DATABASE MANAGEMENT SYSTEM (DBMS)

(R-21 Autonomous)

Department of Electrical & Electronice Engineering

Malla Reddy College of Engineering & Technology

(Accredited by NBA with NAAC-A Grade, UGC-Autonomous, ISO Certified Institution)

Maisammaguda, Near Kompally, Medchal Road, Sec'bad-500 100.

SYLLABUS
(R20A0551) DATABASE MANAGEMENT SYSTEMS

Objectives:

To understand the basic concepts and the applications of database systems

To Master the basics of SQL and construct queries using SQL

To understand the relational database design principles

To become familiar with the basic issues of transaction processing and concurrency controlTo become familiar with database storage structures and access techniques

UNIT I:

INTRODUCTION: Database -Purpose of Database Systems, File Processing System Vs DBMS, History, Characteristic-Three schema Architecture of a database, Functional components of a DBMS.DBMS Languages-Database users and DBA.

UNIT II:

DATABASE DESIGN: ER Model - Objects, Attributes and its Type. Entity set and Relationship set-Design Issues of ER model-Constraints. Keys-primary key, super key, candidate keys. Introduction to relational model-Tabular, Representation of Various ER Schemas.ER Diagram Notations- Goals of ER Diagram- Weak Entity Set-Views.

UNIT III:

STRUCTURED QUERY LANGUAGE: SQL: Overview, The Form of Basic SQL Query -UNION, INTERSECT, and EXCEPT– join operations: equi join and non equi join-Nested queries - correlated and uncorrelated- Aggregate Functions- Null values. Views, Triggers.

UNIT IV:

DEPENDENCIES AND NORMAL FORMS: Importance of a good schema design-Problems encountered with bad schema designs, Motivation for normal forms-functional dependencies, -Armstrong's axioms for FD's- Closure of a set of FD's,-Minimal coversDefinitions of 1NF,2NF, 3NF and BCNF- Decompositions and desirable properties -

UNIT V:

TRANSACTIONS: Transaction concept, transaction state, System log, Commit point, Desirable Properties of a Transaction, concurrent executions, serializability, recoverability, implementation of isolation, transaction definition in SQL, Testing for serializability, Serializability by Locks-Locking Systems with Several Lock Modes-Concurrency Control by Time stamps, validation.

TEXT BOOKS:

1.Abraham Silberschatz, Henry F. Korth, S. Sudarshan,‖ Database System Concepts‖, McGraw- Hill, 6th Edition ,2010.

2. Fundamental of Database Systems, by Elmasri, Navathe, Somayajulu, and Gupta, Pearson Education.

REFERENCE BOOKS:

1.Raghu Ramakrishnan, Johannes Gehrke, ―Database Management System‖, McGraw Hill., 3rd Edition2007.

2. Elmasri & Navathe Fundamentals of Database System,‖ Addison-Wesley Publishing,5th Edition,2008.

3. Date.C.J, ―An Introduction to Database‖, Addison-Wesley Pub Co, 8th Edition,2006.

4. Peterrob, Carlos Coronel, ―Database Systems – Design, Implementation, and Management‖, 9th Edition, Thomson Learning,2009.

URLs:

Outcomes:

- Understand the basic concepts and the applications of database systems
- Master the basics of SQL and construct queries using SQL
- Understand the relational database design principles
- Familiarize with the basic issues of transaction processing and concurrency control

Familiarize with database storage structures and access techniques

UNIT-1

Introduction to Database Management System

As the name suggests, the database management system consists of two parts. They are:

1. Database and
2. Management System

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.

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.

A database in a DBMS could be viewed by lots of different people with different responsibilities.

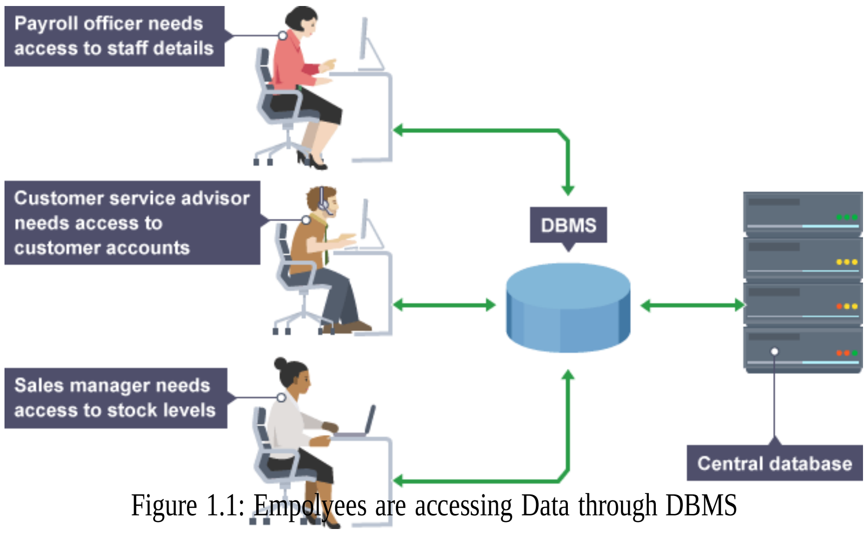

Payroll officer needs access to staff details
Customer service advisor needs access to customer accounts
Sales manager needs access to stock levels
DBMS
Central database

Figure 1.1: Empolyees are accessing Data through DBMS

For example, within a company there are different departments, as well as customers, who each need to seedifferent kinds of data. Each employee in the company will have different levels of access to the database with their own customized **front-end** application.

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.

What is Management System?

A **database-management system** (DBMS) is a collection of interrelated data and a set of programs to access those data. This is a collection of related data with an implicit meaning and hence is a database. The collection of data, usually referred to as the **database**, contains information relevant to an enterprise. The primary goal of a DBMS is to provide a way to store and retrieve database information that is both *convenient* and *efficient*. By **data,** we mean known facts that can be recorded and that have implicit meaning.

The management system 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.

Database systems are designed to manage large bodies of information. Management of data involves both defining structures for storage of information and providing mechanisms for the manipulation of information. In addition, the database system must ensure the safety of the information stored, despite system crashes or attempts at unauthorized access. If data are to be shared among several users, the system must avoid possible anomalous results.

Because information is so important in most organizations, computer scientists have developed a large body of concepts and techniques for managing data. These concepts and technique form the focus of this book. This chapter briefly introduces the principles of database systems.

Database Management System (DBMS) and Its Applications:

A Database management system is a computerized record-keeping system. It is a repository or a container for collection of computerized data files. The overall purpose of DBMS is to allow he users to define, store, retrieve and update the information contained in the database on demand. Information can be anything that is of significance to an individual or organization.

Databases touch all aspects of our lives. Some of the major areas of application are as follows:

1. Banking
2. Airlines
3. Universities
4. Manufacturing and selling
5. Human resources

Enterprise Information

- *Sales*: For customer, product, and purchase information.
- *Accounting*: For payments, receipts, account balances, assets and other accounting information.
- *Human resources*: For information about employees, salaries, payroll taxes, and benefits, and for generationof paychecks.
- *Manufacturing*: For management of the supply chain and for tracking production of items in factories, inventories of items in warehouses and stores, and orders for items.

Online retailers: For sales data noted above plus online order tracking, generation of recommendation lists,and maintenance of online product evaluations.

Banking and Finance

- *Banking*: For customer information, accounts, loans, and banking transactions.
- *Credit card transactions*: For purchases on credit cards and generation of monthly statements.
- *Finance*: For storing information about holdings, sales, and purchases of financial instruments such as stocks and bonds; also for storing real-time market data to enable online trading by customers and automated trading by the firm.

- *Universities*: For student information, course registrations, and grades (in addition to standard enterprise information such as human resources and accounting).
- *Airlines*: For reservations and schedule information. Airlines were among the first to use databases in a geographically distributed manner.
- *Telecommunication*: For keeping records of calls made, generating monthly bills, maintaining balances onprepaid calling cards, and storing information about the communication networks.

Purpose of Database Systems

Database systems arose in response to early methods of computerized management of commercial data. Asan example of such methods, typical of the 1960s, consider part of a university organization that, among other data, keeps information about all instructors, students, departments, and course offerings. One way to keep the information on a computer is to store it in operating system files. To allow users to manipulate the information, the system has a number of application programs that manipulate the files, including programs to:

- ✓ Add new students, instructors, and courses
- ✓ Register students for courses and generate class rosters
- ✓ Assign grades to students, compute grade point averages (GPA), and generate transcripts

System programmers wrote these application programs to meet the needs of the university.
New application programs are added to the system as the need arises. For example, suppose that a university decides to create a new major (say, computer science). As a result, the university creates a new department and creates new permanent files (or adds information to existing files) to record information about all the instructors in the department, students in that major, course offerings, degree requirements, etc. The university may have to write new application programs to deal with rules specific to the new major. New application programs may also have to be written to handle new rules in the university. Thus, as time goes by, the system acquires more files and more application programs.

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) were introduced, organizations usually stored information in such systems. Keeping organizational information in a file- processing system has a number of major disadvantages:

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 structures and the programs may be written inseveral programming languages. Moreover, the same information may be duplicated in several places (files).For example, if a student has a double major (say, music and mathematics) the address and telephone number of that student may appear in a file that consists of student records of students in the Music department and in a file that consists of student records of students in the Mathematics department. This redundancy leads to higher storage and access cost. In addition, it may lead to **data inconsistency**; that is, the various copies ofthe same data may no longer agree. For example, a changed student address may be reflected in the Musicdepartment records but not elsewhere in the system.

Difficulty in accessing data. Suppose that one of the university clerks needs to find out the names of all students who live within a particular postal-code area. The clerk asks the data-processing department to generate such a list. Because the designers of the original system did not anticipate this request, there is noapplication program on hand to meet it. There is, however, an application program to generate the list of *all* students.

The university clerk has now two choices: either obtain the list of all students and extract the needed information manually or ask a programmer to write the necessary application program. Both alternatives areobviously unsatisfactory. Suppose that such a program is written, and that, several days later, the same clerkneeds to trim that list to include only those students who have taken at least 60 credit hours. As expected, a program to generate such a list does not exist. Again, the clerk 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.

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

Integrity problems. The data values stored in the database must satisfy certain types of **consistency constraints**. Suppose the university maintains an account for each department, and records the balance amount in each account. Suppose also that the university requires that the account balance of a departmentmay never fall below zero. Developers enforce these constraints in the system by adding appropriate code inthe 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.

Atomicity problems. A computer system, like any other device, is subject to failure. In many applications, it is crucial that, if a failure occurs, the data be restored to the consistent state that existed prior to the failure.

Consider a program to transfer $500 from the account balance of department *A* to the account balance of department *B*. If a system failure occurs during the execution of the program, it is possible that the $500 wasremoved from the balance of department *A* but was not credited to the balance of department *B*, resulting in an inconsistent database state. Clearly, it is essential to database consistency that either both the credit and debit occur, or that neither occur.

That is, the funds transfer must be *atomic*—it must happen in its entirety or not at all. It is difficult to ensure atomicity in a conventional file-processing system.

Concurrent-access anomalies. For the sake of overall performance of the system and faster response, many systems allow multiple users to update the data simultaneously. Indeed, today, the largest Internet retailers may have millions of accesses per day to their data by shoppers. In such an environment, interaction of concurrent updates is possible and may result in inconsistent data. Consider department *A*, with an account balance of $10,000. If two department clerks debit the account balance (by say $500 and $100, respectively) of department *A* at almost exactly the same time, the result of the concurrent executions may leave the budget 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 programsrun concurrently, they may both read the value $10,000, and write back $9500 and $9900, respectively. Depending on which one writes the value last, the account balance of department *A* may contain either $9500 or $9900, rather than the correct value of $9400. 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 programsthat have not been coordinated previously.

As another example, suppose a registration program maintains a count of students registered for a course, inorder to enforce limits on the number of students registered. When a student registers, the program reads thecurrent count for the courses, verifies that the count is not already at the limit, adds one to the count, and stores the count back in the database. Suppose two students register concurrently, with the count at (say) 39. The two program executions may both read the value 39, and both would then write back 40, leading to an incorrect increase of only 1, even though two students successfully registered for the course and the count should be 41. Furthermore, suppose the course registration limit was 40; in the above case both students would be able toregister, leading to a violation of the limit of 40 students.

Security problems. Not every user of the database system should be able to access all the data. For example, in a university, payroll personnel need to see only that part of the database that has financial information. They do not need access to information about academic records. But, since application programs are added to thefile-processing 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.

Advantages of DBMS:

Controlling of Redundancy: Data redundancy refers to the duplication of data (i.e storing same data multiple times). In a database system, by having a centralized database and centralized control of data by the DBA the unnecessary duplication of data is avoided. It also eliminates the extra time for processing the large volume of data. It results in saving the storage space.

Improved Data Sharing : DBMS allows a user to share the data in any number of application programs.

Data Integrity : Integrity means that the data in the database is accurate. Centralized control of the data helps in permitting the administrator to define integrity constraints to the data in the database. For example: in customer database we can can enforce an integrity that it must accept the customer only from Noida and Meerut city.

Security : Having complete authority over the operational data, enables the DBA in ensuring that the only mean of access to the database is through proper channels. The DBA can define authorization checks to becarried out whenever access to sensitive data is attempted.

Data Consistency: By eliminating data redundancy, we greatly reduce the opportunities for inconsistency. For example: is a customer address is stored only once, we cannot have disagreement on the stored values. Also updating data values is greatly simplified when each value is stored in one place only. Finally, we avoid the wasted storage that results from redundant data storage.

Efficient Data Access: In a database system, the data is managed by the DBMS and all access to the data is through the DBMS providing a key to effective data processing

Enforcements of Standards: With the centralized of data, DBA can establish and enforce the data standards which may include the naming conventions, data quality standards etc.

Data Independence: Ina database system, the database management system provides the interface between the application programs and the data. When changes are made to the data representation, the meta data obtained by the DBMS is changed but the DBMS is continuing to provide the data to application program in the previously used way. The DBMs handles the task of transformation of data wherever necessary.

Reduced Application Development and Maintenance Time: DBMS supports many important functions that are common to many applications, accessing data stored in the DBMS, which facilitates the quick development of application.

Disadvantages of DBMS

1) It is bit complex. Since it supports multiple functionality to give the user the best, the underlying softwarehas become complex. The designers and developers should have thorough knowledge about the software to get the most out of it.
2) Because of its complexity and functionality, it uses large amount of memory. It also needs large memory to run efficiently.
3) DBMS system works on the centralized system, i.e.; all the users from all over the world access this database. Hence any failure of the DBMS, will impact all the users.
4) DBMS is generalized software, i.e.; it is written work on the entire systems rather specific one. Hence some of the application will run slow.

File Processing System Vs DBMS:

Difference between File System and DBMS:

Basis	File System	DBMS
Structure	The file system is software that manages and organizes the files in a storage medium within a computer.	DBMS is software for managing the database.
Data Redundancy	Redundant data can be present in a file system.	In DBMS there is no redundant data.

