm.

PageLSN : For recovery purposes, every page in the database contains the LSN of the most recent log record that describes a change to this page. This is called the PageLSN.

CLR : (Compensation Log Record) : It is written just before the change recorded in an update log record is undone. It describes the action taken to undo the actions recorded in the corresponding update log record and is appended to log tail just like any other record.

Transaction table : It contains one entry for each active transaction. The entry contains the transactionid, the status, and a field called lastLSN, which is the LSN of the most recent log record for this transaction. The status of a transaction can be that it is in progress, committed, or aborted.

Dirty Page table : It contains one entry for each dirty page in buffer pool, that is, each page with changes not yet reflected on disk. The entry contains a field recLSN, which is the LSN of the first log record that caused the page to become dirty.

Checkpointing : It is a snapshot of the DBMS state, and by taking check points periodically, as we will see, the DBMS can reduce the amount of work to be done during restart in the event of a subsequent crash.

Checkpointing in ARIES has three steps :

i) Begin-checkpoint : This record is written to indicate when the checkpoint starts.

ii) End-checkpoint : This record is constructed, including in it the current contents of the transaction table and dirty page table and appended to the log.

iii) End-checkpoint : The third step is carried out after the end-checkpoint record is written to stable storage.

Fuzzy checkpoint : It is a special kind of check point which is inexpensive because it does not require writing out pages in the buffer pool. On the other hand , the effectiveness of the check pointing technique is limited by the earliest recLSN of pages in the dirty pages table, because during restart, we must redo changes starting from the log record whose LSN is equal to this recLSN.

Note : When the system comes back up after a crash, the restart process begins by locating the most recent checkpoint record. For uniformity, the system always begins normal execution by taking checkpoint, in which the transaction table and dirty table are both empty.

Recovery related steps during normal execution :

The recovery manager of DBMS maintain some information during normal execution of transactions to enable it to perform its tasks in the event of a failure. In particular, a log of all modifications to the database is saved on stable storage, which is guaranteed to survive crashes and media failures. Stable storage is implemented by maintaining multiple copies of information on non volatile storage devices such as disks or tapes.

It is important to ensure that the log entries describing a change to the database are written to stable storage before the change is made, otherwise, the system might crash just after the change, leaving us without a record of the change.

The log enables the recovery manager to undo the actions of aborted and incomplete transactions and redo the actions of committed transactions. Further, changes of transactions that did not commit prior to the crash might have been written to disk because of steal approach. Such changes must be identified using the log and then it is undone.

The amount of work involved during recovery is proportional to the changes made by committed transactions that have not been written to disk at the time of the crash. To reduce the time to recover from a crash, the DBMS periodically forces buffer pages to disk during normal execution using background process. A process called checkpointing, which saves information about active transactions and dirty buffer pool pages, also helps to reduce the time taken to recover from a crash.

ARIES algorithm :

It is a recovery algorithm that is designed to work with a steal, no-force approach. When the recovery manager is invoked after a crash, restart proceeds in three phases.

i) In the Analysis phase, it identifies dirty pages in the buffer pool and active transactions at the time of crash.

ii) In the Redo phase, it repeats all actions, starting from an appropriate point in the log, and restores the database state to what it was at the time of the crash.

iii) Finally, in the Undo phase, it undoes the actions of transactions that did not commit, so that the database reflects only the actions of committed transactions.

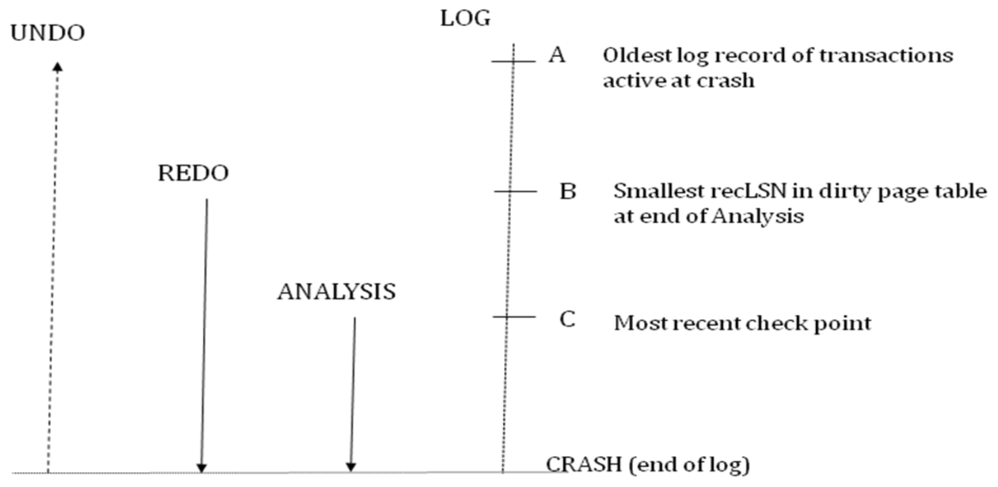
Three main principles lie behind the ARIES recovery algorithm :