Basis	File System	DBMS
Backup and Recovery	It doesn't provide backup and recovery of data if it is lost.	It provides backup and recovery of data even if it is lost.
Query processing	There is no efficient query processing in the file system.	Efficient query processing is there in DBMS.
Consistency	There is less data consistency in the file system.	There is more data consistency because of the process of normalization.
Complexity	It is less complex as compared to DBMS.	It has more complexity in handling as compared to the file system.
Security Constraints	File systems provide less security in comparison to DBMS.	DBMS has more security mechanisms as compared to file systems.
Cost	It is less expensive than DBMS.	It has a comparatively higher cost than a file system.
Data Independence	There is no data independence.	In DBMS data independence exists.
User Access	Only one user can access data at a time.	Multiple users can access data at a time

History:

Data is a collection of facts and figures. The data collection was increasing day to day and they needed to be stored in a device or a software which is safer.

Charles Bachman was the first person to develop the Integrated Data Store (IDS) which was based on network data model for which he was inaugurated with the Turing Award (The most prestigious award which is equivalent to Nobel prize in the field of Computer Science.). It was developed in early 1960's.

In the late 1960's, IBM (International Business Machines Corporation) developed the Integrated Management Systems which is the standard database system used till date in many places. It was developed based on the hierarchical database model. It was during the year 1970 that the relational database model was developed by Edgar Codd. Many of the database models we use today are relational based. It was considered the standardized database model from then.
The relational model was still in use by many people in the market.Later during the same decade (1980's), IBM developed the Structured Query Language (SQL) as a part of R project. It was declared as a standard language for the queries by ISO and ANSI. The Transaction Management Systems for processing transactions was also developed by James Gray for which he was felicitated the Turing Award.
Further, there were many other models with rich features like complex queries, datatypes to insert images and many others. The Internet Age has perhaps influenced the data models much more. Data models were developed using object oriented programming features, embedding with scripting

languages like [Hyper Text Markup Language (HTML)]() for queries. With humongous data being available online, DBMS is gaining more significance day by day.

View of Data

A database system is a collection of interrelated data and a set of programs that allow users to access and modify these data. A major purpose of a database system is to provide users with an *abstract* view of the data. That is, the system hides certain details of how the data are stored and maintained.

Data Abstraction

For the system to be usable, it must retrieve data efficiently. The need for efficiency has led designers to usecomplex data structures to represent data in the database. Since many database-system users are not computer trained, developers hide the complexity from users through several levels of abstraction, to simplifyusers' interactions with the system:

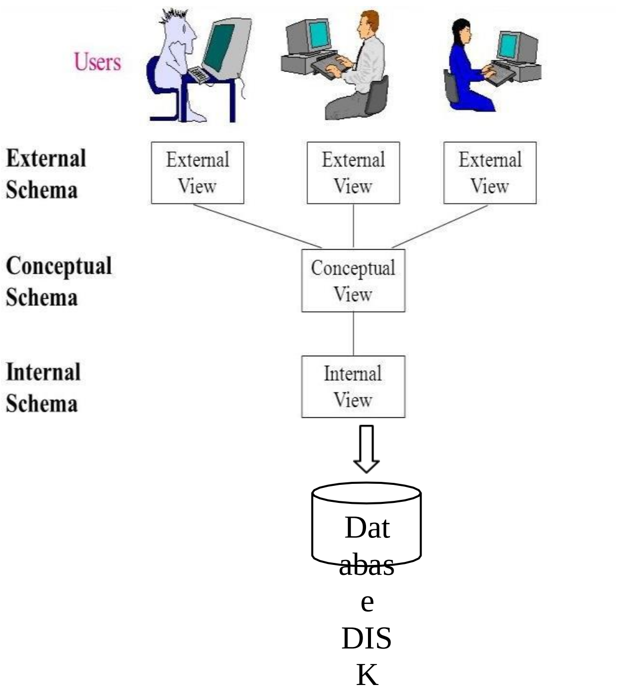


Figure 1.2 : Levels of Abstraction in a DBMS

• **Physical level (or Internal View / Schema)**: The lowest level of abstraction describes *how* the data are actually stored. The physical level describes complex low-level data structures in detail.

• **Logical level (or Conceptual View / Schema)**: The next-higher level of abstraction describes *what* data are stored in the database, and what relationships exist among those data. The logical level thus describes the entire database in terms of a small number of relatively simple structures. Although implementation of the simple structures at the logical level may involve complex physical-level structures, the user of the logical level does not need to be aware of this complexity. This is referred to as **physical data independence**. Databaseadministrators, who must decide what information to keep in the database, use the logical level of abstraction.

• **View level (or External View / Schema):** The highest level of abstraction describes only part of the entiredatabase. Even though the logical level uses simpler structures, complexity remains because of the variety ofinformation stored in a large database. Many users of the database system do not need all this information; instead, they need to access only a part of the database. The view level of abstraction exists to simplify theirinteraction with the system. The system may provide many views for the same database. Figure 1.2 shows the relationship among the three levels of abstraction.

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

type *instructor* = **record**

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

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

- *department*, with fields *dept_name*, *building*, and *budget*
- *course*, with fields *course_id*, *title*, *dept_name*, and *credits*
- *student*, with fields *ID*, *name*, *dept_name*, and *tot_cred*

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

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

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

Instances and Schemas

Databases change over time as information is inserted and deleted. The collection of information stored in the database at a particular moment is called an **instance** of the database. The overall design of the database iscalled the database **schema**. Schemas are changed infrequently, if at all. The concept of database schemasand instances can be understood by analogy to a program written in a programming language. A database schema corresponds to the variable declarations (along with associated type definitions) in a program.

Each variable has a particular value at a given instant. The values of the variables in a program at a point in time correspond to an *instance* of a database schema. Database systems have several schemas, partitionedaccording to the levels of abstraction. The **physical schema** describes the database design at the physical level, while the **logical schema** describes the database design at the logical level. A database may also have several schemas at the view level, sometimes called **subschemas**, which describe different views of the database. Of these, the logical schema is by far the most important, in terms of its effect on application programs, since programmers construct applications by using the logical schema. The physical schema is hidden beneath the logical schema, and can usually be changed easily without affecting application programs. Application programs are said to exhibit **physical data independence** if they do not depend on the physicalschema, and thus need not be rewritten if the physical schema changes.

Data Models

Underlying the structure of a database is the **data model**: a collection of conceptual tools for describing data, data relationships, data semantics, and consistency constraints. A data model provides a way to describe the design of a database at the physical, logical, and view levels.

The data models can be classified into four different categories:

• **Relational Model**. The relational model uses a collection of tables to represent both data and the relationships among those data. Each table has multiple columns, and each column has a unique name. Tables are also known as **relations**. The relational model is an example of a record-based model.
Record-based models are so named because the database is structured in fixed-format records of several types. Each table contains records of a particular type. Each record type defines a fixed number of fields, or attributes. The columns of the table correspond to the attributes of the record type. The relational data model is the most widely used data model, and a vast majority of current database systems are based on the relational model.

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

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

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

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

Database Languages
A database system provides a **data-definition language** to specify the database
schema and a **data-manipulation language** to express database queries and updates. In practice, the data- definition and data-manipulation languages are not
two separate languages; instead they simply form parts of a single database language, such as the
widely used SQL language.

Data-Manipulation Language

A **data-manipulation language (DML)** is a language that enables users to access or manipulate data asorganized by the appropriate data model. The types of access are:

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

There are basically two types:
• **Procedural DMLs** require a user to specify *what* data are needed and *how* to get those data.
• **Declarative DMLs** (also referred to as **nonprocedural DMLs**) require a user to specify *what* data are needed *without* specifying how to get those data.

Declarative DMLs are usually easier to learn and use than are procedural DMLs. However, since a user doesnot have to specify how to get the data, the database system has to figure out an efficient means of accessing

data. A **query** is a statement requesting the retrieval of information. The portion of a DML that involves information retrieval is called a **query language**. Although technically incorrect, it is common practice to use the terms *query language* and *data-manipulation language* synonymously.

Data-Definition Language (DDL)

We specify a database schema by a set of definitions expressed by a special language called a **data-definition language** (**DDL**). The DDL is also used to specify additional properties of the data.

We specify the storage structure and access methods used by the database system by a set of statements in a special type of DDL called a **data storage and definition** language. These statements define the implementation details of the database schemas, which are usually hidden from the users.
The data values stored in the database must satisfy certain **consistency constraints**.
For example, suppose the university requires that the account balance of a department must never be negative. The DDL provides facilities to specify such constraints. The database system checks these constraints everytime the database is updated. In general, a constraint can be an arbitrary predicate pertaining to the database. However, arbitrary predicates may be costly to test. Thus, database systems implement integrity constraintsthat can be tested with minimal overhead.

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

• **Referential Integrity**. There are cases where we wish to ensure that a value that appears in one relation for a given set of attributes also appears in a certain set of attributes in another relation (referential integrity). For example, the department listed for each course must be one that actually exists. More precisely, the *dept name* value in a *course* record must appear in the *dept name* attribute of some record of the *department* relation.
Database modifications can cause violations of referential integrity. When a referential-integrity constraint is violated, the normal procedure is to reject the action that caused the violation.

• **Assertions**. An assertion is any condition that the database must always satisfy. Domain constraints and referential-integrity constraints are special forms of assertions. However, there are many constraints that wecannot express by using only these special forms. For example, "Every department must have at least five courses offered every semester" must be expressed as an assertion. When an assertion is created, the system tests it for validity. If the assertion is valid, then any future modification to the database is allowed only if it does not cause that assertion to be violated.

• **Authorization**. We may want to differentiate among the users as far as the type of access they are permitted on various data values in the database. These differentiations are expressed in terms of **authorization**, the most common being: **read authorization**, which allows reading, but not modification, of data; **insert authorization**, which allows insertion of new data, but not modification of existing data; **update authorization**, which allows modification, but not deletion, of data; and **delete authorization**, which allows deletion of data.We may assign the user all, none, or a combination of these types of authorization.

The DDL, just like any other programming language, gets as input some instructions (statements) and generates some output. The output of the DDL is placed in the **data dictionary**,which contains **metadata**—that is, data about data. The data dictionary is considered to be a special type of table that can only be accessedand updated by the database system itself (not a regular user). The database system consults the data dictionary before reading or modifying actual data.

Data Dictionary

We can define a data dictionary as a DBMS component that stores the definition of data characteristics and relationships. You may recall that such “data about data” were labeled metadata. The DBMS data dictionaryprovides the DBMS with its self describing characteristic. In effect, the data dictionary resembles and X-ray ofthe company’s entire data set, and is a crucial element in the data administration function.
The two main types of data dictionary exist, integrated and stand alone. An integrated data dictionary is included with the DBMS. For example, all relational DBMSs include a built in data dictionary or system catalog that is frequently accessed and updated by the RDBMS. Other DBMSs especially older types, do not have abuilt in data dictionary instead the DBA may use third party stand alone data dictionary systems.
Data dictionaries can also be classified as active or passive. An active data dictionary is automatically updated by the DBMS with every database access, thereby keeping its access information up-to-date. A passive datadictionary is not updated automatically and usually requires a batch process to be run. Data dictionary access information is normally used by the DBMS for query optimization purpose.
The data dictionary’s main function is to store the description of all objects that interact with the database. Integrated data dictionaries tend to limit their metadata to the data managed by the DBMS. Stand alone datadictionary systems are more usually more flexible and allow the DBA to describe and manage all the organization’s data, whether or not they are computerized. Whatever the data dictionary’s format, its existence provides database designers and end users with a much improved ability to communicate. In addition, the data dictionary is the tool that helps the DBA to resolve data conflicts.
Although, there is no standard format for the information stored in the data dictionary several features are common. For example, the data dictionary typically stores descriptions of all:

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

Database Administrators and Database Users

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

Database Users and User Interfaces

There are four different types of database-system users, differentiated by the way they expect to interact withthe system. Different types of user interfaces have been designed for the different types of users.
Naive users are unsophisticated users who interact with the system by invoking one of the application programs that have been written previously. For example, a bank teller who needs to transfer $50 from account *A* to account *B* invokes a program called *transfer*. This program asks the teller for the amount of money to betransferred, the account from which the money is to be transferred, and the account to which the money is to be transferred.

As another example, consider a user who wishes to find her account balance over the World Wide Web. Such a user may access a form, where she enters her account number. An application program at the Web server then retrieves the account balance, using the given account number, and passes this information back to the user.The typical user interface for naive users is a forms interface, where the user can fill in appropriate fields of the form. Naive users may also simply read *reports* generated from the database.

Application programmers are computer professionals who write application programs. Application programmers can choose from many tools to develop user interfaces. **Rapid application development (RAD)** tools are tools that enable an application programmer to construct forms and reports without writing a program. There are also special types of programming languages that combine imperative control structures (for example, for loops, while loops and if-then-else statements) with statements of the data manipulation language. These languages, sometimes called *fourth-generation languages*, often include special features to facilitate the generation of forms and the display of data on the screen. Most major commercial database systems include a fourth generation language.

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

Online analytical processing (OLAP) tools simplify analysts' tasks by letting them view summaries of data in different ways. For instance, an analyst can see total sales by region (for example, North, South, East, and West), or by product, or by a combination of region and product (that is, total sales of each product in each region). The tools also permit the analyst to select specific regions, look at data in more detail (for example, sales by city within a region) or look at the data in less detail (for example, aggregate products together by category).

Another class of tools for analysts is **data mining** tools, which help them find certain kinds of patterns in data. **Specialized users** are sophisticated users who write specialized database applications that do not fit into the traditional data-processing framework.

Among these applications are computer-aided design systems, knowledge base and expert systems, systems that store data with complex data types (for example, graphics data and audio data), and environment-modeling systems.

Database Architecture:

We are now in a position to provide a single picture (Figure 1.3) of the various components of a database system and the connections among them.

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

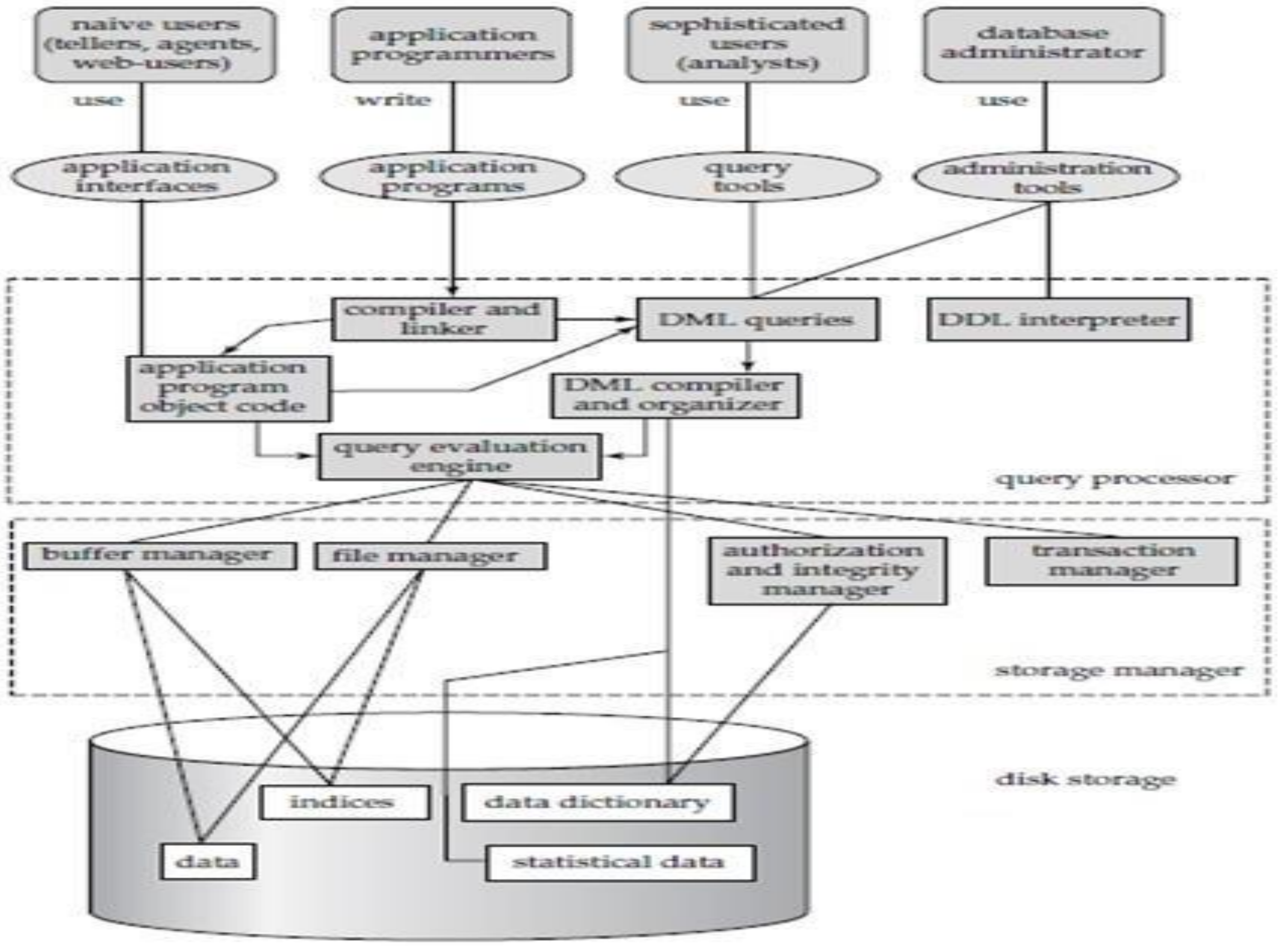


Figure 1.3: Database System Architecture

A database system is partitioned into modules that deal with each of the responsibilities of the overall system. The functional components of a database system can be broadly divided into the **storage manager** and the **query processor** components. The storage manager is important because databases typically require a large amount of storage space. The query processor is important because it helps the database system simplify and facilitate access to data.

It is the job of the database system to translate updates and queries written in a nonprocedural language, at the logical level, into an efficient sequence of operations at the physical level.

Database applications are usually partitioned into two or three parts, as in Figure 1.4. In a two-tier architecture, the application resides at the client machine, where it invokes database system functionality at the server machine through query language statements. Application program interface standards like ODBC and JDBC are used for interaction between the client and the server. In contrast, in a three-tier architecture, the client machine acts as merely a front end and does not contain any direct database calls. Instead, the client end communicates with an application server, usually through a forms interface.

The application server in turn communicates with a database system to access data. The business logic of the application, which says what actions to carry out under what conditions, is embedded in the application server, instead of being distributed across multiple clients. Three-tier applications are more appropriate for large applications, and for applications that run on the WorldWideWeb.

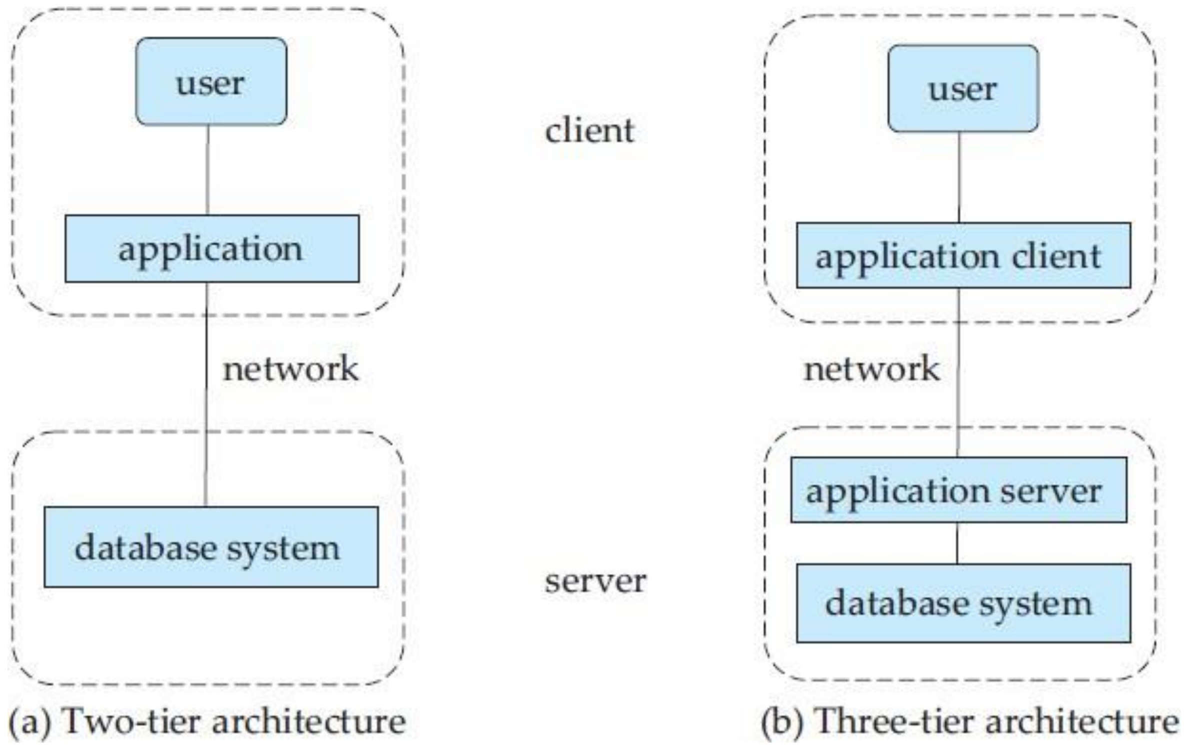


Figure 1.4: Two-tier and three-tier architectures.

Query Processor:

The query processor components include

· **DDL interpreter,** which interprets DDL statements and records the definitions in the data dictionary.

· **DML compiler,** which translates DML statements in a query language into an evaluation plan consisting of low-level instructions that the query evaluation engine understands.

A query can usually be translated into any of a number of alternative evaluation plans that all give the same result. The DML compiler also performs **query optimization**, that is, it picks the lowest cost evaluation plan from among the alternatives.

Query evaluation engine, which executes low-level instructions generated by the DML compiler.

Storage Manager:

A *storage manager* is a program module that provides the interface between the lowlevel data stored in the database and the application programs and queries submitted to the system. The storage manager is responsible for the interaction with the file manager. The raw data are stored on the disk using the file system, which is usually provided by a conventional operating system. The storage manager translates the various DML statements into low-level file-system commands. Thus, the storage manager is responsible for storing, retrieving, and updating data in the database.

The storage manager components include:

· **Authorization and integrity manager**, which tests for the satisfaction of integrity constraints and checks the authority of users to access data.
· **Transaction manager**, which ensures that the database remains in a consistent (correct) state despitesystem failures, and that concurrent transaction executions proceed without conflicting.
· **File manager**, which manages the allocation of space on disk storage and the data structures used to represent information stored on disk.
· **Buffer manager**, which is responsible for fetching data from disk storage into main memory, and deciding what data to cache in main memory. The buffer manager is a critical part of the database system, since it enables the database to handle data sizes that are much larger than the size of main memory.

Transaction Manager:

A **transaction** is a collection of operations that performs a single logical function in a database application. Each transaction is a unit of both atomicity and consistency. Thus, we require that transactions do not violateany database-consistency constraints. That is, if the database was consistent when a transaction started, thedatabase must be consistent when the transaction successfully terminates. **Transaction - manager** ensures that the database remains in a consistent (correct) state despite system failures (e.g., power failures and operating system crashes) and transaction failures.

UNIT II

DATABASE DESIGN

Conceptual Database Design - Entity Relationship(ER) Modeling:

Database Design Techniques
1. ER Modeling (Top down Approach)
2. Normalization (Bottom Up approach)

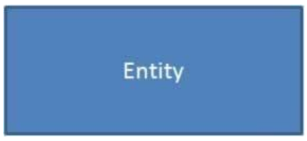
What is ER Modeling?

A graphical technique for understanding and organizing the data independent of the actual database implementation

We need to be familiar with the following terms to go further.

Entity

Any thing that has an independent existence and about which we collect data. It is also known as entity type. In ER modeling, notation for entity is given below.



Entity instance

Entity instance is a particular member of the entity type.Example for entity instance : A particular employee **Regular Entity**

An entity which has its own key attribute is a regular entity.

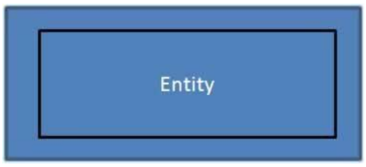
Example for regular entity : Employee.

Weak entity

An entity which depends on other entity for its existence and doesn't have any key attribute of its own is a weak entity.

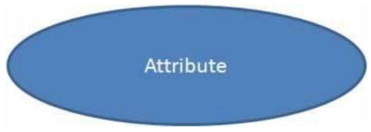
Example for a weak entity : In a parent/child relationship, a parent is considered as a strong entity and the child is a weak entity.

In ER modeling, notation for weak entity is given below.



Attributes

Properties/characteristics which describe entities are called attributes.In ER modeling, notation for attribute is given below.



Domain of Attributes

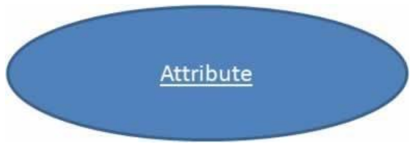
The set of possible values that an attribute can take is called the domain of the attribute. For example, the attribute day may take any value from the set {Monday, Tuesday ... Friday}. Hence this set can be termed as the domain of the attribute day.

Key attribute

The attribute (or combination of attributes) which is unique for every entity instance is called key attribute.

E.g the employee_id of an employee, pan_card_number of a person etc.If the key attribute consists of two or more attributes in combination, it is called a composite key.

In ER modeling, notation for key attribute is given below.



Simple attribute

If an attribute cannot be divided into simpler components, it is a simple attribute. Example for

simple attribute : employee_id of an employee.

Composite attribute

If an attribute can be split into components, it is called a composite attribute.

Example for composite attribute : Name of the employee which can be split into First_name, Middle_name, and Last_name.

Single valued Attributes:

If an attribute can take only a single value for each entity instance, it is a single valued attribute. example for

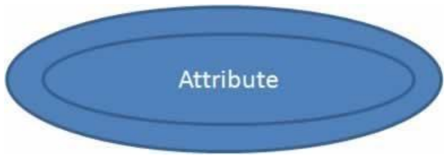
single valued attribute : age of a student. It can take only one value for a particular student. **Multi-valued**

Attributes

If an attribute can take more than one value for each entity instance, it is a multi-valued attribute. Multi-valued

example for multi valued attribute : telephone number of an employee, a particular employee may have multiple telephone numbers.

In ER modeling, notation for multi-valued attribute is given below.



Stored Attribute

An attribute which need to be stored permanently is a stored attributeExample

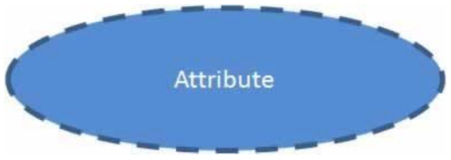
for stored attribute : name of a student

Derived Attribute

An attribute which can be calculated or derived based on other attributes is a derived attribute.

Example for derived attribute : age of employee which can be calculated from date of birth and current date. In ER

modeling, notation for derived attribute is given below.

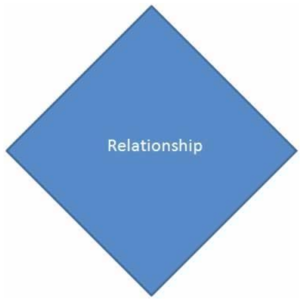


Relationships

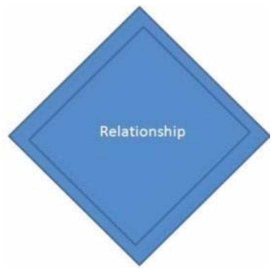
Associations between entities are called relationships

Example : An employee works for an organization. Here "works for" is a relation between the entities employee and organization.

In ER modeling, notation for relationship is given below.



However in ER Modeling, To connect a weak Entity with others, you should use a weak relationship notation as given below



Degree of a Relationship

Degree of a relationship is the number of entity types involved. The n-ary relationship is the general form for degree n. Special cases are unary, binary, and ternary ,where the degree is 1, 2, and 3, respectively.

Example for unary relationship : An employee ia a manager of another employeeExample for binary relationship :

An employee works-for department.

Example for ternary relationship : customer purchase item from a shop keeper

Cardinality of a Relationship

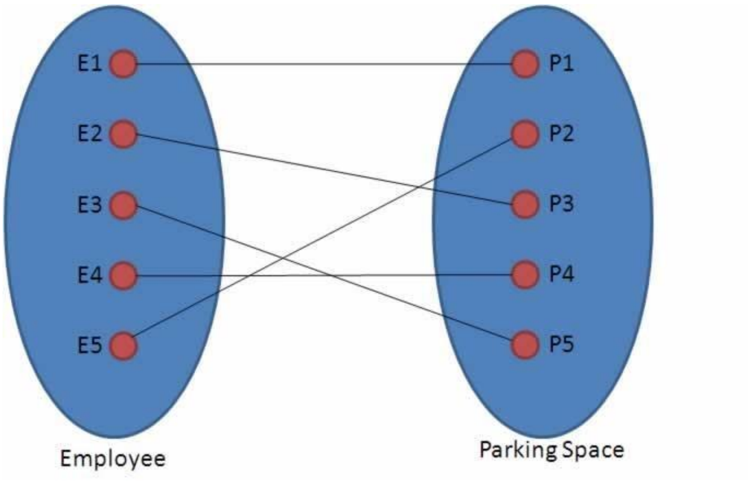
Relationship cardinalities specify how many of each entity type is allowed. Relationships can have four possible connectivities as given below.

1. One to one (1:1) relationship
2. One to many (1:N) relationship
3. Many to one (M:1) relationship
4. Many to many (M:N) relationship

The minimum and maximum values of this connectivity is called the cardinality of the relationship

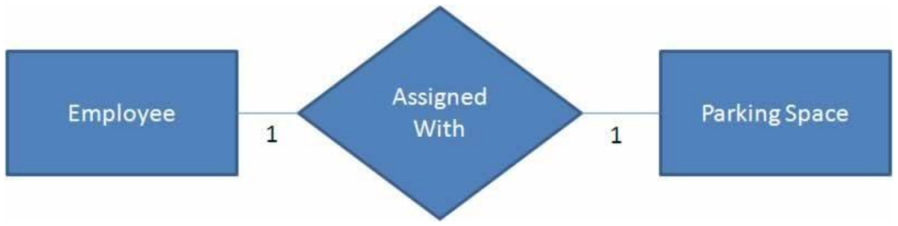
Example for Cardinality – One-to-One (1:1)

Employee is assigned with a parking space.