- Write-Ahead logging : Any change to a database object is first recorded in the log, the record in the log must be written to stable storage before the change to the database object is written to disk.
- Repeating history during Redo : On restart following a crash, ARIES retraces all actions of the DBMS before the crash and brings the system back to the exact state that it was in at the time of the crash. Then, it undoes the actions of transactions still active at the time of the crash.
- Logging changes during Undo : Changes made to the database while undoing a transaction are logged to ensure such an action is not repeated in the event of repeated restarts.

ARIES algorithm is popular recovery algorithm, because it acts as a base of its simplicity and flexibility. In particular, ARIES can support concurrency control protocols that involve locks of finer granularity than a page. The second and third points are also important in dealing with operations where redoing and undoing the operation are not exact inverse of each other.

Recovering from a System Crash :

When the system is restarted after a crash, the recovery manager proceeds in three phases, as shown in the following figure.


UNDO
LOG
A Oldest log record of transactions active at crash
REDO
B Smallest recLSN in dirty page table at end of Analysis
ANALYSIS
C Most recent check point
CRASH (end of log)

Three Phases of Restart in ARIES

(LSN : Log Sequence Number, CLR: Compensation Log Record)

The Analysis phase begins by examining the most recent begin check point record, whose LSN is denoted in C in figure and proceeds forward in the log until the last log record.

The Redo phase follows Analysis and redoes all changes to any page that might have been dirty at the time of the crash. This set of pages and the starting point for Redo are determined during Analysis. The Undo phase follows Redo and undoes the changes all transactions active at the time of crash, again, this set of transactions is identified during the Analysis phase. Note that Redo applies changes in the order in which they were originally carried out, Undo reverses changes in the opposite order, reversing the most recent changes first.

i) Analysis phase : The analysis phase performs three tasks

- It determines the point in the log at which to start the Redo pass.
- It determines pages in the buffer pool that were dirty at the time of the crash.
- It identifies transactions that were active at the time of the crash and must be undone.

Analysis begins by examining the most recent begin_check_point log record and initializing the dirty page table and transaction table to the copies of those structures in the next end_checkpoint record. Thus, these tables are initialized to the set of dirty pages and active transactions at the time of the checkpoint. Analysis then scans the log in the forward direction until it reaches the end of the log.

At the end of the Analysis phase, the transaction table contains an accurate list of all transactions that were active at the time of the crash, this is the set of transactions with status U. The dirty page table includes all pages that were dirty at the time of the crash but may also contain some pages that were written to disk.

ii) Redo Phase :

During this phase, ARIES reapplies the updates of all transactions, committed or otherwise. Further, if a transaction was aborted before the crash and its updates were undone, the actions described in CLRs are also reapplied. The Redo phase begins with the log record that has the smallest recLSN of all pages in the dirty page table constructed by the Analysis pass, because, this log record indentifies the oldest update that may not have been written to disk prior to the crash. The action must be redone unless one of the following conditions holds :

- The affected page is not in the dirty page table.
- The affected page is in the dirty page table, but the recLSN for the entry is greater than the LSN of the log record being checked.
- The pageLSN is greater than or equal to the LSN of the log record being checked.

iii) Undo Phase :

This phase begins with the transaction table constructed by the Analysis phase, which identified all transactions active at the time of the crash, and includes the LSN of the most recent log record for each such transaction. Such transactions are called loser transactions. All actions of losers must be undone, and further, these actions must be undone in the reverse of the order in which they appear in the log.

Aborting a transaction : Aborting a transaction is just a special case of the Undo phase of restart in which a single transaction, rather than set of transactions, is undone.

Media recovery :

It is based on periodically making a copy of the database. Because copying a large database object such as a file can take a long time and the DBMS must be allowed to continue with its operations in the meantime, creating a copy is handled in a manner similar to taking a fuzzy check point.

When a database object such as a file or a page is corrupted, the copy of that object is brought up-to-date by using the log to identify and reapply the changes of committed transactions and undo the changes of uncommitted transactions .

The begin-checkpoint LSN of the most recent complete checkpoint is recorded along with the copy of the database object to minimize the work in reapplying changes of committed transactions. Finally, the updates of transactions that are incomplete at the time of media recovery or that were aborted after the fuzzy copy was completed need to be undone to ensure that the page reflects only the actions of committed transactions. The set of such transactions can be identified as in the Analysis pass and we omit the details.