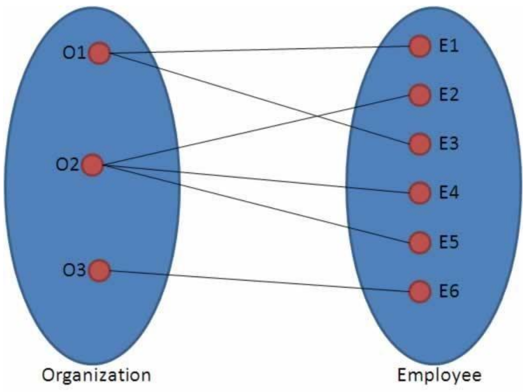
One employee is assigned with only one parking space and one parking space is assigned to only one employee. Hence it is a 1:1 relationship and cardinality is One-To-One (1:1)

In ER modeling, this can be mentioned using notations as given below



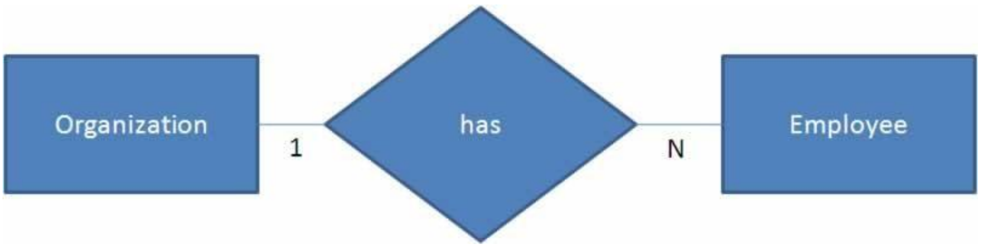
Example for Cardinality – One-to-Many (1:N)

Organization has employees



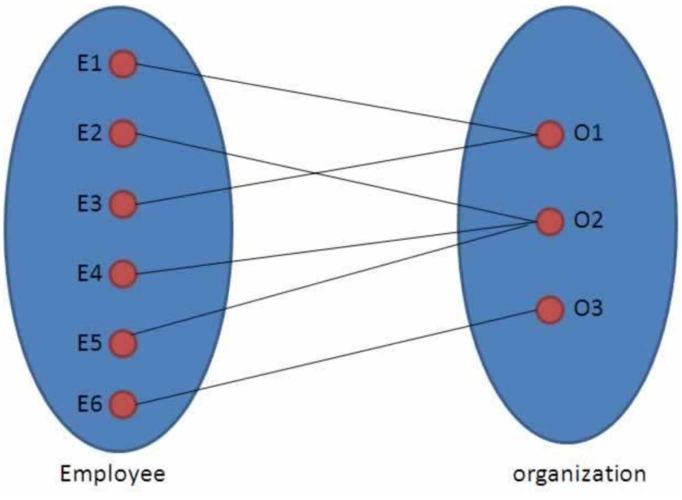
One organization can have many employees , but one employee works in only one organization. Hence it is a 1:N relationship and cardinality is One-To-Many (1:N)

In ER modeling, this can be mentioned using notations as given below



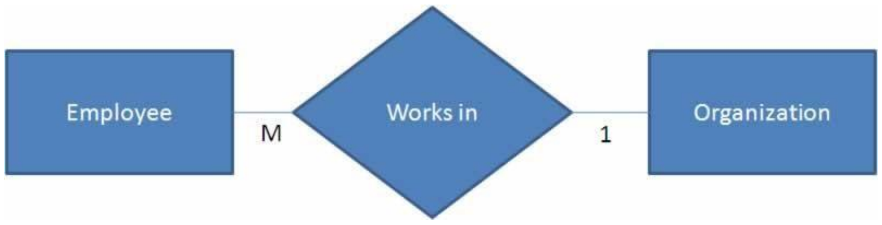
Example for Cardinality – Many-to-One (M :1)

It is the reverse of the One to Many relationship. employee works in organization



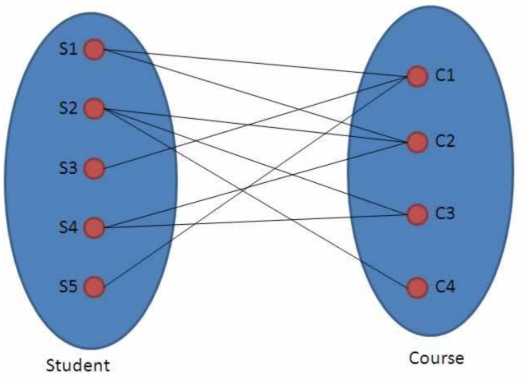
One employee works in only one organization But one organization can have many employees. Hence it is a M:1 relationship and cardinality is Many-to-One (M :1)

In ER modeling, this can be mentioned using notations as given below.



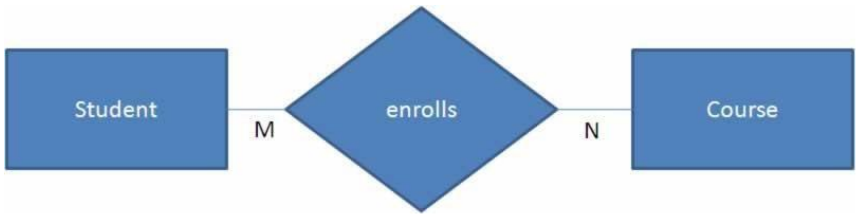
Cardinality – Many-to-Many (M:N)

Students enrolls for courses



One student can enroll for many courses and one course can be enrolled by many students. Hence it is a M:N relationship and cardinality is Many-to-Many (M:N)

In ER modeling, this can be mentioned using notations as given below



Relationship Participation

1. Total

In total participation, every entity instance will be connected through the relationship to another instance of the other participating entity types

2. Partial

Example for relationship participation

Consider the relationship - Employee is head of the department.

Here all employees will not be the head of the department. Only one employee will be the head of the department. In other words, only few instances of employee entity participate in the above relationship. So employee entity's participation is partial in the said relationship.

However each department will be headed by some employee. So department entity's participation is total in the said relationship.

Advantages and Disadvantages of ER Modeling (Merits and Demerits of ER Modeling)

Advantages

1. ER Modeling is simple and easily understandable. It is represented in business users language and it can be understood by non-technical specialist.

2. Intuitive and helps in Physical Database creation.

3. Can be generalized and specialized based on needs.

4. Can help in database design.

5. Gives a higher level description of the system.

Disadvantages

1. Physical design derived from E-R Model may have some amount of ambiguities or inconsistency.

2. Sometime diagrams may lead to misinterpretations

Relational Model

The relational model is today the primary data model for commercial data processing applications. It attained its primary position because of its simplicity, which eases the job of the programmer, compared to earlier data models such as the network model or the hierarchical model. In this, we first study the fundamentals of the relational model. A substantial theory exists for relational databases.

Structure of Relational Databases:

A relational database consists of a collection of **tables**, each of which is assigned a unique name. For example, consider the *instructor* table of Figure:1.5, which stores information about instructors. The table has four column headers: *ID*, *name*, *dept name*, and *salary*. Each row of this table records information about an instructor, consisting of the instructor's *ID*, *name*, *dept name*, and *salary*. Similarly, the *course* table of Figure 1.6 storesinformation about courses, consisting of a *course id*, *title*, *dept name*, and *credits*, for each course. Note that each instructor is identified by the value of the column *ID*, while each course is identified by the value of the column *course id*.

Figure 1.7 shows a third table, *prereq*, which stores the prerequisite courses for each course. The table has two columns, *course id* and *prereq id*. Each row consists of a pair of course identifiers such that the second course is a prerequisite for the first course.

Thus, a row in the *prereq* table indicates that two courses are *related* in the sense that one course is a prerequisite for the other. As another example, we consider the table *instructor*, a row in the table can be thought of as representing the relationship between a specified *ID* and the corresponding values for *name*,*dept name*,and *salary* value

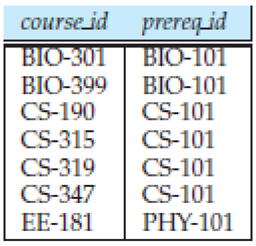
ID	*name*	*dept_name*	*salary*
10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	60000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	62000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000

Figure 1.5: The *instructor* relation (2.1)

In general, a row in a table represents a *relationship* among a set of values. Since a table is a collection of such relationships, there is a close correspondence between the concept of *table* and the mathematical concept of *relation*, from which the relational data model takes its name. In mathematical terminology, a *tuple* is simply asequence (or list) of values. A relationship between *n* values is represented mathematically by an *n-tuple* of values, i.e., a tuple with *n* values, which corresponds to a row in a table.

course_id	*title*	*dept_name*	*credits*
BIO-101	Intro. to Biology	Biology	4
BIO-301	Genetics	Biology	4
BIO-399	Computational Biology	Biology	3
CS-101	Intro. to Computer Science	Comp. Sci.	4
CS-190	Game Design	Comp. Sci.	4
CS-315	Robotics	Comp. Sci.	3
CS-319	Image Processing	Comp. Sci.	3
CS-347	Database System Concepts	Comp. Sci.	3
EE-181	Intro. to Digital Systems	Elec. Eng.	3
FIN-201	Investment Banking	Finance	3
HIS-351	World History	History	3
MU-199	Music Video Production	Music	3
PHY-101	Physical Principles	Physics	4

Figure: 1.6: The *course* relation (2.2)



course_id	*prereq_id*
BIO-301	BIO-101
BIO-399	BIO-101
CS-190	CS-101
CS-315	CS-101
CS-319	CS-101
CS-347	CS-101
EE-181	PHY-101

Figure: 1.7: The *prereq* relation. **(2.3)**

Thus, in the relational model the term **relation** is used to refer to a table, while the term **tuple** is used to refer toa row. Similarly, the term **attribute** refers to a column of a table.

Examining Figure 1.5, we can see that the relation *instructor* has four attributes:

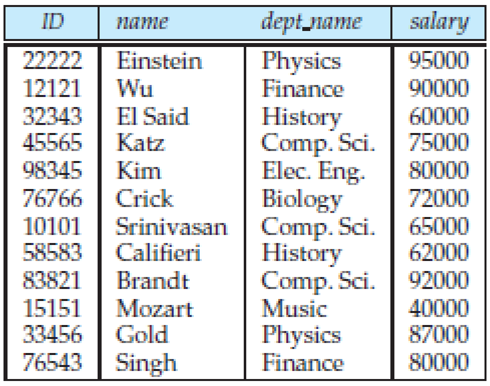
ID, name, dept name,* and *salary.

We use the term **relation instance** to refer to a specific instance of a relation, i.e., containing a specific set ofrows. The instance of *instructor* shown in Figure 1.5 has 12 tuples, corresponding to 12 instructors.

In this topic, we shall be using a number of different relations to illustrate the various concepts underlying the relational data model. These relations represent part of a university. They do not include all the data an actualuniversity database would contain, in order to simplify our presentation.

The order in which tuples appear in a relation is irrelevant, since a relation is a *set* of tuples. Thus, whether the tuples of a relation are listed in sorted order, as in Figure 1.5, or are unsorted, as in Figure 1.8, does not matter; the relations in the two figures are the same, since both contain the same set of tuples. For ease of exposition,we will mostly show the relations sorted by their first attribute. For each attribute of a relation, there is a set of permitted values, called the **domain** of that attribute. Thus, the domain of the *salary* attribute of the *instructor* relation is the set of all possible salary values, while the domain of the *name* attribute is the set of all possible instructor names.

We require that, for all relations *r*, the domains of all attributes of *r* be atomic. A domain is **atomic** if elements of the domain are considered to be indivisible units.



ID	*name*	*dept_name*	*salary*
22222	Einstein	Physics	95000
12121	Wu	Finance	90000
32343	El Said	History	60000
45565	Katz	Comp. Sci.	75000
98345	Kim	Elec. Eng.	80000
76766	Crick	Biology	72000
10101	Srinivasan	Comp. Sci.	65000
58583	Califieri	History	62000
83821	Brandt	Comp. Sci.	92000
15151	Mozart	Music	40000
33456	Gold	Physics	87000
76543	Singh	Finance	80000

Figure: 1.8: Unsorted display of the *instructor* relation. (2-4)

For example, suppose the table *instructor* had an attribute *phone number*, which can store a set of phone numbers corresponding to the instructor. Then the domain of *phone number* would not be atomic, since an element of the domain is a set of phone numbers, and it has subparts, namely the individual phone numbers in the set.

The important issue is not what the domain itself is, but rather how we use domain elements in our database. Suppose now that the *phone number* attribute stores a single phone number. Even then, if we split the value from the phone number attribute into a country code, an area code and a local number, we would be treating it as a nonatomic value. If we treat each phone number as a single indivisible unit, then the attribute *phone number* would have an atomic domain.

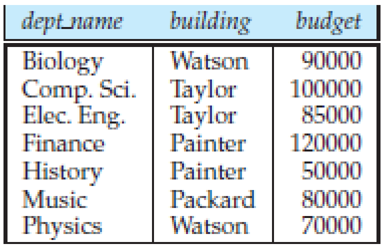
The **null** value is a special value that signifies that the value is unknown or does not exist. For example, suppose as before that we include the attribute *phone number* in the *instructor* relation. It may be that an instructor does not have a phone number at all, or that the telephone number is unlisted. We would then have to use the nullvalue to signify that the value is unknown or does not exist. We shall see later that null values cause a number of difficulties when we access or update the database, and thus should be eliminated if at all possible. We shall assume null values are absent initially.

Database Schema

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

instant in time. The concept of a relation corresponds to the programming-language notion of a variable, whilethe concept of a **relation schema** corresponds to the programming-language notion of type definition.

In general, a relation schema consists of a list of attributes and their corresponding domains. The concept of arelation instance corresponds to the programming-language notion of a value of a variable. The value of a given variable may change with time;



dept_name	*building*	*budget*
Biology	Watson	90000
Comp. Sci.	Taylor	100000
Elec. Eng.	Taylor	85000
Finance	Painter	120000
History	Painter	50000
Music	Packard	80000
Physics	Watson	70000

Figure 1.9: The *department* relation.(2-5)

similarly the contents of a relation instance may change with time as the relation is updated. In contrast, the schema of a relation does not generally change. Although it is important to know the difference between a relation schema and a relation instance, we often use the same name, such as *instructor*, to refer to both the schema and the instance. Where required, we explicitly refer to the schema or to the instance, for example "the *instructor* schema," or "an instance of the *instructor* relation." However, where it is clear whether we mean theschema or the instance, we simply use the relation name.

Consider the *department* relation of Figure 1.9. The schema for that relation is

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

Note that the attribute *dept name* appears in both the *instructor* schema and the *department* schema. This duplication is not a coincidence. Rather, using common attributes in relation schemas is one way of relating tuples of distinct relations.

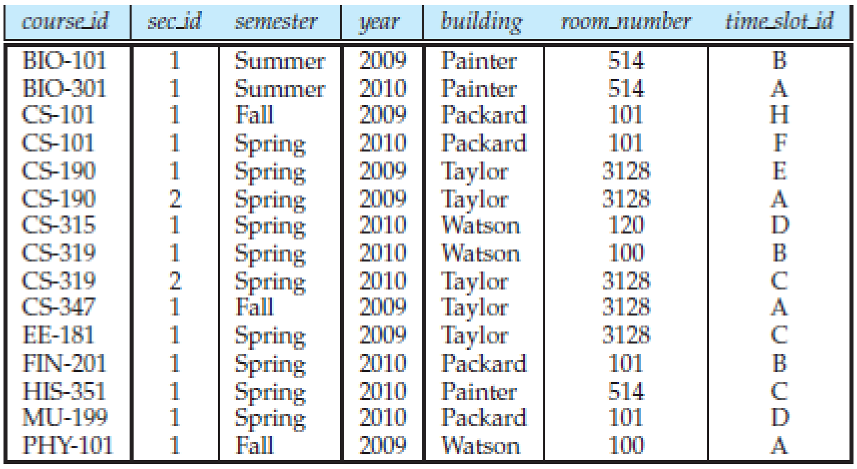
For example, suppose we wish to find the information about all the instructors who work in the Watson building. We look first at the *department* relation to find the *dept name* of all the departments housed in Watson. Then, for each such department, we look in the *instructor* relation to find the information about the instructor associatedwith the corresponding *dept name*.

Let us continue with our university database example. Each course in a university may be offered multiple times, across different semesters, or even within a semester.We need a relation to describe each individual offering, or section, of the class. The schema is

***section* (*course id, sec id, semester, year, building, room number, time slot id*)**

Figure 1.10 shows a sample instance of the *section* relation. We need a relation to describe the association between instructors and the class sections that they teach. The relation schema to describe this association is

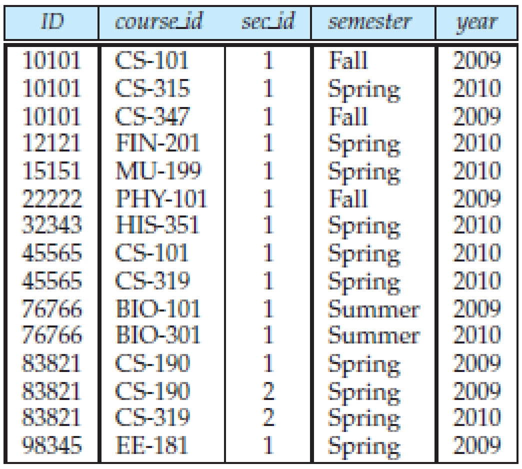
***teaches* (*ID, course id, sec id, semester, year*)**



course_id	*sec_id*	*semester*	*year*	*building*	*room_number*	*time_slot_id*
BIO-101	1	Summer	2009	Painter	514	B
BIO-301	1	Summer	2010	Painter	514	A
CS-101	1	Fall	2009	Packard	101	H
CS-101	1	Spring	2010	Packard	101	F
CS-190	1	Spring	2009	Taylor	3128	E
CS-190	2	Spring	2009	Taylor	3128	A
CS-315	1	Spring	2010	Watson	120	D
CS-319	1	Spring	2010	Watson	100	B
CS-319	2	Spring	2010	Taylor	3128	C
CS-347	1	Fall	2009	Taylor	3128	A
EE-181	1	Spring	2009	Taylor	3128	C
FIN-201	1	Spring	2010	Packard	101	B
HIS-351	1	Spring	2010	Painter	514	C
MU-199	1	Spring	2010	Packard	101	D
PHY-101	1	Fall	2009	Watson	100	A

Figure 1.10: The *section* relation.(2-6)

Figure 1.11 shows a sample instance of the *teaches* relation. As you can imagine, there are many more relations maintained in a real university database. In addition to those relations we have listed already, *instructor*, *department*, *course*, *section*, *prereq*, and *teaches*,we use the following relations in this text:



ID	*course_id*	*sec_id*	*semester*	*year*
10101	CS-101	1	Fall	2009
10101	CS-315	1	Spring	2010
10101	CS-347	1	Fall	2009
12121	FIN-201	1	Spring	2010
15151	MU-199	1	Spring	2010
22222	PHY-101	1	Fall	2009
32343	HIS-351	1	Spring	2010
45565	CS-101	1	Spring	2010
45565	CS-319	1	Spring	2010
76766	BIO-101	1	Summer	2009
76766	BIO-301	1	Summer	2010
83821	CS-190	1	Spring	2009
83821	CS-190	2	Spring	2009
83821	CS-319	2	Spring	2010
98345	EE-181	1	Spring	2009

Figure: 1.11: The *teaches* relation.(2-7)

- *student* (*ID*, *name*, *dept name*, *tot cred*)
- *advisor* (*s id*, *i id*)
- *takes* (*ID*, *course id*, *sec id*, *semester*, *year*, *grade*)
- *classroom* (*building*, *room number*, *capacity*)
- *time slot* (*time slot id*, *day*, *start time*, *end time*)

Keys

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

A **superkey** is a set of one or more attributes that, taken collectively, allow us to identify uniquely a tuple in therelation. For example, the *ID* attribute of the relation *instructor* is sufficient to distinguish one instructor tuple from another. Thus, *ID* is a superkey. The *name* attribute of *instructor*, on the other hand, is not a superkey, because several instructors might have the same name. Formally, let *R* denote the set of attributes in the schema of relation *r*. If we say that a subset *K* of *R* is a *superkey* for *r* , we are restricting consideration to instances of relations *r* in which no two distinct tuples have the same values on all attributes in *K*. That is, if $t1$ and $t2$ are in *r*and $t1 = t2$, then $t1.K = t2.K$.

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

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

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

Primary keys must be chosen with care. As we noted, the name of a person is obviously not sufficient, because there may be many people with the same name. In the United States, the social-security number attribute of aperson would be a candidate key. Since non-U.S. residents usually do not have social-security numbers, international enterprises must generate their own unique identifiers.

An alternative is to use some unique combination of other attributes as a key. The primary key should be chosen such that its attribute values are never, or very rarely, changed. For instance, the address field of a person should not be part of the primary key, since it is likely to change. Social-security numbers, on the other hand, are guaranteed never to change. Unique identifiers generated by enterprises generally do not change, except if two enterprises merge; in such a case the same identifier may have been issued by both enterprises, and a reallocation of identifiers may be required to make sure they are unique.

It is customary to list the primary key attributes of a relation schema before the other attributes; for example, the *dept name* attribute of *department* is listed first, since it is the primary key. Primary key attributes are also underlined. A relation, say $r1$, may include among its attributes the primary key of another relation, say $r2$. Thisattribute is called a **foreign key** from $r1$, referencing $r2$.

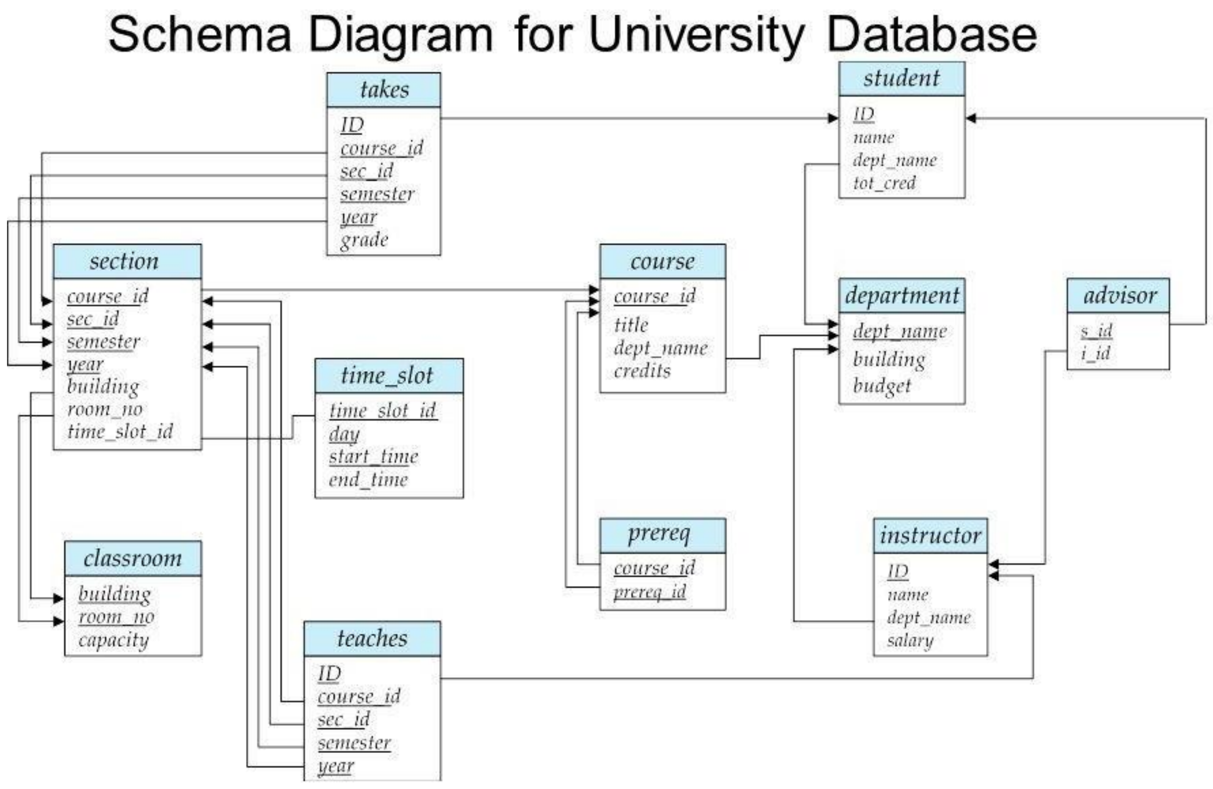
The relation $r1$ is also called the **referencing relation** of the foreign key dependency, and $r2$ is called the**referenced relation** of the foreign key. For example, the attribute *dept name* in *instructor* is a foreign key from*instructor*, referencing *department*, since *dept name* is the primary key of *department*. In any database instance, given any tuple, say ta, from the *instructor* relation, there must be some tuple, say tb, in the *department* relationsuch that the value of the *dept name* attribute of ta is the same as the value of the primary key, *dept name*, of tb.

Now consider the *section* and *teaches* relations. It would be reasonable to require that if a section exists for acourse, it must be taught by at least one instructor; however, it could possibly be taught by more than one instructor. To enforce this constraint, we would require that if a particular (*course id*, *sec id*, *semester*, *year*) combination appears in *section*, then the same combination must appear in *teaches*. However, this set of values does not form a primary key for *teaches*, since more than one instructor may teach one such section. As a result, we cannot declare a foreign key constraint from *section* to *teaches* (although we can define a foreign key constraint in the other direction, from *teaches* to *section*).

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

Schema Diagrams

A database schema, along with primary key and foreign key dependencies, can be depicted by **schemadiagrams**. Figure 1.12 shows the schema diagram for our university organization. Each relation appears as abox, with the relation name at the top in blue, and the attributes listed inside the box. Primary key attributes are shown underlined. Foreign key dependencies appear as arrows from the foreign key attributes of the referencing relation to the primary key of the referenced relation.

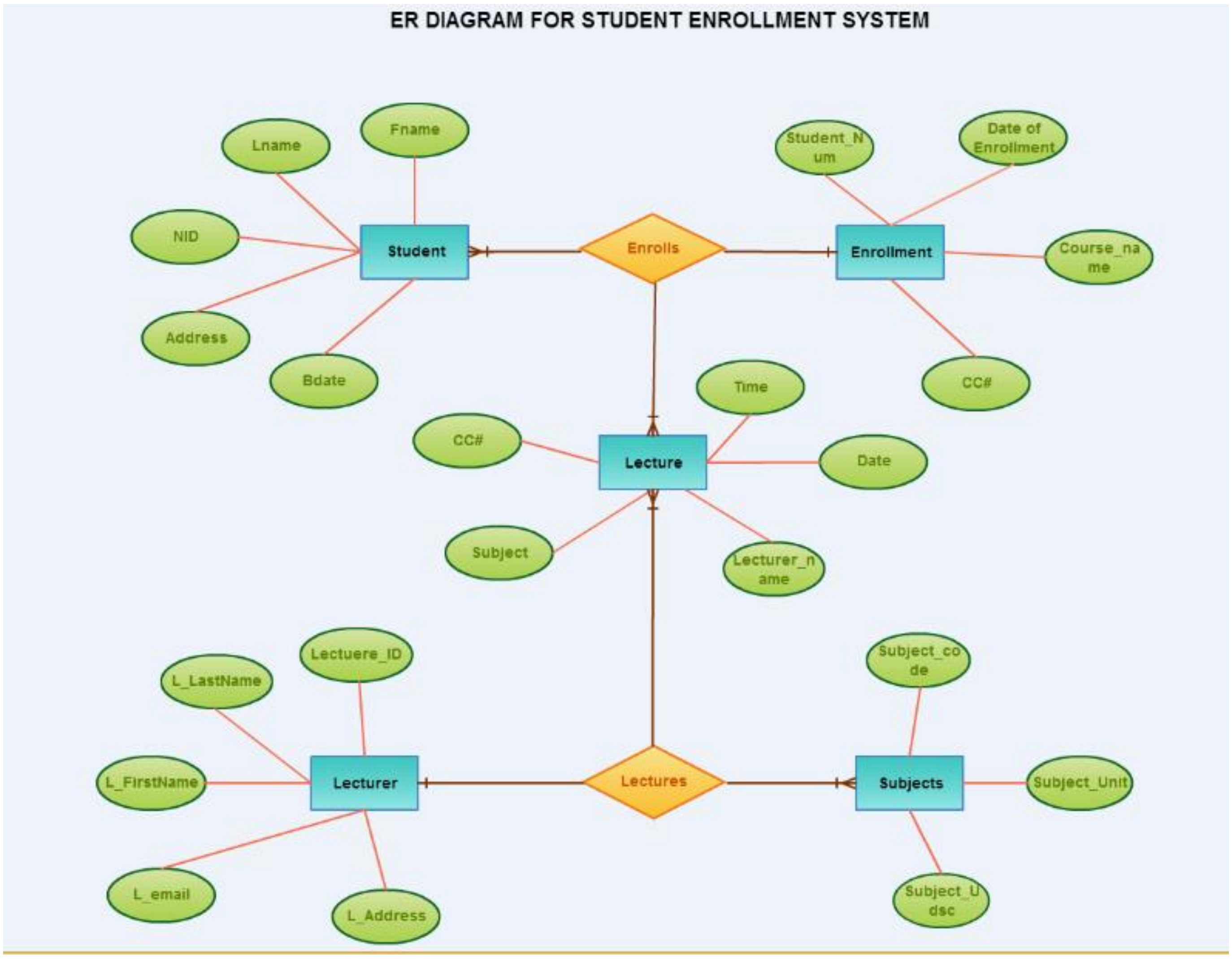


Referential integrity constraints other than foreign key constraints are not shown explicitly in schema diagrams. We will study a different diagrammatic representation called the entity-relationship diagram.

Introduction to relational model-Tabular, Representation of Various ER Schemas

What is an ER diagram?

An Entity Relationship Diagram (ERD) is a visual representation of **different entities within a system and how they relate to each other**. For example, the elements writer, novel, and a consumer may be described using ER diagrams the following way:



Specific objectives of an ER diagram are to provide: **A unified view of the entities, attributes, and relationships that comprise the overall data structure of the database**.

Weak entity set in ER Diagram:

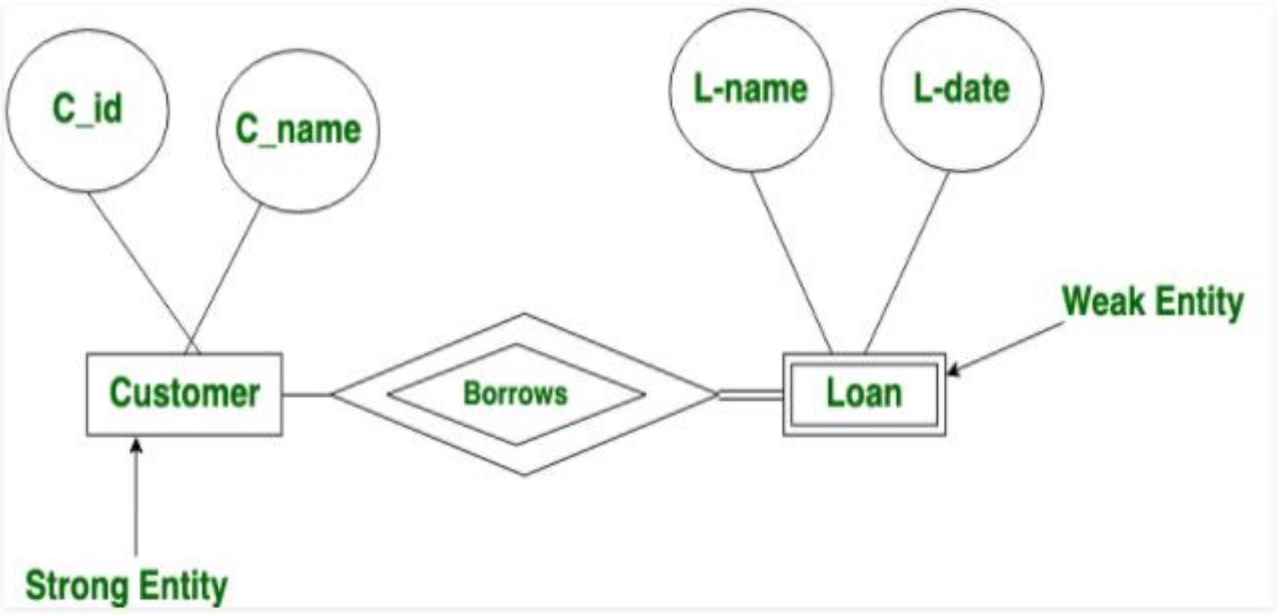
An entity type should have a key attribute which uniquely identifies each entity in the entity set, but there exists some entity type for which key attribute can't be defined. These are called Weak Entity type.

The entity sets which do not have sufficient attributes to form a primary key are known as **weak entity sets** and the entity sets which have a primary key are known as strong entity sets.

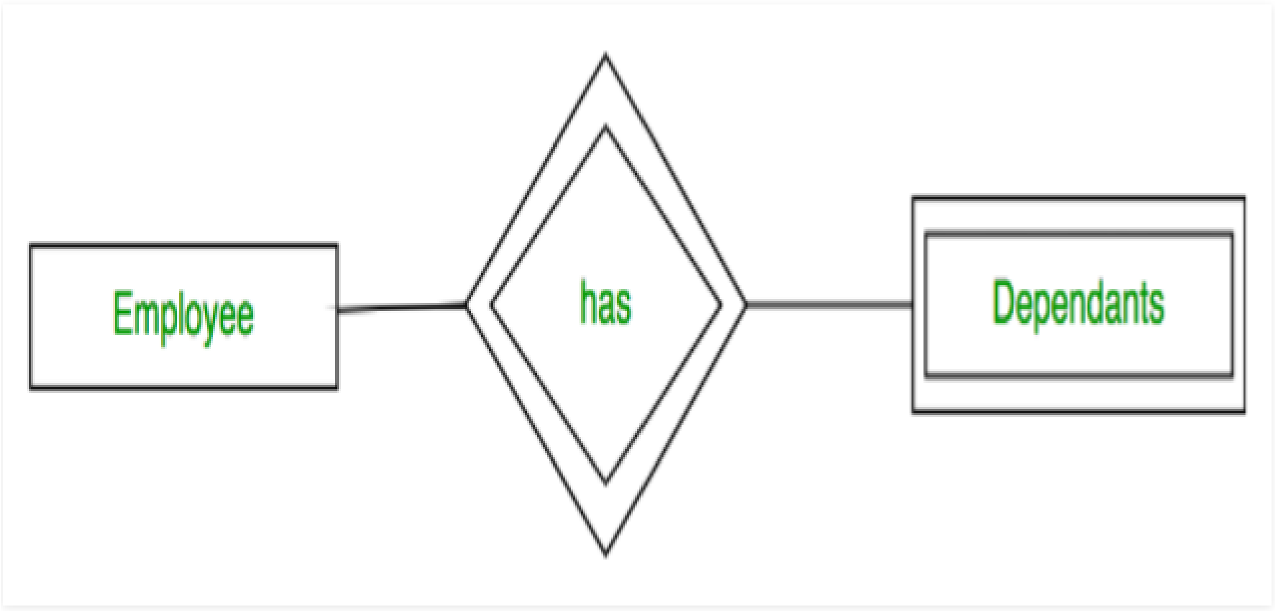
As the weak entities do not have any primary key, they cannot be identified on their own, so they depend on some other entity (known as owner entity). The weak entities have total participation constraint (existence dependency) in its identifying relationship with owner identity. Weak entity types have partial keys. Partial Keys are set of attributes with the help of which the tuples of the weak entities can be distinguished and identified.

Note – Weak entity always has total participation but Strong entity may not have total participation.

Weak entity is **depend on strong entity** to ensure the existence of weak entity. Like strong entity, weak entity does not have any primary key, It has partial discriminator key. Weak entity is represented by double rectangle. The relation between one strong and one weak entity is represented by double diamond.

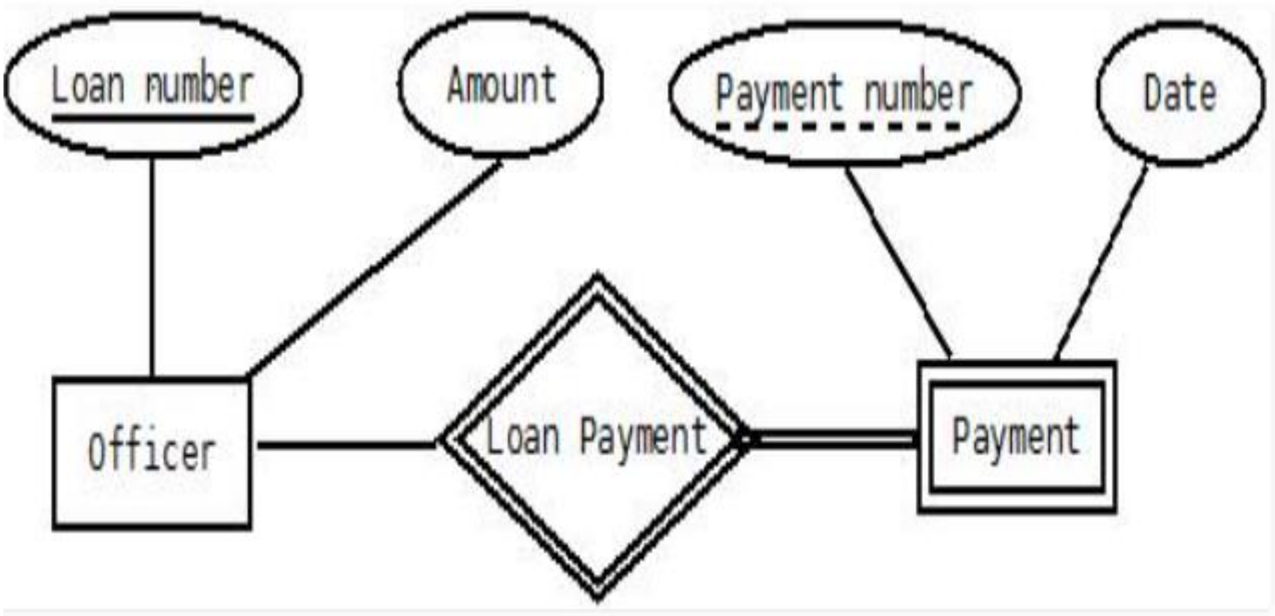


Weak entities are represented with **double rectangular** box in the ER Diagram and the identifying relationships are represented with double diamond. Partial Key attributes are represented with dotted lines.



Example-1:

In the below ER Diagram, 'Payment' is the weak entity. 'Loan Payment' is the identifying relationship and 'Payment Number' is the partial key. Primary Key of the Loan along with the partial key would be used to identify the records.



Views in SQL

- Views in SQL are considered as a virtual table. A view also contains rows and columns.
- To create the view, we can select the fields from one or more tables present in the database.
- A view can either have specific rows based on certain condition or all the rows of a table.

Sample table:

Student_Detail

STU_ID	NAME	ADDRESS
1	Stephan	Delhi
2	Kathrin	Noida
3	David	Ghaziabad
4	Alina	Gurugram

Student_Marks

STU_ID	NAME	MARKS	AGE
1	Stephan	97	19
2	Kathrin	86	21
3	David	74	18
4	Alina	90	20
5	John	96	18

1. Creating view

A view can be created using the **CREATE VIEW** statement. We can create a view from a single table or multiple tables.

Syntax:

```
CREATE VIEW view_name AS
SELECT column1, column2.....
FROM table_name
WHERE condition;
```

2. Creating View from a single table

In this example, we create a View named DetailsView from the table Student_Detail.

Query:

```
CREATE VIEW DetailsView AS
SELECT NAME, ADDRESS
FROM Student_Details
WHERE STU_ID < 4;
```

Just like table query, we can query the view to view the data.

```
SELECT * FROM DetailsView;
```

Output:

NAME	ADDRESS
Stephan	Delhi
Kathrin	Noida
David	Ghaziabad

3. Creating View from multiple tables

View from multiple tables can be created by simply include multiple tables in the SELECT statement.

In the given example, a view is created named MarksView from two tables Student_Detail and Student_Marks.

Query:

```
CREATE VIEW MarksView AS
SELECT Student_Detail.NAME, Student_Detail.ADDRESS, Student_Marks.MARKS
FROM Student_Detail, Student_Mark
WHERE Student_Detail.NAME = Student_Marks.NAME;
```

To display data of View MarksView:

```
SELECT * FROM MarksView;
```

NAME	ADDRESS	MARKS
Stephan	Delhi	97
Kathrin	Noida	86
David	Ghaziabad	74
Alina	Gurugram	90

4. Deleting View

A view can be deleted using the Drop View statement.

Syntax

```
DROP VIEW view_name;
```

Example:

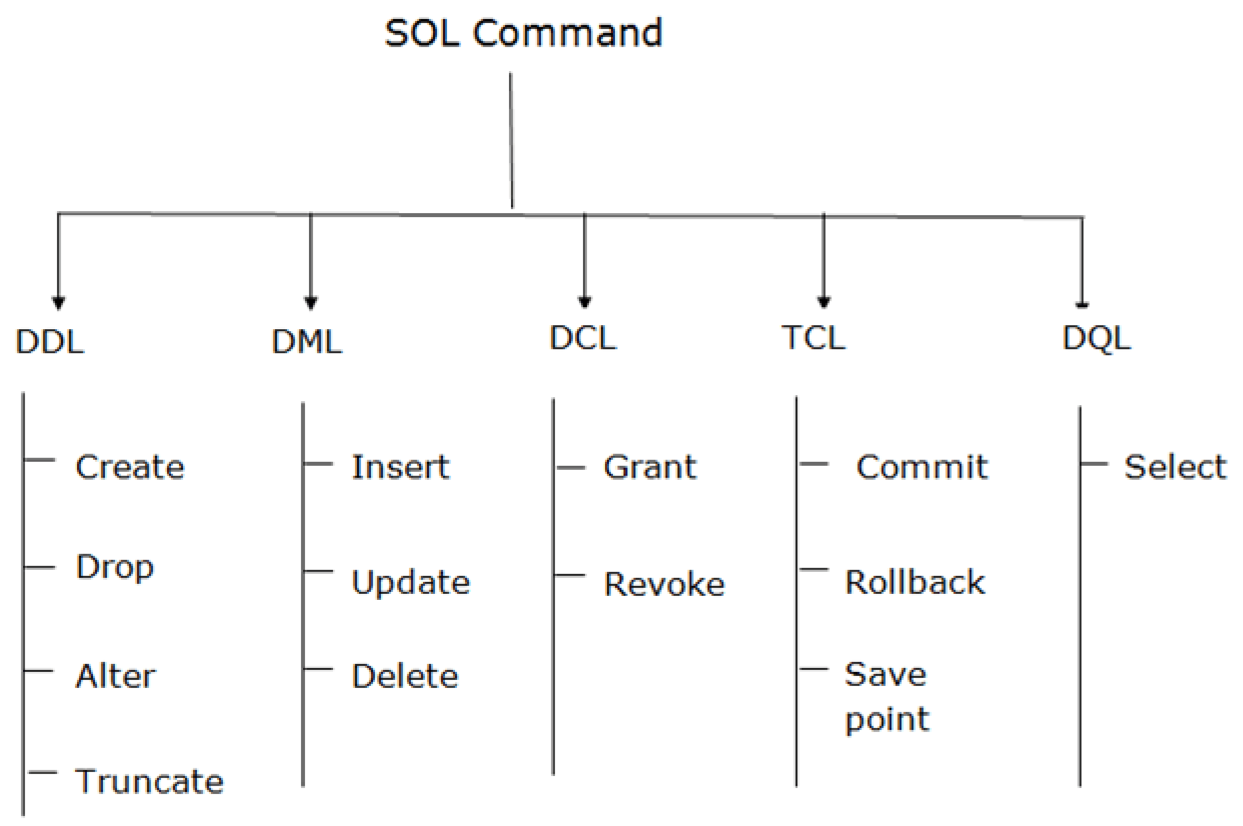
If we want to delete the View **MarksView**, we can do this as:

```
DROP VIEW MarksView;
```

UNIT III
STRUCTURED QUERY LANGUAGE

Types of SQL Commands

here are five types of SQL commands: DDL, DML, DCL, TCL, and DQL.



Data Definition Language (DDL)

- DDL changes the structure of the table like creating a table, deleting a table, altering a table, etc.
- All the command of DDL are auto-committed that means it permanently save all the changes in the database.

Here are some commands that come under DDL:

- CREATE
- ALTER

- DROP
- TRUNCATE

a. CREATE It is used to create a new table in the database.

Syntax:

CREATE TABLE TABLE_NAME (COLUMN_NAME DATATYPES[,....]);

Example:

CREATE TABLE EMPLOYEE(Name VARCHAR2(20), Email VARCHAR2(100), DOB DATE);

b. DROP: It is used to delete both the structure and record stored in the table.

Syntax

1. DROP TABLE table_name;

Example

1. DROP TABLE EMPLOYEE;

c. ALTER: It is used to alter the structure of the database. This change could be either to modify the characteristics of an existing attribute or probably to add a new attribute.

Syntax:

To add a new column in the table

1. ALTER TABLE table_name ADD column_name COLUMN-definition;

To modify existing column in the table:

1. ALTER TABLE table_name MODIFY(column_definitions....);

EXAMPLE

1. ALTER TABLE STU_DETAILS ADD(ADDRESS VARCHAR2(20));
2. ALTER TABLE STU_DETAILS MODIFY (NAME VARCHAR2(20));

d. TRUNCATE: It is used to delete all the rows from the table and free the space containing the table.

Syntax:

1. TRUNCATE TABLE table_name;

Example:

1. TRUNCATE TABLE EMPLOYEE;

2. Data Manipulation Language

- DML commands are used to modify the database. It is responsible for all form of changes in the database.
- The command of DML is not auto-committed that means it can't permanently save all the changes in the database. They can be rollback.

Here are some commands that come under DML:

- INSERT
- UPDATE
- DELETE

a. INSERT: The INSERT statement is a SQL query. It is used to insert data into the row of a table.

Syntax:

```
1. INSERT INTO TABLE_NAME
2.    (col1, col2, col3,.... col N)
3.    VALUES (value1, value2, value3, .... valueN);
```

Or

```
1. INSERT INTO TABLE_NAME
2. VALUES (value1, value2, value3, .... valueN);
```

For example:

```
1. INSERT INTO javatpoint (Author, Subject) VALUES ("Sonoo", "DBMS");
```

b. UPDATE: This command is used to update or modify the value of a column in the table.

Syntax:

```
1. UPDATE table_name SET [column_name1= value1,...column_nameN = valueN] [WHERE CONDITION]
```

For example:

```
1. UPDATE students
2. SET User_Name = 'Sonoo'
3. WHERE Student_Id = '3'
```

c. DELETE: It is used to remove one or more row from a table.

Syntax:

```
1. DELETE FROM table_name [WHERE condition];
```

For example:

```
1. DELETE FROM javatpoint
2. WHERE Author="Sonoo";
```

3. Data Control Language

DCL commands are used to grant and take back authority from any database user.

Here are some commands that come under DCL:

- Grant
- Revoke

a. Grant: It is used to give user access privileges to a database.

Example

```
1. GRANT SELECT, UPDATE ON MY_TABLE TO SOME_USER, ANOTHER_USER;
```

b. Revoke: It is used to take back permissions from the user.

Example

```
1. REVOKE SELECT, UPDATE ON MY_TABLE FROM USER1, USER2;
```

4. Transaction Control Language

TCL commands can only use with DML commands like INSERT, DELETE and UPDATE only.

These operations are automatically committed in the database that's why they cannot be used while creating tables or dropping them.

Here are some commands that come under TCL:

- COMMIT
- ROLLBACK
- SAVEPOINT

a. Commit: Commit command is used to save all the transactions to the database.

Syntax:

```
1. COMMIT;
```

Example:

```
1. DELETE FROM CUSTOMERS
2. WHERE AGE = 25;
3. COMMIT;
```

b. Rollback: Rollback command is used to undo transactions that have not already been saved to the database.

Syntax:

```
1. ROLLBACK;
```

Example:

```
1. DELETE FROM CUSTOMERS
2.     WHERE AGE = 25;
3.     ROLLBACK;
```

c. SAVEPOINT: It is used to roll the transaction back to a certain point without rolling back the entire transaction.

Syntax:

```
1.     SAVEPOINT SAVEPOINT_NAME;
```

5. Data Query Language

DQL is used to fetch the data from the database.

It uses only one command:

- SELECT

a. SELECT: This is the same as the projection operation of relational algebra. It is used to select the attribute based on the condition described by WHERE clause.

Syntax:

```
SELECT expressions
FROM TABLES
 WHERE conditions;
```

For example:

```
SELECT emp_name
FROM employee
WHERE age > 20;
```

UNION, INTERSECT, and EXCEPT:

1. UNION, UNION ALL
2. INTERSECT
3. EXCEPT

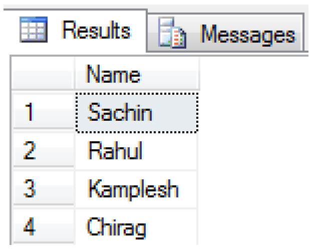
Create Table

```sql
CREATE TABLE Speakers
(
  Name VARCHAR(25),
)

CREATE TABLE Authors
(
  Name VARCHAR(25),
)
```

Insert data into the table

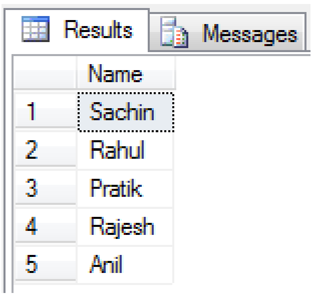
```sql
INSERT INTO Speakers VALUES ('Sachin')
INSERT INTO Speakers VALUES ('Rahul')
INSERT INTO Speakers VALUES ('Kamplesh')
INSERT INTO Speakers VALUES ('Chirag')

INSERT INTO Authors VALUES ('Sachin')
INSERT INTO Authors VALUES ('Rahul')
INSERT INTO Authors VALUES ('Pratik')
INSERT INTO Authors VALUES ('Rajesh')
      INSERT INTO Authors VALUES ('Anil')
```

Speakers



	Name
1	Sachin
2	Rahul
3	Kamplesh
4	Chirag

Authors



	Name
1	Sachin
2	Rahul
3	Pratik
4	Rajesh
5	Anil

UNION

Union is used to combine the results of two queries into a single result set of all matching rows. Both the queries must result in the same number of columns and compatible data types in order to unite. All duplicate records are removed automatically unless UNION ALL is used.

Generally, it can be useful in the applications where tables aren't perfectly normalized, for example, a data warehouse application.

Syntax

{ <query_specification> | (<query_expression>) }
{ UNION | UNION ALL}

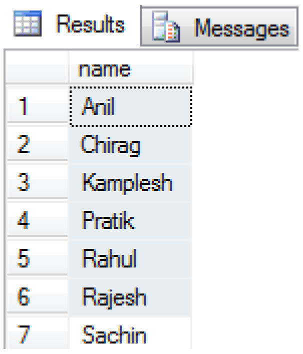
{ <query_specification> | (<query_expression>) }

Example-1

You want to invite all the Speakers and Authors for the annual conference. Hence, how will you prepare the invitation list?

```
select name from Speakers
union
select name from Authors
order by name
```

Output



	name
1	Anil
2	Chirag
3	Kamplesh
4	Pratik
5	Rahul
6	Rajesh
7	Sachin

As you can see here, the default order is ascending order and you have to use in the last query instead of both queries.

UNION ALL

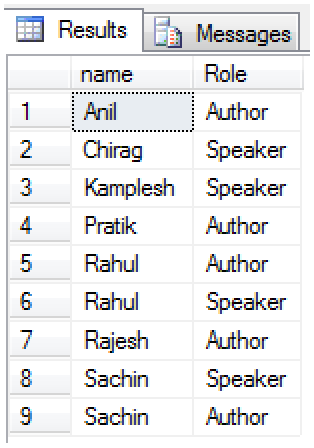
It will not remove duplicate records. It can be faster than UNION.

Example-2

You want to give a prize to all the Speakers and Authors at the annual conference. Hence, how will you prepare the prize list?

```
select name, 'Speaker' as 'Role' from Speakers
union all
select name, 'Author' as 'Role' from Authors
order by name
```

Output



	name	Role
1	Anil	Author
2	Chirag	Speaker
3	Kamplesh	Speaker
4	Pratik	Author
5	Rahul	Author
6	Rahul	Speaker
7	Rajesh	Author
8	Sachin	Speaker
9	Sachin	Author

INTERSECT

It is used to take the result of two queries and returns the only those rows which are common in both result sets. It removes duplicate records from the final result set.

Syntax

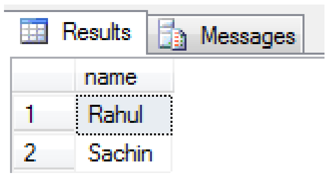
{ <query_specification> | (<query_expression>) }
{ EXCEPT | INTERSECT }
{ <query_specification> | (<query_expression>) }

Example-3

You want the list of people who are Speakers and they are also Authors. Hence, how will you prepare such a list?

```
select name from Speakers
intersect
select name from Authors
order by name
```

Output



```
   name
1  Rahul
2  Sachin
```

EXCEPT

It is used to take the distinct records of two one query and returns the only those rows which do not appear in the second result set.

Syntax

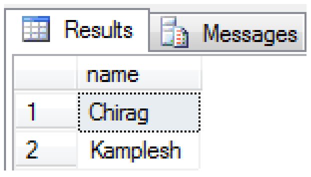
{ <query_specification> | (<query_expression>) }
{ EXCEPT | INTERSECT }
{ <query_specification> | (<query_expression>) }

Example-4

You want the list of people who are only Speakers and they are not Authors. Hence, how will you prepare such a list?

```
select name from Speakers
except
select name from Authors
order by name
```

Output



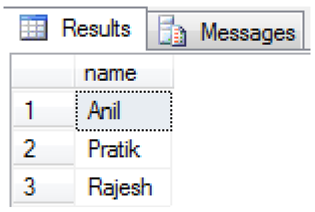
```
   name
1  Chirag
2  Kamplesh
```

Example-5

You want the list of people who are only Authors and they are not Speakers. Hence, how will you prepare such a list?

```
select name from Authors
except
select name from Speakers
order by name
```

Output

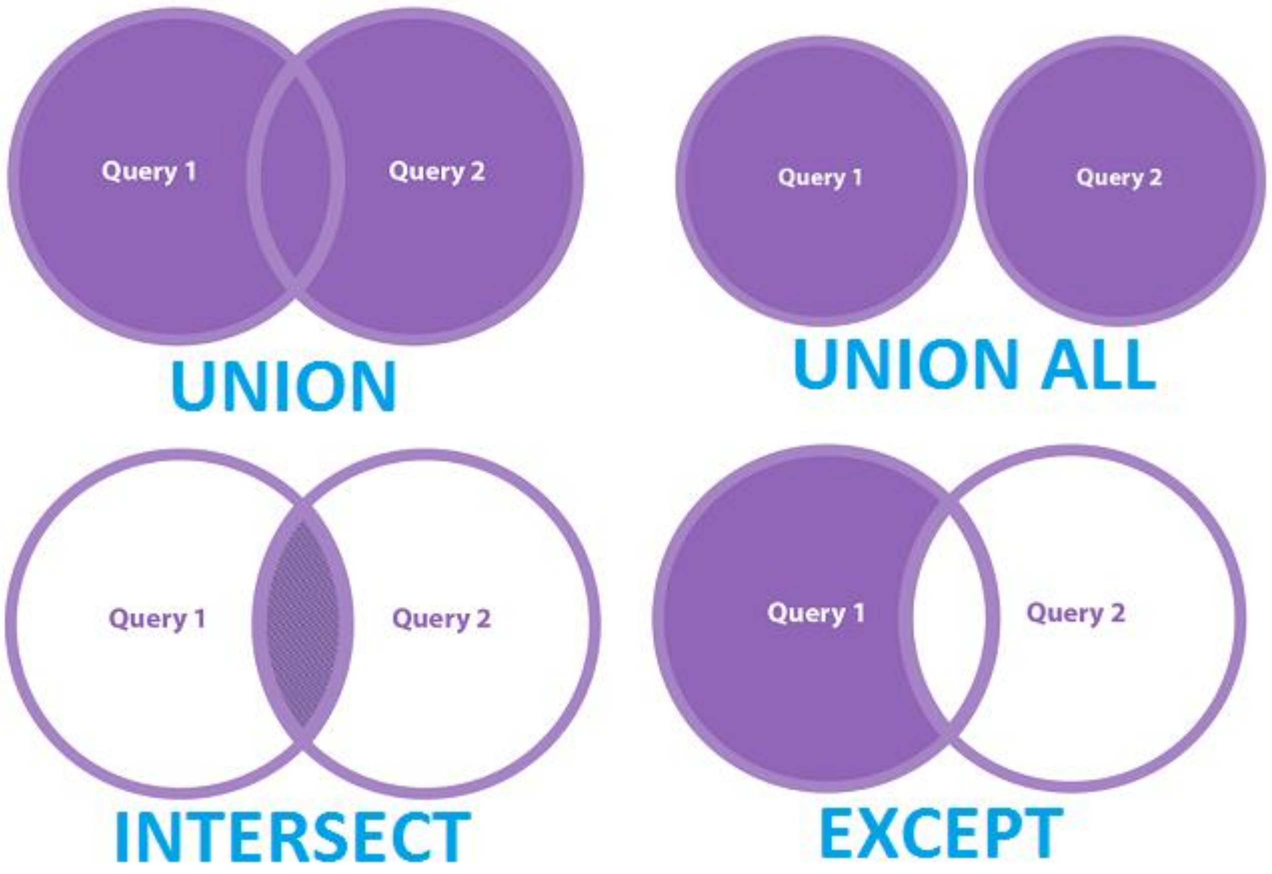


	name
1	Anil
2	Pratik
3	Rajesh

Basic Rules on Set Operations

- Result sets of all the queries must be the same number of columns.
- In all result sets the data type of each of the columns must be well matched and compatible with the data type of its corresponding columns in another result set.
- For sorting the result, the ORDER BY clause can be applied to the last query.

Difference between UNION, UNION ALL, INTERSECT, and EXCEPT Operators



Summary

Now I believe you understand the key important things about UNION, UNION ALL, INTERSECT, EXCEPT in MS SQL.

- UNION combines results from both tables.
- UNION ALL combines two or more result sets into a single set, including all duplicate rows.
- INTERSECT takes the rows from both the result sets which are common in both.
- EXCEPT takes the rows from the first result data but does not in the second result set.

EQUI Join and NON EQUI Join:

Example –

Let's Consider the two tables given below.

Table name — Student

In this table, you have I'd, name, class and city are the fields.

```
Select * from Student;
```

id	name	class	city
3	Hina	3	Delhi
4	Megha	2	Delhi
6	Gouri	2	Delhi

Table name — Record

In this table, you have I’d, class and city are the fields.

```
Select * from Record;
```

Id	class	city
9	3	Delhi
10	2	Delhi
12	2	Delhi

1. EQUI JOIN :

EQUI JOIN creates a JOIN for equality or matching column(s) values of the relative tables. EQUI JOIN also create JOIN by using JOIN with ON and then providing the names of the columns with their relative tables to check equality using equal sign (=).

Syntax :

```
SELECT column_list

FROM table1, table2....

WHERE table1.column_name =

table2.column_name;
```

Example –

```
SELECT student.name, student.id, record.class, record.city

FROM student, record

WHERE student.city = record.city;
```

Or

Syntax :

```
SELECT column_list

FROM table1

JOIN table2

[ON (join_condition)]
```

Example –

```
SELECT student.name, student.id, record.class, record.city

FROM student

JOIN record

ON student.city = record.city;
```

Output :

Name	Id	class	City
Hina	3	3	Delhi
Megha	4	3	Delhi

Name	Id	class	City
Gouri	6	3	Delhi
Hina	3	2	Delhi
Megha	4	2	Delhi
Gouri	6	2	Delhi
Hina	3	2	Delhi
Megha	4	2	Delhi
Gouri	6	2	Delhi

2. NON EQUI JOIN :
NON EQUI JOIN performs a JOIN using comparison operator other than equal(=) sign like >, <, >=, <= with conditions.

Syntax:

```
SELECT *
FROM table_name1, table_name2
WHERE table_name1.column [> |  < |  >= | <= ] table_name2.column;
```

Example –

```
SELECT student.name, record.id, record.city
FROM student, record
 WHERE Student.id < Record.id ;
```

Output :

name	id	City
Hina	9	Delhi
Megha	9	Delhi
Gouri	9	Delhi
Hina	10	Delhi
Megha	10	Delhi
Gouri	10	Delhi

name	id	City
Hina	12	Delhi
Megha	12	Delhi
Gouri	12	Delhi

NESTED QUERIES:

A nested query is a query that has another query embedded within it. The embedded query is called a subquery.

A subquery typically appears within the WHERE clause of a query. It can sometimes appear in the FROM clause or HAVING clause.

Example

Let’s learn about nested queries with the help of an example.

Find the names of employee who have regno=103

The query is as follows −

```
select E.ename from employee E where E.eid IN (select S.eid from salary S
where S.regno=103);
```

Student table

The student table is created as follows −

```
create table student(id number(10), name varchar2(20),classID number(10),
marks varchar2(20));
Insert into student values(1,'pinky',3,2.4);
Insert into student values(2,'bob',3,1.44);
Insert into student values(3,'Jam',1,3.24);
Insert into student values(4,'lucky',2,2.67);
Insert into student values(5,'ram',2,4.56);
select * from student;
```

Output

You will get the following output −

Id	Name	classID	Marks
1	Pinky	3	2.4
2	Bob	3	1.44

Id	Name	classID	Marks
3	Jam	1	3.24
4	Lucky	2	2.67
5	Ram	2	4.56

teacher table

The teacher table is created as follows −

Example

```
Create table teacher(id number(10), name varchar(20), subject
varchar2(10), classID number(10), salary number(30));
Insert into teacher values(1,’bhanu’,’computer’,3,5000);
Insert into teacher values(2,'rekha','science',1,5000);
Insert into teacher values(3,'siri','social',NULL,4500);
Insert into teacher values(4,'kittu','mathsr',2,5500);
select * from teacher;
```

Output

You will get the following output −

Id	Name	Subject	classID	Salary
1	Bhanu	Computer	3	5000
2	Rekha	Science	1	5000
3	Siri	Social	NULL	4500
4	Kittu	Maths	2	5500

Class table

The class table is created as follows −

Example

```
Create table class(id number(10), grade number(10), teacherID number(10),
noofstudents number(10));
insert into class values(1,8,2,20);
insert into class values(2,9,3,40);
insert into class values(3,10,1,38);
select * from class;
```

Output

You will get the following output −

Id	Grade	teacherID	No.ofstudents
1	8	2	20
2	9	3	40
3	10	1	38

Now let’s work on nested queries

Example 1

```
Select AVG(noofstudents) from class where teacherID IN(
Select id from teacher
Where subject='science' OR subject='maths');
```

Output

You will get the following output −

```
20.0
```

Example 2

```
SELECT * FROM student
WHERE classID = (
   SELECT id
   FROM class
   WHERE noofstudents = (
      SELECT MAX(noofstudents)
      FROM class));
```

Output

You will get the following output −

```
4|lucky |2|2.67
5|ram   |2|4.56
```

Subqueries can be categorized as *correlated* or *uncorrelated*:

- A correlated subquery refers to one or more columns from outside of the subquery. (The columns are typically referenced inside the WHERE clause of the subquery.) A correlated subquery can be thought of as a filter on the table that it refers to, as if the subquery were evaluated on each row of the table in the outer query.
- An uncorrelated subquery has no such external column references. It is an independent query, the results of which are returned to and used by the outer query once (not per row).

For example:

```
-- Uncorrelated subquery:
select c1, c2
  from table1 where c1 = (select max(x) from table2);

-- Correlated subquery:
select c1, c2
  from table1 where c1 = (select x from table2 where y = table1.c2);
```

Scalar vs. Non-scalar Subqueries

Subqueries can also be categorized as *scalar* or *non-scalar*:

- A scalar subquery returns a single value (one column of one row). If no rows qualify to be returned, the subquery returns NULL.
- A non-scalar subquery returns 0, 1, or multiple rows, each of which may contain 1 or multiple columns. For each column, if there is no value to return, the subquery returns NULL. If no rows qualify to be returned, the subquery returns 0 rows (not NULLs).

Differences Between Correlated and Non-Correlated Subqueries

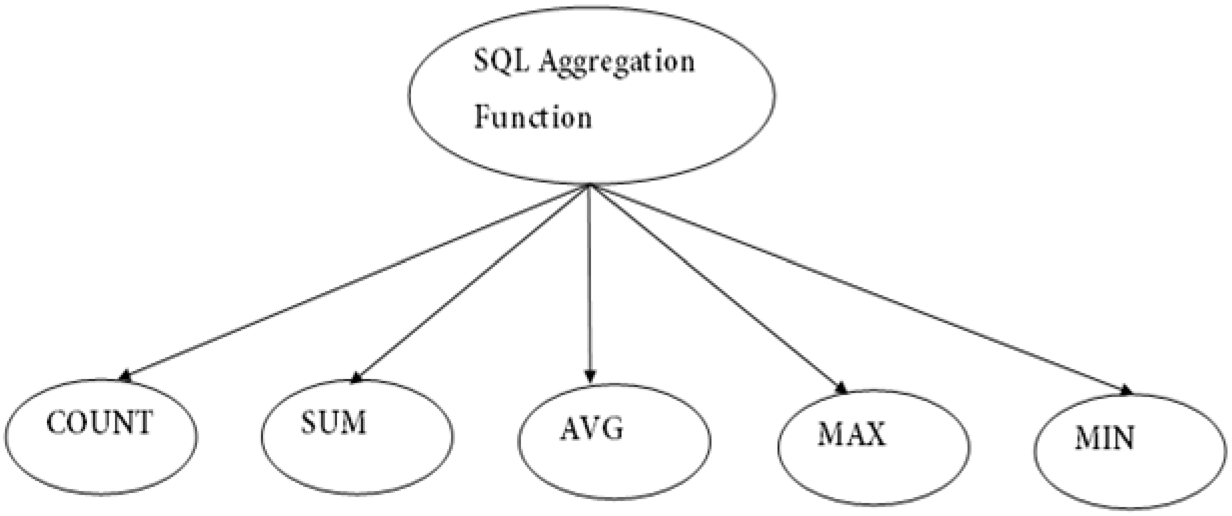
The following query demonstrates an uncorrelated subquery in a WHERE clause. The subquery gets the per capita GDP of Brazil, and the outer query selects all the jobs (in any country) that pay less than the per-capita GDP of Brazil. The subquery is uncorrelated because the value that it returns does not depend upon any column of the outer query. The subquery only needs to be called once during the entire execution of the outer query.

```
select p.name, p.annual_wage, p.country
  from pay as p
  where p.annual_wage < (select per_capita_gdp
                         from international_gdp
                         where name = 'Brazil');
```

The next query demonstrates a correlated subquery in a WHERE clause. The query lists jobs where the annual pay of the job is less than the per-capita GDP in that country. This subquery is correlated because it is called once for each row in the outer query and is passed a value, `p.country` (country name), from the row.

```
select p.name, p.annual_wage, p.country
  from pay as p
  where p.annual_wage < (select max(per_capita_gdp)
                         from international_gdp i
                         where p.country = i.name);
```

Aggregate Functions:



1. COUNT FUNCTION

- COUNT function is used to Count the number of rows in a database table. It can work on both numeric and non-numeric data types.
- COUNT function uses the COUNT(*) that returns the count of all the rows in a specified table. COUNT(*) considers duplicate and Null.

Syntax

COUNT(*)
or
COUNT([ALL|DISTINCT] expression)

Sample table:

PRODUCT_MAST

PRODUCT	COMPANY	QTY	RATE	COST

Item1	Com1	2	10	20
Item2	Com2	3	25	75
Item3	Com1	2	30	60
Item4	Com3	5	10	50
Item5	Com2	2	20	40
Item6	Cpm1	3	25	75
Item7	Com1	5	30	150
Item8	Com1	3	10	30
Item9	Com2	2	25	50
Item10	Com3	4	30	120

Example: COUNT()

```
SELECT COUNT(*)
FROM PRODUCT_MAST;
```

Output:

```
10
```

Example: COUNT with WHERE

```
SELECT COUNT(*)
FROM PRODUCT_MAST;
WHERE RATE>=20;
```

Output:

```
7
```

Example: COUNT() with DISTINCT

```
SELECT COUNT(DISTINCT COMPANY)
FROM PRODUCT_MAST;
```

Output:

```
3
```

Example: COUNT() with GROUP BY

```
SELECT COMPANY, COUNT(*)
```

FROM PRODUCT_MAST
GROUP BY COMPANY;

Output:

```
Com1     5
Com2     3
Com3     2
```

Example: COUNT() with HAVING

SELECT COMPANY, COUNT(*)
FROM PRODUCT_MAST
GROUP BY COMPANY
HAVING COUNT(*)>2;

Output:

```
Com1     5
Com2     3
```

2. SUM Function

Sum function is used to calculate the sum of all selected columns. It works on numeric fields only.

Syntax

SUM()
or
SUM([ALL|DISTINCT] expression)

Example: SUM()

SELECT SUM(COST)
FROM PRODUCT_MAST;

Output:

```
670
```

Example: SUM() with WHERE

SELECT SUM(COST)
FROM PRODUCT_MAST
WHERE QTY>3;

Output:

```
320
```

Example: SUM() with GROUP BY

SELECT SUM(COST)

FROM PRODUCT_MAST

WHERE QTY>3

1. GROUP BY COMPANY;

Output:

```
Com1     150
Com2     170
```

Example: SUM() with HAVING

SELECT COMPANY, SUM(COST)

FROM PRODUCT_MAST

GROUP BY COMPANY

HAVING SUM(COST)>=170;

Output:

```
Com1     335
Com3     170
```

3. AVG function

The AVG function is used to calculate the average value of the numeric type. AVG function returns the average of all non-Null values.

Syntax

AVG()

or

AVG([ALL|DISTINCT] expression)

Example:

SELECT AVG(COST)

FROM PRODUCT_MAST;

Output:

```
67.00
```

4. MAX Function

MAX function is used to find the maximum value of a certain column. This function determines the largest value of all selected values of a column.

Syntax

MAX()

or

MAX([ALL|DISTINCT] expression)

Example:

SELECT MAX(RATE)

FROM PRODUCT_MAST;

```
30
```

5. MIN Function

MIN function is used to find the minimum value of a certain column. This function determines the smallest value of all selected values of a column.

Syntax

MIN()

or

MIN([ALL|DISTINCT] expression)

Example:

SELECT MIN(RATE)

FROM PRODUCT_MAST;

Output:

```
10
```

VIEWS:

In SQL, a view is **a virtual table based on the result-set of an SQL statement**. A view contains rows and columns, just like a real table. The fields in a view are fields from one or more real tables in the database.

SQL CREATE VIEW Statement

In SQL, a view is a virtual table based on the result-set of an SQL statement.

A view contains rows and columns, just like a real table. The fields in a view are fields from one or more real tables in the database.

You can add SQL statements and functions to a view and present the data as if the data were coming from one single table.

A view is created with the `CREATE VIEW` statement.

CREATE VIEW Syntax

```
CREATE VIEW view_name AS
SELECT column1, column2, ...
```

```
FROM table_name
WHERE condition;
```

Note: A view always shows up-to-date data! The database engine recreates the view, every time a user queries it.

SQL CREATE VIEW Examples

The following SQL creates a view that shows all customers from Brazil:

Example

```
CREATE VIEW [Brazil Customers] AS
SELECT CustomerName, ContactName
FROM Customers
WHERE Country = 'Brazil';
```

SQL Updating a View

A view can be updated with the `CREATE OR REPLACE VIEW` statement.

SQL CREATE OR REPLACE VIEW Syntax

```
CREATE OR REPLACE VIEW view_name AS
SELECT column1, column2, ...
FROM table_name
WHERE condition;
```

The following SQL adds the "City" column to the "Brazil Customers" view:

Example

```
CREATE OR REPLACE VIEW [Brazil Customers] AS
SELECT CustomerName, ContactName, City
FROM Customers
WHERE Country = 'Brazil';
```

SQL Dropping a View

A view is deleted with the `DROP VIEW` statement.

SQL DROP VIEW Syntax

```
DROP VIEW view_name;
```

The following SQL drops the "Brazil Customers" view:

Example

```
DROP VIEW [Brazil Customers];
```

Triggers:

Trigger: A trigger is **a stored procedure in database which automatically invokes whenever a special event in the database occurs**. For example, a trigger can be invoked when a row is inserted into a specified table or when certain table columns are being updated.

Syntax:

```
create trigger [trigger_name]

[before | after]

{insert | update | delete}

on [table_name]

[for each row]

[trigger_body]
```

Explanation of syntax:

1. create trigger [trigger_name]: Creates or replaces an existing trigger with the trigger_name.
2. [before | after]: This specifies when the trigger will be executed.
3. {insert | update | delete}: This specifies the DML operation.
4. on [table_name]: This specifies the name of the table associated with the trigger.
5. [for each row]: This specifies a row-level trigger, i.e., the trigger will be executed for each row being affected.
6. [trigger_body]: This provides the operation to be performed as trigger is fired

BEFORE and AFTER of Trigger:

BEFORE triggers run the trigger action before the triggering statement is run.
AFTER triggers run the trigger action after the triggering statement is run.

Example:

Given Student Report Database, in which student marks assessment is recorded. In such schema, create a trigger so that the total and average of specified marks is automatically inserted whenever a record is insert.
Here, as trigger will invoke before record is inserted so, BEFORE Tag can be used.

Suppose the database Schema –

```
mysql> desc Student;

+-------+-------------+------+-----+---------+----------------+

| Field | Type        | Null | Key | Default | Extra          |

+-------+-------------+------+-----+---------+----------------+

| tid   | int(4)      | NO   | PRI | NULL    | auto_increment |

| name  | varchar(30) | YES  |     | NULL    |                |

| subj1 | int(2)      | YES  |     | NULL    |                |

| subj2 | int(2)      | YES  |     | NULL    |                |

| subj3 | int(2)      | YES  |     | NULL    |                |
| total | int(3)      | YES  |     | NULL    |                |
| per   | int(3)      | YES  |     | NULL    |                |
+-------+-------------+------+-----+---------+----------------+
7 rows in set (0.00 sec)
```

SQL Trigger to problem statement.

```
create trigger stud_marks
before INSERT
on
Student
for each row
set Student.total = Student.subj1 + Student.subj2 + Student.subj3,
Student.per = Student.total * 60 / 100;
```

Above SQL statement will create a trigger in the student database in which whenever subjects marks are entered, before inserting this data into the database, trigger will compute those two values and insert with the entered values. i.e.,

```
mysql> insert into Student values(0, "ABCDE", 20, 20, 20, 0, 0);
Query OK, 1 row affected (0.09 sec)

mysql> select * from Student;
+-----+-------+-------+-------+-------+-------+------+
| tid | name  | subj1 | subj2 | subj3 | total | per  |
+-----+-------+-------+-------+-------+-------+------+
| 100 | ABCDE |    20 |    20 |    20 |    60 |   36 |
+-----+-------+-------+-------+-------+-------+------+
1 row in set (0.00 sec)
```